



PPC401A1

**Embedded Controller Core
User's Manual**

13H6946 00001

Fifth Preliminary Edition (October 1997)

This edition of *IBM PPC401A1 Embedded Controller Core User's Manual* applies to the IBM PPC401A1 32-bit embedded controller core, until otherwise indicated in new versions or technical newsletters.

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS MANUAL "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

IBM does not warrant that the products in this publication, whether individually or as one or more groups, will meet your requirements or that the publication or the accompanying product descriptions are error-free.

This publication could contain technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or program(s) described in this publication at any time.

It is possible that this publication may contain references to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Any reference to an IBM licensed program in this publication is not intended to state or imply that you can use only IBM's licensed program. You can use any functionally equivalent program instead.

No part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the written permission of IBM.

Requests for copies of this publication and for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

Address comments about this publication to:

IBM Corporation
Department YM5A
P.O. Box 12195
Research Triangle Park, NC 27709

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1996, 1997. All rights reserved

Printed in the United States of America.

4 3 2 1

Notice to U.S. Government Users—Documentation Related to Restricted Rights—Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corporation.

Patents and Trademarks

IBM may have patents or pending patent applications covering the subject matter in this publication. The furnishing of this publication does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 208 Harbor Drive, Stamford, CT 06904, United States of America.

The following terms are trademarks of IBM Corporation:

IBM

PowerPC

PowerPC Architecture

PowerPC Embedded Controllers

RISCWatch

Other terms which are trademarks are the property of their respective owners.

Contents

Figures	xv
Tables	xix
About this Book	xxiii
1. Overview	1-1
PPC401A1 Features	1-1
PowerPC Architecture	1-2
The PPC401A1 as a PowerPC Implementation	1-3
PPC401A1 Organization	1-3
RISC Processor Core	1-4
Instruction and Data Cache Controllers	1-5
Instruction Cache Unit (ICU)	1-5
Data Cache Unit (DCU)	1-6
Timers	1-6
Storage Control	1-7
Debug	1-7
Development Tool Support	1-7
Debug Modes	1-8
Core Interfaces	1-8
PLB (Processor Local Bus)	1-9
DCR (Device Control Register)	1-9
Clock and Power Management	1-9
JTAG	1-9
Interrupts	1-9
Data Types	1-9
Register Set Summary	1-10
General Purpose Registers	1-10
Special Purpose Registers	1-10
Machine State Register	1-10
Condition Register	1-10
Device Control Registers	1-11
Addressing Modes	1-11
2. Programming Model	2-1
Memory Organization and Addressing	2-1
Storage Attributes	2-1
Registers	2-2
General Purpose Registers	2-2
Special Purpose Registers	2-3
Count Register (CTR)	2-4
Link Register (LR)	2-5
Fixed Point Exception Register (XER)	2-6
Special Purpose Register General (SPRG0-SPRG3)	2-8
Processor Version Register (PVR)	2-8
Condition Register (CR)	2-9
CR Fields after Compare Instructions	2-10

The CR0 Field	2-11
The Time Base	2-12
Machine State Register	2-13
Device Control Registers	2-14
Data Types and Alignment	2-15
Alignment for Storage Reference and Cache Control Instructions	2-15
Alignment and Endian Operation	2-16
Summary of Instructions Causing Alignment Exceptions	2-16
Byte Ordering	2-17
Structure Mapping Examples	2-18
Big-Endian Mapping	2-18
Little Endian Mapping	2-19
PowerPC Byte Ordering	2-19
PowerPC Endian Mode	2-19
Byte Ordering in PowerPC Little Endian Mode	2-20
Control of PowerPC Endian Mode	2-22
Addressing in PowerPC Little Endian Mode	2-23
Little Endian Mode Alignment Requirements	2-23
Switching Endian Modes	2-24
Direct Memory Access in PowerPC Little Endian Mode	2-24
Endian Storage Attribute	2-25
Fetching Instructions from Little Endian Storage Regions	2-25
Accessing Data in Little Endian Storage Regions	2-26
Control of the Endian Storage Attribute	2-27
PowerPC Byte-Reverse Instructions	2-27
Instruction Processing	2-30
Branching Control	2-30
AA Field on Unconditional Branches	2-31
AA Field on Conditional Branches	2-31
BI Field on Conditional Branches	2-31
BO Field on Conditional Branches	2-31
Branch Prediction	2-33
Speculative Accesses	2-34
Speculative Accesses in the PPC401A1	2-34
Pre-fetch Distance Down an Unresolved Branch Path	2-34
Pre-fetch of Branches to Count Register and Branches to Link Register	2-35
Preventing Inappropriate Speculative Accesses	2-35
Fetching Past an Interrupt-causing or Interrupt-returning Instruction	2-36
Fetching Past tw or twi Instructions	2-36
Fetching Past an Unconditional Branch	2-37
Suggested Locations of Memory-Mapped Hardware	2-37
Summary	2-37
Privileged Mode Operation	2-38
MSR Bits and Exception Handling	2-38
Privileged Instructions	2-39
Privileged SPRs	2-39
Privileged DCRs	2-40

Synchronization	2-40
Context Synchronization	2-40
Execution Synchronization	2-43
Storage Synchronization	2-44
Instruction Set	2-44
Instructions Specific to IBM PowerPC Embedded Controllers	2-45
Storage Reference Instructions	2-46
Arithmetic and Logical Instructions	2-46
Compare Instructions	2-47
Branch Instructions	2-47
Condition Register Logical Instructions	2-48
Rotate and Shift Instructions	2-48
Cache Control Instructions	2-49
Interrupt Control Instructions	2-49
Processor Management Instructions	2-50
Extended Mnemonics	2-50
3. I/O Interfaces	3-1
Signal Naming Conventions	3-2
Signal Name Cross-Reference	3-3
Timing Definitions	3-4
Clock and Power Management (CPM) Interface	3-5
CPM Interface I/O Symbol	3-6
CPM Interface I/O Signal Table	3-7
CPM Interface I/O Signal Descriptions	3-7
CPU Control Interface	3-10
CPU Control Interface I/O Symbol	3-10
CPU Control Interface I/O Signal Table	3-10
CPU Control Interface I/O Signal Descriptions	3-11
Test Mode Matrix (TMM)/LSSD Test Interface	3-12
TMM/Test Interface I/O Symbol	3-14
TMM/Test Interface I/O Signal Table	3-14
TMM/Test Interface I/O Signal Descriptions	3-15
Reset Interface	3-20
Reset Requirements	3-20
Reset Connectivity Example	3-21
Reset Interface I/O Symbol	3-23
Reset Interface I/O Signal Table	3-23
Reset Interface I/O Signal Descriptions	3-24
Instruction-side Bus Interface Unit (BIU)Interface	3-27
Instruction-Side BIU Interface I/O Symbol	3-28
Instruction-Side BIU I/O Signal Table	3-28
Instruction-Side BIU Interface I/O Signal Descriptions	3-29
Instruction-Side BIU Interface Timing Diagrams	3-35
ICU/BIU Timing Diagram Guidelines	3-35
ICU/BIU Back-to-Back Cacheable Fetches (Custom BIU)	3-38
ICU/BIU Fetch Cacheable-NonCacheables-Cacheable Mix (Custom BIU)	3-40
ICU/BIU-PLB Fetch NonCacheables to Cacheable (PLB-Compliant)	3-42

Data-Side Bus Interface Unit (BIU) Interface	3-44
Data-Side BIU Interface I/O Symbol	3-45
Data-Side BIU Interface I/O Signal Table	3-46
Data-Side BIU Interface I/O Signal Descriptions	3-47
Data-Side BIU Interface Timing Diagrams	3-58
DCU/BIU Timing Diagram Guidelines	3-58
DCU/BIU Non-Cacheable Mix (Custom BIU)	3-62
DCU/BIU-PLB Aborted Read Requests	3-64
DCU/BIU Store Write-through Miss w/ Fill-Flush, then Store WT Hits (Custom BIU)	3-66
DCU/BIU Non-Cacheable Write, Cacheable Read Mix	3-68
DCU/BIU Non-cacheable, Cacheable Read Mix (Custom BIU)	3-70
DCU/BIU-PLB Non-cacheable Read, Cacheable Write Mix	3-72
DCU/BIU Back to back Cacheable Reads (Custom BIU)	3-74
DCU/BIU Back toBack Cacheable Reads with Cache Hits Included (Custom BIU)	3-76
DCU/BIU-PLB Cacheable Miss to Dirty Line (Fill-Flush-Fill)	3-78
DCU/BIU-PLB Cacheable Write, Read Mix (Custom BIU)	3-80
DCU/BIU-PLB Cacheable, Non-cacheable Write Mix	3-82
Device Control Register (DCR) Interface	3-84
DCR Chain Implementation	3-86
DCR Interface I/O Symbol	3-87
DCR Interface I/O Signal Table	3-88
DCR Interface I/O Signal Descriptions	3-88
DCR Interface I/O Timing Diagrams	3-90
Core and ASIC at Same Frequency, with Combinatorial DCR_cpuAck	3-92
Core and ASIC at Same Frequency, with Latched DCU_cpuAck	3-94
Clock-doubled Core, Latched DCR_cpuAck	3-96
Clock-doubled ASIC, Latched DCU_cpuAck	3-98
External Interrupt Controller (EIC) Interface	3-99
EIC Interface I/O Signal Table	3-100
EIC Interface I/O Signal Descriptions	3-100
JTAG Interface	3-102
JTAG Interface I/O Symbol	3-103
JTAG Interface I/O Signal Table	3-103
JTAG Interface I/O Signal Descriptions	3-104
Debug (DBG) Interface	3-108
DBG Interface I/O Symbol	3-108
DBG Interface I/O Signal Table	3-108
DBG Interface I/O Signal Descriptions	3-108
4. Initialization	4-1
Processor State After Reset	4-1
PPC401A1 Initial Processor Sequencing	4-2
Initialization Requirements	4-3
Initialization Code Example	4-4
5. Exceptions, Interrupts, and Timers	5-1
Interrupts and Exceptions	5-1

Architectural Definitions and Behavior	5-2
Behavior of the PPC401A1 Implementation	5-3
Exception-handling Priorities	5-4
Critical and Non-critical Exceptions	5-5
General Exception Handling Registers	5-7
Machine State Register (MSR)	5-7
Save/Restore Registers 0 and 1 (SRR0–SRR1)	5-8
Save/Restore Registers 2 and 3 (SRR2– SRR3)	5-9
Exception Vector Prefix Register (EVPR)	5-10
Exception Syndrome Register (ESR)	5-10
Data Exception Address Register (DEAR)	5-12
Critical Interrupt Exception	5-13
Machine Check Exceptions	5-13
Instruction Machine Check Handling	5-14
Data Machine Check Handling	5-15
Data Storage Exceptions	5-15
External Interrupt Exception	5-16
External Interrupt Exception Handling	5-17
Alignment Exception	5-17
Program Exceptions	5-18
System Call Exception	5-20
Programmable Interval Timer (PIT) Exception	5-20
Fixed Interval Timer (FIT) Exception	5-21
Watchdog Timer Exception	5-22
Debug Exception Handling	5-23
Timer Facilities	5-24
Time Base	5-25
Comparison with PowerPC Architecture Time Base	5-26
Programmable Interval Timer (PIT)	5-28
Fixed Interval Timer (FIT)	5-29
Watchdog Timer	5-30
Timer Status Register (TSR)	5-32
Timer Control Register (TCR)	5-33
6. Cache Operations	6-1
ICU Organization	6-1
ICU Operations	6-3
Instruction Cacheability Control	6-4
ICU Coherency	6-4
DCU Organization	6-4
DCU Operations	6-5
DCU Write Strategies	6-6
Data Cacheability Control	6-6
DCU Coherency	6-7
Cache Instructions	6-7
ICU Instructions	6-7
DCU Instructions	6-8
Cache Control and Debugging Features	6-9

ICU Debugging	6-10
DCU Debugging	6-11
Cache Line Locking	6-12
Locking Lines in the ICU and DCU Cache Arrays	6-12
Unlocking Lines in the ICU and DCU	6-13
DCU Performance	6-14
Pipeline Stalls	6-14
Cache Operation Priorities	6-15
Simultaneous Cache Operations	6-16
Sequential Cache Operations	6-16
7. Debugging and JTAG Facilities	7-1
Development Tool Support	7-1
Debug Modes	7-1
Internal Debug Mode	7-1
External Debug Mode	7-2
Processor Control	7-2
Processor Status	7-3
Debug Events	7-4
Debug Registers	7-5
Debug Control Register (DBCR)	7-5
Note on the DAC Compare Size Field (DBCR[D1S])	7-6
Debug Status Register (DBSR)	7-7
Data Address Compare Register (DAC)	7-8
Data Address Compare (DAC) Applied to Cache Instructions	7-9
DAC Applied to String Instructions	7-10
Instruction Address Compare Register (IAC1)	7-10
Debug Interface	7-10
IEEE 1149.1 Test Access Port (JTAG Debug Port)	7-11
JTAG Connector	7-11
JTAG Instructions	7-13
JTAG Boundary Scan	7-13
8. Storage Control	8-1
Storage Attributes	8-1
Data Cache Write-through Register (DCWR)	8-2
Data Cache Cacheability Register (DCCR)	8-4
Instruction Cache Cacheability Register (ICCR)	8-6
Storage Guarded Register (SGR)	8-8
Storage Little-Endian Register (SLER)	8-10
9. Instruction Set	9-1
Instruction Set Portability	9-1
Instruction Formats	9-2
Pseudocode	9-2
Register Usage	9-5
Alphabetical Instruction Listing	9-5
add	9-6

addc	9-7
adde	9-8
addi	9-9
addic	9-10
addic.	9-11
addis	9-12
addme	9-13
addze	9-14
and	9-15
andc	9-16
andi.	9-17
andis.	9-18
b	9-19
bc	9-20
bcctr	9-28
bclr	9-32
cmp	9-37
cmpi	9-38
cmpl	9-39
cmpli	9-40
cntlzw	9-41
crand	9-42
crandc	9-43
creqv	9-44
crnand	9-45
crnor	9-46
cror	9-47
crorc	9-48
crxor	9-49
dcbf	9-50
dcbi	9-51
dcbst	9-52
dcbt	9-53
dcbtst	9-55
dcbz	9-57
dccci	9-59
dcread	9-61
divw	9-63
divwu	9-64
eieio	9-65
eqv	9-66
extsb	9-67
extsh	9-68
icbi	9-69
icbt	9-71
iccci	9-73
icread	9-74

isync	9-76
lbz	9-78
lbzu	9-79
lbzux	9-80
lbzx	9-81
lha	9-82
lhau	9-83
lhaux	9-84
lhax	9-85
lhbrx	9-86
lhz	9-87
lhzu	9-88
lhzux	9-89
lhzx	9-90
lmw	9-91
lswi	9-92
lswx	9-94
lwarx	9-96
lwbrx	9-98
lwz	9-99
lwzu	9-100
lwzux	9-101
lwzx	9-102
mcrf	9-103
mcrxr	9-104
mfcrr	9-105
mfdcr	9-106
mfmsr	9-107
mfmspr	9-108
mtcrf	9-110
mtdcr	9-112
mtmsr	9-113
mtspr	9-114
mulhw	9-116
mulhwu	9-117
mulli	9-118
mullw	9-119
nand	9-120
neg	9-121
nor	9-122
or	9-123
orc	9-124
ori	9-125
oris	9-126
rfci	9-127
rfi	9-128
rlwimi	9-129

rlwinm	9-130
rlwnm	9-133
sc	9-134
slw	9-135
sraw	9-136
srawi	9-137
srw	9-138
stb	9-139
stbu	9-140
stbux	9-141
stbx	9-142
sth	9-143
sthbrx	9-144
sthu	9-145
sthux	9-146
sthx	9-147
stmw	9-148
stswi	9-149
stswx	9-150
stw	9-152
stwbrx	9-153
stwcx.	9-154
stwu	9-156
stwux	9-157
stwx	9-158
subf	9-159
subfc	9-160
subfe	9-162
subfic	9-163
subfme	9-164
subfze	9-165
sync	9-166
tw	9-167
twi	9-171
wrttee	9-175
wrtteei	9-176
xor	9-177
xori	9-178
xoris	9-179
10. Register Summary	10-1
Reserved Registers	10-1
Reserved Fields	10-1
General Purpose Registers	10-2
Machine State Register and Condition Register	10-2
Special Purpose Registers	10-2
Device Control Registers	10-4
Alphabetical Register Listing	10-4

11. Signal Summary	11-1
Signal Naming Conventions	11-1
A. Instruction Summary	A-1
Instruction Set and Extended Mnemonics – Alphabetical	A-1
Instructions Sorted by Opcode	A-38
Instruction Formats	A-46
Instruction Fields	A-46
Instruction Format Diagrams	A-48
B. Instructions By Category	B-1
Instruction Set Summary Categories	B-1
Instructions Specific to PowerPC Embedded Controllers	B-1
Privileged Instructions	B-3
Assembler Extended Mnemonics	B-4
Storage Reference Instructions	B-26
Arithmetic and Logical Instructions	B-30
Condition Register Logical Instructions	B-35
Branch Instructions	B-36
Comparison Instructions	B-37
Rotate and Shift Instructions	B-38
Cache Control Instructions	B-39
Interrupt Control Instructions	B-40
Processor Management Instructions	B-41
C. Code Optimization and Instruction Timings	C-1
Code Optimization Guidelines	C-1
Condition Register Bits for Boolean Variables	C-1
CR Logical Instruction for Compound Branches	C-1
Floating-Point Emulation	C-2
Instruction Cache Usage	C-2
Data Cache Usage	C-2
CR Dependencies	C-3
LR and CTR Dependencies	C-3
Branch Prediction	C-3
Alignment	C-3
Instruction Timings	C-4
General Rules	C-4
Branches	C-4
String Instructions	C-5
Data Cache Loads and Stores	C-6
Instruction Cache Misses	C-6
Index	X-1

Figures

Figure 1-1.	PPC401A1 Block Diagram	1-4
Figure 2-1.	General Purpose Register (R0-R31)	2-3
Figure 2-2.	Count Register (CTR)	2-5
Figure 2-3.	Link Register (LR)	2-5
Figure 2-4.	Fixed Point Exception Register (XER)	2-6
Figure 2-5.	Special Purpose Register General (SPRG0-SPRG3)	2-8
Figure 2-6.	Processor Version Register (PVR)	2-9
Figure 2-7.	Condition Register (CR)	2-10
Figure 2-8.	Machine State Register (MSR)	2-13
Figure 2-9.	PPC401A1 Data Types	2-15
Figure 2-10.	Normal Word Load or Store (Big Endian Storage Region)	2-28
Figure 2-11.	Byte-reverse Word Load or Store (Little Endian Storage Region)	2-28
Figure 2-12.	Byte-reverse Word Load or Store (Big Endian Storage Region)	2-29
Figure 2-13.	Normal Word Load or Store (Little Endian Storage Region)	2-29
Figure 2-14.	PPC401A1 Instruction Queue	2-30
Figure 3-1.	CPM Interface I/O Symbol	3-6
Figure 3-2.	CPU Control Interface I/O Symbol	3-10
Figure 3-3.	TMM/Test Interface I/O Symbol	3-14
Figure 3-4.	Sample On-chip Reset Connection	3-22
Figure 3-5.	Reset Interface I/O Symbol	3-23
Figure 3-6.	Instruction-Side BIU Interface I/O Symbol	3-28
Figure 3-7.	ICU/BIU Back-to-Back Cacheable Fetches	3-39
Figure 3-8.	ICU/BIU Fetch Cacheable-NonCacheables-Cacheable	3-41
Figure 3-9.	ICU/BIU-PLB Fetch NonCacheables to Cacheable	3-43
Figure 3-10.	Data Side BIU Interface I/O Symbol	3-45
Figure 3-11.	DCU/BIU-PLB Non-Cacheable Mix (Custom BIU)	3-63
Figure 3-12.	DCU/BIU-PLB Aborted Read Requests	3-65
Figure 3-13.	DCU/BIU-PLB Store write-through Miss, then Store write-through Hits	3-67
Figure 3-14.	DCU/BIU-PLB Non-Cacheable Write, Cacheable Read Mix	3-69
Figure 3-15.	DCU/BIU-PLB Non-cacheable, Cacheable Read Mix (Custom BIU)	3-71
Figure 3-16.	DCU/BIU-PLB Non-cacheable Read, Cacheable Write Mix	3-73
Figure 3-17.	DCU/BIU-PLB Back to back Cacheable Reads (Custom BIU)	3-75
Figure 3-18.	DCU/BIU-PLB Back to back Cacheable Reads w/ Hits (Custom BIU)	3-77
Figure 3-19.	DCU/BIU-PLB Cacheable Mix	3-79
Figure 3-20.	DCU/BIU-PLB Cacheable Write, Read Mix (Custom BIU)	3-81
Figure 3-21.	DCU/BIU-PLB Cacheable, Non-Cacheable Write Mix	3-83
Figure 3-22.	DCR Block Diagram	3-85
Figure 3-23.	DCR Bus Implementation	3-86

Figure 3-24.	DCR Interface I/O Symbol	3-87
Figure 3-25.	Same clocks, Combinatorial Acknowledge	3-92
Figure 3-26.	Same clocks, Latched DCR_cpuAck	3-94
Figure 3-27.	Clock-doubled, Latched DCR_cpuAck	3-96
Figure 3-28.	Clock-doubled ASIC, Latched DCR_cpuAck	3-98
Figure 3-29.	EIC Interface I/O Symbol	3-100
Figure 3-30.	JTAG Interface I/O Symbol	3-103
Figure 3-31.	DBG Interface I/O Symbol	3-108
Figure 5-1.	Machine State Register (MSR)	5-7
Figure 5-2.	Save/Restore Register 0 (SRR0)	5-8
Figure 5-4.	Save/Restore Register 2 (SRR2)	5-9
Figure 5-3.	Save/Restore Register 1 (SRR1)	5-9
Figure 5-5.	Save/Restore Register 1 (SRR1)	5-10
Figure 5-6.	Exception Vector Prefix Register (EVPR)	5-10
Figure 5-7.	Exception Syndrome Register (ESR)	5-11
Figure 5-8.	Data Exception Address Register (DEAR)	5-12
Figure 5-9.	Relationship of Timer Facilities to the Base Clock	5-24
Figure 5-10.	Time Base Register (TBHI, TBHU)	5-25
Figure 5-11.	Time Base Register (TBLO, TBLU)	5-26
Figure 5-12.	Time Base Comparison	5-27
Figure 5-13.	Programmable Interval Timer (PIT)	5-29
Figure 5-14.	Watchdog Timer State Machine	5-31
Figure 5-15.	Timer Status Register (TSR)	5-32
Figure 5-16.	Timer Control Register (TCR)	5-33
Figure 6-1.	Instruction Cache Organization	6-2
Figure 6-2.	Instruction Flow	6-3
Figure 6-3.	Data Cache Organization	6-5
Figure 6-4.	Cache Debug Control Register (CDBCR)	6-9
Figure 6-5.	Instruction Cache Debug Data Register (ICDBDR)	6-10
Figure 7-1.	Debug Control Register (DBCR)	7-5
Figure 7-2.	Debug Status Register (DBSR)	7-7
Figure 7-3.	Data Address Compare Register (DAC)	7-8
Figure 7-4.	Instruction Address Compare Register (IAC1)	7-10
Figure 7-5.	JTAG Connector (top view) Physical Layout	7-12
Figure 8-1.	Data Cache Write-through Register (DCWR)	8-2
Figure 8-2.	Data Cache Cacheability Register (DCCR)	8-4
Figure 8-3.	Instruction Cache Cacheability Register (ICCR)	8-6
Figure 8-4.	Storage Guarded Register (SGR)	8-8
Figure 8-5.	Storage Little-Endian Register (SLER)	8-11
Figure 10-1.	Cache Debug Control Register (CDBCR)	10-5

Figure 10-2.	Condition Register (CR)	10-6
Figure 10-3.	Count Register (CTR)	10-7
Figure 10-4.	Data Address Compare Register (DAC)	10-8
Figure 10-5.	Debug Control Register (DBCR)	10-9
Figure 10-6.	Debug Status Register (DBSR)	10-11
Figure 10-7.	Data Cache Cacheability Register (DCCR)	10-13
Figure 10-8.	Data Cache Write-through Register (DCWR)	10-15
Figure 10-9.	Data Exception Address Register (DEAR)	10-17
Figure 10-10.	Exception Syndrome Register (ESR)	10-18
Figure 10-11.	Exception Vector Prefix Register (EVPR)	10-19
Figure 10-12.	General Purpose Register (R0-R31)	10-20
Figure 10-13.	Instruction Address Compare Register (IAC1)	10-21
Figure 10-14.	Instruction Cache Cacheability Register (ICCR)	10-22
Figure 10-15.	Instruction Cache Debug Data Register (ICDBDR)	10-24
Figure 10-16.	Link Register (LR)	10-25
Figure 10-17.	Machine State Register (MSR)	10-26
Figure 10-18.	Programmable Interval Timer (PIT)	10-28
Figure 10-19.	Processor Version Register (PVR)	10-29
Figure 10-20.	Storage Guarded Register (SGR)	10-30
Figure 10-21.	Storage Little-Endian Register (SLER)	10-32
Figure 10-22.	Special Purpose Register General (SPRG0-SPRG3)	10-34
Figure 10-23.	Save/Restore Register 0 (SRR0)	10-35
Figure 10-24.	Save/Restore Register 1 (SRR1)	10-36
Figure 10-25.	Save/Restore Register 2 (SRR2)	10-37
Figure 10-26.	Save/Restore Register 1 (SRR1)	10-38
Figure 10-27.	Time Base High Register (TBHI)	10-39
Figure 10-28.	Time Base High User-mode (TBHU)	10-40
Figure 10-29.	Time Base Low Register (TBLO)	10-41
Figure 10-30.	Time Base Low User-mode (TBLU)	10-42
Figure 10-31.	Timer Control Register (TCR)	10-43
Figure 10-32.	Timer Status Register (TSR)	10-44
Figure 10-33.	Fixed Point Exception Register (XER)	10-45
Figure A-1.	Instruction Format	A-49
Figure A-2.	B Instruction Format	A-49
Figure A-3.	SC Instruction Format	A-49
Figure A-4.	D Instruction Format	A-49
Figure A-5.	X Instruction Format	A-50
Figure A-6.	XL Instruction Format	A-51
Figure A-7.	XFX Instruction Format	A-51
Figure A-8.	XO Instruction Format	A-51

Figure A-9.	M Instruction Format	A-51
-------------	----------------------------	------

Tables

Table 2-1.	PPC401A1 SPRs	2-4
Table 2-2.	XER-Updating Arithmetic Instructions	2-7
Table 2-3.	Alignment Exception Summary	2-16
Table 2-4.	Bits of the BO Field	2-32
Table 2-5.	Conditional Branch BO Field	2-32
Table 2-6.	Example Memory Mapping	2-37
Table 2-7.	Instruction Execution Privileges and Operating Modes	2-38
Table 2-8.	Privileged Instructions	2-39
Table 2-9.	PPC401A1 Instruction Set Functional Summary	2-45
Table 2-10.	Instructions Specific to IBM PowerPC Embedded Controllers	2-45
Table 2-11.	Storage Reference Instructions	2-46
Table 2-12.	Arithmetic and Logical Instructions	2-47
Table 2-13.	Compare Instructions	2-47
Table 2-14.	Branch Instructions	2-48
Table 2-15.	Condition Register Logical Instructions	2-48
Table 2-16.	Rotate and Shift Instructions	2-48
Table 2-17.	Cache Control Instructions	2-49
Table 2-18.	Interrupt Control Instructions	2-49
Table 2-19.	Processor Management Instructions	2-50
Table 3-1.	Signal Name Prefix Definitions	3-3
Table 3-2.	CPM Interface I/O Signal Summary	3-7
Table 3-3.	CPU Control Interface I/O Signal Summary	3-10
Table 3-4.	TMM Mode Definitions	3-12
Table 3-5.	TMM/Test Interface I/O Signals	3-15
Table 3-6.	Clock Control to the PPC401A1 Latches	3-17
Table 3-7.	Reset Levels Supported by the PPC401A1	3-20
Table 3-8.	Valid Reset Input Combinations	3-21
Table 3-9.	Reset Interface I/O Signal Table	3-23
Table 3-10.	Instruction-Side BIU Interface Signal Summary	3-28
Table 3-11.	BIU_icuWdAddr(0:1) Decode	3-33
Table 3-12.	Key to ICU/PLB Timing Diagram Abbreviations	3-36
Table 3-13.	Data-Side BIU Interface I/O Signal Summary	3-46
Table 3-14.	DCU/PLB(BIU) Transfer Type Decoding	3-50
Table 3-15.	DCU_biuDTags(0:3) Allowed Values	3-50
Table 3-16.	PLB Prefetch Limitations for Guarded/Burst Mode Combinations	3-52
Table 3-17.	BIU_dcuWdAddr(0:1) Decode	3-56
Table 3-18.	Key to DCU/PLB Timing Diagram Abbreviations	3-59
Table 3-19.	DCU-BIU/PLB Timing Diagram Reference	3-60

Table 3-20.	DCR Interface I/O Signal Table	3-88
Table 3-21.	Non-Critical Category Exceptions Enabled by MSR[EE]	3-99
Table 3-22.	Critical Category Exceptions Enabled by MSR[CE]	3-99
Table 3-23.	EIC Interface I/O Signal Summary	3-100
Table 3-24.	JTAG Interface I/O Signal Summary	3-103
Table 3-25.	DBG Interface I/O Signal Summary	3-108
Table 4-1.	Contents of Registers After Reset	4-1
Table 5-1.	Exception-handling Priorities	5-4
Table 5-2.	Exception Vector Offsets	5-6
Table 5-3.	ESR Alteration by Various Exceptions	5-12
Table 5-4.	Register Settings during Critical Interrupt Exceptions	5-13
Table 5-5.	Register Settings during Machine Check—Instruction Exceptions	5-15
Table 5-6.	Register Settings during Machine Check—Data Exceptions	5-15
Table 5-7.	Register Settings during Data Storage Exceptions	5-16
Table 5-8.	Register Settings during External Interrupt Exceptions	5-17
Table 5-9.	Alignment Exception Summary	5-17
Table 5-10.	Register Settings during Alignment Error Exceptions	5-18
Table 5-11.	ESR Usage for Program Exceptions	5-19
Table 5-12.	Register Settings during Program Exceptions	5-19
Table 5-13.	Register Settings during System Call Exceptions	5-20
Table 5-14.	Register Settings during Programmable Interval Timer Exceptions	5-21
Table 5-15.	Register Settings during Fixed Interval Timer Exceptions	5-21
Table 5-16.	Register Settings during Watchdog Timer Exceptions	5-22
Table 5-17.	SRR2 during Debug Exceptions	5-23
Table 5-18.	Register Settings during Debug Exceptions	5-23
Table 5-19.	FIT Controls	5-29
Table 5-20.	Watchdog Timer Controls	5-30
Table 6-1.	Priority Changes With Different Data Cache Operations	6-15
Table 7-1.	Debug Events	7-4
Table 7-2.	DAC Applied to Cache Instructions	7-9
Table 7-3.	JTAG Connector Signals	7-12
Table 7-4.	JTAG Instructions	7-13
Table 9-1.	Instructions in the IBM PowerPC Embedded Environment	9-1
Table 9-2.	Operator Precedence	9-5
Table 9-3.	Extended Mnemonics for addi	9-9
Table 9-4.	Extended Mnemonics for addic	9-10
Table 9-5.	Extended Mnemonics for addic.	9-11
Table 9-6.	Extended Mnemonics for addis	9-12
Table 9-7.	Extended Mnemonics for bc, bca, bcl, bcla	9-21
Table 9-8.	Extended Mnemonics for bcctr, bcctrl	9-29

Table 9-9.	Extended Mnemonics for bclr, bclrl	9-33
Table 9-10.	Extended Mnemonics for cmp	9-37
Table 9-11.	Extended Mnemonics for cmpi	9-38
Table 9-12.	Extended Mnemonics for cmpl	9-39
Table 9-13.	Extended Mnemonics for cmpli	9-40
Table 9-14.	Extended Mnemonics for creqv	9-44
Table 9-15.	Extended Mnemonics for crnor	9-46
Table 9-16.	Extended Mnemonics for cror	9-47
Table 9-17.	Extended Mnemonics for crxor	9-49
Table 9-18.	Data Cache Array Tag Information	9-61
Table 9-19.	Instruction Cache Array Tag Information	9-74
Table 9-20.	Extended Mnemonics for mfspr	9-109
Table 9-21.	Extended Mnemonics for mtrcf	9-111
Table 9-22.	Extended Mnemonics for mtspr	9-115
Table 9-23.	Extended Mnemonics for nor, nor.	9-122
Table 9-24.	Extended Mnemonics for or, or.	9-123
Table 9-25.	Extended Mnemonics for ori	9-125
Table 9-26.	Extended Mnemonics for rlwimi, rlwimi.	9-129
Table 9-27.	Extended Mnemonics for rlwinm, rlwinm.	9-130
Table 9-28.	Extended Mnemonics for rlwnm, rlwnm.	9-133
Table 9-29.	Extended Mnemonics for subf, subf., subfo, subfo.	9-159
Table 9-30.	Extended Mnemonics for subfc, subfc., subfco, subfco.	9-161
Table 9-31.	Extended Mnemonics for tw	9-168
Table 9-32.	Extended Mnemonics for twi	9-172
Table 10-1.	PPC401A1 General Purpose Registers	10-2
Table 10-2.	Special Purpose Registers	10-3
Table 10-3.	ICU Tag Information	10-24
Table 11-1.	Signal Name Prefix Definitions	11-2
Table 11-2.	Ford Signal I/O Sorted Alphabetically by Signal Name	11-3
Table 11-3.	Ford Signal I/O Sorted Alphabetically by Hard Macro Name	11-5
Table A-1.	PPC401A1 Instruction Syntax Summary	A-2
Table A-2.	PPC401A1 Instructions by Opcode	A-38
Table B-1.	PPC401A1 Instruction Set Functional Summary	B-1
Table B-2.	Instructions Specific to PowerPC Embedded Controllers	B-2
Table B-3.	Privileged Instructions	B-3
Table B-4.	Extended Mnemonics for PPC401A1	B-5
Table B-5.	Storage Reference Instructions	B-26
Table B-6.	Arithmetic and Logical Instructions	B-31
Table B-7.	Condition Register Logical Instructions	B-35
Table B-8.	Branch Instructions	B-36

Table B-9.	Comparison Instructions	B-37
Table B-10.	Rotate and Shift Instructions	B-38
Table B-11.	Cache Control Instructions	B-39
Table B-12.	Interrupt Control Instructions	B-40
Table B-13.	Processor Management Instructions	B-41

About this Book

This user's manual provides the architectural overview, programming model, and detailed information about the registers and the instruction set of the IBM™ PowerPC™ 401A1 (PPC401A1™) 32-bit RISC embedded controller core.

The PPC401A1 RISC embedded controller core features

- PowerPC Architecture™
- Single-cycle execution for most instructions
- Instruction cache and data cache, with support for cache line locking
- Support for true Little-Endian operation
- Interrupt interface for one critical and one non-critical interrupt signal
- JTAG interface
- Extensive development tool support

Who Should Use This Book

This book is for system hardware and software developers, and for application developers who need to understand the PPC401A1. The audience should understand embedded controller design, embedded system design, operating systems, RISC processing, and design for testability.

How to Use This Book

This book describes the PPC401A1 device architecture, programming model, external interfaces, internal registers, and instruction set. This book contains the following chapters:

Chapter 1	Overview
Chapter 2	Programming Model
Chapter 3	I/O Interfaces
Chapter 4	Initialization
Chapter 5	Exceptions, Interrupts, and Timers
Chapter 6	Cache Operations
Chapter 7	Debugging and JTAG Facilities
Chapter 8	Storage Control
Chapter 9	Instruction Set
Chapter 10	Register Summary
Chapter 11	Signal Summary

This book contains the following appendixes:

Appendix A	Chapter A
Appendix B	Chapter B
Appendix C	Chapter C

To help readers find material in these chapters, the book contains:

Contents,	on p. v.
Figures,	on p. xv.
Tables,	on p. xix.
Index,	on p. X-1.

Conventions

The following is a brief list of notational conventions frequently used in this manual. Also, see Section 9.3, “Pseudocode,” on p. 9-2, which describes the notational conventions used in instruction descriptions.

$\overline{\text{Active_Low}}$	An overbar indicates an active-low signal.
0x1f	Hexadecimal numbers.
0b1001	Binary numbers.
FLD	A named field.
FLD _b	A bit in a named field.
RA, RS, . . .	A general purpose register (GPR).
(RA)	The contents of a GPR.
(RA 0)	The contents of a GPR or 0, if the RA field is 0.
REG _b	A bit in a named register.
REG _{b:b}	A range of bits in a named register.
REG _{b,b} , . . .	A list of bits, by number or name, in a named register.
REG[FLD]	A field of a named register.
CR _{FLD}	The field in the condition register pointed to by a field of an instruction.
²⁴ S	The sign bit replicated (sign-extended) 24 times.
xx	Bit positions that are don't-cares.

Related Publications

The following book describes the PowerPC Architecture:

The PowerPC Architecture: A Specification for a New Family of RISC Processors (Order Number 52G7487)

The following book describes the PowerPC 401 core models:

- *PowerPC 401 Core Models User's Guide* (Part Number 13H6990)

To obtain copies of these publications, contact your IBM Microelectronics representative.

The IBM PPC401A1 32-bit reduced instruction set computer (RISC) embedded controller core, implements the PowerPC Architecture with extensions for embedded applications.

This chapter describes:

- PPC401A1 features
- The layered PowerPC Architecture
- The PPC401A1 implementation of the IBM PowerPC Embedded Environment, an extension of the PowerPC Architecture for embedded applications
- PPC401A1 organization, including a block diagram and descriptions of the functional units
- PPC401A1 registers
- PPC401A1 addressing modes

1.1 PPC401A1 Features

The PPC401A1 32-bit RISC embedded controller core provides high performance and low power consumption. The PPC401A1 RISC CPU executes at sustained speeds approaching one cycle per instruction. On-chip instruction and data caches can be reduce chip count and design complexity in systems and improve system throughput.

PPC401A1 features include:

- PowerPC RISC fixed-point CPU
 - Thirty-two 32-bit general purpose registers (GPRs)
 - Branch prediction
 - Single-cycle execution for most instructions
 - Hardware multiply/divide for faster integer arithmetic
 - Enhanced string and multiple-word handling
 - Programmable Interval Timer (PIT), Fixed Interval Timer (FIT), and watchdog timer

- Storage Control
 - Separate, configurable instruction and data cache units
 - Flush queue for fill-first operations during cache misses
 - Operand forwarding during cache line fills
- Memory Management
 - True little endian operation and support for PowerPC Endian modes
- Core interfaces that support a wide range of function and performance:
 - Separate 32-bit instruction and data interfaces to the processor local bus (PLB) or other bus interface unit (BIU) designs
 - 32-bit device-paced Device Control Register (DCR) interface for system control
 - Clock and power management
 - JTAG debug interface

1.2 PowerPC Architecture

The PowerPC Architecture comprises three levels of standards:

- PowerPC User Instruction Set Architecture, including the base user-level instruction set, user-level registers, programming model, data types, and addressing modes. This is referred to as Book I of the PowerPC Architecture.
- PowerPC Virtual Environment Architecture, describing the memory model, cache model, cache-control instructions, address aliasing, and related issues. While accessible from the user level, these features are intended to be accessed from within library routines provided by the system software. This is referred to as Book II of the PowerPC Architecture.
- PowerPC Operating Environment Architecture, including the memory management model, supervisor-level registers, and the exception model. These features are not accessible from the user level. This is referred to as Book III of the PowerPC Architecture.

Book I and Book II define instructions and facilities available to the application programmer. Book III defines features, such as system-level instructions, that are not directly accessible by user applications.

The PowerPC Architecture guarantees application code compatibility across all PowerPC implementations to help maximize the cross-platform portability of applications developed for PowerPC processors. This is accomplished through compliance with the first level of architectural standard, the PowerPC User Instruction Set Architecture, which is common for all PowerPC implementations.

1.3 The PPC401A1 as a PowerPC Implementation

The PPC401A1 implements the PowerPC User Instruction Set Architecture, user-level registers, programming model, data types, and addressing modes for 32-bit fixed-point operations. The PPC401A1 fully complies with specifications for 32-bit implementations of the PowerPC User Instruction Set Architecture. The 64-bit operations are not supported, nor are the floating point operations. Such operations are trapped and can be emulated in software.

Most of the architected features of the PPC401A1 are compatible with the specifications for the PowerPC Virtual Environment and Operating Environment Architectures, as specified for processors such as the 6xx family of PowerPC processors. The PPC401A1 also provides a number of optimizations and extensions to the lower layers of the PowerPC Architecture. The full architecture of the PPC401A1 is defined by the PowerPC Embedded Environment and the PowerPC User Instruction Set Architecture.

The primary extensions of the PowerPC Architecture defined in the Embedded Environment are:

- A simplified memory management mechanism with enhancements for embedded applications
- An enhanced, dual-level interrupt structure
- An architected DCR address space for integrated system control functions
- The addition of several instructions to support these modified and extended resources

Finally, some of the specific implementation features of the PPC401A1 are beyond the scope of the PowerPC Architecture. These features are included to enhance performance, integrate functionality, and reduce system complexity in embedded control applications.

1.4 PPC401A1 Organization

The PPC401A1 consists of a three-stage pipelined processor core, memory management unit (MMU), separate instruction and data cache units, JTAG, debug, trace logic, and three timers. The PowerPC User Instruction Set Architecture and special purpose registers (SPRs) provide a high degree of user control over configuration and operation of the functional units, both interface and core.

illustrates the logical organization of the PPC401A1.

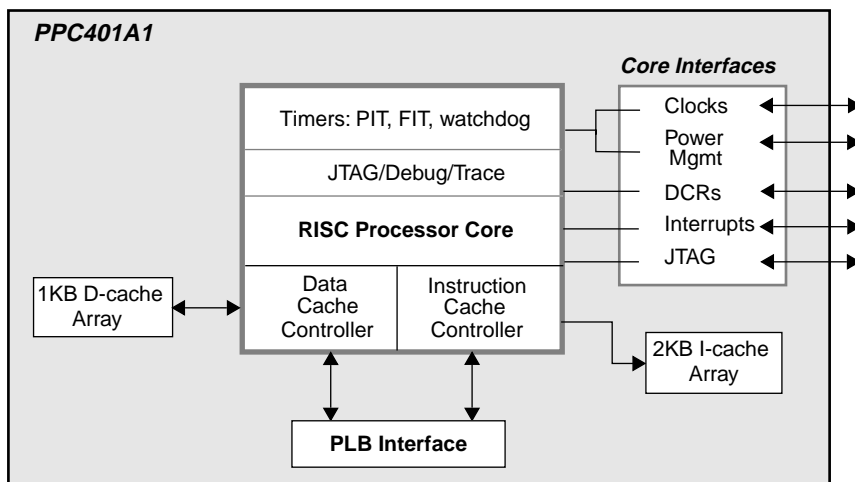


Figure 1-1. PPC401A1 Block Diagram

1.4.1 RISC Processor Core

The RISC processor core comprises a three-stage instruction pipeline with fetch, decode, and execute stages.

The fetcher provides an instruction stream to the execution unit (EXU). The fetcher speculatively requests up to two instructions from the instruction cache unit (ICU), using two prefetch buffers (PFBs) to queue incoming instructions when the EXU is busy. If the EXU is not busy, instructions from the ICU are forwarded directly to the decode stage. To reduce external bus contention, fetching is suspended when the PFBs are full or requests to fill available positions are pending.

To save area, the pre-fetch buffers do not have dedicated address registers. Because branches are only examined in decode, addresses associated with the pre-fetch buffers can be calculated from the internal decode address register as pre-fetched instructions move into the decode stage. When a branch is predicted taken, the next sequential instruction (NSI), if available, is saved in PFB 0. Because a branch predicted taken, but determined to be not taken, does not need to refetch the NSI, instruction fetch latency is reduced.

The fetcher uses an instruction bit to predict the direction of a conditional branch. If an unresolved conditional branch is encountered, the fetcher speculatively fetches along the predicted address path until the conditional branch is resolved. If the branch was incorrectly predicted, the fetcher aborts all instruction requests made down the mispredicted path. Aborts from the fetcher can propagate through the ICU to terminate requests to the BIU.

The PPC401A1 EXU is optimized for minimal area and power dissipation, yet completes most instructions in one cycle. The EXU supports aligned and unaligned storage accesses

and is the first PowerPC implementation that supports true little endian storage accesses. Hardware multiply and divide is implemented using existing data flow; the APU interface can be used to attach hardware units to accelerate these operations.

The EXU provides a 32 x 32 register file with two read ports and one write port. Although store instructions require three operands, and the load with update instructions require two write ports for single-cycle performance, significant area was saved by the reducing the register ports and the associated dataflow, resulting in 2-cycle loads and stores.

When the EXU is presented with a load or store instruction, the EXU decomposes the instruction into pseudo operations. All load and store operations, including strings and multiples, share the same structure. This simplifies the control logic. The control logic to support unaligned accesses was a natural extension of the byte steering logic required to handle string operations combined with the similar structure created by the pseudo operations. Such support for unaligned storage accesses can eliminate gaps, traditionally found in data structures where data must be aligned, to reduce system memory size.

An adjunct register improves the performance of string and unaligned storage accesses, and enables the data cache unit (DCU) to handle 1-, 2-, 3-, or 4-byte requests. The 1-, 2-, and 3-byte requests allow the EXU to align to a word boundary in one transfer. Once on a word boundary, word requests are made until the byte count is exhausted, or until the transfer is completed by a 1-, 2-, or 3-byte transfer. The adjunct register collects the unaligned data for load/store operations to allow word accesses to be made to the DCU.

The EXU uses one carry-skip adder for all arithmetic operations, including multiply/divide, and effective address calculation. The adjunct register stores the multiplier and dividend for multiply and divide operations.

1.4.2 Instruction and Data Cache Controllers

The PPC401A1 core provides a 2KB instruction cache unit (ICU) and a 1KB data cache unit (DCU) that allow concurrent accesses and minimize pipeline stalls. Both cache units are two-way set associative, use a 16-byte line size, and provide array built-in self test (ABIST) for manufacturing. The instruction set provides a rich assortment of cache control instructions, including instructions to read tag information and data arrays. See Chapter 6, “Cache Operations,” for detailed information about the ICU and DCU.

The cache units, optimized for minimal size and power consumption, maintain high performance. The cache units are PLB-compliant for use in the IBM Core+ASIC program.

1.4.2.1 Instruction Cache Unit (ICU)

The ICU provides one instruction per cycle to the EXU over a 32-bit bus. A line buffer (built into the output of the array for manufacturing test) enables the ICU to be accessed only once for every four instructions, to reduce power consumption by the array.

The ICU can forward any or all of the four words of a line fill to the EXU to minimize pipeline stalls caused by cache misses. The ICU aborts speculative fetches abandoned by the EXU,

eliminating unnecessary line fills and enabling the ICU to handle the next EXU fetch. Aborting abandoned requests also eliminates unnecessary external bus activity to increase external bus utilization.

Cache line locking can completely eliminate ICU misses in critical code. Cache line locking can be performed line by line, and is controlled by the Cache Debug Control Register (CDBCR) and Power PC 4xx instructions.

1.4.2.2 Data Cache Unit (DCU)

The DCU transfers one, two, three, or four bytes per cycle, depending on the number of byte enables presented by the CPU. The DCU contains a single-element command and store data queue to reduce pipeline stalls; this queue enables the DCU to independently process load/store and cache control instructions. Dynamic PLB request prioritization reduces pipeline stalls even further. When the DCU is busy with a low-priority request while a subsequent storage operation requested by the CPU is stalled, the DCU automatically increases the priority of the current request to the PLB.

The DCU uses a two-line flush queue to minimize pipeline stalls caused by cache misses. Line flushes are postponed until after a line fill is completed. Registers comprise the first position of the flush queue; the line buffer built into the output of the array for manufacturing test serves as the second position of the flush queue. Pipeline stalls are further reduced by forwarding the requested word to the CPU during the line fill. Single-queued flushes are non-blocking. When a flush operation is pending, the DCU can continue to access the array to determine subsequent load or store hits. Under these conditions, load hits can occur concurrently with store hits to write-back memory without stalling the pipeline. Requests abandoned by the CPU can also be aborted by the cache controller.

Cache line locking can completely eliminate DCU line misses in critical data. Cache line locking can be performed line by line, and is controlled by the CDBCR and Power PC 4xx instructions.

The DCU provides two additional features that allow the programmer to tailor its performance for a given application. The DCU can function in write-back or write-through mode, as controlled by the Data Cache Write-through Register (DCWR) or the translation look-aside buffer (TLB); performance of the cache controller can be tuned for a balance of performance and memory coherency. Write-on-allocate, controlled by the CDBCR[WOA] field, can inhibit line fills caused by a store miss to further reduce potential pipeline stalls and unwanted external bus traffic.

1.4.3 Timers

The PPC401A1 contains a time base and three timers: the Programmable Interval Timer (PIT), the Fixed Interval Timer (FIT), and a watchdog timer. The time base is a 64-bit counter incremented either by an internal signal equal to the CPU clock rate or by a separate external timer clock signal. No interrupts are generated when the time base rolls over.

The PIT is a 32-bit register that is decremented at the same rate as the time base is incremented. The user loads the PIT register with a value to create the desired delay. When the register is decremented to zeros, the timer stops decrementing, a bit is set in the Timer Status Register (TSR), and a PIT interrupt is generated. Optionally, the PIT can be programmed to reload automatically the last value written to the PIT register, after which the PIT begins decrementing again. The Timer Control Register (TCR) contains the interrupt enable for the PIT interrupt.

The FIT generates periodic interrupts based on selected bits in the time base. Users can select one of four intervals for the timer period by setting the appropriate bits in the TCR. When the selected bit in the time base changes from 0 to 1, a bit is set in the TSR and a FIT interrupt is generated. The FIT interrupt enable is contained in the TCR.

The watchdog timer generates a periodic interrupt based on selected bits in the time base. Users can select one of four time periods for the interval and the type of reset generated if the watchdog timer expires twice without an intervening clear from software.

1.4.4 Storage Control

The PPC401A1 has a 4 gigabyte (GB) address space, which is presented as a flat address space.

The PPC401A1 does not support address translation.

The PowerPC Architecture provides WIGE (write-through, cacheability, guarded, endian) storage attributes that control memory accesses. The PPC401A1 provides storage attribute control registers that set the various storage attributes for regions of memory. Each storage attribute control register contains 32 fields. Each field sets the associated storage attribute for a 128MB memory region. See Chapter 8, “Storage Control,” for more information about the storage attributes and storage attribute control registers.

1.4.5 Debug

The PPC401A1 debug facilities include debug modes for the various types of debugging used during hardware and software development. Also included are debug events that allow developers to control the debug process. Debug modes and debug events are controlled using debug registers in the chip. The debug registers are accessed either through software running on the processor, or through the JTAG port. The JTAG port can also be used for board test.

The debug modes, events, controls, and interfaces provide a powerful combination of debug facilities for a complete set of hardware and software development tools such as RISCWatch and OS Open from IBM.

1.4.5.1 Development Tool Support

The PPC401A1 provides powerful debug support for a wide range of hardware and software development tools.

The OS Open Real-time Operating System Debugger is an example of an operating system-aware debugger, implemented using software traps.

RISCWatch an example of a development tool that uses the external debug mode, debug events, and the JTAG port to support hardware and software development and debugging.

Logic analyzers from Hewlett-Packard and Tektronix provide PPC401A1 disassembler support and support for the RISCTrace feature of RISCWatch.

1.4.5.2 Debug Modes

The PPC401A1 supports two debug modes, internal and external; each mode supports a different type of debug tool used in embedded systems development. Internal debug mode supports ROM monitors, and external debug mode supports emulators. Both modes can be enabled simultaneously. The debug modes are controlled by the Debug Control Register (DBCR).

Internal debug mode supports accessing architected processor resources, setting hardware and software breakpoints, and monitoring processor status. In internal debug mode, debug events can generate debug exceptions, which can interrupt normal program flow so that monitor software can collect processor status and alter processor resources.

Internal debug mode relies on exception-handling software, running in the processor, and an external communications path, to debug software problems. This mode is used while the processor is executing instructions and enables debugging of problems in application or operating system code.

Access to debugger software executing in the processor, while in internal debug mode, is through a communications port on the processor board, such as a serial port.

External debug mode, accessed through a JTAG port, supports stopping and starting the processor, accessing architected processor resources, setting hardware and software breakpoints, and monitoring processor status. In external debug mode, debug events can architecturally “freeze” the processor. While the processor is frozen, normal instruction execution stops, and the architected processor resources can be accessed and altered.

External debug mode relies only on internal processor resources to debug system hardware and software problems. This mode can also be used for software development on systems without a control program, or to debug control program code.

1.4.6 Core Interfaces

The PPC401A1 core interfaces support a range of I/O interfaces the simplify the attachment of on-chip and off-chip devices. Chapter 3, “I/O Interfaces,” provides detailed descriptions of the available interfaces and their signals.

1.4.6.1 PLB (Processor Local Bus)

The PLB-compliant interface provides separate 32-bit address and data buses for the instruction and data sides.

1.4.6.2 DCR (Device Control Register)

The DCR interface supports the attachment of registers for device control. These registers are accessed using the **mfdcr** and **mtdcr** instructions.

1.4.6.3 Clock and Power Management

This interface supports several methods of clock distribution and power management.

1.4.6.4 JTAG

The PPC401A1 JTAG port is enhanced to support the attachment of a debug tool such as the RISCWatch product from IBM Microelectronics. Through the JTAG test access port, a debug workstation can single-step the processor and interrogate internal processor state to facilitate software debugging. The enhancements comply with the IEEE 1149.1 specification for vendor-specific extensions, and are therefore compatible with standard JTAG hardware for boundary-scan system testing.

1.4.6.5 Interrupts

The PPC401A1 provides an interface to an interrupt controller that is logically outside the PPC401A1 processor core. This controller combines the asynchronous interrupt inputs and presents them to the core as a single interrupt signal. The sources of asynchronous interrupts are external signals, the JTAG/debug unit, and any implemented peripherals.

1.4.7 Data Types

PPC401A1 operands are bytes, halfwords, or words. Multiple words or strings of bytes can be transferred using the load/store multiple and load/store string instructions. Data is represented in two's complement notation or in unsigned fixed-point format.

The address of a multi-byte operand is always the lowest memory address occupied by that operand. Byte ordering can be selected as big endian (the lowest memory address of an operand contains its most significant byte) or as little endian (the lowest memory address of an operand contains its least significant byte).

The PowerPC Endian mode can be set to automatically change when entering and leaving an interrupt handler. The PPC401A1 also supports true little endian addressing and data types. See Section 2.4, "Byte Ordering," on p. 2-17, for more information about PowerPC endian modes and true little endian operation.

1.4.8 Register Set Summary

The registers can be grouped into basic categories based on function and access mode: general purpose registers (GPRs), special purpose registers (SPRs), the machine state register (MSR), the condition register (CR), and, in standard products, device control registers (DCRs).

Chapter 10, “Register Summary,” provides a register diagram and a register field description table for each register.

1.4.8.1 General Purpose Registers

The PPC401A1 contains 32 GPRs; each register contains 32 bits. The contents of the GPRs can be transferred from memory using load instructions and stored to memory using store instructions. GPRs, which are specified as operands in many PPC401A1 instructions, can also hold instruction results and the contents of other registers.

1.4.8.2 Special Purpose Registers

Special Purpose Registers (SPRs), which are part of the PowerPC Architecture, are accessed using the **mtspr** and **mfspr** instructions. SPRs control the use of the debug facilities, timers, interrupts, storage control attributes, and other architected processor resources.

The only SPRs that are not privileged for read and write access are the Count Register (CTR), Link Register (LR), and Fixed Point Exception Register (XER). User-mode programs have read-only access to the Time Base High User-mode (TBHU) and Time Base Low User-mode (TBLU) time base registers.

1.4.8.3 Machine State Register

The PPC401A1 contains a 32-bit Machine State Register (MSR). The contents of a GPR can be written to the MSR using the **mtmsr** instruction, and the MSR contents can be read into a GPR using the **mfmssr** instruction. The MSR contains fields that control the operation of the PPC401A1.

1.4.8.4 Condition Register

The PPC401A1 contains a 32-bit Condition Register (CR). These bits are grouped into eight 4-bit fields, CR[CR0]–CR[CR7]. Instructions are provided to perform logical operations on CR fields and bits within fields and to test CR bits within fields. The CR fields, which are set by compare instructions, can be used to control branches. CR[CR0] can be set implicitly by arithmetic instructions.

1.4.8.5 Device Control Registers

Device Control Registers (DCRs), which are architecturally outside of the processor core, are accessed using the **mtdcr** and **mfdcr** instructions. DCRs are used to control, configure, and hold status for various functional units that are not part of the processor core. Although the PPC401A1 does not contain DCRs, the **mtdcr** and **mfdcr** instructions are provided.

The **mtdcr** and **mfdcr** instructions are privileged, for all DCRs; therefore, all accesses to DCRs are privileged. See Section 2.8, “Privileged Mode Operation,” on p. 2-38.

All DCR numbers are reserved, and should be neither read nor written, unless they are part of an IBM Core+ASIC implementation.

1.4.9 Addressing Modes

The PPC401A1 supports the following addressing modes to allow efficient retrieval and storage of data in memory:

- Base plus displacement addressing
- Indexed addressing
- Base plus displacement addressing and indexed addressing, with update

In the base plus displacement addressing mode, an effective address (EA) is formed by adding a displacement to a base address contained in a GPR (or to an implied base of 0). The displacement is an immediate field in an instruction.

In the indexed addressing mode, the EA is formed by adding an index contained in a GPR to a base address contained in a GPR (or to an implied base of 0).

The base plus displacement and the indexed addressing modes also have a “with update” mode. In “with update” mode, the effective address calculated for the current operation is saved in the base GPR, and can be used as the base in the next operation. The “with update” mode relieves the processor from repeatedly loading a GPR with an address for each piece of data, regardless of the proximity of the data in memory.

Programming Model

The programming model of the PPC401A1 embedded controller describes how the following features and operations of the PPC401A1 appear to programmers:

- Memory organization and addressing, starting on p. 2-1
- Registers, starting on p. 2-2
- Data types and alignment, starting on p. 2-15
- Byte ordering, starting on p. 2-17
- Instruction processing, starting on p. 2-30
- Branching control, starting on p. 2-30
- Speculative accesses, starting on p. 2-34
- Privileged mode operation, starting on p. 2-38
- Synchronization, starting on p. 2-40
- Instruction set, starting on p. 2-44

2.1 Memory Organization and Addressing

The PowerPC Architecture defines a 32-bit, 4 gigabyte (GB) flat address space for instructions and data.

The user's manuals for standard products containing the PPC401A1 describe their memory organizations and physical address maps.

2.1.1 Storage Attributes

The PowerPC Architecture defines storage attributes that control data and instruction accesses. Storage attributes are provided to control cache write-through policy (the W storage attribute), cacheability (the I storage attribute), memory coherency in multi-processor environments (the M storage attribute), and guarding against speculative memory accesses (the G storage attribute). The IBM PowerPC Embedded Environment defines additional storage attributes for storage compression (the K storage attribute) and byte ordering (the E storage attribute).

The PPC401A1 provides control mechanisms for the WIGE attributes. Because the

PPC401A1 does not provide hardware support for multi-processor environments, the M storage attribute is not implemented in hardware.

The storage attribute control registers control the storage attributes. These registers are:

- Data Cache Write-through Register (DCWR)
- Data Cache Cacheability Register (DCCR)
- Instruction Cache Cacheability Register (ICCR)
- Storage Guarded Register (SGR)
- Storage Little-Endian Register (SLER)

Chapter 10, “Register Summary,” contains bit descriptions for these registers.

Each storage attribute control register contains 32 bits; each bit controls one of thirty-two 128 MB storage attribute control regions. Bit 0 of each register controls the lowest-order region, with ascending bits controlling ascending regions in memory. Each region is selected by address bits A0:A4. The storage attributes in each storage attribute region are set independently.

2.2 Registers

Some of the more commonly-used registers are described in this section. Other registers are covered in their respective topic chapters (for example, the cache registers are described in Chapter 6, “Cache Operations”). All registers are summarized in Chapter 10, “Register Summary.”

All registers in the PPC401A1 are 32-bit registers. The registers are grouped into categories, based on access mode: general purpose registers (GPRs), special purpose registers (SPRs), the time base, the Machine State Register (MSR), the Condition Register (CR), and, in standard products, device control registers (DCRs).

For all registers with fields marked as *reserved*, the reserved fields should be written as 0 and read as *undefined*. That is, when writing to a register with a reserved field, write a 0 to the reserved field. When reading from a register with a reserved field, ignore that field. A good coding practice is to perform the initial write to a register with reserved fields as described, and to perform all subsequent writes to the register using a read-modify-write strategy: read the register, use logical instructions to alter defined fields, leaving reserved fields unmodified, and write the register.

2.2.1 General Purpose Registers

The PPC401A1 contains 32 general purpose registers (GPRs); each contains 32 bits. Data from memory can be loaded into GPRs using load instructions; the contents of GPRs can be

stored in memory using store instructions. Most integer instructions reference GPRs. See Table 10-1 on p. 10-2 for the numbering of the GPRs.

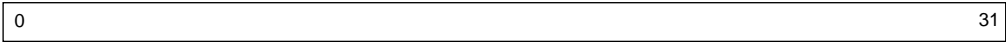


Figure 2-1. General Purpose Register (R0-R31)



2.2.2 Special Purpose Registers

Special Purpose Registers (SPRs), which are part of the PowerPC Architecture and the IBM PowerPC Embedded Environment, are accessed using the **mtspr** and **mfspr** instructions.

SPRs control the use of the debug facilities, timers, interrupts, storage control attributes, and other architected processor resources. Table 10-2 on p. 10-3 shows the mnemonic, name, and number for each SPR. Table 2-1 lists the PPC401A1 SPRs by function and points to the pages where the SPRs are described more fully.

Table 2-1. PPC401A1 SPRs

Function	Register				Access	Page	
Branch Control	CTR				User	2-4	
	LR				User	2-5	
Debug	CDBCR				Privileged	6-9	
	DAC1				Privileged	7-8	
	DBCR				Privileged	7-5	
	DBSR				Privileged	7-7	
	IAC1				Privileged	7-10	
	ICDBDR				Privileged	6-10	
Fixed-point Exception	XER				User	2-6	
General-purpose	SPRG0	SPRG1	SPRG2	SPRG3	Privileged	2-8	
Interrupts and Exceptions	DEAR				Privileged	5-12	
	ESR				Privileged	5-10	
	EVPR				Privileged	5-10	
	SRR0				SRR1	Privileged	5-8
	SRR2				SRR3	Privileged	5-9
Processor Version	PVR				Privileged, read-only	2-8	
Storage Attributes	DCCR				Privileged	8-4	
	DCWR				Privileged	8-2	
	ICCR				Privileged	8-6	
	SGR				Privileged	8-8	
	SLER				Privileged	8-10	
Timer Facilities	TBHI	TBLO			Privileged	5-25	
	TBHU	TBLU			User read-only	5-25	
	PIT	Privileged			5-28		
	TCR	Privileged			5-33		
	TSR	Privileged			5-32		

Except for the Link Register (LR), the Count Register (CTR), the Fixed-point Exception Register (XER), and the Time Base High User-mode (TBHU) and Time Base Low User-mode (TBLU), all SPRs are privileged. See Section 2.8.3, “Privileged SPRs,” on p. 2-39. Note that the Processor Version Register (PVR) is read-only.

2.2.2.1 Count Register (CTR)

The CTR is written from a GPR using the **mtspr** instruction. The CTR contents can be used as a loop count that is decremented and tested by some branch instructions. This usage does not incur any performance penalty; the branches execute in the normal branch

instruction execution time. Alternatively, the CTR contents can specify a target address for the **bcctr** instruction, enabling indirectly-addressed branching to any address.

The CTR is available to user programs.



Figure 2-2. Count Register (CTR)

0:31		Count	Used as count for branch conditional with decrement instructions, or as address for branch-to-counter instructions
------	--	-------	--

2.2.2.2 Link Register (LR)

The LR is written from a GPR using the **mtspr** instruction or branch instructions that have the LK bit set to 1. Such branch instructions load the LR with the address of the instruction following the branch instruction (4 + address of the branch instruction). Thus, the LR contents can be a return address for a subroutine which was entered using the branch.

The LR contents can be used as a target address for the **bclr** instruction. This allows indirectly-addressed branching to any address.

When the LR contents represent an instruction address, LR_{30:31} are assumed to be zero, because all instructions must be word-aligned. However, when LR is written using **mtspr** and then read using **mfspr**, all 32 bits are returned.

The LR is available to user programs.

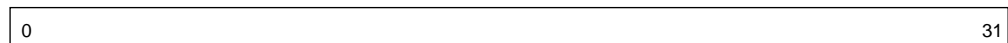


Figure 2-3. Link Register (LR)

0:31		Link Registers contents	If (LR) represents an instruction address, LR _{30:31} should be zero.
------	--	-------------------------	--

2.2.2.3 Fixed Point Exception Register (XER)

The XER records overflow and carry conditions from arithmetic operations.

The Summary Overflow (SO) field does not necessarily indicate that an overflow occurred on the most recent arithmetic operation, but that one occurred previously sometime since the last clearing of the XER. XER[SO] can be set to zero only by using the **mtspr** instruction or the **mcrxr** instruction.

The TBC field can be written, using **mtspr**, with a byte count for load/store string instructions. The XER is available to user programs.

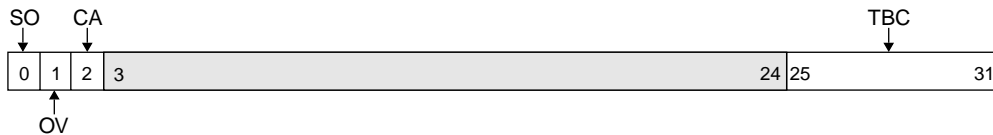


Figure 2-4. Fixed Point Exception Register (XER)

0	SO	Summary Overflow 0 No overflow has occurred. 1 Overflow has occurred.	Can be <i>set</i> by mtspr or using arithmetic instructions with the “OE” option (see Table 2-2 on p. 2-7); can be <i>reset</i> by mtspr or by mcrxr .
1	OV	Overflow 0 No overflow has occurred. 0 Overflow has occurred.	Can be <i>set</i> by mtspr or arithmetic instructions with the “OE” option (see Table 2-2 on p. 2-7); can be <i>reset</i> by mtspr , by mcrxr , or by arithmetic instructions with the “OE” option.
2	CA	Carry 0 Carry has not occurred. 1 Carry has occurred.	Can be <i>set</i> by mtspr or arithmetic instructions that update the CA field (see Table 2-2 on p. 2-7); can be <i>reset</i> by mtspr , by mcrxr , or by arithmetic instructions that update the CA field.
3:24		Reserved	
25:31	TBC	Transfer Byte Count	Used by lswx and stswx ; written by mtspr

Table 2-2. XER-Updating Arithmetic Instructions

Update XER[CA]		Update XER[OV] Set XER[SO]	
addc	subfc	addo	mullwo
addc.	subfc.	addo.	mullwo.
addco	subfco	addco	nego
addco.	subfco.	addco.	nego.
adde	subfe	addeo	subfo
adde.	subfe.	addeo.	subfo.
addeo	subfeo	addmeo	subfco
addeo.	subfeo.	addmeo.	subfco.
addic	subfic	addzeo	subfeo
addic.	subfme	addzeo.	subfeo.
addme	subfme.	divwo	subfmeo
addme.	subfmeo	divwo.	subfmeo.
addmeo	subfmeo.	divwuo	subfzeo
addmeo.	subfze	divwuo.	subfzeo.
addze	subfze.		
addze.	subfzeo		
addzeo	subfzeo.		
addzeo.			

Several special cases are associated with the use of the XER bits:

XER[SO] Summary overflow; set to 1 when an instruction causes XER[OV] to be set to 1, except for **mtspr**(XER), which sets XER[SO,OV] to the value of bit positions 0 and 1 in the source register, respectively. Once set, XER[SO] is not reset until an **mtspr**(XER) is executed with data that explicitly puts a 0 in the SO bit, or until an **mcrxr** instruction is executed.

XER[OV] Overflow; set to indicate whether or not an instruction that updates XER[OV] produces a result that “overflows” the 32-bit target register. XER[OV] = 1 indicates overflow. For arithmetic operations, this occurs when an operation has a carry-in to the most-significant bit of the instruction result that does not equal the carry-out of the most-significant bit (that is, the exclusive-or of the carry-in and the carry-out is 1).

The following instructions set XER[OV] differently. The specific behavior is indicated in the instruction descriptions.

- Move instructions
mcrxr, **mtspr**(XER)
- Multiply and divide instructions
mullwo, **mullwo.**, **divwo**, **divwo.**, **divwuo**, **divwuo.**

XER[CA]

Carry; set to indicate whether or not an instruction that updates XER[CA] produces a result that has a carry-out of the most-significant bit. XER[CA] = 1 indicates a carry.

The following instructions set XER[CA] differently. The specific behavior is indicated in the instruction descriptions.

- Move instructions
mcrxr, **mtspr**(XER)
- Shift-algebraic operations
sraw, **srawi**

XER[TBC]

Transfer Byte Count.

This field provides a byte count for the **lswx** and **stswx** instructions.

This field is updated by **mtspr**(XER).

2.2.2.4 Special Purpose Register General (SPRG0-SPRG3)

These four registers are provided as temporary storage locations. For example, a supervisor routine might save the contents of a GPR to an SPRG, and later restore the GPR from it. This is faster than the standard save/restore to a memory location. These registers are written to using the **mtspr** instruction and read from using the **mfspir** instruction.

Access to the SPRGs is privileged. See Section 2.8.3, “Privileged SPRs,” on p. 2-39 for more information.



Figure 2-5. Special Purpose Register General (SPRG0-SPRG3)

0-31		General data	Privileged user-specified; no hardware usage.
------	--	--------------	---

2.2.2.5 Processor Version Register (PVR)

The PVR is a read-only register that identifies the processor by Version and Revision numbers. Software can use features that depend upon an exact identification of the target processor. Such software can examine the PVR to select appropriate features dynamically.

The 16-bit Version number (comprised of the FAM and MEM fields) is assigned by the PowerPC Architecture process.

The 16-bit Revision number (comprised of the CORE and CHIP fields) is assigned by the chip implementer.

Access to the PVR is privileged. See Section 2.8.3, “Privileged SPRs,” on p. 2-39 for more information.

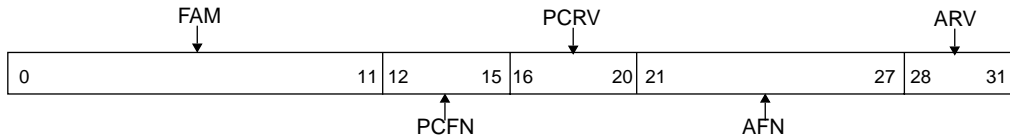


Figure 2-6. Processor Version Register (PVR)

0:11	FAM	Processor Family. Identifies a PowerPC family, such as 4xx or 6xx.	0x002 for the 4xx family.
12:15	PCFN	Processor Core Function. Identifies a specific processor core implementation.	1 for PPC401A1.
16:20	PCRV	Processor Core Revision. Identifies a revision of the processor core defined by the PFN field.	.
21:27	AFN	ASIC Function. An assigned identifier for an ASIC containing a PowerPC 400 Series processor core.	
28:31	ARV	ASIC Revision. An assigned identifier for a revision of the ASIC defined by the AFN field.	

2.2.3 Condition Register (CR)

The Condition Register (CR) contains eight 4-bit fields (CR0–CR7), as shown in Figure 2-7. The CR reflects the results of some operations (as indicated in the instruction descriptions in Chapter 9, “Instruction Set”). The CR supports condition testing and conditional branching.

Fields of the CR can be set in any of the following ways:

- Specified fields can be set by writing to the CR from a GPR (**mtcrf** instruction).
- A specified field can be set by writing to the field from another CR field (**mcrf** instruction) or from the XER (**mcrxr** instruction).
- CR[CR0] can be set as the implicit result of various fixed-point instructions.
- The bits in a specified field can be set as the result of a Compare instruction.

Additional instructions perform logical operations on one or more bits in a CR field (the CR-logical instructions); other instructions (the branch conditional instructions) test the bits in a CR field.

If a CR field is set by a compare instruction, the bits in the selected field are set as described in Section 2.2.3.1. The CR[CR0] field is altered implicitly by numerous instructions; the interpretation of CR[CR0] is discussed further in Section 2.2.3.2.

The CR is non-privileged. See Section 2.8.3, “Privileged SPRs,” on p. 2-39 for more information.

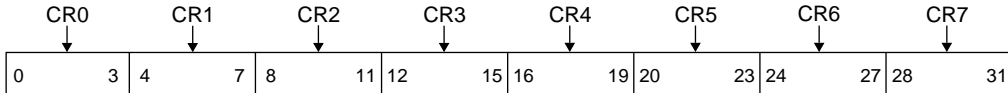


Figure 2-7. Condition Register (CR)

Figure 2-7. Condition Register (CR)			
0:3	CR0	Condition Register Field 0	CR[CRn] _{0:3} indicate less than, greater than, equal to, and summary overflow, respectively.
4:7	CR1	Condition Register Field 1	See the description of CR[CR0].
8:11	CR2	Condition Register Field 2	See the description of CR[CR0].
12:15	CR3	Condition Register Field 3	See the description of CR[CR0].
16:19	CR4	Condition Register Field 4	See the description of CR[CR0].
20:23	CR5	Condition Register Field 5	See the description of CR[CR0].
24:27	CR6	Condition Register Field 6	See the description of CR[CR0].
28:31	CR7	Condition Register Field 7	See the description of CR[CR0].

2.2.3.1 CR Fields after Compare Instructions

Compare instructions compare the values of two 32-bit numbers. The two types of compare instructions, *arithmetic* and *logical*, are distinguished by the interpretation given to the 32-bit numbers. For *arithmetic* compares, the numbers are considered to be signed, where 31 bits are significant; the most-significant bit is a sign bit. For *logical* compares, the numbers are considered to be unsigned (all 32 bits are significant; there is no sign bit). As an example, consider the comparison of 0 with 0xFFFF FFFF. In an *arithmetic* compare, 0 is larger; in a *logical* compare, 0xFFFF FFFF is larger.

A compare instruction can direct its results to any CR field. The BF field (bits 6:8) of the instruction specifies the CR field. The first data operand of a compare instruction specifies a GPR. The second data operand specifies another GPR, or immediate data derived from the

IM field (bits 16:31) of the immediate instruction form. The contents of the GPR specified by the first data operand are compared with the contents of the GPR specified by the second data operand (or with the immediate data). See descriptions of the compare instructions (p. 9-37 through p. 9-40) for precise details.

After a compare, the specified CR field is interpreted as follows:

LT (bit 0)	The first operand is less than the second operand.
GT (bit 1)	The first operand is greater than the second operand.
EQ (bit 2)	The first operand is equal to the second operand.
SO (bit 3)	Summary overflow; a copy of XER[SO].

2.2.3.2 The CR0 Field

After the execution of compare instructions with $BF = 0$, the $CR[CR0]$ is interpreted as described in Section above. The “dot” forms of arithmetic and logical instructions also alter $CR[CR0]$. After most fixed-point instructions that update $CR[CR0]$, the bits of $CR0$ are interpreted as follows:

LT (bit 0)	Less than 0; set if the most-significant bit of the 32-bit result is 1.
GT (bit 1)	Greater than 0; set if the 32-bit result is non-zero and the most-significant bit of the result is 0.
EQ (bit 2)	Equal to zero; set if the 32-bit result is 0.
SO (bit 3)	Summary overflow; a copy of XER[SO] at instruction completion.

The $CR[CR0]_{LT, GT, EQ}$ subfields are set as the result of an algebraic comparison of the instruction result to 0, regardless of the type of instruction that sets $CR[CR0]$. If the instruction result is 0, the EQ subfield is set to 1. If the result is not 0, whether the LT subfield or the GT subfield is set depends on the value of the most-significant bit of the instruction result.

When updating $CR[CR0]$, the most significant bit of an instruction result is considered a sign bit, even for instructions that produce results that are not usually thought of as signed. For example, logical instructions such as **and.**, **or.**, and **nor.** update $CR[CR0]_{LT, GT, EQ}$ using such an arithmetic comparison to 0, although the result of such a logical operation is often not actually an arithmetic result.

Note that if an arithmetic overflow occurs, the “sign” of an instruction result indicated by $CR[CR0]_{LT, GT, EQ}$ might not represent the “true” (infinitely precise) algebraic result of the instruction that set $CR0$. For example, if an **add.** instruction adds two large positive numbers and the magnitude of the result cannot be represented as a two’s-complement number in a 32-bit register, an overflow occurs and $CR[CR0]_{LT, SO}$ are set, although the infinitely precise result of the add is positive.

Adding the largest 32-bit twos-complement negative number, 0x8000 0000, to itself results in an arithmetic overflow and 0x0000 0000 is recorded in the target register. CR[CR0]_{EQ, SO} is set, indicating a result of 0, but the infinitely precise result is negative.

The CR[CR0]_{SO} subfield is a copy of XER[SO]. Instructions that do not alter the XER[SO] bit cannot cause an overflow, but even for these instructions CR[CR0]_{SO} is a copy of XER[SO].

Some instructions set CR[CR0] differently or do not specifically set any of the subfields. These instructions include:

- Compare instructions
cmp, cmpi, cmpl, cmpli
- CR logical instructions
crand, crandc, creqv, crnand, crnor, cror, crorc, crxor, mcrf
- Move CR instructions
mtcrf, mcrxr
- **stwcx**

The instruction descriptions provide detailed information about how the listed instructions alter CR[CR0].

2.2.4 The Time Base

The PPC401A1 implements a 64-bit time base. The time base, which increments once during each period of the time base clock, provides a time reference. The time base is accessed using the 32-bit registers TBLO and TBHI. Software access to the time base is through the **mfspr** and **mtspr** instructions.

Access to the time base registers TBHI and TBLO is privileged.

User-mode read-only access to the Time Base is provided by reading from different SPR numbers. Specifically, read-only access to TBHI is accomplished by reading TBHU, and read-only access to TBLO is accomplished by reading TBLU. Both TBHU and TBLU are read using **mfspr** instructions. An **mtspr** to these registers is boundedly undefined.

The time base differs from the time base described in the Power PC Architecture. See Section 5.15.1, “Time Base,” on p. 5-25, for detailed differences between the PPC401A1 time base and the time base described in the Power PC Architecture.

2.2.5 Machine State Register

The Machine State Register (MSR) controls important chip functions, such as the enabling or disabling of interrupts and debugging exceptions.

The MSR can be written from a GPR using the **mtmsr** instruction. The contents of the MSR can be written into a GPR using the **mfmsr** instruction. The MSR[EE] (External Interrupt Enable) bit may be set/cleared atomically using the **wrttee** or **wrtteei** instructions.

The MSR contents are automatically saved, altered, and restored by the interrupt-handling mechanism. See Section 5.3, “General Exception Handling Registers,” on p. 5-7.

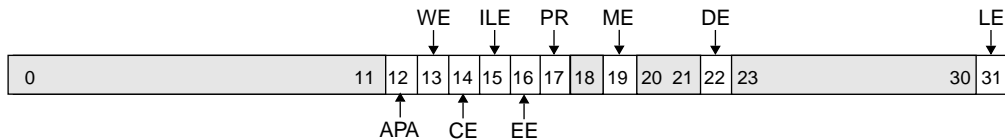


Figure 2-8. Machine State Register (MSR)

0:12		Reserved	
12	APA	Auxiliary Processor Available 0 Auxiliary processor not available. 1 Auxiliary processor available.	
13	WE	Wait State Enable 0 The processor is not in the wait state. 1 The processor enters the wait state until an exception is taken, or the PPC401A1 is reset, or an external debug tool clears WE.	
14	CE	Critical Interrupt Enable 0 Critical interrupts are disabled. 1 Critical interrupts are enabled.	CE controls the critical interrupt input and watchdog timer first time-out interrupts.
15	ILE	Interrupt Little Endian 0 Interrupt handlers execute in big endian mode. 1 Interrupt handlers execute in PowerPC little endian mode.	MSR(ILE) is copied to MSR(LE) when an interrupt is taken.
16	EE	External Interrupt Enable 0 Asynchronous exceptions are disabled. 1 Asynchronous exceptions are enabled.	EE controls the non-critical external interrupt input, Programmable Interval Timer, and Fixed Interval Timer interrupts.
17	PR	Problem State 0 Supervisor State (all instructions allowed) 1 Problem State (some instructions not allowed)	

Figure 2-8. Machine State Register (MSR) (cont.)

18		Reserved
19	ME	Machine Check Enable 0 Machine check exceptions are disabled 1 Machine check exceptions are enabled.
20:21		Reserved
22	DE	Debug Exception Enable 0 Debug exceptions are disabled. 1 Debug exceptions are enabled.
23:30		Reserved
31	LE	Little Endian 0 Processor executes in big endian mode. 1 Processor executes in PowerPC little endian mode.

2.2.6 Device Control Registers

Device Control Registers (DCRs), on-chip registers that exist architecturally outside the processor core, are not part of the IBM PowerPC Embedded Environment. The Embedded Environment simply defines the existence of a DCR address space and the instructions that access the DCRs, but does not define any DCRs. The instructions that access the DCRs are **mtdcr** (move to device control register) and **mfdcr** (move from device control register).

2.3 Data Types and Alignment

PPC401A1 data types consist of bytes (eight bits), halfwords (two bytes), words (four bytes), and strings (one or more bytes containing character data). Figure 2-9 shows the byte, halfword, and word data types and their bit and byte definitions.

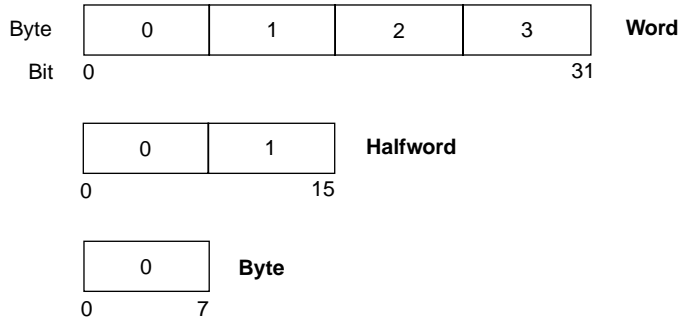


Figure 2-9. PPC401A1 Data Types

Data is represented in two's complement notation or in an unsigned integer format; data representation is independent of alignment issues.

The address of an a data object is always the lowest address of any byte comprising the object.

All instructions are words, and are word-aligned (the byte address is divisible by 4).

2.3.1 Alignment for Storage Reference and Cache Control Instructions

The storage reference instructions (loads and stores; see Table 2-11, "Storage Reference Instructions," on p. 2-46) move data to and from storage. The data cache control instructions (see Table 2-17, "Cache Control Instructions," on p. 2-49) control the contents and operation of the data cache unit (DCU). Both types of instructions form an effective address (EA). The method of calculating the EA for the storage reference and cache control instructions is detailed in the description of those instructions. See Chapter 9, "Instruction Set," for more information.

Cache control instructions ignore the four least significant bits in the EA; no alignment restrictions exist in the DCU because of EAs. However, storage control attributes for a storage region can cause alignment exceptions. Specifically, when a **dcbz** instruction references a region that is non-cacheable or for which write-through caching is enabled, an alignment exception is taken. Such exceptions result from the storage control attributes, not from EA alignment.

Alignment requirements for the EAs of the storage reference instructions and the **dcread** cache control instruction depends on the instruction and the endian mode of the PPC401A1 (see Section 2.4, "Byte Ordering," on p. 2-17 for information about endian operation).

Table 2-3, “Alignment Exception Summary,” on p. 2-16, summarizes the instructions that cause alignment exceptions.

The data targets of instructions are of types that depend upon the instruction. The load/store instructions have the following “natural” alignments (note that the PPC401A1 implementation handles misalignments within and across word boundaries):

- Load/store word instructions have word targets, word-aligned.
- Load/ store halfword instructions have halfword targets, halfword-aligned.
- Load/store byte instructions have byte targets, byte-aligned (that is, any alignment).

Misalignments are addresses that are not naturally aligned on data type boundaries. An address not divisible by four is misaligned with respect to word instructions. An address not divisible by two is misaligned with respect to halfword instructions.

2.3.2 Alignment and Endian Operation

When the PPC401A1 is operating as a big endian processor (MSR[LE] = 0), EA misalignments do not cause alignment exceptions except as summarized in Table 2-3,.

When the PPC401A1 is in PowerPC little endian mode (MSR[LE] = 1), EAs formed by the storage reference instructions must be aligned on a corresponding operand boundary. An alignment exception is taken for a storage reference instruction whenever the calculated EA does not match the required data alignment for the instruction. Misalignment indicates a coding error of improper data alignment or address calculation, or both. In general, the alignment error handler is expected to emulate the failing operation.

In PowerPC little endian mode, load/store string and multiple instructions always cause alignment exceptions.

The endian storage control attribute does not independently affect alignment behavior. In little endian storage regions, the alignment of data is treated as it is in big endian storage regions; no special alignment exceptions occur when accessing data in little endian storage regions. Note that the alignment exceptions that apply to big endian region accesses also apply to little endian storage region accesses.

2.3.3 Summary of Instructions Causing Alignment Exceptions

Table 2-3 summarizes the instructions that cause alignment exceptions and the conditions under which the alignment exceptions occur.

Table 2-3. Alignment Exception Summary

PPC401A1 MSR	Instructions Causing Alignment Exceptions	Conditions
MSR[LE] = 0	dcbz	EA in non-cacheable or write-through storage
	dcread, lwarx, stwcx.	EA not word-aligned

Table 2-3. Alignment Exception Summary (cont.)

PPC401A1 MSR	Instructions Causing Alignment Exceptions	Conditions
MSR[LE] = 1	dcbz	EA in non-cacheable or write-through storage
	lha, lhau, lhaux, lhax, lhbrx, lhz, lhzu, lhzux, lhzx, sth, sthbrx, sthu, sthux, sthx	EA not halfword-aligned
	dcread, lwarx, lwbrx, lwz, lwzu, lwzux, lwzx, stw, stwbrx, stwcx., stwu, stwux, stwx	EA not word-aligned
	lmw, lswi, lswx, stmw, stswi, stswx, stswcx.	Always

2.4 Byte Ordering

If scalars (individual data items and instructions) were indivisible, there would be no such concept as “byte ordering.” It is meaningless to consider the order of bits or groups of bits within the smallest addressable unit of storage; nothing can be observed about such order. Only when scalars, which the programmer and processor regard as indivisible quantities, can comprise more than one addressable unit of storage does the question of order arise.

For a machine in which the smallest addressable unit of storage is the 64-bit doubleword, there is no question of the ordering of bytes within doublewords. All transfers of individual scalars between registers and storage are of doublewords, and the address of the byte containing the high-order eight bits of a scalar is no different from the address of a byte containing any other part of the scalar.

For the PowerPC Architecture, as for most computer architectures currently implemented, the smallest addressable unit of storage is the 8-bit byte. Many scalars are halfwords, words, or doublewords, which consist of groups of bytes. When a word-length 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 eight bits of the scalar, which byte contains the next-highest-order eight bits, and so on.

Given a scalar that contains 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”) eight bits of the scalar, the next sequential address to the next-highest-order eight bits, and so on.

This ordering 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/390, and Motorola 680x0 are examples of computers using this byte ordering.

- The ordering that assigns the lowest address to the lowest-order (“rightmost”) eight bits of the scalar, the next sequential address to the next-lowest-order eight bits, and so on.

This ordering 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.

2.4.1 Structure Mapping Examples

The following C language structure, *s*, contains an assortment of scalars and a character string. The comments show the value assumed to be in each structure element; these values show how the bytes comprising each structure element are mapped into storage.

```
struct {
    int a;           /* 0x1112_1314 word */
    long long b;     /* 0x2122_2324_2526_2728 doubleword */
    char *c;         /* 0x3132_3334 word */
    char d[7];       /* 'A','B','C','D','E','F','G' array of bytes */
    short e;         /* 0x5152 halfword */
    int f;           /* 0x6162_6364 word */
} s;
```

C structure mapping rules permit the use of padding (skipped bytes) to align scalars on desirable boundaries. The structure mapping examples show each scalar aligned at its natural boundary. This alignment introduces padding of four bytes between *a* and *b*, one byte between *d* and *e*, and two bytes between *e* and *f*. The same amount of padding is present in both big endian and little endian mappings.

2.4.1.1 Big-Endian Mapping

The big endian mapping of structure *s* follows. (The data is highlighted in the structure mappings. Addresses, in hexadecimal, are below the data stored at the address. The contents of each byte, as defined in structure *s*, is shown as a (hexadecimal) number or character (for the string elements)).

11 0x00	12 0x01	13 0x02	14 0x03	0x04	0x05	0x06	0x07
21 0x08	22 0x09	23 0x0A	24 0x0B	25 0x0C	26 0x0D	27 0x0E	28 0x0F
31 0x10	32 0x11	33 0x12	34 0x13	'A' 0x14	'B' 0x15	'C' 0x16	'D' 0x17
'E' 0x18	'F' 0x19	'G' 0x1A	0x1B	51 0x1C	52 0x1D	0x1E	0x1F
61 0x20	62 0x21	63 0x22	64 0x23	0x24	0x25	0x26	0x27

2.4.1.2 Little Endian Mapping

Structure *s* is shown mapped little endian.

14 0x00	13 0x01	12 0x02	11 0x03	0x04	0x05	0x06	0x07
28 0x08	27 0x09	26 0x0A	25 0x0B	24 0x0C	23 0x0D	22 0x0E	21 0x0F
34 0x10	33 0x11	32 0x12	31 0x13	'A' 0x14	'B' 0x15	'C' 0x16	'D' 0x17
'E' 0x18	'F' 0x19	'G' 0x1A	0x1B	52 0x1C	51 0x1D	0x1E	0x1F
64 0x20	63 0x21	62 0x22	61 0x23	0x24	0x25	0x26	0x27

2.4.2 PowerPC Byte Ordering

By default, the PowerPC Architecture is big endian. This book describes the processor as if it operated only in a big endian fashion. In fact, the PowerPC Architecture and the IBM PowerPC Embedded Environment support little endian operation as well.

Two independent mechanisms support little endian operation. The first, defined by the PowerPC Architecture, provides endian mode control using bits in the MSR. The second is an endian storage attribute. It is defined by the IBM PowerPC Embedded Environment, and is not part of the PowerPC Architecture. Subsequent sections explain both mechanisms in more detail. For more information, see *The PowerPC Architecture: A Specification for a New Family of RISC Processors* and *The IBM PowerPC Embedded Environment*.

2.4.3 PowerPC Endian Mode

PowerPC endian mode is useful for system environments in which some processes and their associated data structures are written as little endian, and other processes are written as big endian. The PowerPC endian mode mechanism handles such bi-endian systems and manages communications and data sharing between processes running in the system. However, because of how PowerPC endian mode operates, it does not provide for direct processor connections to little endian hardware, nor for operating the PPC401A1 in a hardware system environment that is connected in a little endian manner. Instead, for such environments, one should use the endian (E) storage attribute described in Section 2.4.4, “Endian Storage Attribute,” on p. 2-25.

When the PPC401A1 operates with the PowerPC endian mode set to little endian, instructions and data in memory *appear*, from the programmer's point of view, to be arranged in little endian format. However, instructions and data in memory are arranged in a unique order that is neither big endian nor little endian. In addition, the processor manipulates the low-order address bits used for all instruction fetches and data references

such that, when combined with the unique ordering of the bytes in memory, the instructions and data appear to the executing program to be arranged in true little endian order. Section 2.4.3.1 describes this unique byte arrangement and the address manipulation in detail, while Section 2.4.3.2 explains how the PowerPC endian mode is controlled.

2.4.3.1 Byte Ordering in PowerPC Little Endian Mode

When the processor operates in PowerPC little endian mode, bytes, *in memory*, are rearranged from the order in which they would appear in a true little endian environment. Specifically, for each aligned doubleword (eight bytes) of memory, the eight bytes are reversed across the doubleword. For example, for the aligned doubleword at addresses A0–A7, the byte at A0 in little endian format is instead placed at A7 for PowerPC little endian mode. Likewise, the byte from A1 is moved to A6, A2 to A5, A3 to A4, A4 to A3, A5 to A2, A6 to A1, and A7 to A0. This is repeated for the next doubleword at addresses A8–A15, and so on.

Structure *s* (defined in Section 2.4.1, “Structure Mapping Examples,” on p. 2-18) would appear *in memory* as follows after being rearranged as described.

				11	12	13	14
0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
21	22	23	24	25	26	27	28
0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
'D'	'C'	'B'	'A'	31	32	33	34
0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
	51	52			'G'	'F'	'E'
0x18	0x19	0x1A	0x1B	0x1C	0x1D	0x1E	0x1F
				61	62	63	64
0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27

Note that this arrangement of bytes is neither big endian nor little endian, but is rather the result of taking the bytes from the little endian mapping and “swapping” them byte-for-byte across each doubleword. For this unique arrangement of bytes to appear to the executing program as equivalent to a true little endian arrangement, the address of each storage reference (whether for instruction fetches or for data accesses from load and store instructions) must be modified.

Specifically, the address of each storage access is modified by exclusive-ORing the low-order three bits of the address (addresses for instruction fetches are modified as for word accesses, because all PowerPC instructions are words).

Access Type	Address Modification
Byte	XOR with 0b111
Halfword	XOR with 0b110
Word	XOR with 0b100

To see how this address modification, combined with the unique ordering of bytes in memory, results in the appearance to the executing program of a true little endian byte arrangement, consider the following example, using the value of the word *a* from structure *s*. If *a* were stored, in little endian format, to address 00, it would appear as follows:

14	13	12	11
0x00	0x01	0x02	0x03

This memory could be accessed using word, halfword, or byte accesses in a true little endian system with the following results:

Word load from address 00	0x1112_1314
Halfword load from address 00	0x1314
Halfword load from address 02	0x1112
Byte load from address 00	0x14
Byte load from address 01	0x13
Byte load from address 02	0x12
Byte load from address 03	0x11

For programmers to view memory in PowerPC little endian mode as equivalent to memory in a true little endian system, the values observed for each kind of access must match those shown.

The following example shows how *a* is arranged in memory when stored, in PowerPC little endian mode, to address 0:

11	12	13	14
0x04	0x05	0x06	0x07

This word could then be accessed using word, halfword, or byte accesses in PowerPC little endian mode with the following results:

Word load from (processor) address 00	converts to (memory) address 04	0x1112_1314
Halfword load from (processor) address 00	converts to (memory) address 06	0x1314
Halfword load from (processor) address 02	converts to (memory) address 04	0x1112
Byte load from (processor) address 00	converts to (memory) address 07	0x14
Byte load from (processor) address 01	converts to (memory) address 06	0x13
Byte load from (processor) address 02	converts to (memory) address 05	0x12
Byte load from (processor) address 03	converts to (memory) address 04	0x11

This example shows that a program, running on a PPC401A1 operating in PowerPC little endian mode, views word *a* in memory as if it were instead arranged in true little endian format. Similar results are obtained for the other members of structure *s*.

It should be recognized that because *instructions* in PowerPC Architecture are defined as aligned words, their addressing is also affected by endian mode. Specifically, each pair of words in an aligned doubleword of memory are reversed with respect to each other when operating in PowerPC little endian mode. So, even though both a big endian and a little endian program may have a sequence of instructions at, say, addresses 00, 04, 08, 12, and so on, and the executing program will request these instructions in order, the address modification causes the little endian program executing in PowerPC little endian mode to receive the instructions *from memory* in the following order of addresses: 04, 00, 12, 08, and so on.

Care must be taken when loading little endian programs into memory to ensure that the instructions are arranged in the proper order. See Section 2.4.3.5, “Switching Endian Modes,” on p. 2-24, for more detailed information.

2.4.3.2 Control of PowerPC Endian Mode

The selection of the PowerPC endian mode is controlled by two bits in the MSR: the little endian mode bit (MSR[LE]) and the Interrupt little endian bit (MSR[ILE]).

MSR[LE] describes the current endian mode. If MSR[LE] = 1, the processor is executing in PowerPC little endian mode. Otherwise, the processor executes in big endian mode.

When the PPC401A1 takes an interrupt, the MSR contents are saved in either Save/Restore Register 1 (SRR1) or Save/Restore Register 3 (SRR3), depending on the interrupt type. The content of MSR[ILE] replaces the content of MSR[LE]. The PPC401A1 can switch endian modes in this fashion when entering an interrupt handler. The original value of MSR[LE] is restored from SRR1 or SRR3 upon leaving the interrupt handler (using an **rfi** or **rftci** instruction as appropriate) and returning to the previously executing program. Hence, the PPC401A1 can also switch endian modes when leaving an interrupt handler. This mode-switching capability enables an operating system written in one endian mode to support application programs written in the other mode.

The PPC401A1 resets to big endian mode, $\text{MSR}[\text{LE}] = 0$ and $\text{MSR}[\text{ILE}] = 0$.

2.4.3.3 Addressing in PowerPC Little Endian Mode

The address modification performed in PowerPC little endian mode affects only those addresses that are presented to the storage subsystem (including the caches). Specifically, it does *not* affect the original calculation of addresses, nor the value of addresses saved in registers as part of the semantics of instruction execution.

For example, the following address values are calculated independently of endian mode, and are stored in the appropriate registers without modification:

- The address placed into the LR by a branch with link update instruction, which is equal to the Program Counter (PC) + 4
- The offset in a relative branch instruction, which reflects the difference between the addresses of the branch and target instructions as they appear to the executing program (*not necessarily* as they appear in the actual memory arrangement)
- The address placed into RA by a load/store with update instruction, which is the value computed as described in the instruction description
- The address saved in system registers, such as SRR0, SRR2, and the DEAR, as computed by the executing program and as defined for these registers

These examples do not include all addresses that are not affected by the little endian address modification.

The cache management instructions (**dcbi**, **icbi**, and others) are unaffected by endian mode, because the addresses used by these instructions refer to an entire cache block (16 bytes) and the low-order four bits of the address are not used.

2.4.3.4 Little Endian Mode Alignment Requirements

The “trick” of Exclusive OR-ing the low-order three bits of the address of an individual scalar does not work unless the scalar is aligned in memory to the size of the scalar. To illustrate, consider the following example of a word w (containing 0x1112_1314) stored in memory at address 05, and arranged in little endian format:

					14	13	12
0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
11							
0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F

In PowerPC little endian mode, word *w* would be arranged in memory as follows (remember that the bytes in each aligned doubleword are reversed in the format used by PowerPC little endian mode):

12 0x00	13 0x01	14 0x02	0x03	0x04	0x05	0x06	0x07
0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	11 0x0F

Note that the unaligned word *w* spans two doublewords. The two parts of the unaligned word are not contiguous in memory. Applying the address modification to a load word at address 0x05 results in address 0x01; the load word from address 0x05 causes the four bytes at addresses 0x01, 0x02, 0x03, and 0x04 to be accessed—clearly an incorrect result. Because of the complexity of dealing with this unusual arrangement of unaligned scalars when operating in PowerPC little endian mode, the PPC401A1 generates alignment exceptions when attempting to execute any of the following instruction types, if the PPC401A1 is in PowerPC little endian mode:

- Unaligned halfword or word load/store instruction
- String or multiple instruction (**lmw**, **lswi**, **lswx**, **stmw**, **stswi**, **stswx**)

Note that although there are other conditions that can result in alignment exceptions, the alignment exceptions caused by those conditions occur regardless of endian mode. See Section 5.8, “Alignment Exception,” on p. 5-17, for more information.

2.4.3.5 Switching Endian Modes

Because bytes in memory are arranged differently when operating in the different endian modes, care must be taken, when switching modes, to convert programs and data structures to the new mode. The operating system must understand the differences in the two memory formats, and must reorder the bytes in memory, as appropriate, before dispatching a new process that accesses these memory structures in the new endian mode.

For example, if a process executing in big endian mode creates a data structure, and a new process executing in little endian mode will access this data structure, the operating system must reverse the eight bytes within each aligned doubleword in the data structure before passing control to the new process.

2.4.3.6 Direct Memory Access in PowerPC Little Endian Mode

Another aspect of the unique arrangement of bytes used by PowerPC little endian mode that must be considered is that of access to memory by devices other than the PPC401A1. Because these other devices, such as a direct memory access (DMA) device or another non-PowerPC processor, are not likely to handle the special address modifications associated with PowerPC little endian mode, they must be aware of the special arrangement of bytes used by the PPC401A1 when it operates in little endian mode.

For example, if an I/O device loads a little endian program and data structure from a disk and places it into memory so that the PPC401A1 can execute the program in little endian mode, the I/O device must reverse the eight bytes in each aligned doubleword after reading the data from the disk and before writing it to memory. Alternatively, the operating system running on the PPC401A1 must understand that the program image loaded from the disk was placed into memory in true little endian format, in which case the operating system must rearrange the bytes before executing the program.

2.4.4 Endian Storage Attribute

The endian storage attribute (E bit), defined in the IBM PowerPC Embedded Environment, also supports using the PPC401A1 in a little endian system. For every storage reference (instruction fetch or load/store access), an E bit is associated with the address region of the storage reference. The E bit specifies whether that region is organized as big endian (E = 0) or little endian (E = 1).

Unlike the organization of memory when using the PowerPC little endian mode, bytes in storage regions that are programmed as little endian using the E bit are arranged in true little endian format. Furthermore, no address modification is performed when accessing storage regions programmed as E = 1. Instead, when accessing storage regions with E = 1, the PPC401A1 reorders the bytes as they are transferred between the processor and memory. Unlike PowerPC little endian mode, the E storage attribute supports direct connections to little endian hardware and to memory containing little endian programs and data structures that may be shared with other little endian devices.

The on-the-fly reversal of bytes accessed in little endian storage regions is handled in one of two ways, depending on whether the storage access is an instruction fetch or a data (load/store) access. The following sections describe byte reversal for the two kinds of storage accesses.

2.4.4.1 Fetching Instructions from Little Endian Storage Regions

The PowerPC Architecture defines instructions as aligned words (four bytes) in memory. As such, instructions in a big endian program image are arranged with the most significant byte (MSB) of the instruction word at the lowest numbered address.

Consider the big endian mapping of instruction p at address 00, where, for example, $p = \text{add } r7, r7, r4$:

MSB			LSB
0x00	0x01	0x02	0x03

On the other hand, in a little endian program the same instruction is arranged with the least significant byte (LSB) of the instruction word at the lowest numbered address:

LSB			MSB
0x00	0x01	0x02	0x03

When an instruction is fetched from memory, the instruction must be placed in the pipeline in the proper order. Otherwise, the instruction decoder cannot recognize it. Because the PowerPC Architecture, by default, is big endian, the MSB of an instruction word is assumed to be at the lowest address. Therefore, when instructions are fetched from little endian storage regions, the four bytes of an instruction word must be reversed before the instruction is decoded. In the PPC401A1, the byte reversal occurs between memory and the ICU. The ICU always contains instructions in big endian format, regardless of whether the storage region containing the instruction was programmed as big endian or little endian. Thus, the bytes are already in the proper order when an instruction is transferred from the ICU to the decode stage of the pipeline.

If a storage region is reprogrammed from one endian format to the other, the contents of the storage region must be reloaded with program and data structures in the appropriate endian format. If the contents of instruction memory change, the ICU must be made coherent with the updates. The ICU must be invalidated and the updated memory contents must be fetched in the new endian format so that the proper byte reversal (or for big endian, no byte reversal) occurs before the new instructions are placed in the ICU.

2.4.4.2 Accessing Data in Little Endian Storage Regions

Unlike instruction fetches from little endian storage regions, data accesses from little endian storage regions are *not* byte-reversed between memory and the DCU. Data byte ordering, in memory, depends on the data type (byte, halfword, or word) of a specific data item. It is only when moving a data item *of a specific type* from or to a GPR that it becomes known whether byte reversal is required due to the endian format of the data item. Therefore, byte reversal during load/store accesses is performed between the DCU and the GPR file, depending on whether the load/store was for a byte, halfword, or word.

Referring to the big endian and little endian mappings of structure *s*, as shown in Section 2.4.1, “Structure Mapping Examples,” on p. 2-18, the differences between the byte locations of any data item in the structure depends upon the size of the particular data item. For example (again referring to the big endian and little endian mappings of structure *s*):

- The word *a* has its four bytes reversed within the word spanning addresses 00–03.
- The halfword *e* has its two bytes reversed within the halfword spanning addresses 1C–1D.

Note that the array of bytes *d*, where each data item is a byte, is not reversed when the big endian and little endian mappings are compared. For example, the character 'A' is located at address 14 in both the big endian and little endian mappings.

The size of the data item being loaded or stored must be known before the processor can decide whether, and if so, how to reorder the bytes when moving them between a GPR and storage.

When accessing data in a little endian storage region:

- For byte loads/stores, no reordering of bytes occurs.
- For halfword loads/stores, bytes are reversed within the halfword.
- For word loads/stores, bytes are reversed within the word.

Note that this mechanism applies, regardless of the alignment of data.

For example, when loading a data word from a little endian storage region, all four bytes of the word are retrieved from memory (or the DCU). Then, the bytes are placed in the GPR so that the byte from the lowest address is placed in the LSB of the GPR.

In little endian storage regions, the alignment of data is treated as it is in big endian storage regions. Unlike PowerPC little endian mode, no special alignment exceptions occur when accessing data in little endian storage regions. Note that the alignment exceptions that apply to big endian region accesses also apply to little endian storage region accesses. See Section 5.8, “Alignment Exception,” on p. 5-17, for detailed descriptions of conditions causing alignment exceptions.

2.4.4.3 Control of the Endian Storage Attribute

The endian (E) storage attribute, for a given access, is controlled by the Storage Little-Endian Register (SLER), which is a storage attribute control register similar to those controlling the other storage attributes.

The SLER is a 32-bit register that provides the E storage attribute for each 128MB storage attribute control region in the 4GB address space. The high-order five bits of the storage address select, from the SLER, the E storage attribute associated with the address region. Setting a bit to 1 in the SLER specifies that the associated storage region is little endian. See Chapter 8, “Storage Control,” for more information about the storage attribute control registers.

2.4.4.4 PowerPC Byte-Reverse Instructions

The PowerPC Architecture defines byte-reverse load/store instructions, which can perform a function similar to the action taken automatically by the PPC401A1 when it accesses data in little endian storage regions using the normal load/store instructions. However, the byte-reverse load/store instructions are not as generally useful as the endian storage attribute mechanism.

For big endian storage regions, the normal (non-byte-reverse) load/store instructions operate as defined in the instruction descriptions, moving the more significant bytes of the register to and from the lower-numbered memory addresses. The load/store with byte-

reverse instructions move the more significant bytes of the register to and from the higher numbered memory addresses.

The opposite is true for little endian storage regions, where the normal load/store instructions give the same results that load/store with byte-reverse instructions do in big endian storage regions. Load/store with byte-reverse instructions give the same results that normal load/store instructions do in big endian storage regions.

As Figures 2-10 through 2-13 illustrate, a normal store to a big endian storage region is the same as a byte-reverse store to a little endian storage region, while a normal store to a little endian storage region is the same as a byte-reverse store to a big endian storage region.

Figure 2-10 illustrates the contents of a GPR and memory (starting at address 00) after a normal load/store in a big endian storage region.

MSB		LSB		
11	12	13	14	GPR
11	12	13	14	Memory
0x00	0x01	0x02	0x03	

Figure 2-10. Normal Word Load or Store (Big Endian Storage Region)

Note that the results are identical to the results of a load/store with byte-reverse in a little endian storage region, as illustrated in Figure 2-11.

MSB		LSB		
11	12	13	14	GPR
11	12	13	14	Memory
0x00	0x01	0x02	0x03	

Figure 2-11. Byte-reverse Word Load or Store (Little Endian Storage Region)

Figure 2-12 illustrates the contents of a GPR and memory (starting at address 00) after a load/store with byte-reverse in a big endian storage region.

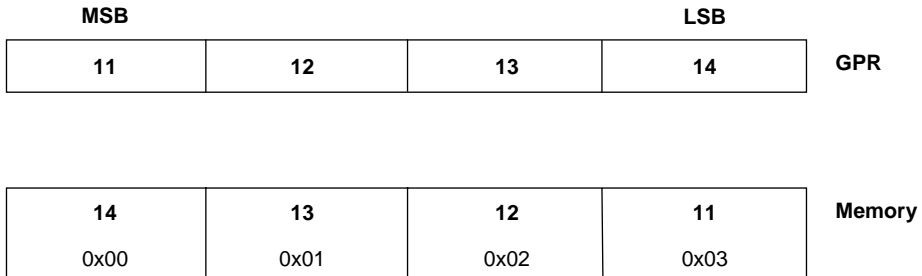


Figure 2-12. Byte-reverse Word Load or Store (Big Endian Storage Region)

Note that the results are identical to the results of a normal load/store in a little endian storage region, as illustrated in Figure 2-13.

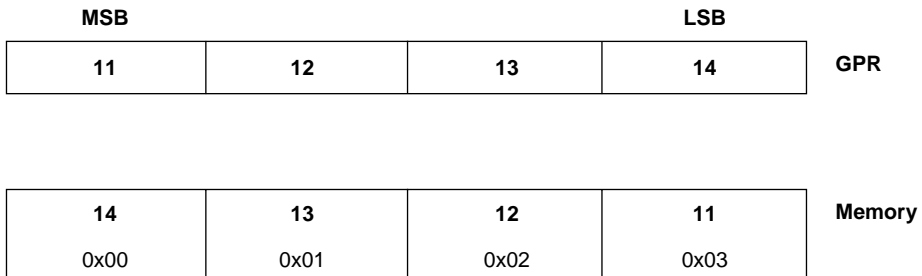


Figure 2-13. Normal Word Load or Store (Little Endian Storage Region)

The E storage attribute augments the byte-reverse load/store instructions in two important ways:

- The load/store with byte-reverse instructions do not solve the problem of fetching instructions from a program image in true little endian format.

Only the endian storage attribute mechanism supports the fetching of true little endian program images.

- Typical compilers cannot make general use of the byte-reverse load/store instructions, so these instructions are ordinarily used only in special, hand-coded device drivers.

Compilers can, however, take full advantage of the endian storage attribute mechanism, enabling application programmers working in a high-level language, such as C, to compile programs and data structures into little endian format.

2.5 Instruction Processing

The instruction queue, illustrated in Figure 2-14, contains three queue locations: pre-fetch buffer 1 (PFB1), PFB0, and decode (DCD). This queue implements a pipeline with the following functional stages: fetch, decode, and execute. Instructions are fetched from the instruction cache unit (ICU) and dispatched to the execution unit (EXU).

Instructions are fetched, at the request of the EXU, from the ICU. Cacheable instructions are forwarded directly to the instruction queue and stored in the cache. Non-cacheable instructions are also forwarded directly to the instruction queue, but are not stored in the cache. Fetched instructions drop to the empty queue location closest to the EXU. If the queue is empty, an entering instruction drops directly to DCD. PFB0 and PFB1 simply buffer instructions when the pipeline stalls.

Instructions are decoded entirely in DCD. Branches are predicted and determined during decoding. After decoding (and determination, for branch instructions), the instruction is dispatched to the execution unit (EXU), where it is executed.

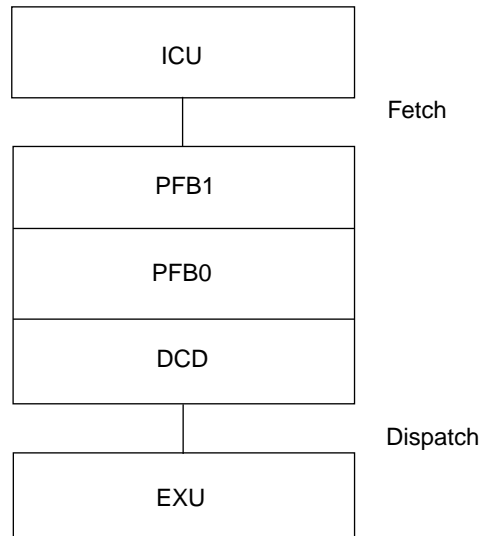


Figure 2-14. PPC401A1 Instruction Queue

2.6 Branching Control

The PPC401A1, which provides a variety of conditional and unconditional branching instructions, uses the branch prediction techniques described in Section 2.6.5, “Branch Prediction,” on p. 2-33.

2.6.1 AA Field on Unconditional Branches

The unconditional branches (**b**, **ba**, **bl**, **bla**) carry the displacement to the branch target address as a 26-bit value (the 24-bit LI field right-extended with two zeroes). This displacement is regarded as a signed 26-bit number covering an address range of $\pm 32\text{MB}$.

For the relative (AA = 0) forms (**b**, **bl**), the target address is the Current Instruction Address (CIA, the address of the branch instruction) plus the signed displacement.

For the absolute (AA = 1) forms (**ba**, **bla**), the target address is zero plus the signed displacement. If the sign bit (LI[0]) is zero, the displacement is the target address. If the sign bit is one, the address is “below zero” and wraps to high memory. For example, if the displacement is 0x3FF FFFC (the 26-bit representation of negative four), the target address is 0xFFFF FFFC (zero minus four bytes, or four bytes from the top of memory).

2.6.2 AA Field on Conditional Branches

The conditional branches (**bc**, **bca**, **bcl**, **bcla**) carry the displacement to the branch target address as a 16-bit value (the 14-bit BD field right-extended with two zeroes). This displacement is regarded as a signed 16-bit number, covering an address range of $\pm 32\text{KB}$.

For the relative (AA = 0) forms (**bc**, **bcl**), the target address is the current instruction address (CIA, the address of the branch instruction) plus the signed displacement.

For the absolute (AA = 1) forms (**bca**, **bcla**), the target address is zero plus the signed displacement. If the sign bit (BD[0]) is zero, the displacement is the target address. If the sign bit is one, the address is “below zero” and wraps to high memory. For example, if the displacement is 0xFFFFC (the 16-bit representation of negative four), the target address is 0xFFFF FFFC (zero minus four bytes, or four bytes from the top of memory).

2.6.3 BI Field on Conditional Branches

Conditional branch instructions can test one bit of the Condition Register (CR). The value of the BI field specifies the bit to be tested (bit 0–31). The content of the BI field is meaningless unless BO[0] = 0.

2.6.4 BO Field on Conditional Branches

The BO field specifies the condition under which a branch is taken, and how the branch affects the CTR.

Conditional branch instructions can test one bit in the CR. This option is selected when BO[0] = 0; if BO[0] = 1, the CR does not participate in the branch condition test. If this option is selected, the condition is satisfied (branch can occur) if CR[BI] = BO[1].

Conditional branch instructions can decrement the Count Register (CTR) by one, and after the decrement, test the CTR value. This option is selected when BO[2] = 0. If this option is selected, BO[3] specifies the condition that must be satisfied to allow a branch to be taken. If

BO[3] = 0, CTR \neq 0 is required for a branch to occur. If BO[3] = 1, CTR = 0 is required for a branch to occur.

If BO[2] = 1, the contents of CTR are left unchanged, and the CTR does not participate in the branch condition test.

Table 2-4 summarizes the usage of the bits of the BO field. BO[4] is further discussed in Section 2.6.5.

Table 2-4. Bits of the BO Field

BO Bit	Description
BO[0]	CR Test Control 0 Test CR bit specified by BI field for value specified by BO[1] 1 Do not test CR
BO[1]	CR Test Value 0 If BO[0] = 0, test for CR[BI] = 0. 1 If BO[0] = 0, test for CR[BI] = 1.
BO[2]	CTR Test Control 0 Decrement CTR by one and test whether CTR satisfies the condition specified by BO[3]. 1 Do not change CTR, do not test CTR.
BO[3]	CTR Test Value 0 If BO[2] = 0, test for CTR \neq 0. 1 If BO[2] = 0, test for CTR = 0.
BO[4]	Branch Prediction Reversal 0 Apply standard branch prediction. 1 Reverse the standard branch prediction.

Table 2-5 lists specific BO field contents, and the resulting actions. In Table 2-5, z represents a mandatory value of zero, and y is a branch prediction option discussed in Section 2.6.5.

Table 2-5. Conditional Branch BO Field

BO Value	Description
0000y	Decrement the CTR, then branch if the decremented CTR \neq 0 and CR[BI]=0.
0001y	Decrement the CTR, then branch if the decremented CTR = 0 and CR[BI] = 0.
001zy	Branch if CR[BI] = 0.
0100y	Decrement the CTR, then branch if the decremented CTR \neq 0 and CR[BI] = 1.
0101y	Decrement the CTR, then branch if the decremented CTR=0 and CR[BI] = 1.
011zy	Branch if CR[BI] = 1.
1z00y	Decrement the CTR, then branch if the decremented CTR \neq 0.
1z01y	Decrement the CTR, then branch if the decremented CTR = 0.
1z1zz	Branch always.

2.6.5 Branch Prediction

Conditional branches present a problem to the fetcher. A branch might be taken; if not taken, the branch simply falls through to the next sequential instruction. The PPC401A1 attempts to predict whether or not a branch is taken before all information necessary to determine the branch direction is available. This decision is called a *branch prediction*. The fetcher can then pre-fetch instructions down the predicted path. If the prediction is correct, time is saved because the branched-to instruction is available in the instruction queue. Otherwise, time is lost while the correct instruction is fetched into the instruction queue. To be effective, branch prediction must be correct most of the time.

The PPC401A1 uses PowerPC branch prediction to minimize incorrect predictions, and enables software to reverse the standard branch prediction, which is defined as follows:

Predict that the branch is to be taken if $((BO[0] \wedge BO[2]) \vee s) = 1$

where s is bit 16 of the instruction (the sign bit of the displacement for all **bc** forms, and zero for all **bclr** and **bcctr** forms).

$(BO[0] \wedge BO[2]) = 1$ only when the conditional branch tests nothing (the “branch always” condition). Obviously, the branch should be predicted taken for this case.

If the branch tests anything, $(BO[0] \wedge BO[2]) = 0$, and s entirely controls the prediction. The standard prediction for this case derives from considering the relative form of **bc**, often used at the end of loops to control the number of times that a loop is executed. The branch is taken each time the loop is executed except the last, so it is best if the branch is predicted taken. The branch target is the beginning of the loop, so the branch displacement is negative and $s = 1$. Because this situation is so common, a branch is taken if $s = 1$.

If branch displacements are positive, $s = 0$, and the branch is predicted not taken. If the branch instruction is any form of **bclr** or **bcctr** except the “branch always” forms, then $s = 0$, and the branch is predicted not taken.

There is a peculiar consequence of this prediction algorithm for the absolute forms of **bc** (**bca** and **bcla**). As described in Section 2.6.2, if $s = 1$, the branch target is in high memory. If $s = 0$, the branch target is in low memory. Because these are absolute-addressing forms, there is no reason to treat high and low memory differently. Nevertheless, for the high memory case the standard prediction is taken, and for the low memory case the standard prediction is not taken.

$BO[4]$ is the *prediction reversal bit*. If $BO[4] = 0$, the standard prediction is applied. If $BO[4] = 1$, the reverse of the standard prediction is applied. For the cases in Table 2-5 where $BO[4] = y$, software can reverse the standard prediction. This should only be done when the standard prediction is likely to be wrong. Note that for the “branch always” condition, reversal of the standard prediction is not allowed.

The PowerPC Architecture requires assemblers to provide a way to conveniently control branch prediction. For any conditional branch mnemonic, a suffix may be added to the mnemonic to control prediction, as follows:

- + Predict branch to be taken
- Predict branch to be not taken

For example, **bcctr+** causes BO[4] to be selected appropriately to force the branch to be predicted taken.

2.7 Speculative Accesses

The PowerPC Architecture permits implementations to perform speculative accesses to memory, either for instruction fetching, or for data loads. A speculative access is defined as any access which is not required by a sequential execution model.

For example, pre-fetching instructions beyond an undetermined conditional branch is a speculative fetch; if the branch is not in the predicted direction, the program, as executed, never needs the instructions from the predicted path. Similarly, in a superscalar processor that performs out-of-order execution, a program can speculatively fetch a load instruction that is past an undetermined branch.

Sometimes speculative accesses are inappropriate, however. For example, attempting to fetch instructions from addresses that cannot contain instructions can cause problems. To protect against errant accesses to “sensitive” memory or I/O devices, the PowerPC Architecture provides the G (guarded) storage attribute, which can be used to specify memory regions from which speculative accesses are prohibited. (Actually, speculative accesses to guarded storage are allowed in certain limited circumstances; if an instruction in a cache block will be executed, the rest of the cache block can be speculatively accessed.)

2.7.1 Speculative Accesses in the PPC401A1

The PPC401A1 does not perform out-of-order execution, nor does the PPC401A1 perform speculative loads.

The PPC401A1 provides the Storage Guarded Register (SGR) to enable and disable speculative instruction fetching in each of thirty-two 128MB regions of memory. When a region is guarded (speculative fetching is disallowed), pre-fetching is disabled for that region. A fetch request must be completely resolved (no longer speculative) before it is issued. There is a considerable performance penalty for fetching from guarded storage, so guarding should be used only when required.

Note that, following any reset, the PPC401A1 operates with all of storage guarded.

2.7.1.1 Pre-fetch Distance Down an Unresolved Branch Path

The fetcher will speculatively access up to five instructions down a predicted branch path, whether taken or sequential. The unresolved branch is in the DCD stage of the instruction

queue (see Section 2.5 for a description of the instruction queue). If PFB0 and PFB1 are full, no further speculative accesses occur. If PFB0 or PFB1 is empty, the fetcher requests the next speculative instruction from the ICU; that instruction is placed in PFB0 or PFB1. If the fetched instruction is at the end of a cache line, and if PFB1 is empty, the fetcher requests the next cache line. The instruction at the beginning of the cache line is placed in PFB1. In this case, five instructions are speculatively accessed. The fetcher can speculatively access no more than four instructions (a cache line) from the cache with a single request, assuming the speculative address is cacheable.

If the address is non-cacheable (as controlled by the Instruction Cache Cacheability Register (ICCR)), no more than two instructions are speculatively accessed.

2.7.1.2 Pre-fetch of Branches to Count Register and Branches to Link Register

When the fetcher predicts that a **bctr** or **blr** instruction is taken, it will not attempt to access the target address in the Count Register (CTR) or Link Register (LR) if an executing instruction updates the CTR or LR ahead of the branch in DCD in the instruction queue. (See Section 2.5 for description of the instruction queue). The fetcher recognizes that the CTR or LR contains data left from an earlier use of the CTR or LR. Such data is probably not valid.

In such cases, the fetcher does not fetch the instruction at the target address until the instruction updating the CTR or LR completes and EXU is empty; only then are the “correct” CTR or LR contents known. This prevents the fetcher from speculatively accessing a completely “random” address. When the CTR or LR contents are known to be correct, the fetcher will access no more than five instructions down the sequential or taken path of an unresolved branch, or at the address contained in the CTR or LR.

2.7.2 Preventing Inappropriate Speculative Accesses

A memory-mapped I/O device that has a status register that is automatically reset when read provides a simple example of storage that should not be speculatively accessed. Consider a serial port that reads the receive buffer on the port and then resets the RxRdy bit in the status register. If the processor speculatively loads from this register, and an intervening branch or interrupt takes the program flow away from the code containing the load instruction and then returns, the wrong result will be obtained when the status register is read again.

Similarly, if the program code is in memory “next to” the I/O device (for example, code goes from 0x0000 0000 to 0x0000 0FFF, and the I/O device is at 0x0000 1000), pre-fetching past the end of the code can “hit” the I/O device.

Guarded storage can prevent pre-fetching past the “end” of memory. The fetcher will attempt to fetch past the last valid address, likely getting machine checks on the fetches to invalid addresses. While the machine checks do not result in an exception until the processor attempts to execute an instruction at an invalid address, some systems may suffer from the

attempt to access such an invalid address. For example, an external memory controller might log the error.

System designers can avoid problems from speculative fetching in other ways, without using the guarded storage attributes. The rest of this section describes ways to guard against speculative instruction fetches to sensitive addresses in unguarded memory regions.

2.7.2.1 Fetching Past an Interrupt-causing or Interrupt-returning Instruction

Suppose a **bctr** or **blr** instruction follows an interrupt-causing or interrupt-returning instruction (**sc**, **rfi**, or **rfti**). The fetcher does not prevent speculatively fetching past one of these instructions. In other words, the fetcher does not treat the interrupt-causing and interrupt-returning instructions specially when deciding whether to predict down a branch path. Instructions after an **rfti**, for example, are considered to be on the determined branch path.

To understand the implications of this situation, consider the code sequence:

```

handler:      aaa
              bbb
              rfi
subroutine:   bctr

```

When executing the interrupt handler, the fetcher does not recognize the **rfti** as a break in the program flow, and speculatively fetches the target of the **bctr**, which is really the first instruction of a subroutine that has not been called. Therefore, the CTR might contain an invalid pointer.

To protect against such a pre-fetch, the software should insert an unconditional branch hang (**b \$**) just after the **rfti**. This prevents the hardware from pre-fetching the wrong “target” of the **bctr**.

Consider also the above code sequence, with the **rfti** instruction replaced by an **sc** instruction. The purpose of the system call is to initialize the CTR with the appropriate value for the **bctr** to branch to, upon return from the system call. The **sc** handler returns to the instruction following the **sc**, which can’t be a branch hang. Instead, software could put a **mtctr** just before the **sc** to load a nonsensitive address into the CTR. This address will be used as the prediction address before the **sc** executes. An alternative would be to put a **mfctr** or **mtctr** between the **sc** and the **bctr**; the **mtctr** prevents the fetcher from speculatively accessing the address contained in the CTR before initialization.

2.7.2.2 Fetching Past **tw** or **twi** Instructions

The interrupt-causing instructions, **tw** and **twi**, do not require the special handling described in Section 2.7.2.1. These instructions are typically used by debuggers, which implement software breakpoints by substituting a trap instruction for the instruction originally at the breakpoint address. In a code sequence **mtlrr** followed by **blr** (or **mtctr** followed by **bctr**), replacement of **mtlrr/mtctr** by **tw** or **twi** leaves the LR/CTR uninitialized. It would be

inappropriate to fetch from the **blr/bctr** target address. This situation is common, and the fetcher is designed to prevent the problem.

2.7.2.3 Fetching Past an Unconditional Branch

When an unconditional branch is in DCD in the instruction queue, the fetcher recognizes that the sequential instructions following the branch are unnecessary. These sequential addresses are not accessed. Addresses at the branch target are accessed instead.

Therefore, placing an unconditional branch just before the start of a sensitive address space (for example, at the “end” of a memory area that borders an I/O device) guarantees that addresses in the sensitive area will not be speculatively fetched.

2.7.2.4 Suggested Locations of Memory-Mapped Hardware

Table 2-6 shows two address regions of the PPC401A1. Suppose a system designer can map all I/O devices and all ROM and SRAM devices, for example, anywhere into either region. The choices made by the designer can prevent speculative accesses to the memory-mapped I/O devices.

Table 2-6. Example Memory Mapping

0x7800 0000 – 0x7FFF FFFF (SGR bit 15)	128MB Region 2
0x7000 0000 – 0x77FF FFFF (SGR bit 14)	128MB Region 1

A simple way to avoid the problem of cacheable instruction fetches colliding with I/O devices meant to be non-cacheable would be to map all ROM and SRAM devices into Region 2, and all I/O devices into Region 1.

Thus, addresses in Region 1 should be accessed only by non-cacheable load/store instructions accessing I/O devices; no speculative fetches should occur. Region 1 could be set as Guarded in the SGR; no performance penalty would result, since by design there is no possibility of pre-fetching from Region 1. Accesses to Region 2 would be for code and program data. Speculative fetches in Region 2 can never access addresses in Region 1. Note that this hardware organization makes the use of the SGR to protect Region 1 redundant and optional.

The use of these regions could be reversed (code in Region 1 and I/O devices in Region 2), if Region 2 is set as Guarded in the SGR. Pre-fetching from the top of Region 1 could attempt to speculatively access the bottom of Region 2, but Guarding will prevent a speculative access from occurring. The performance penalty is slight, under the assumption that code infrequently executes near the top of Region 1.

2.7.3 Summary

In summary, software should take the following actions to prevent speculative accesses to sensitive data areas, if the sensitive data areas are not in guarded storage:

- Protect against accesses to “random” values in the LR or CTR on **blr** or **bctr** branches following **rfi**, **rfdi**, or **sc** instructions by putting appropriate instructions before or after the **rfi**, **rfdi**, or **sc** instruction. See Section 2.7.2.1.
- Protect against “running past” the end of memory into a bordering I/O device by putting an unconditional branch at the end of the memory area. See Section 2.7.2.3.
- Recognize that a maximum of five words (20 bytes) can be prefetched past an unresolved conditional branch, either down the target path or the sequential path. See 2.7.1.1 above.
- Of course, software should not code branches with known unsafe targets (either instruction counter-relative or LR- or CTR-based), on the assumption that they are “protected” by guaranteeing that the unsafe direction is “not-taken”. The pre-fetcher can assume that if a branch “might” be taken, it is safe to fetch down the target path.

2.8 Privileged Mode Operation

In the PowerPC Architecture, several terms describe two operating modes that have different instruction execution privileges. When a processor is “privileged mode,” it can execute all instructions in the instruction set. This mode is also called the “supervisor state.” The other mode, in which certain instructions cannot be executed, is called the “user mode,” or “problem state.” These terms are used in pairs:

Table 2-7. Instruction Execution Privileges and Operating Modes

Privileged	Nonprivileged
Privileged Mode	User Mode
Supervisor State	Problem State

The architecture uses the PR in the Machine Status Register (MSR) to controls the execution mode. When $\text{MSR}[\text{PR}] = 1$, the processor is in user mode (problem state); when $\text{MSR}[\text{PR}] = 0$, the processor is in privileged mode (supervisor state).

2.8.1 MSR Bits and Exception Handling

Attempting to execute a privileged instruction while $\text{MSR}[\text{PR}] = 1$ causes a privileged violation program exception (see Section 5.9, “Program Exceptions,” on p. 5-18). The PPC401A1 does not execute the instruction, and the least-significant 16 bits of the program counter are loaded with 0x0700, the address of an exception processing routine.

The current value of the $\text{MSR}[\text{PR}]$ bit is saved in the SRR1/SRR3 (along with all the other MSR bits) upon any interrupt, and the $\text{MSR}[\text{PR}]$ bit is set to 0, in all cases. This means that all exception handlers operate in privileged mode.

The Exception Syndrome Register (ESR) distinguishes different types of program exceptions. ESR[PPR] is set when the exception was caused by a privileged exception. Software is not required to clear this ESR bit.

2.8.2 Privileged Instructions

The following instructions are privileged and cannot be executed when MSR[PR] = 1:

Table 2-8. Privileged Instructions

dcbi	
dccci	
dcread	
icbt	
iccci	
icread	
mfocr	
mfmsr	
mfmspr	For all SPRs except CTR, LR, TBHU, TBLU, XER. See Section 2.8.3
mtocr	
mtmsr	
mtmspr	For all SPRs except CTR, LR, XER. See Section 2.8.3
rfti	
rfti	
wrttee	
wrtteei	

2.8.3 Privileged SPRs

All SPRs are privileged, except for the LR, the CTR, the TBHU, the TBLU, and the XER. Except for moves to and from non-privileged SPRs, attempts to execute **mfmspr** and **mtmspr** instructions while in user mode result in privileged violation program exceptions.

In a **mfmspr** or **mtmspr** instruction, the 10-bit SPRN field specifies the SPR number of the source or destination SPR. The SPRN field contains two five-bit subfields, SPRN_{0:4} and SPRN_{5:9}. The assembler handles the unusual register number encoding to generate the SPRF field. In the *machine code* for the **mfmspr** and **mtmspr** instructions, the SPRN subfields are *reversed* (ending up as SPRF_{5:9} and SPRF_{0:4}) for compatibility with the POWER Architecture.

In the PowerPC Architecture, SPR numbers having a 1 in the most-significant bit of the SPRF field are privileged.

The following example illustrates how SPR numbers appear in assembler language coding and in machine coding of the **mf spr** and **mt spr** instructions.

In assembler language coding, SRR0 is SPR 26. Note that the assembler handles the unusual register number encoding to generate the SPRF field.

```
mf spr r5,26
```

When the SPR number is considered as a binary number (0b00000 11010), the most-significant bit is 0. However, the machine code for the instruction reverses the subfields, resulting in the following SPRF field: 0b11010 00000. The most-significant bit is 1; SRR0 is privileged.

When an SPR number is considered as a hexadecimal number, the second digit of the three-digit hexadecimal number indicates whether an SPR is privileged. If the second digit is odd (1, 3, 5, 7, 9, B, D, F), the SPR is privileged.

For example, the SPR number of SRR0 is 26 (0x01A). The second hexadecimal digit is odd; SRR0 is privileged. In contrast, the LR is SPR 8 (0x008); the second hexadecimal digit is not odd; the LR is nonprivileged.

2.8.4 Privileged DCRs

The **mt dcr** and **mf dcr** instructions themselves are privileged, in all cases. All DCRs are privileged.

2.9 Synchronization

The PPC401A1 supports the synchronization operations of the PowerPC Architecture. The following book, chapter, and section numbers refer to related information in *The PowerPC Architecture: A Specification for a New Family of RISC Processors*:

- Book II, Section 1.8.1, “Storage Access Ordering” and “Enforce In-order Execution of I/O”
- Book III, Section 1.7, “Synchronization”
- Book III, Chapter 7, “Synchronization Requirements for Special Registers and Lookaside Buffers”

2.9.1 Context Synchronization

The context of a program is the environment (for example, privilege and relocation) in which the program executes. Context is controlled by the content of certain registers, such as the Machine State Register (MSR), and includes the content of all GPRs and SPRs.

An instruction or event is “context synchronizing” if it satisfies the following requirements:

1. All instructions that *precede* a context synchronizing operation must complete in the context that existed *before* the context synchronizing operation.
2. All instructions that *follow* a context synchronizing operation must complete in the context that exists *after* the context synchronizing operation.

Such instructions and events are called “context synchronizing operations.” In the PPC401A1, these include most interrupts and the **isync**, **rftci**, **rfti**, and **sc** instructions.

However, “context” specifically excludes the contents of memory. A context synchronizing operation does not guarantee that subsequent instructions observe the memory context established by previous instructions. To guarantee memory access ordering in the PPC401A1, one must use either an **eieio** instruction or a **sync** instruction. Note that for the PPC401A1, the **eieio** and **sync** instructions are implemented identically. See Section 2.9.3 on p. 2-44.

The contents of DCRs are not considered as part of the processor “context” managed by a context synchronizing operation. DCRs are peripherals of a processor, and are analogous to memory-mapped registers. Their context is managed in a manner similar to that of memory contents.

Finally, implementations of the PowerPC Architecture can exempt the machine check exception from context synchronization control. If the machine check exception is exempted, an instruction that *precedes* a context synchronizing operation can cause a machine check exception *after* the context synchronizing operation occurs and additional instructions have completed.

The following scenarios use pseudocode examples to illustrate these limitations of context synchronization. Subsequent text explains software can further guarantee “storage ordering.”

1. Consider the following instruction sequence:

```
STORE non-cacheable to address XYZ
isync
XYZ instruction
```

In this sequence, the **isync** instruction does not guarantee that the XYZ instruction is fetched after the STORE has occurred to memory. There is no guarantee which XYZ instruction will execute; either the old version or the new (stored) version might.

2. , which assumes that a PPC401A1 is part of a standard product that uses DCRs to provide bus region control using DCR:

```
STORE non-cacheable to address XYZ
isync
MTDCR to change a bus region containing XYZ
```

In this sequence, there is no guarantee that the **STORE** will occur before the **mtdcr** instruction changing the bus region control DCR. The **STORE** could fail because of a configuration error.

To see what context synchronization accomplishes, consider an interrupt that changes the PowerPC endian mode. An interrupt is a context synchronizing operation. Any interrupt causes the MSR to be updated; its old value is saved in SRR1 or SRR3. The MSR[ILE] bit is copied to MSR[LE]. The MSR is part of the processor context; the context synchronizing operation guarantees that all instructions that precede the interrupt complete using the pre-interrupt value of the MSR[LE]; all instructions that follow the interrupt complete using the post-interrupt value.

Consider, on the other hand, some code that uses the **mtmsr** instruction to change the value of the MSR[LE] bit, which changes the PowerPC endian mode. In this case, the MSR is changed, changing the context. It is possible, for example, that pre-fetched instructions expect to access big endian objects after the **mtmsr** has changed the endian mode to PowerPC little endian. This could cause problems. To prevent such problems, the code should execute a context synchronization operation, such as **isync**, immediately after the **mtmsr** instruction.

How can software ensure that the contents of memory and DCRs are synchronized in the instruction stream? The **eieio** instruction or the **sync** instruction perform this task. These instructions guarantee storage ordering; all memory accesses that precede **eieio** or **sync** affect the results of all subsequent memory accesses. Neither **eieio** nor **sync** guarantee that instruction pre-fetching follows the **eieio** or **sync**. The instructions do not cause the pre-fetch queues to be purged and instructions to be refetched. See Section 2.9.3 for more information about **sync** and **eieio**.

Instruction cache state is part of context. A context synchronization operation is required to guarantee instruction cache access ordering.

3. Consider the following instruction sequence, which is required for self-modifying code:

STORE	Change data cache contents
dcbst	Flush the new data cache contents to memory
sync	Guarantee that dcbst completes before subsequent instructions begin
icbi	Context changing operation; invalidates instruction cache contents.
isync	Context synchronizing operation; causes refetch using new instruction cache context text and new memory context, due to the previous sync .

Similarly, if software wishes to ensure that all storage accesses are complete before executing a **mtdcr** to change a bus region (Example 2), the software must issue a **sync** after all storage accesses and before the **mtdcr**. Likewise, if the software is to ensure that all instruction fetches after the **mtdcr** use the new bank register contents, the software must issue an **isync**, after the **mtdcr** and before the first instruction that should be fetched in the new context.

The **isync** instruction guarantees that all subsequent instructions are fetched and executed using the context established by all previous instructions. The **isync** instruction is a context synchronizing operation; **isync** causes all pre-fetched instructions to be discarded and refetched.

The following example illustrates the use of **isync** with debug exceptions:

mtdbcr	Set up an instruction address compare (IAC) event
isync	Wait for the new Debug Control Register (DBCR) context to be established
XYZ	This instruction is at the IAC address; an isync was necessary to guarantee that the IAC event will happen at the execution of this instruction

2.9.2 Execution Synchronization

For completeness, consider the definition of execution synchronizing as it relates to context synchronization. Execution synchronization is architecturally a subset of context synchronization.

Execution synchronization guarantees that the following requirement is met:

All instructions that *precede* an execution synchronizing operation must complete in the context that existed *before* the execution synchronizing operation.

The following requirement need not be met:

All instructions that *follow* an execution synchronizing operation must complete in the context that exists *after* the execution synchronizing operation.

Execution synchronization ensures that preceding instructions execute in the old context; subsequent instructions might execute in either the new or old context (indeterminate). The PPC401A1 provides three execution synchronizing operations: the **eieio**, **mtmsr**, and **sync** instructions.

Because **mtmsr** is execution synchronizing, it guarantees that previous instructions complete using the old MSR value. (Consider the previous example of using **mtmsr** to change the endian mode.) However, to guarantee that subsequent instructions use the new MSR value, we have to insert a context synchronization operation, such as **isync**.

Note that the PowerPC Architecture requires MSR[EE] (the external interrupt bit) to be, in effect, execution synchronizing: if a **mtmsr** turns on the EE bit, and an external interrupt is pending, the exception must be taken before the instruction that follows **mtmsr** is executed. However, the **mtmsr** instruction is not a context synchronizing operation, so the PPC401A1 does not, for example, discard pre-fetched instructions and refetch. Note that the **wrtee** and **wrteei** instructions can change the value of MSR[EE], but are not execution synchronizing.

Finally, while **sync** and **eieio** are execution synchronizing, they are also more restrictive in their requirement of memory ordering. Stating that an operation is execution synchronizing

does not imply storage ordering. This is an additional specific requirement of **sync** and **eiio**.

2.9.3 Storage Synchronization

The **sync** instruction guarantees that all previous storage references complete with respect to the PPC401A1 before the **sync** instruction completes (therefore, before any subsequent instructions begin to execute). The **sync** instruction is execution synchronizing.

Consider the following use of **sync**:

stw	Store to I/O device
sync	Wait for store to actually complete off chip
mtdcr	Reconfigure device

The **eiio** instruction guarantees the order of storage accesses. All storage accesses that precede **eiio** complete before any storage accesses that follow the instruction, as in the following example:

stb X	Store to I/O device, address X; this resets a status bit in the device
eiio	Guarantee stb X completes before next instruction
lbz Y	load from I/O device, address Y; this is the status register updated by stb X . eiio was necessary, because the read and write addresses are different, but affect each other

The PPC401A1 implements both **sync** and **eiio** identically, in the manner described above for **sync**. In the PowerPC Architecture, **sync** can function across all processors in a multiprocessor environment; **eiio** functions only within its executing processor. The PPC401A1 is a uniprocessor; in this implementation, **sync** does not guarantee memory ordering across multiprocessors.

2.10 Instruction Set

The PPC401A1 instruction set contains instructions defined in the PowerPC Architecture and instructions specific to the IBM PowerPC 400 family of embedded controllers.

Chapter 9, "Instruction Set," contains detailed descriptions of each instruction, including pseudocode. Appendix A, "Instruction Summary," alphabetically lists each instruction and extended mnemonic and provides a short-form description. Appendix B, "Instructions By Category," provides short-form descriptions of instructions, grouped by the instruction categories listed in Table 2-9.

Table 2-9 summarizes the PPC401A1 instruction set functions by categories. Instructions within each category are described in subsequent sections.

Table 2-9. PPC401A1 Instruction Set Functional Summary

Storage Reference	load, store
Arithmetic and Logical	add, subtract, negate, multiply, divide, and, andc, or, orc, xor, nand, nor, xnor, sign extension, count leading zeros
Comparison	compare, compare logical, compare immediate
Branch	branch, branch conditional, branch to LR, branch to CTR
CR Logical	crand, crandc, cror, crorc, crnand, crnor, crxor, crxnor, move CR field
Rotate/Shift	rotate and insert, rotate and mask, shift left, shift right
Cache Control	invalidate, touch, zero, flush, store, read
Interrupt Control	write to external interrupt enable bit, move to/from MSR, return from interrupt, return from critical interrupt
Processor Management	system call, synchronize, trap, move to/from DCRs, move to/from SPRs, move to/from CR

2.10.1 Instructions Specific to IBM PowerPC Embedded Controllers

To support functions required in embedded real-time applications, the IBM PowerPC 400 family of embedded controllers defines instructions that are not defined in the PowerPC Architecture.

Table 2-10 lists the instructions specific to IBM PowerPC embedded controllers. Programs using these instructions are not portable to PowerPC implementations that are not part of the IBM PowerPC 400 family of embedded controllers.

Table 2-10. Instructions Specific to IBM PowerPC Embedded Controllers

dccci	mfdcr
dcread	mtdcr
iccci	rfci
icbt	wrtree
icread	wrtreei

2.10.2 Storage Reference Instructions

Load and store instructions transfer data between memory and the GPRs. These instructions operate on bytes, halfwords, and words. Storage reference instructions also support loading or storing multiple registers, character strings, and byte-reversed data.

Table 2-11 shows the storage reference instructions in the PPC401A1.

Table 2-11. Storage Reference Instructions

Loads					Stores			
Byte	Halfword Algebraic	Halfword	Multiple and String	Word	Byte	Halfword	Multiple and String	Word
lbz lbzu lbzux lbzx	lha lhau lhaux lhax	lhrx lh lhzu lhzux lhzx	lmw lswi lswx	lwarx lwbrx lwz lwzu lwzux lwzx	stb stbu stbux stbx	sth sthbrx sthu sthux sthx	stmw stswi stswx	stw stwbrx stwu stwux stwx stwcx.

2.10.3 Arithmetic and Logical Instructions

Arithmetic operations are performed on integer or ordinal operands stored in registers. Instructions that perform operations on two operands are defined in a three-operand format; an operation is performed on the operands, which are stored in two registers. The result is placed in a third register. Instructions that perform operations on one operand are defined in a two-operand format; the operation is performed on the operand in a register and the result is placed in another register. Several instructions also have immediate formats in which an operand is a field in the instruction.

Most arithmetic and logical instructions can set the Condition Register (CR) based on the result of the instruction. The instructions having mnemonics ending in . (period) are the forms that set the CR.

Table 2-12 lists the arithmetic and logical instructions in the .

Table 2-12. Arithmetic and Logical Instructions

Arithmetic						Logical		
add	addi	divw	mulhw	subf	subfic	and	eqv	nor
add.	addic	divw.	mulhw.	subf.	subme	and.	eqv.	nor.
addo	addic.	divwo	mulhwu	subfo	subme.	andc		
addo.	addis	divwo.	mulhwu.	subfo.	submeo	andc.	extsb	or
addc	addme	divwu	mulli	subfc	submeo.	andi.	extsb.	or.
addc.	addme.	divwu.	mullw	subfc.	subfze	andis.		orc
addco	addmeo	divwuo	mullw.	subfco	subfze.		extsh	orc.
addco.	addmeo.	divwuo.	mullwo	subfco.	subfzeo	cntlzw	extsh.	ori
adde	addze		mullwo.	subfe	subfzeo.	cntlzw.		oris
adde.	addze.			subfe.			nand	
addeo	addzeo		neg	subfeo			nand.	xor
addeo.	addzeo.		neg.	subfeo.				xor.
			nego					xori
			nego.					xoris

2.10.4 Compare Instructions

These instructions perform arithmetic or logical comparisons between two operands and set the CR.

Table 2-13 lists the comparison instructions in the PPC401A1.

Table 2-13. Compare Instructions

Arithmetic	Logical
cmp	cmpl
cmpi	cmpli

2.10.5 Branch Instructions

These instruction unconditionally or conditionally branch to any address. Conditional branch instructions can test condition codes set by a previous instruction and branch accordingly. Conditional branch instructions can also decrement and test the Count Register as part of branch determination, and can save the return address in the Link Register. The target address for a branch can be a displacement from the current instruction address or an absolute address, or contained in the link or count registers.

Table 2-14 lists the branch instructions in the PPC401A1.

Table 2-14. Branch Instructions

Unconditional	Conditional
b ba bl bla	bc bca bcl bcla bcctr bcctrl bclr bclrl

2.10.6 Condition Register Logical Instructions

These instructions combine the results of several comparisons without incurring the overhead of conditional branching. Code performance can significantly improve if multiple conditions are tested before a branch decision.

Table 2-15 lists the condition register logical instructions in the PPC401A1.

Table 2-15. Condition Register Logical Instructions

crand	crnor
crandc	cror
creqv	crorc
crnand	crxor
	mcrf

2.10.7 Rotate and Shift Instructions

These instructions rotate or shift operands stored in the GPRs. Rotate instructions can also mask rotated operands.

Table 2-16 lists the rotate and shift instructions in the PPC401A1.

Table 2-16. Rotate and Shift Instructions

Rotate	Shift
rlwimi	slw
rlwimi.	slw.
rlwinm	sraw
rlwinm.	sraw.
rlwnm	srawi
rlwnm.	srawi.
	srw
	srw.

2.10.8 Cache Control Instructions

These instructions indirectly control the contents of the data and instruction caches. Users can fill, flush, invalidate, and zero blocks (16-byte lines) in the data cache. Users can invalidate and fill individual lines in the instruction cache, and invalidate congruence classes in both caches.

Table 2-17 lists the cache control instructions in the PPC401A1.

Table 2-17. Cache Control Instructions

Data Cache	Instruction Cache
dcbf dcbi dcbst dcbt dcbtst dcbz dccc dcread	icbi icbt iccci icread

2.10.9 Interrupt Control Instructions

These instructions move data between GPRs and the MSR, return from interrupts, and enable or disable maskable external interrupts.

Table 2-18 lists the interrupt control instructions in the PPC401A1.

Table 2-18. Interrupt Control Instructions

mfmsr mtmsr rfi rfci wrtee wrteei
--

2.10.10 Processor Management Instructions

These instructions move data between the GPRs and control registers in the PPC401A1, and provide traps, system calls, and synchronization controls.

Table 2-19 lists the processor management instructions in the PPC401A1.

Table 2-19. Processor Management Instructions

eieio	mcrxr	mtcrf
isync	mfcrr	mtdcr
sync	mfdcr	mtspr
	mfsprr	sc
		tw
		twi

2.10.11 Extended Mnemonics

In addition to mnemonics for instructions supported directly by hardware, the PowerPC Architecture defines numerous *extended mnemonics*.

An extended mnemonic translates directly into the mnemonic of a hardware instruction, typically with carefully specified operands. For example, the PowerPC Architecture does not define a “shift right word immediate” instruction, because the “rotate left word immediate then AND with mask,” (**rlwinm**) instruction can accomplish the same result:

rlwinm RA,RS,32–n,n,31

However, because the required operands are not obvious, the PowerPC Architecture defines an extended mnemonic:

srwi RA,RS,n

Extended mnemonics transfer the problem of remembering complex or frequently used operand combinations to the assembler, and can more clearly reflect a programmer’s intentions. Thus, programs can be more readable.

Refer to the following chapter and appendixes for lists of the extended mnemonics:

- Chapter 9, “Instruction Set,” lists extended mnemonics under the associated hardware instruction mnemonics.
- Appendix A, “Instruction Summary,” lists extended mnemonics alphabetically, along with the hardware instruction mnemonics.
- Table B-4 on p. B-5 in Appendix B, “Instructions By Category,” lists all extended mnemonics.

The PPC401A1 provides basic instruction processing functions. Other common computing functions are provided through several interfaces. The following interfaces provide I/O signals that connect to other macros and to ASIC logic:

- Clock and power management (CPM), on p. 3-4
- CPU control, on p. 3-10
- Test mode matrix (TMM)/level-sensitive scan design (LSSD), on p. 3-11
- Reset, on p. 3-20
- Instruction-side processor local bus (PLB), on p. 3-27
- Data-side processor local bus (PLB), on p. 3-44
- Device Control Register (DCR), on p. 3-84
- External interrupt controller (EIC), on p. 3-99
- Joint Test Action Group (JTAG) test access port (TAP), on p. 3-102
- Debug (DBG), on p. 3-108

This chapter describes these interfaces in detail. Each section describes an interface, providing a textual overview, an I/O symbol, a summary signal table followed by detailed signal descriptions, and, where appropriate, detailed timing diagrams. The overviews summarize the interface functions. The I/O symbols provide a quick view of the signal names and their directions with respect to the PPC401A1 macro. The summary signals also provide signal names and directions, along with basic timing information and signal termination for unused signals. The detailed signal descriptions provide logic design information. The timing diagrams, which provide timings for numerous transactions, illustrate best-case performance (in general) when the core is attached to the IBM Core+ASIC PLB macro, or to custom bus interface unit (BIU) designs.

3.1 Signal Naming Conventions

Signal names used in this document follow the format defined below:

PREFIX1_prefix2SigName1[_sigName2][_ NEG][[(m:n)]]

where:

- PREFIX1 is an *uppercase* prefix identifying the source of the signal, either by unit name (such as CPU, DCU) or by the interface type (such as DCR, LSSD).
- _prefix2 is a *lowercase* prefix identifying the destination of the signal, either by unit name (e.g. cpu, dcu) or by the interface type (e.g. dcr, lssd).
- SigName1 is a *mixed case* name reflecting the primary function of the signal.
- [sigName2] (optional) is a *mixed case* name reflecting the secondary function of the signal, and is separated from SigName1 by an underscore.
- [_NEG] (optional) denotes that a signal is active low. Unless so denoted, all signals are active high.
- [(m:n)] (optional) indicates a bussed signal. This notation is for reading convenience only; the actual core signal names must be unbussed, 1-bit signal names. “Bussed” signals appear on the macro symbol in an expanded format, each including the base name and a one or two digit suffix (depending on whether the highest bit position must be represented by one or two digits) identifying that signal's bit position on the bus.

Actual PPC401A1 hard macro signal names do not follow the format above. These names must be unbussed, and contain only uppercase letters and numbers. Hard macro signal names are limited to seven characters. Other characters, such as underscores, are illegal. These restrictions make the hard core macro compatible with a variety of vendor development tools for chip design, simulation, synthesis, timing, and so on. The hard macro signal names appear in the signal descriptions for each in PPC401A1 interface, which are organized by the signal names that follow the naming conventions.

Such signal names are used throughout this chapter to ease reading of the PPC401A1 signal names and more clearly identify function. These signal names also appear at the test mode matrix (TMM) boundary, an unsynthesized “soft” core that provides a test wrapper around the PPC401A1. These soft cores need not follow the naming restrictions enforced for the hard cores.

Table 3-1 defines the prefixes used in the signal names. The second column in the table identifies whether the prefix refers to a logic unit that resides *on* the PPC401A1 or *off* of the PPC401A1. For example, the DCU unit, which resides on the PPC401A1, is an on-core unit.

The CPM unit, which does not reside on the PPC401A1, is an off-core unit, is not part of the PPC401A1.

Table 3-1. Signal Name Prefix Definitions

Prefix1 (prefix2)	Definition	On/Off Core
BIU (biu)	Bus Interface Unit	Off
CORE (core)	Generic PPC401xx core identifier	On
CPM (cpm)	Clock and power management	Off
CPU (core)	Core CPU	On
DBG (dbg)	Debug unit	On
DCR (dcr)	Device Control Register	Off
DCU (dcu)	Data cache unit	On
EIC (eic)	External interrupt controller	Off
ICU (icu)	Instruction cache unit	On
JTAG (jtag)	JTAG	On
LSSD (lssd)	Level-sensitive scan design	On, Off
TIE (tie)	TIE (statically, to GND or V_{DD})	Off
TMM	Test/TMM control	On, Off
XXX (xxx)	Unspecified ASIC unit	Off

3.2 Signal Name Cross-Reference

Chapter 11, “Signal Summary,” lists the signals described in this chapter alphabetically and also provides the I/O type, hard macro signal name, and page reference for each signal.

3.3 Timing Definitions

The signal summary tables provided for each interface provide signal timing information for the I/O signals, as defined in the following list.

The signal timing informations apply to all core I/O interfaces. Actual input set-up times and output delays may vary from these guidelines. Early timing analysis should be performed, at the chip level, on all Core+ASIC chips using the PPC401A1 to ensure that timing objectives can be met.

Begin	Valid within 8% of the clock cycle from the rise of the core clock signal.
Early	Valid within 18% of the clock cycle from the rise of the core clock signal.
Early +	Valid within 28% of the clock cycle from the rise of the core clock signal.
Middle -	Valid within 33% of the clock cycle from the rise of the core clock signal.
Middle	Valid within 43% of the clock cycle from the rise of the core clock signal.
Middle +	Valid within 53% of the clock cycle from the rise of the core clock signal.
Late -	Valid within 58% of the clock cycle from the rise of the core clock signal.
Late	Valid within 68% of the clock cycle from the rise of the core clock signal.
End	Valid within 78% of the clock cycle from the rise of the core clock signal.

3.4 Clock and Power Management (CPM) Interface

The CPM interface provides clock control capabilities for power-sensitive applications.

There are three primary methods of clock control for the PPC401A1:

- Custom local gating (within the core, not controllable)

The core logic is designed so that most latches are *not* clocked every cycle. Instead of using feedback and constant clocking to retain latch values, clocks within the core are gated so that latches are only updated when necessary. The gating may be done to the L1 portion of a latch, the L2 portion of a latch, or a combination of the two. This design methodology has accounted for extremely good power numbers for all of the 4xx processor cores. Each register (collection of latches) has its own clock splitter. A register can also have unique clock gating logic that feeds into its clock splitter. The gating controls whether or not the value of the register changes on a cycle by cycle basis. This customized local gating is built-in and is not controllable at the CPM boundary.

- Global local gating (Potential sleep mode)

The core has two clock enable inputs, CPM_cpuAllowCoreClk and CPM_cpuAllowTimerClk. These signals feed into the clock splitters of all of the registers belonging to their respective clock zone. These are called “global local” gating signals because of the way that one entry pin fans out to each local clock splitter in its clock zone. This gating inhibits the clock from propagating through the clock splitters to all of the L1/L2 latches in its zone.

The PPC401A1 logic is divided into three clock zones: timer, core and jtag.

The timer zone comprises the logic around the PPC401A1 timer facilities. This zone is separated to enable a programmer to run the timers continually while the rest of the core logic sleeps. A timer event (interrupt request or reset) can then wake up the rest of the core logic. The timer zone is controlled by the CPM_cpuAllowTimerClk signal.

The core zone comprises the logic that does not belong to the jtag or timer zones. This zone contains most of the core logic. The core zone is controlled by the CPM_cpuAllowCoreClk signal.

The jtag zone runs off of the JTAG TCK clock, and is not controllable at the CPM boundary. It is a violation of the JTAG standard to have global-local gating control on the JTAG TCK signal. The TCK input must be free from on-chip control to enable basic JTAG functionality when the rest of the chip, including the ASIC designer's CPM macro, is not running.

- Global gating (Potential sleep mode)

This method of clock gating blocksg the clock to the core so that the clock input into the core, CPM_cpuSystemClock, no longer toggles. If a sleep mode employs this type of loba gating, the CPM_cpuSystemClock signal should be held at a logic 1.

The PPC401A1 should wake from sleep mode at any of the following events:

- Assertion of a PPC401A1-generated timer interrupt or timer reset
- Assertion of a reset request at the system or chip level from a source other than the PPC401A1
- Assertion of an off-core external or critical interrupt input, if enabled by the appropriate Machine State Register (MSR) bit.
- Assertion of the debugHalt chip I/O signal, indicating that an external debug tool wants to access the PPC401A1 processor. (See Section 3.13.3.1 on p. 3-109 for a description of the debugHalt signal.)

3.4.1 CPM Interface I/O Symbol

Figure 3-1 illustrates the inputs and outputs of the CPM interface.

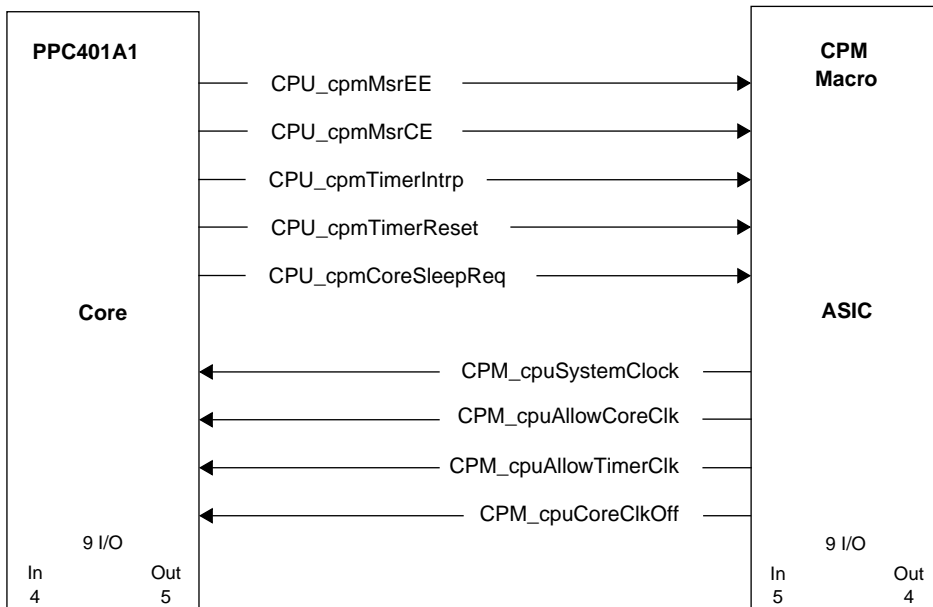


Figure 3-1. CPM Interface I/O Symbol

3.4.2 CPM Interface I/O SignalTable

Table 3-2 summarizes the CPM interface signals, which are described in detail in text following the table:

Table 3-2. CPM Interface I/O Signal Summary

Signal	I/O Type	If Unused	Function
CPM_cpuSystemClock	I	Required	The core's primary clock input (for all non-JTAG logic). This input is also used as an LSSD B clock during Core_LSSD mode.
CPM_cpuAllowCoreClk	I	1	Enables the internal generation of clocks for the bulk of the core logic. May be used to put the core in a sleep mode.
CPM_cpuAllowTimerClk	I	1	Enables the internal generation of clocks for the core's timer logic. May be used to put the timer logic in a sleep mode.
CPM_cpuCoreClkOff	I	0	Indicates that Clock and Power Management logic has disabled the clocks to the core.
CPU_cpmMsrEE	O	No connect	Indicates that the MSR's EE (External Interrupt Enable) bit is set.
CPU_cpmMsrCE	O	No connect	Indicates that the MSR's CE (Critical Interrupt Enable) bit is set.
CPU_cpmTimerIntrp	O	No connect	Indicates that a timer interrupt request has been generated. May be used to wake the CPU from a sleep mode.
CPU_cpmTimerReset	O	No connect	Indicates that the watchdog timer has generated a reset request. May be used to wake the CPU from a sleep mode.
CPU_cpmCoreSleepReq	O	No connect	Indicates that the core is requesting to be put into a sleep mode.

3.4.3 CPM Interface I/O Signal Descriptions

The following subsections describe the CPM interface I/O signals.

Each subsection heading names a signal and give its I/O type. Each subsection heading also provides the corresponding hard macro signal name, referred to as the core name.

3.4.3.1 CPM_cpuSystemClock (input)

Core Name: **CMSYSCK**

This signal is the system clock that is sent to the core. It is the source clock for all core registers, except for the JTAG TCK clocked registers and TMM latches, and custom arrays. This clock is split at each register clock splitter to create the C1/C2 clocks for that register. (The internal C2 clock has the same phase as this input signal.) If a sleep mode employs

gating of the CPM_cpuSystemClock signal (ie. the clock input does not switch) to the core, then this input should be held at a logic 1 during sleep mode.

This signal also provides the LSSD B Clock which is used to latch the L1 data output into the L2 portion of the latch for the PPC401A1 logic (except for the JTAG TCK clocked registers and the TMM latches) during Core_LSSD Test mode. This signal will be forced to a logic 0 internally during ASIC_LSSD mode to satisfy BTV test requirements that this clock be inactive to the core's internal scan chains during ASIC_LSSD mode.

3.4.3.2 CPM_cpuAllowCoreClk (input)

Core Name: **CMALWCC**

This signal is used in the clock splitter for the PPC401A1 latches (except for some timer latches covered below and the JTAG TCK clocked signals) to enable generation of clocks. If this signal is 1, the clocks are enabled into the latches. If this signal is 0, the clocks are disabled into the latches. This signal is used to put the core logic into a sleep mode during which the PPC401A1 clock input may be actively changing, but the clocks to the latches are disabled at the clock splitters. It should be fed from a CPM unit, or it can be tied to a logic 1 to continually enable clocks in the absence of such a unit.

3.4.3.3 CPM_cpuAllowTimerClk (input)

Core Name: **CMALWTC**

This signal is used in the clock splitter for a set of Timer latches in the PPC401A1 to enable generation of clocks. If this signal is a 1, the clocks are enabled into the latches. If this signal is a 0, the clocks are disabled into the latches. This signal is used to put the core's Timer logic into a sleep mode during which the PPC401A1 clock input may be actively changing, but the clocks to the timer latches are disabled at the clock splitters. It should be fed from a CPM unit, or it can be tied to a logic 1 to continually enable clocks in the absence of such a unit.

3.4.3.4 CPM_cpuCoreClkOff (input)

Core Name: **CMCLOFF**

This Core Clock Off signal is a status indicator that is sent from a CPM unit to the core where it is latched up in an internal register that is accessible using a debug tool such as RISCWatch. This signal should be asserted by a CPM unit when it has disabled clocks to the PPC401A1 logic by either of the methods below:

- Driving a logic 0 on the CPM_cpuAllowCoreClock input to the core
- Disabling the pulsing of the CPM_cpuSystemClock input to the core by holding the CPM_cpuSystemClock signal high (logic 1).

This status signal enables RISCWatch to detect that the core is asleep and needs to be woken by the assertion of the IOS_dbgDebugHalt input pin.

3.4.3.5 CPU_cpmMsrEE (output)Core Name: **CMMSREE**

This signal implements the External Interrupt Enable bit of the MSR (MSR[EE]). It is sent out for use in a CPM unit to enable external interrupts generated in logic outside of the core to wake the processor from asleep mode.

3.4.3.6 CPU_cpmMsrCE (output)Core Name: **CMMSRCE**

This signal implements the Critical Interrupt Enable bit of the MSR (MSR[CE]). It is sent out for use in a CPM unit to enable critical interrupts generated in logic outside of the core to wake the processor from a sleep mode.

3.4.3.7 CPU_cpmTimerIntrp (output)Core Name: **CMTMINT**

This timer interrupt request signal is sent to the CPM unit as a method of waking the core processor from a sleep mode. It is the logical OR of timer interrupt requests from the Programmable Interval Timer (PIT) and Fixed Interval Timer (FIT), which are enabled by MSR[EE], and the watchdog timer, which is enabled by the MSR[CE]. The logical AND of MSR[CE, EE] with the appropriate timer interrupt is done within the PPC401A1.

3.4.3.8 CPU_cpmTimerReset (output)Core Name: **CMTMRST**

This timer reset request signal is sent to a CPM unit as a method of waking the core processor from a sleep mode. It is the logical OR of the system, chip, and core reset request signals generated by the second expiration of the watchdog timer.

3.4.3.9 CPU_cpmCoreSleepReq (output)Core Name: **CMSLPRQ**

This core sleep request signal is sent to a CPM unit to indicate that the core is requesting to be put into a sleep mode. This signal is asserted by the processor when the CPU is in the wait state, and thus no longer processing instructions, the caches are idle (have completed all fill and/or flush operations initiated by processing of an instruction).

3.5 CPU Control Interface

The CPU control interface is primarily used to provide CPU setup information to the PPC401A1. It is also used to report the detection of a machine check condition within the PPC401A1. Lastly, it provides a few miscellaneous signals out to the ASIC logic that can be used by a bus interface unit (BIU) to report bus trace status.

3.5.1 CPU Control Interface I/O Symbol

Figure 3-2 illustrates the inputs and outputs of the CPU control interface.

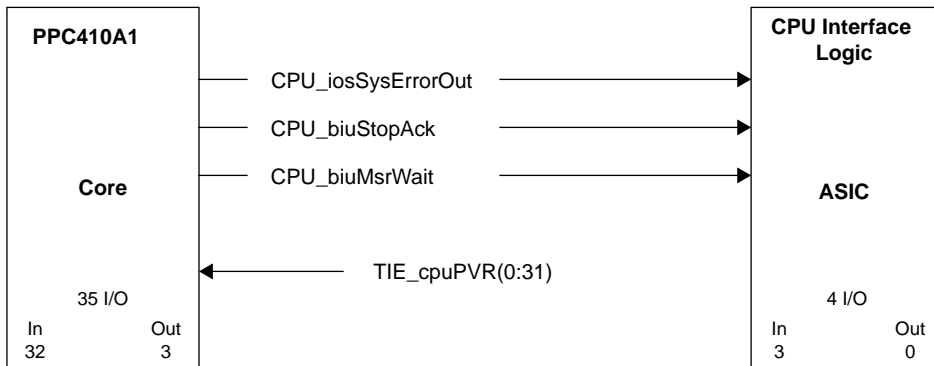


Figure 3-2. CPU Control Interface I/O Symbol

3.5.2 CPU Control Interface I/O Signal Table

Table 3-3 summarizes the CPU control interface signals, which are described in detail in text following the table.

Each subsection heading names the topic signal and give its I/O type. Each subsection heading also provides the hard macro signal cross-reference, referred to as the core name.

Table 3-3. CPU Control Interface I/O Signal Summary

Signal	I/O Type	If Unused	Function
TIE_cpuPVR(0:31)	I	Required	Unique chip ID. Contact IBM rep for proper assignment.
CPU_iosSysErrorOut	O	No connect	Indicates that a machine check error has been detected by the core.
CPU_biuStopAck	O	No connect	Indicates that the processor is stopped
CPU_biuMsrWait	O	No connect	Indicates that the MSR's WE (Wait Enable) bit is set

3.5.3 CPU Control Interface I/O Signal Descriptions

The following subsections describe the CPU control interface signals.

Each subsection heading names a signal and give its I/O type. Each subsection heading also provides the corresponding hard macro signal name, referred to as the core name.

3.5.3.1 TIE_cpuPVR(0:31) (input)

Core Name: **CPPVRnn**

This Processor Version Register (PVR) bus input must have each bit tied to V_{DD} or GND to uniquely identify the chip configuration. The identifying value of the bus must be supplied by Embedded PowerPC Technical Support (919/543-5701, ppcsupp@raleigh.ibm.com). Each chip released through the IBM Core+ASIC program must have a unique PVR.

The value of this PVR bus input is accessed as the PVR using the **mfspr** instruction. Access is read-only.

3.5.3.2 CPU_iosSysErrorOut (output)

Core Name: **CPERROR**

This signal is asserted when a machine check error is detected by the core. The machine check is a result of the attempted execution of an instruction that was fetched from the BIU/PLB and transferred to the PPC401A1 with the BIU_icuBusErr signal asserted, or from the reception of the BIU_dcuBusErr signal. This output signal remains a logic 1 until the machine check error is cleared in the Exception Syndrome Register (ESR).

3.5.3.3 CPU_biuStopAck (output)

Core Name: **CPSTPAK**

This signal is driven by the CPU to indicate that the processor is stopped. It is driven in response to a stop request that can be generated by an external debug tool (such as RISCWatch) through JTAG or the IOS_dbgDebugHaltXXX_dbgDebugHalt signal.

This bit is sent out of the core for use by ASIC logic that may want to incorporate BIU bus status (as an addition to the CPU trace status) out to a trace tool such as RISCTrace.

3.5.3.4 CPU_biuMsrWait (output)

Core Name: **CPMSRWE**

This signal is asserted by the CPU when the Wait Enable bit of the MSR (MSR[WE] is set. The CPU remains in the wait state until an exception is taken, the PPC401A1 is reset, or an external debug tool (such as RISCWatch) clears the MSR[WE].

This bit is sent out of the core for use by ASIC logic that may want to incorporate BIU bus status (as an addition to the CPU trace status) out to a trace tool such as RISCTrace.

3.6 Test Mode Matrix (TMM)/LSSD Test Interface

The TMM/LSSD test interface is comprised of the signals necessary to select the mode of operation of the PPC401A1 and control manufacturing test on the PPC401A1 and the customer's ASIC logic. To understand how this is accomplished, it is imperative to gain a basic understanding of the concept of the TMM.

The TMM is actually composed of two levels of logic.

The inner TMM level, referred to as *tmimi*, is within the boundary of the PPC401A1. It primarily consists of a set of specialized TMM latches that provide isolation of all of the PPC401A1 inputs and outputs, except for a few signals that are used for TMM control, TMM latch scan input and output, and LSSD clocking. This level of the TMM, since it is within the boundary of the PPC401A1 hard macro, is not observable to the ASIC customer.

The outer TMM level, referred to as *tmimo*, is a synthesizable wrapper that surrounds the PPC401A1 hardmacro. It's main function is to provide a customer's chip with the capability to selectively perform various manufacturing tests on the PPC401A1 or the remainder of the ASIC logic from the chip I/O boundary. It may also be used to run limited functional code on the PPC401A1 for functional verification, if the need arises. These capabilities are accomplished by way of sharing a subset of the chip I/O used by the ASIC customer. The pin sharing is accomplished by using multiplexing logic in the outer TMM. During normal chip functional mode, the customer's logic feeds the I/O. During the three test modes, the pins are fed by the PPC401A1 or ASIC logic, depending on the test mode. This added functionality does not come without a cost.. The TMM introduces some slight performance penalty (due to the muxing) and an area increase in order to accomplish its objectives.

This section does not provide a detailed description of the outer TMM. However, for the purposes of this document, you will need to understand the distinction between the two levels of the TMM and get a general grasp of what the TMM levels are doing and how they are doing it. (For more detailed information regarding the outer TMM (TMMO), including its inner structure and how to hook it up on a chip, ask your IBM Applications Engineer for the document titled "Using the Test Mode Matrix with the PPC401A1 Processor Core".)

The TMM operates in one of four modes. These modes are determined by the values on the TMM_coreTest and TMM_LSSDTest signals, as defined in Table 3-4, "TMM Mode Definitions"

Table 3-4. TMM Mode Definitions

TMM Mode (CoreTest /LssdTest) Definitions		
EncodedValue	Mode	Description
00	Chip_Functional	Normal chip functional mode
01	ASIC_LSSD Test	ASIC logic is under LSSD test control
10	Core_AVF Test	Core AVF test - functional pattern application
11	Core_LSSD Test	PPC401A1 logic is under LSSD test control

In **Chip_Functional** mode (“00”) the outer TMM simply routes the ASIC customer's connections to/from the PPC401A1 and the chip boundary, through the TMM layer of hierarchy. In this mode, the TMM is essentially transparent, except that an additional propagation delay will be inserted into most PPC401A1 input paths and some ASIC functional outputs to the chip boundary, depending on the pin-sharing selections made at the outer TMM level.

LSSD test for core+ASIC chips that use the PPC401A1 hard macro is *not* done as a single flat entity. Instead, the PPC401A1 LSSD Test is done separately from the remainder of the ASIC logic LSSD test. This decision was made primarily because the PPC401A1 has some untestable logic (including the clock splitters) for normal LSSD test. In order to boost the testability of the PPC401A1, functional patterns - or AVPs (Architectural Verification Patterns) - are run on the PPC401A1. By isolating the PPC401A1, a *single* set of PPC401A1 AVP test patterns may be used for *every* chip that uses the PPC401x2. If the core was not isolated, running AVPs, new patterns would need to be created for each new chip. The patterns would be *unique* for *each* chip that uses the PPC401A1. The resources required to perform, track, and control this effort is prohibitive. For this reason and others I will not go into, the Core_LSSD and Core_AVP test modes were separated from ASIC_LSSD test mode.

In **Core_LSSD** Test mode (“11”) the outer TMM routes the PPC401A1's scan chain inputs and outputs between the PPC401A1 and the chip I/O boundary. It also routes the LSSD test (input) signals and clocks from the chip I/O boundary to the PPC401A1. The bulk of the PPC401A1 logic inputs are fed through the TMM latches which are in the inner TMM level, or *tmimi*, located just within the PPC401A1. Remember that these TMM latches provide a boundary isolation of the PPC401A1's inner logic and the non-PPC401A1 ASIC logic (which includes the outer TMM.) . These TMM latches are loaded by scanning a pattern into them via the TMM_scanIn inputs. The PPC401A1 outputs are captured into the TMM latches and then scanned out through the TMM_scanOut signal. The PPC401A1's TMM latch scan input and output are also routed to the chip I/O through the outer TMM in Core_LSSD Test mode.

In **Core_AVP** mode, the outer TMM routes the signals used to run functional patterns between the PPC401A1 and the chip I/O boundary. There are 132 PPC401A1 I/O (not including the mode bits, TMM_coreTest and TMM_LSSDTest) that are used in Core_AVP mode. There are many more I/O at the core, but the outer TMM muxes outputs, shares inputs and creates bidis out of unidirectional buses in order to get the number of I/O required down to the 132 number. The ASIC designer must identify which of his functional I/O are candidates to share these 132 I/O required for test.

In **ASIC_LSSD** Test mode (“01”) the outer TMM routes the LSSD test (input) signals and clocks from the chip I/O boundary to the ASIC logic. The ASIC inputs which are fed from the PPC401A1 are fed by the TMM latches within the *tmimi*, which provide isolation of the ASIC logic from the core. These TMM latches are loaded by scanning a pattern into them via the TMM_scanIn inputs. The ASIC outputs which normally feed into the PPC401A1 are captured into the TMM latches and then scanned out through the TMM_scanOut signal. The PPC401A1's TMM latch scan input and output are routed to the chip I/O through the outer TMM in ASIC_LSSD Test mode, as they were in Core_LSSD Test mode. You can see that

the TMM Latches within the PPC401A1, and therefore the clocks that control them, are used in both LSSD modes.

3

3.6.1 TMM/Test Interface I/O Symbol

The TMM/test interface in Figure 3-3 illustrates the connection between the PPC401A1 and the outer TMM (TMMO). Because the TMMO is not part of the PPC401A1 and must be synthesized with the rest of the ASIC logic, the TMMO is considered ASIC logic. Signal names identical to those at the PPC401A1 boundary are brought out at the TMMO interface to the rest of the ASIC logic.

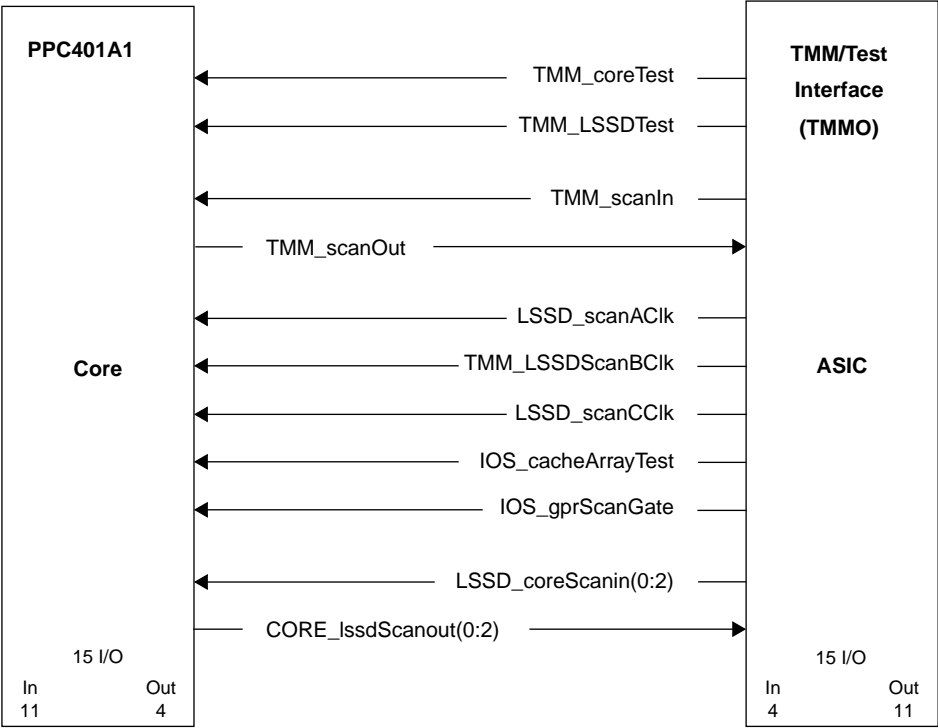


Figure 3-3. TMM/Test Interface I/O Symbol

3.6.2 TMM/Test Interface I/O Signal Table

Table 3-4 summarizes the TMM/Test interface signals, which are described in detail in text following the table.

Engineering Note: The If Unused column *does not* refer to use of the signal during the manufacturing test modes shown in Table 3-4, “TMM Mode Definitions,” on p. 3-12. All signals listed in Table 3-4 are required for

manufacturing test. “If unused” refers to use of this signal in functional mode only, during normal operation and, if implemented, scan flush initialization. *Scan flush initialization does not reset the PPC401A1*. Rather, it initializes the scan rings to some determinate state before application of a reset.

Table 3-5. TMM/Test Interface I/O Signals

Signal	I/O Type	If Unused	Function
TMM_coreTest	I	Required	TMM mode bit. Designates a Core-related test mode.
TMM_LSSDTest	I	Required	TMM mode bit. Designates an LSSD-related test mode.
TMM_LSSDScanBClk	I	0	LSSD B clock allows the value of the L1 output of a TMM latch into the L2 of the latch in an LSSD mode. Should be a '0' during Chip_Functional mode.
TMM_scanIn	I	0	LSSD Scan In port for the TMM latches within the PPC401x2.
TMM_scanOut	O	No connect	LSSD Scan Out port for the TMM latches within the PPC401x2.
LSSD_scanAClk	I	0	LSSD A clock allows scan data into the L1 of a latch in an LSSD mode.
LSSD_scanCClk	I	1	LSSD C clock captures system data into the L1 of a latch in an LSSD mode.
IOS_cacheArrayTest	I	0	LSSD array test enables Array Built In Self Test (ABIST) during Core_LSSD test mode.
IOS_gprScanGate	I	1	LSSD scan gate must be set to '0' during scan operations to gate off certain logic paths.
LSSD_coreScanin(0:2)	I	0b000	The core's LSSD scan in ports (non-TMM latches).
CORE_issdScanout(0:2)	O	No connect	The core's corresponding LSSD scan out ports (non-TMM latches).

3.6.3 TMM/Test Interface I/O Signal Descriptions

Each subsection heading names a signal and give its I/O type. Each subsection heading also provides the corresponding hard macro signal name, referred to as the core name.

3.6.3.1 TMM_coreTest (input)

Core Name: **TMCORET**

This signal, along with TMM_LSSDTest, indicates the mode of operation of a Core+ASIC chip. The TMM_coreTest signal indicates that the PPC401x2 is being selected for either architectural verification pattern (AVP) testing or for LSSD testing. AVP testing increases manufacturing test coverage for non-LSSD testable core logic, and provides a functional

verification tool for the core in an isolated environment. See Table 3-4, which lists the TMM operating modes.

3.6.3.2 TMM_LSSDTest (input)

Core Name: **TMLSSDT**

This signal, along with TMM_coreTest, indicates which mode of operation the Core+ASIC chip is in. The TMM_LSSDTest signal indicates that some area of the chip is being selected for LSSD testing. See Table 3-4, “TMM Mode Definitions,” on p. 3-12 for a list of the different TMM Modes of operation.

3.6.3.3 TMM_LSSDScanBClk (input)

Core Name: **TMLSSDB**

This signal is the LSSD Test Mode B Clock which is used to clock the L1 data into the L2 portion of a latch during LSSD Test Modes. This signal is only sent to the TMM Latches that are located within the PPC401x2. LSSD_tmmiBClk is sent to the TMM latches within the PPC401x2 during ASIC_LSSD mode or Core_LSSD mode. During Chip_Functional mode or Core_AVP mode, this signal is held to a '0' internally.

This signal is NOT used as an LSSD Test Mode B Clock for any other latches of the PPC401x2 (all non-TMM latches). The LSSD B Clocks for the core's non-TMM latches are CPM_cpuSystemClock (non-JTAG logic) and IOS_jtagTestClock (JTAG logic).

Refer to Table 3-6, “Clock Control to the PPC401A1 Latches,” on p. 3-17 to see how this signal (referred to as BClk₁) is used during the different TMM Modes. Be sure to look under the “TMM Latch” heading. You may want to compare the operation of this signal, which is only used during LSSD modes, with the operation of the other B Clocks (referred to as BClk₂) for the non-TMM latches, which share duty with the core clock in Chip_Functional mode and Core_AVP mode.

3.6.3.4 TMM_scanIn (input)

Core Name: **TMSCANI**

This signal is the LSSD Test Mode ScanIn for the TMM Latches within the core. The TMM Latch scan chain must be a separate scan chain since it is used in both Core_LSSD and ASIC_LSSD test. This input feeds the first TMM latch in the scan chain.

3.6.3.5 TMM_scanOut (output)

Core Name: **TMSCANO**

This signal is the LSSD Test Mode ScanOut for the TMM Latches within the core. The TMM Latch scan chain must be a separate scan chain since it is used in both Core_LSSD and ASIC_LSSD test. This output is driven by the last TMM latch in the scan chain.

3.6.3.6 LSSD_scanAClk (input)

Core Name: **LSSDA**

This signal is the LSSD Test Mode A Clock which is used to clock the scanIn data (pin I0) into the L1 portion of a latch during LSSD Test Modes. The LSSD_scanAClk signal feeds the TMM latches in the PPC401A1 through one set of gating conditions (in order to use these latches in either ASIC_LSSD or Core_LSSD mode), and feeds all non-TMM latches in the

PPC401A1 through a different set of gating conditions (for use with the specialized clock splitters used within the design). Refer to the descriptions that follow and Table 3-6, “Clock Control to the PPC401A1 Latches,” on p. 3-17 to see how the signal is used during the different TMM Modes.

LSSD_coreAClk is sent to the TMM latches within the PPC401A1 during ASIC_LSSD mode or Core_LSSD mode. During Chip_Functional mode or Core_AVP mode, this signal is held to 0 internally.

LSSD_coreAClk is sent to the non-TMM latches within the PPC401A1 during Core_LSSD mode only. During all other TMM modes of operation, it is held to a 0 internally.

Table 3-6. Clock Control to the PPC401A1 Latches

TMM Mode	Clocks to PPC401A1 TMM Latches			Clocks to PPC401A1 Non-TMM Latches		
	A _{g1}	B _{g1}	C _{g1}	A _{g2}	B _{g2}	C _{g2}
Chip_Func	Forced to 0	Forced to 0	Forced to 0	Forced to 0	Bclk ₂	Forced to 1
ASIC_LSSD	AClk	Bclk ₁	CClk	Forced to 0	Forced to 0	Forced to 0
Core_AVP	Forced to 0	Forced to 0	Forced to 0	Forced to 0	Bclk ₂	Forced to 1
Core_LSSD	AClk	Bclk ₁	CClk	AClk	Bclk ₂	CClk

The following definitions apply:
 AClk : LSSD_coreAClk input at the PPC401A1 boundary
 Bclk₁: LSSD_tmmiBclk input at the PPC401A1 boundary (TMM latch use only)
 Bclk₂: CPM_coreClock_issdBclk or JTE_jtiTCK_issdBclk input, as appropriate, at the PPC401A1 boundary
 CClk : LSSD_coreCClk input at the PPC401A1 boundary
 g1: gating conditions for clocks used at the TMM latches within the PPC401A1
 g2: gating conditions for clocks used at non-TMM latches within the PPC401A1

3.6.3.7 LSSD_scanCClk (input)

Core Name: **LSSDC**

This signal is the LSSD Test Mode C Clock which is used to capture the system data into the L1 portion of a latch during LSSD Test Modes. The LSSD_scanCClk signal feeds the TMM latches in the PPC401A1 through one set of gating conditions (in order to use these latches in either ASIC_LSSD or Core_LSSD mode), and feeds all non-TMM latches in the PPC401A1 through a different set of gating conditions (for use with the specialized clock splitters used within the design). Refer to the descriptions that follow and Table 3-6, “Clock Control to the PPC401A1 Latches,” on p. 3-17 to see how the signal is used during the different TMM Modes.

LSSD_coreCClk is sent to the TMM latches within the PPC401A1 during ASIC_LSSD mode or Core_LSSD mode. In Chip_Functional mode or Core_AVP mode, this signal is held to 0 internally.

LSSD_coreCClk is sent to the non-TMM latches within the PPC401A1 during Core LSSD mode only. During ASIC_LSSD mode this signal is held to a ‘0’ internally. During Chip_Functional mode or Core_AVP mode, this signal is held to a ‘1’ internally to enable clocking.

3.6.3.8 IOS_cacheArrayTest (input)

Core Name: **XARYTST**

This test mode signal is used to enable the PPC401A1 Array Built-In Self Test (ABIST) logic. This signal must be controllable at the core boundary during Core_LSSD test mode. In Core_LSSD mode, a logic '1' on this input places the core in ABIST mode. For all other TMM modes of operation, the ASIC designer should ensure that this signal will be a logic '0'. For the CPPC401M1 (Ford) core, this signal is forced to a logic 0 in all TMM modes other than Core_LSSD test mode, during which it is controllable at the core boundary. Therefore, the ASIC designer does not need to force the logic 0 values.

3.6.3.9 IOS_gprScanGate (input)

Core Name: **XGPRSG**

This test mode signal is used prohibit certain operations during Core_LSSD scan operations in order to protect circuitry within the PPC401x2. This signal should be controllable at the core boundary during Core_LSSD test mode. In all other TMM modes of operation, the ASIC designer should ensure that this signal will be a logic '1'. This signal is forced to a logic '1' in all TMM modes other than Core_LSSD test mode, during which it is controllable at the core boundary. Therefore, the ASIC designer does not need to force the logic '1' value.

In Core_LSSD mode, this signal should be a logic '0' whenever a scan operation is taking place. A scan operation can be defined as an LSSD test mode operation in which the LSSD A clock and LSSD B clock are being used to scan data through the LSSD scan chains, either in alternating order or in a "hot flush" scenario in which both clocks are active at the same time and the ScanIn data is being flushed through the chain.

In Core_LSSD mode, if the LSSD patterns are attempting to perform a fault capture, this signal will be driven to a logic '1' or logic '0' as needed for the test.

An example of the usage of this input is at the GPR Register Array. This signal must be a logic 0 to prohibit interference with a scan operation. The scan sequencing might produce a "Write" strobe to the GPR that could interfere with the scan operation (at the GPR's output latches) or even damage the circuitry within the GPR Register Array. If the chip is in Core_LSSD test mode and this signal is a '0', the GPR file cannot be written into. If the chip is in Core_LSSD test mode and this signal is a '1', the GPR file is able to be written into, if the logic allows it.

3.6.3.10 LSSD_coreScanin(0:2) (input)

Core Name: **LSSDSIn**

These signals are the LSSD Test Mode ScanIn signals for the PPC401A1 latches (non-TMM) within the core. These scan chains are used during Core_LSSD Test Mode for manufacturing test. These inputs feed the first latch in each of the three scan chains. Scan chain 0 travels through the cache units while scan chains 1 and 2 travel through various units of the CPU.

3.6.3.11 CORE_IssdScanout(0:2) (output)

Core Name: **LSSDSOn**

These signals are the LSSD Test Mode ScanOut signals for the PPC401A1 latches (non-TMM) within the core. These scan chains are used during Core_LSSD Test Mode for

manufacturing test. These outputs are driven by the last latch of each of the three scan chains. Scan chain 0 travels through the cache units while scan chains 1 and 2 travel through various units of the CPU.

3.7 Reset Interface

Resetting the PPC401A1 is accomplished by asserting of reset inputs into the core. The core is not reset using a scan-flush mechanism. Attempts to reset the core using scan-flush result in indeterminate logic states in the processor core. However, it is possible to *initialize* the PPC401A1 using a scan-flush technique. Some customers may wish to *initialize* the PPC401A1 out of its undefined state to remove X's in simulation (a practice which is not endorsed by the PPC401A1 design group since you may be masking code and hardware initialization problems). If a customer does implement a scan-flush *initialization*, they will still need to properly reset the PPC401A1 by the methods described below after the scan-flush initialization.

The PPC401A1 supports three types of reset, defined in Table 3-7, to provide granularity for resetting system resources..

Table 3-7. Reset Levels Supported by the PPC401A1

Reset Type	Definition
Core Reset	Resets the core, including the data and instruction caches.
Chip Reset	Resets the ASIC, including the core and on-chip peripherals and logic.
System Reset	Intended to reset the entire system including the core, the ASIC, and all logic external to the ASIC.

The PPC401A1 provides inputs for each reset level. The inputs are used to reset processor core logic (for core and system reset inputs) and to record the most recent reset type (for all three reset inputs).

The core can generate reset request outputs for each type of reset. Reset logic external to the processor is required to process these reset request outputs and generate the appropriate reset inputs to the core. The PPC401A1 does not initiate internal reset activity based on its reset request outputs. It resets internal logic only when its reset inputs are asserted. The core requests resets as the result to watchdog timer time-outs, internal (software) debug facilities, and JTAG debug facilities.

A system is not required to support all three types of reset, as long as the minimum reset requirements are met. The core interprets resets as described and generates requests for each type of reset, if programmed to do so. Distinguishing reset types makes it easier to isolate errors during system debug. For example, a system that distinguishes between a system, chip, and core resets can reset just the core, and then use on-chip or JTAG debug facilities to locate the source of an error outside the core. Insufficient reset granularity affects the ability to debug system problems and to recover from system problems.

3.7.1 Reset Requirements

Reset logic external to the processor core is required to process core reset request outputs and generate the appropriate reset inputs to the core. This logic must:

- Synchronize the core, chip and system reset inputs to the core with the core clock.

- Ensure that all reset inputs to the core remain active for at least four core clock cycles.
- Enforce the valid reset input combinations, as defined in Table 3-8.

Table 3-8. Valid Reset Input Combinations

System Reset	Chip Reset	Core Reset
1	1	1
0	1	1
0	0	1
0	0	0

- Ensure proper power-on reset (POR) operation. At POR, the system reset input (and, therefore, chip and core reset inputs) must be asserted for at least four core clock cycles. The JTAG reset input, `IOS_jtagResetIn` must also be asserted for at least four core clock cycles at POR. These minimum reset input durations consider only PPC401A1 reset requirements. Logic external to the core may require additional cycles of reset.
- Receive PPC401A1 reset requests and generate valid reset inputs, the remaining chip logic, and the off-chip system logic as required by the system design.

3.7.2 Reset Connectivity Example

Figure 3-4 illustrates a sample reset connection on an ASIC incorporating the PPC401A1. In the figure, `System Reset` is a negative active asynchronous bidirectional chip I/O that enables the chip to receive *and* generate system resets. `System Reset` is inverted and double-latched to present a positive active synchronous signal to the processor core. The double latching demetastabilizes the reset signal sent to the PPC401A1.

The externally generated system reset is logically ORed with an internally generated system reset as an input to the PPC401A1. In this configuration, external reset logic must provide the minimum number of cycles to reset the core for an externally supplied system reset. The reset stretch/control and distribution logic block provides the minimum number of cycles to reset the processor core for resets generated on-chip. The system reset is also logically ORed into the chip reset input to the PPC401A1 to maintain valid reset input combinations. Likewise, chip reset is logically ORed into core reset in order to maintain valid reset input combinations to the processor core.

The reset stretch/control and distribution logic block comprises the on-chip reset controller logic. This synchronous logic block distributes the proper resets throughout the chip, maintains the minimum number of cycles of reset required by the processor core and other chip logic, and processes reset requests from the core and other sources of reset requests. This logic block supports all three reset types (system, chip, and core).

The `IOS_jtagResetIn` core input is shown with two possible connections. Path1 connects this reset input to its own chip I/O. Path 2 connects `IOS_jtagResetIn` to the system reset chip I/O. Path2 saves a chip I/O, but could cause a JTAG debug tool (such as RISCWatch) to

become unsynchronized with the PPC401A1 JTAG controller when unexpected system resets occur. The path1 connection eliminates this problem.

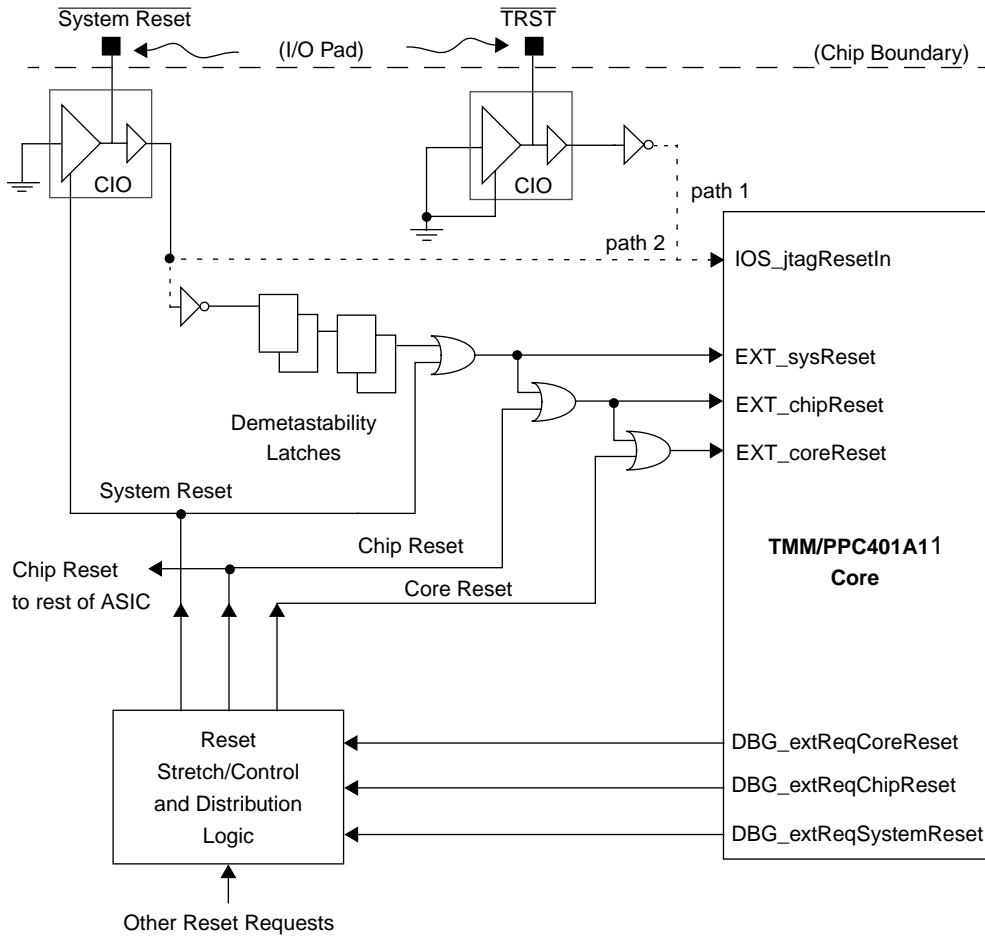


Figure 3-4. Sample On-chip Reset Connection

3.7.3 Reset Interface I/O Symbol

Signals shown with dashed lines represent signals that are involved in reset but are not fully described in this section.

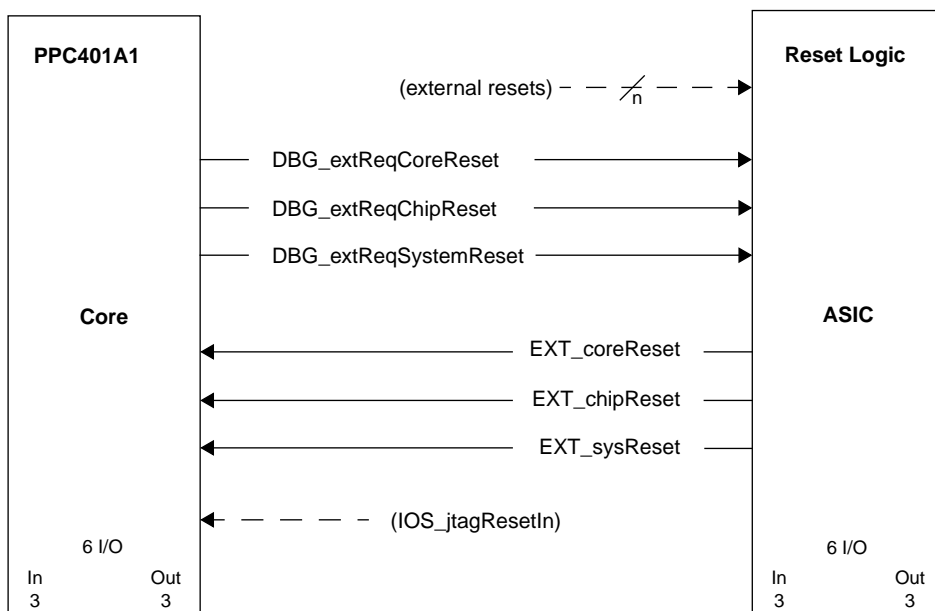


Figure 3-5. Reset Interface I/O Symbol

3.7.4 Reset Interface I/O Signal Table

Table 3-9 summarizes the reset interface signals, which are described in detail in text following the table.

Table 3-9. Reset Interface I/O Signal Table

Signal	I/O Type	If Unused	Function
DBG_extReqCoreReset	O	Required	Indicates that an event occurred to generate a core reset <i>request</i>
DBG_extReqChipReset	O	Required	Indicates that an event occurred to generate a chip reset <i>request</i>
DBG_extReqSystemReset	O	Required	Indicates that an event occurred to generate a system reset <i>request</i>
EXT_coreReset	I	Required	Core reset input that resets internal core logic. At least four cycles are required.

Table 3-9. Reset Interface I/O Signal Table

Signal	I/O Type	If Unused	Function
EXT_chipReset	I	Required	Chip reset input that alerts the core that a chip reset was driven.
EXT_sysReset	I	Required	System reset input that 1) alerts the core that a system reset was driven and 2) resets internal core debug logic in the JTAG unit.
IOS_jtagResetIn	I	Required	JTAG reset input <i>must</i> be asserted at POR. May be used as a positive active TRST thereafter. (See Section 3.12.3.4 on pg 3-105)

3.7.5 Reset Interface I/O Signal Descriptions

The following subsections describe the reset interface signals.

Each subsection heading names a signal and give its I/O type. Each subsection heading also provides the corresponding hard macro signal name, referred to as the core name.

3.7.5.1 DBG_extReqCoreReset (output)

Core Name: **DBRSTCO**

This core output signal is used to indicate that an event has occurred within the core to generate a **core** reset request. If this core reset request feeds combinatorially back into the EXT_coreReset input, the core reset request signal will be deasserted after eight clock cycles. If this core reset request signal does NOT feed back into the EXT_coreReset input combinatorially, then the core reset request indication should be latched by the ASIC logic when it is asserted. After it is latched, the signal should be ignored until the resulting core reset input into the core is deasserted. This reset request can be generated within the core by:

- JTAG or the RISCWatch debugger ,
- Software setting the Debug Control Register (DBCR) field DBCR[RST] = 01
- The second expiration of the watchdog timer when the Timer Control Register (TCR) field TCR[WRC] = 01

The assertion of this signal does not indicate that reset activity occurred within the core. Resetting the *core* is left to the ASIC designer. The core reset signal generated by the ASIC designer should feed into the EXT_coreReset input. The EXT_coreReset input should be asserted for four PPC401A1 clock cycles to guarantee that the PPC401A1 is put into its reset state. See Figure 3-4for a suggested reset implementation.

3.7.5.2 DBG_extReqChipReset (output)

Core Name: **DBRSTCH**

This core output signal indicates that an event has occurred within the core to generate a chip reset request. If this chip reset request feeds combinatorially back into the

EXT_chipReset input, the chip reset request signal will be deasserted after eight clock cycles. If this chip reset request signal does not feed back into the EXT_chipReset input combinatorially, then the chip reset request indication should be latched by the ASIC logic when it is asserted. After it is latched, the signal should be ignored until the resulting chip reset input into the core is deasserted. This reset request can be generated within the core by:

- JTAG or the RISCWatch debugger ,
- Software setting the DBCR[RST] = 10
- The second expiration of the watchdog timer when TCR[WRC] = 10

The assertion of this signal does not indicate that reset activity occurred within the chip or core. The resetting of the *chip* and *core* is left to the ASIC designer. The chip reset signal generated by the ASIC designer should feed into the EXT_chipReset input and should also cause the EXT_coreReset input to the core to be asserted as long as the chip reset input is asserted. The EXT_chipReset signal, and therefore the EXT_coreReset signal, should be asserted for at least four core clock cycles to guarantee that the PPC401A1 is put into its reset state. Other on-chip ASIC logic may require that these signals be held active for additional cycles. See Figure 3-4 for a suggested reset implementation.

3.7.5.3 DBG_extReqSystemReset (output)

Core Name: **DBRSTSY**

This core output signal is used to indicate that an event has occurred within the core to generate a system reset request. This system reset request signal will remain active until two cycles after the assertion of the EXT_sysReset input into the core. This reset request can be generated within the core by:

- JTAG or the RISCWatch debugger ,
- Software setting the DBCR[RST] = 11
- The second expiration of the watchdog timer when TCR[WRC] = 11

The assertion of this signal does not indicate that reset activity occurred within the system, chip or core. The resetting of the *system*, *chip*, and *core* is left to the ASIC designer. The system reset signal generated by the ASIC designer should feed into the EXT_sysReset input and should also cause the EXT_chipReset and EXT_coreReset inputs to the core to be asserted as long as the system reset input is asserted. The EXT_sysReset signal, and therefore the EXT_chipReset and EXT_coreReset signals, should be asserted for at least four core clock cycles to guarantee that the PPC401A1 is put into its reset state. System requirements may require these signals to be active for additional cycles. See Figure 3-4 for a suggested reset implementation.

3.7.5.4 EXT_coreReset (input)

Core Name: **XRSTCOR**

This core input *must* be asserted to reset the PPC401A1, including the data and instruction caches. The core reset can be generated from sources outside of the core, or as a response

to the core's appropriate reset request output signals. The assertion of any higher level reset at the EXT_chipReset or EXT_sysReset inputs should also cause the EXT_coreReset signal to be asserted. This signal should be asserted for at least four core clock cycles to guarantee that the PPC401A1 is put into its reset state. See Figure 3-4 for a suggested reset implementation.

3.7.5.5 EXT_chipReset (input)

Core Name: **XRSTCHP**

This core input should be asserted when the Core+ASIC chip is reset. The chip reset can be generated from sources outside of the core, or as a response to the core's appropriate reset request output signals. The assertion of this signal *must* also cause the EXT_coreReset input to the core to be asserted. The assertion of a higher-level reset at the EXT_sysReset input should also cause the EXT_chipReset and EXT_coreReset signals to be asserted.

This chip reset does not cause additional reset activity in the core, beyond that accomplished by a core reset. It is only used for status, to identify the most recent reset type. This signal should be asserted for at least four core clock cycles to guarantee that the PPC401A1 is put into its reset state. ASIC logic may require these signals to be held active for additional cycles. See Figure 3-4 for a suggested reset implementation.

3.7.5.6 EXT_sysReset (input)

Core Name: **XRSTSYS**

This core input should be asserted when the *System*, including the Core+ASIC chip, is reset. System reset is required at POR to reset the JTAG port internal to the processor core. The system reset can be generated from sources outside of the core, or as a response to the core's appropriate reset request output signals. The assertion of this signal *must* also cause the EXT_chipReset and EXT_coreReset inputs to the core to be asserted. This signal should be asserted for at least four core clock cycles to guarantee that the PPC401A1 is put into its reset state. System requirements may require that these signals to be active for additional cycles. See Figure 3-4 for a suggested reset implementation.

If the system reset is driven from an external source, the source must follow the system reset guidelines in Chapter 4, "Initialization." In general, the system reset signal driven from an external source is first synchronized to the system clock and then presented to the remainder of the chip hardware as a synchronous reset signal. A minimum number of cycles may be required on the system reset signal to allow the synchronization to the system clocks on the chip and to ensure a proper startup state for certain components of the chip.

3.7.5.7 IOS_jtagResetIn (input)

Core Name: **JTRSTIN**

See Section 3.12.3.4 on p. 3-105 for a description of IOS_jtagResetIn.

3.8 Instruction-side Bus Interface Unit (BIU)Interface

The instruction-side PLB interface enables the instruction cache unit (ICU) to read instructions from memory. The ICU cannot write to memory across the PLB interface. This interface has a dedicated 30 bit address bus output, `ICU_biuABus(0:29)`, and a dedicated 32 bit read data bus input, `BIU_icuRdDBus(0:31)`.

Engineering Note: The instruction-side read data bus can be combined with the data-side read data bus at the chip level to create a shared read data bus.

The address bus designates the target instruction to be read. The cacheability and guardedness storage attributes of the address are indicated by the `ICU_biuWordNLine` and `ICU_biuGuardedMem` signals respectively. This ICU interface to the PLB (or custom BIU) is capable of one data transfer every cycle.

For cache-inhibited accesses, one instruction (one word) is transferred. When BIU collects the instruction, it should place entire word on the read data bus. For cache-inhibited reads, the `BIU_icuWdAddr(0:1)` bits are ignored by the ICU.

For cacheable accesses, a cache line (four instruction words) is expected to be transferred, one word at a time. Wait states can be inserted as needed to allow the BIU to match the transfer rate of the external bus, or to pack the bytes of the instruction from a non-word device to form a full word. See Section 3.9.4, “Data-Side BIU Interface Timing Diagrams” for detailed timing diagrams regarding transfers across the ICU interface to the BIU.

For a cacheable read, the four words of the cache line containing the target word can be transferred to the ICU in target-word-first, sequential, or any other order. When the target instruction is transferred to the ICU, the ICU forwards the instruction to the instruction queue and concurrently sends it to the ICU cache array. Therefore, the BIU should provide target-word-first delivery whenever possible. The order of the delivery of the words for a cache line fill is determined by the BIU design and should be indicated by the `BIU_icuWdAddr(0:1)` signals.

3.8.1 Instruction-Side BIU Interface I/O Symbol

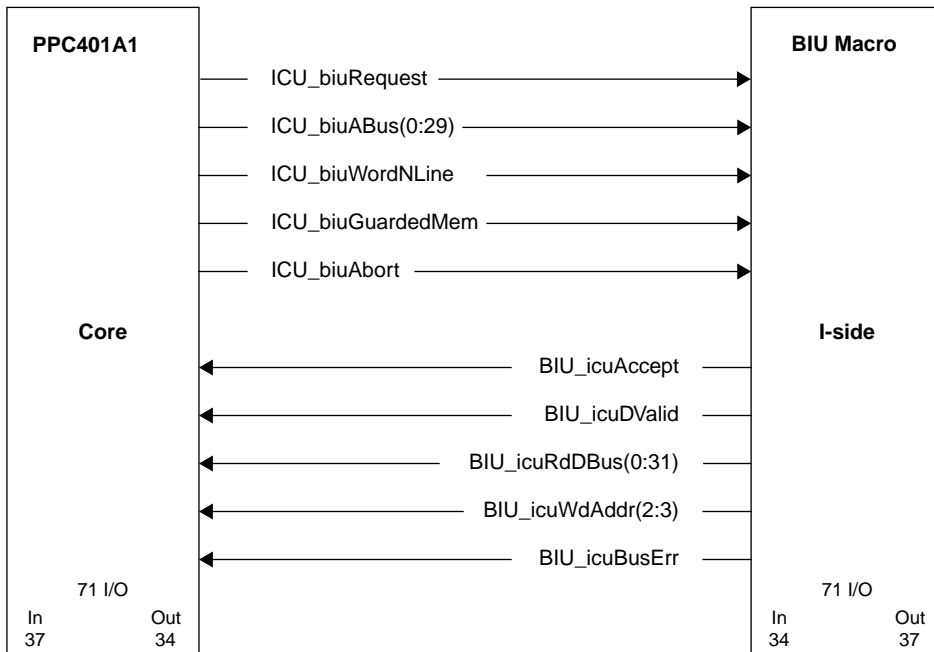


Figure 3-6. Instruction-Side BIU Interface I/O Symbol

3.8.2 Instruction-Side BIU I/O Signal Table

Table 3-10 summarizes the instruction-side BIU interface signals, which are described in detail in text following the table.

Table 3-10. Instruction-Side BIU Interface Signal Summary

Signal	I/O Type	If Unused	Function)
ICU_biuRequest	O	No connect	Requests an instruction fetch transfer across the BIU_icuRdDBus(0:31)
ICU_biuABus(0:29)	O	No connect	Indicates the address of the memory which is being accessed for a read transfer with the ICU.
ICU_biuWordNLine	O	No connect	Indicates the size of the transfer being requested as either a single word cache-inhibited transfer or a four word cache line transfer.

Table 3-10. Instruction-Side BIU Interface Signal Summary

Signal	I/O Type	If Unused	Function)
ICU_biuGuardedMem	O	No connect	Indicates whether the requested instruction is in a storage region marked as guarded by the G storage attribute
ICU_biuAbort	O	No connect	Indicates that the current ICU transfer request is being aborted.
BIU_icuAccept	I	0	Indicates that the current transfer being requested by the ICU is being accepted by the BIU.
BIU_icuDValid	I	0	Indicates that a requested instruction word is being presented on the BIU_icuRdDBus(0:31)
BIU_icuRdDBus(0:31)	I	0x00000000	The 32-bit memory-aligned data bus (ICU read data bus) used to transfer instructions from the BIU to the ICU.
BIU_icuWdAddr(0:1)	I	0b00	During cacheable transfers, indicates which word of a four word cache line is being transferred across the ICU read data bus.
BIU_icuBusErr	I	0	Indicates that an error was detected by the BIU/PLB during the transfer of a word associated with a ICU request. This signal <i>must</i> be presented with BIU_icuDValid

3.8.3 Instruction-Side BIU Interface I/O Signal Descriptions

The following subsections describe the instruction-side BIU interface signals.

Each subsection heading names a signal and give its I/O type. Each subsection heading also provides the corresponding hard macro signal name, referred to as the core name.

3.8.3.1 ICU_biuRequest (output)

Core Name: **IKREQST**

This signal is asserted by the ICU to request an instruction fetch across the read data bus. When this signal is active the values on the ICU_biuABus(0:29), ICU_biuWordNLine, and ICU_biuGuardedMem will identify the word address, cacheability, and guardedness attribute of the instruction being requested.

If the request is acknowledged (accepted) by the BIU, the request signal will be deasserted in the cycle following the activation of BIU_icuAccept, unless another request is available. If another request is available, the request line will stay active, but the request identifiers (address bus, cacheability, and guardedness signals) change as necessary for the new request.

The BIU must latch the request identifiers at the end of the same cycle that it accepts the request. It is possible for the BIU to accept the request the same cycle that the request is presented.

The ICU can present a second request before the last accepted request has been fully satisfied. The BIU can accept the second request in the same manner that any other request is accepted. If the BIU does accept the second request before the first is fully satisfied, it must complete the return of data for the first request before it can send the data associated with the second request. The BIU must maintain in-order servicing of accepted requests.

If a request is not accepted, and the ICU no longer requires the instruction word whose address first appeared with the request, the ICU can assert the ICU_biuAbort signal to remove the outstanding request from the interface. In the cycle following the assertion of the abort, a new request may be asserted, with the request identifiers changing as necessary for the new request. The request signal itself need not be deasserted between an aborted request and a new one. If an outstanding request is being accepted by the BIU in the same cycle that it is being aborted by the ICU, the BIU must abort the transfer. It may not transfer data associated with the aborted request across the read data bus.

This interface can transfer data (and therefore make new requests) every cycle, as demonstrated in the timing diagrams in Section 3.8.4, “Instruction-Side BIU Interface Timing Diagrams”.

3.8.3.2 ICU_biuABus(0:29) (output)

Core Name: **IKADRnn**

This 30-bit address bus is driven by the ICU and indicates the word address of the target instruction being requested by the ICU. The cacheability of this address is indicated on the ICU_biuWordNLine signal. This signal is valid whenever the ICU_biuRequest signal is asserted. The BIU should latch the value on this address bus at the end of the same cycle that it accepts the request.

Once a request is presented, the request identifiers will retain their values until the cycle after the request is accepted by the BIU or the cycle after the request is aborted by the ICU. At that time, they may change to describe the next request, if one is indicated by the assertion of the ICU_biuRequest signal.

3.8.3.3 ICU_biuWordNLine (output)

Core Name: **IKWORDX**

This signal is driven by the ICU to indicate whether the instruction being requested on ICU_biuABus(0:29) is cache-inhibited or cacheable. This signal will be valid any time that the ICU_biuRequest signal is asserted. The BIU should latch this signal at the end of the same cycle that it accepts the request.

An ICU_biuWordNLine high value (logic 1) indicates to the BIU that the addressed instruction is cache-inhibited, either because the address is to a non-cacheable region, or because the instruction cache array is disabled or otherwise unavailable for this address. Because the address is cache-inhibited, the request is to read a single instruction (word) at the target address. For a cache-inhibited transfer, the BIU_icuWdAddr(0:1) signals are ignored when the instruction word is transferred to the ICU.

An ICU_biuWordNLine low value (logic 0) indicates to the BIU that the addressed instruction is cacheable. Therefore, the request is to read the cache line (four instructions) which

contains the target instruction. During a cache line fill, the four words of the cache line containing the target word can be transferred to the ICU in target-word-first, sequential, or any other order. When the target instruction is transferred to the ICU, the ICU forwards the instruction to the instruction queue and concurrently sends it to the ICU cache array. Therefore, the BIU should provide target-word-first delivery whenever possible. The order of the delivery of the words for a cache line fill is determined by the BIU design and is indicated by the BIU_icuWdAddr(0:1) signals.

After a request is presented, the request identifiers retain their values until the cycle after the request is accepted by the BIU, or the cycle after the request is aborted by the ICU. At that time, the request identifiers may change to describe the next request, if one is indicated by the assertion of the ICU_biuRequest signal.

3.8.3.4 ICU_biuGuardedMem (output)

Core Name: **IKGUARD**

This signal indicates that the requested instruction fetch is from a storage region marked as guarded. This signal is valid whenever the ICU_biuRequest signal is asserted. The BIU should latch this signal at the end of the same cycle that it accepts the request.

After a request is presented, the [request identifiers](#) retain their values until the cycle after the request is accepted by the BIU or the cycle after the request is aborted by the ICU. At that time, the [request identifiers](#) may change to describe the next request, if one is indicated by the assertion of the ICU_biuRequest signal.

This signal can be left unconnected on the ICU interface, and is not implemented in subsequent PPC4xx core designs. The PowerPC Architecture allows prefetching of instructions to the next 1KB boundary, and further on, even to the next 1KB boundary, even in guarded memory, which must be “well-behaved.” This simply means that a system designer should organize the memory so that architecturally allowed prefetching in instruction space does not cause problems in the address space. Therefore, a design that complies with the PowerPC architecture does not benefit from the G storage attribute on the instruction side.

3.8.3.5 ICU_biuAbort (output)

Core Name: **IKABORT**

This abort request signal is asserted if the ICU decides that it no longer requires the instruction transfer that it is requesting. This signal is valid ONLY when the ICU_biuRequest signal is active (it should be ignored otherwise), and may only be used to abort a request which has not been, or is currently being, accepted. During the cycle following the cycle of an aborted request, the ICU will either deactivate its request signal or make a new request by changing the request identifiers - the address bus, cacheability, and guardedness attribute signals.

If the abort signal becomes active in the cycle that the BIU_icuAccept signal also becomes active, the BIU is responsible to ensure that the transfer DOES NOT proceed any further. This implies that no BIU_icuDValid signals can be sent for an aborted ICU read operation. (There are no write operations from the ICU to the BIU.)

It is allowable for the abort signal to become active after the accept, while the BIU is servicing an accepted (and therefore, NOT aborted) request. This occurs:

- If the ICU_biuRequest line is inactive, the abort signal is to be ignored, since it only has meaning while the request signal is active. Thus, since the BIU is in the process of performing an accepted request, it must fully complete the accepted operation.
- If the request line is active, then the abort signal pertains to the NEW request, which will be removed or replaced by the ICU unit in the next cycle. Thus, since the BIU is in the process of performing an accepted request, it must fully complete the accepted operation.

The valid value of ICU_biuAbort signal arrives late in the cycle. Therefore, in the BIU, this signal must have a minimal amount of set-up time to be latched.

3.8.3.6 BIU_icuAccept (input)

Core Name: **IKACCP**

This signal is asserted by the BIU in response to an ICU request to indicate that the request for an instruction fetch has been acknowledged (or accepted). The BIU should latch the address bus, cacheability, and guardedness attribute signals at the end of the cycle that it presents the BIU_icuAccept signal. The accept signal is asserted for one cycle only, once per request, and must precede the instruction transfer across the read data bus. It is permissible for the BIU to accept the request in the same cycle that the request is presented.

The ICU can present a new request before the last accepted request has been fully satisfied. The BIU can accept the new request in the same manner that any other request is accepted. If the BIU does accept the second request before the first is fully satisfied, it must complete the return of data for the first request before it can send the data associated with the second request. The BIU must maintain in-order servicing of accepted requests. The ICU cannot present an additional request until the first of two accepted requests is completely satisfied (that is, the cycle after the last BIU_icuDValid for the first accepted transfer.)

3.8.3.7 BIU_icuDValid (input)

Core Name: **IKDVAL**

This ICU data valid signal is asserted by the BIU to indicate to the ICU that an instruction word of the ICU request is available on the ICU read data bus, BIU_icuRdDBus(0:31). This signal will be active one cycle per word transferred. The ICU will latch the instruction from the ICU read data bus at the end of the cycle in which BIU_icuDValid is asserted.

For a cache-inhibited transfer, the BIU_icuDValid signal will be asserted for one cycle, as the instruction word is presented across the ICU read data bus.

For a cacheable transfer, all four words of the cache line that contains the target instruction will be transferred, one word at a time, with the BIU_icuDValid signal being asserted once for each word, as it is presented across the read data bus. For cacheable accesses, the BIU_icuWdAddr(0:1) signals must indicate which word of the cache line is currently being transferred across the read data bus.

There may be any number of wait states between data transfers on the read data bus, allowing the BIU to pack data from a non-word device until a full word is collected, and/or to match the data transfer rate with the device on the external bus. If there are zero wait states on the external bus and the device is a word device, then the BIU_icuDValid can remain on for four consecutive cycles, each cycle presenting a different word of the line fill.

3.8.3.8 BIU_icuRdDBus(0:31) (input)

Core Name: **IKDInn**

This bus, also referred to as the ICU read data bus, is a 32-bit memory-aligned data bus which is used to transfer instructions from the BIU to the ICU. The data on this bus is valid when it is qualified by BIU_icuDValid. The data valid signal indicates to the ICU that the instruction associated with the read transfer in progress is available on the ICU read data bus, and that it must be latched at the end of the current cycle. Data is always presented across the ICU read data bus as a word.

If the transfer is a cache-inhibited ICU read, the request is always for one instruction word. All four bytes of the BIU_icuRdDBus(0:31) must be valid.

For a cacheable transfer, all four words of the cache line that contains the target instruction are transferred, one word at a time, with the BIU_icuDValid signal being asserted once for each word, as it is presented across the read data bus. For cacheable accesses, the BIU_icuWdAddr(0:1) signals must indicate which word of the cache line is currently being transferred across the read data bus. All bytes on the BIU_icuRdDBus(0:31) must be valid for each word transfer.

3.8.3.9 BIU_icuWdAddr(0:1) (input)

Core Name: **IKWADrn**

These ICU word address signals are used when the words associated with a cacheable request are being sent from the BIU back to the ICU. They are used to indicate which word of the four word cache line is currently being transferred across the BIU_icuRdDBus(0:31), as shown in Table 3-17. These signals are valid when qualified by BIU_icuDValid.

Table 3-11. BIU_icuWdAddr(0:1) Decode

BIU_icuWdAddr(0:1)	Decode
00	Word 0
01	Word 1
10	Word 2
11	Word 3

For cache-inhibited read transfers, the ICU disregards the BIU_icuWdAddr(0:1) signals.

For cacheable read transfers, the ICU uses these signals to write the instruction to the appropriate word location within the cache line. If the CPU is waiting for the target instruction, they are also used to forward the target word to the instruction queue. The four words of the cache line containing the target word can be transferred to the caches in target-word-first, sequential, or any other order. The order of the delivery of the words for a cache line fill is determined by the BIU design. When the target instruction is transferred to

the ICU, the ICU forwards the instruction to the instruction queue and concurrently sends it to the ICU cache array. Therefore, the BIU should provide target-word-first delivery whenever possible.

3.8.3.10 BIU_icuBusErr (input)

Core Name: **IKBERR**

This signal is driven by the BIU to indicate that it has detected an error while attempting to transfer the current word of the ICU request being serviced. This signal may only be asserted while the BIU_icuDValid signal is active. The presence of BIU_icuBusErr does not terminate a cacheable transfer. Any remaining BIU_icuDValid signals are still required to complete the transfer.

If an error is encountered during the transfer of any portion of a word that is to be sent back to the ICU, the error signal must be asserted during the cycle that BIU_icuDValid is presented for that word. When a target word is sent to the fetcher, the error indication for that word travels with it. If the target word has an error associated with it and the CPU attempts to execute this instruction, an instruction machine check exception occurs, causing a machine check interrupt to be taken, if enabled. However, if the CPU never attempts to execute this instruction (perhaps it was speculatively fetched), no machine check exception occurs.

For a cache-inhibited transfer which has an error, the error signal should simply be asserted with the BIU_icuDValid signal when the instruction word is being sent to the ICU.

For a cacheable transfer, the error signal is sent on a per-word basis, as it applies to the word currently being sent to the ICU (as identified by BIU_icuWdAddr(0:1)), during its BIU_icuDValid cycle. The error signal travels with the target instruction word into the fetcher.

Cacheable transfers cause a cache line fill as the words are transferred to the ICU. As each word is transferred to the ICU, it is written into the ICU cache array. Target words are forwarded to the instruction queue, along with any corresponding error indication for that word. If a word of a cache line contains an error, the line, although written into the cache array, is invalidated. If any target word sent to the fetcher has an error associated with it and the CPU attempts to execute it, a machine check exception occurs, causing a machine check interrupt to be taken, if enabled.

For errors such as protection checks or non-configured checks, the error signal should be asserted for each word transferred since the data is invalid for all four words of the line fill. For errors such as external bus errors and BIU timeout checks, the error signal should be asserted with each word to which they apply.

3.8.4 Instruction-Side BIU Interface Timing Diagrams

The timing diagrams in this section represent typical scenarios for the transfers which may occur on the PPC401A1 ICU-PLB interface when connected to a customer-specific BIU or to the IBM Core+ASIC processor local bus (PLB) soft macro.

The ICU interface can only perform reads to access instructions across the ICU/PLB interface.

3.8.4.1 ICU/BIU Timing Diagram Guidelines

In order to make the timing diagrams as concise as possible, the following assumptions (numbered items) were made for each diagram, unless explicitly stated otherwise in the description for that diagram.

1. The timing diagrams generally show the best achievable timing relationships for the indicated transfer sequences.
2. Requests are acknowledged (via PLB_icuAddrAck) by the PLB/Custom BIU in the same cycle in which they are presented on the interface.

Requests may be acknowledged by the PLB in the same cycle that the ICU presents the request, or during a subsequent cycle if it is currently busy doing other work.

3. The PLB_icuRdDack (read data acknowledge) signal is asserted in the cycle immediately following the acknowledge of a read request.

For a given read request, the earliest cycle that a custom BIU may assert the PLB_icuRdDack signal to return data is the cycle immediately following the acknowledge for that request.

The earliest cycle that a PLB-Compliant BIU (such as the IBM PLB soft macro) may present the PLB_icuRdDack signal is during the second cycle following the acknowledge for that request.

There may be any number of additional cycles between the acknowledge cycle and the first PLB_icuRdDack assertion, or between PLB_icuRdDack assertions for a cacheable transfer, to allow the BIU to gather data from a slow or non-word device. The PPC401A1 will wait for the assertion of the appropriate number of PLB_icuRdDack signals to satisfy the request.

4. All transfers assume that the BIU is able to access an entire word of data per cycle.

The transfer size for the ICU/PLB interface is a (four byte instruction) word. When the PPC401A1 makes a non-cacheable read request, it expects the request to be satisfied by one transfer across the ICU/PLB interface. The BIU may need to pack data from a byte device into a single word transfer across the ICU/PLB interface.

5. All cacheable reads assume target word first delivery then sequential transfer of the remainder of the line, wrapping as necessary to lower word addresses of the same line.

Target word *first* delivery allows the instruction to be passed to the fetcher as soon as possible, aiding performance by getting the target instruction into the execution pipeline more quickly. Unlike the DCU, target word *last* delivery will not force an additional cycle delay for the next assertion of the request signal (ICU_plbRequest) on the interface.

6. It is assumed that instruction execution is sufficiently fast to allow the fetcher to make an instruction fetch request to the ICU each cycle. In other words, the ICU is presenting requests as fast as it can, limited only by its own logic and the PLB/BIU bus protocols.
7. Key to Timing Diagram Abbreviations

Table 3-12. Key to ICU/PLB Timing Diagram Abbreviations

Abbreviation	Description	Where Used
c#	Cacheable request identifier	Request, AddrAck, RdDAck
nc#	Non-cacheable request identifier	Request, AddrAck, RdDAck
adr#	Address for identified (#) request	ABus(0:29)
d#	Data associated with a non-cacheable request or, as shown in Figure 3-7, cycles 3–6 and 9–12, cacheable request data, organized by word address	RdDBus(0:31)
d# _#	Data associated with a cacheable request	RdDBus(0:31)
Subscripts	Used to identify a particular word of a 4 word line	RdDAck, RdDBus(0:31)
#	Used to identify a particular word of a 4 word line	RdWdAddr(2:3)

3.8.4.2 ICU/BIU Back-to-Back Cacheable Fetches (Custom BIU)

Figure Highlights

- Shows the fastest that back to back cacheable requests can be made to a custom BIU
- Code flow is sequential, starting at address 'adr3' (the last word of a line) and continuing on through address 'adr7'
- The "ICU/BIU Timing Diagram Guidelines" on page 3-35 apply.

The cycle 1 cache miss determination for adr3 causes the PLB request of cycle 2, which is immediately acknowledged (via AddrAck) by the BIU. The instruction for adr3 (d3) is transferred in cycle 3, as indicated by the BIU_icuWdAddr(0:1) signals, and bypassed (note the *byp designation on the Request row) by the ICU to the CPU's instruction fetcher as it is [concurrently] written into the I-cache. The next CPU fetch address is adr4. The ICU recognizes that adr4 is not a part of the adr3 cache line, and the remainder of the cache line fill proceeds through cycle 6, with the ICU simply receiving the data and writing it into the I-cache, without bypassing data to the fetcher.

Since the I-cache is a blocking cache, the cache miss determination for adr4 is not made until cycle 7. This causes the PLB request of cycle 8, which is immediately acknowledged by the BIU. The instruction for adr4 (d4) is transferred in cycle 9 and bypassed to the fetcher as it is placed in the I-cache. As the remainder of the I-cache line fill operation proceeds across the interface in cycles 10 through 12, the ICU also fulfills the fetcher's sequential requests for adr5, adr6, and adr7, bypassing the incoming data to the fetcher as it receives it, recognizing each address match through the use of its line fill address for adr4 and the BIU_icuWdAddr(0:1) signals.

A PLB-compliant BIU cannot return RdDAck until the second cycle following the assertion of AddrAck. In the diagram below, the first Rd DAck would push out to cycle 4 and the "c2" request would occur in cycle 9.

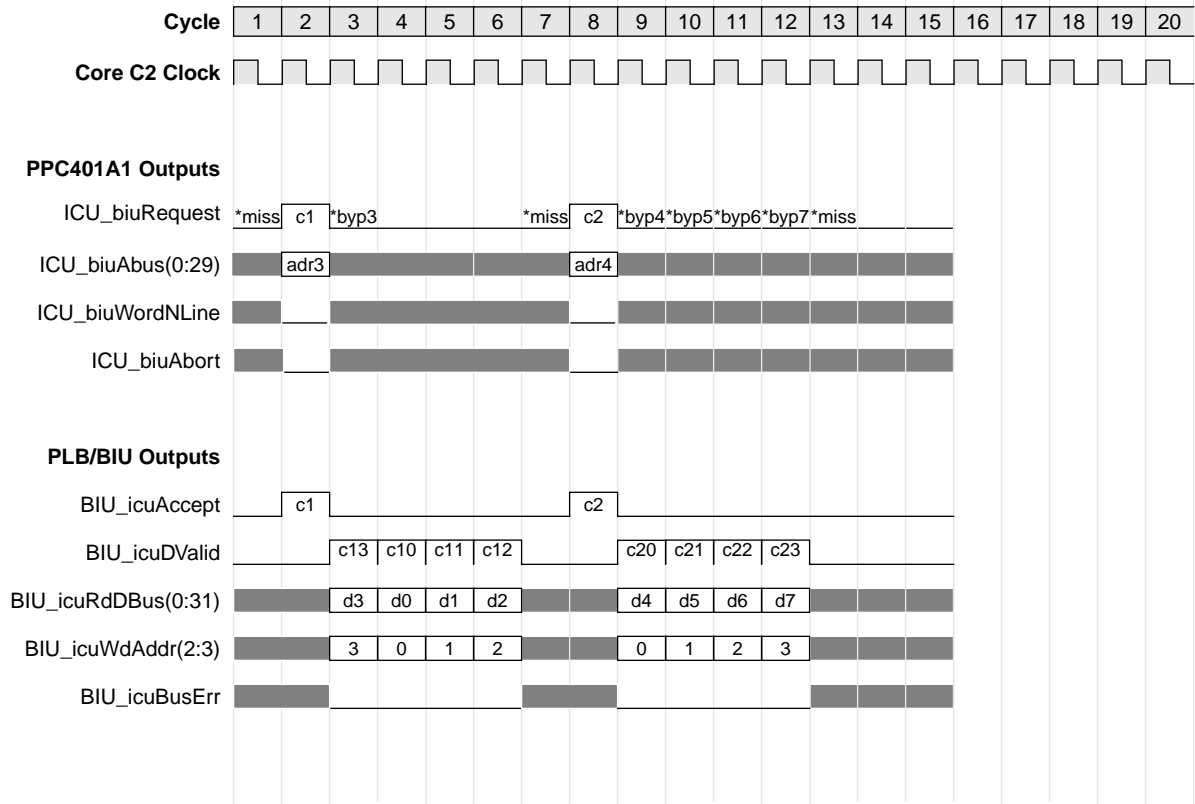


Figure 3-7. ICU/BIU Back-to-Back Cacheable Fetches

3.8.4.3 ICU/BIU Fetch Cacheable-NonCacheables-Cacheable Mix (Custom BIU)

Figure Highlights

- Non-cacheable requests can immediately follow a cacheable or non-cacheable request.
- A string of non-cacheables requests can be requested consecutively from a custom BIU.
- Shows the fastest that a non-cacheable request can be followed by a cacheable request to a custom BIU.
- The “ICU/BIU Timing Diagram Guidelines” on page 3-35 apply.

The cacheable PLB request of cycle 2 is acknowledged (via AddrAck) by the BIU in the same cycle. The ICU is capable of presenting a non-cacheable request in the next cycle (3) which is also immediately acknowledged. The PLB/Custom BIU (CBIU) can acknowledge a second request but must satisfy the requests in the order in which they were made. Since the ICU cannot have more than two outstanding requests, it will not make another request until the first request is fully satisfied. Therefore, when the fourth RdDack ($c1_3$) of the four word cache line is detected in cycle 6, a new non-cacheable request (nc3) may be presented in cycle 7 - and is immediately acknowledged by the BIU. Also in cycle 7, the RdDack of the “nc2” request is detected, and another non-cacheable request (nc4) is made in cycle 8. For a custom BIU (non PLB-compliant) which is capable of providing same cycle AddrAck and next cycle RdDack, a string of non-cacheable requests can be made each cycle, as shown in cycles 7 through 11.

The earliest that a cacheable request can follow a non-cacheable request for a custom BIU is shown in cycles 11 through 14. These timing relationships are non-PLB compliant transfers in that the RdDack signals begin one cycle earlier than a PLB-compliant design can return them.

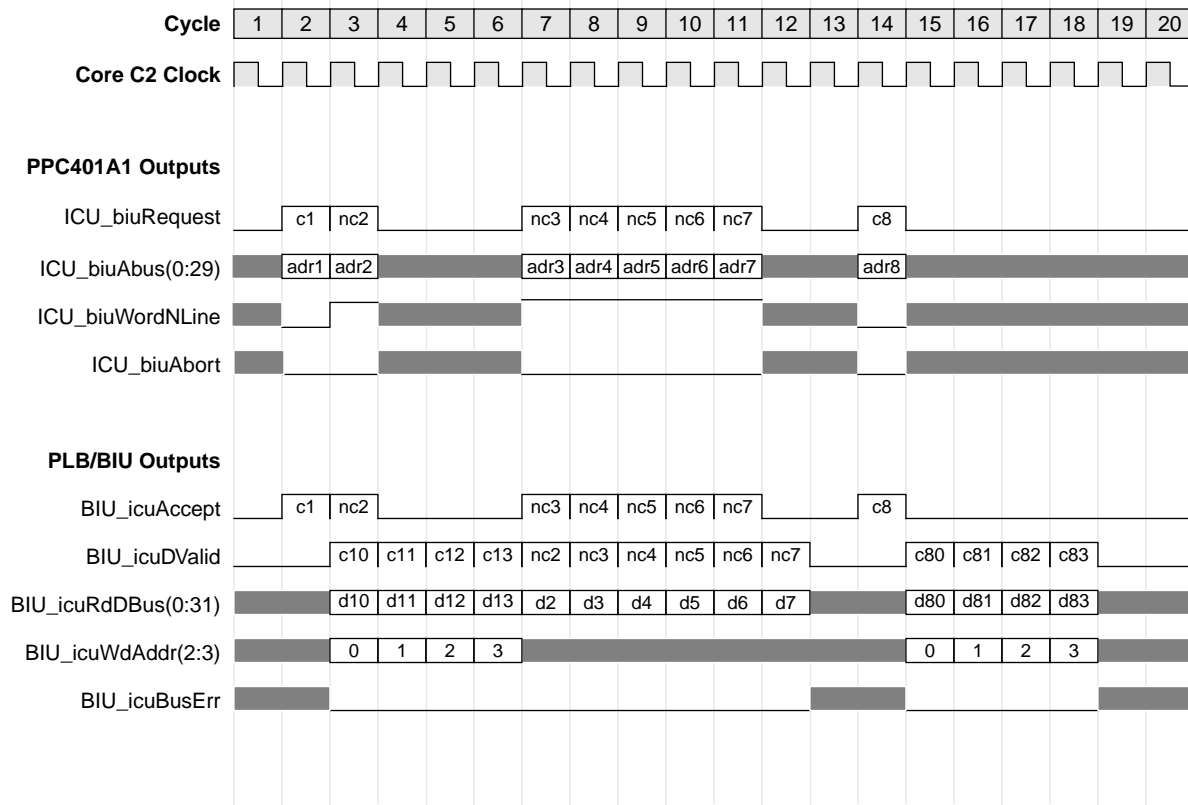


Figure 3-8. ICU/BIU Fetch Cacheable-NonCacheables-Cacheable

3.8.4.4 ICU/BIU-PLB Fetch NonCacheables to Cacheable (PLB-Compliant)

Figure Highlights

3

- Non-cacheable request can immediately follow another non-cacheable request.
- A string of non-cacheable requests cannot be sustained with a PLB-compliant BIU.
- Shows the fastest that a non-cacheable request can be followed by a cacheable request to a PLB-compliant BIU.
- The “ICU/BIU Timing Diagram Guidelines” on page 3-35 apply, except that the RdDAck behavior is PLB-compliant (See Item 3.).

The non-cacheable PLB request of cycle 2 is acknowledged (by AddrAck) by the PLB-compliant BIU in the same cycle. The ICU is capable of presenting a non-cacheable request in the next cycle (3) which is also immediately acknowledged. The PLB can acknowledge a second request but must satisfy the requests in the order in which they were made. Since the ICU cannot have more than two outstanding requests, it will not make another request until the first request is fully satisfied. Therefore, when the RdDAck of the “nc1” request is detected in cycle 4, a new non-cacheable request (nc3) may be presented in cycle 5 - and is immediately acknowledged by the BIU. Also in cycle 5, the RdDAck of the “nc2” request is detected, and another non-cacheable request (nc4) is made in cycle 6. For a PLB-compliant BIU a string of non-cacheable requests will follow the pattern of cycles 2 through 7.

The earliest that a cacheable request can follow a non-cacheable request for a PLB compliant BIU is shown in cycles 9 through 13. Notice that this is one cycle more than the non-PLB-compliant case shown in Figure 3-8.

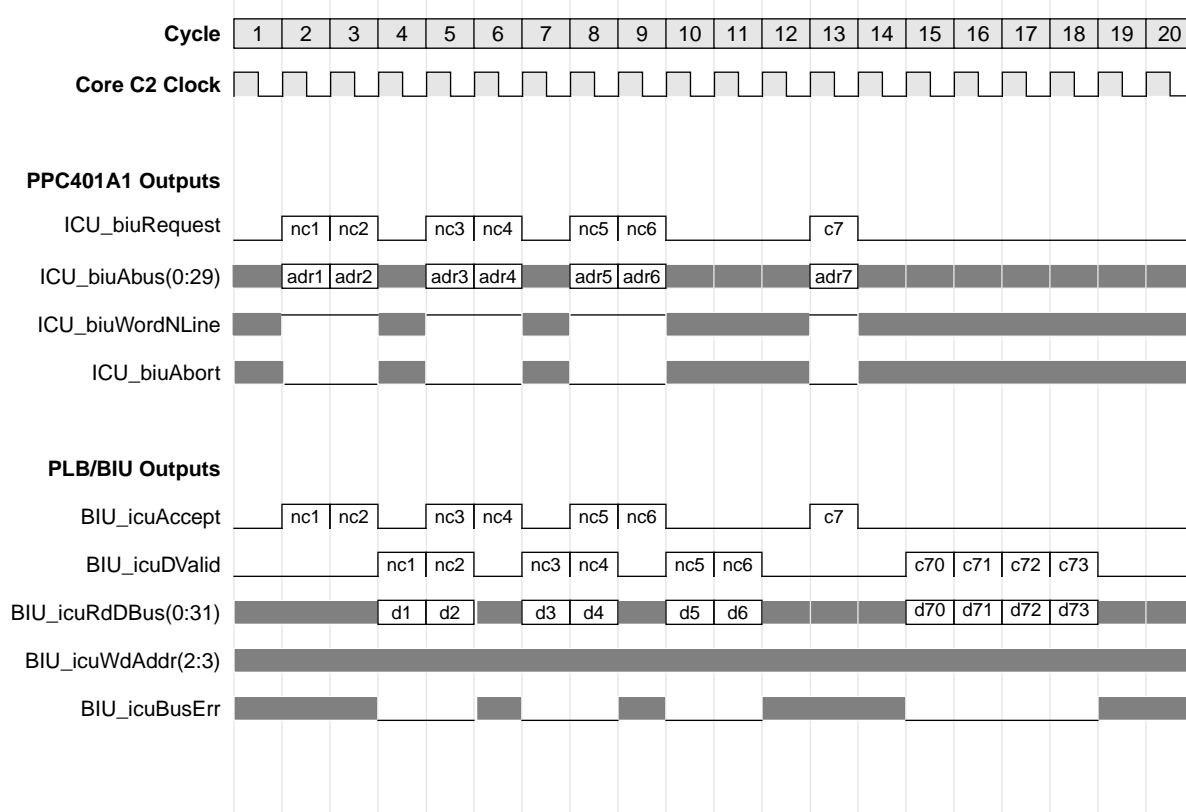


Figure 3-9. ICU/BIU-PLB Fetch NonCacheables to Cacheable

3.9 Data-Side Bus Interface Unit (BIU) Interface

The data-side BIU interface enables the data cache unit (DCU) to load and store data from and to cacheable and cache-inhibited off-chip memory. The interface has a dedicated 32 bit address bus output, a dedicated 32 bit write data bus output, and a dedicated 32 bit read data bus input.

Engineering Note: The data-side read data bus may be combined with the instruction-side read data bus at the chip level to create a shared read data bus.

The address bus designates the target word address of the data to be read/written. The cacheability and guardedness of the address are indicated by the `DCU_biuBytesNLine` and `DCU_biuGuardedMem` signals respectively. A byte enable bus (a set of four data tags) specifies which bytes of a target data word are being requested to be loaded or stored. The byte enables (data tags) can specify one, two, three, or four contiguous bytes. The BIU packs or unpacks data to or from the DCU so that the information sent across the appropriate DCU data bus is presented a word at a time.

For cache-inhibited data reads, the address bus designates the target word address and the data tags, or byte enables, indicate which bytes of the target word are read. After the data has been collected by the BIU, it places those bytes in the corresponding byte positions on the read data bus. The values of any remaining bytes of the data word on the read data bus not requested by the data tags are don't cares. For cache-inhibited reads, the `BIU_dcuWdAddr(0:1)` bits are ignored by the DCU. If the target address is in a guarded storage region (`DCU_biuGuardedMem = 1`), only the specific bytes designated by the `DCU_biuDTags(0:3)` signals can be read by the BIU.

For cache-inhibited data writes, the data word is placed on the write data bus. The address bus designates the target word address, and the data tags indicates which bytes of the target word are to be written with the corresponding data from the write data bus. On a cache-inhibited write, only those bytes designated by the `DCU_biuDTags(0:3)` signals can be written by the BIU.

For cacheable accesses, an entire cache line (four data words) is transferred, one word at a time. For a cacheable transfer, the BIU should disregard the data tags, because the entire cache line is being read/written. Wait states can be inserted as needed to allow the BIU to match the transfer rate of the external bus, to pack the bytes of the data word from a non-word device (read operation), or to unpack the bytes of the data word to a non-word device (write operation). See Section 3.9.4, "Data-Side BIU Interface Timing Diagrams," for detailed timing diagrams describing transfers across the DCU interface to the BIU.

For a cacheable data read, the four words that comprise the cache line containing the target word may be transferred to the DCU in any order (target-word-first, sequential, other). The order of the delivery of the words for a cache line fill is determined by the BIU design and is indicated by the `BIU_dcuWdAddr(0:1)` signals. When the target data is transferred to the DCU, the DCU forwards the data to the load/store unit, allowing the CPU pipeline to advance, and concurrently sends the to the DCU cache array. Therefore, the BIU should provide target-word-first delivery whenever possible.

For a cacheable data write, the four words that comprise the cache line containing the target word will be transferred across the DCU write data bus in ascending word address order, word0 to word3.

3.9.1 Data-Side BIU Interface I/O Symbol

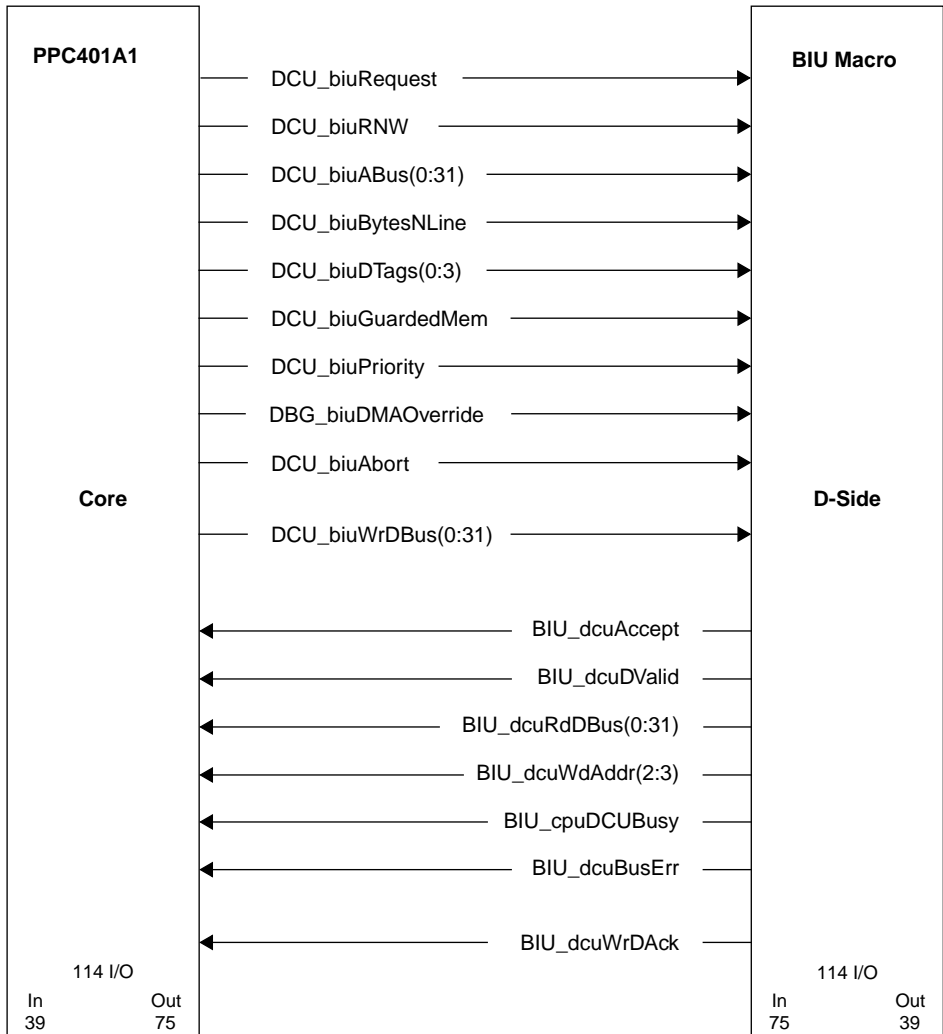


Figure 3-10. Data Side BIU Interface I/O Symbol

3.9.2 Data-Side BIU Interface I/O Signal Table

Table 3-13 summarizes the Data-Side BIU interface signals, which are described in detail in text following the table.

Table 3-13. Data-Side BIU Interface I/O Signal Summary

Signal	I/O Type	If Unused	Function
DCU_biuRequest	O	No connect	Requests a data transfer between the DCU and BIU
DCU_biuRNW	O	No connect	Indicates whether the requested transfer is a read or a write transfer
DCU_biuABus(0:31)	O	No connect	Indicates the address of the memory which is being accessed for a read or write data transfer with the DCU.
DCU_biuBytesNLine	O	No connect	Indicates the size of the data being requested as a one to four byte cache-inhibited word xfer or a four word cache line transfer.
DCU_biuDTags(0:3)	O	No connect	Indicate which bytes in a word are to be transferred during a cache-inhibited transfer; DCU_biuDTags(0:3) are ignored for cacheable transfers
DCU_biuGuardedMem	O	No connect	Indicates whether a requested data transfer is to/from a storage region marked as guarded by the G (Guarded) storage attribute
DCU_biuPriorityDCU_plbPriority	O	No connect	Indicates that the current DCU request is a high priority request because the CPU pipeline is stalled, waiting for the data transfer to complete.
DBG_biuDMAOverride	O	No connect	Indicates that the DCU request should be given highest priority. Enables RISCWatch to gain access to memory across the DCU-BIU interface.
DCU_biuAbort	O	No connect	Indicates that the current DCU transfer request is being aborted.
DCU_biuWrDBus(0:31)	O	No connect	The 32-bit memory-aligned data bus (a.k.a. DCU write data bus) used to transfer write data from the DCU to the BIU.
BIU_dcuAccept	I	0	Indicates that the current transfer being requested by the DCU is being accepted by the BIU.
BIU_dcuDValid	I	0	Indicates that a data word associated with a read request is being presented on the BIU_dcuRdDBus(0:31)
BIU_dcuRdDBus(0:31)	I	0x00000000	The 32-bit memory-aligned data bus (DCU read data bus) used to transfer read data from the BIU to the DCU.
BIU_dcuWdAddr(0:1)	I	0b00	During cacheable transfers, indicates which word of a 4-word cache line is being transferred across the DCU read data bus.

Table 3-13. Data-Side BIU Interface I/O Signal Summary

Signal	I/O Type	If Unused	Function
BIU_dcuWrDack	I	0	Indicates that a data word associated with a write request is being accepted by the BIU from the DCU_biuWrDBus(0:31), and that it will be latched at the end of the cycle.
BIU_cpuDCUBusy	I	0	Indicates that the BIU is busy performing an operation that was initiated by the DCU
BIU_dcuBusErr	I	0	Indicates that an error was detected by the BIU/PLB during the transfer of a data word associated with a DCU request. (Can only be presented while the BIU_cpuDCUBusy signal is asserted.)

3.9.3 Data-Side BIU Interface I/O Signal Descriptions

The following subsections describe the Data-Side BIU I/O signals.

Each subsection heading names the topic signal and give its I/O type. Each subsection heading also provides the hard macro signal cross-reference, referred to as the core name.

3.9.3.1 DCU_biuRequest (output)

Core Name: **DKREQST**

This signal is asserted by the DCU to request a data transfer operation between the DCU and the BIU. When this signal is active, the values on the DCU_biuRNW, DCU_biuABus(0:31), DCU_biuBytesNLine, and and DCU_biuGuardedMem will identify the direction of the transfer, the word address, cacheability, and guardedness of the target data. For a cache-inhibited transfer, the DCU_biuDTags(0:3) will identify which bytes of the addressed data word are to be read/written. For a write operation, the first dataword to be transferred will be placed on DCU_biuWrDBus(0:31), the write data bus, when the request is asserted.

If the request is accepted by the BIU, the request signal will be deasserted in the cycle following the activation of BIU_dcuAccept, unless another request is available. If another request is available, the request line will stay active, but the request identifiers - address bus, cacheability, guardedness, read/write, and data tag signals - will change as necessary for the new request.

The BIU must latch the request identifiers at the end of the same cycle that it accepts the request. It is possible for the BIU to accept the request the same cycle that the request is presented.

In certain situations, the DCU can present a second request before the last accepted request has been fully satisfied. The BIU can accept the second request in the same manner that any other request is accepted. The BIU must service requests of the same direction in the order in which they were received. Also, if the BIU gets a write request to a memory address (or line) and then a subsequent read request to the same memory address

(or line), it must allow the first (write) request to be satisfied such that the second (read) request will return the new data for that address.

If a request has not been accepted and the DCU no longer requires the data transfer associated with the request, the DCU may assert the DCU_biuAbort signal to remove the outstanding request from the interface. In the cycle following the assertion of the abort, a new request may be asserted, with the request identifiers changing as necessary for the new request. The request signal itself need not be deasserted between an aborted request and a new one. If an outstanding request is being accepted by the BIU in the same cycle that it is being aborted by the DCU, the BIU must abort the transfer. It may not transfer data associated with the aborted request across the read data bus (read operation), or to an external device (write operation).

ENGGNOTE: Due to the design of the pipeline, the PPC401A1 does not abort write (store/flush) operations.

See Section 3.9.4, “Data-Side BIU Interface Timing Diagrams,” for detailed timing diagrams regarding transfers across the DCU interface to the BIU.

3.9.3.2 DCU_biuRNW (output)

Core Name: **DKRDNWR**

This signal is used to indicate to the BIU whether the DCU is requesting a read or a write operation. This signal will be valid any time that the DCU_biuRequest signal is asserted. It will be accompanied by the other request identifiers - address bus, cacheability, guardedness, and data tags - to fully specify the request. The BIU should latch this signal at the end of the same cycle that it accepts the request.

If the signal is high (logic “1”), the request is for the BIU to supply data to be read into the DCU. If the signal is low (logic “0”), the request is for the BIU to write data from the DCU to an external device. For a write operation, the first dataword to be transferred will be placed on DCU_biuWrDBus(0:31), the write data bus, when the DCU_biuRequest is asserted.

Once a request is presented, the request identifiers will retain their values until the cycle after the request is accepted by the BIU or the cycle after the request is aborted by the DCU. At that time, they may change to describe the next request, if one is indicated by the assertion of the DCU_biuRequest signal.

3.9.3.3 DCU_biuABus(0:31) (output)

Core Name: **DKADRnn**

This 32-bit DCU Address Bus is driven by the DCU and indicates the byte address of the target data to be read/written for the current request. This bus will be valid any time that the DCU_biuRequest signal is asserted. It will be accompanied by the other request identifiers - cacheability, guardedness, read/write, and data tags - to fully specify the request. The BIU should latch the value on this address bus at the end of the same cycle that it accepts the request.

For non-cacheable requests, if the data tags indicate that the transfer is for multiple bytes, the address will point to the lowest byte address of the bytes being requested.

Once a request is presented, the request identifiers will retain their values until the cycle after the request is accepted by the BIU or the cycle after the request is aborted by the DCU. At that time, they may change to describe the next request, if one is indicated by the assertion of the DCU_biuRequest signal.

3.9.3.4 DCU_biuBytesNLine (output)

Core Name: **DKBYTEX**

This signal is driven by the DCU to indicate whether the data being requested on DCU_biuABus(0:31) is cache-inhibited or cacheable. This signal will be valid any time that the DCU_biuRequest signal is asserted. It will be accompanied by the other request identifiers - address bus, guardedness, read/write, and data tags - to fully specify the request. The BIU should latch this signal at the end of the same cycle that it accepts the request.

A DCU_biuBytesNLine high value (logic '1') indicates to the BIU that the addressed data word is cache-inhibited. Since the address is cache-inhibited, only the bytes specified in the DCU_biuDTags(0:3) should be accessed and transferred to/from the DCU. One to four bytes will be transferred, as indicated by the DCU_biuDTags(0:3).

For a cache-inhibited read operation, when the requested data is gathered by the BIU, it will be presented as a single transfer to the DCU via the DCU read data bus, BIU_dcuRdDBus(0:31).

For a cache-inhibited write operation, the data to be written out will be taken from the write data bus, DCU_biuWrDBus(0:31), and transferred to an external device by the BIU.

A DCU_biuBytesNLine low value (logic '0') indicates to the BIU that the addressed data word is cacheable. Therefore, the request is to read/write the entire cache line (four data words) which contains the target data.

For a cacheable data read, the four words that comprise the cache line containing the target word may be transferred to the DCU in any order (target-word-first, sequential, other). The order of the delivery of the words for a cache line fill is determined by the BIU design and is indicated by the BIU_dcuWdAddr(0:1) signals. When the target data is transferred to the DCU, the DCU will bypass the data to the load/store unit, allowing the CPU pipeline to advance, while concurrently sending it to the D-cache. Therefore, it is advantageous for the BIU to provide target-word-first delivery whenever possible.

For a cacheable data write, the four words that comprise the cache line containing the target word will be transferred across the DCU write data bus in ascending word address order, word0 to word3.

Once a request is presented, the request identifiers will retain their values until the cycle after the request is accepted by the BIU or the cycle after the request is aborted by the DCU. At that time, they may change to describe the next request, if one is indicated by the assertion of the DCU_biuRequest signal.

Table 3-14, “DCU/PLB(BIU) Transfer Type Decoding” shows how the BIU should interpret the DCU_biuRNW and DCU_biuBytesNLine signal combinations.

Table 3-14. DCU/PLB(BIU) Transfer Type Decoding

RNW BytesNLine	Transfer Type
00	One to Four* Byte Write
01	D-cache Line Flush
10	One to Four* Byte Read
11	D-cache Line Fill
*As indicated in DCU_biuDTags(0:3)	

3.9.3.5 DCU_biuDTags(0:3) (output)

Core Name: **DKDTAGn**

These data tag signals, or byte enables, are driven by the DCU. These signals are valid whenever the DCU_biuRequest signal is asserted for a cache-inhibited transfer (DCU_biuBytesNLine = 1). This signal is associated with the other request identifiers, address bus, , guardedness, and read/write, to fully specify the request. The BIU should latch this signal at the end of the same cycle that it accepts the request.

For a cache-inhibited transfer, these data tags identify which bytes of the target word being addressed on DCU_biuABus(0:31) are to be read from or written to, as shown in Table 3-15, “DCU_biuDTags(0:3) Allowed Values”.

For a cache-inhibited read operation, the BIU should place the requested bytes on the BIU_dcuRdDBus(0:31) in the proper memory alignment. If a cache-inhibited read operation is from guarded memory, the BIU may access **ONLY** the bytes explicitly requested by the data tags.

For a cache-inhibited write operation, the BIU can only write out the bytes identified by the [data tags](#) from the DCU_biuWrDBus(0:31) to external devices.

Table 3-15. DCU_biuDTags(0:3) Allowed Values

DCU_biuDTags(0:3)	Transfer Request
0000	Invalid
0001	Byte 3
0010	Byte 2
0011	Halfword 2,3
0100	Byte 1
0101	Invalid
0110	Unaligned Halfword 1,2
0111	Bytes 1,2,3
1000	Byte 0
1001	Invalid

Table 3-15. DCU_biuDTags(0:3) Allowed Values

DCU_biuDTags(0:3)	Transfer Request
1010	Invalid
1011	Invalid
1100	Halfword 0,1
1101	Invalid
1110	Bytes 0,1,2
1111	Word

For a cacheable transfer, these signals must be ignored by the BIU. The [data tags](#) identify the original targetted entity. However, a cacheable transfer implies a transfer of a cache line (four words; all bytes of each word), and the BIU is responsible to transfer all four words of the cache line.

Once a request is presented, the request identifiers will retain their values until the cycle after the request is accepted by the BIU or the cycle after the request is aborted by the DCU. At that time, they may change to describe the next request, if one is indicated by the assertion of the DCU_biuRequest signal.

3.9.3.6 DCU_biuGuardedMem (output)

Core Name: **DKGUARD**

This signal indicates that the data transfer being requested is to an area marked as a guarded area of memory. This signal will be valid any time that the DCU_biuRequest signal is asserted. According to the PowerPC architecture, if a data side transfer is performed to/from an area marked as a guarded area of memory, then the BIU is not permitted to read/write any more data than the data that was explicitly requested by the DCU.

For a non-cacheable request (DCU_biuBytesNLine = '1') with guarded memory, the BIU is only allowed to read/write the bytes specified by the DCU_biuDTags(0:3) for the word specified by DCU_biuABus(0:29).

For a cacheable request (DCU_biuBytesNLine = '0') with guarded memory, the BIU is only allowed to read/write the data line specified by DCU_biuABus(0:27).

If a data transfer being requested is to/from an area marked as an unguarded area of memory, then the BIU may speculatively fetch data in anticipation of a future request by the master, up to the next 1KB boundary.

For chip designs that use the PLB macro, an additional PLB "Burst" mode allows greater flexibility for prefetching data in response to a PLB master request. (The DCU and ICU interfaces are both PLB masters, but are restricted to adhere to the PowerPC Architectural rules relating to prefetching.) The various rules for PLB prefetching in response to a generic master request are shown in Table 3-16, "PLB Prefetch Limitations for Guarded/Burst Mode

Combinations". The PPC401A1 should be wired to the PLB such that the core operates in non-burst mode.

Table 3-16. PLB Prefetch Limitations for Guarded/Burst Mode Combinations

Guarded	Burst	PLB/BIU Prefetch Limitations
No	No	Up to 1KB boundary*
No	Yes	No limit
Yes	No	Specific bytes or line requested*
Yes	Yes	Up to 1KB boundary
*PowerPC implementations must observe these architectural limitations		

Once a request is presented, the request identifiers will retain their values until the cycle after the request is accepted by the BIU or the cycle after the request is aborted by the DCU. At that time, they may change to describe the next request, if one is indicated by the assertion of the DCU_biuRequest signal.

3.9.3.7 DCU_biuPriority (output)

Core Name: **DKHIPRI**

This DCU Priority signal is asserted to indicate that the request being asserted by the DCU is, or has become, a high priority request. It is considered a high priority request because CPU pipeline execution is stalled and waiting for this request to be serviced. This signal is only valid when the DCU_biuRequest signal is active. Unlike the other request identifiers, this signal may change value (logic 0 to logic 1) while the request waits to be accepted - without an abort.

As an example, consider a write non-cacheable request (generated as a result of a store instruction) for which the priority bit is a logic 0 initially. If the instruction behind it is another store command, the DCU cannot accept the new CPU command, so the pipeline stalls until the first write request gets accepted. In this example, the priority signal will be changed from a logic 0 to a logic 1 for the first write request in the cycle that the CPU pipeline stall condition is detected. Also, all load instructions that require a transfer with the BIU/PLB will cause the priority signal to be asserted when the request is first presented.

The BIU designer should understand that the PPC401A1 does not have an ICU_biuPriority signal with which to dynamically change the priority of ICU requests. The following rules apply to the PPC401A1:

- If the ICU and DCU are both requesting a BIU transfer and the DCU_biuPriority signal is a logic 0, then the ICU request should be serviced.
- If the ICU and DCU are both requesting a BIU transfer and the DCU_biuPriority signal is a logic 1, then the DCU request should be serviced.

3.9.3.8 DBG_biuDMAOverride (output)

Core Name: **DBDMAOV**

This signal tells the BIU/PLB's arbitration logic to make the DCU request its highest priority request. This is done to enable a debug tool such as RISCWatch to have its DCU requests have the highest priority possible when trying to access memory.

3.9.3.9 DCU_biuAbort (output)

Core Name: **DKABORT**

This abort request signal is asserted if the DCU decides that it no longer requires the data transfer that it is requesting. This signal is valid **ONLY** when the DCU_biuRequest signal is active (it should be ignored otherwise), and may only be used to abort a request which has not been, or is currently being, accepted. During the cycle following the cycle of an aborted request, the DCU will either deactivate its request signal or make a new request by changing the request identifiers - the address bus, cacheability, guardedness, read/write, and data tag signals.

If the abort signal becomes active in the cycle that the BIU_dcuAccept signal also becomes active, the BIU is responsible to ensure that the transfer **DOES NOT** proceed any further. For an aborted read operation, this implies that no BIU_dcuDValid signals can be sent. For an aborted write operation, this implies that no (further) BIU_dcuWrDAck signals can be sent. It is possible and allowable that the BIU sent a BIU_dcuWrDAck in the same cycle as the BIU_dcuAccept. However, no further BIU_dcuWrDAck signals are allowed for the aborted write operation and the data associated with that transfer may not be written.

(NOTE: Due to the nature of the PPC401A1's pipeline, the PPC401A1 does not abort write (store/flush) operations.)

It is allowable for the abort signal to become active after the [accept](#) cycle, while the BIU is servicing an accepted (and not aborted) request. There are two cases of this:

1. If the DCU_biuRequest line is inactive, then the abort signal is to be ignored, since it only has meaning while the request signal is active. Since the BIU is in the process of performing an accepted request, it must fully complete the accepted operation.
2. If the request line is active, then the abort signal pertains to the **NEW** request, which will be removed or replaced by the DCU unit in the next cycle. Since the BIU is in the process of performing an accepted request, it must fully complete the accepted operation.

The DCU_biuAbort signal is a late-arriving signal. It's valid value arrives late in the cycle. Therefore, in the BIU, this signal must have a minimal amount of set-up time to be latched.

3.9.3.10 DCU_biuWrDBus(0:31) (output)

Core Name: **DKDOnn**

This bus, also referred to as the DCU write data bus, is a 32-bit memory-aligned data bus which is used to transfer write data from the DCU to the BIU. The DCU will place the first data word of the transfer on this bus when it raises the DCU_biuRequest line for a write operation (DCU_biuRNW is a logic 0). The data on this write data bus must be latched by the BIU at the end of the cycle in which the BIU asserts the BIU_dcuWrDAck signal.

For a cache-inhibited write, the DCU will retain the value of the write data bus through the end of the cycle that the BIU asserts the BIU_dcuWrDack signal, at which time the DCU considers the transfer to be complete.

For a cacheable write (line flush) the DCU will place the Word-0 (the lowest word address of the cache line) data on the write data bus through the end of the cycle that the BIU asserts the first BIU_dcuWrDack signal. The DCU will then place Word-1 data on the write data bus through the end of the cycle that it detects the next assertion of BIU_dcuWrDack, etc., until the fourth assertion of BIU_dcuWrDack, at which time the DCU considers the transfer to be complete.

3.9.3.11 BIU_dcuAccept (input)

Core Name: **DKACPT**

This signal is asserted by the BIU in response to a DCU request to indicate that the transfer request has been acknowledged (i.e. accepted). The BIU must latch the request identifiers - the address bus, cacheability, guardedness, read/write, and data tag signals at the end of the cycle that it presents the BIU_dcuAccept signal. The DCU_biuRequest signal will be deasserted by the core in the next cycle unless another request is pending. The accept signal is asserted for one cycle only, once per request. It is permissible for the BIU to accept the request the same cycle that the request is presented. For a read operation, the accept signal must precede the first BIU_dcuDValid assertion. For a write operation, the write data acknowledge signal can be asserted along with the accept signal.

In certain situations, the DCU can present a second request before the last accepted request has been fully satisfied. The BIU can accept the second request in the same manner that any other request is accepted. Also, if the BIU gets a write request to a memory address (or line) and then a subsequent read request to the same memory address (or line), it must allow the first (write) request to be satisfied such that the second (read) request will return the new data for that address.

3.9.3.12 BIU_dcuDValid (input)

Core Name: **DKDVAL**

This DCU Data Valid signal is asserted by the BIU during read operations to indicate to the DCU that a data word of the DCU read request is available on the DCU read data bus, BIU_dcuRdDBus(0:31). This signal will be active one cycle per word transferred. The DCU will latch the data word from the DCU read data bus at the end of the cycle in which BIU_dcuDValid is asserted.

For a cache-inhibited transfer, the BIU_dcuDValid signal is asserted for one cycle, as the data word is presented across the DCU read data bus.

For a cacheable transfer, all four words of the cache line that contains the target data will be transferred, one word at a time, with the BIU_dcuDValid signal being asserted once for each word, as it is presented across the DCU read data bus. For cacheable accesses, the BIU_dcuWdAddr(0:1) signals must indicate which word of the cache line is currently being transferred across the DCU read data bus.

There may be any number of wait states between data transfers on the DCU read data bus, allowing the BIU to pack data from a non-word device until a full word is collected, and/or to match the data transfer rate with the device on the external bus. If there are zero wait states on the external bus and the device is a word device, then the BIU_dcuDValid can remain on for four consecutive cycles, each cycle presenting a different word of the line fill.

3.9.3.13 BIU_dcuRdDBus(0:31) (input)

Core Name: **DKDInn**

This bus, also referred to as the DCU read data bus, is a 32-bit memory-aligned data bus which is used to transfer read data from the BIU to the DCU. The data on this bus is valid when it is qualified by BIU_dcuDValid. The data valid signal is used to indicate to the DCU that data associated with the read transfer in progress is available on the DCU read data bus, and that it must be latched at the end of the current cycle. Data is always presented across the DCU read data bus as a word entity. It is up to the DCU to remember the size and address of the requested data.

If the transfer in progress is a cache-inhibited DCU read, the request could have been made for less than a full word of data. Only the data necessary to satisfy the request, as originally specified in the DCU_biuDTags(0:3), need be accessed by the BIU and valid when this word is transferred across the DCU read data bus.

If the transfer in progress is a cacheable DCU read, the request is seen as a cache line fill request and the four words that comprise the cache line will be transferred across the DCU read data bus, one word at a time, qualified by the BIU_dcuWdAddr(0:1) signals which must indicate which word of the cache line is currently being transferred. For cacheable reads, all bytes on the BIU_dcuRdDBus(0:31) must be valid for each word transfer.

3.9.3.14 BIU_dcuWdAddr(0:1) (input)

Core Name: **DKWADRn**

These DCU word address signals are used when the words associated with a cacheable request are being sent from the BIU back to the DCU. They are used to indicate which word of the four word cache line is currently being transferred across the BIU_dcuRdDBus(0:31), as shown in Table 3-17, "BIU_dcuWdAddr(0:1) Decode". These signals are valid when qualified by BIU_dcuDValid.

For cache-inhibited read transfers, the DCU disregards the BIU_dcuWdAddr(0:1) signals.

For cacheable read transfers, the DCU will use these signals to write the data to the appropriate word location within the cache line. The four words that comprise the cache line containing the target word may be transferred to the cache in any order (target-word-first, sequential, other). The order of the delivery of the words for a cache line fill is determined by the BIU design. When the target data word is transferred to the DCU, it is forwarded to the CPU and concurrently sent it to the DCU cache array. This enables the CPU to get the target

data as soon as possible so that CPU processing may continue. Therefore, the BIU should provide target-word-first delivery whenever possible.

Table 3-17. BIU_dcuWrAddr(0:1) Decode

BIU_dcuWrAddr(0:1)	Decode
00	Word 0
01	Word 1
10	Word 2
11	Word 3

3.9.3.15 BIU_dcuWrDAck (input)

Core Name: **DKWDACK**

This DCU Write Data Acknowledge signal is asserted by the BIU during write operations to indicate to the DCU that the DCU_biuWrDBus(0:31) data word will be accepted and latched by the BIU at the end of the current cycle. This signal will be active one cycle per word transferred. If the DCU has more data associated with this transfer, this signal is an indication for the DCU to place the next data word on the DCU_biuWrDBus(0:31) in the following cycle. The BIU_dcuWrDAck signal can be asserted during the same cycle that the BIU_dcuAccept signal is active for this request. The PPC401A1 will not make a second *write* request until the cycle after the last write data ack is asserted by the BIU.

For a cache-inhibited transfer, the BIU_dcuWrDAck signal must be asserted for one cycle, as the data word is accepted by the BIU from the DCU_biuWrDBus(0:31). The bytes of the data word that are to be written out to storage will have been specified by DCU_biuDTags(0:3) at the time the DCU_biuRequest signal was asserted. Likewise, the target address to be written will have been specified on the DCU_biuABus(0:31) when the request was asserted. When the BIU_dcuWrDAck signal is asserted for a cache-inhibited request, the DCU considers the operation complete and continues on. It does not wait for the operation to be completed out to memory.

For a cacheable transfer (a cache line flush) all four words of the cache line that contains the target data word will be transferred, one word at a time, with the BIU_dcuWrDAck signal being asserted once for each word, as it is accepted across the write data bus, DCU_biuWrDBus(0:31). The DCU will transfer the words in ascending word order, starting with Word-0 (the lowest word address of the cache line) and ascending to Word-3. There may be any number of wait states between BIU_dcuWrDAck cycles on the DCU write data bus, allowing the BIU to unpack data to a non-word device, and/or to match the data transfer rate with the device on the external bus. If there are zero wait states on the external bus and the device is a word device, then the BIU_dcuWrDAck signal can remain on for four consecutive cycles, each cycle accepting a different word of the line flush. When the last BIU_dcuWrDAck signal is asserted for a cacheable request, the DCU considers the operation complete and continues on. It does not wait for the operation to be completed out to memory.

3.9.3.16 BIU_cpuDCUBusy (input)

Core Name: **CPBSYDK**

This signal indicates that the BIU is busy performing a DCU-initiated operation. It should be asserted the cycle after a request is accepted and remain asserted (logic '1') until all operations initiated by the DCU have been fully completed by the BIU.

For a single read request, this signal should be asserted the cycle after the accept cycle and deasserted the cycle after the last BIU_dcuDValid is presented for the request.

For a single write request, this signal should be asserted the cycle after the accept cycle and deasserted the cycle after the memory transfer has been completed to the target bytes/line.

If multiple requests are initiated and overlap, the busy signal should be asserted the cycle after the accept cycle of the first request and remain asserted until all activity associated with the overlapped requests is completed by the BIU.

This signal is used by the CPU in conjunction with the sync instruction. The sync instruction requires that all storage operations that were initiated prior to the issue of the sync instruction are fully completed before further instruction processing is allowed.

As an example, instructions could be read into the processor, modified in the D-cache, and then stored back out into memory with a dcbf instruction. The sync instruction will cause the CPU pipeline to hold until the BIU_cpuDCUBusy signal goes inactive, signifying that the modified code is now fully out in memory (as a result of the completion of the dcbf instruction) and is available to the instruction fetcher. Of course in such an example, you would want to invalidate the I-cache for that instruction (using icbi) and then perform an isync to make sure you actually go to memory to get the updated code.

3.9.3.17 BIU_dcuBusErr (input)

Core Name: **DKBERR**

This signal is driven by the BIU to indicate that it has detected an error while attempting to transfer data associated with a DCU request. The assertion of this signal should coincide with the BIU latching and locking the pertinent error information into registers for software investigation of the error. Unlike the ICU side error signal, this signal may be asserted outside of the actual DCU-BIU transfer due to this interface's write capability.

This error indication is asynchronous with respect to instruction execution. For example, a cacheable load may cause both a cache line fill of the target word's cache line, and a cache line flush of the least recently used dirty cache line in the congruence class. Consider the case where the last word written for the cache line flush causes an error. This error can occur long after the actual instruction that caused the initial bus activity has been executed in the CPU pipeline.

For a read operation in which an error has been detected, this signal should only be asserted by the BIU while the BIU_dcuDValid signal is active. The error signal should be active for one cycle only, per word transferred.

For a write operation in which an error is detected, this signal can be asserted any time from the cycle following the accept cycle until many cycles after the final BIU_dcuWrDack. The error signal should be active for one cycle only, per error encountered.

The presence of this error signal does not terminate the DCU-BIU transfers. The current transfer should be completed to the DCU without interruption. The `BIU_dcuDValid` signals (for a read operation) or the `BIU_dcuWrDack` signals (for a write operation) are still required to complete the transfer - one for a cache-inhibited transfer, and four for a cacheable transfer. If the DCU request being serviced is cacheable, the D-cache will load the entire cache line, regardless of any errors that are associated with that data. When the BIU has completed this transfer, it should continue on to the next request.

Detection of this error signal by the CPU will cause a Machine Check Exception to occur, causing a Machine Check interrupt to be taken, if the ME bit of the MSR is on.

When an error occurs, it would be helpful to provide the failing address and the type of error encountered. The BIU designer should consider implementation of a BEAR (Bus Error Address Register) and a BESR (Bus Error Syndrome Register).

The BEAR should be a 32-bit register that contains the address of the external bus access that caused the error to occur.

The BESR might minimally consist of fields that describe the error type and whether the operation being performed was a read or write operation. Possible error types might include Protection Check, Non-Configured Check, External Bus Error, and BIU Timeout Check.

For errors such as Protection Checks or Non-Configured Checks, the error signal should be asserted for each word transferred since the data is invalid for all four words of the line fill. For errors such as External Bus Errors and BIU Timeout Checks, the error signal should be asserted with each word to which they apply.

3.9.4 Data-Side BIU Interface Timing Diagrams

The following timing diagrams represent typical scenarios for transfers that may occur on the PPC401A1 DCU-PLB interface when it is connected to the IBM processor local bus (PLB) soft macro or to an application-specific BIU.

3.9.4.1 DCU/BIU Timing Diagram Guidelines

To make the timing diagrams concise, the following assumptions apply for each timing diagram, unless explicitly stated otherwise.

1. The timing diagrams generally show the best achievable timing relationships for the indicated transfer sequences. Table 3-19 indicates which timing diagram illustrates the limiting case for each transfer sequence.
2. Requests are acknowledged by the PPC401A1 in the same cycle in which they are presented on the interface.

Requests may be acknowledged by the PLB in the same cycle that the DCU presents the request, or during a subsequent cycle if it is currently busy doing other work.

3. The `PLB_dcuRdDack` (read data acknowledge) signal is asserted in the cycle immediately following the acknowledge of a read request.

For a read request, the earliest cycle that the PLB_dcuRdDAck signal may be asserted to return data is the cycle immediately following the acknowledge for that request.

The earliest cycle in which the PLB soft macro can present the PLB_dcuRdDAck signal is during the second cycle following the acknowledge for that request.

There can be any number of cycles between the acknowledge cycle and the first PLB_dcuRdDAck assertion, or between PLB_dcuRdDAck assertions for a cacheable transfer, to enable the BIU to gather data from slow or non-word devices. The PPC401A1 waits for the assertion of the appropriate number of PLB_dcuRdDAck signals to satisfy the request.

4. All transfers assume that the BIU can access a word of data per cycle.

When the PPC401A1 requests a non-cacheable read, it expects the request to be satisfied by one transfer across the DCU-PLB interface. The BIU may need to pack data from a byte device into a single word transfer (of one to four contiguous bytes) across the DCU-PLB interface. Similarly, the BIU may need to unpack the data from the PPC401A1 to a byte device. When the PPC401A1 makes a cacheable read request, it expects the request to be satisfied by four word transfers across the DCU/PLB interface.

5. All cacheable reads assume target word first delivery

If a cacheable read presents the target word *last*, the request following the conclusion of the read transfer can be pushed out by one additional cycle.

6. All cacheable writes send the data words out in sequential order from word 0 to word 3, where word 0 is the lowest addressable word of the four-word transfer.
7. The DCU_plbBE(0:3) (byte enable) signals are only significant for non-cacheable transfers, and can be ignored for cacheable transfers (all bytes of each word must be transferred).

Table 3-18 lists the abbreviations used in the timing diagrams.

Table 3-18. Key to DCU/PLB Timing Diagram Abbreviations

Abbreviation	Description	Where Used
rc#, wc#	Read/write cacheable request identifier	Request, AddrAck, RdDAck, WrDAck
rnc#, wnc#	Read/write non-cacheable request identifier	Request, AddrAck, RdDAck, WrDAck
adr#	Address for identified (#) request	ABus(0:29)
d#	Data associated with a non-cacheable, aligned word request	RdDBus(0:31), WrDBus(0:31)
xx23	data associated with a non-cacheable, partial word request, as dictated by BE(0:3). ("x" indicates a don't care)	RdDBus(0:31), WrDBus(0:31)
d#_#	Data associated with a cacheable request	RdDBus(0:31), WrDBus(0:31)
0111	Byte enables for current non-cacheable request	BE(0:3)

Table 3-18. Key to DCU/PLB Timing Diagram Abbreviations

Abbreviation	Description	Where Used
Subscripts	Identifies a particular word in a cache line	RdDAck, RdDBus(0:31), WrDAck, WrDBus(0:31)
#	Identifies a particular word in a cache line	RdWdAddr(2:3)

Table 3-19 shows the types of back-to-back operations depicted in the timing diagrams. The first and second major column headings describe the type of operations involved (read or write, cacheable or non-cacheable). The remaining columns provide a sequence ID that is referred to in the text, the referenced timing diagram, and the cycles relevant to the listed operations.

Table 3-19. DCU–BIU/PLB Timing Diagram Reference

First Operation				Second Operation				Sequence ID	Reference Diagram	Relevant Cycles
Read		Write		Read		Write				
C	NC	C	NC	C	NC	C	NC			
			X				X	1	Figure 3-11	2–6
			X			X		2 2(write-through)	Figure 3-21 Figure 3-13	6–9 3–5
			X		X			3	Figure 3-11	6–7
			X	X				4	Figure 3-14	2–4
		X					X	5	Figure 3-21	2–6
		X				X		6	Figure 3-21	9–14
		X			X			7	Figure 3-16	5–7, 12–13
		X		X				8	Figure 3-20	3–4, 11–12
	X						X	9	Figure 3-11	7–11
	X					X		10	Figure 3-16	4–5, 7–12
	X				X			11 11(abort)	Figure 3-11 Figure 3-12	12–15, 15–18, 8–13
	X			X				12	Figure 3-15	2–6
X							X	13 13(write-through)	Figure 3-14 Figure 3-13	4–10 2–3, 18–19
X						X		14	Figure 3-19	2–3, 8–9
X					X			15 15(abort)	Figure 3-15 Figure 3-12	6–11 2–7
X				X				16 17	Figure 3-17 Figure 3-18	2–8,8–14 all

3.9.4.2 DCU/BIU Non-Cacheable Mix (Custom BIU)

Figure Highlights

- Read non-cacheable request can immediately follow a write non-cacheable request (CBIU/PLB).
- A string of write non-cacheable requests can be made every other cycle (CBIU/PLB).
- A string of read non-cacheable requests can be made every third cycle (Custom BIU), or every 4th cycle (PLB)
- The fastest that a write non-cacheable request can follow a read non-cacheable request in a non-write-through situation. (Custom BIU)
- Covers the following Sequence IDs from Table 3-19, “DCU–BIU/PLB Timing Diagram Reference,” on p. 3-60: 1,3,9,11.
- The “DCU/BIU Timing Diagram Guidelines” on page 3-58 apply.

The sequence of instructions depicted below is as follows: Write (store) eight unaligned bytes (cycles 2-6), followed by a read (load) of an aligned word (cycles 7-9), a write (store) of an aligned word (cycles 10-11), a read (load) of an unaligned word (cycles 12-17), and a read of an aligned word (cycles 18-20). Notice how the BE (Byte Enable) bits are used to direct which data bytes are transferred.

If these transfers were done to a PLB-compliant BIU, the RdDack assertions would be delayed a single cycle for each of the “rnc” transfers. This would delay the request assertions of cycles 11, 15, and 18 by an additional cycle each.

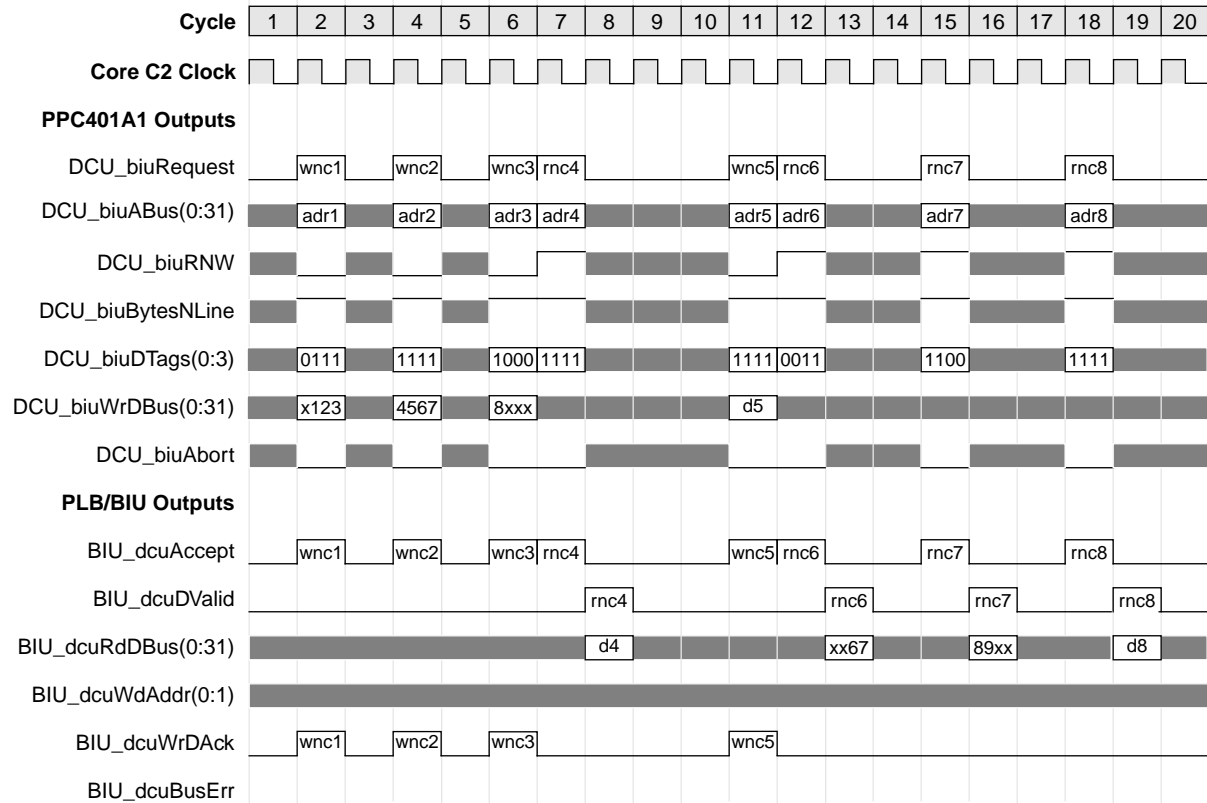


Figure 3-11. DCU/BIU-PLB Non-Cacheable Mix (Custom BIU)

3.9.4.3 DCU/BIU-PLB Aborted Read Requests

Figure Highlights

- Shows several typical aborted read operations (PLB or Custom BIU).
- Write Operations are not aborted for the PPC401A1.
- Shows the fastest that a new request can follow an abort on the PLB/BIU interface.
- Covers the following Sequence IDs from Table 3-19, “DCU–BIU/PLB Timing Diagram Reference,” on p. 3-60: 11(abort), 15(abort).
- The “DCU/BIU Timing Diagram Guidelines” on page 3-58 apply.

Three different abort scenarios are shown. In cycle 2, a request is presented on the interface and is immediately acknowledged by the BIU/PLB via AddrAck. Yet in the *same* cycle, a decision is made by the DCU to abort the request. Similarly, the request of cycle 7 is presented and is acknowledged by the BIU/PLB in cycle 8 via the assertion of AddrAck. In cycle 8, however, the DCU once again aborts the request. In cycle 13 a new request is presented. This time the abort is asserted by the DCU before the BIU/PLB is able to assert the AddrAck signal. IN ALL THREE CASES, the BIU/PLB is responsible to cease any transfer of data to the DCU associated with the aborted request.

The fastest that a new non-cacheable read request can be made on the interface is in the fifth cycle following the cycle during which the abort signal was asserted. Non-cacheable writes and any cacheable requests take an additional cycle since they need to wait for data, or be resolved as a miss, respectively, before the request can be generated.

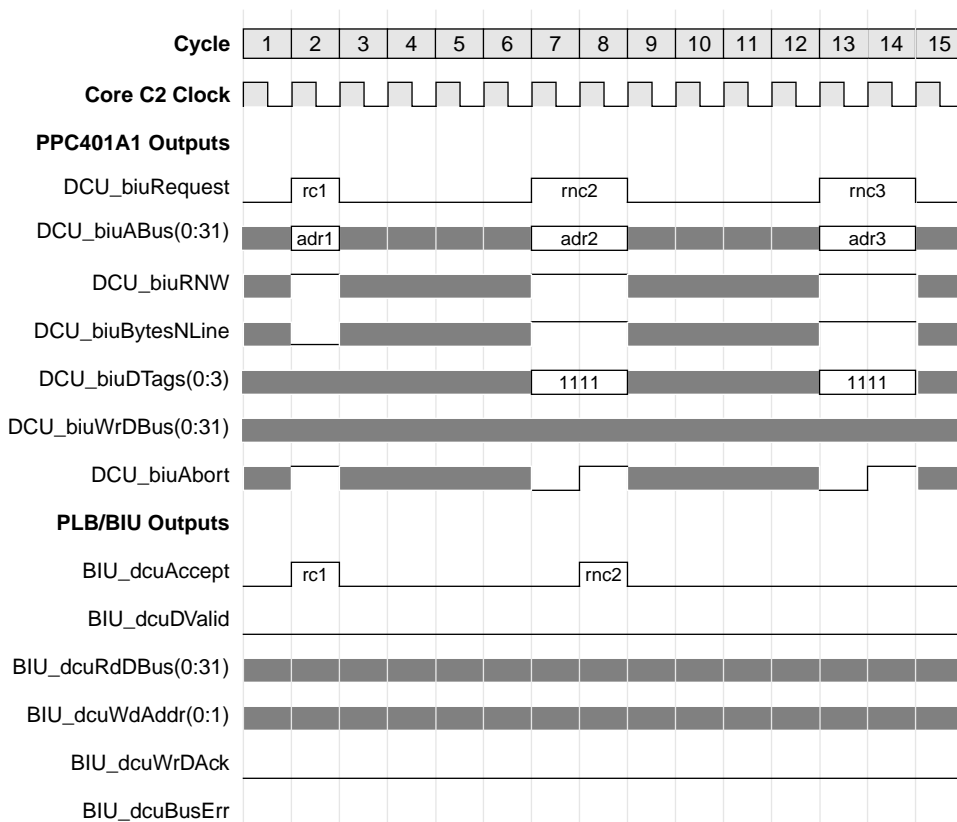


Figure 3-12. DCU/BIU-PLB Aborted Read Requests

3.9.4.4 DCU/BIU Store Write-through Miss w/ Fill-Flush, then Store WT Hits (Custom BIU)

Figure Highlights

- Shows a typical store write-through miss with subsequent hits to the same cache line. (Custom BIU)
- The fastest that a write non-cacheable (write-through ONLY) request can follow a read cacheable request. (Custom BIU)
- The fastest that a write cacheable request can follow a write non-cacheable (write-through ONLY) request. (CBIU/PLB)
- Covers the following Sequence IDs from Table 3-19, “DCU–BIU/PLB Timing Diagram Reference,” on p. 3-60: 2(write-through), 13(write-through).
- The “DCU/BIU Timing Diagram Guidelines” on page 3-58 apply.

The sequence of instructions depicted below begins with a store cacheable miss to a write-through region which causes 1) a line fill “rc1” (cycles 1-6), 2) the write non-cacheable (write-through) operation (cycle 3), and 3) a flush “wc3” of the line moved out of the cache (cycles 5-8). In cycle 7, the cycle after the last RdDack of the line fill, the next store cacheable write-through address (adr4) hits in the cache and its corresponding write-through “wnc4” occurs in cycle 9, delayed a cycle because it must wait for the “wc3₃” WrDack assertion of the “wc3” flush transfer before it can place the “wnc4” data (d4) on the WrDBus. (Write-through operations use the flush state machine. If the line fill “rc1” did not cause the “wc3” flush, then the activity for cycles 9 and on could be shifted one cycle to the left.) The next store cacheable write-through address (adr5) is presented to the DCU in cycle 10, hits in cycle 11 and causes the “wnc5” transfer of cycle 12. This sequence repeats in cycles 13-15 for address adr6.

In cycle 16 the next store cacheable address (adr7) is presented to the DCU. This address misses in the cache (cycle 17) and the entire sequence shown in cycles 1-15 repeats, though only the first few cycles are shown.

Cycles 2-3 depict the best case timing for a rc (read cacheable) to wnc (write non-cacheable), but ONLY for a write-through scenario. For the best case rc to wnc timing of a non-write-through case see Figure 3-14. Cycles 3-5 depict the best case timing for a wnc to wc, but ONLY for a write-through scenario. For the best case wnc to wc timing of a non-write-through case see Figure 3-21.

This diagram assumes that the rc1 transfer is target word first. If it were target word last, everything from cycle 7 and on would push out an additional cycle to the right.

If these transfers were done to a PLB-compliant BIU, the first RdDack assertion would be delayed a single cycle for each of the “rc” transfers. This would delay the last RdDack (rc1₃) to cycle 7, the hit1 for adr4 into cycle 8, and the remainder of the transfer (cycles 9-15) would remain as shown.

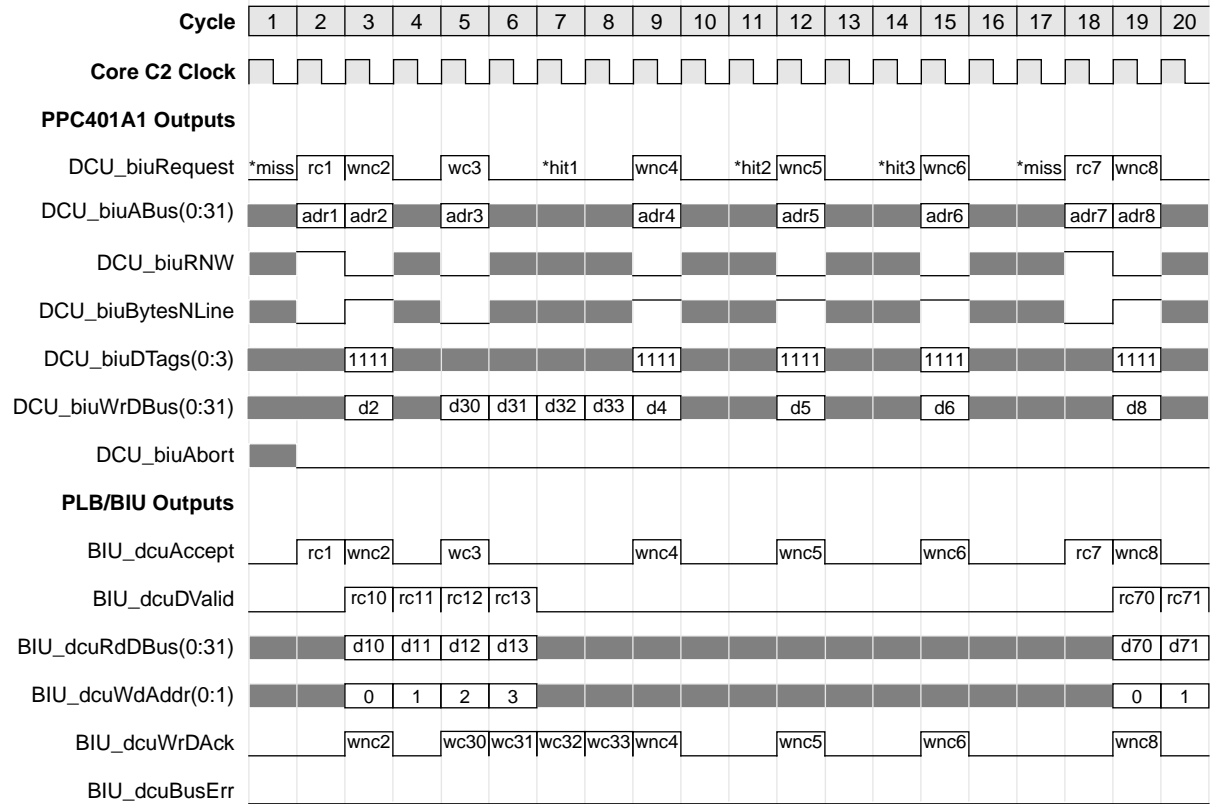


Figure 3-13. DCU/BIU-PLB Store write-through Miss, then Store write-through Hits

3.9.4.5 DCU/BIU Non-Cacheable Write, Cacheable Read Mix

Figure Highlights

- The fastest that a read cacheable request can follow a write non-cacheable request.
- The fastest that a write non-cacheable request can follow a read cacheable request (custom BIU).
- Covers sequence IDs 4 and 13 from Table 3-19, “DCU–BIU/PLB Timing Diagram Reference,” on p. 3-60.

The timing diagram depicts the following instruction sequence:

1. Write a non-cacheable, aligned halfword (cycles 1–2)
2. Read a cache line (cycles 3–8)
3. Write a non-cacheable, non-write-through aligned word (cycles 9–10)

Note the use of the Byte Enables for the halfword write to `adr1`.

If these transfers were done to a PLB-compliant BIU, the first `RdDAck` (`rc20`) would be delayed a single cycle for the `adr2` request, pushing cycles 5–10 out one cycle. Also, if the target word were the last word transferred to the DCU (`d23`), the request in cycle 10 would not be asserted until cycle 11.

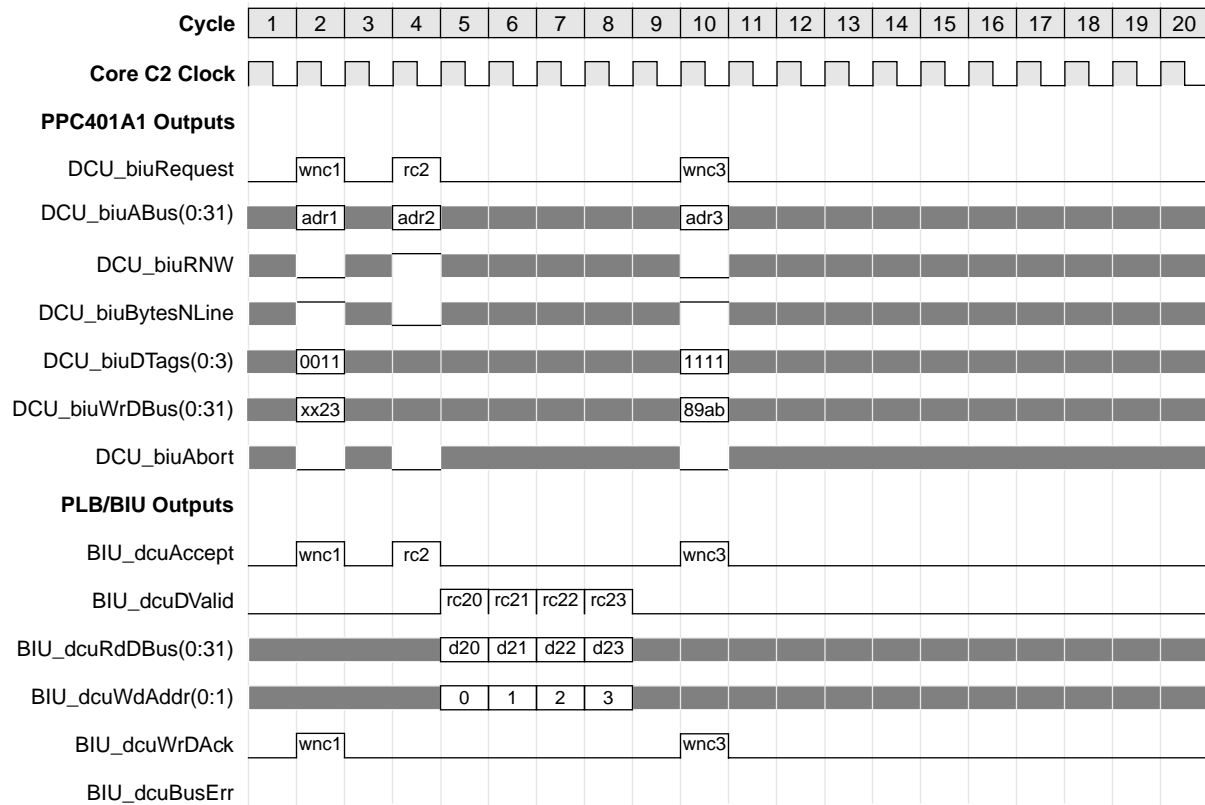


Figure 3-14. DCU/BIU-PLB Non-Cacheable Write, Cacheable Read Mix

3.9.4.6 DCU/BIU Non-cacheable, Cacheable Read Mix (Custom BIU)

Figure Highlights

3

- The fastest that a read cacheable request can follow a read non-cacheable request. (Custom BIU)
- The fastest that a read non-cacheable request can follow a read cacheable request. (Custom BIU)
- Covers the following Sequence IDs from Table 3-19, “DCU–BIU/PLB Timing Diagram Reference,” on p. 3-60: 12,15.
- The “DCU/BIU Timing Diagram Guidelines” on page 3-58 apply.

The sequence of instructions depicted below is: Read a non-cacheable, aligned word (cycles 2-4), Read a cache line (cycles 5-10), and Read a non-cacheable, aligned word (cycles 11-12).

If these transfers were done to a PLB-compliant BIU, the first RdDack would be delayed a single cycle for each of the three transfers. Also, if the target word is the last word transferred to the DCU, then the request of cycle 11 would not be asserted until cycle 12.

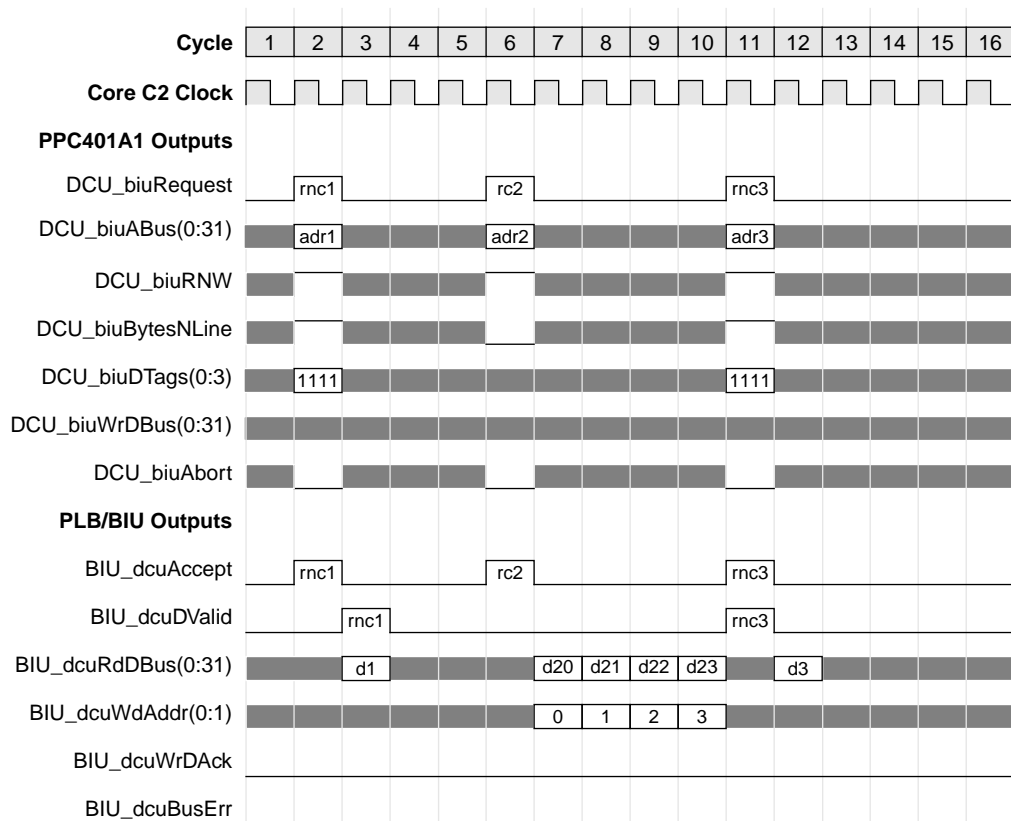


Figure 3-15. DCU/BIU-PLB Non-cacheable, Cacheable Read Mix (Custom BIU)

3.9.4.7 DCU/BIU-PLB Non-cacheable Read, Cacheable Write Mix

Figure Highlights

3

- The fastest that a write cacheable request can follow a read non-cacheable request. (CBIU/PLB)
- The fastest that a read non-cacheable request can follow a write cacheable request. (CBIU/PLB)

Covers the following Sequence IDs from Table 3-19, “DCU–BIU/PLB Timing Diagram Reference,” on p. 3-60: 7,10.

The “DCU/BIU Timing Diagram Guidelines” on page 3-58 apply, except for the ‘nop’ usage indicated in the paragraph below.

The sequence of instructions depicted below is: a dcbf (data cache block flush) that causes a flush of a cache line “wc2”, followed by a load non-cacheable “rnc1” (which actually get requested in reverse order due to the interrelationships of the DCU’s main and flush state machines), followed by another read non-cacheable (cycles 7–8), and a dcbf/nop/read non-cacheable CPU instruction sequence that will allow the dcbf (cycles 12–15) to be requested ahead of the read non-cacheable (cycles 13-14). The best case for read non-cacheable to write cacheable requests is shown in cycles 4-5 and is the result of the reordering of the CPU requests. Otherwise, the best case read non-cacheable to write cacheable delay is shown in cycles 7–12.

If these transfers were done to a PLB-compliant BIU, the RdDAck would be delayed a single cycle for each of the read non-cacheable transfers. This would delay the rnc3 request assertion into cycle 8.

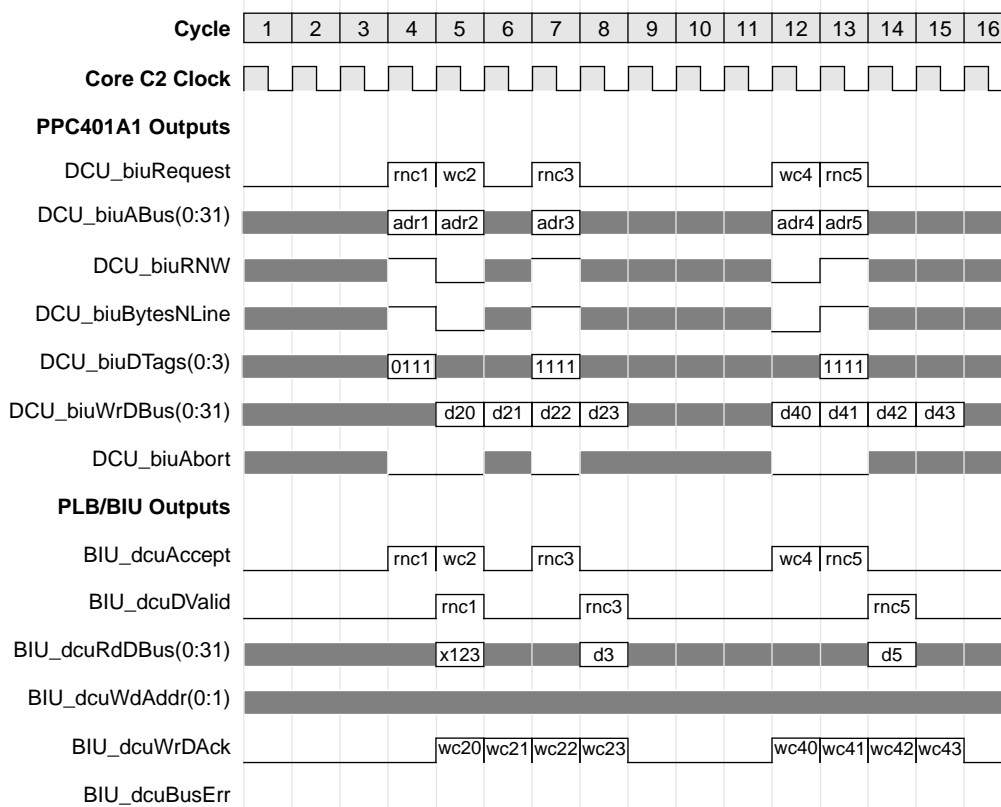


Figure 3-16. DCU/BIU-PLB Non-cacheable Read, Cacheable Write Mix

3.9.4.8 DCU/BIU Back to back Cacheable Reads (Custom BIU)

Figure Highlights

3

- The fastest that a read cacheable request can follow another read cacheable request. (Custom BIU)
- Covers Sequence ID 16 from Table 3-19, “DCU–BIU/PLB Timing Diagram Reference,” on p. 3-60.
- The “DCU/BIU Timing Diagram Guidelines” on page 3-58 apply.

The sequence depicted below can represent the transfers generated with a Custom BIU as a result of a series of “dcbt” instructions or Read Cacheables in the CPU. In the case of a series of Read Cacheables, if the target word is the last word transferred to the DCU, then the next request would be asserted a cycle later than shown. This occurs because the pipeline is stalled, waiting for the target word. Since the pipeline does not stall for dcbt instructions, this penalty cycle would not be seen for a series of dcbt's.

If these transfers were done to a PLB-compliant BIU, the first RdDack would be delayed a single cycle for each of the three transfers, delaying the request assertions of cycle 8 and 14 by an additional cycle each.

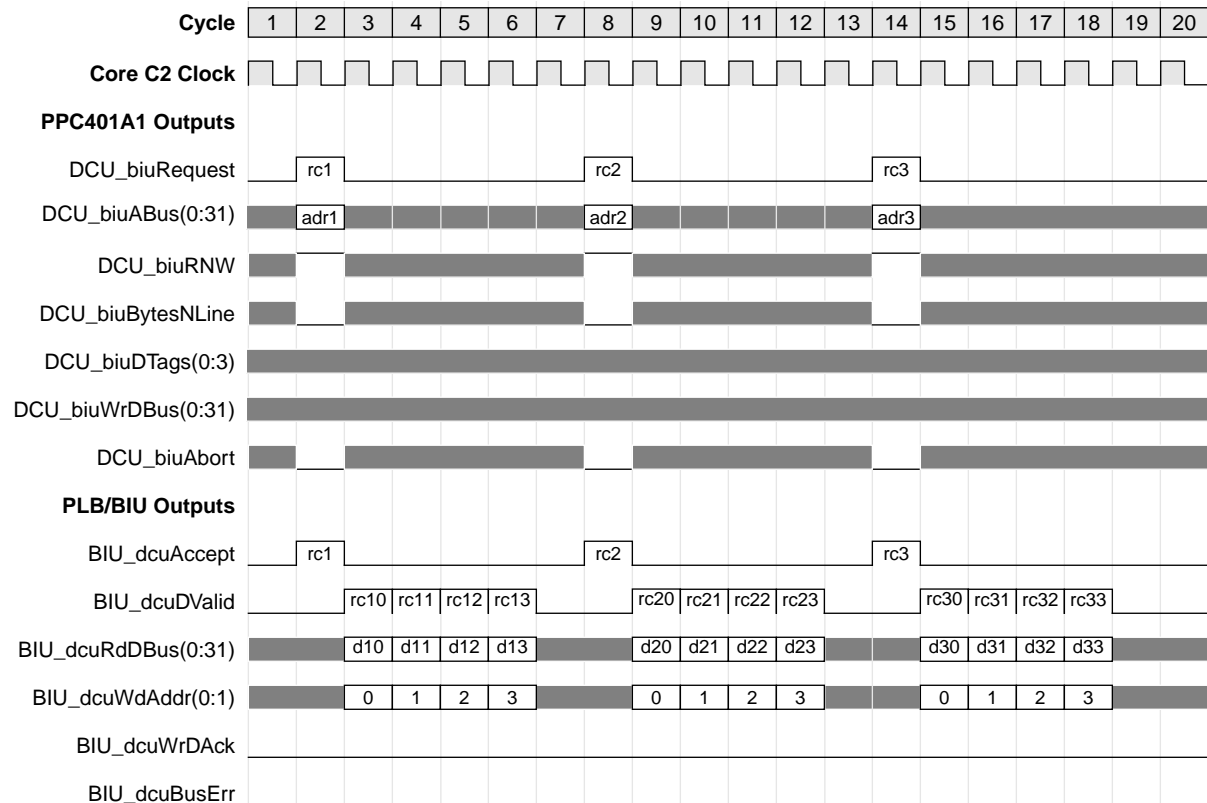


Figure 3-17. DCU/BIU-PLB Back to back Cacheable Reads (Custom BIU)

3.9.4.9 DCU/BIU Back toBack Cacheable Reads with Cache Hits Included (Custom BIU)

Figure Highlights

- Illustrative example of sequential loads that initially miss in the D-cache.
- Covers Sequence ID 17 from Table 3-19, “DCU–BIU/PLB Timing Diagram Reference,” on p. 3-60.
- The “DCU/BIU Timing Diagram Guidelines” on page 3-58 apply.

The sequences depicted below represents the best-case transfers generated with a Custom BIU as a result of a series of sequential load cacheables that initially miss in the D-cache. The first address misses in cycle 1, causing the “rc1” PLB request of cycle 2. In cycle 3, the target data is presented across the interface and the DCU bypasses the target word to the CPU. Since the DCU is blocking during a line fill (no cache accesses and no additional commands accepted from the CPU), the DCU does not look to satisfy the next load data command from the CPU (which is presented to the DCU in cycle 4) until it detects that the line transfer is complete - which it does in cycle 6 when the last RdDAck is detected. Therefore, in cycle 7 the cache is accessed, a hit determination is made (for adr1), and the data (d1₁) is sent to the CPU. In cycle 8 the next load command is presented to the DCU by the CPU. In cycle 9 the cache is accessed, a hit determination is made (for adr2) and the data (d1₂) is sent to the CPU. This process repeats for cycles 10 and 11 for adr3 data. In cycle 12 the next load command (for adr4) is presented to the DCU. In cycle 13 the cache is accessed and a miss determination is made. This causes the “rc2” PLB request in cycle 14, and the entire sequence repeats, but is not shown in its entirety.

If the target word is the last word transferred to the DCU (d1₃ in cycle 6), then cycle 7 would be the cycle in which the CPU would present the next load command and the cache access of cycle 7 would shift to the right into cycle 8. This occurs because the CPU pipeline is stalled, waiting for the target word.

If these transfers were done to a PLB-compliant BIU, the first RdDAck would be delayed a single cycle for each of the PLB transfers, shifting cycles 3 through 14 one cycle to the right.

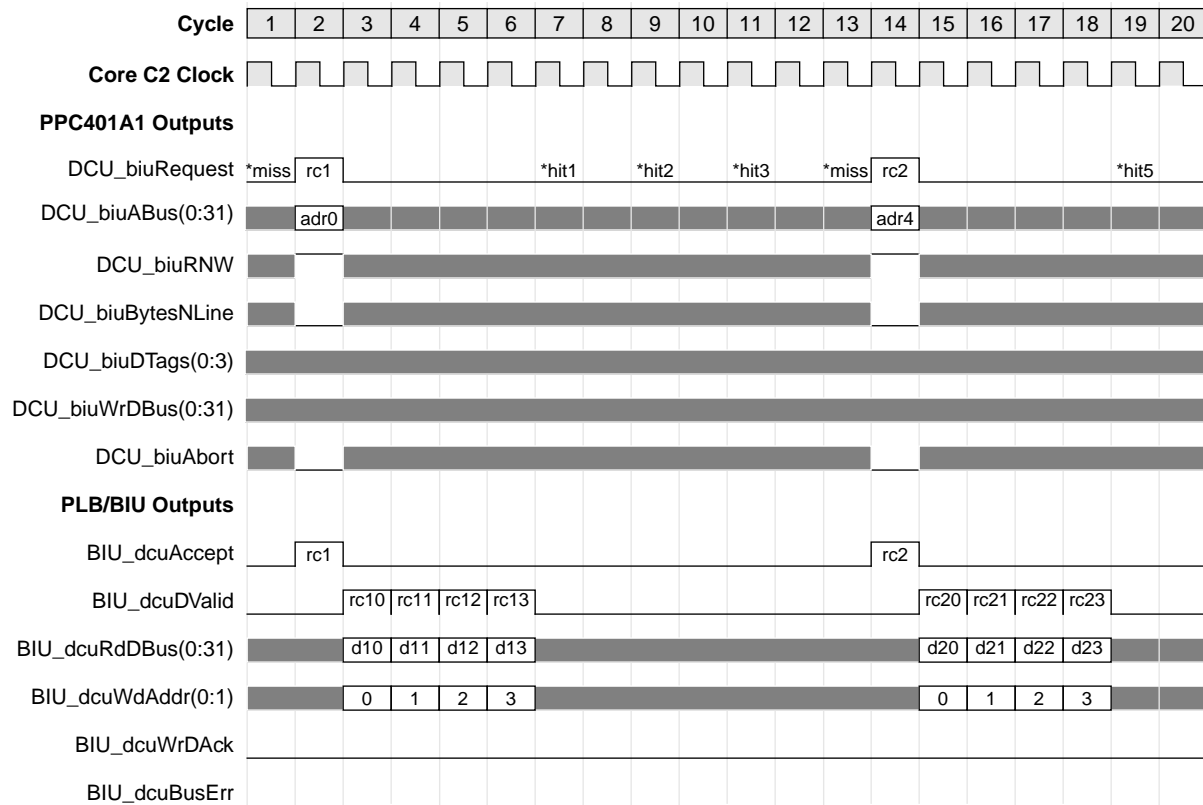


Figure 3-18. DCU/BIU-PLB Back to back Cacheable Reads w/ Hits (Custom BIU)

3.9.4.10 DCU/BIU-PLB Cacheable Miss to Dirty Line (Fill-Flush-Fill)

Figure Highlights

- The fastest that a write cacheable request can follow a read cacheable request. (CBIU/PLB)
- Covers Sequence ID 14 from Table 3-19, “DCU–BIU/PLB Timing Diagram Reference,” on p. 3-60.
- The “DCU/BIU Timing Diagram Guidelines” on page 3-58 apply.

The sequence of instructions depicted below is: Read cacheable that causes a line fill (cycles 2-6) and a flush of a cache line (cycles 3-6), followed by two more read cacheables that do the same. The read cacheable op could be a dcbt or a load from a cacheable region.

If these transfers were done to a PLB-compliant BIU, the first RdDAck would be delayed a single cycle for each of the read cacheable transfers. This would shift the read activity for cycles 3-6 one cycle to the right, delaying rc3 into cycle 9. The remainder of the transfer would shift out as well, shifting additional cycles for each occurrence of the first RdDAck associated with each read cacheable. Also, if the target word is the last word transferred to the DCU, then the next read cacheable request would be asserted a cycle later.

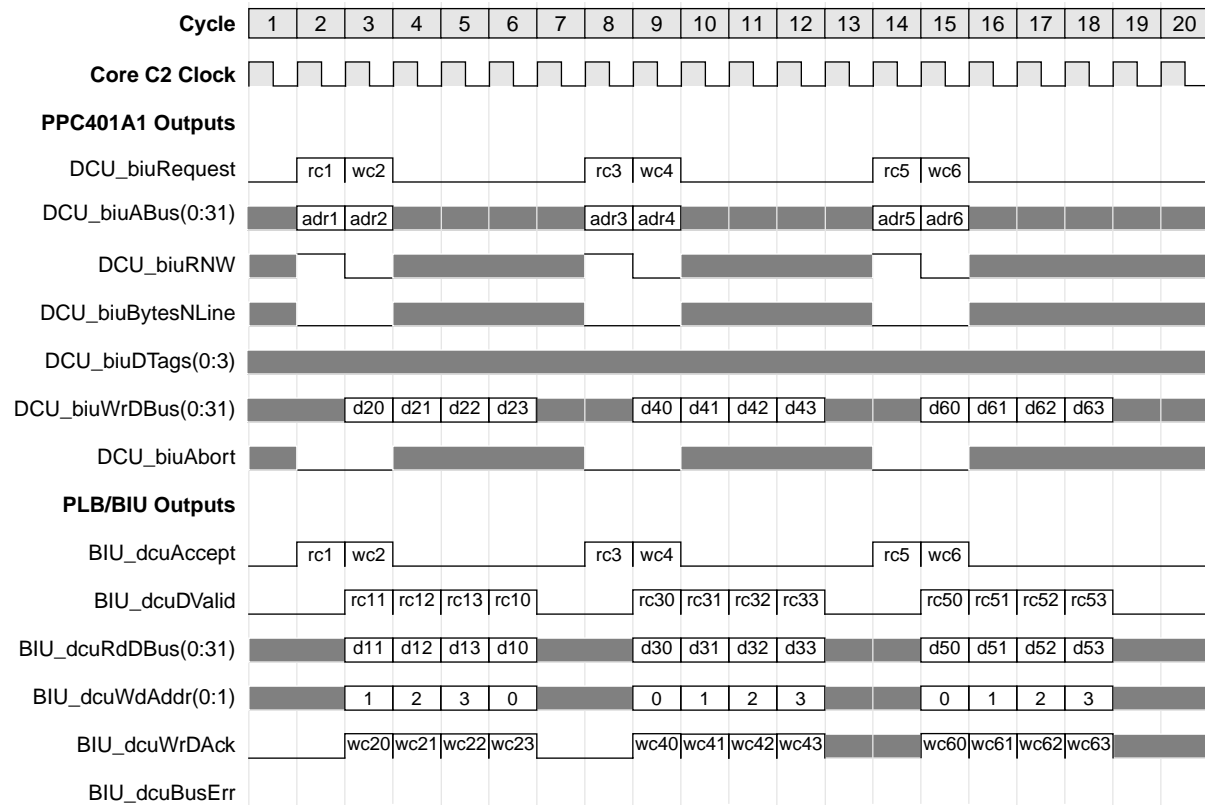


Figure 3-19. DCU/BIU-PLB Cacheable Mix

3.9.4.11 DCU/BIU-PLB Cacheable Write, Read Mix (Custom BIU)

Figure Highlights

- The fastest that a read cacheable request can follow a write cacheable request. (CBIU/PLB)
- Covers Sequence ID 8 from Table 3-19, “DCU–BIU/PLB Timing Diagram Reference,” on p. 3-60.
- The “DCU/BIU Timing Diagram Guidelines” on page 3-58 apply.

The sequence of instructions depicted below is: a dcbf (data cache block flush) that causes a flush of a cache line (cycles 3-6), then a cacheable read that causes a line fill (without a corresponding flush) (cycles 4-8) followed by another dcbf (cycles 11-14) and another cacheable read (cycles 12-16). Note that if the rc2 line fill caused a flush of a dirty cache line, that flush request would occur in cycle 8 (due to the use of the flush state machine in conjunction with the main state machine) and another cacheable read (such as rc4) request could follow in cycle 9.

If the transfers shown were done to a PLB-compliant BIU, the first RdDAck would be delayed a single cycle for each of the read cacheable transfers. This would shift the read activity for cycles 5-8 one cycle to the right, delaying wc3 into cycle 12. Also, if the target word is the last word transferred to the DCU, then the wc3 request of cycle 11 would not be asserted until cycle 12.

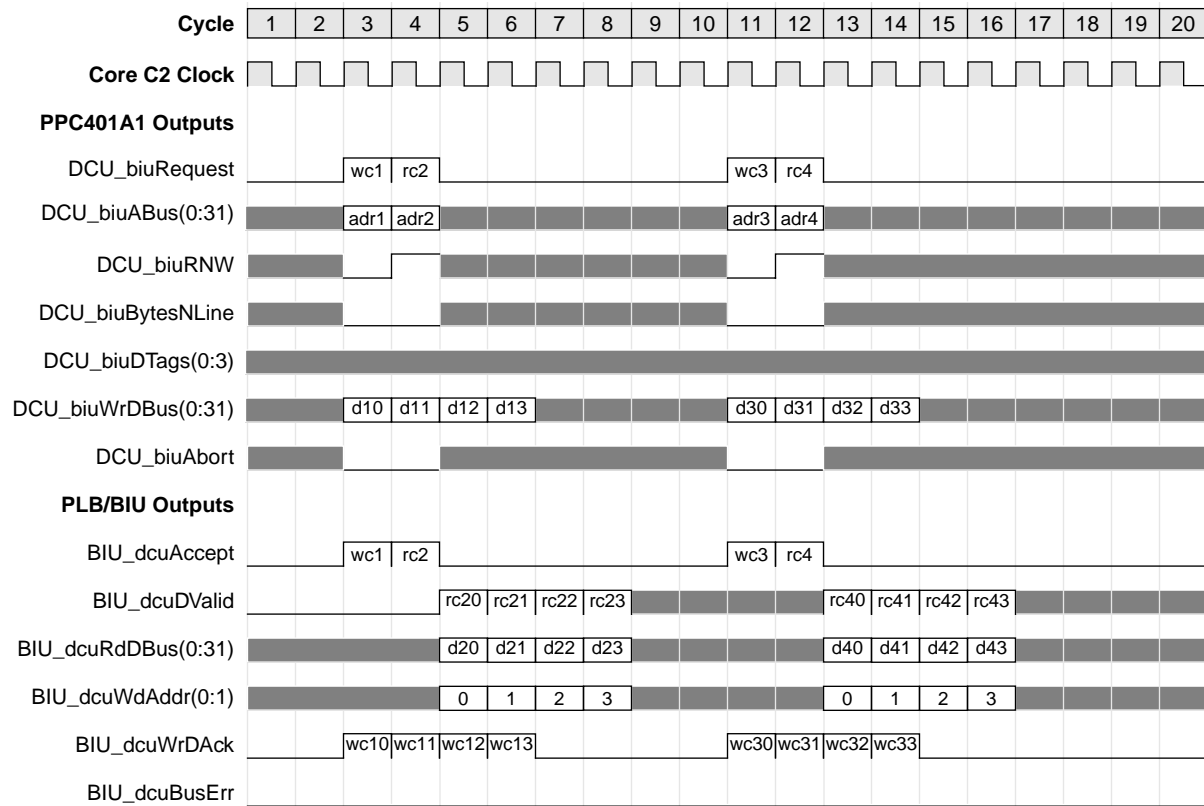


Figure 3-20. DCU/BIU-PLB Cacheable Write, Read Mix (Custom BIU)

3.9.4.12 DCU/BIU-PLB Cacheable, Non-cacheable Write Mix

Figure Highlights

- The fastest that a write cacheable request can follow a write non-cacheable (non write-through) request. (CBIU/PLB)
- The fastest that a write non-cacheable request can follow a write cacheable request. (CBIU/PLB)
- The fastest that back-to-back write cacheable requests can be presented. (CBIU/PLB)
- Covers the following Sequence IDs from Table 3-19, “DCU–BIU/PLB Timing Diagram Reference,” on p. 3-60: 2, 5, 6.
- The “DCU/BIU Timing Diagram Guidelines” on page 3-58 apply.

These diagrams apply for either a custom BIU or a PLB-compliant BIU.

The sequence of instructions depicted below is: Write cacheable (cycles 1-5), write non-cacheable (non-write-through), aligned word (cycles 6-7), write cacheable (cycles 8-12), write cacheable (cycles 13-17).

Cycles 6-9 depict the best case timing for a wnc to wc for a non write-through case. If wnc2 was a cacheable hit write-through request, then the wc3 request would be delayed an additional cycle, into cycle 10. For a cacheable miss write-through scenario see Figure 3-13. There, you will see that the wc3 request follows the wnc2 request one cycle *earlier* than shown here.

The write cacheables repeat as shown in cycles 7 (when the miss determination is made, causing the PLB request of cycle 9) through 12, when the first transfer completes and the next miss determination is made.

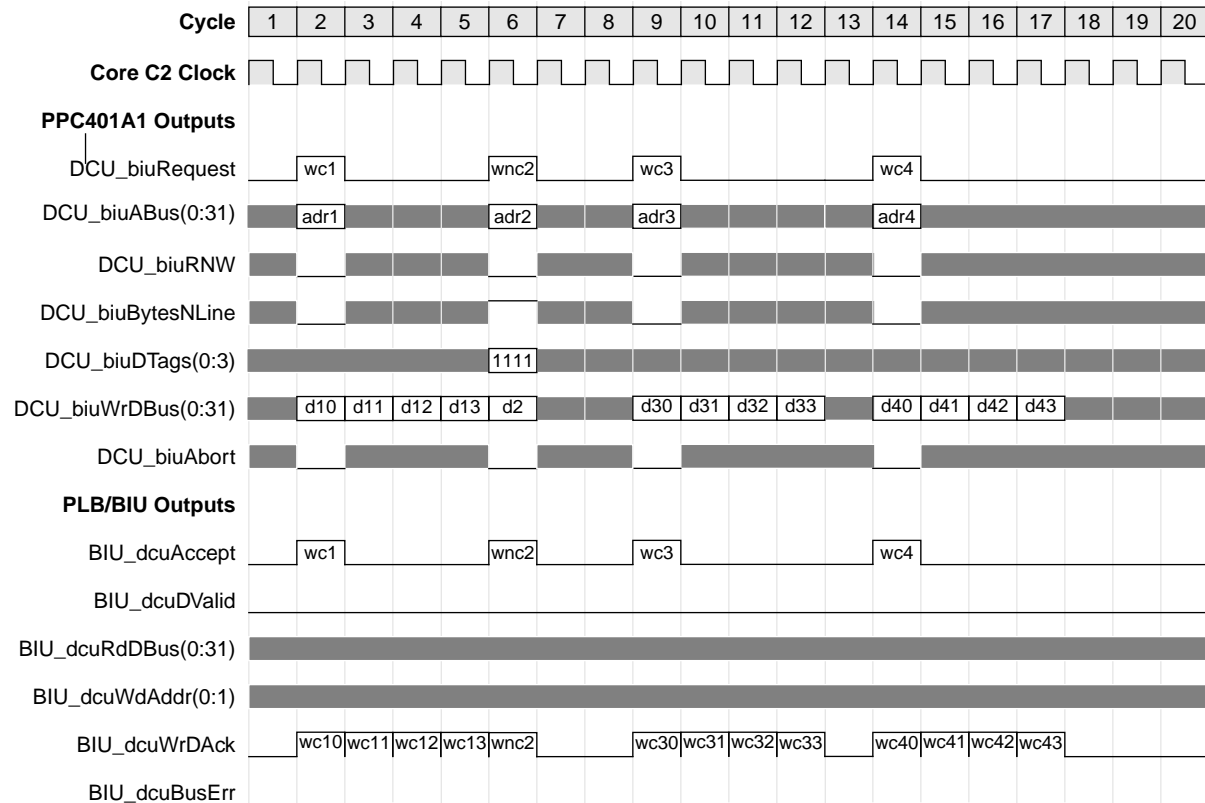


Figure 3-21. DCU/BIU-PLB Cacheable, Non-Cacheable Write Mix

3.10 Device Control Register (DCR) Interface

The DCR interface provides a mechanism for the CPU Core to set up non-core (peripheral) facilities that reside on-chip. For example, programmable facilities in an external bus interface unit may be configured for usage with external memory devices according to their transfer characteristics and address assignments. The DCR's are accessed through the use of the PowerPC instructions Move from DCR (**mfdcr**) and Move from DCR (**mtddcr**).

The DCR interface comprises an address bus, an input and output data bus, some controls that indicate if an operation is a read (move from) or write (move to) operation, and an acknowledge signal which indicates whether a move-to operation is complete, or whether move-from data is on the bus. The acknowledge signal is interlocked with the move-to or move-from signal so that the transfer take a minimum of three CPU cycles.

This interlock mechanism enables the DCR interface to be connected to peripheral units that are clocked at different frequencies than the core clock.

Engineering Note: The rising edge of the slower clock (either the CPU clock or the peripheral clock) must always correspond to the rising edge of the faster clock. This implies that the clocks for the ASIC logic that contains the DCR and the clocks for the PPC401A1 are derived from a common source. The common source can be multiplied or divided before being sent to the PPC401A1 or ASIC logic.

The address bus and outbound control signals are sent to each unit that requires a DCR interface. The data bus portion of the DCR interface is set up as a distributed mux in a ring that travels to each of the design units that requires a DCR interface. Each unit will either pass the data bus input along unmodified, or, if it is being addressed, place its information on the data bus. See Figure 3-23 for an example of DCR interface implementation on a Core+ASIC chip.

The acknowledge signal can be distributed similarly to the data bus, logically ORing the DCR acknowledge signal with the ack preceding acknowledge signal. Should this distribution prove to be timing prohibitive, all of the acknowledge signals can be ORed together and connected to the core's DCR_cpuDAck input. This is called combinatorial acknowledge.

If the The DCR logic does not return DCR_cpuDAck in 16 processor cycles, the processor times out. No error occurs on a DCR operation time out. The processor simply executes the next instruction.

Figure 3-22 provides a block diagram of the DCR interface.

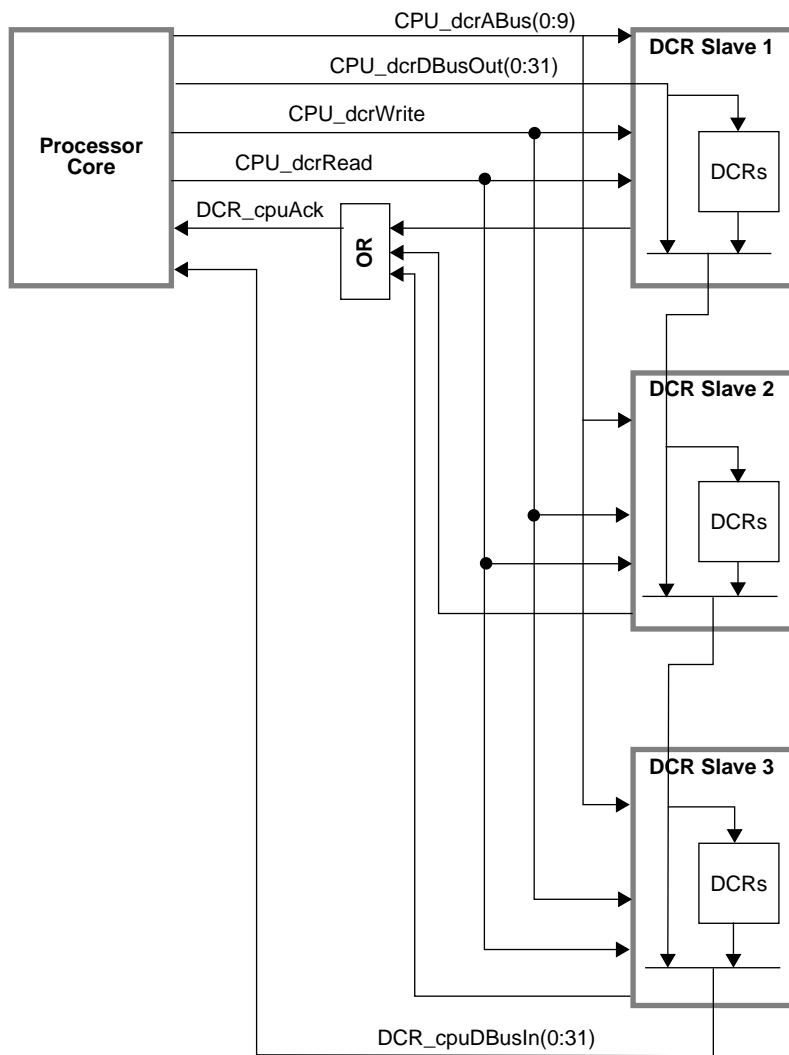


Figure 3-22. DCR Block Diagram

Figure 3-23 illustrates a logical implementation of the DCR bus interface. To ensure its reusability across several Core+ ASIC environment, it is recommended that all DCR slave logic containing DCRs adhere to the implementation specified here. This implementation allows the DCR slave to run at different clock speeds than the processor core. The bypass mux should be implemented to keep the path delay to a minimum when not selected. This function can be implemented as a 2-1 mux or equivalent logic.

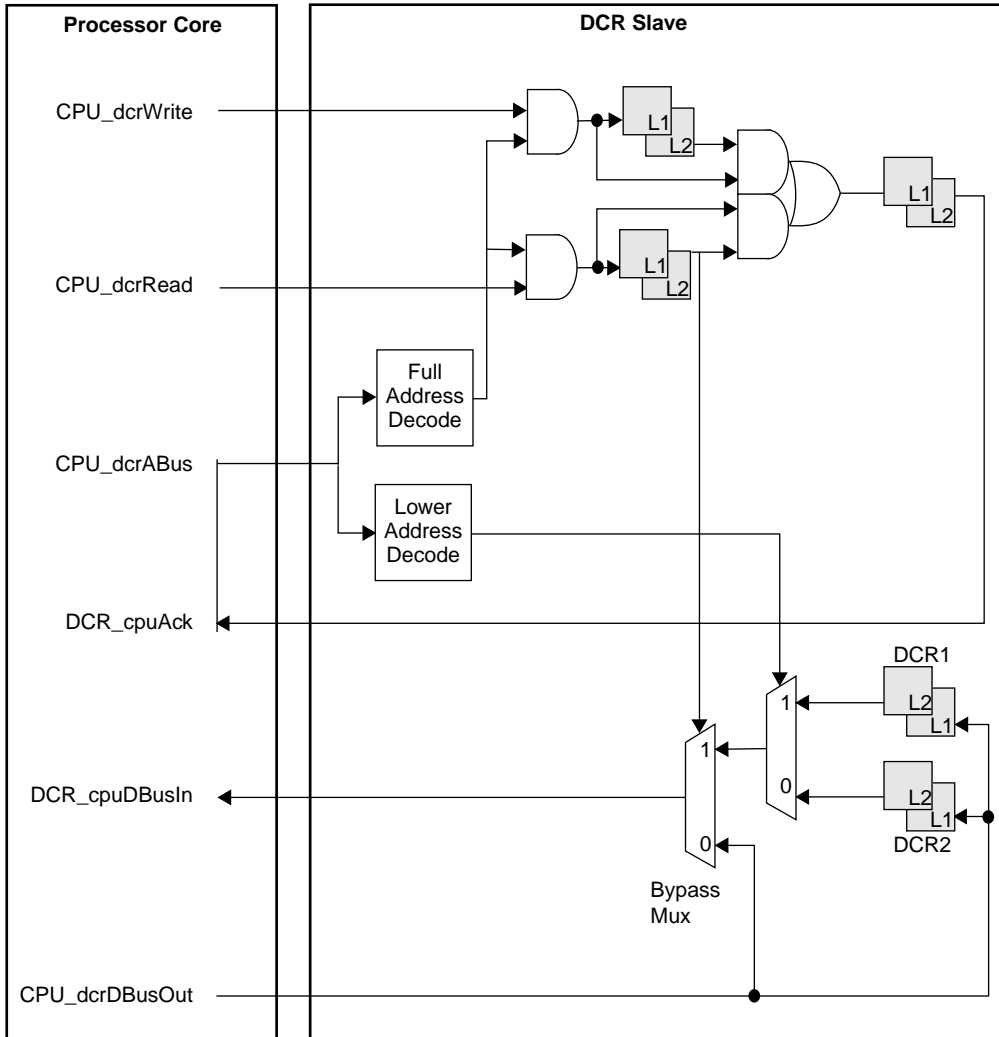


Figure 3-23. DCR Bus Implementation

Because the PPC401A1 is the only bus master on the bus, the address bus is driven by the PPC401A1 and received by each unit containing DCRs. The data bus could be designed as a distributed logical-OR, in a ring design, starting with the CPU to dcr Data Bus Output, continuing serially through each BIU device and terminating at the DCR to cpu Data Bus Input. This design will enable future expansion of the BIU without changing the I/O on either the Core or the existing BIU devices. The OR mechanism can be either an OR block or a mux, as appropriate. Each unit should only pick off the bits of the DCR bus that it requires. All unrequired bits should by-pass a unit.

Other DCR interface control signals, from the PPC401A1 to the BIU devices (the read (or mfdcr) signal and the Write (or mtdcr) signal) are wired the same as the address bus.

The acknowledge signal, which is driven from each unit containing DCRs to the PPC401A1, can be wired similarly to the data bus, as a ring, or simply ORed within the chip logic.

3.10.2 DCR Interface I/O Symbol

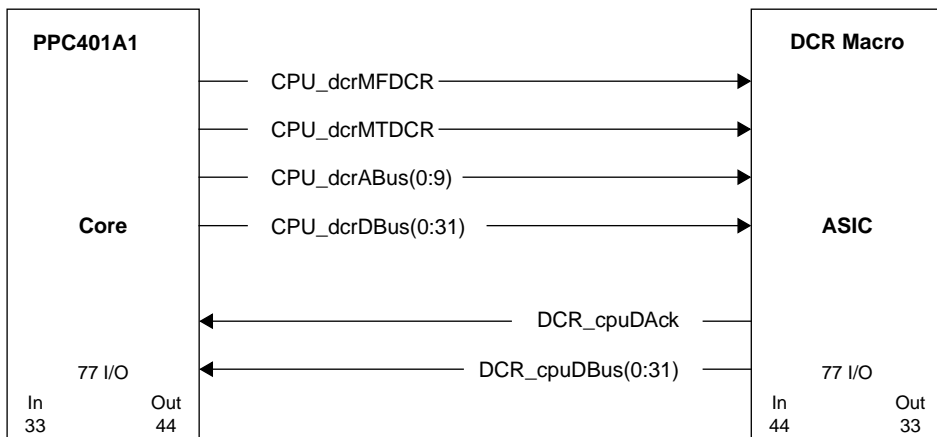


Figure 3-24. DCR Interface I/O Symbol

3.10.3 DCR Interface I/O Signal Table

Table 3-20 summarizes the DCR interface signals, which are described in detail in text following the table.

Table 3-20. DCR Interface I/O Signal Table

Signal	I/O Type	If Unused	Function
CPU_dcrMFDCRCPU_dcrRead	O	No connect	Indicates that the CPU is requesting to read data from a DCR into the CPU.
CPU_dcrMTDCRCPU_dcrWrite	O	No connect	Indicates that the CPU is requesting to write data from the CPU to a DCR.
CPU_dcrABus(0:9)	O	No connect	Indicates the address of the DCR which is being accessed for a read or write DCR operation.
CPU_dcrDBusOut(0:31)	O	No connect or wrap to DBus In	The 32-bit data bus output used to transfer write data from the CPU to a DCR. This bus also drives 32'h0 during DCR read operations.
DCR_cpuDAck	I	0	Indicates that the target DCR data is available on DCR_cpuDBus (read), or that the new DCR data is latched (write).
DCR_cpuDBusIn(0:31)	I	0x00000000 or wrapped DBus Out	The 32-bit data bus input used to transfer read data from a DCR to the CPU.

3.10.4 DCR Interface I/O Signal Descriptions

The following subsections describe the DCR interface I/O signals.

Each subsection heading names a signal and give its I/O type. Each subsection heading also provides the corresponding hard macro signal name, referred to as the core name.

3.10.4.1 CPU_dcrMFDCR (output)

Core Name: **DCRMVFR**

The CPU asserts CPU_dcrMFDCR to indicate that a move from DCR operation (**mfdcr** instruction) is in progress and that the address on CPUdcrABus(0:9) is valid. Once asserted, this operation cannot be interrupted. This signal indicates that the unit containing the DCR whose address corresponds to the address on the CPU_dcrABus(0:9) should drive that DCR data out onto the DCR data bus chain to be sent the CPU. The target DCR data must be passed, unchanged, around the rest of the chain and returned to the CPU using the DCR_cpuDBus(0:31) input.

Engineering Note: Unless the CPU_dcrMFDCR signal is active (or the ack for this mfdcr has not yet fallen), no DCR unit can drive data on the DCR bus ring. This is because the DCR bus is ORed with the internal SPR bus.

After CPU_dcrMFDCR is asserted, the CPU waits for up to 16 cycles for assertion of DCR_cpuDAck, which indicates that the mtdcr instruction is complete. If DCR_cpuDAck is not asserted within 16 cycles, the CPU times out and executes the next instruction.

CPU_dcrMFDCR is held inactive during reset.

3.10.4.2 CPU_dcrMTDCR (output)

Core Name: **DCRMVTO**

The CPU asserts CPU_dcrMTDCR to indicate that a move to DCR operation (**mtdcr** instruction) is in progress and that the address on CPUdcrABus(0:9) is valid. Once asserted, this operation is uninterruptable. This signal indicates that the unit which contains the DCR whose address corresponds to the address on the CPU_dcrABus(0:9) should latch the data on its DCR Data In input, which will be the data originally placed on the CPU_dcrDBus(0:31) by the CPU, into the target DCR. (See Figure 3-23.)

After CPU_dcrMTDCR is asserted, the CPU waits for up to 16 cycles for assertion of DCR_cpuDAck, which indicates that the **mtdcr** instruction is complete. If DCR_cpuDAck is not asserted within 16 cycles, the CPU times out and then executes the next instruction.

CPU_dcrMTDCR is held inactive during reset.

3.10.4.3 CPU_dcrABus(0:9) (output)

Core Name: **DCRADRn**

CPUdcrABus(0:9) is the DCR address bus, which is valid when CPU_dcrMTDCR or CPU_dcrMFDCR is asserted and invalid at all other times. External DCR slave logic must qualify this bus with CPU_dcrMTDCR or CPU_dcrMFDCR. The DCR address remains stable during mtdcr and mfdcr execution.

3.10.4.4 CPU_dcrDBus(0:31) (output)

Core Name: **DCRDOnn**

CPU_dcrDBus(0:31) is the DCR write data bus and is driven by the CPU to the first DCR chain. This bus is only valid when the CPU_dcrMFDCR or the CPU_dcrMTDCR signals are asserted.

For a mtdcr (move to DCR) operation, the CPU will drive the data to be written at the target DCR on this bus. For a mfdcr (move from DCR) operation, the CPU will drive a 32 bit hex value of 0x00000000 on this data out bus. In this way, if a unit's DCR is being read from and it is only an 8-bit DCR, the unit does not have to drive out the remaining 24 bits of 0s. The unit should source or sink only the bits of the DCR data bus that it requires.

During reset, the CPU drives CPU_dcrDBus(0:31) with 0x00000000. The DCR slave logic can use this to initialize the DCRs.

3.10.4.5 DCR_cpuDAck (input)

Core Name: **DCRDACK**

DCR_cpuDAck, the DCR transfer acknowledge signal, is driven by the unit that contains the targeted DCR back to the CPU to indicate that the requested DCR operation is being completed.

For a **mtdcr** operation, the **DCR_cpuDAck** signal should be asserted when the DCR has been updated, that is, on the cycle that the DCR latch output changes. The **DCR_cpuDAck** signal should not be deasserted until the fall of the **CPU_dcrMTDCR** signal. It may be deasserted on the next ASIC clock edge.

For a **mfdcr** operation, the **DCR_cpuDAck** signal should be asserted to indicate that valid DCR data is being presented on the DCR data bus, which will feed the core's **DCR_cpuDBus(0:31)** input. The **DCR_cpuDAck** signal should not be deasserted until the fall of the **CPU_dcrMFDCR** signal. It may be deasserted on the next ASIC clock edge.

DCR_cpuDAck should be deasserted in the cycle following the deassertion of **CPU_dcrMTDCR** or **CPU_dcrMFDCR**. Note that the CPU waits for **DCR_cpuDAck** to be deasserted for the current **mtdcr** or **mfdcr** instruction before starting a new **mtdcr** or **mfdcr** instruction.

3.10.4.6 DCR_cpuDBus(0:31) (input)

Core Name: **DCRDInn**

DCR_cpuDBus(0:31) is the DCR read data bus and represents the DCR bus as driven through the last unit on the DCR chain back to the CPU.

Engineering Note: It is important to note that unless the **MFDCR** signal is active (or the ack for this **mfdcr** has not yet fallen) , no DCR unit is allowed to drive data on the DCR bus ring. This is because the DCR bus is ORed with the PPC401A1 internal SPR bus.

For a **mfdcr** (move from DCR) operation, the CPU drives a 32-bit hex value, 0x00000000, on the data out bus. This value passes unchanged through all units that precede the target DCR unit. The unit containing the target DCR places the contents of the target DCR on the data bus for circulation through the remainder of the chain. The contents of the target DCR data are passed unchanged through all units that follow the target DCR unit, eventually returning the data to the CPU for processing using **DCR_cpuDBus(0:31)**. The ASIC logic must drive the addressed DCR data as long as **CPU_dcrMFDCR** and **DCR_cpuDAck** are active, and can optionally drive the addressed DCR data as long as only **DCR_cpuDAck** is active. Data must be valid when **DCR_cpuDAck** is asserted.

3.10.5 DCR Interface I/O Timing Diagrams

The following timing diagrams show several mode 0 scenarios and the only mode 1 scenario.

Each diagram shows the fastest possible access for the given scenario, using the recommended DCR interface implementation.

Also, note that each diagram shows a **mfdcr** (move from dcr) using **CPU_dcrMFDCR**, and a **mtdcr** (move to dcr) using **CPU_dcrMTDCR**, occurring concurrently. This cannot be done in the hardware! **mfdcr** and **mtdcr** operations cannot overlap. This is shown only to reduce the number of DCR timing diagrams.

3.10.5.1 Core and ASIC at Same Frequency, with Combinatorial DCR_cpuAck

In this scenario, the fastest back-to-back access is four cycles.

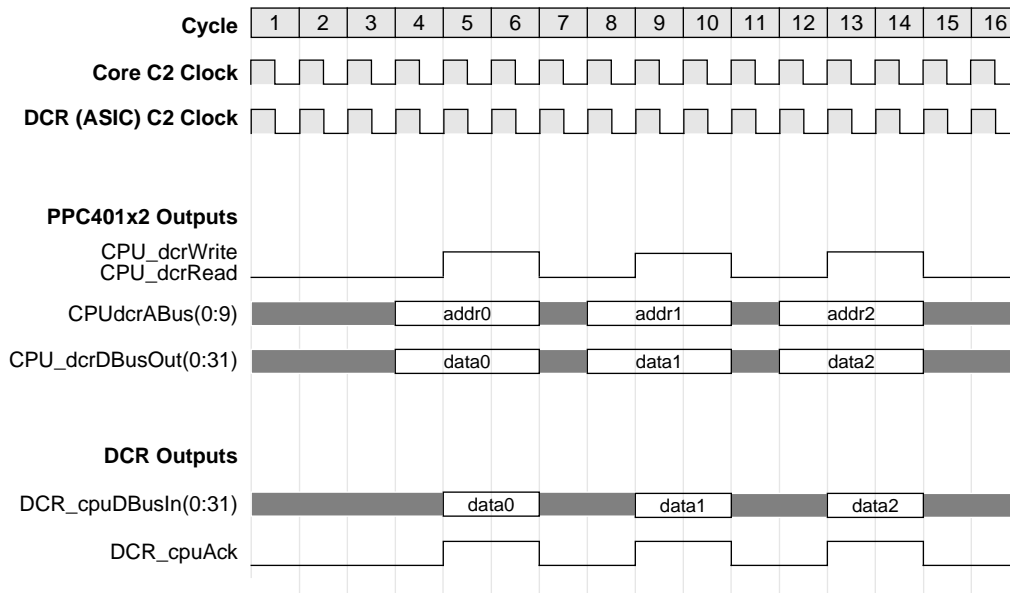


Figure 3-25. Same clocks, Combinatorial Acknowledge

3.10.5.2 Core and ASIC at Same Frequency, with Latched DCR_cpuAck

In this scenario, the fastest back-to-back access is every seven cycles.

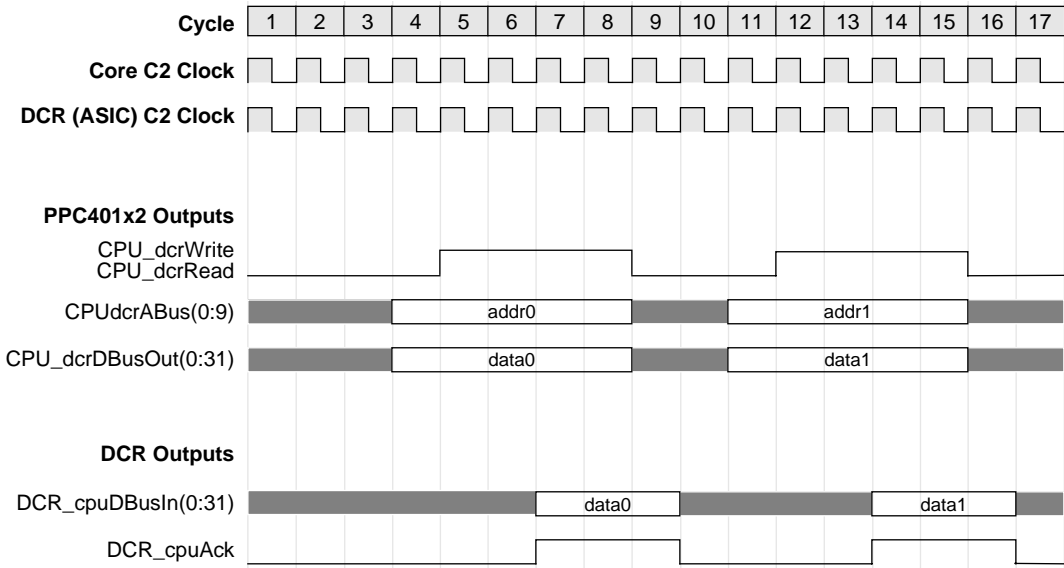


Figure 3-26. Same clocks, Latched DCR_cpuAck

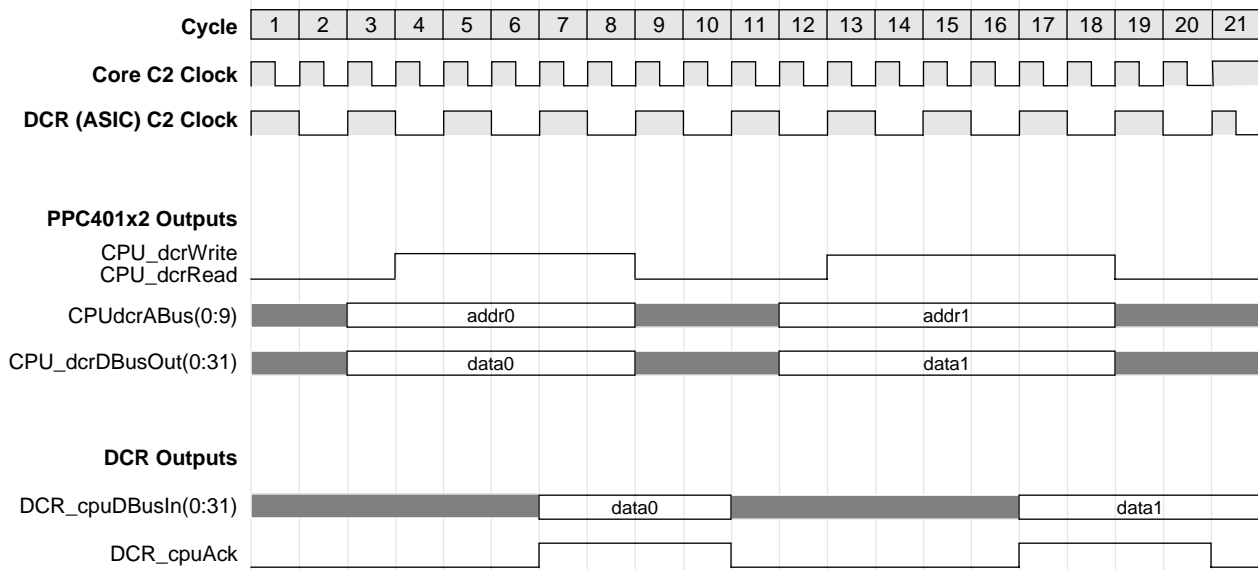


Figure 3-27. Clock-doubled, Latched DCR_cpuAck

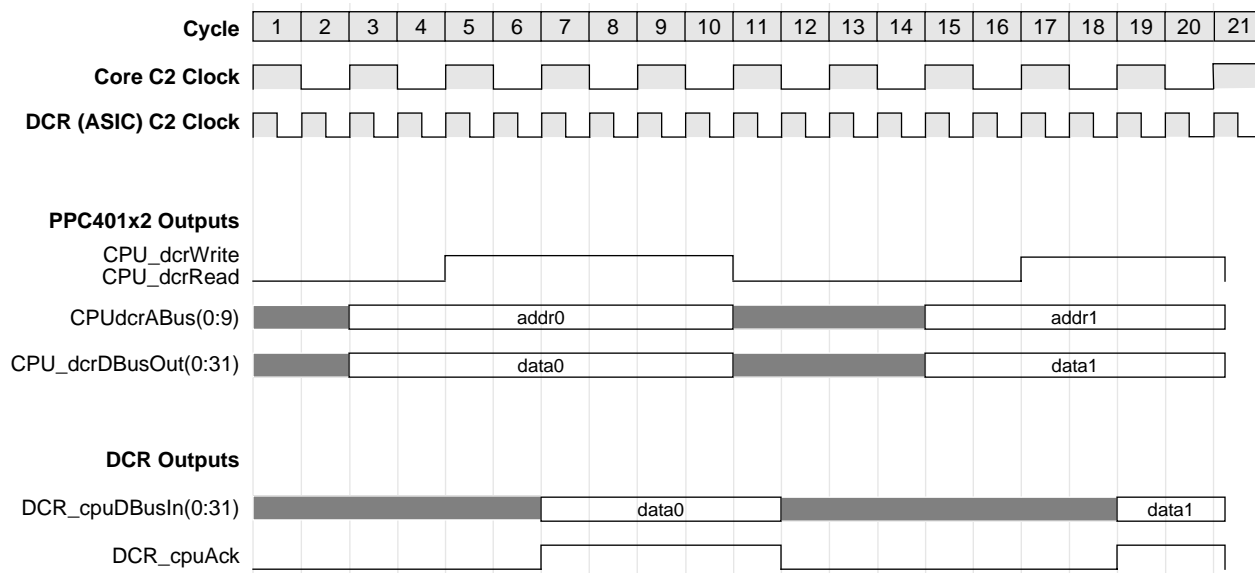


Figure 3-28. Clock-doubled ASIC, Latched DCU_cpuAck

3.11 External Interrupt Controller (EIC) Interface

The IBM PowerPC Embedded Environment defines two architected interrupts, the external interrupt and the critical interrupt. Off-core non-critical interrupts should feed into the PPC401A1 through the EIC_cpuNonCritical signal. Off-core critical exceptions should feed into the PPC401A1 through the EIC_cpuCritical signal.

The external interrupt uses Save/Restore Register 0 (SRR0) and SRR1. The external interrupt is pending when a non-critical exception occurs and the interrupt enable bit for that exception is asserted. MSR[EE] enables interrupts received by the EIC_cpuNonCritical input. The non-critical exceptions that are enabled by the MSR[EE] bit are shown in Table 3-21.

Table 3-21. Non-Critical Category Exceptions Enabled by MSR[EE]

Non-critical Interrupt - Use SRR0, SRR1 - Interrupt Vector: x4000		
Non-critical Category Exceptions	On/Off Core	Description
External input interrupt request	Off	A logical OR of non-critical exceptions from outside the core, sent to the core using the EIC_cpuNonCritical signal.
FIT	On	Fixed Interval Timer
PIT	On	Programmable Interval Timer

The critical interrupt uses Save/Restore Register 2 (SRR2) and Save/Restore Register 3 (SRR3). The critical interrupt is pending when a critical exception has occurred and the interrupt enable bit for that exception is asserted. MSR[CE] enables interrupts received by the EIC_cpuCritical input. The critical exceptions enabled by the MSR[CE] bit are shown in Table 3-22.

Table 3-22. Critical Category Exceptions Enabled by MSR[CE]

Critical Interrupt - Use SRR2, SRR3 - Interrupt Vector: x7000		
Critical Category Exceptions	On/Off Core	Description
Critical input interrupt request	Off	A logical OR of critical exceptions from outside the core, sent to the core using the EIC_cpuCritical signal.
Watchdog timer expiration	On	Second watchdog timer expiration

The EIC logic can be responsive to edge- or level-sensitive exceptions by the ASIC designer. As for two inputs that feed into the PPC401A1, once either signal is asserted, it must remain asserted until it is turned off by system software.

3.11.1 EIC Interface I/O Symbol

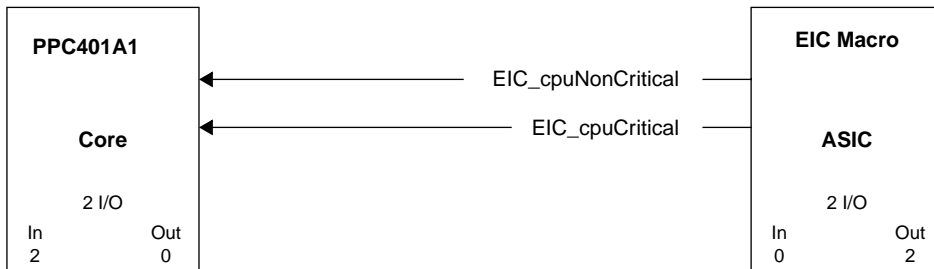


Figure 3-29. EIC Interface I/O Symbol

3.11.2 EIC Interface I/O Signal Table

Table 3-23 summarizes the EIC interface signals, which are described in detail in text following the table.

Table 3-23. EIC Interface I/O Signal Summary

Signal	I/O Type	If Unused	Function
EIC_cpuNonCriticalEIC_cpuExtInputIRQ	I	0	Indicates that an external non-critical exception occurred.
EIC_cpuCriticalEIC_cpuCritInputIRQ	I	0	Indicates that a critical exception occurred.

3.11.3 EIC Interface I/O Signal Descriptions

The following subsections describe the ISOCM interface signals.

Each subsection heading names the topic signal and give its I/O type. Each subsection heading also provides the hard macro signal cross-reference, referred to as the core name.

3.11.3.1 EIC_cpuNonCritical (input)

Core Name: **EIINTNC**

This input is driven by the External Interrupt Controller (EIC) unit to the CPU to indicate that an External exception, whose reporting is enabled, has occurred in logic outside of the core. Once this signal is asserted, it must remain asserted until it is turned off by system software. By definition, the External exception is a non-critical category exception. Conceptually, this input is the logical OR of all customer specific non-critical category exceptions that may occur outside of the core.

If the EXT_cpuNonCritical input signal is asserted and the MSR[EE] bit is or has been asserted, an External Interrupt will be pending. It is indeterminate when the interrupt will be

taken, since the core may be committed to perform other operations. There is a minimum of four cycles from assertion of the EXT_cpuNonCritical signal (cycle1) to the interrupt's target instruction dropping into the decode stage of the pipeline (cycle 4).

3.11.3.2 EIC_cpuCritical (input)

Core Name: EIINTCR

This input is driven by the External Interrupt Controller (EIC) unit to the CPU to indicate that a Critical exception, whose reporting is enabled, has occurred in logic outside of the core. Once this signal is asserted, it must remain asserted until it is turned off by system software. By definition, the Critical exception is a critical category exception. Conceptually, this input is the logical OR of all customer specific critical category exceptions that may occur outside of the core.

If the EXT_cpuCritical input signal is asserted and the MSR[CE] bit is or has been asserted, a Critical Interrupt will be pending. It is indeterminate when the interrupt will be taken, since the core may be committed to perform other operations. There is a minimum of four cycles from assertion of the EXT_cpuCritical signal (cycle1) to the interrupt's target instruction dropping into the decode stage of the pipeline (cycle 4).

3.12 JTAG Interface

The JTAG interface provides basic JTAG chip testing functionality. It also provides the ability for an external debug tool, such as RISCWatch, to gain control of the processor for debug purposes.

With one exception, the JTAG interface follows IEEE Std 1149.1, which defines a test access port (TAP) and boundary scan architecture. A reset input to the PPC401A1, `IOS_jtagResetIn`, which can be connected (through an inverter) to the JTAG $\overline{\text{TRST}}$ chip input, *must* be connected for resetting JTAG/Debug logic at power-on reset (POR). In the standard, $\overline{\text{TRST}}$ is listed as an optional signal. For the PPC401A1, this input is required. Refer to the definition of the reset signal in this section and also to the example on-chip reset connection illustrated in Figure 3-4 for details. Other than this exception, the JTAG interface supports the minimum functionality of IEEE Std 1149.1 and supports some user-specific instructions, as allowed by the standard, that provide the ability to gain control of the processor for debug.

If the ASIC designer wishes to incorporate optional features of IEEE Std 1149.1, such as the use of a Device ID Register, a parallel TAP port must be built to work with the PPC401A1., and PPC401A1-specific instruction decodes must be identified so that the ASIC design can combine PPC401A1 functions with additional desired functions. Contact IBM Embedded PowerPC Technical Support (919/543-5701; ppcsupp@raleigh.ibm.com) for details.

To use the JTAG interface to connect to RISCWatch, see the description of the `IOS_dbgDebugHalt` signal in Section 3.13.3.1 on p. 3-109. This signal provides additional debug capabilities and may be required, depending on whether clock management is implemented. The `debugHalt` signal is an optional pin on the RISCWatch connector.

3.12.1 JTAG Interface I/O Symbol

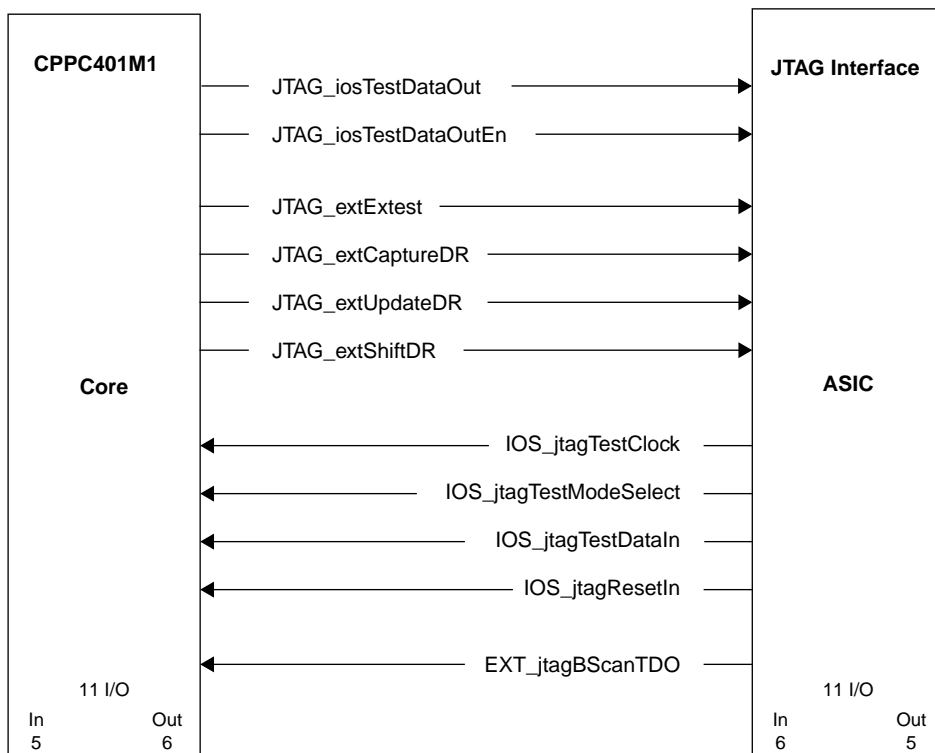


Figure 3-30. JTAG Interface I/O Symbol

3.12.2 JTAG Interface I/O Signal Table

Table 3-24 summarizes the JTAG interface signals, which are described in detail in text following the table.

Table 3-24. JTAG Interface I/O Signal Summary

Signal	I/O Type	If Unused	Function
IOS_jtagTestClock(TCK)	I	See IEEE 1149.1	JTAG Test Clock (TCK). The core's JTAG logic source clock. This input is also used as an LSSD B clock during Core_LSSD mode.
IOS_jtagTestModeSelect(TMS)	I	1	JTAG Test Mode Select (TMS). Determines the mode in which the TAP operates.

Table 3-24. JTAG Interface I/O Signal Summary

Signal	I/O Type	If Unused	Function
IOS_jtagTestDataIn(TDI)	I	1	JTAG Test Data In (TDI). JTAG serial data in port.
IOS_jtagResetIn	I	Required	JTAG reset input MUST be asserted at POR. May be used as a positive active TRST thereafter.
JTAG_iosTestDataOut(TDO)	O	No connect	JTAG Test Data Out (TDO). JTAG serial data out port.
JTAG_iosTestDataOutEn	O	No connect	The core's driver enable signal for the JTAG TDO signal.
EXT_jtagBScanTDO	I	0	Input fed from the last JTAG boundary scan latch's ScanOut port. (Feeds the TDO signal when scanning.)
JTAG_extExtest	O	No connect	Indicates that the JTAG <i>EXTTEST</i> instruction is selected.
JTAG_extCaptureDR	O	No connect	Indicates that the JTAG <i>Capture-DR</i> controller state is selected.
JTAG_extShiftDR	O	No connect	Indicates that the JTAG <i>Shift-DR</i> controller state is selected.
JTAG_extUpdateDR	O	No connect	Indicates that the JTAG <i>Update-DR</i> controller state is selected.

3.12.3 JTAG Interface I/O Signal Descriptions

The following subsections describe the ISOCM interface signals.

Each subsection heading names a signal and give its I/O type. Each subsection heading also provides the corresponding hard macro signal name, referred to as the core name.

3.12.3.1 IOS_jtagTestClock(TCK) (input)

Core Name: JTTCLK

This JTAG Test Clock is the source clock for the PPC401A1 TAP. The maximum clock rate into the TCK is one half of the processor SysClk rate.

This clock is independent of the system clock(s) for the chip so that test operations can be synchronized between the various chips on a printed wiring board. Both the rising and falling edges of the clock are significant. The rising edge is used to load signals applied at the TAP input pins (Test Mode Select (TMS) and Test Data Input (TDI)), while the falling edge is used to clock signals out through the TAP Test Data Output (TDO) pin.

This signal also provides the LSSD B Clock which is used to latch the L1 data output into the L2 portion of a latch during Core_LSSD Test mode for the PPC401A1 logic that is normally clocked by the JTAG TCK clock input. This signal will be forced to a logic 0 internally during ASIC_LSSD mode to satisfy test requirements that this clock be inactive to the core's internal scan chains during ASIC_LSSD mode.

3.12.3.2 IOS_jtagTestModeSelect(TMS) (input)

Core Name: **JTTMS**

This JTAG Test Mode Select pin is sampled by the TAP on the rising edge of TCK. The TAP state machine uses the TMS pin to determine the mode in which the TAP operates.

The operation of the test logic is controlled by the sequence of '1's and '0's applied at this input, with the signal value typically changing on the falling edge of **TCK**. This signal sequence is fed to the TAP controller which samples the value at **TMS** on each rising edge of **TCK** and uses this information to generate the clock and control signals required by the other test logic blocks. On the chip, TMS should be equipped with a pull-up resistor or otherwise designed such that, when it is not driven from an external source, the test logic perceives a logic '1'.

3.12.3.3 IOS_jtagTestDataIn(TDI) (input)

Core Name: **JTTDI**

This JTAG Test Data Input signal is used to input serial data into the TAP. When the TAP enables the use of the TDI signal, the TDI signal is sampled on the rising edge of TCK and this data is input to the selected TAP shift register.

Data applied at this serial input is fed into the instruction register or into a test data register, depending on the sequence previously applied at TMS. Typically, the signal applied at TDI will be controlled to change state following the falling edge of TCK, while the registers shift in the value received on the rising edge. Like TMS, TDI should be equipped with a pull-up resistor or otherwise designed such that, when it is not driven from an external source, the test logic perceives a logic '1'.

3.12.3.4 IOS_jtagResetIn (input)

Core Name: **JTRSTIN**

This JTAG Reset In pin provides for asynchronous reset of the TAP controller. If this signal is asserted to a logic 1, the TAP controller will be asynchronously reset to the Test-Logic-Reset controller state. This signal is **positive** active at the core boundary. Its usage is mandatory in order to reset JTAG/Debug facilities in the PPC401A1.

$\overline{\text{TRST}}$ (called TRST* in the IEEE 1149.1 standard) is a negative active signal at the chip boundary. The ASIC designer can connect this signal (through an inverter for polarity adjustment) with a $\overline{\text{TRST}}$ chip input pin. The $\overline{\text{TRST}}$ pin and the chip reset input that feeds into the PPC401A1 system reset input (See Section 3.12.3.4 on p. 3-105) *must both* be active at power-on reset (POR) to clear any logic that could put the PPC401A1 into a stop state or continuous reset request state. After that, separate control of the PPC401A1 JTAG logic reset and non-JTAG logic reset can be accomplished. The system designer must carefully determine how to make the JTAG reset logically responsive to the system and chip resets, depending on the debug requirements and debug tool requiremnts.

This signal must be active during POR to initialize internal JTAG interface logic (that clocked by the JTAG TCK) within the PPC401A1 to a state that guarantees non-interference with the operation of the PPC401A1. If the internal JTAG interface logic is not reset, it might possibly power on in a state that is telling the core to stop the processor or to issue a system reset,

rendering the core unable to process instructions. If this happens, the condition will persist until this reset signal is activated and the responsible JTAG internal logic gets reset.

3.12.3.5 JTAG_iosTestDataOut(TDO) (output)

Core Name: **JTTDO**

This JTAG Test Data Out pin is used to transmit data from the PPC401A1 TAP. Data from the selected TAP shift register is shifted out on TDO.

This serial output from the test logic is fed from the instruction register or from a test data register depending on the sequence previously applied at TMS. During shifting, data applied at TDI will appear at TDO after a number of cycles of TCK determined by the length of the register included in the serial path. The signal driven through TDO changes state following the falling edge of TCK. When data is not being shifted through the chip, TDO should be placed in high impedance.

3.12.3.6 JTAG_iosTestDataOutEn (output)

Core Name: **JTTDOEN**

This signal is the driver enable signal to the ASIC chip's three-state driver for the Test Data Out (TDO) signal.

3.12.3.7 EXT_jtagBScanTDO (input)

Core Name: **JTBNDISI**

This input should be fed by the last JTAG boundary scan register scan output for the Core+ASIC chip. When properly instructed by the TAP controller, this input is sent to the Test Data Out (TDO) output for boundary scan observability. The boundary scan ring comprises the TDI chip input going to the first boundary scan element, whose output goes to the next boundary scan element, and so on, until the last boundary scan element sends its output into this core input for transmission off chip across TDO.

3.12.3.8 JTAG_extExttest (output)

Core Name: **JTEXTST**

This signal is the JTAG standard *EXTEST* instruction which allows testing of off-chip circuitry and board level interconnections using JTAG boundary scan. (Note that the terminology "off-chip circuitry and board level interconnections" may need to be modified in a system-on-a-chip environment.)

3.12.3.9 JTAG_extCaptureDR (output)

Core Name: **JTCAPDR**

This signal is a JTAG standard state signal that is used by the JTAG boundary scan logic to allow data to be parallel-loaded into test data registers selected by the current instruction.

3.12.3.10 JTAG_extShiftDR (output)

Core Name: **JTSHFDR**

This signal is a JTAG standard state signal that is used by the JTAG boundary scan logic to shift data one stage towards its serial output on each rising edge of TCK.

3.12.3.11 JTAG_extUpdateDR (output)

Core Name: JTUPDDR

This signal is a JTAG standard state signal that is used by the JTAG boundary scan logic to allow data to be latched onto the parallel outputs of certain test data registers from the shift-register path.

3.13 Debug (DBG) Interface

The Debug interface inputs into the PPC401A1 may be used to provide additional debug enhancements for the ASIC customer. The signals on this interface may be used to provide information and control to an external debug tool, such as RISCWatch.

3.13.1 DBG Interface I/O Symbol

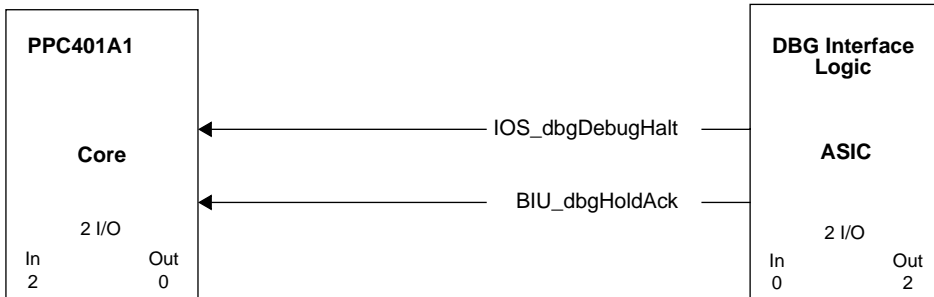


Figure 3-31. DBG Interface I/O Symbol

3.13.2 DBG Interface I/O Signal Table

Table 3-25 summarizes the DBG interface signals, which are described in detail in text following the table.

Table 3-25. DBG Interface I/O Signal Summary

Signal	I/O Type	If Unused	Function
IOS_dbgDebugHalt	I	0	Indicates that an external (to the core) source is requesting to stop the processor for debug purposes.
		0	Feeds the UDE bit of the DBSR. Allows customer debug logic to interrupt normal CPU flow.

3.13.3 DBG Interface I/O Signal Descriptions

The following subsections describe the ISOCM interface signals.

Each subsection heading names the topic signal and give its I/O type. Each subsection heading also provides the hard macro signal cross-reference, referred to as the core name.

3.13.3.1 IOS_dbgDebugHalt (input)

Core Name: **DBHALT**

This signal enables an external source to stop the processor. It is intended to be taken out to a chip pin to allow an external debugger, such as RISCWatch, to request that the processor halt its instruction processing so that the external debugger can then get control of the processor. External debuggers can also issue a stop command to the PPC401A1 via the JTAG interface, however, this stop request is cleared when the PPC401A1 is reset and the external debug tool will need to re-acquire control while the processor is fetching instructions. On the contrary, when using debugHalt to stop the processor, a PPC401A1 reset will NOT cause the debugHalt control signal to reset and the processor will be stopped at the reset vector. The debugHalt chip input for on the RISCWatch connector is negative active and will need to be inverted (and synchronized to the core clock) before being brought into this positive active core input.

In the event that the Core+ASIC chip has clock control circuitry and the clocks to the core are turned off (CPM_cpuCoreClkOff is set to a logic 1), the debugHalt signal should be used by an external debugger, such as RISCWatch, as a mechanism to alert Clock and Power Management control logic to re-enable clocks in order to perform RISCWatch debug activity. If clock control circuitry exists that can prevent the core from getting clocks, and this circuitry may be active when RISCWatch debug activity will be done, then the debugHalt signal will be required to re-enable clocks to the core.

When the debugHalt signal is deasserted (and no Stop Request is active via the JTAG interface) the chip should return to the sleep mode it was in prior to the assertion of debugHalt by RISCWatch, as long as no other condition that would cause the chip to leave that sleep mode is preventing it from doing so.

3.13.3.2 BIU_dbgHoldAck (input)

Core Name: **DBHLDK**

This Hold Acknowledge signal is a logic 1 when the BIU relinquishes its external buses to an external master. If the BIU raises this signal, it is an indication to the DBG unit that the processor may not get the external bus as soon as it otherwise would. This signal was put in for the RISCWatch tool (or similar debug tool) software to allow it to understand that the processor may not be responding as timely as expected due to the BIU devoting the external bus to other bus masters.

For BIU designs that do not have external bus master support, this signal should be tied to a logic 0.

4

Initialization

This chapter describes the initial state of the processor after a reset, and contains an example of the initialization code required to begin executing application code. Initialization of external system components or system-specific chip facilities may also need to be performed in addition to the basic initialization described in this chapter.

Section 3.7, “Reset Interface,” on p. 3-20, describes the PPC401A1 reset interface.

4.1 Processor State After Reset

After a reset, the contents of Special Purpose Registers (SPRs) control the initial processor state. Chapter 10, “Register Summary,” contains descriptions of the registers.

In general, the contents of SPRs are undefined after a reset. Reset initializes the minimum number of SPR fields required for allow successful instruction fetching. System software fully configures the processor.

The MCI field of the Exception Syndrome Register (ESR) is cleared so that it can be determined if there has been a machine check during initialization, before Machine Check exceptions are enabled.

Two SPRs contain status on the type of reset that has occurred. The DBSR contains the most recent reset type. The TSR contains the most recent watchdog reset.

Table 4-1 shows the reset state of the initialized SPR fields.

Table 4-1. Contents of Registers After Reset

Register	Field	Core Reset	Comment
MSR	WE	0	Wait state disabled
	CE	0	Critical interrupts disabled
	EE	0	External interrupts disabled
	PR	0	Supervisor mode
	ME	0	Machine check interrupts disabled
	DE	0	Debug interrupts disabled

Table 4-1. Contents of Registers After Reset

Register	Field	Core Reset	Comment
	LE	0	Big endian
DBCR	EDM	0	Debug mode and events disabled
	RST	00	No reset action
DBSR	MRR	Most recent reset type	Most recent reset type
ESR	MCI	0	Machine check exception has not occurred
ICCR	S0:S31	0x00000000	Instruction cache is disabled
PVR	FAM MEM CORE CHIP	0x002 0x1 0x0 0xnnn	Processor Family
SGR	G0:G31	0xFFFFFFFF	Storage is guarded
SLER	S0:S31	0x00000000	Storage is big endian
TCR	WRC	00	Watchdog timer reset disabled
TSR	WRS	Copy of TCR[WRC]	If reset caused by watchdog timer
		Undefined	After Power-up
		Unchanged	If reset not caused by watchdog timer

4.2 PPC401A1 Initial Processor Sequencing

After any reset, the processor core fetches the word at address 0xFFFFFFFFC and attempts to execute it. The instruction at 0xFFFFFFFFC is typically a branch to initialization code. Because the processor is initialized in big endian mode, initialization code must be in big endian format until the SLER is configured otherwise. Unless the instruction at 0xFFFFFFFFC is an unconditional branch, fetching can wrap to address 0x00000000 and attempt to execute the instruction at this location.

The system must provide non-volatile memory or memory initialized by some other mechanism, external to the processor, at location 0xFFFFFFFFC and at the location of the initialization code, before a reset.

4.3 Initialization Requirements

When any reset is performed, the processor is initialized to a minimum configuration to start executing initialization code. Initialization code is necessary to complete the processor and system configuration.

The initialization code example in this section performs the configuration tasks required to prepare the PPC401A1 to boot an operating system or run an application program.

Some portions of the initialization code work with system components that are beyond the scope of this manual.

Initialization code should perform the following tasks to configure the processor resources.

To improve instruction fetching performance:

- Initialize the SGR appropriately for guarded or unguarded storage. Since all storage is initially guarded and speculative fetching is inhibited to guarded storage, reprogramming the SGR will improve performance for unguarded regions.

Configure the following storage attribute control registers, if necessary:

- Initialize the SLER to configure the storage byte ordering.

Before executing instructions as cacheable:

- Invalidate the instruction cache.
- Initialize the ICCR to configure instruction cacheability.

Before using storage access instructions:

- Invalidate the data cache.
- Initialize CDBCR to determine if a store miss results in a line fill (WOA).
- Initialize the DCWR to select copy-back or write-through caching.
- Initialize the DCCR to configure data cacheability.

Before allowing interrupts (synchronous or asynchronous):

- Initialize the EVPR to point to vector table.
- Provide vector table with branches to interrupt handlers.
- Initialize MSR(ILE) bit for big endian or PowerPC little endian mode for interrupt handlers.

Before enabling asynchronous interrupts:

- Initialize timer facilities.
- Initialize MSR to enable appropriate interrupts.

Initialize other processor features, such as cache line locking, debug, and trace.

Initialize non-processor resources.

- Initialize system memory as required by the operating system or application code
- Initialize off-chip system facilities.
- Start the execution of operating system or application code.

4.4 Initialization Code Example

The following initialization code example illustrate the steps that should be taken to initialize the processor before an operating system or user programs begin execution. The example is presented in pseudo-code, function calls are named similarly to PPC401A1 mnemonics where appropriate. Specific implementations may require different ordering of these sections to ensure proper operation.

```

/* _____ */
/*   PPC401A1 Initialization Pseudo Code   */
/* _____ */
@0xFFFFFFFFC:          /* initial instruction fetch from 0xFFFFFFFFC */
    ba(init_code);      /* branch from initial address to initialization code*/

@init_code:

/* _____ */
/* Configure guarded attribute for performance and cacheability. */
/* _____ */

    mtspr(SGR, guarded_attribute);

/* _____ */
/* Configure endianness and compression. */
/* _____ */

    mtspr(SLER, endianness);

/* _____ */
/* Invalidate the instruction cache and enable cacheability */
/* _____ */

address = 0;          /* start at first line */
for (line = 0; line < n_lines; line++) /* I-cache has n_lines congruence classes */
{
    iccci(address);    /* invalidate congruence class */
    address += 16;      /* point to the next congruence class */
}
    mtspr(ICCR, i_cache_cacheability); /* enable I-cache*/
isync;

```

```

/* _____ */
/* Invalidate the data cache and enable cacheability */
/* _____ */

address = 0; /* start at first line */
for (line = 0; line < m_lines; line++) /* D-cache has m_lines congruence classes */
{
    dccci(address); /* invalidate congruence class */
    address += 16; /* point to the next congruence class */
}
mtspr(CDBCR, store-miss_line-fill);
mtspr(DCCR, copy-back_write-thru);
mtspr(DCCR, d_cache_cacheability); /* enable D-cache */
isync;

/* _____ */
/* Prepare system for synchronous interrupts */
/* _____ */

mtspr(EVPR, prefix_addr); /* initialize exception vector prefix */

/* Initialize vector table and interrupt handlers if not already done */

/* Initialize MSR(ILE) */

mtmsr(MSR(ILE));

/* _____ */
/* Prepare system for asynchronous interrupts */
/* _____ */

/* Initialize and configure timer facilities */

mtspr(PIT, 0); /* clear PIT so no PIT indication after TSR cleared */
mtspr(TSR, 0xFFFFFFFF); /* clear TSR */
mtspr(TCR, timer_enable); /* enable desired timers */
mtspr(TBLO, 0); /* reset time base low first to avoid ripple */
mtspr(TBHI, time_base_u); /* set time base, hi first to catch possible ripple */
mtspr(TBLO, time_base_l); /* set time base, low */
mtspr(PIT, pit_count); /* set desired PIT count */

/* Initialize the MSR */

```

```

/* Exceptions must be enabled immediately after timer facilities to avoid missing a */
/* timer exception. */
/*
/* The MSR also controls privileged/user mode and the wait state. */
/* These must be initialized by the operating system or application code. */
/* _____ */

```

```
mtmsr(machine_state);
```

```

/* _____ */
/* Initialization of other processor facilities should be performed at this time */
/* _____ */

/* _____ */
/* Initialization of non-processor facilities should be performed at this time */
/* _____ */

/* _____ */
/* Branch to operating system or application code can occur at this time */
/* _____ */

```

5

Exceptions, Interrupts, and Timers

PPC401A1 exceptions are generated by signals from external peripherals, instructions, the internal timer facility, debug events, and error conditions. Two external interrupt signals are provided in the PPC401A1, one critical and one non-critical. Both external interrupts are maskable.

This chapter begins by defining the terminology of exceptions and interrupts in Section 5.1, “Interrupts and Exceptions,” on p. 5-1.

Table 5-2, “Exception Vector Offsets,” on p. 5-6 lists the exceptions which are handled by the PPC401A1 in the order of exception vector offsets. Detailed descriptions of each exception follow, in the same order. Table 5-2 provides an index to the descriptions.

Several registers support exception handling and control. Section 5.3, “General Exception Handling Registers,” on p. 5-7, describes the general exception handling registers:

- Data Exception Address Register (DEAR)
- Exception Syndrome Register (ESR)
- Exception Vector Prefix Register (EVPR)
- Machine State Register (MSR)
- Save/Restore Registers (SRR0–SRR3)

Critical and non-critical external interrupt signals are enabled by the MSR.

Section 5.5, “Machine Check Exceptions,” on p. 5-13, describes machine checks.

Section 5.15, “Timer Facilities,” on p. 5-24, describes the timer facilities of the PPC401A1, including the time base and the registers Time Base Low Register (TBLO), Time Base Low User-mode (TBLU), Time Base High Register (TBHI), and Time Base High User-mode (TBHU). This section also describes the timer-related registers: the Timer Status Register (TSR) and the Timer Control Register (TCR), and the Programmable Interval Timer (PIT) register.

5.1 Interrupts and Exceptions

An *interrupt* is the action in which the processor saves its old context (MSR and instruction pointer) and begins execution at a pre-determined interrupt-handler address, with a modified MSR. *Exceptions* are events which, if enabled, cause the processor to take an interrupt.

5.1.1 Architectural Definitions and Behavior

Precise interrupts are those for which the instruction pointer saved by the interrupt must be either the address of the excepting instruction or the address of the next sequential instruction. *Imprecise* interrupts are those for which it is possible (not required, just possible) for the saved instruction pointer to be something else, possibly prohibiting guaranteed software recovery.

Note that “precise” and “imprecise” are defined assuming that the interrupts are unmasked (enabled to occur) when the associated exception occurs. Consider an exception that would cause a precise interrupt, were the interrupt enabled at the time of the exception, but that occurs while the interrupt is masked. Some exceptions of this type can cause the interrupt to occur later, immediately upon its enabling. In such a case, the interrupt is not considered precise with respect to the enabling instruction, but imprecise (“delayed precise”) with respect to the cause of the exception.

Asynchronous interrupts are caused by events which are independent of instruction execution. All asynchronous interrupts are precise, and the following rules apply:

1. All instructions prior to the one whose address is reported to the exception handling routine (in the save/restore register) have completed execution. However, some storage accesses generated by these preceding instructions may not have completed.
2. No subsequent instruction has begun execution, including the instruction whose address is reported to the exception handling routine.
3. The instruction having its address reported to the exception handler may appear not to have begun execution, or may have partially completed

Synchronous interrupts are caused directly by the execution (or attempted execution) of instructions. Synchronous interrupts can be either precise or imprecise.

For synchronous precise interrupts, the following rules apply:

1. The save/restore register addresses either the instruction causing the exception or the next sequential instruction. Which instruction is addressed is determined by the interrupt type and status bits.
2. All instructions preceding the instruction causing the exception have completed execution. However, some storage accesses generated by these preceding instructions may not have completed.
3. The instruction causing the exception may appear not to have begun execution (except for causing the exception), may have partially completed, or may have completed, depending on the interrupt type.
4. No subsequent instruction has begun execution.

The PPC401A1 does not implement any imprecise interrupts. Refer to The IBM PowerPC Embedded environment for an architectural description of imprecise interrupts.

Machine check interrupts are a special case typically caused by some kind of hardware or

storage subsystem failure, or by an attempt to access an invalid address. A machine check can be indirectly caused by the execution of an instruction, but not recognized or reported until long after the processor has executed past the instruction that caused the machine check. As such, machine check interrupts cannot properly be thought as synchronous, nor as precise or imprecise. However, in the PPC401A1, machine checks are handled precisely as critical interrupts (see Section 5.2, “Critical and Non-critical Exceptions,” on p. 5-5). For machine checks, the following general rules apply:

1. No instruction following the one whose address is reported to the machine check handler in the save/restore register has begun execution.
2. The instruction whose address is reported to the machine check handler in the save/restore register, and all previous instructions, may or may not have completed successfully. All previous instructions that would ever complete have completed, within the context existing before the machine check interrupt. No further interrupt (other than possible additional machine checks) can occur as a result of those instructions.

5.1.2 Behavior of the PPC401A1 Implementation

All exceptions are handled precisely. Precise handling implies that the address of the excepting instruction (for synchronous exceptions other than the system call exception), or the address of the next instruction to be executed (asynchronous exceptions and the system call exception), is passed to an exception handling routine. Precise handling also implies that all instructions that precede the instruction whose address is reported to the exception-handling routine have executed and that no subsequent instruction has begun execution. The specific instruction whose address is reported may not have begun execution or may have partially completed, as specified for each precise exception type.

Synchronous precise exceptions include most debug exceptions, program exceptions, data storage exceptions, system call, and alignment exceptions.

Asynchronous precise exceptions include the critical interrupt exception, external interrupts, internal peripherals, internal timer facility exceptions, and some debug events.

The machine check exceptions, which are neither synchronous or asynchronous, are handled precisely.

The synchronism of instruction-side machine checks (errors that occur while attempting to fetch an instruction from external memory) require further explanation. Fetch requests to cacheable memory that miss in the instruction cache (ICU) cause an instruction cache line fill (four words). If any words in the fetched line are associated with an error, an exception will occur upon attempted execution and the cache line will be invalidated.

It is improper to declare an exception when an erroneous word is passed to the fetcher; the address could be the result of an incorrect speculative access. It is quite likely that no attempt will be made to execute an instruction from the erroneous address. A machine check instruction-side machine check exception occurs only when execution is attempted. If the exception occurs, execution is suppressed, SRR2 contains the erroneous address, and the ESR indicates that instruction-side machine check occurred. Although such an exception is

clearly asynchronous to the erroneous memory access, it is handled synchronously with respect to the attempted execution from the erroneous address.

Except for machine checks, all PPC401A1 exceptions are handled precisely:

- The address of the excepting instruction (for synchronous exceptions, other than the system call exception) or the address of the next sequential instruction (for asynchronous exceptions and the system call exception) is passed to the exception-handling routine.
- All instructions that precede the instruction whose address is reported to the exception-handling routine have completed execution and that no subsequent instruction has begun execution. The specific instruction whose address is reported might not have begun execution or might have partially completed, as specified for each exception type.

5.1.3 Exception-handling Priorities

In the PPC401A1, only one exception is handled at a time. Multiple simultaneous exceptions are handled in the priority order shown in Table 5-1.

Table 5-1. Exception-handling Priorities

Priority	Exception Type	Critical or Non-critical	Causing Conditions
1	Machine check—data	Critical	External bus error during data-side access
2	Debug—IAC	Critical	IAC debug event while in internal debug mode
3	Machine check—instruction	Critical	Attempted execution of instruction for which an external bus error occurred during fetch
4	Debug—UDE, EXC	Critical	UDE or EXC debug event while in internal debug mode
5	Critical interrupt input	Critical	Active level on the critical interrupt input
6	Watchdog timer—first time-out	Critical	Posting of an enabled first time-out of the watchdog timer in the TSR
7	Program	Non-critical	Attempted execution of illegal instructions, TRAP instruction, or privileged instruction in problem state
	System call	Non-critical	Execution of the sc instruction
8	Data storage exception—cache line locking	Non-critical	MSR[PR] = 1, and: dcbt and CDBCR[DUXE] = 1; dcbz and CDBCR[DLXE] = 1; icbi and CDBCR[IUXE] = 1.

Table 5-1. Exception-handling Priorities (cont.)

9	Alignment	Non-critical	Misaligned data accesses in PowerPC Little Endian mode; dcbz to non-cacheable address or write-through storage; Any string or multiple instruction in PowerPC Little-Endian mode; Non-word-aligned dcread , lwarx , and stwcx as described in Table 5-9, "Alignment Exception Summary," on p. 5-17.
10	Debug—IC, BT, TIE, DAC	Critical	IC, BT, TIE, or DAC debug event while in internal debug mode
11	External interrupt input	Non-critical	Interrupts from the ExtInt pin
12	Fixed Interval Timer (FIT)	Non-critical	Posting of an enabled FIT interrupt in the TSR
13	Programmable Interval Timer (PIT)	Non-critical	Posting of an enabled PIT interrupt in the TSR

5.2 Critical and Non-critical Exceptions

The PPC401A1 processes exceptions as non-critical and critical. Six exceptions are defined as *non-critical*: program exception, system call exception, alignment exception, an active external interrupt input, fixed interval timer (FIT) exception, and PIT exception. Five exceptions are defined as *critical*: machine check exceptions (instruction- and data-side), debug exceptions (any of the three types), exceptions caused by an active critical interrupt input, and the first time-out from the watchdog timer.

When a *non-critical* exception is taken, Save/Restore Register 0 (SRR0) is written with the address of the excepting instruction (most synchronous exceptions) or the next sequential instruction to be processed (asynchronous exceptions and system call).

If the PPC401A1 was executing a multi-cycle instruction (load/store, multiply, divide, or cache operation), the instruction is terminated and its address is written in SRR0. When load instructions terminate, the addressing registers are not updated. This ensures that the instructions can be restarted; if the addressing registers were in the range of registers to be loaded, this would be an invalid form in any event. Some target registers of a load instruction may have been written by the time of the exception; when the instruction restarts, the registers will simply be written again. Similarly, some of the target memory of a store instruction may have been written, and will be written again when the instruction restarts.

Save/Restore Register 1 (SRR1) is written with the contents of the MSR; the MSR is then updated to reflect the new machine context. The new MSR contents take effect beginning with the first instruction of the exception handling routine.

Exception handling routine instructions are fetched at an address determined by the exception type. The address of the exception handling routine is formed by concatenating

the 16 high-order bits of the EVPR and the exception vector offset. (A user must initialize the EVPR contents at power-up using an **mtspr** instruction.)

Table 5-2 shows the exception vector offsets for the exception types. Note that there may be multiple sources of the same exception type; exceptions of the same type are mapped to the same exception vector, regardless of source. In such cases, the exception handling routine must examine status registers to determine the exact source of the exception.

At the end of the exception handling routine, execution of an **rfi** instruction forces the contents of SRR0 and SRR1 to be written to the program counter and the MSR, respectively. Execution then begins at the address in the program counter.

Critical exceptions are processed similarly. When a critical exception is taken, Save/Restore Register 2 (SRR2) and Save/Restore Register 3 (SRR3) hold the next sequential address to be processed when returning from the exception and the contents of the MSR, respectively. At the end of the critical exception handling routine, execution of an **rfci** instruction writes the contents of SRR2 and SRR3 into the program counter and the MSR, respectively.

Table 5-2. Exception Vector Offsets

Offset	Exception Type	Exception Class	Category	Page
0x0100	Critical interrupt	Asynchronous precise	Critical	5-13
0x0200	Machine check—data	—	Critical	5-15
	Machine check—instruction	—	Critical	5-14
0x0500	External interrupt	Asynchronous precise	Non-critical	5-16
0x0600	Alignment	Synchronous precise	Non-critical	5-17
0x0700	Program	Synchronous precise	Non-critical	5-18
0x0C00	System Call	Synchronous precise	Non-critical	5-20
0x1000	PIT	Asynchronous precise	Non-critical	5-20
0x1010	FIT	Asynchronous precise	Non-critical	5-21
0x1020	Watchdog timer	Asynchronous precise	Critical	5-22
0x2000	Debug exception—IC, BT, TIE, IA1, DR1, DW1	Synchronous precise	Critical	5-23
	Debug exception—UDE, EXC	Asynchronous precise	Critical	

5.3 General Exception Handling Registers

The general exception handling registers are the Machine State Register (MSR), SRR0–SRR3, the Exception Vector Prefix Register (EVPR), the Exception Syndrome Register (ESR), and the Data Exception Address Register (DEAR).

5.3.1 Machine State Register (MSR)

The MSR is a 32-bit register that holds the current context of the PPC401A1. When a non-critical interrupt is taken, the MSR contents are written to SRR1; when a critical interrupt is taken, the MSR contents are written to SRR3. When an **rfi** or **rfci** instruction executes, the contents of the MSR are read from SRR1 or SRR3, respectively.

The MSR contents can be read into general purpose registers (GPRs) using an **mfmsr** instruction. The contents of a GPR can be written to the MSR using an **mtmsr** instruction. The MSR[EE] bit may be set/cleared atomically using the **wrtee** or **wrteei** instructions.

Figure 5-1 shows the MSR bit definitions and describes the function of each bit.

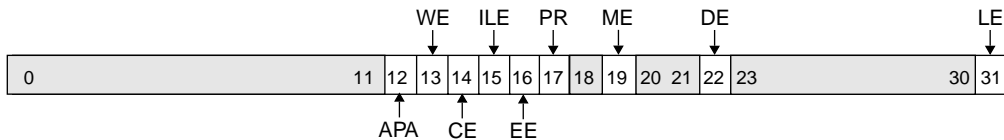


Figure 5-1. Machine State Register (MSR)

0:12		Reserved	
12	APA	Auxiliary Processor Available 0 Auxiliary processor not available. 1 Auxiliary processor available.	
13	WE	Wait State Enable 0 The processor is not in the wait state. 1 The processor is in the wait state.	If MSR[WE] = 1, the processor remains in the wait state until an exception is taken, a reset occurs, or an external debug tool clears WE.
14	CE	Critical Interrupt Enable 0 Critical interrupts are disabled. 1 Critical interrupts are enabled.	Controls the critical interrupt input and watchdog timer first time-out interrupts.
15	ILE	Interrupt Little Endian 0 Interrupt handlers execute in big endian mode. 1 Interrupt handlers execute in PowerPC little endian mode.	Copied to MSR(LE) when an interrupt is taken.
16	EE	External Interrupt Enable 0 Asynchronous exceptions are disabled. 1 Asynchronous exceptions are enabled.	Controls the non-critical external interrupt input, Programmable Interval Timer, and Fixed Interval Timer interrupts.

Figure 5-1. Machine State Register (MSR) (cont.)

17	PR	Problem State 0 Supervisor State (all instructions allowed) 1 Problem State (some instructions not allowed)
18		Reserved
19	ME	Machine Check Enable 0 Machine check exceptions are disabled 1 Machine check exceptions are enabled.
20:21		Reserved
22	DE	Debug Exception Enable 0 Debug exceptions are disabled. 1 Debug exceptions are enabled.
23:30		Reserved
31	LE	Little Endian 0 Processor executes in big endian mode. 1 Processor executes in PowerPC little endian mode.

5.3.2 Save/Restore Registers 0 and 1 (SRR0–SRR1)

SRR0 and SRR1 are 32-bit registers that hold the interrupted machine context when a non-critical interrupt is processed. On interrupt, SRR0 is set to the current or next instruction address and the contents of the MSR are written to SRR1. When an **rfi** instruction is executed at the end of the interrupt handler, the program counter and the MSR are restored from SRR0 and SRR1, respectively.

The contents of SRR0 and SRR1 can be written into GPRs using the **mfspr** instruction. The contents of GPRs can be written to SRR0 and SRR1 using the **mtspr** instruction.

Figure 5-2 shows the bit definitions for SRR0.

0	29	30	31
---	----	----	----

Figure 5-2. Save/Restore Register 0 (SRR0)

0:29		SRR0 receives an instruction address when a non-critical interrupt is taken; the Program Counter is restored from SRR0 when rfi executes.
------	--	--

Figure 5-2. Save/Restore Register 0 (SRR0)

30:31		Reserved
-------	--	----------

Figure 5-2 shows the bit definitions for SRR1.

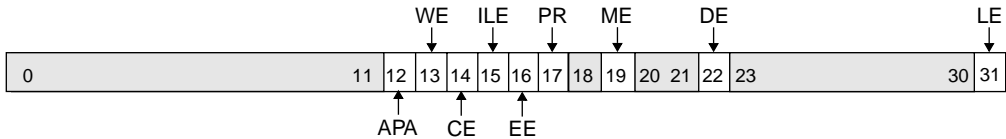


Figure 5-3. Save/Restore Register 1 (SRR1)

0:31	SRR1 receives a copy of the MSR when a critical interrupt is taken; the MSR is restored from SRR1 when rfi executes.
------	---

5.3.3 Save/Restore Registers 2 and 3 (SRR2– SRR3)

SRR2 and SRR3 are 32-bit registers that hold the interrupted machine context when a critical interrupt is processed. On interrupt, SRR2 is set to the current or next instruction address and the contents of the MSR are written to SRR3. When an **rfti** instruction is executed at the end of the interrupt handler, the program counter and the MSR are restored from SRR2 and SRR3, respectively.

The contents of SRR2 and SRR3 can be written to GPRs using the **mf spr** instruction. The contents of GPRs can be written to SRR2 and SRR3 using the **mt spr** instruction.

Figure 5-4 shows the SRR2 bit definitions.

0	29	30	31
---	----	----	----

Figure 5-4. Save/Restore Register 2 (SRR2)

0:29	SRR2 receives an instruction address when a critical interrupt is taken; the Program Counter is restored from SRR2 when rfti executes.
30:31	Reserved

Figure 5-5 shows the bit definitions for SRR3.

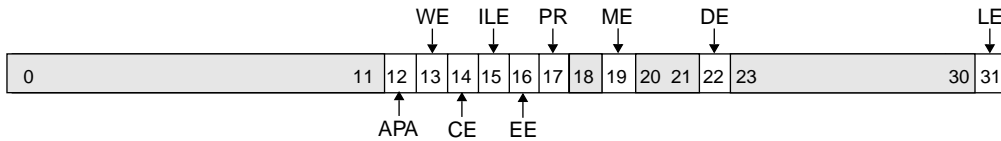


Figure 5-5. Save/Restore Register 1 (SRR1)

0:31	SRR3 receives a copy of the MSR when a critical interrupt is taken; the MSR is restored from SRR3 when rfci executes.
------	--

5.3.4 Exception Vector Prefix Register (EVPR)

The EVPR is a 32-bit register whose high-order 16 bits contain the prefix for the address of an exception processing routines. The 16-bit exception vector offsets (shown in Table 5-2, “Exception Vector Offsets,” on p. 5-6) are concatenated to the right of the high-order 16 bits of the EVPR to form the 32-bit address of the exception processing routine.

The contents of the EVPR can be written to a GPR using the **mfspr** instruction. The contents of a GPR can be written to EVPR using the **mtspr** instruction.

Figure 5-6 shows the EVPR bit definitions.

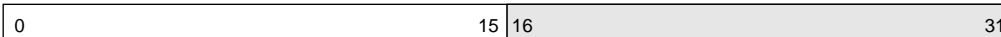


Figure 5-6. Exception Vector Prefix Register (EVPR)

0:15	Exception Vector Prefix
16:31	Reserved

5.3.5 Exception Syndrome Register (ESR)

The ESR is a 32-bit register whose bits help to specify the exact cause of various synchronous exceptions. These exceptions include instruction and data side machine checks, data storage exceptions, and program exceptions.

Section 5.5.1, “Instruction Machine Check Handling,” on p. 5-14, describes instruction machine checks. Section 5.6, “Data Storage Exceptions,” on p. 5-15, describes data storage exceptions. Section 5.9, “Program Exceptions,” on p. 5-18, describes program exceptions.

Although exception-handling routines are not required to reset the ESR, it is recommended that instruction machine check handlers reset the ESR; Section 5.5.1, “Instruction Machine Check Handling,” on p. 5-14 describes why such resets are recommended.

The contents of the ESR can be written to a GPR using the **mfspr** instruction. The contents of a GPR can be written to the ESR using the **mtspr** instruction

Figure 5-7 shows the ESR bit definitions

Figure 5-7. Exception Syndrome Register (ESR)

0	MCI	Machine check—instruction 0 Instruction machine check did not occur. 1 Instruction machine check occurred.
1:3		Reserved
4	PIL	Program exception—illegal 0 Illegal Instruction error did not occur. 1 Illegal Instruction error occurred.
5	PPR	Program exception—privileged 0 Privileged instruction error did not occur. 1 Privileged instruction error occurred.
6	PTR	Program exception—trap 0 Trap with successful compare did not occur. 1 Trap with successful compare occurred.
7:9		Reserved
10:11	DLK	Data Storage exception— lock fault 00 No lock exception 01 dcbf unlock exception 10 icbi unlock exception 11 dcbz lock-out exception
12	PAU	Program exception—auxiliary processor unavailable 0 Auxiliary processor unavailable exception did not occur. 1 Auxiliary processor unavailable exception occurred.

In general, ESR bits are set to indicate the kind of precise interrupt that occurred; other bits are cleared. The machine check—instruction (ESR[MCI]) bit behaves differently, however. Because instruction-side machine checks can occur without an interrupt being taken (if MSR[ME] = 0), this bit is set even when other ESR-setting exceptions (data storage and program) are occurring. Thus, data storage and program exceptions leave ESR[MCI] alone,

but clear the data storage and program exception bits that are not associated with the specific data storage or program exception that is occurring. Enabled instruction-side machine checks ($\text{MSR}[\text{ME}] = 1$) set $\text{ESR}[\text{MCI}]$ and clear the data storage and program exception bits.

If a machine check—instruction exception occurs but is disabled ($\text{MSR}[\text{ME}] = 0$), it sets $\text{ESR}[\text{MCI}]$ but leaves the data storage and program exception bits alone. If a machine check—instruction exception occurs while $\text{MSR}[\text{ME}] = 0$, *and* the instruction upon which the machine check—instruction exception is occurring also is some other kind of ESR-setting instruction (program or data storage exception), $\text{ESR}[\text{MCI}]$ is set to indicate that a machine check—instruction exception occurred; the other ESR bits will be set or cleared to indicate the other exception. These scenarios are summarized in Table 5-3.

Table 5-3. ESR Alteration by Various Exceptions

Scenario	ESR _{4:6}
Program exception without MCI	Set to type
Data storage exception without MCI	Cleared
Enabled MCI	Cleared
Disabled MCI, no others	Unchanged
Disabled MCI with program exception	Set to type
Disabled MCI with data storage exception	Cleared

5.3.6 Data Exception Address Register (DEAR)

The DEAR is a 32-bit register that contains the address of the access for which an alignment or data storage exception (cache line locking) error (both synchronous precise errors) occurred.

The contents of the DEAR can be written to a GPR using the **mfspr** instruction. The contents of a GPR can be written to the DEAR using the **mtspr** instruction.

Figure 5-8 shows the DEAR bit definitions.



Figure 5-8. Data Exception Address Register (DEAR)

0:31		Address of Data Error (synchronous)
------	--	-------------------------------------

5.4 Critical Interrupt Exception

An external source requests a critical interrupt by driving the critical interrupt input active, as controlled by the IOCR[CIL] bit. The critical exception is recognized if enabled by MSR[CE].

MSR[CE] also enables the watchdog timer first-time-out exception. However, the watchdog interrupt has a different exception vector than the critical pin interrupt. See Section 5.13, “Watchdog Timer Exception,” on p. 5-22.

After detecting a critical interrupt, if no synchronous precise exceptions are outstanding, the PPC401A1 immediately takes the critical interrupt exception and writes the address of the next instruction to be executed in SRR2. Simultaneously, the contents of the MSR are saved in SRR3. MSR[CE] is reset to 0 to prevent another critical interrupt or the watchdog timer first time-out exception from interrupting the critical interrupt exception handler before SRR2 and SRR3 get saved. MSR[DE] is reset to 0 to disable debug exceptions during the critical interrupt exception handler.

The MSR is also written with the values shown in Table 5-4. The high-order 16 bits of the program counter are then loaded with the contents of the EVPR and the low-order 16 bits of the program counter are loaded with 0x0100. Exception processing begins at the address in the program counter.

Inside the exception handling routine, after the contents of SRR2/SRR3 are saved, critical interrupts can be enabled again by setting MSR[CE] = 1.

Executing an **rfci** instruction restores the program counter from SRR2 and the MSR from SRR3, and execution resumes at the address in the program counter.

Table 5-4. Register Settings during Critical Interrupt Exceptions

SRR2	Written with the address of the next instruction to be executed
SRR3	Written with the contents of the MSR
MSR	WE, PR, CE, EE, DE \leftarrow 0 ME \leftarrow unchanged ILE \leftarrow unchanged LE \leftarrow ILE
PC	EVPR[0:15] 0x0100

5.5 Machine Check Exceptions

When an external bus error occurs on an instruction fetch, and execution of that instruction is subsequently attempted, a machine check—instruction exception occurs.

When an external bus error occurs while attempting data accesses, a machine check—data exception occurs.

When an instruction-side machine check interrupt occurs, the PPC401A1 stores the address of the excepting instruction in SRR2. When a data-side machine check occurs, the

PPC401A1 stores the address of the next sequential instruction in SRR2. Simultaneously, for all machine check exceptions, the contents of the MSR are loaded into SRR3.

The MSR Machine Check Enable bit (MSR[ME]) is reset to 0 to disable another machine check from interrupting the machine check exception handling routine. The other MSR bits are loaded with the values shown in Table 5-5 and Table 5-6. The high-order 16 bits of the program counter are then written with the contents of the EVPR and the low-order 16 bits of the program counter are written with 0x0200. Exception processing begins at the new address in the program counter.

Executing an **rfci** instruction restores the program counter from SRR2 and the MSR from SRR3, and execution resumes at the address in the program counter.

5.5.1 Instruction Machine Check Handling

When a machine check occurs on an instruction fetch, *and execution of that instruction is subsequently attempted*, a machine check—instruction exception occurs. If enabled by MSR[ME], the processor reports the machine check—instruction exception by vectoring to the machine check handler (EVPR[0:15] || 0x0200), setting ESR[MCI]. Note that only a bus error can cause a machine check—instruction exception. Taking the vector automatically clears MSR[ME] and the other MSR fields.

Note that it is improper to declare a machine check—instruction exception when the instruction is fetched, because the address is possibly the result of an incorrect speculation by the fetcher. It is quite likely that no attempt will be made to execute an instruction from the erroneous address. The exception will occur only if execution of the instruction is subsequently attempted.

When a machine check occurs on an instruction fetch, the erroneous instruction is never written to the instruction cache unit (ICU). Fetch requests to cacheable memory that miss in the ICU cause an instruction cache line fill (four words). If any words in the fetched line are associated with an error, an exception occurs upon attempted execution and the cache line is invalidated. If any word in the line is in error, the cache line is invalidated after the line fill.

ESR[MCI] is set, even if MSR[ME] = 0. This means that if a machine check—instruction exception occurs while running in code in which MSR[ME] is disabled, the machine check—instruction exception is recorded in the ESR, but no interrupt occurs. Software running with MSR[ME] disabled can sample ESR[MCI] to determine whether at least one machine check—instruction exception occurred during the disabled execution.

After MSR[ME] is enabled again, if a new machine check—instruction exception occurs, it will be recorded in ESR[MCI] and the machine check—instruction exception handler will be invoked. However, enabling MSR[ME] again does *not* cause a Machine Check interrupt to occur simply due to the presence of ESR[MCI] indicating that a machine check—instruction exception occurred while MSR[ME] was disabled. The machine check—instruction exception must occur while MSR[ME] is enabled for the machine check interrupt to be taken.

Software should, in general, clear the ESR bits before returning from a machine check interrupt, to avoid any ambiguity when handling subsequent machine check exceptions.

Table 5-5. Register Settings during Machine Check—Instruction Exceptions

SRR2	Written with the address that caused the machine check.
SRR3	Written with the contents of the MSR
MSR	WE, PR, CE, EE, ME, DE \leftarrow 0 ILE \leftarrow unchanged LE \leftarrow ILE
PC	EVPR[0:15] 0x0200
ESR	MCI \leftarrow 1

5.5.2 Data Machine Check Handling

When a machine check occurs on an data access, a machine check—data exception occurs. The handling of machine check—data exceptions is implementation-specific.

5.6 Data Storage Exceptions

Table 5-6. Register Settings during Machine Check—Data Exceptions

SRR2	Written with the address of the next sequential instruction.
SRR3	Written with the contents of the MSR
MSR	WE, PR, CE, EE, ME, DE \leftarrow 0 ILE \leftarrow unchanged LE \leftarrow ILE
PC	EVPR[0:15] 0x0200
ESR	MCI \leftarrow 0

The data storage exception occurs on a cache line locking error. The cache line locking errors occur when certain cache control instructions attempt to access a cache line, *in user mode*, when the lock exception bits are enabled in the Cache Debug Control Register (CDBCR). The data storage exception occurs regardless of whether the cache line is locked. Section 6.5, “Cache Line Locking,” on p. 6-12, describes cache line locking, including resulting exceptions.

The following cache line locking errors occur in user mode:

- If **dcbf** attempts to access a cache line while the DCU unlock exception is enabled (CDBCR[DUXE] = 1).
- If **dcbz** references a cacheable address while the DCU lockout exception is enabled (CDBCR[DLXE] = 1). An alignment error also occurs; however, in this case, the data storage exception takes priority.
- If **dcbz** references a non-cacheable address while the DCU unlock exception is enabled (CDBCR[DUXE] = 1). An alignment error, which takes priority, also occurs.
- If **icbi** attempts to invalidate a locked cache line, while the ICU unlock exception is enabled (CDBCR[IUXE] = 1).

When a data storage exception occurs, the PPC401A1 suppresses execution of the instruction causing the exception and writes the EA of the instruction address in SRR0. The current contents of the MSR are saved into SRR1. The DEAR is written with the EA of the failed access. ESR[DLK] is set to indicate the cause of the exception.

The high-order 16 bits of the program counter are then written with the contents of the EVPR and the low-order 16 bits of the program counter are written with 0x0300. Exception processing begins at the new address in the program counter.

Table 5-7. Register Settings during Data Storage Exceptions

SRR0	Written with the EA of the instruction causing the data storage exception
SRR1	Written with the value of the MSR at the time of the exception
MSR	WE, PR, EE ← 0 CE, ME, DE ← unchanged ILE ← unchanged LE ← ILE
PC	EVPR[0:15] 0x0300
DEAR	Written with the EA of the failed access
ESR	The DLK and DST fields are set to indicate the cause of the exception. See Figure 5-7 for details of the DLK field.

5.7 External Interrupt Exception

External interrupt exceptions are triggered by active levels on the external interrupt inputs. All external interrupting events are presented to the processor as a single external interrupt. External interrupts are enabled or disabled by the MSR[EE] bit.

Programming Note: The MSR[EE] bit also enables the occurrence of PIT and FIT interrupts. However, after timer interrupts, control passes to different exception vectors than for the interrupts discussed in the preceding paragraph. Therefore, these timer exceptions are described in Section

5.11, “Programmable Interval Timer (PIT) Exception,” on p. 5-20 and Section 5.12, “Fixed Interval Timer (FIT) Exception,” on p. 5-21.

5.7.1 External Interrupt Exception Handling

When $\text{MSR}[\text{EE}] = 1$ (external interrupts are enabled), and a non-critical external interrupt exception occurs, and this exception is the highest priority exception condition, the processor immediately writes the address of the next sequential instruction into SRR0. Simultaneously, the contents of the MSR are saved in SRR1.

When the processor takes a non-critical external interrupt, $\text{MSR}[\text{EE}]$ is reset to 0. This disables other external interrupts from interrupting the exception handler before SRR0 and SRR1 are saved. The MSR is also written with the other values shown in Table 5-8. The high-order 16 bits of the program counter are written with the contents of the EVPR and the low-order 16 bits of the program counter are written with 0x0500. Exception processing begins at the address in the program counter.

Executing an **rfi** instruction restores the program counter from SRR0 and the MSR from SRR1, and execution resumes at the address in the program counter.

Table 5-8. Register Settings during External Interrupt Exceptions

SRR0	Written with the address of the next sequential instruction
SRR1	Written with the contents of the MSR
MSR	WE, PR, EE \leftarrow 0 CE, ME, DE \leftarrow unchanged ILE \leftarrow unchanged LE \leftarrow ILE
PC	EVPR[0:15] 0x0500

5.8 Alignment Exception

Alignment exceptions are caused by misaligned data accesses by storage reference instructions, a **dcbz** instruction to non-cacheable or write through storage, or load/store multiple and string instructions when $\text{MSR}[\text{LE}] = 1$ (in PowerPC Little Endian mode). Table 5-9 summarizes the instructions and conditions causing alignment exceptions.

Table 5-9. Alignment Exception Summary

PPC401A1 MSR	Instructions Causing Alignment Exceptions	Conditions
MSR[LE] = 0	dcbz	EA in non-cacheable or write-through storage
	dcread, lwarx, stwcx.	EA not word-aligned

Table 5-9. Alignment Exception Summary

PPC401A1 MSR	Instructions Causing Alignment Exceptions	Conditions
MSR[LE] = 1	dcbz	EA in non-cacheable or write-through storage
	lha, lhau, lhaux, lhax, lhbrx, lhz, lhzu, lhzux, lhzx, sth, sthbrx, sthu, sthux, sthx	EA not halfword-aligned
	dcread, lwarx, lwbrx, lwz, lwzu, lwzux, lwzx, stw, stwbrx, stwcx., stwu, stwux, stwx	EA not word-aligned
	lmw, lswi, lswx, stmw, stswi, stswx	Always

Execution of an instruction causing an alignment exception is prohibited from completing. SRR0 is written with the address of that instruction and the current contents of the MSR are saved into SRR1. The DEAR is written with the address that caused the alignment error. The MSR bits are written with the values shown in Table 5-10. The high-order 16 bits of the program counter are written with the contents of the EVPR and the low-order 16 bits of the program counter are written with 0x0600. Exception processing begins at the new address in the program counter.

Executing an **rfi** instruction restores the program counter from SRR0 and the MSR from SRR1, and execution resumes at the address in the program counter

Programming Note: Alignment exceptions cannot be disabled. To avoid overwrites of SRR0 and SRR1 by alignment exceptions that occur within a handler, exception handlers should save these registers as soon as possible.

Table 5-10. Register Settings during Alignment Error Exceptions

SRR0	Written with the address of the instruction causing the alignment exception
SRR1	Written with the contents of the MSR
MSR	WE, PR, EE ← 0 CE, ME, DE ← unchanged ILE ← unchanged LE ← ILE
PC	EVPR[0:15] 0x0600
DEAR	Written with the address that caused the alignment violation

5.9 Program Exceptions

Program exceptions are caused by attempting to execute an illegal operation, by executing a trap instruction with conditions satisfied, or by attempting to execute a privileged instruction while in the problem state.

The ESR bits that differentiate these situations (see Table 5-11) are mutually exclusive: when a program exception occurs, the appropriate bit is set and the others are cleared. These exceptions are not maskable.

Table 5-11. ESR Usage for Program Exceptions

Bit	Exception Cause
ESR[PIL]	Illegal
ESR[PPR]	Privileged
ESR[PTR]	Trap

The program exception interrupt handler does not need to reset the ESR.

When execution of an illegal instruction is attempted, or when execution of a privileged instruction is attempted in problem state, the PPC401A1 does not execute the instruction, and it writes the address of the excepting instruction into SRR0.

Trap instructions can be used as a program exception or a debug event, or both (see Section 7.5, “Debug Events,” on p. 7-4, for information about debug events). When a trap instruction is detected as a program exception, the PPC401A1 writes the address of the trap instruction into SRR0. See **tw** on p. 9-167 and **twi** on p. 9-171 for a detailed discussion of the behavior of trap instructions with various exceptions enabled.

After any program exception, the contents of the MSR are written into SRR1 and the MSR bits are written with the values shown in Table 5-12. The high-order 16 bits of the program counter are written with the contents of the EVPR; the low-order 16 bits of the program counter are written with 0x0700. Exception processing begins at the new address in the program counter.

Executing an **rfi** instruction restores the program counter from SRR0 and the MSR from SRR1, and execution resumes at the address in the program counter.

Table 5-12. Register Settings during Program Exceptions

SRR0	Written with the address of the excepting instruction
SRR1	Written with the contents of the MSR
MSR	WE, PR, EE \leftarrow 0 CE, ME, DE \leftarrow unchanged ILE \leftarrow unchanged LE \leftarrow ILE
PC	EVPR[0:15] 0x0700
ESR	Written with the type of program exception. (See Table 5-3 and Table 5-11)

5.10 System Call Exception

System call exceptions occur when a **sc** instruction is executed. The PPC401A1 writes the address of the instruction following the **sc** into SRR0. The contents of the MSR are written into SRR1 and the MSR bits are written with the values shown in Table 5-13. The high-order 16 bits of the program counter are then written with the contents of the EVPR and the low-order 16 bits of the program counter are written with 0x0C00. Exception processing begins at the new address in the program counter.

Executing an **rfi** instruction restores the program counter from SRR0 and the MSR from SRR1, and execution resumes at the address in the program counter.

Table 5-13. Register Settings during System Call Exceptions

SRR0	Written with the address of the instruction following the sc instruction
SRR1	Written with the contents of the MSR
MSR	WE, PR, EE \leftarrow 0 CE, ME, DE \leftarrow unchanged ILE \leftarrow unchanged LE \leftarrow ILE
PC	EVPR[0:15] 0x0C00

5.11 Programmable Interval Timer (PIT) Exception

For a discussion of the PPC401A1 timer facilities, see Section 5.15, “Timer Facilities,” on p. 5-24. The PIT is described in Section 5.15.2, “Programmable Interval Timer (PIT),” on p. 5-28.

If the PIT exception is enabled by TCR[PIE] and MSR[EE], the PPC401A1 initiates a PIT interrupt after detecting a time-out from the PIT. Time-out is detected when, at the beginning of a clock cycle, TSR[PIS] = 1. (This occurs on the cycle after the PIT decrements on a PIT count of 1.) The PPC401A1 immediately takes the interrupt. The address of the next sequential instruction is saved in SRR0; simultaneously, the contents of the MSR are written into SRR1 and the MSR is written with the values shown in Table 5-14. The high-order 16 bits of the program counter are then written with the contents of the EVPR and the low-order 16 bits of the program counter are written with 0x1000. Exception processing begins at the address in the program counter.

To clear a PIT interrupt, the exception handling routine must clear the PIT interrupt bit, TSR[PIS]. Clearing is performed by writing a word to TSR, using an **mtspr** instruction, that has 1 in bit positions to be cleared and 0 in all other bit positions. The data written to the TSR is not direct data, but a mask; a 1 clears the bit and 0 has no effect.

Executing an **rfi** instruction restores the program counter from SRR0 and the MSR from SRR1, and execution resumes at the address in the program counter.

Table 5-14. Register Settings during Programmable Interval Timer Exceptions

SRR0	Written with the address of the next instruction to be executed
SRR1	Written with the contents of the MSR
MSR	WE, PR, EE \leftarrow 0 CE, ME, DE \leftarrow unchanged ILE \leftarrow unchanged LE \leftarrow ILE
PC	EVPR[0:15] 0x1000
TSR	PIS \leftarrow 1

5.12 Fixed Interval Timer (FIT) Exception

For a discussion of the PPC401A1 timer facilities, see Section 5.15, “Timer Facilities,” on p. 5-24. The FIT is described in Section 5.15.3, “Fixed Interval Timer (FIT),” on p. 5-29.

If the FIT exception is enabled by TCR[FIE] and MSR[EE], the PPC401A1 initiates a FIT interrupt after detecting a time-out from the FIT. Time-out is detected when, at the beginning of a clock cycle, TSR[FIS] = 1. (This occurs on the second cycle after the 0 \rightarrow 1 transition of the appropriate time-base bit.) The PPC401A1 immediately takes the interrupt. The address of the next sequential instruction is written into SRR0; simultaneously, the contents of the MSR are written into SRR1 and the MSR is written with the values shown in Table 5-15. The high-order 16 bits of the program counter are then written with the contents of the EVPR and the low-order 16 bits of the program counter are written with 0x1010. Exception processing begins at the address in the program counter.

To clear a FIT interrupt, the exception handling routine must clear the FIT interrupt bit, TSR[FIS]. Clearing is performed by writing a word to TSR, using an **mtspr** instruction, that has 1 in any bit positions to be cleared and 0 in all other bit positions. The data written to the TSR is not direct data, but a mask; a 1 clears a bit and 0 has no effect.

Executing an **rfi** instruction restores the program counter from SRR0 and the MSR from SRR1, and execution resumes at the address in the program counter.

Table 5-15. Register Settings during Fixed Interval Timer Exceptions

SRR0	Written with the address of the next sequential instruction
SRR1	Written with the contents of the MSR
MSR	WE, PR, EE \leftarrow 0 CE, ME, DE \leftarrow unchanged ILE \leftarrow unchanged LE \leftarrow ILE

Table 5-15. Register Settings during Fixed Interval Timer Exceptions (cont.)

PC	EVPR[0:15] 0x1010
TSR	FIS \leftarrow 1

5.13 Watchdog Timer Exception

For a general description of the PPC401A1 timer facilities, see Section 5.15, “Timer Facilities,” on p. 5-24. The watchdog timer (WDT) is described in Section 5.15.4, “Watchdog Timer,” on p. 5-30.

If the WDT exception is enabled by TCR[WIE] and MSR[CE], the PPC401A1 initiates a WDT interrupt after detecting the first WDT time-out. First time-out is detected when, at the beginning of a clock cycle, TSR[WIS] = 1. (This occurs on the second cycle after the 0 \rightarrow 1 transition of the appropriate time-base bit while TSR[ENW] = 1 and TSR[WIS] = 0.) The PPC401A1 immediately takes the interrupt. The address of the next sequential instruction is saved in SRR2; simultaneously, the contents of the MSR are written into SRR3 and the MSR is written with the values shown in Table 5-16. The high-order 16 bits of the program counter are then written with the contents of the EVPR and the low-order 16 bits of the program counter are written with 0x1020. Exception processing begins at the address in the program counter.

To clear the WDT interrupt, the exception handling routine must clear the WDT interrupt bit TSR[WIS]. Clearing is done by writing a word to TSR (using **mtspr**), with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The data written to the status register is not direct data, but a mask; a 1 causes the bit to be cleared, and a 0 has no effect.

Executing the return from critical interrupt instruction (**rfci**) restores the contents of the program counter and the MSR from SRR2 and SRR3, respectively, and the PPC401A1 resumes execution at the contents of the program counter.

Table 5-16. Register Settings during Watchdog Timer Exceptions

SRR2	Written with the address of the next sequential instruction
SRR3	Written with the contents of the MSR
MSR	WE, PR, CE, EE, DE \leftarrow 0 ME \leftarrow unchanged ILE \leftarrow unchanged LE \leftarrow ILE
PC	EVPR[0:15] 0x1020
TSR	WIS \leftarrow 1

5.14 Debug Exception Handling

Debug exceptions can be either *synchronous* or *asynchronous*. The following debug events generate synchronous exceptions: instruction address compare (also referred to as IAC), data address compare (also referred to as DAC), trap with condition satisfied (TIE), branch taken (BT), and instruction completion (IC). The following debug events generate asynchronous exceptions: unconditional debug event (UDE) and exceptions (EXC). See Section 7.5, “Debug Events,” on p. 7-4, for more information about debug events.

For debug events, SRR2 is written with an address, which varies with the type of debug event, as shown in Table 5-17:

Table 5-17. SRR2 during Debug Exceptions

Debug Event	Address Saved in SRR2
IAC DAC TDE BT	Address of the instruction that caused the event
IC	Address of the instruction <i>following</i> the instruction that caused the event
UDE	Address of next instruction to be executed at time of UDE
EDE	Interrupt vector address of the initial exception that caused the exception debug event

SRR3 is written with the contents of the MSR and the MSR is written with the values shown in Table 5-18. The high-order 16 bits of the program counter are then written with the contents of the EVPR; the low-order 16 bits of the program counter are written with 0x2000. Exception processing begins at the address in the program counter.

Executing an **rfci** instruction restores the program counter from SRR2 and the MSR from SRR3, and execution resumes at the address in the program counter.

Table 5-18. Register Settings during Debug Exceptions

SRR2	Written with an address as described in Table 5-17
SRR3	Written with the contents of the MSR
MSR	WE, PR, CE, EE, DE \leftarrow 0 ME \leftarrow unchanged ILE \leftarrow unchanged LE \leftarrow ILE
PC	EVPR[0:15] 0x2000
DBSR	Set to indicate type of debug event.

5.15 Timer Facilities

The PPC401A1 provides four timer facilities: a time base, a programmable interval timer (PIT), a Fixed Interval Timer (FIT), and a watchdog timer (WDT). These facilities, which share the same base clock frequency, can support:

- Time-of-day functions
- Data logging functions
- Peripherals requiring periodic schedule service
- General system maintenance

Additionally, the timer facilities can help a system to recover from faulty hardware or software.

Figure 5-9 shows the relationship of these facilities and the base clock.

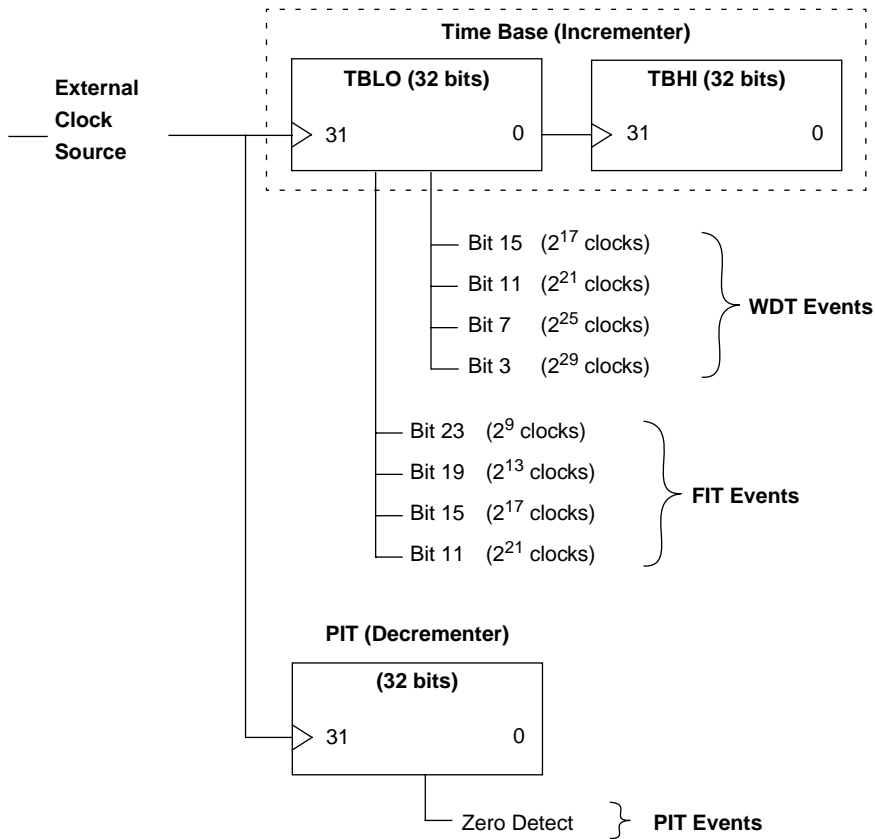


Figure 5-9. Relationship of Timer Facilities to the Base Clock

5.15.1 Time Base

The PPC401A1 implements a 64-bit time base. The time base, which increments once during each period of the time base clock, provides a time reference. The time base is accessed using the 32-bit registers TBLO and TBHI. Software access to the time base is through the **mf spr** and **mt spr** instructions.

Access to the time base registers TBHI and TBLO is privileged.

User-mode read-only access to the Time Base is provided by reading from different SPR numbers. Specifically, read-only access to TBHI is accomplished by reading TBHU, and read-only access to TBLO is accomplished by reading TBLU. Both TBHU and TBLU are read using **mf spr** instructions. An **mt spr** to these registers is boundedly undefined.

The period of the 64-bit Time Base is approximately 7794 years for a 75 MHz time base clock. The time base does not generate interrupts, even when it wraps. For most applications, the time base is set at system reset and only read thereafter. Note that the FIT and the watchdog timer (discussed below) are driven by 0→1 transitions of selected bits of TBLO. Transitions caused by software alteration of TBLO, using **mt spr**, have the same effect as transitions caused by normal incrementing of the time base.

Figure 5-10 illustrates TBHI(TBHU); Figure 5-10 illustrates TBLO(TBLU).



Figure 5-10. Time Base Register (TBHI, TBHU)

0:31		Time High	Current count, high-order. Note: TBHU provides read-only access to the time base register TBHI. The contents of TBHU and TBHI never differ).
------	--	-----------	--

0	31
---	----

Figure 5-11. Time Base Register (TBLO, TBLU)

0:31		Time Low	Current count, low-order. Note: TBLU provides read-only access to the time base register TBLO. The contents of TBLU and TBLO never differ).
------	--	----------	---

5.15.1.1 Comparison with PowerPC Architecture Time Base

The time base of the PPC401A1 is functionally similar to the time base defined in the PowerPC Architecture, but the two are not the same. Descriptions of the differences, as an aid to porting code, follow.

For the PowerPC Architecture:

- The PowerPC Architecture defines a 64-bit time base that can be accessed as a pair of 32-bit registers. Different register numbers are used for read access (user or privileged mode) and write access (privileged mode).
- The PowerPC Architecture provides a **mftb** (move from time base) instruction for user-mode read access to the time base. The register numbers (0x10C and 0x10D) used to specify the time base registers are not SPR numbers. However, the opcode of **mftb** differs from the opcode of **mfsprr** by only one bit. The PowerPC Architecture allows an implementation to ignore this bit and handle **mftb** as **mfsprr**. Therefore, the time base register numbers cannot specify other SPRs. Further, PowerPC compilers can use the **mftb** opcode only with the register numbers specified in the PowerPC Architecture as read-access time base registers (0x10C and 0x10D).
- The PowerPC Architecture does not provide privileged-mode-only read access to the time base (by definition, the user-mode read access mechanism is also available in privileged mode).
- The PowerPC Architecture provides privileged-mode write access to the time base using **mtsprr** with SPR numbers 0x11C and 0x11D.

For the PPC401A1:

- The PPC401A1 defines a 64-bit time base that can be accessed as a pair of 32-bit registers. Different register numbers are used for read access (user or privileged mode) and write access (privileged mode). The PPC401A1 also allows read access (in privileged mode only) using the same register number as a write access.
- The PPC401A1 provides user-mode and privileged-mode read access to the time base using **mfsprr** with SPR numbers 0x3CD and 0x3CC. The PPC401A1 provides privileged-mode-only read access via **mfsprr** instructions with SPR numbers 0x3DD and 0x3DC. The PPC401A1 does not implement **mttb**; an attempt to use it results in a program exception caused by the “illegal” opcode.
- The PPC401A1 provides privileged-mode write access to the time base using **mtspr** instructions with SPR numbers 0x3DD and 0x3DC.

Figure 5-12 summarizes the differences between the time bases.:

Figure 5-12. Time Base Comparison

PowerPC Architecture				PPC401A1		
Access Instructions		Register Number	Access Restrictions	Access Instructions	Register Number	Access Restrictions
Upper 32 bits	mttbu RT <i>Extended mnemonic for mttb RT,TBU</i>	0x10D	Read-Only	mttbhu RT <i>Extended mnemonic for mfspr RT,TBHU</i>	0x3CC	Read-Only
	mttbu RS <i>Extended mnemonic for mtspr TBU,RS</i>	0x11D	Privileged; Write-Only	mttbhi RT <i>Extended mnemonic for mfspr RT,TBHI</i> mttbhi RS <i>Extended mnemonic for mtspr TBHI,RS</i>	0x3DC	Privileged; Read Privileged; Write

Figure 5-12. Time Base Comparison (cont.)

PowerPC Architecture				PPC401A1		
Access Instructions		Register Number	Access Restrictions	Access Instructions	Register Number	Access Restrictions
Lower 32 bits	mftb RT <i>Extended mnemonic for mftb RT,TBL</i>	0x10C	Read-Only	mftblu RT <i>Extended mnemonic for mfspr RT,TBLU</i>	0x3CD	Read-Only
	mttbl <i>Extended mnemonic for mtspr TBL,RS</i>	0x11C	Privileged; Write-Only	mftblo RT <i>Extended mnemonic for mfspr RT,TBLO</i> mttblo RS <i>Extended mnemonic for mtspr TBLO,RS</i>	0x3DD	Privileged; Read Privileged; Write

5.15.2 Programmable Interval Timer (PIT)

The PIT is a 32-bit register that decrements at the same rate as the time base. The PIT is read or written using **mfspr** or **mtspr**. Writing to the PIT, using **mtspr**, simultaneously writes to a hidden reload register. Reading the PIT using **mfspr** returns the current PIT contents; the hidden reload register cannot be read. When a non-zero value is written to the PIT, it begins to decrement. A PIT event occurs when a decrement occurs on a PIT count of 1. When a PIT event occurs, the following occur:

1. If the PIT is in auto-reload mode ($\text{TCR}[\text{ARE}] = 1$), the PIT is loaded with the last value an **mtspr** wrote to the PIT. A decrement from a PIT count of 1 immediately causes a reload; no intermediate PIT content of 0 occurs.

If the PIT is not in auto-reload mode ($\text{TCR}[\text{ARE}] = 0$), a decrement from a PIT count of simply causes a PIT content of 0.

2. $\text{TSR}[\text{PIS}]$ is set to 1.
3. If enabled ($\text{TCR}[\text{PIE}] = 1$ and $\text{MSR}[\text{EE}] = 1$), a PIT interrupt is taken. See Section 5.11, “Programmable Interval Timer (PIT) Exception,” on p. 5-20, for details of register behavior during a PIT interrupt.

The interrupt handler should use software to reset the $\text{TSR}[\text{PIS}]$ bit. This is done by using **mtspr** to write a word to the TSR having a 1 in $\text{TSR}[\text{PIS}]$ and any other bits to be cleared, and a 0 in all other bits. The data written to the TSR is not direct data, but a mask. A 1 clears a bit; a 0 has no effect.

Using **mtspr** to force the PIT to 0 *does not* cause a PIT interrupt. However, decrementing that was ongoing at the instant of the **mtspr** instruction can cause the appearance of an

interrupt. To eliminate the PIT as a source of interrupts, write a 0 to TCR[PIE], the PIT interrupt enable bit.

To eliminate all PIT activity:

- 1. Write a 0 to TCR[PIE]. This prevents PIT activity from causing interrupts.
- 2. Write a 0 to TCR[ARE]. This disables the PIT auto-reload feature.
- 3. Write zeroes to the PIT to halt PIT decrementing. Although this action does not cause a pit PIT interrupt to become pending, a near-simultaneous decrement might have done so.
- 4. Write a 1 to TSR[PIS] (PIT Interrupt Status bit). This clears TSR[PIS] to 0 (see Section 5.15.5, “Timer Status Register (TSR),” on p. 5-32). This also clears any pending PIT interrupt. Because the PIT freezes, no further PIT events are possible.

If the auto-reload feature is disabled (TCR[ARE] = 0) after the PIT decrements to 0, the PIT remains 0 until software uses **mtspr** to reload it.

On all resets, TCR[ARE] = 0, which disables the auto-reload feature.

Figure 5-13 illustrates the PIT.



Figure 5-13. Programmable Interval Timer (PIT)

0:31		Programmed Interval Remaining	The number of clocks remaining until the PIT event
------	--	-------------------------------	--

5.15.3 Fixed Interval Timer (FIT)

The FIT provides timer interrupts having a repeatable period, facilitating system maintenance. The FIT is functionally similar to an auto-reload PIT, except that fewer selections of interrupt periods are available.

The FIT exception occurs on 0→1 transitions of selected bits from the time base, as shown in the following table:

Table 5-19. FIT Controls

TCR[FP]	TBLO Bit	Period (Time Base Clocks)	Period (75 Mhz Clock)
0, 0	23	2 ⁹ clocks	11.64 μsec
0, 1	19	2 ¹³ clocks	186.15 μsec

Table 5-19. FIT Controls

TCR[FP]	TBLO Bit	Period (Time Base Clocks)	Period (75 Mhz Clock)
1, 0	15	2^{17} clocks	2.979 msec
1, 1	11	2^{21} clocks	47.66 msec

The TSR[FIS] bit logs a FIT exception as a pending interrupt. A FIT interrupt occurs if TCR[FIE] and MSR[EE] are enabled at the time of the FIT exception. Section 5.12, “Fixed Interval Timer (FIT) Exception,” on p. 5-21, describes register behavior during a FIT interrupt.

The interrupt handler must use software to reset the TSR[FIS] bit. This is done by using **mtspr** to write a word to the TSR having a 1 in TSR[FIS] and any other bits to be cleared, and a 0 in all other bits. The data written to the TSR is not direct data, but a mask. A 1 clears a bit and a 0 has no effect.

5.15.4 Watchdog Timer

The watchdog timer (WDT) aids system recovery from software or hardware faults.

A WDT timeout occurs on 0→1 transitions of selected bits from the time base, as shown in the following table:

Table 5-20. Watchdog Timer Controls

TCR[WP]	TBLO Bit	Period (Time Base Clocks)	Period (33 MHz Clock)
0,0	15	2^{17} clocks	2.979 msec
0,1	11	2^{21} clocks	47.66 msec
1,0	7	2^{25} clocks	0.762 sec
1,1	3	2^{29} clocks	12.202 sec

If a WDT timeout occurs while TSR[WIS] = 0 and TSR[ENW] = 1, a WDT interrupt occurs if the interrupt is enabled by TCR[WIE] and MSR[CE]. Section 5.13 describes register behavior during a WDT interrupt.

The interrupt handler must use software to reset the TSR[WIS] bit. This is done by using **mtspr** to write a word to the TSR having a 1 in TSR[WIS] and any other bits to be cleared, and a 0 in all other bits. The data written to the TSR is not direct data, but a mask. A 1 clears a bit and a 0 has no effect.

If a WDT timeout occurs while TSR[WIS] = 1 and TSR[ENW] = 1, a hardware reset occurs if enabled by a non-zero value of TCR[WRC]. The assumption is that TSR[WIS] was not cleared because the processor could not execute the watchdog handler, leaving reset as the

only way to restart the system. Note that after TCR[WRC] is set to a non-zero value, it cannot be reset by software. This prevents errant software from disabling the WDT reset capability.

Figure 5-14 describes the watchdog state machine. In the figure, numbers in parentheses refer to descriptions of operating modes that follow the table.

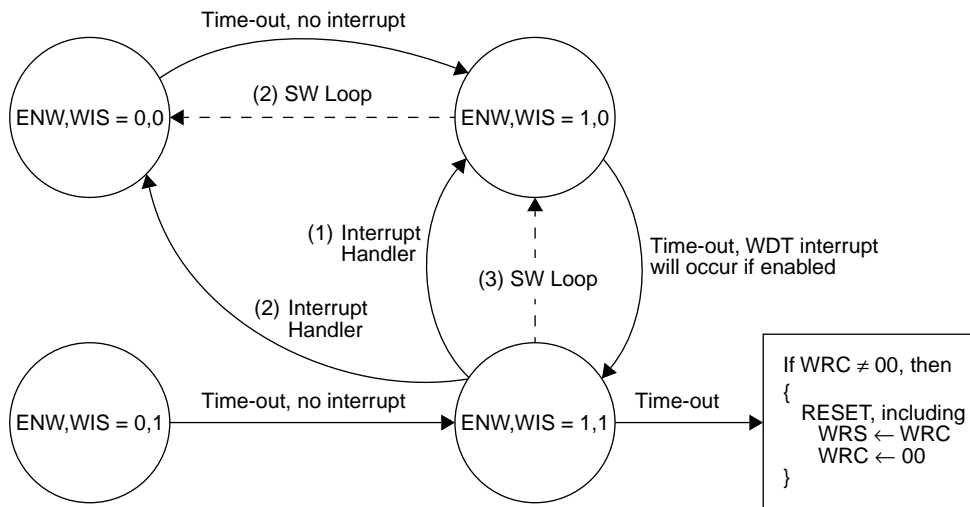


Figure 5-14. Watchdog Timer State Machine

Enable Next WDT TSR[ENW]	WDT Status TSR[WIS]	Action when timer interval expires
0	0	Set enable next watchdog (TSR[ENW] = 1).
0	1	Set TSR[ENW] = 1.
1	0	Set the watchdog interrupt status (TSR[WIS] = 1). If TCR[WIE] = 1 and MSR[CE] = 1, then interrupt.
1	1	Cause the watchdog reset action specified by TCR[WRC]. On reset, copy pre-reset TCR[WRC] into TSR[WRS] and clear TCR[WRC].

The controls described in the above table imply three different operating modes that a programmer can select for the WDT. The modes assume that TCR[WRC] was set to allow processor reset by the WDT:

1. Always take a pending WDT interrupt, and never attempt to prevent its occurrence. (This mode is described in the preceding text.)
 1. Clear TSR[WIS] in the WDT handler.
 2. Never use TSR[ENW].

2. Always take a pending WDT interrupt, but avoid it whenever possible.

This assumes that a recurring code loop of reliable duration exists outside the interrupt handlers, or that a FIT interrupt handler is operational. One of these mechanisms clears TSR[ENW] more frequently than the watchdog period.

1. Clear TSR[ENW] to 0 in loop or in FIT handler.

To clear TSR[ENW], use **mtspr** to write a 1 to TSR[ENW] (and to any other bits that are to be cleared), with 0 in all other bit locations.

2. Clear TSR[WIS] in WDT handler. (This is an unexpected event.)

3. Never take a WDT interrupt.

This assumes that a recurring code loop of reliable duration exists outside the interrupt handlers, or that a FIT interrupt handler is operational. This method only guarantees one WDT period before a reset occurs.)

1. Clear TSR[WIS] in the loop or in FIT handler.
2. Never use TSR[ENW].

5.15.5 Timer Status Register (TSR)

The TSR can be accessed for read or write-to-clear.

Status registers are generally set by hardware and read and cleared by software. The **mfspr** instruction reads the TSR. Clearing the TSR is performed by writing a word to the TSR, using **mtspr**, having a 1 in all bit positions to be cleared and a 0 in all other bit positions. The data written to the TSR is not direct data, but a mask. A 1 clears the bit and a 0 has no effect.

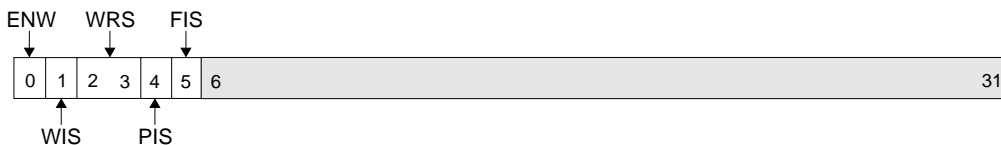


Figure 5-15. Timer Status Register (TSR)

0	ENW	Enable Next Watchdog 0 Action on next Watchdog event is to set TSR[0]. 1 Action on next Watchdog event is governed by TSR[1].	See Section 5.15.4 on p. 5-30.
1	WIS	Watchdog Interrupt Status 0 No Watchdog interrupt is pending. 1 Watchdog interrupt is pending.	

Figure 5-15. Timer Status Register (TSR)

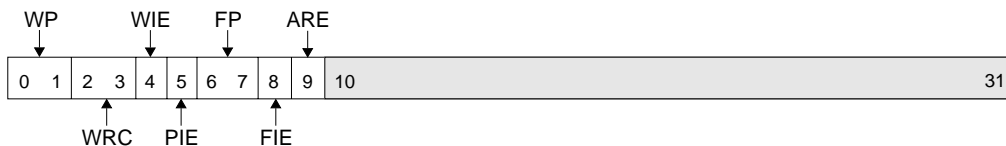
2:3	WRS	Watchdog Reset Status 00 No Watchdog reset has occurred. 01 Core reset was forced by the Watchdog. 10 Chip reset was forced by the Watchdog. 11 System reset was forced by the Watchdog.
4	PIS	PIT Interrupt Status 0 No PIT interrupt is pending. 1 PIT interrupt is pending.
5	FIS	FIT Interrupt Status 0 No FIT interrupt is pending. 1 FIT interrupt is pending.
6:31		Reserved

5.15.6 Timer Control Register (TCR)

The TCR controls PIT, FIT, and WDT operation.

The TCR[WRC] field is cleared to 0 by all processor resets. (Chapter 4, “Reset and Initialization,” describes the types of processor reset). This field is set only by software. However, hardware does not allow software to clear the field after it is set. After software writes a 1 to a bit in the field, that bit remains a 1 until any reset occurs. This prevents errant code from disabling the WDT reset function.

All processor resets clear TCR[ARE] to 0, disabling the auto-reload feature of the PIT.

**Figure 5-16. Timer Control Register (TCR)**

0:1	WP	Watchdog Period 00 2^{17} clocks 01 2^{21} clocks 10 2^{25} clocks 11 2^{29} clocks
-----	----	---

Figure 5-16. Timer Control Register (TCR)

2:3	WRC	Watchdog Reset Control 00 No Watchdog reset will occur. 01 Core reset will be forced by the Watchdog. 10 Chip reset will be forced by the Watchdog. 11 System reset will be forced by the Watchdog.	TCR[WRC] resets to 00. This field can be set by software, but cannot be cleared by software, except by a software-induced reset.
4	WIE	Watchdog Interrupt Enable 0 Disable WDT interrupt. 1 Enable WDT interrupt.	
5	PIE	PIT Interrupt Enable 0 Disable PIT interrupt. 1 Enable PIT interrupt.	
6:7	FP	FIT Period 00 2^9 clocks 01 2^{13} clocks 10 2^{17} clocks 11 2^{21} clocks	
8	FIE	FIT Interrupt Enable 0 Disable FIT interrupt. 1 Enable FIT interrupt.	
9	ARE	Auto Reload Enable 0 Disable auto reload. 1 Enable auto reload.	Disables on reset.
10:31		Reserved	

Cache Operations

The PPC401A1 incorporates two internal caches, a 2KB instruction cache unit (ICU) and a 1KB data cache unit (DCU).

The ICU controls instruction accesses to main memory and stores frequently used instructions to reduce the overhead of instruction transfers between the instruction queue and external memory, minimizing access latency for frequently executed instructions. The DCU controls data accesses to main memory and stores frequently used data to reduce the overhead of data transfers between the GPRs and external memory, minimizing access latency for frequently used data. Instructions and data can be accessed in the cache much faster than in main memory.

The ICU and DCU feature:

- Line fills in target-word-first, sequential, or any other order
- A separate bypass path to handle instructions and data in cache-inhibited memory, and to improve performance during line fills
- Cache line locking

The DCU features byte-writeability to improve the performance of byte and halfword operations, and supports write-back and write-through write strategies.

Section 6.1, “ICU Organization,” on p. 6-1, and Section 6.2, “DCU Organization,” on p. 6-4, describe the organization of the ICU and DCU, respectively.

6.1 ICU Organization

The ICU manages data transfers between external cacheable memory and the instruction queue in the execution unit.

The ICU contains a two-way set-associative 2KB cache memory. Each way is organized as 64 lines of four words (four instructions) each.

As shown in Figure 6-1, tag ways A and B store instruction address bits $A_{0:21}$ for each line in cache ways A and B. Instruction address bits $A_{22:27}$ serve as the index to the cache array.

The two cache lines that correspond to the same line index (one in each way) are called a congruence class.

Tags (Two-way Set)		Instructions (Two-way Set)	
Way A	Way B	Way A	Way B
A _{0:21} Line 0 A	A _{0:21} Line 0 B	Line 0 A	Line 0 B
A _{0:21} Line 1 A	A _{0:21} Line 1 B	Line 1 A	Line 1 B
•	•	•	•
•	•	•	•
•	•	•	•
A _{0:21} Line 62 A	A _{0:21} Line 62 B	Line 62 A	Line 62 B
A _{0:21} Line 63A	A _{0:21} Line 63B	Line 63 A	Line 63 B

Figure 6-1. Instruction Cache Organization

When a cache line is to be loaded, the cache way to receive the line is determined by using an least-recently-used (LRU) policy. The index, determined by the data address, selects a congruence class. Within a congruence class, the line which was accessed most recently is retained, and the other line is marked as LRU, using an LRU bit in the tag array. The line to receive the incoming data is the LRU line. After the cache line fill, the LRU bit is then set to identify as least-recently-used the line opposite the line just filled.

Figure 6-2 shows the relationships between the ICU and the instruction queue.

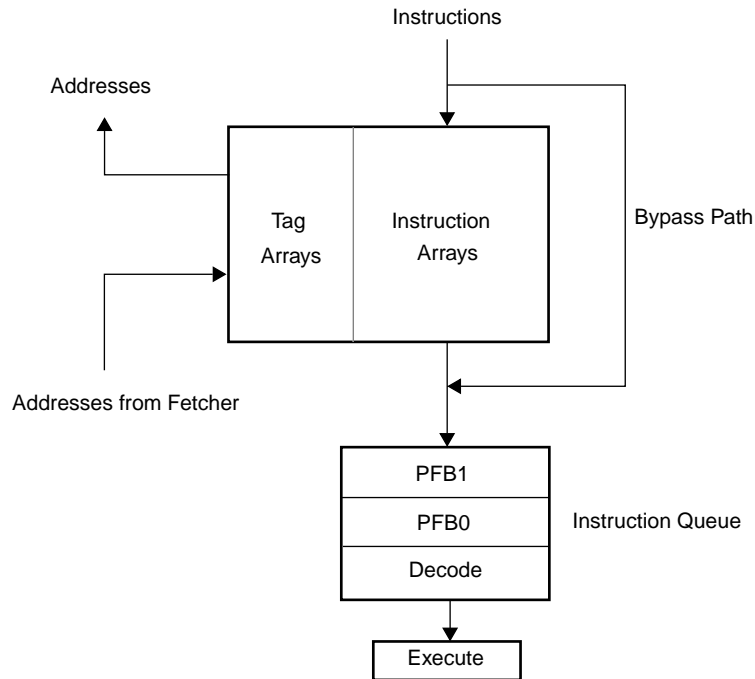


Figure 6-2. Instruction Flow

The bypass path handles instructions in cache-inhibited memory and improves performance during line fill operations. If a request from the fetcher obtains an entire line from memory, the queue does not have to wait for the entire line to reach the cache. The target word (the word requested by the fetcher) is sent on the bypass path to the queue while the line fill proceeds, even if the selected line fill order is not target-word-first.

6.1.1 ICU Operations

Instructions from cacheable memory regions are copied into the instruction cache, from which they can be accessed by the fetcher far more quickly than they can be obtained from memory. Cache lines are loaded either target-word-first or sequentially. Target-word-first fills start at the requested word, continue to the end of the line, and then wrap to fill the remaining words at the beginning of the line. Sequential fills start at the first word of the cache line and proceed sequentially to the last word of the line.

Cache line fills always run to completion, even if the instruction stream branches away from the rest of the line. As requested instructions are received, they go to the fetcher before the line fills in the cache. The filled line is always placed in the ICU; if an external memory

subsystem error occurs during the fill, the line is marked as invalid. During a clock cycle, the ICU can send one instruction to the fetcher.

6.1.2 Instruction Cacheability Control

Instruction cacheability is controlled by the Instruction Cache Cacheability Register (ICCR). Each bit in the ICCR (ICCR[S0:S31]) controls the cacheability of a 128MB region (see Section 8.1.3, “Instruction Cache Cacheability Register (ICCR),” on p. 8-6). If ICCR[Sn] = 1, caching is enabled for the specified region.

The performance of the PPC401A1 is significantly lower while executing in cache-inhibited regions.

Following system reset, all ICCR bits are reset to 0 so that no memory regions are cacheable. Before regions can be designated as cacheable in the ICCR, it is necessary to execute the **iccci** instruction 64 times (once for each congruence class in the cache array). This invalidates all congruence classes before enabling the cache. The ICCR can then be reconfigured appropriately and the ICU can begin normal operation.

6.1.3 ICU Coherency

The ICU does not “snoop” external memory or the DCU. Programmers must follow special procedures for ICU synchronization when self-modifying code is used or if a peripheral device updates memory containing instructions.

The following code example illustrates the necessary steps for self-modifying code. This example assumes that *addr1* is both data and instruction cacheable.

```

stw      regN, addr1    # the data in regN is to become an instruction at addr1
dcbst    addr1          # forces data from the data cache to memory
sync     # wait until the data actually reaches the memory
icbi     addr1          # the previous value at addr1 might already be in
                        # the instruction cache; invalidate it in the cache
isync    # the previous value at addr1 may already have been
                        # pre-fetched into the queue; invalidate the queue
                        # so that the instruction must be re-fetched
```

6.2 DCU Organization

The DCU manages data transfers between external cacheable memory and the general-purpose registers in the execution unit.

The DCU contains a two-way set-associative 1KB cache memory. Each way is organized as 32 lines of four words (four instructions) each.

As shown in Figure 6-3, tag ways A and B store instruction address bits A_{0:22} for each line in cache ways A and B. Instruction address bits A_{23:27} serve as the index to the cache array.

The two cache lines that correspond to the same line index (one in each way) are called a congruence class.

Tags (Two-way Set)		Instructions (Two-way Set)	
Way A	Way B	Way A	Way B
A _{0:23} Line 0 A	A _{0:23} Line 0 B	Line 0 A	Line 0 B
A _{0:23} Line 1 A	A _{0:23} Line 1 B	Line 1 A	Line 1 B
•	•	•	•
•	•	•	•
•	•	•	•
A _{0:23} Line 62 A	A _{0:23} Line 62 B	Line 30 A	Line 30 B
A _{0:23} Line 63A	A _{0:23} Line 63B	Line 31 A	Line 31 B

Figure 6-3. Data Cache Organization

When a cache line is to be loaded, the cache way to receive the line is determined by using an LRU policy. The index, determined by the data address, selects a congruence class. Within a congruence class, the line which was accessed most recently is retained, and the other line is marked as LRU, using an LRU bit in the tag array. The line to receive the incoming data is the LRU line. After the cache line fill, the LRU bit is then set to identify as least-recently-used the line opposite the line just filled.

A bypass path handles data operations in cache-inhibited memory and improves performance during line fill operations.

6.2.1 DCU Operations

Data from cacheable memory regions are copied into lines in the data cache, either target-word-first or sequentially. Target-word-first fills start at the requested word, continue to the end of the line, and then wrap to fill the remaining words at the beginning of the line. Sequential fills start at the first word of the cache line and proceed sequentially to the last word of the line. In both types of fills, the line is marked valid when the fourth word is filled.

GPRs receive the requested byte, halfword, or fullword of data immediately upon being received from main storage using a cache bypass mechanism. As requested data is received, it is forwarded to the target at the same time the data is written to the cache. The filled line is always placed in the DCU. The DCU can send a byte, halfword, or fullword to the target in two clock cycles.

Cache flushing (copying data in the cache that has been updated by the processor to main storage) and filling (loading requested data from main storage into the cache) are triggered by load, store and cache control instructions executed by the processor. Cache flushes are always sequential, starting at the first word of the cache block and proceeding sequentially to the end of the block.

Cache lines are always completely flushed or filled, even if the program does not request the rest of the bytes in the line, or if a bus error occurs after a bus interface unit accepts the request for the line fill. If a bus error occurs during a line fill, the line is filled and the data is marked valid. However, the line can contain invalid data, and a machine check exception occurs.

The DCU supports byte-writeability to improve the performance of byte and halfword store operations.

6.2.2 DCU Write Strategies

DCU operations can use write-back or write-through strategies to maintain coherency with external cacheable memory.

The write-back strategy updates only the data cache, not external memory, during store operations. Only modified data lines are flushed to external memory, and then only when necessary to free up locations for incoming lines, or when lines are explicitly flushed using **dcbf** or **dcbst** instructions. Cache flushes are always sequential, starting at the first word of the cache block and proceeding sequentially to the end of the block. The write-back strategy minimizes the amount of external bus activity and avoids unnecessary contention for the external bus between the ICU and the DCU.

The write-back strategy is contrasted with the write-through strategy, in which stores are written simultaneously to the cache and to external memory. A write-through strategy can simplify maintaining coherency between cache and memory. However, because the DCU cannot accept a new command until such a store completes in external memory, performance is generally slower.

The write strategy is controlled by the Data Cache Write-through Register (DCWR). Each bit in the DCWR (DCWR[W0:W31]) controls the write strategy of a 128MB storage region (see Section 8.1.1, “Data Cache Write-through Register (DCWR),” on p. 8-2). If DCWR[W_n] = 0, the write-back strategy is enabled for the specified region; if DCWR[W_n] = 1, the write-through strategy is enabled.

Programming Note: The PowerPC Architecture does not support memory models in which write-through is enabled and caching is inhibited.

The DCU can control whether a cache line is allocated in the cache on store misses. The Cache Debug Control Register (CDBCR) write-on-allocate (WOA) bit controls the allocate-on-write policy. If CDBCR[WOA] = 0, store misses cause a line fill. If CDBCR[WOA] = 1, store misses do not cause a line fill, but result in a non-cacheable store.

6.2.3 Data Cacheability Control

Data cacheability is controlled by the Data Cache Cacheability Register (DCCR). Each bit in the DCCR (DCCR[S0:S31]) controls the cacheability of a 128MB region (see Section 8.1.2, “Data Cache Cacheability Register (DCCR),” on p. 8-4). If DCCR[S_n] = 1, caching is enabled for the specified region.

The performance of the PPC401A1 is significantly lower while executing in cache-disabled regions.

Following system reset, all DCCR bits are reset to 0 so that no memory regions are cacheable. Before regions can be designated as cacheable in the DCCR, it is necessary to execute the **dccci** instruction 32 times (once for each congruence class in the cache array). This invalidates all congruence classes before enabling the cache. The DCCR can then be reconfigured appropriately and the ICU can begin normal operation.

Programming Note: If a data block corresponding to the effective address (EA) exists in the cache, but the EA is non-cacheable, loads and stores (including **dcbz**) to that address are considered programming errors (the cache block should previously have been flushed). The only instructions that can legitimately access such an EA in the data cache are the cache management instructions **dcbf**, **dcbi**, **dcbst**, **dcbt**, **dcbtst**, **dccci**, and **dcread**.

6.2.4 DCU Coherency

The DCU does not provide snooping. Application programs must carefully use cache-inhibited regions and cache control instructions to ensure proper operation of the cache in systems where external devices can update memory.

6.3 Cache Instructions

For detailed descriptions of the instructions described in the following sections, see Chapter 9, “Instruction Set.”

In the instruction descriptions, the term “block” is synonymous with cache line. A block is the unit of storage operated on by all cache block instructions.

6.3.1 ICU Instructions

The following instructions control instruction cache operations:

- **icbi** Instruction Cache Block Invalidate

Invalidates a cache block. If the data cache unlock exception is enabled, a data storage exception occurs, regardless of whether the target line is locked.

- **icbt** Instruction Cache Block Touch

Initiates a block fill, enabling a program to begin a cache block fetch before the program needs an instruction in the block. The program can subsequently branch to the instruction address and fetch the instruction without incurring a cache miss. This is a privileged-mode instruction.

- **iccci** Instruction Cache Congruence Class Invalidate

Invalidates a congruence class (in the PPC401A1, both ways). This is a privileged-mode instruction.

- **icread** Instruction Cache Read

Reads either an instruction cache tag entry or an instruction word from an instruction cache line, typically for debugging. Bits in the CDBCR control instruction behavior (see Section 6.4, “Cache Control and Debugging Features,” on p. 6-9). This is a privileged-mode instruction.

6.3.2 DCU Instructions

Data cache flushes and fills are triggered by load, store and cache control instructions. Cache control instructions are provided to fill, flush, or invalidate cache blocks.

The following instructions control data cache operations.

- **dcbf** Data Cache Block Flush

Flushes a line, if found in the cache and marked as modified, to external memory; the line is then marked invalid. If the line is found in the cache and is not marked modified, the line is marked invalid but is not flushed. This operation is performed regardless of whether the address is marked cacheable. If the data cache unlock exception is enabled, a data storage exception occurs, regardless of whether the target line is locked.

- **dcbi** Data Cache Block Invalidate

Invalidates a block, if found in the cache, regardless of whether the address is marked cacheable. Any modified data is not flushed to memory. This is a privileged-mode instruction.

- **dcbst** Data Cache Block Store

Stores a block, if found in the cache and marked as modified, into external memory; the block is not invalidated but is no longer marked as modified. If the block is marked as not modified in the cache, no operation is performed. This operation is performed regardless of whether the address is marked cacheable.

- **dcbt** Data Cache Block Touch

Fills a block with data, if the address is cacheable and the data is not already in the cache. If the address is non-cacheable, this instruction is a no-op.

- **dcbtst** Data Cache Block Touch for Store

Implemented identically to the **dcbt** instruction in the PPC401A1 for compatibility with compilers and other tools.

- **dcbz** Data Cache Block Set to Zero

Fills a line in the cache with zeros and marks the line as modified. If the line is not currently in the cache (and the address is marked as cacheable and non-write-through), the line is established, filled with zeros, and marked as modified without actually filling the line from external memory. If the line is marked as either non-cacheable or write-through, an alignment exception results. If the target is not in the cache and the data cache lock-out exception is enabled, a data storage exception occurs.

- **dccci** Data Cache Congruence Class Invalidate

Invalidates a congruence class (in the PPC401A1, both ways). This is a privileged-mode instruction.

- **dcread** Data Cache Read

Reads either a data cache tag entry or a data word from a data cache line, typically for debugging. Bits in the CDBCR control instruction behavior (see Section 6.4, “Cache Control and Debugging Features,” on p. 6-9). This is a privileged-mode instruction.

6.4 Cache Control and Debugging Features

Registers and instructions are provided to control cache operation and help debug cache problems. For ICU debug, the **icread** instruction and the Instruction Cache Debug Data Register (ICDBDR) are provided (see Section 6.4.1, “ICU Debugging,” on p. 6-10). For DCU debug, the **dcread** instruction is provided (see Section 6.4.2, “DCU Debugging,” on p. 6-11).

The CDBCR controls the behavior of the **icread** and the **dcread** instructions.

Figure 6-4 illustrates the CDBCR fields.

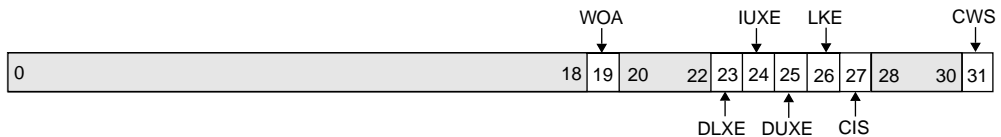


Figure 6-4. Cache Debug Control Register (CDBCR)

0:18		Reserved
19	WOA	Write-on-Allocate 0 All store misses result in a line fill. 1 Store misses do not cause a line fill, but result in a non-cacheable store.
20:22		Reserved
23	DLXE	DCU Lock-out Exception Enable 0 DCU lock-out exception is disabled. 1 DCU lock-out exception is enabled.

Figure 6-4. Cache Debug Control Register (CDBCR) (cont.)

24	IUXE	ICU Unlock Exception Enable 0 ICU unlock exception is disabled. 1 ICU unlock exception is enabled.
25	DUXE	DCU Unlock Exception Enable 0 DCU unlock exception is disabled. 1 DCU unlock exception is enabled.
26	LKE	Lock Enable 0 Line locking is disabled. 1 Line locking is enabled.
27	CIS	Cache Information Select 0 Information is cache data. 1 Information is cache tag.
28:30		Reserved
31	CWS	Cache Way Select 0 Cache way is A. 1 Cache way is B.

6.4.1 ICU Debugging

The **icread** instruction enables the reading of the instruction cache entries for the congruence class specified by $EA_{22:27}$. The cache information is read into the ICDBDR; from there it can subsequently be moved, using a **mfspr** instruction, into a GPR.

Figure 6-5 illustrates the ICDBDR.

0	31
---	----

Figure 6-5. Instruction Cache Debug Data Register (ICDBDR)

0:31	Instruction cache information from icread
------	--

If $CDBCR[CIS] = 0$, the data is a word of ICU data from the addressed line, specified by $EA_{28:29}$. If $CDBCR[CWS] = 0$, the data is from the A-way; otherwise, the data from the B-way.

If $CDBCR[CIS] = 1$, the cache information is the cache tag. If $CDBCR[CWS] = 0$, the tag is from the A-way; otherwise, the tag is from the B-way.

ICU tag information is placed into the ICDBDR as shown:

0:22	TAG	Cache Tag
23:24		Reserved
25	LK	Cache Line Lock 0 Unlocked 1 Locked
26		Reserved
27	V	Cache Line Valid 0 Not valid 1 Valid
28:30		Reserved
31	LRU	Least Recently Used (LRU) 0 A-way LRU 1 B-way LRU

Programming Note: The instruction pipeline does not wait for data from an **icread** instruction to arrive before attempting to use the contents the ICDBCR.

The following code sequence ensures proper results:

```
icread r5,r6           # read cache information
isync                  # ensure completion of icread
mficbdr r7             # move information to GPR
```

6.4.2 DCU Debugging

The **dcread** instruction provides a debugging tool for reading the data cache entries for the congruence class specified by $EA_{23:27}$. The cache information is read into a GPR.

If $CDBCR[CIS] = 0$, the data is a word of DCU data from the addressed line, specified by $EA_{28:29}$. If $CDBC[CWS] = 0$, the data is from the A-way; otherwise, the data is from the B-way.

If $CDBCR[CIS] = 1$, the cache information is the cache tag. If $CDBC[CWS] = 0$, the tag is from the A-way; otherwise the tag is from the B-way.

DCU tag information is placed into the GPR as shown:

0:22	TAG	Cache Tag
23:24		Reserved
25	LK	Cache Line Locked 0 Unlocked 1 Locked

26	D	Cache Line Dirty 0 Not dirty 1 Dirty
27	V	Cache Line Valid 0 Not valid 1 Valid
28:30		Reserved
31	LRU	Least Recently Used (LRU) 0 A-way LRU 1 B-way LRU

Note: A “dirty” cache line is one which has been accessed by a store instruction after it was established, and can be inconsistent with external memory.

6.5 Cache Line Locking

Lines in the ICU and DCU can be locked. Locked lines remain in the cache arrays until they are explicitly unlocked. Locking frequently used instructions and data in the respective cache units can ensure that the instructions and data are available as quickly as possible.

If a line is locked in a congruence class in the ICU or DCU, that line does not participate in the LRU replacement for that congruence class. Typically, on a cache miss, the LRU unlocked line is replaced.

Except for the **dcba**, **dcbt**, **dcbtst**, **dcbz**, and **icbt** instructions, a load, store, or cache instruction that misses in the cache, and references a locked-out congruence class (a congruence class in which both lines are locked), behaves as if the instruction references non-cacheable storage. The **dcbt**, **dcbtst**, and **icbt** instructions do not update the cache in this case, but the PLB appears to have performed a line fill. The **dcba** and **dcbz** instructions behave as described in Section 6.5.2, “Unlocking Lines in the ICU and DCU,” on p. 6-13.

6.5.1 Locking Lines in the ICU and DCU Cache Arrays

Line locking is controlled by the CDBCR[LKE] bit (see Figure 6-4 on p. 6-9). To lock lines in the cache units, an instruction sequence, executing in privileged mode, sets CDBCR[LKE] = 1 to enable locking, and establishes cache lines as needed, using a **dcba**, **dcbt**, **dcbtst**, **dcbz**, or **icbt** instruction as appropriate to establish and lock each line. Locked lines are marked as locked in the appropriate cache tag array. The instruction sequence can then set CDBCR[LKE] = 0, disabling line locking, and return the processor to user mode, making the locked lines available to application code.

Some instructions that establish cache lines in the DCU are non-privileged. However, the enabling and disabling of cache line locking requires the use of the privileged **mtcdbr** instruction (actually, **mtcdbr** is an extended mnemonic for the **mtspr** instruction). Note that

cache line locking should be performed thoughtfully; limiting the cache line locking window in a privileged code sequence as described is recommended.

Programming Note: The **mtcdbc** extended mnemonic is not context- or execution-synchronizing. Software must place appropriate synchronization instructions before and after an **mtcdbc** to ensure that the locking instructions execute in the proper context.

6.5.2 Unlocking Lines in the ICU and DCU

Several instructions unlock locked cache lines: the privileged instructions **dccci**, **dcbi**, and **iccci**; and the non-privileged instructions **dcba**, **dcbf**, **dcbz**, and **icbi**.

In user mode, the behavior of the non-privileged unlocking cache instructions depends upon the settings of the CDBCR[DLXE], CDBCR[IUXE], and CDBCR[DUXE] bits. These bits are associated with data cache instructions:

- **dcbf**

The CDBCR[DUXE] field (DUXE stands for Data-cache Unlock Exception Enable) controls whether an attempt to flush a locked data cache line causes an exception.

If the field is disabled (CDBCR[DUXE] = 0), the target line is flushed and unlocked, and no exception is taken. If the field is enabled (CDBCR[DUXE] = 1), **dcbf** causes a data storage exception. The exception occurs regardless of whether the target line is locked.

- **dcbz**

The CDBCR[DLXE] field controls whether an attempt to replace an locked data cache line causes an exception.

Assume that the **dcbz** instruction hits in the cache, that the W and I storage attributes for the target are 0, and that the target is a valid cache line. Assume further that the target line is locked.

If the DLXE field is enabled (CDBCR[DLXE] = 1), a data storage exception occurs if the target is not in the cache. The exception occurs regardless of whether the target congruence class is locked-out.

If the DLXE field is disabled (CDBCR[DLXE] = 0), the target line is unlocked and removed. A new line is established. If CDBCR[LKE] = 1, the newly established line is locked. No exception is taken.

If **dcbz** references a non-cacheable address (alignment exception), and CDBCR[DLXE] = 1 (data storage exception), the alignment exception takes priority.

If **dcbz** references a cacheable address, write-through required (alignment exception), and CDBCR[DLXE] = 1 (data storage exception), the data storage exception takes priority.

- **icbi**

The CDBCR[IUXE] field (IUXE stands for Instruction-cache Unlock Exception Enable) controls whether an attempt to invalidate a locked instruction cache line causes an exception.

If the field is disabled (CDBCR[IUXE] = 0), the target line is invalidated and unlocked, and no exception is taken. If the field is enabled (CDBCR[IUXE] = 1), a data storage exception occurs. The exception occurs regardless of whether the target line is locked.

In privileged mode, no data storage exceptions occur when the privileged and non-privileged cache instructions execute, regardless of the setting of the locked-out and unlocked exception bits in the CDBCR.

- **dcbz**

dcbz replaces the LRU unlocked line in a congruence class, regardless of whether the congruence class is locked out.

6.6 DCU Performance

DCU performance depends upon the application and the design of the attached external bus controller, but, in general, cache hits complete in two cycles without stalling the CPU pipeline. Under certain conditions and limitations of the DCU, the pipeline stalls (stops executing instructions) until the DCU completes current operations.

Several factors affect DCU performance, including:

- Pipeline stalls
- DCU priority
- Simultaneous cache operations
- Sequential cache operations

6.6.1 Pipeline Stalls

The CPU issues commands for cache operations to the DCU. If the DCU can immediately perform the requested cache operation, no pipeline stall occurs. In some cases, however, the DCU cannot immediately perform the requested cache operation, and the pipeline stalls until the DCU can perform the pending cache operation.

A load miss or a load to a storage region marked as non-cacheable always stalls the pipeline until the load is aborted or the requested load data becomes available to the CPU, enabling it to resume instruction execution.

Other pipeline stalls occur when the CPU issues a command for a cache operation while one of the following cache operations, previously requested, is in progress:

- A line fill resulting from a store

- Two pending line flushes
- A **dcbt** instruction
- A store to a region of memory marked as write-through

A store miss stalls the pipeline only when the CPU issues a request for a cache operation while the line fill resulting from the store miss is in progress. When the line fill finishes, instruction execution resumes.

Multiple line flushes can stall the pipeline. The DCU can flush up to two lines to memory before stalling the pipeline when the CPU issues another command for a cache operation before the second line flush begins.

When a **dcbt** instruction results in a cache miss and is followed by another cache operation, the pipeline stalls until the line fill resulting from the **dcbt** miss finishes. Then, instruction execution resumes.

When a storage region is marked as write-through, all stores, when followed immediately by another cache operation, stall the pipeline until the cycle after the BIU acknowledges the store data. This results in at least a 1-cycle delay to complete the write through. However, if a store to a storage region marked as write-through misses, and the device supplying the data for the resulting line fill has a store queue, the store queue holds the store data until the line fill finishes. In this case, the write-through appears to complete before the line fill.

6.6.2 Cache Operation Priorities

The DCU uses a priority signal, **DCU_biuPriority**, to improve performance when pipeline stalls occur. When the pipeline is stalled because of a data cache operation, the DCU asserts the priority signal to the external bus. **DCU_biuPriority** tells the external bus that the DCU requires immediate service, and is valid only when the data cache is requesting access to the BIU interface. **DCU_biuPriority** is asserted for all loads that require external data, or when the data cache is requesting the external bus and stalling an operation that is being presented to the data cache.

Table 6-1 provides examples of when the priority is asserted and deasserted.

Table 6-1. Priority Changes With Different Data Cache Operations

Instruction	Priority	Next Instruction	Priority
Any load	1	N/A	N/A
Any store	0	Any other cache operation when the line fill was not accepted on the BIU interface.	1
dcbf	0	Any cache hit.	0
dcbf/dcbst	0	Load non-cache when requesting access to the BIU interface.	1

Table 6-1. Priority Changes With Different Data Cache Operations

Instruction	Priority	Next Instruction	Priority
dcbf/dcbst	0	Any line fill when dcbf/dcbst requests access to the BIU interface.	1
dcbf/dcbst	0	dcbf/dcbst when the first dcbf/dcbst requests access to the BIU interface.	1
dcbt	0	Any cache hit.	0
dcbt	0	Any other cache operation when a line fill was not accepted on the BIU interface.	1
dcbi/dccci/dcbz	0	N/A	N/A

6.6.3 Simultaneous Cache Operations

Some cache operations can occur simultaneously to improve DCU performance. For example, when a line must be allocated in the data cache and another line flushed to memory, the operations can be performed simultaneously. A line fill caused by a store miss to a storage region marked as write-through, and a load to a non-cacheable storage region marked followed by a line flush, can also be performed simultaneously. Support for simultaneous operations depends upon the configuration of the external bus.

6.6.4 Sequential Cache Operations

Some common cache operations, when performed sequentially, can limit DCU performance: sequential loads/stores to non-cacheable storage regions, sequential line fills, and sequential line flushes.

In the case of sequential cache hits, the most commonly occurring operations, the DCU loads or stores data every two cycles. In such cases, the DCU does not affect performance.

However, when a load from a non-cacheable storage region is accepted by the PLB in the same cycle as the load request, and the load data is presented in the next cycle, three cycles are required to complete the load.

The following equation determines the number of cycles required to complete a series of sequential loads from non-cacheable storage regions:

$$\text{Total cycles} = (2 + (car + cgd)) \times \text{Number of loads}$$

where *car* is the number of cycles required to accept the load request (if the request is accepted in the same cycle, *car* = 0) and *cgd* is the number of cycles required to get the load data after the request is accepted.

Similarly, when a store to a non-cacheable storage region is accepted and the store data is presented in the same cycle, at least three cycles are required to complete the store.

The following equation determines the number of cycles required to complete a series of sequential stores to storage regions marked as non-cacheable:

$$\text{Total cycles} = (3 + (car + cad)) \times \text{Number of stores}$$

where *car* is the number of cycles required to accept the store request (if the request is accepted in the same cycle, *car* = 0) and *cad* is the number of cycles required to acknowledge the store data after the request is accepted.

Sequential line fills can limit DCU performance. Line fills occur when a load/store or **dcbt** instruction misses in the cache. For more information about sequential line fills and their timings, see Chapter 3, “I/O Interfaces.”

Sequential line flushes from the DCU to main memory also limit DCU performance. Flushes occur when a line fill replaces a valid line that is marked dirty (modified), or when a **dcbf** instruction flushes a specific line. If two flushes are pending, the DCU stalls any new data cache operations until the first flush finishes and the second begins. For more information about sequential flushes and their timings, see Chapter 3, “I/O Interfaces.”

7

Debugging and JTAG Facilities

The PPC401A1 debug facilities include support for debug modes for debugging during hardware and software development and debug events that allow developers to control the debug process. Debug registers control the debug modes and debug events. The debug registers are accessed through software running on the processor or through a JTAG debug port. The debug interface is the JTAG debug port. The JTAG debug port can also be used for board test.

The debug modes, events, controls, and interface provide a powerful combination of debug facilities for a wide range of hardware and software development tools.

7.1 Development Tool Support

The OS Open Real-time Operating System Debugger product from IBM is an example of an operating system-aware debugger, implemented using software traps.

The RISCWatch product from IBM is an example of a development tool that uses the external debug mode, debug events, and the JTAG debug port to implement a hardware and software development tool.

7.2 Debug Modes

There are two debug modes in the PPC401A1; each supports a type of debug tool commonly used in embedded systems development:

- Internal debug mode, which supports ROM monitors
- External debug mode, which supports emulators

Both modes can be enabled simultaneously; the Debug Control Register (DBCR) controls the internal and external debug modes.

7.2.1 Internal Debug Mode

Internal debug mode supports access to all architected processor resources, setting hardware and software breakpoints, and monitoring processor status. In this mode, debug events generate debug exceptions, which can interrupt normal program flow so that monitor software can collect processor status and alter processor resources.

Internal debug mode relies on exception handling software at a dedicated interrupt vector and an external communications path to debug software problems. This mode, used while the processor executes instructions, enables debugging of operating system or application programs.

In this mode, debugger software is accessed through a communications port, such as a serial port, external to the processor core.

7.2.2 External Debug Mode

External debug mode provides access to all architected processor resources and supports stopping, starting, and stepping the processor, setting hardware and software breakpoints, and monitoring processor status. In this mode, debug events cause the processor to become architecturally frozen. While the processor is frozen, normal instruction execution stops and all architected processor resources can be accessed and altered. External bus activity continues in external debug mode.

The JTAG mechanism can also pass instructions to the processor for execution, allowing a JTAG debug tool to display and alter processor resources, including memory.

The JTAG mechanism prevents the occurrence of a privileged exception when a privileged instruction is executed while the processor is in user mode.

Storage access control remains in effect while in external debug mode.

External debug mode relies only on internal processor resources, so it can be used to debug both system hardware and software problems. This mode can also be used for software development on systems without a control program, or to debug control program problems.

Access to the processor, while in external debug mode, is through the JTAG debug port.

7.3 Processor Control

The PPC401A1 provides the following debug facilities for processor control. Not all facilities are available in all debug modes.

Instruction Step	The processor is stepped one instruction at a time, while stopped, using the JTAG debug port.
Instruction Stuff	While the processor is stopped, instructions can be stuffed into the processor and executed using the JTAG debug port.

Halt	The processor can be stopped by activating an external halt signal on an external event, such as a logic analyzer trigger. This signal freezes the processor architecturally. While frozen, normal instruction execution stops and all architected processor resources can be accessed and altered using the JTAG debug port. Normal execution resumes when the halt signal is deactivated.
Stop	The processor can be stopped using the JTAG debug port. Activating a stop causes the processor to become architecturally frozen. While frozen, normal instruction execution stops and the architected processor resources can be accessed and altered using the JTAG debug port. Normal execution resumes when this bit is reset.
Reset	An external reset signal, the JTAG debug port, or the Debug Control Register (DBCR) can request a reset of the processor. The JTAG debug port can request core, chip, and system resets. See Section 3.7, “Reset Interface,” on p. 3-20, for more information.
Debug Events	A debug event triggers a debug operation. The operation depends on the debug mode. For more information and a list of debug events, see Section 7.5, “Debug Events,” on p. 7-4.
Freeze Timers	The JTAG debug port or the DBCR can control timer resources. The timers can be enabled to run, freeze always, or freeze on a debug event.
Trap Instructions	The trap instructions tw and twi can be used, with debug events, to implement software breakpoints.

7.4 Processor Status

The processor execution status, exception status, and most recent reset can be monitored.

Execution Status	The JTAG debug port can monitor processor execution status to determine whether the processor is stopped, waiting, or running.
Exception Status	The JTAG debug port can monitor the status of pending synchronous exceptions.
Most Recent Reset	The JTAG debug port can read the Debug Status Register (DBSR) to determine the type of the most recent reset.

7.5 Debug Events

Debug events, enabled by the DBCR and recorded in the DBSR, trigger debug operations. A debug event occurs when an event listed in Table 7-1 is detected. The debug operation is performed after the debug event.

In internal debug mode, the processor generates a debug exception when a debug event occurs. In external debug mode, the processor stops when a debug event occurs. When internal and external debug mode are both enabled, the processor also stops on any debug event. When external and internal debug mode are both disabled, debug events are recorded in the DBSR, but no action is taken.

Table 7-1. Debug Events

Event	Description
Exception	This debug event occurs after an exception. Exception debug events always include the non-critical class of exceptions unless in internal debug mode. When only internal debug mode is enabled, the critical class of exceptions are not included. This debug event is disabled if the Machine State Register (MSR) field MSR[DE] = 0 and the DBCR field DBCR[IDM] = 1.
Branch Taken	This debug event occurs before the execution of a branch instruction that is determined to be taken. This debug event is disabled if MSR[DE] = 0 and DBCR[IDM] = 1.
Data Address Compare (DAC)	This debug event occurs before the execution of an instruction that accesses a data address that matches the contents of the Data Address Compare Register (DAC). The DBCR can set the DAC to occur on reads/writes from byte addresses, or from any byte within halfword, word, or quad-word addresses.
Instruction Address Compare (IAC)	This debug event occurs before the execution of an instruction at an address that matches the contents of the Instruction Address Compare Register (IAC1).
Instruction Completion	This debug event occurs after the completion of any instruction. This debug event is disabled if MSR[DE] = 0 and DBCR[IDM] = 1.
Trap	This debug event occurs before the execution of a trap instruction where the conditions are such that the trap will occur.
Unconditional	This debug event occurs immediately upon being set by the JTAG debug port or the XXX_cpuUncondDebugEvent signal. This signal supports user-defined debug events, using ASIC logic external to the PPC401A1.

7.6 Debug Registers

Several debug registers, available to debug tools running on the processor, are not intended for use by application code. Debug tools control debug resources such as debug events. Application code that uses debug resources can cause the debug tools to fail, as well as other unexpected results, such as program hangs and processor resets.

Application code should not use the debug resources, including the debug registers.

7.6.1 Debug Control Register (DBCR)

The DBCR can enable debug events, reset the processor, control timer operation during debug events, enable JTAG interrupts, and set the processor debug mode.

Application code should not use the DBCR.

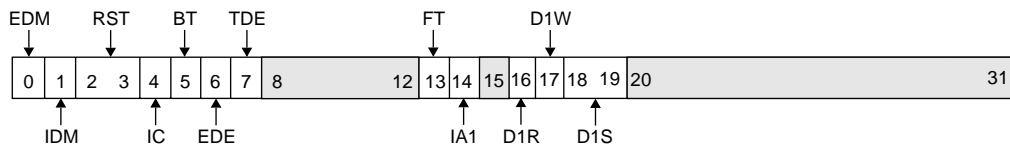


Figure 7-1. Debug Control Register (DBCR)

0	EDM	External Debug Mode 0 Disable 1 Enable	
1	IDM	Internal Debug Mode 0 Disable 1 Enable	
2:3	RST	Reset 00 No action 01 Core reset 10 Chip reset 11 System reset Attention: Writing 01, 10, or 11 to this field causes a processor reset request.	
4	IC	Instruction Completion Debug Event 0 Disable 1 Enable	Instruction completion does not cause a debug event if MSR[DE] = 0 in internal debug mode
5	BT	Branch Taken Debug Event 0 Disable 1 Enable	Branch taken does not cause a debug event if MSR[DE] = 0 in internal debug mode
6	EDE	Exception Debug Event 0 Disable 1 Enable	Critical exceptions do not cause debug events if MSR[DE] = 0 in internal debug mode

Figure 7-1. Debug Control Register (DBCR) (cont.)

7	TDE	TRAP Debug Event 0 Disable 1 Enable
8:12		Reserved
13	FT	Freeze timers on debug event 0 Free-run timers 1 Freeze timers
14	IA1	Instruction Address Compare 1 Enable 0 Disable 1 Enable
15		Reserved
16	D1R	DAC Read Enable 0 Disable 1 Enable
17	D1W	DAC Write Enable 0 Disable 1 Enable
18:19	D1S	DAC Size 00 Compare all bits 01 Ignore the least significant bit (lsb) 10 Ignore the two lsbs 11 Ignore the four lsbs Exact address compare Byte within halfword address compare Byte within word address compare Quadword address compare
20:31		Reserved

7.6.1.1 Note on the DAC Compare Size Field (DBCR[D1S])

The data address compare (DAC) debug event can be set to react to any byte in a larger block of memory, in addition to reacting to an exact address match. The DAC Compare Size field (DBCR[D1S]) allows DAC debug events to react to any byte in a halfword, word, or line of four words (quadword).

The address for a DAC is the effective address (EA) of a storage reference instruction.

As an example, suppose that a DAC debug event should react to byte 3 of a word-aligned target. A DAC set for exact compare would not recognize a reference to that byte by load/store word or load/store halfword instructions, because the byte address is not the EA of such instructions. In such a case, the D1S field must be set for a wider capture range (for example, to ignore the two least significant bits (LSbs) if word operations to the misaligned byte are to be detected). The wider capture range results in excess debug events (events that are within the specified capture range, but reflect byte operations in addition to the desired byte). Such excess debug events must be handled by software.

7.6.2 Debug Status Register (DBSR)

The DBSR contains status on debug events and the most recent reset; the status is obtained by reading the DBSR. The status bits are normally set by debug events or by any of the three reset types.

Clearing the DBSR fields is performed by writing a word to the DBSR, using the **mtdbsr** extended mnemonic, having a 1 in all bit positions to be cleared and a 0 in the all other bit positions. The data written to the DBSR is not direct data, but a mask. A 1 clears the bit and a 0 has no effect.

Application code should not use the DBSR.

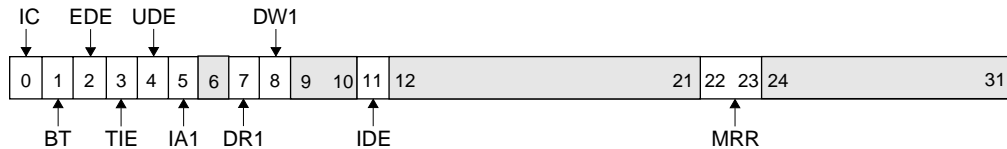


Figure 7-2. Debug Status Register (DBSR)

0	IC	Instruction Completion Debug Event 0 Event didn't occur 1 Event occurred
1	BT	Branch Taken Debug Event 0 Event didn't occur 1 Event occurred
2	EDE	Exception Debug Event 0 Event didn't occur 1 Event occurred
3	TIE	TRAP Instruction Debug Event 0 Event didn't occur 1 Event occurred
4	UDE	Unconditional Debug Event 0 Event didn't occur 1 Event occurred
5	IA1	IAC1 Debug Event 0 Event didn't occur 1 Event occurred
6		Reserved
7	DR1	DAC Read Debug Event 0 Event didn't occur 1 Event occurred

Figure 7-2. Debug Status Register (DBSR) (cont.)

8	DW1	DAC Write Debug Event 0 Event didn't occur 1 Event occurred	
9:10		Reserved	
11	IDE	Imprecise Debug Event 0 Event didn't occur 1 Event occurred	
12:21		Reserved	
22:23	MRR	Most Recent Reset 00 No reset has occurred since last cleared by software. 01 Core reset 10 Chip reset 11 System reset	This field is set to a value, indicating the type of reset, when a reset occurs.
24:31		Reserved	

7.6.3 Data Address Compare Register (DAC)

The PPC401A1 can take a debug event upon storage or cache references to an address specified in the DAC register. The address in the DAC is the address of an operand (effective address, or EA) of a storage reference or cache instruction. The fields DBCR[D1R] and DBCR[D1W] control the DAC Read and DAC Write debug events, respectively.

The address in DAC specifies an exact byte EA for a DAC debug event. However, one may want to take a debug event on any byte within a halfword (that is, ignore the least significant bit (LSb) of the DAC); on any byte within a word (that is, ignore the two LSbs of DAC), or on any byte within a line of four words (that is, ignore four LSbs of DAC). The DBCR[D1S] field controls the addressing options.

0	31
---	----

Figure 7-3. Data Address Compare Register (DAC)

0:31		Data Address Compare (DAC) byte address	DBCR[D1S] determines byte, halfword, or word usage.
------	--	---	---

7.6.3.1 Data Address Compare (DAC) Applied to Cache Instructions

Some cache instructions may cause DAC debug events. There are several special cases.

Table 7-2 summarizes possible DAC debug events by cache instruction:

Table 7-2. DAC Applied to Cache Instructions

Instruction	Possible DAC Debug Event	
	DAC-Read	DAC-Write
icbi	Yes	No
icbt	Yes	No
iccci	No	No
icread	No	No
dcba	No	Yes
dcbf	No	Yes
dcbi	No	Yes
dcbst	No	Yes
dcbt	Yes (if cacheable)	No
dcbtst	Yes (if cacheable)	No
dcbz	No	Yes
dccci	No	No
dcread	No	No

Architecturally, the **dcbi** and **dcbz** instructions are “stores.” These instructions can change data, or cause the loss of data by invalidating a dirty line. Therefore, they can cause DAC-Write debug events.

The **dccci** instruction also could be considered a “store” because it can change data by invalidating a dirty line. However, **dccci** is not address-specific; it affects an entire congruence class regardless of the operand address of the instruction. Because it is not address-specific, **dccci** does not cause DAC-Write debug events.

Architecturally, the **dcbt**, **dcbtst**, **dcbf**, and **dcbst** instructions are “loads.” These instructions do not change data. Flushing or storing a cache line from the cache is not architecturally a “store” because a store had already updated the cache; the **dcbf** or **dcbst** instruction only updates the copy in main memory.

The **dcbt** and **dcbtst** instructions can cause DAC-Read debug events independent of cacheability.

Although **dcbf** and **dcbst** are architecturally “loads,” these instructions can create DAC-Write (but not DAC-Read) debug events. In a debug environment, the fact that external memory is being written is the event of interest.

Even though **dcread** and **dccci** are not address-specific (they affect a congruence class regardless of the instruction operand address), and are considered “loads,” on PPC401A1 they do not cause DAC debug events.

All ICU operations (**icbi**, **icbt**, **iccci**, and **icread**) are architecturally treated as “loads.” **icbi** and **icbt** cause DAC debug events. **iccci** and **icread** do not cause DAC debug events on PPC401A1.

7.6.3.2 DAC Applied to String Instructions

An **stswx** instruction with a string length of 0 is a no-op. The **lswx** instruction with the string length equal to 0 does not alter the RT contents with undefined data, as allowed by the PowerPC Architecture. Neither **stswx** nor **lswx** with zero length causes a DAC debug event because storage is not accessed by these instructions.

7.6.4 Instruction Address Compare Register (IAC1)

The PPC401A1 can take a debug event upon an attempt to execute an instruction from an address. The address, which must be word aligned, is defined in the IAC register. The DBCR[IA1] field of the DBCR controls the Instruction Address Compare (IAC) debug event.

0	29	30	31
---	----	----	----

Figure 7-4. Instruction Address Compare Register (IAC1)

0:29		Instruction Address Compare word address	Omit two low-order bits of complete address.
30:31		Reserved	

7.7 Debug Interface

The PPC401A1 processor provides a JTAG interface to support hardware and software test and debug. Typically, the interface connects to a debug port external to the PPC401A1; the debug port is typically connected to a JTAG connector on a processor board.

7.7.1 IEEE 1149.1 Test Access Port (JTAG Debug Port)

The IEEE 1149.1 Test Access Port (TAP), commonly called the JTAG (Joint Test Action Group) debug port, is an architectural standard described in IEEE Std 1149.1–1990, *IEEE Standard Test Access Port and Boundary Scan Architecture*. The standard describes a method for accessing internal chip facilities using a four- or five-signal interface.

The JTAG debug port, originally designed to support scan-based board testing, is enhanced to support the attachment of debug tools. The enhancements, which comply with the IEEE 1149.1 specifications for vendor-specific extensions, are compatible with standard JTAG hardware for boundary-scan system testing.

JTAG Signals	The JTAG debug port implements the four required JTAG signals: TCLK, TMS, TDI, and TDO the optional TRST signal.
JTAG Clock Requirements	The frequency of the TCLK signal can range from DC to one-half of the internal chip clock frequency.
JTAG Reset Requirements	The JTAG debug port logic is reset at the same time as a system reset. the JTAG TAP controller returns to the Test-Logic Reset state.

7.7.1.1 JTAG Connector

A 16-pin male 2x8 header connector is suggested as the JTAG debug port connector for chips that incorporate PPC401A1. This connector definition matches the requirements of the RISCWatch debugger from IBM. The connector is shown in Figure 7-5 and the signals are shown in Table 7-3 on p. 7-12. The connector should be placed as close as possible to the chip containing the PPC401A1 to ensure signal integrity.

Note that position 14 does not contain a pin.

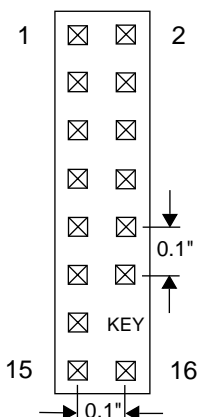


Figure 7-5. JTAG Connector (top view) Physical Layout

Table 7-3. JTAG Connector Signals

Pin	I/O	Signal	Description
1	O	TDO	JTAG Test Data Out
2		No connect (NC)	Reserved
3	I	TDI ¹	JTAG Test Data In
4		$\overline{\text{TRST}}$	JTAG Reset
5		NC	Reserved
6		+POWER ²	Processor Power OK
7	I	TCK ³	JTAG Test Clock
8		NC	Reserved
9	I	TMS ¹	JTAG Test Mode Select
10		NC	Reserved
11	I	HALT ³	Processor Halt
12		NC	Reserved
13		NC	Reserved
14		Key	The pin at this position should be removed.
15		NC	Reserved
16		GND	Ground

1. A 10K ohm pullup resistor should be connected to this signal to reduce chip power consumption. The pullup resistor is not required.

2. The +POWER signal, sourced from the target development board, indicates whether the processor is operating. This signal does not supply power to the RISCWatch hardware or to the processor. The active level on this signal can be +5V or +3.3V (note that the PPC401A1 may have either +5V or +3.3V I/O, but the processor itself must be powered by +3.3V). A series resistor (1K ohm or less) should be used to provide short circuit current-limiting protection.
3. A 10K ohm pullup resistor must be connected to these signals to ensure proper chip operation when these inputs are not used.

7.7.1.2 JTAG Instructions

The JTAG debug port provides the standard **extest**, **sample/preload**, and **bypass** instructions. Invalid instructions behave as the **bypass** instruction. There are four private instructions.

Table 7-4. JTAG Instructions

Instruction	Code	Comments
Extest	0000	IEEE 1149.1 standard.
Sample/Preload	0001	IEEE 1149.1 standard.
JTAG 5	0101	Private
JTAG 7	0111	Private
JTAG B	1011	Private
Bypass	1111	IEEE 1149.1 standard.

7.7.1.3 JTAG Boundary Scan

Boundary Scan Description Language (BSDL), IEEE 1149.1b-1994, is a supplement to IEEE 1149.1-1990 and IEEE 1149.1a-1993 *Standard Test Access Port and Boundary-Scan Architecture*. BSDL, a subset of the IEEE 1076-1993 Standard VHSIC Hardware Description Language (VHDL), allows a rigorous description of testability features in components which comply with the standard. It is used by automated test pattern generation tools for package interconnect tests and by electronic design automation (EDA) tools for synthesized test logic and verification. BSDL supports robust extensions that can be used for internal test generation and to write software for hardware debug and diagnostics.

The primary components of BSDL include the logical port description, the physical pin map, the instruction set, and the boundary register description.

The logical port description assigns symbolic names to the pins of a chip. Each pin has a logical type of in, out, inout, buffer, or linkage that defines the logical direction of signal flow.

The physical pin map correlates the logical ports of the chip to the physical pins of a specific package. A BSDL description can have several physical pin maps; each map is given a unique name.

Instruction set statements describe the bit patterns that must be shifted into the Instruction Register to place the chip in the various test modes defined by the standard. Instruction set statements also support descriptions of instructions that are unique to the chip.

The boundary register description lists each cell or shift stage of the Boundary Register. Each cell has a unique number; the cell numbered 0 is the closest to the Test Data Out (TDO) pin and the cell with the highest number is closest to the Test Data In (TDI) pin. Each cell contains additional information, including: cell type, logical port associated with the cell, logical function of the cell, safe value, control cell number, disable value, and result value.

The PPC401A1 has a 4 gigabyte (GB) address space, which is presented as a flat address space.

The PPC401A1 does not support address translation.

8.1 Storage Attributes

The PowerPC Architecture defines storage attributes to control several aspects of storage behavior. The attributes supported by the PPC401A1 are: the W attribute, which controls write-through policy; the I attribute, which controls cacheability; the M attribute, which controls memory coherency; the G attribute, which controls guarding against speculative accesses; and the E attribute, which controls byte ordering (endianness). Note that the E attribute is not defined in the PowerPC Architecture; this attribute is defined in the IBM PowerPC Embedded Environment.

The PPC401A1 provides several SPRs to control storage attributes. These registers, called storage attribute control registers, are:

- DCWR
- DCCR
- ICCR
- SGR
- SLER

These registers, which are all privileged, are read from and written to using the **mfspr** and **mtspr** instructions, respectively.

Each storage attribute control register contains 32 bits; each bit controls one of 32 storage attribute regions. Each region, defined by address bits $A_{0:4}$, contains 128MB. These registers divide the 4GB real memory address space into thirty-two 128MB sections such that bit 0 is associated with the lowest 128MB region, bit 1 the next-lowest 128MB region, and so on.

The storage attributes in each storage attribute control region are set independently.

Figure 8-1 through Figure 8-5 provide bit descriptions for the SAC registers.

8.1.1 Data Cache Write-through Register (DCWR)

The DCWR controls write-through policy (the W storage attribute) for the data cache unit (DCU). Write-through is not applicable to the instruction cache unit (ICU).

The PowerPC Architecture does not support memory models in which write-through is enabled and caching is inhibited.

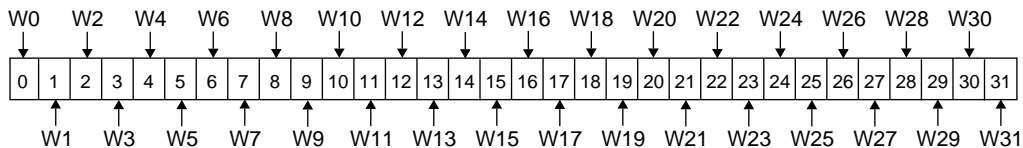


Figure 8-1. Data Cache Write-through Register (DCWR)

0	W0	0 Write-back 1 Write-through	0x0000 0000–0x07FF FFFF
1	W1	0 Write-back 1 Write-through	0x0800 0000–0x0FFF FFFF
2	W2	0 Write-back 1 Write-through	0x1000 0000–0x17FF FFFF
3	W3	0 Write-back 1 Write-through	0x1800 0000–0x1FFF FFFF
4	W4	0 Write-back 1 Write-through	0x2000 0000–0x27FF FFFF
5	W5	0 Write-back 1 Write-through	0x2800 0000–0x2FFF FFFF
6	W6	0 Write-back 1 Write-through	0x3000 0000–0x37FF FFFF
7	W7	0 Write-back 1 Write-through	0x3800 0000–0x3FFF FFFF
8	W8	0 Write-back 1 Write-through	0x4000 0000–0x47FF FFFF
9	W9	0 Write-back 1 Write-through	0x4800 0000–0x4FFF FFFF
10	W10	0 Write-back 1 Write-through	0x5000 0000–0x57FF FFFF
11	W11	0 Write-back 1 Write-through	0x5800 0000–0x5FFF FFFF
12	W12	0 Write-back 1 Write-through	0x6000 0000–0x67FF FFFF

Figure 8-1. Data Cache Write-through Register (DCWR)

13	W13	0 Write-back 1 Write-through	0x6800 0000–0x6FFF FFFF
14	W14	0 Write-back 1 Write-through	0x7000 0000–0x77FF FFFF
15	W15	0 Write-back 1 Write-through	0x7800 0000–0x7FFF FFFF
16	W16	0 Write-back 1 Write-through	0x8000 0000–0x87FF FFFF
17	W17	0 Write-back 1 Write-through	0x8800 0000–0x8FFF FFFF
18	W18	0 Write-back 1 Write-through	0x9000 0000–0x97FF FFFF
19	W19	0 Write-back 1 Write-through	0x9800 0000–0x9FFF FFFF
20	W20	0 Write-back 1 Write-through	0xA000 0000–0xA7FF FFFF
21	W21	0 Write-back 1 Write-through	0xA800 0000–0xAFFF FFFF
22	W22	0 Write-back 1 Write-through	0xB000 0000–0xB7FF FFFF
23	W23	0 Write-back 1 Write-through	0xB800 0000–0xBFFF FFFF
24	W24	0 Write-back 1 Write-through	0xC000 0000–0xC7FF FFFF
25	W25	0 Write-back 1 Write-through	0xC800 0000–0xCFFF FFFF
26	W26	0 Write-back 1 Write-through	0xD000 0000–0xD7FF FFFF
27	W27	0 Write-back 1 Write-through	0xD800 0000–0xDFFF FFFF
28	W28	0 Write-back 1 Write-through	0xE000 0000–0xE7FF FFFF
29	W29	0 Write-back 1 Write-through	0xE800 0000–0xEFFF FFFF
30	W30	0 Write-back 1 Write-through	0xF000 0000–0xF7FF FFFF
31	W31	0 Write-back 1 Write-through	0xF800 0000–0xFFFF FFFF

8.1.2 Data Cache Cacheability Register (DCCR)

The DCCR controls the cacheability (the I storage attribute) of data accesses. Note that the polarity of this bit is opposite that of the I attribute as defined in the PowerPC Architecture.

Following any reset, all DCCR bits are set to 0. No memory regions are cacheable. Before memory regions can be designated as cacheable in the DCCR, it is necessary to execute the **dccci** instruction 32 times, once for each congruence class in the DCU cache array. This procedure invalidates all 32 congruence classes. The DCCR can then be reconfigured and the DCU can begin normal operation.

The PowerPC Architecture does not support memory models in which write-through is enabled and caching is inhibited.

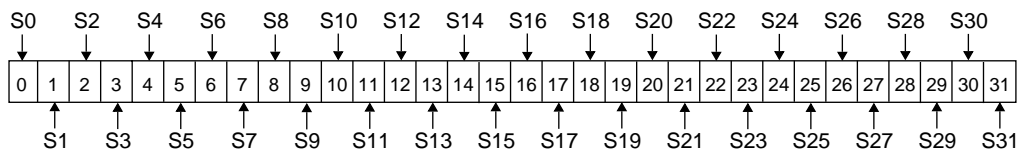


Figure 8-2. Data Cache Cacheability Register (DCCR)

0	S0	0 Noncacheable 1 Cacheable	0x0000 0000–0x07FF FFFF
1	S1	0 Noncacheable 1 Cacheable	0x0800 0000–0x0FFF FFFF
2	S2	0 Noncacheable 1 Cacheable	0x1000 0000–0x17FF FFFF
3	S3	0 Noncacheable 1 Cacheable	0x1800 0000–0x1FFF FFFF
4	S4	0 Noncacheable 1 Cacheable	0x2000 0000–0x27FF FFFF
5	S5	0 Noncacheable 1 Cacheable	0x2800 0000–0x2FFF FFFF
6	S6	0 Noncacheable 1 Cacheable	0x3000 0000–0x37FF FFFF
7	S7	0 Noncacheable 1 Cacheable	0x3800 0000–0x3FFF FFFF
8	S8	0 Noncacheable 1 Cacheable	0x4000 0000–0x47FF FFFF
9	S9	0 Noncacheable 1 Cacheable	0x4800 0000–0x4FFF FFFF

Figure 8-2. Data Cache Cacheability Register (DCCR) (cont.)

10	S10	0 Noncacheable 1 Cacheable	0x5000 0000–0x57FF FFFF
11	S11	0 Noncacheable 1 Cacheable	0x5800 0000–0x5FFF FFFF
12	S12	0 Noncacheable 1 Cacheable	0x6000 0000–0x67FF FFFF
13	S13	0 Noncacheable 1 Cacheable	0x6800 0000–0x6FFF FFFF
14	S14	0 Noncacheable 1 Cacheable	0x7000 0000–0x77FF FFFF
15	S15	0 Noncacheable 1 Cacheable	0x7800 0000–0x7FFF FFFF
16	S16	0 Noncacheable 1 Cacheable	0x8000 0000–0x87FF FFFF
17	S17	0 Noncacheable 1 Cacheable	0x8800 0000–0x8FFF FFFF
18	S18	0 Noncacheable 1 Cacheable	0x9000 0000–0x97FF FFFF
19	S19	0 Noncacheable 1 Cacheable	0x9800 0000–0x9FFF FFFF
20	S20	0 Noncacheable 1 Cacheable	0xA000 0000–0xA7FF FFFF
21	S21	0 Noncacheable 1 Cacheable	0xA800 0000–0xAFFF FFFF
22	S22	0 Noncacheable 1 Cacheable	0xB000 0000–0xB7FF FFFF
23	S23	0 Noncacheable 1 Cacheable	0xB800 0000–0xBFFF FFFF
24	S24	0 Noncacheable 1 Cacheable	0xC000 0000–0xC7FF FFFF
25	S25	0 Noncacheable 1 Cacheable	0xC800 0000–0xCFFF FFFF
26	S26	0 Noncacheable 1 Cacheable	0xD000 0000–0xD7FF FFFF
27	S27	0 Noncacheable 1 Cacheable	0xD800 0000–0xDFFF FFFF
28	S28	0 Noncacheable 1 Cacheable	0xE000 0000–0xE7FF FFFF

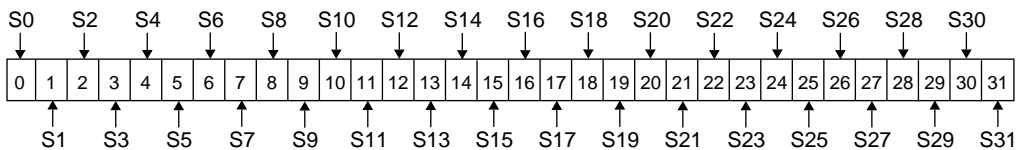
Figure 8-2. Data Cache Cacheability Register (DCCR) (cont.)

29	S29	0 Noncacheable 1 Cacheable	0xE800 0000–0xEFFF FFFF
30	S30	0 Noncacheable 1 Cacheable	0xF000 0000–0xF7FF FFFF
31	S31	0 Noncacheable 1 Cacheable	0xF800 0000–0xFFFF FFFF

8.1.3 Instruction Cache Cacheability Register (ICCR)

The ICCR controls the cacheability (the I storage attribute) of instruction fetches. Note that the polarity of this bit is opposite that of the I attribute as defined in the PowerPC Architecture.

Following any reset, all ICCR bits are set to 0. No memory regions are cacheable. Before memory regions can be designated as cacheable in the ICCR, it is necessary to execute the **iccci** instruction 64 times, once for each congruence class in the ICU cache array. This procedure invalidates all 64 congruence classes. The ICCR can then be reconfigured and the ICU can begin normal operation.

**Figure 8-3. Instruction Cache Cacheability Register (ICCR)**

0	S0	0 Noncacheable 1 Cacheable	0x0000 0000–0x07FF FFFF
1	S1	0 Noncacheable 1 Cacheable	0x0800 0000–0x0FFF FFFF
2	S2	0 Noncacheable 1 Cacheable	0x1000 0000–0x17FF FFFF
3	S3	0 Noncacheable 1 Cacheable	0x1800 0000–0x1FFF FFFF
4	S4	0 Noncacheable 1 Cacheable	0x2000 0000–0x27FF FFFF
5	S5	0 Noncacheable 1 Cacheable	0x2800 0000–0x2FFF FFFF

Figure 8-3. Instruction Cache Cacheability Register (ICCR) (cont.)

6	S6	0 Noncacheable 1 Cacheable	0x3000 0000–0x37FF FFFF
7	S7	0 Noncacheable 1 Cacheable	0x3800 0000–0x3FFF FFFF
8	S8	0 Noncacheable 1 Cacheable	0x4000 0000–0x47FF FFFF
9	S9	0 Noncacheable 1 Cacheable	0x4800 0000–0x4FFF FFFF
10	S10	0 Noncacheable 1 Cacheable	0x5000 0000–0x57FF FFFF
11	S11	0 Noncacheable 1 Cacheable	0x5800 0000–0x5FFF FFFF
12	S12	0 Noncacheable 1 Cacheable	0x6000 0000–0x67FF FFFF
13	S13	0 Noncacheable 1 Cacheable	0x6800 0000–0x6FFF FFFF
14	S14	0 Noncacheable 1 Cacheable	0x7000 0000–0x77FF FFFF
15	S15	0 Noncacheable 1 Cacheable	0x7800 0000–0x7FFF FFFF
16	S16	0 Noncacheable 1 Cacheable	0x8000 0000–0x87FF FFFF
17	S17	0 Noncacheable 1 Cacheable	0x8800 0000–0x8FFF FFFF
18	S18	0 Noncacheable 1 Cacheable	0x9000 0000–0x97FF FFFF
19	S19	0 Noncacheable 1 Cacheable	0x9800 0000–0x9FFF FFFF
20	S20	0 Noncacheable 1 Cacheable	0xA000 0000–0xA7FF FFFF
21	S21	0 Noncacheable 1 Cacheable	0xA800 0000–0xAFFF FFFF
22	S22	0 Noncacheable 1 Cacheable	0xB000 0000–0xB7FF FFFF
23	S23	0 Noncacheable 1 Cacheable	0xB800 0000–0xBFFF FFFF
24	S24	0 Noncacheable 1 Cacheable	0xC000 0000–0xC7FF FFFF

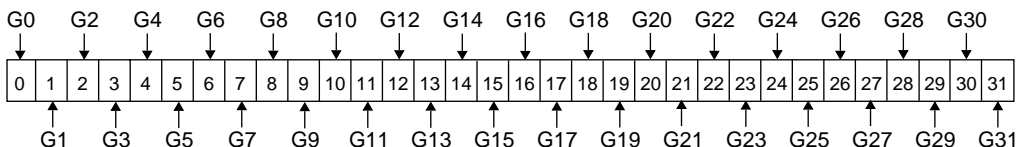
Figure 8-3. Instruction Cache Cacheability Register (ICCR) (cont.)

25	S25	0 Noncacheable 1 Cacheable	0xC800 0000–0xCFFF FFFF
26	S26	0 Noncacheable 1 Cacheable	0xD000 0000–0xD7FF FFFF
27	S27	0 Noncacheable 1 Cacheable	0xD800 0000–0xDFFF FFFF
28	S28	0 Noncacheable 1 Cacheable	0xE000 0000–0xE7FF FFFF
29	S29	0 Noncacheable 1 Cacheable	0xE800 0000–0xEFFF FFFF
30	S30	0 Noncacheable 1 Cacheable	0xF000 0000–0xF7FF FFFF
31	S31	0 Noncacheable 1 Cacheable	0xF800 0000–0xFFFF FFFF

8.1.4 Storage Guarded Register (SGR)

The SGR controls the guarded (G) storage attribute. This attribute can disable speculative accesses to regions of storage, such as memory-mapped I/O devices, from which instructions will never be fetched. This attribute does not affect data accesses; the PPC401A1 does not perform speculative loads or stores. For instruction fetches, a memory access to a guarded region does not occur until the execution pipeline is empty, guaranteeing that the access is necessary and therefore not speculative. For this reason, performance is degraded when executing out of guarded regions, and software should avoid unnecessarily marking regions of instruction storage as guarded.

After any reset, the SGR is set to 0xFFFF FFFF, marking all of storage as guarded. For best performance, system software should clear the guarded attribute of appropriate regions as soon as possible.

**Figure 8-4. Storage Guarded Register (SGR)**

0	G0	0 Normal 1 Guarded	0x0000 0000–0x07FF FFFF
---	----	-----------------------	-------------------------

Figure 8-4. Storage Guarded Register (SGR) (cont.)

1	G1	0 Normal 1 Guarded	0x0800 0000–0x0FFF FFFF
2	G2	0 Normal 1 Guarded	0x1000 0000–0x17FF FFFF
3	G3	0 Normal 1 Guarded	0x1800 0000–0x1FFF FFFF
4	G4	0 Normal 1 Guarded	0x2000 0000–0x27FF FFFF
5	G5	0 Normal 1 Guarded	0x2800 0000–0x2FFF FFFF
6	G6	0 Normal 1 Guarded	0x3000 0000–0x37FF FFFF
7	G7	0 Normal 1 Guarded	0x3800 0000–0x3FFF FFFF
8	G8	0 Normal 1 Guarded	0x4000 0000–0x47FF FFFF
9	G9	0 Normal 1 Guarded	0x4800 0000–0x4FFF FFFF
10	G10	0 Normal 1 Guarded	0x5000 0000–0x57FF FFFF
11	G11	0 Normal 1 Guarded	0x5800 0000–0x5FFF FFFF
12	G12	0 Normal 1 Guarded	0x6000 0000–0x67FF FFFF
13	G13	0 Normal 1 Guarded	0x6800 0000–0x6FFF FFFF
14	G14	0 Normal 1 Guarded	0x7000 0000–0x77FF FFFF
15	G15	0 Normal 1 Guarded	0x7800 0000–0x7FFF FFFF
16	G16	0 Normal 1 Guarded	0x8000 0000–0x87FF FFFF
17	G17	0 Normal 1 Guarded	0x8800 0000–0x8FFF FFFF
18	G18	0 Normal 1 Guarded	0x9000 0000–0x97FF FFFF
19	G19	0 Normal 1 Guarded	0x9800 0000–0x9FFF FFFF

Figure 8-4. Storage Guarded Register (SGR) (cont.)

20	G20	0 Normal 1 Guarded	0xA000 0000–0xA7FF FFFF
21	G21	0 Normal 1 Guarded	0xA800 0000–0xAFFF FFFF
22	G22	0 Normal 1 Guarded	0xB000 0000–0xB7FF FFFF
23	G23	0 Normal 1 Guarded	0xB800 0000–0xBFFF FFFF
24	G24	0 Normal 1 Guarded	0xC000 0000–0xC7FF FFFF
25	G25	0 Normal 1 Guarded	0xC800 0000–0xCFFF FFFF
26	G26	0 Normal 1 Guarded	0xD000 0000–0xD7FF FFFF
27	G27	0 Normal 1 Guarded	0xD800 0000–0xDFFF FFFF
28	G28	0 Normal 1 Guarded	0xE000 0000–0xE7FF FFFF
29	G29	0 Normal 1 Guarded	0xE800 0000–0xEFFF FFFF
30	G30	0 Normal 1 Guarded	0xF000 0000–0xF7FF FFFF
31	G31	0 Normal 1 Guarded	0xF800 0000–0xFFFF FFFF

8.1.5 Storage Little-Endian Register (SLER)

The SLER controls the endian (E) storage attribute. This attribute determines the byte ordering of storage. Section 2.4, “Byte Ordering,” on p. 2-17, provides a detailed description of byte ordering in the IBM PowerPC Embedded Environment.

After any reset, all SLER bits are set to 0 (big endian).

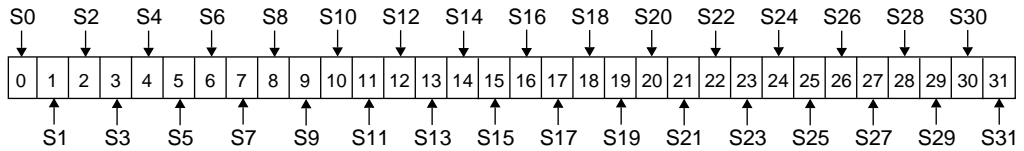


Figure 8-5. Storage Little-Endian Register (SLER)

0	S0	0 Big endian or PowerPC little endian 1 True little endian	0x0000 0000–0x07FF FFFF
1	S1	0 Big endian or PowerPC little endian 1 True little endian	0x0800 0000–0x0FFF FFFF
2	S2	0 Big endian or PowerPC little endian 1 True little endian	0x1000 0000–0x17FF FFFF
3	S3	0 Big endian or PowerPC little endian 1 True little endian	0x1800 0000–0x1FFF FFFF
4	S4	0 Big endian or PowerPC little endian 1 True little endian	0x2000 0000–0x27FF FFFF
5	S5	0 Big endian or PowerPC little endian 1 True little endian	0x2800 0000–0x2FFF FFFF
6	S6	0 Big endian or PowerPC little endian 1 True little endian	0x3000 0000–0x37FF FFFF
7	S7	0 Big endian or PowerPC little endian 1 True little endian	0x3800 0000–0x3FFF FFFF
8	S8	0 Big endian or PowerPC little endian 1 True little endian	0x4000 0000–0x47FF FFFF
9	S9	0 Big endian or PowerPC little endian 1 True little endian	0x4800 0000–0x4FFF FFFF
10	S10	0 Big endian or PowerPC little endian 1 True little endian	0x5000 0000–0x57FF FFFF
11	S11	0 Big endian or PowerPC little endian 1 True little endian	0x5800 0000–0x5FFF FFFF
12	S12	0 Big endian or PowerPC little endian 1 True little endian	0x6000 0000–0x67FF FFFF
13	S13	0 Big endian or PowerPC little endian 1 True little endian	0x6800 0000–0x6FFF FFFF
14	S14	0 Big endian or PowerPC little endian 1 True little endian	0x7000 0000–0x77FF FFFF

Figure 8-5. Storage Little-Endian Register (SLER) (cont.)

15	S15	0 Big endian or PowerPC little endian 1 True little endian	0x7800 0000–0x7FFF FFFF
16	S16	0 Big endian or PowerPC little endian 1 True little endian	0x8000 0000–0x87FF FFFF
17	S17	0 Big endian or PowerPC little endian 1 True little endian	0x8800 0000–0x8FFF FFFF
18	S18	0 Big endian or PowerPC little endian 1 True little endian	0x9000 0000–0x97FF FFFF
19	S19	0 Big endian or PowerPC little endian 1 True little endian	0x9800 0000–0x9FFF FFFF
20	S20	0 Big endian or PowerPC little endian 1 True little endian	0xA000 0000–0xA7FF FFFF
21	S21	0 Big endian or PowerPC little endian 1 True little endian	0xA800 0000–0xAFFF FFFF
22	S22	0 Big endian or PowerPC little endian 1 True little endian	0xB000 0000–0xB7FF FFFF
23	S23	0 Big endian or PowerPC little endian 1 True little endian	0xB800 0000–0xBFFF FFFF
24	S24	0 Big endian or PowerPC little endian 1 True little endian	0xC000 0000–0xC7FF FFFF
25	S25	0 Big endian or PowerPC little endian 1 True little endian	0xC800 0000–0xCFFF FFFF
26	S26	0 Big endian or PowerPC little endian 1 True little endian	0xD000 0000–0xD7FF FFFF
27	S27	0 Big endian or PowerPC little endian 1 True little endian	0xD800 0000–0xDFFF FFFF
28	S28	0 Big endian or PowerPC little endian 1 True little endian	0xE000 0000–0xE7FF FFFF
29	S29	0 Big endian or PowerPC little endian 1 True little endian	0xE800 0000–0xEFFF FFFF
30	S30	0 Big endian or PowerPC little indian 1 True little endian	0xF000 0000–0xF7FF FFFF
31	S31	0 Big endian or PowerPC little endian 1 True little endian	0xF800 0000–0xFFFF FFFF

9

Instruction Set

Descriptions of the PPC401A1 instructions follow. Each description contains the following elements:

- Instruction names (mnemonic and full)
- Instruction syntax
- Instruction format diagram
- Pseudocode description
- Prose description
- Registers altered
- Architecture notes identifying the associated PowerPC Architecture component

Where appropriate, instruction descriptions list invalid instruction forms and provide programming notes.

9.1 Instruction Set Portability

To support embedded real-time applications, the instruction sets of the PPC401A1 and other IBM PowerPC 400Series embedded controllers implement the IBM PowerPC Embedded Environment, which is not part of the PowerPC Architecture defined in *The PowerPC Architecture: A Specification for a New Family of RISC Processors*.

Programs using these instructions are not portable to PowerPC implementations that do not implement the IBM PowerPC Embedded Environment. Table 9-1 lists instructions in the IBM PowerPC Embedded Environment that are implemented in the PPC401A1.

Table 9-1. Instructions in the IBM PowerPC Embedded Environment

dccci	mfdcr
dcread	mtdcr
iccci	rfci
icbt	wrtree
icread	wrtreei

9.2 Instruction Formats

For more detailed information about instruction formats, including a summary of instruction field usage and instruction format diagrams for the PPC401A1, see Section A.3, “Instruction Formats,” on p. A-46.

Instructions are four bytes long. Instruction addresses are always word-aligned.

Instruction bits 0 through 5 always contain the primary opcode. Many instructions have an extended opcode in another field. The remaining instruction bits contain additional fields. All instruction fields belong to one of the following categories:

- **Defined**

These instructions contain values, such as opcodes, that cannot be altered. The instruction format diagrams specify the values of defined fields.

- **Variable**

These fields contain operands, such as general purpose register selectors and immediate values, that may vary from execution to execution. The instruction format diagrams specify the operands in variable fields.

- **Reserved**

Bits in a reserved field should be set to 0. In the instruction format diagrams, reserved fields are shaded.

If any bit in a defined field does not contain the expected value, the instruction is illegal and an illegal instruction exception occurs. If any bit in a reserved field does not contain 0, the instruction form is invalid and its result is architecturally undefined. Unless otherwise noted, the PPC401A1 executes all invalid instruction forms without causing an illegal instruction exception.

9.3 Pseudocode

The pseudocode that appears in the instruction descriptions provides a semi-formal language for describing instruction operations.

The pseudocode uses the following notation:

←	Assignment
^	AND logical operator
¬	NOT logical operator
∨	OR logical operator
⊕	Exclusive-OR (XOR) logical operator

+	Twos complement addition
−	Twos complement subtraction, unary minus
×	Multiplication
÷	Division yielding a quotient
%	Remainder of an integer division; $(33 \% 32) = 1$.
	Concatenation
=, ≠	Equal, not equal relations
<, >	Signed comparison relations
\leq , \geq	Unsigned comparison relations
if...then...else...	Conditional execution; if <i>condition</i> then <i>a</i> else <i>b</i> , where <i>a</i> and <i>b</i> represent one or more pseudocode statements. Indenting indicates the ranges of <i>a</i> and <i>b</i> . If <i>b</i> is null, the else does not appear.
do	Do loop. “to” and “by” clauses specify incrementing an iteration variable; “while” and “until” clauses specify terminating conditions. Indenting indicates the range of the loop.
leave	Leave innermost do loop or do loop specified in a leave statement.
n	A decimal number
0xn	A hexadecimal number
0bn	A binary number
FLD	An instruction field
FLD _b	A bit in a named instruction field
FLD _{b:b}	A range of bits in a named instruction field
FLD _{b,b, ...}	A list of bits, by number or name, in a named instruction field
REG _b	A bit in a named register
REG _{b:b}	A range of bits in a named register
REG _{b,b, ...}	A list of bits, by number or name, in a named register
REG[FLD]	A field in a named register
REG[FLD, FLD ...]	A list of fields in a named register
REG[FLD:FLD]	A range of fields in a named register
GPR(r)	General Purpose Register (GPR) <i>r</i> , where $0 \leq r \leq 31$.
(GPR(r))	The contents of GPR <i>r</i> , where $0 \leq r \leq 31$.

DCR(DCRN)	A Device Control Register (DCR) specified by the DCRF field in an mfocr or mtocr instruction
SPR(SPRN)	An SPR specified by the SPRF field in an mfspir or mtspir instruction
RA, RB, ...	GPRs
(Rx)	The contents of a GPR, where <i>x</i> is A, B, S, or T
(RA 0)	The contents of the register RA or 0, if the RA field is 0.
$c_{0:3}$	A four-bit object used to store condition results in compare instructions.
n_b	The bit or bit value <i>b</i> is replicated <i>n</i> times.
xx	Bit positions which are don't-cares.
CEIL(<i>x</i>)	Least integer $\geq x$.
EXTS(<i>x</i>)	The result of extending <i>x</i> on the left with sign bits.
PC	Program counter.
RESERVE	Reserve bit; indicates whether a process has reserved a block of storage.
CIA	Current instruction address; the 32-bit address of the instruction being described by a sequence of pseudocode. This address is used to set the next instruction address (NIA). Does not correspond to any architected register.
NIA	Next instruction address; the 32-bit address of the next instruction to be executed. In pseudocode, a successful branch is indicated by assigning a value to NIA. For instructions that do not branch, the NIA is CIA +4.
MS(addr, <i>n</i>)	The number of bytes represented by <i>n</i> at the location in main storage represented by <i>addr</i> .
EA	Effective address; the 32-bit address, derived by applying indexing or indirect addressing rules to the specified operand, that specifies an location in main storage.
ROTL((RS), <i>n</i>)	Rotate left; the contents of RS are shifted left the number of bits specified by <i>n</i> .
MASK(MB,ME)	Mask having 1s in positions MB through ME (wrapping if MB > ME) and 0's elsewhere.
instruction(EA)	An instruction operating on a data or instruction cache block associated with an EA.

The following table lists the pseudocode operators and their associativity in descending order of precedence:

Table 9-2. Operator Precedence

Operators	Associativity
REG _b , REG[FLD], function evaluation	Left to right
ⁿ b	Right to left
¬, − (unary minus)	Right to left
×, ÷	Left to right
+, −	Left to right
	Left to right
=, ≠, <, >, ^u <, ^u >	Left to right
∧, ⊕	Left to right
∨	Left to right
←	None

9.4 Register Usage

Each instruction description lists the registers altered by the instruction. Some register changes are explicitly detailed in the instruction description (for example, the target register of a load instruction). Other registers are changed, with the details of the change not included in the instruction description. This category frequently includes the Condition Register (CR) and the Fixed-point Exception Register (XER). For discussion of CR, see Section 2.2.3, “Condition Register (CR),” on p. 2-9. For discussion of XER, see Section 2.2.2.3, “Fixed Point Exception Register (XER),” on p. 2-6.

9.5 Alphabetical Instruction Listing

The following pages list the instructions available in the PPC401A1 in alphabetical order.

add

Add

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

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

$(RT) \leftarrow (RA) + (RB)$

The sum of the contents of register RA and the contents of register RB is placed into register RT.

Registers Altered

- RT
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

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

31	RT	RA	RB	OE	10	Rc
0	6	11	16	21	22	31

```

(RT) ← (RA) + (RB)
if (RA) + (RB)  $\geq 2^{32} - 1$  then
    XER[CA] ← 1
else
    XER[CA] ← 0

```

The sum of the contents of register RA and register RB is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the add operation.

Registers Altered

- RT
- XER[CA]
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

adde

Add Extended

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

31	RT	RA	RB	OE	138	Rc
0	6	11	16	21 22		31

$(RT) \leftarrow (RA) + (RB) + XER[CA]$
if $(RA) + (RB) + XER[CA] \geq 2^{32} - 1$ then
 $XER[CA] \leftarrow 1$
else
 $XER[CA] \leftarrow 0$

The sum of the contents of register RA, register RB, and XER[CA] is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the add operation.

Registers Altered

- RT
- XER[CA]
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

addi RT, RA, IM

14	RT	RA	IM
0	6	11	16
			31

$$(RT) \leftarrow (RA|0) + \text{EXTS}(IM)$$

If the RA field is 0, the IM field, sign-extended to 32 bits, is placed into register RT.

If the RA field is nonzero, the sum of the contents of register RA and the contents of the IM field, sign-extended to 32 bits, is placed into register RT.

Registers Altered

- RT

Programming Note

To place an immediate, sign-extended value into the GPR specified by the RT field, set the RA field to 0.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-3. Extended Mnemonics for addi

Mnemonic	Operands	Function	Other Registers Changed
lal	RT, D(RA)	Load address ($RA \neq 0$); D is an offset from a base address that is assumed to be (RA). $(RT) \leftarrow (RA) + \text{EXTS}(D)$ <i>Extended mnemonic for addi RT,RA,D</i>	
li	RT, IM	Load immediate. $(RT) \leftarrow \text{EXTS}(IM)$ <i>Extended mnemonic for addi RT,0,IM</i>	
subi	RT, RA, IM	Subtract $\text{EXTS}(IM)$ from $(RA 0)$. Place result in RT. <i>Extended mnemonic for addi RT,RA,-IM</i>	

addic

Add Immediate Carrying

addic RT, RA, IM

12	RT	RA	IM
0	6	11	16
			31

```
(RT) ← (RA) + EXTS(IM)
if (RA) + EXTS(IM)  $\geq 2^{32} - 1$  then
    XER[CA] ← 1
else
    XER[CA] ← 0
```

The sum of the contents of register RA and the contents of the IM field, sign-extended to 32 bits, is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the add operation.

Registers Altered

- RT
- XER[CA]

Architecture Note

9

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-4. Extended Mnemonics for addic

Mnemonic	Operands	Function	Other Registers Changed
subic	RT, RA, IM	Subtract EXTS(IM) from (RA) Place result in RT; place carry-out in XER[CA]. <i>Extended mnemonic for addic RT,RA,-IM</i>	

addic. RT, RA, IM

13	RT	RA	IM
0	6	11	16
			31

```

(RT) ← (RA) + EXT(S)(IM)
if (RA) + EXT(S)(IM)  $\geq 2^{32} - 1$  then
    XER[CA] ← 1
else
    XER[CA] ← 0

```

The sum of the contents of register RA and the contents of the IM field, sign-extended to 32 bits, is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the add operation.

Registers Altered

- RT
- XER[CA]
- CR[CR0]_{LT, GT, EQ, SO}

Programming Note

addic. is one of three instructions that implicitly update CR[CR0] without having an RC field. The other instructions are **andi.** and **andis.**

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

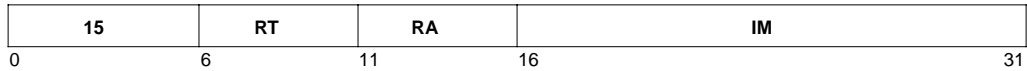
Table 9-5. Extended Mnemonics for addic.

Mnemonic	Operands	Function	Other Registers Changed
subic.	RT, RA, IM	Subtract EXT(S)(IM) from (RA). Place result in RT; place carry-out in XER[CA]. <i>Extended mnemonic for addic. RT,RA,-IM</i>	CR[CR0]

addis

Add Immediate Shifted

addis RT, RA, IM



$$(RT) \leftarrow (RA|0) + (IM \parallel ^{16}0)$$

If the RA field is 0, the IM field is concatenated on its right with sixteen 0-bits and placed into register RT.

If the RA field is nonzero, the contents of register RA are added to the contents of the extended IM field. The sum is stored into register RT.

Registers Altered

- RT

Programming Note

An **addi** instruction stores a sign-extended 16-bit value in a GPR. An **addis** instruction followed by an **ori** instruction stores an arbitrary 32-bit value in a GPR, as shown in the following example:

addis RT, 0, high 16 bits of value
ori RT, RT, low 16 bits of value

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-6. Extended Mnemonics for addis

Mnemonic	Operands	Function	Other Registers Changed
lis	RT, IM	Load immediate shifted. (RT) ← (IM ¹⁶ 0) <i>Extended mnemonic for addis RT,0,IM</i>	
subis	RT, RA, IM	Subtract (IM ¹⁶ 0) from (RA 0). Place result in RT. <i>Extended mnemonic for addis RT,RA,-IM</i>	

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

31	RT	RA		OE	234	Rc
0	6	11	16	21 22		31

```

(RT) ← (RA) + XER[CA] + (−1)
if (RA) + XER[CA] + 0xFFFF FFFF  $\geq$   $2^{32} - 1$  then
    XER[CA] ← 1
else
    XER[CA] ← 0

```

The sum of the contents of register RA, XER[CA], and −1 is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the add operation.

Registers Altered

- RT
- XER[CA]
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1
- XER[SO, OV] if OE contains 1

Invalid Instruction Forms

- Reserved fields

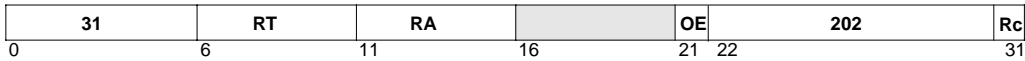
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

addze

Add to Zero Extended

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



```
(RT) ← (RA) + XER[CA]
if (RA) + XER[CA] > 232 - 1 then
    XER[CA] ← 1
else
    XER[CA] ← 0
```

The sum of the contents of register RA and XER[CA] is placed into register RT.
XER[CA] is set to a value determined by the unsigned magnitude of the result of the add operation.

Registers Altered

- RT
- XER[CA]
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1
- XER[SO, OV] if OE contains 1

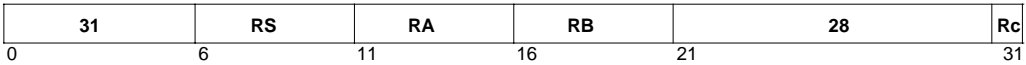
Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

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



$(RA) \leftarrow (RS) \wedge (RB)$

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

Registers Altered

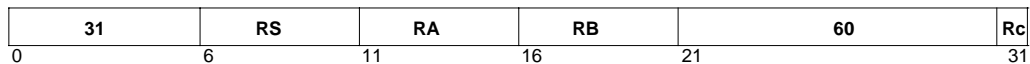
- RA
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

AND with Complement

Rc=0
Rc=1



$$(RA) \leftarrow (RS) \wedge \neg(RB)$$

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

Registers Altered

- RA
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1

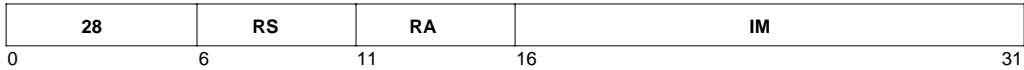
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

andi.

AND Immediate

andi. RA, RS, IM



$$(RA) \leftarrow (RS) \wedge (^{16}0 \parallel IM)$$

The IM field is extended to 32 bits by concatenating 16 0-bits on its left. The contents of register RS is ANDed with the extended IM field; the result is placed into register RA.

Registers Altered

- RA
- CR[CR0]_{LT, GT, EQ, SO}

Programming Note

The **andi.** instruction can test whether any of the 16 least-significant bits in a GPR are 1-bits.

andi. is one of three instructions that implicitly update CR[CR0] without having an Rc field. The other instructions are **addic.** and **andis.**

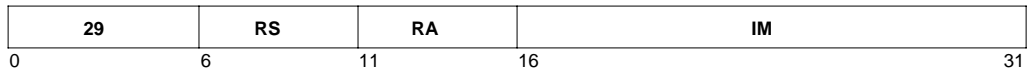
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

andis.

AND Immediate Shifted

andis. RA, RS, IM



$$(RA) \leftarrow (RS) \wedge (IM \parallel 160)$$

The IM field is extended to 32 bits by concatenating 16 0-bits on its right. The contents of register RS are ANDed with the extended IM field; the result is placed into register RA.

Registers Altered

- RA
- CR[CR0]_{LT, GT, EQ, SO}

Programming Note

The **andis.** instruction can test whether any of the 16 most-significant bits in a GPR are 1-bits.

andis. is one of three instructions that implicitly update CR[CR0] without having an Rc field. The other instructions are **addic.** and **andi..**

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

b	target	AA=0, LK=0
ba	target	AA=1, LK=0
bl	target	AA=0, LK=1
bla	target	AA=1, LK=1

18	LI	AA	LK
0	6	30	31

```
If AA = 1 then
    LI ← target6:29
    NIA ← EXTS(LI || 20)
else
    LI ← (target – CIA)6:29
    NIA ← CIA + EXTS(LI || 20)
if LK = 1 then
    (LR) ← CIA + 4
PC ← NIA
```

The next instruction address (NIA) is the effective address of the branch. The NIA is formed by adding a displacement to a base address. The displacement is obtained by concatenating two 0-bits to the right of the LI field and sign-extending the result to 32 bits.

If the AA field contains 0, the base address is the address of the branch instruction, which is also the current instruction address (CIA). If the AA field contains 1, the base address is 0.

Program flow is transferred to the NIA.

If the LK field contains 1, then (CIA + 4) is placed into the LR.

Registers Altered

- LR if LK contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

bc

Branch Conditional

bc	BO, BI, target	AA=0, LK=0
bca	BO, BI, target	AA=1, LK=0
bcl	BO, BI, target	AA=0, LK=1
bcla	BO, BI, target	AA=1, LK=1

16	BO	BI	BD	AA	LK
0	6	11	16	30	31

```

if  $BO_2 = 0$  then
     $CTR \leftarrow CTR - 1$ 
if  $(BO_2 = 1 \vee ((CTR = 0) = BO_3)) \wedge (BO_0 = 1 \vee (CR_{BI} = BO_1))$  then
    if  $AA = 1$  then
         $BD \leftarrow target_{16:29}$ 
         $NIA \leftarrow EXTS(BD \parallel ^0)$ 
    else
         $BD \leftarrow (target - CIA)_{16:29}$ 
         $NIA \leftarrow CIA + EXTS(BD \parallel ^0)$ 
else
     $NIA \leftarrow CIA + 4$ 
if  $LK = 1$  then
     $(LR) \leftarrow CIA + 4$ 
 $PC \leftarrow NIA$ 

```

9 If bit 2 of the BO field contains 0, the CTR is decremented.

The BI field specifies a bit in the CR to be used as the condition of the branch.

The next instruction address (NIA) is the effective address of the branch. The NIA is formed by adding a displacement to a base address. The displacement is obtained by concatenating two 0-bits to the right of the BD field and sign-extending the result to 32 bits.

If the AA field contains 0, the base address is the address of the branch instruction, which is also the current instruction address (CIA). If the AA field contains 1, the base address is 0.

The BO field controls options that determine when program flow is transferred to the NIA. The BO field also controls Branch Prediction, a performance-improvement feature. See Section 2.6.4 and Section 2.6.5 for a complete discussion.

If the LK field contains 1, then $(CIA + 4)$ is placed into the LR.

Registers Altered

- CTR if BO_2 contains 0
- LR if LK contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-7. Extended Mnemonics for bc, bca, bcl, bcla

Mnemonic	Operands	Function	Other Registers Changed
bdnz	target	Decrement CTR; branch if CTR \neq 0. <i>Extended mnemonic for</i> bc 16,0,target	
bdnza		<i>Extended mnemonic for</i> bca 16,0,target	
bdnzl		<i>Extended mnemonic for</i> bcl 16,0,target	(LR) \leftarrow CIA + 4.
bdnzla		<i>Extended mnemonic for</i> bcla 16,0,target	(LR) \leftarrow CIA + 4.
bdnzf	cr_bit, target	Decrement CTR; branch if CTR \neq 0 AND CR _{cr_bit} = 0. <i>Extended mnemonic for</i> bc 0,cr_bit,target	
bdnzfa		<i>Extended mnemonic for</i> bca 0,cr_bit,target	
bdnzfl		<i>Extended mnemonic for</i> bcl 0,cr_bit,target	(LR) \leftarrow CIA + 4.
bdnzfla		<i>Extended mnemonic for</i> bcla 0,cr_bit,target	(LR) \leftarrow CIA + 4.
bdnzt	cr_bit, target	Decrement CTR; branch if CTR \neq 0 AND CR _{cr_bit} = 1. <i>Extended mnemonic for</i> bc 8,cr_bit,target	
bdnzta		<i>Extended mnemonic for</i> bca 8,cr_bit,target	
bdnztl		<i>Extended mnemonic for</i> bcl 8,cr_bit,target	(LR) \leftarrow CIA + 4.
bdnztla		<i>Extended mnemonic for</i> bcla 8,cr_bit,target	(LR) \leftarrow CIA + 4.

bc

Branch Conditional

Table 9-7. Extended Mnemonics for bc, bca, bcl, bcla (cont.)

Mnemonic	Operands	Function	Other Registers Changed
bdz	target	Decrement CTR; branch if CTR = 0. <i>Extended mnemonic for</i> bc 18,0,target	
bdza		<i>Extended mnemonic for</i> bca 18,0,target	
bdzl		<i>Extended mnemonic for</i> bcl 18,0,target	(LR) \leftarrow CIA + 4.
bdzla		<i>Extended mnemonic for</i> bcla 18,0,target	(LR) \leftarrow CIA + 4.
bdzf	cr_bit, target	Decrement CTR; branch if CTR = 0 AND CR _{cr_bit} = 0. <i>Extended mnemonic for</i> bc 2,cr_bit,target	
bdzfa		<i>Extended mnemonic for</i> bca 2,cr_bit,target	
bdzfl		<i>Extended mnemonic for</i> bcl 2,cr_bit,target	(LR) \leftarrow CIA + 4.
bdzfla		<i>Extended mnemonic for</i> bcla 2,cr_bit,target	(LR) \leftarrow CIA + 4.
bdzt	cr_bit, target	Decrement CTR; branch if CTR = 0 AND CR _{cr_bit} = 1. <i>Extended mnemonic for</i> bc 10,cr_bit,target	
bdzta		<i>Extended mnemonic for</i> bca 10,cr_bit,target	
bdztl		<i>Extended mnemonic for</i> bcl 10,cr_bit,target	(LR) \leftarrow CIA + 4.
bdztla		<i>Extended mnemonic for</i> bcla 10,cr_bit,target	(LR) \leftarrow CIA + 4.

Table 9-7. Extended Mnemonics for bc, bca, bcl, bcla (cont.)

Mnemonic	Operands	Function	Other Registers Changed
beq	[cr_field,] target	Branch if equal; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 12,4*cr_field+2,target	
beqa		<i>Extended mnemonic for</i> bca 12,4*cr_field+2,target	
beql		<i>Extended mnemonic for</i> bcl 12,4*cr_field+2,target	(LR) ← CIA + 4.
beqla		<i>Extended mnemonic for</i> bcla 12,4*cr_field+2,target	(LR) ← CIA + 4.
bf	cr_bit, target	Branch if CR _{cr_bit} = 0. <i>Extended mnemonic for</i> bc 4,cr_bit,target	
bfa		<i>Extended mnemonic for</i> bca 4,cr_bit,target	
bfl		<i>Extended mnemonic for</i> bcl 4,cr_bit,target	LR
bfla		<i>Extended mnemonic for</i> bcla 4,cr_bit,target	LR
bge	[cr_field,] target	Branch if greater than or equal; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 4,4*cr_field+0,target	
bgea		<i>Extended mnemonic for</i> bca 4,4*cr_field+0,target	
bgei		<i>Extended mnemonic for</i> bcl 4,4*cr_field+0,target	LR
bgeia		<i>Extended mnemonic for</i> bcla 4,4*cr_field+0,target	LR

bc

Branch Conditional

Table 9-7. Extended Mnemonics for bc, bca, bcl, bcla (cont.)

Mnemonic	Operands	Function	Other Registers Changed
bgt	[cr_field,] target	Branch if greater than; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 12,4*cr_field+1,target	
bgta		<i>Extended mnemonic for</i> bca 12,4*cr_field+1,target	
bgtl		<i>Extended mnemonic for</i> bcl 12,4*cr_field+1,target	LR
bgsla		<i>Extended mnemonic for</i> bcla 12,4*cr_field+1,target	LR
ble	[cr_field,] target	Branch if less than or equal; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 4,4*cr_field+1,target	
blea		<i>Extended mnemonic for</i> bca 4,4*cr_field+1,target	
blel		<i>Extended mnemonic for</i> bcl 4,4*cr_field+1,target	LR
blela		<i>Extended mnemonic for</i> bcla 4,4*cr_field+1,target	LR
blt	[cr_field,] target	Branch if less than; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 12,4*cr_field+0,target	
blta		<i>Extended mnemonic for</i> bca 12,4*cr_field+0,target	
bltl		<i>Extended mnemonic for</i> bcl 12,4*cr_field+0,target	(LR) ← CIA + 4.
blsla		<i>Extended mnemonic for</i> bcla 12,4*cr_field+0,target	(LR) ← CIA + 4.

Table 9-7. Extended Mnemonics for bc, bca, bcl, bcla (cont.)

Mnemonic	Operands	Function	Other Registers Changed
bne	[cr_field,] target	Branch if not equal; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 4,4*cr_field+2,target	
bnea		<i>Extended mnemonic for</i> bca 4,4*cr_field+2,target	
bnel		Extended mnemonic for bcl 4,4*cr_field+2,target	(LR) ← CIA + 4.
bnela		<i>Extended mnemonic for</i> bcla 4,4*cr_field+2,target	(LR) ← CIA + 4.
bng	[cr_field,] target	Branch if not greater than; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 4,4*cr_field+1,target	
bnga		<i>Extended mnemonic for</i> bca 4,4*cr_field+1,target	
bngl		<i>Extended mnemonic for</i> bcl 4,4*cr_field+1,target	(LR) ← CIA + 4.
bngla		<i>Extended mnemonic for</i> bcla 4,4*cr_field+1,target	(LR) ← CIA + 4.
bnl	[cr_field,] target	Branch if not less than; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 4,4*cr_field+0,target	
bnla		<i>Extended mnemonic for</i> bca 4,4*cr_field+0,target	
bnll		<i>Extended mnemonic for</i> bcl 4,4*cr_field+0,target	(LR) ← CIA + 4.
bnlla		<i>Extended mnemonic for</i> bcla 4,4*cr_field+0,target	(LR) ← CIA + 4.

bc

Branch Conditional

Table 9-7. Extended Mnemonics for bc, bca, bcl, bcla (cont.)

Mnemonic	Operands	Function	Other Registers Changed
bns	[cr_field,] target	Branch if not summary overflow; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 4,4*cr_field+3,target	
bnsa		<i>Extended mnemonic for</i> bca 4,4*cr_field+3,target	
bnsi		<i>Extended mnemonic for</i> bcl 4,4*cr_field+3,target	(LR) ← CIA + 4.
bnsia		<i>Extended mnemonic for</i> bcla 4,4*cr_field+3,target	(LR) ← CIA + 4.
bnu	[cr_field,] target	Branch if not unordered; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 4,4*cr_field+3,target	
bnuia		<i>Extended mnemonic for</i> bca 4,4*cr_field+3,target	
bnul		<i>Extended mnemonic for</i> bcl 4,4*cr_field+3,target	(LR) ← CIA + 4.
bnula		<i>Extended mnemonic for</i> bcla 4,4*cr_field+3,target	(LR) ← CIA + 4.
bso	[cr_field,] target	Branch if summary overflow; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 12,4*cr_field+3,target	
bsoa		Extended mnemonic for bca 12,4*cr_field+3,target	
bsol		<i>Extended mnemonic for</i> bcl 12,4*cr_field+3,target	(LR) ← CIA + 4.
bsola		<i>Extended mnemonic for</i> bcla 12,4*cr_field+3,target	(LR) ← CIA + 4.

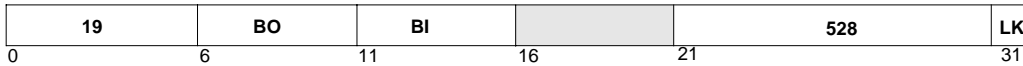
Table 9-7. Extended Mnemonics for bc, bca, bcl, bcla (cont.)

Mnemonic	Operands	Function	Other Registers Changed
bt	cr_bit, target	Branch if CR _{cr_bit} = 1. <i>Extended mnemonic for</i> bc 12,cr_bit,target	
bta		<i>Extended mnemonic for</i> bca 12,cr_bit,target	
btl		<i>Extended mnemonic for</i> bcl 12,cr_bit,target	(LR) ← CIA + 4.
btla		<i>Extended mnemonic for</i> bcla 12,cr_bit,target	(LR) ← CIA + 4.
bun	[cr_field], target	Branch if unordered; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 12,4*cr_field+3,target	
buna		<i>Extended mnemonic for</i> bca 12,4*cr_field+3,target	
bunl		<i>Extended mnemonic for</i> bcl 12,4*cr_field+3,target	(LR) ← CIA + 4.
bunla		<i>Extended mnemonic for</i> bcla 12,4*cr_field+3,target	(LR) ← CIA + 4.

bcctr

Branch Conditional to Count Register

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



```
if BO2 = 0 then
    CTR ← CTR - 1
if (BO2 = 1 ∨ ((CTR = 0) = BO3)) ∧ (BO0 = 1 ∨ (CRBI = BO1)) then
    NIA ← CTR0:29 || 200
else
    NIA ← CIA + 4
if LK = 1 then
    (LR) ← CIA + 4
PC ← NIA
```

The BI field specifies a bit in the CR to be used as the condition of the branch.

The next instruction address (NIA) is the target address of the branch. The NIA is formed by concatenating the 30 most significant bits of the CTR with two 0-bits on the right.

The BO field controls options that determine when program flow is transferred to the NIA. The BO field also controls Branch Prediction, a performance-improvement feature. See Section 2.6.4 and Section 2.6.5 for a complete discussion.

If the LK field contains 1, then (CIA + 4) is placed into the LR.

Registers Altered

- CTR if BO₂ contains 0
- LR if LK contains 1

Invalid Instruction Forms

- Reserved fields
- If bit 2 of the BO field contains 0, the instruction form is invalid, but the pseudocode applies. If the branch condition is true, the branch is taken; the NIA is the contents of the CTR after it is decremented.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-8. Extended Mnemonics for bcctr, bcctrl

Mnemonic	Operands	Function	Other Registers Changed
bctr		Branch unconditionally to address in CTR. <i>Extended mnemonic for bcctr 20,0</i>	
bctrl		<i>Extended mnemonic for bcctrl 20,0</i>	(LR) \leftarrow CIA + 4.
beqctr	[cr_field]	Branch, if equal, to address in CTR; use CR0 if cr_field is omitted. <i>Extended mnemonic for bcctr 12,4*cr_field+2</i>	
beqctrl		<i>Extended mnemonic for bcctrl 12,4*cr_field+2</i>	(LR) \leftarrow CIA + 4.
bfctr	cr_bit	Branch, if CR _{cr_bit} = 0, to address in CTR. <i>Extended mnemonic for bcctr 4,cr_bit</i>	
bfctrl		<i>Extended mnemonic for bcctrl 4,cr_bit</i>	(LR) \leftarrow CIA + 4.
bgectr	[cr_field]	Branch, if greater than or equal, to address in CTR; use CR0 if cr_field is omitted. <i>Extended mnemonic for bcctr 4,4*cr_field+0</i>	
bgectrl		<i>Extended mnemonic for bcctrl 4,4*cr_field+0</i>	(LR) \leftarrow CIA + 4.
bgtctr	[cr_field]	Branch, if greater than, to address in CTR; use CR0 if cr_field is omitted. <i>Extended mnemonic for bcctr 12,4*cr_field+1</i>	
bgtctrl		<i>Extended mnemonic for bcctrl 12,4*cr_field+1</i>	(LR) \leftarrow CIA + 4.

bcctr

Branch Conditional to Count Register

Table 9-8. Extended Mnemonics for bcctr, bcctrl (cont.)

Mnemonic	Operands	Function	Other Registers Changed
blectr	[cr_field]	Branch, if less than or equal, to address in CTR; use CR0 if cr_field is omitted. <i>Extended mnemonic for bcctr 4,4*cr_field+1</i>	
blectrl		<i>Extended mnemonic for bcctrl 4,4*cr_field+1</i>	(LR) ← CIA + 4.
bltctr	[cr_field]	Branch, if less than, to address in CTR; use CR0 if cr_field is omitted. <i>Extended mnemonic for bcctr 12,4*cr_field+0</i>	
bltctrl		<i>Extended mnemonic for bcctrl 12,4*cr_field+0</i>	(LR) ← CIA + 4.
bnctr	[cr_field]	Branch, if not equal, to address in CTR; use CR0 if cr_field is omitted. <i>Extended mnemonic for bcctr 4,4*cr_field+2</i>	
bnctrl		<i>Extended mnemonic for bcctrl 4,4*cr_field+2</i>	(LR) ← CIA + 4.
bngctr	[cr_field]	Branch, if not greater than, to address in CTR; use CR0 if cr_field is omitted. <i>Extended mnemonic for bcctr 4,4*cr_field+1</i>	
bngctrl		<i>Extended mnemonic for bcctrl 4,4*cr_field+1</i>	(LR) ← CIA + 4.
bnlctr	[cr_field]	Branch, if not less than, to address in CTR; use CR0 if cr_field is omitted. <i>Extended mnemonic for bcctr 4,4*cr_field+0</i>	
bnlctrl		<i>Extended mnemonic for bcctrl 4,4*cr_field+0</i>	(LR) ← CIA + 4.
bnsctr	[cr_field]	Branch, if not summary overflow, to address in CTR; use CR0 if cr_field is omitted. <i>Extended mnemonic for bcctr 4,4*cr_field+3</i>	
bnsctrl		<i>Extended mnemonic for bcctrl 4,4*cr_field+3</i>	(LR) ← CIA + 4.

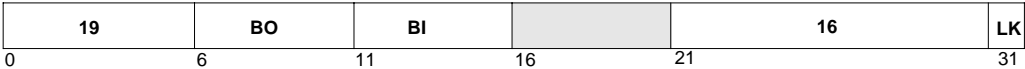
Table 9-8. Extended Mnemonics for bcctr, bcctrl (cont.)

Mnemonic	Operands	Function	Other Registers Changed
bnuctr	[cr_field]	Branch, if not unordered, to address in CTR; use CR0 if cr_field is omitted. <i>Extended mnemonic for bcctr 4,4*cr_field+3</i>	
bnuctrl		<i>Extended mnemonic for bcctrl 4,4*cr_field+3</i>	(LR) \leftarrow CIA + 4.
bsoctr	[cr_field]	Branch, if summary overflow, to address in CTR; use CR0 if cr_field is omitted. <i>Extended mnemonic for bcctr 12,4*cr_field+3</i>	
bsoctrl		<i>Extended mnemonic for bcctrl 12,4*cr_field+3</i>	(LR) \leftarrow CIA + 4.
btctr	cr_bit	Branch if CR _{cr_bit} = 1 to address in CTR. <i>Extended mnemonic for bcctr 12,cr_bit</i>	
btctrl		<i>Extended mnemonic for bcctrl 12,cr_bit</i>	(LR) \leftarrow CIA + 4.
bunctr	[cr_field]	Branch if unordered to address in CTR; use CR0 if cr_field is omitted. <i>Extended mnemonic for bcctr 12,4*cr_field+3</i>	
bunctrl		<i>Extended mnemonic for bcctrl 12,4*cr_field+3</i>	(LR) \leftarrow CIA + 4.

bclr

Branch Conditional to Link Register

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



```
if BO2 = 0 then
    CTR ← CTR - 1
if (BO2 = 1 ∨ ((CTR = 0) = BO3)) ∧ (BO0 = 1 ∨ (CRBI = BO1)) then
    NIA ← LR0:29 || 200
else
    NIA ← CIA + 4
if LK = 1 then
    (LR) ← CIA + 4
PC ← NIA
```

If bit 2 of the BO field contains 0, the CTR is decremented.

The BI field specifies a bit in the CR to be used as the condition of the branch.

The next instruction address (NIA) is the target address of the branch. The NIA is formed by concatenating the 30 most significant bits of the LR with two 0-bits on the right.

The BO field controls options that determine when program flow is transferred to the NIA. The BO field also controls Branch Prediction, a performance-improvement feature. See Section 2.6.4 and Section 2.6.5 for a complete discussion.

If the LK field contains 1, then (CIA + 4) is placed into the LR.

Registers Altered

- CTR if BO₂ contains 0
- LR if LK contains 1

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-9. Extended Mnemonics for bclr, bclrl

Mnemonic	Operands	Function	Other Registers Changed
bclr		Branch unconditionally to address in LR. <i>Extended mnemonic for bclr 20,0</i>	
bclrl		<i>Extended mnemonic for bclrl 20,0</i>	(LR) ← CIA + 4.
bdnzlr		Decrement CTR; branch if CTR ≠ 0, to address in LR. <i>Extended mnemonic for bclr 16,0</i>	
bdnzlrl		<i>Extended mnemonic for bclrl 16,0</i>	(LR) ← CIA + 4.
bdnzflr	cr_bit	Decrement CTR; branch if CTR ≠ 0 AND CR _{cr_bit} = 0 to address in LR. <i>Extended mnemonic for bclr 0,cr_bit</i>	
bdnzflrl		<i>Extended mnemonic for bclrl 0,cr_bit</i>	(LR) ← CIA + 4.
bdnztlr	cr_bit	Decrement CTR; branch if CTR ≠ 0 AND CR _{cr_bit} = 1 to address in LR. <i>Extended mnemonic for bclr 8,cr_bit</i>	
bdnztlrl		<i>Extended mnemonic for bclrl 8,cr_bit</i>	(LR) ← CIA + 4.
bdzlr		Decrement CTR; branch if CTR = 0 to address in LR. <i>Extended mnemonic for bclr 18,0</i>	
bdzlrl		<i>Extended mnemonic for bclrl 18,0</i>	(LR) ← CIA + 4.

bclr

Branch Conditional to Link Register

Table 9-9. Extended Mnemonics for bclr, bclrl (cont.)

Mnemonic	Operands	Function	Other Registers Changed
bdzflr	cr_bit	Decrement CTR; branch if CTR = 0 AND CR _{cr_bit} = 0 to address in LR. <i>Extended mnemonic for</i> bclr 2,cr_bit	
bdzflrl		<i>Extended mnemonic for</i> bclrl 2,cr_bit	(LR) ← CIA + 4.
bdztlr	cr_bit	Decrement CTR; branch if CTR = 0 AND CR _{cr_bit} = 1 to address in LR. <i>Extended mnemonic for</i> bclr 10,cr_bit	
bdztlrl		<i>Extended mnemonic for</i> bclrl 10,cr_bit	(LR) ← CIA + 4.
beqlr	[cr_field]	Branch if equal to address in LR; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 12,4*cr_field+2	
beqlrl		<i>Extended mnemonic for</i> bclrl 12,4*cr_field+2	(LR) ← CIA + 4.
bflr	cr_bit	Branch if CR _{cr_bit} = 0 to address in LR. <i>Extended mnemonic for</i> bclr 4,cr_bit	
bflrl		<i>Extended mnemonic for</i> bclrl 4,cr_bit	(LR) ← CIA + 4.
bge1r	[cr_field]	Branch, if greater than or equal, to address in LR; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 4,4*cr_field+0	
bge1rl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+0	(LR) ← CIA + 4.
bg1tr	[cr_field]	Branch, if greater than, to address in LR; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 12,4*cr_field+1	
bg1trl		<i>Extended mnemonic for</i> bclrl 12,4*cr_field+1	(LR) ← CIA + 4.

Table 9-9. Extended Mnemonics for bclr, bclrl (cont.)

Mnemonic	Operands	Function	Other Registers Changed
blelr	[cr_field]	Branch, if less than or equal, to address in LR; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 4,4*cr_field+1	
blelrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+1	(LR) ← CIA + 4.
bltlr	[cr_field]	Branch, if less than, to address in LR; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 12,4*cr_field+0	
btllrl		<i>Extended mnemonic for</i> bclrl 12,4*cr_field+0	(LR) ← CIA + 4.
bnelr	[cr_field]	Branch, if not equal, to address in LR; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 4,4*cr_field+2	
bnelrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+2	(LR) ← CIA + 4.
bnglr	[cr_field]	Branch, if not greater than, to address in LR; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 4,4*cr_field+1	
bnglrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+1	(LR) ← CIA + 4.
bnllr	[cr_field]	Branch, if not less than, to address in LR; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 4,4*cr_field+0	
bnllrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+0	(LR) ← CIA + 4.
bnslr	[cr_field]	Branch if not summary overflow to address in LR; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 4,4*cr_field+3	
bnslrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+3	(LR) ← CIA + 4.

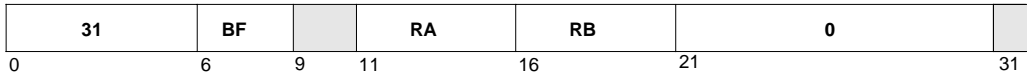
bclr

Branch Conditional to Link Register

Table 9-9. Extended Mnemonics for bclr, bclrl (cont.)

Mnemonic	Operands	Function	Other Registers Changed
bnulr	[cr_field]	Branch if not unordered to address in LR; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 4,4*cr_field+3	
bnulrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+3	(LR) ← CIA + 4.
bsolr	[cr_field]	Branch if summary overflow to address in LR; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 12,4*cr_field+3	
bsolrl		<i>Extended mnemonic for</i> bclrl 12,4*cr_field+3	(LR) ← CIA + 4.
btlr	cr_bit	Branch if CR _{cr_bit} = 1 to address in LR. <i>Extended mnemonic for</i> bclr 12,cr_bit	
btlrl		<i>Extended mnemonic for</i> bclrl 12,cr_bit	(LR) ← CIA + 4.
bunlr	[cr_field]	Branch if unordered to address in LR; use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 12,4*cr_field+3	
bunrlrl		<i>Extended mnemonic for</i> bclrl 12,4*cr_field+3	(LR) ← CIA + 4.

cmp BF, 0, RA, RB



```

c0:3 ← 40
if (RA) < (RB) then c0 ← 1
if (RA) > (RB) then c1 ← 1
if (RA) = (RB) then c2 ← 1
c3 ← XER[SO]
n ← BF
CR[CRn] ← c0:3

```

The contents of register RA are compared with the contents of register RB using a 32-bit signed compare.

The CR field specified by the BF field is updated to reflect the results of the compare and the value of XER[SO] is placed into the same CR field.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- CR[CRn] where n is specified by the BF field

Invalid Instruction Forms

- Reserved fields

Programming Note

The PowerPC Architecture defines this instruction as **cmp BF,L,RA,RB**, where L selects operand size for 64-bit PowerPC implementations. For all 32-bit PowerPC implementations, L = 0 is required (L = 1 is an invalid form); hence for PPC401A1, use of the extended mnemonic **cmpw BF,RA,RB** is recommended.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

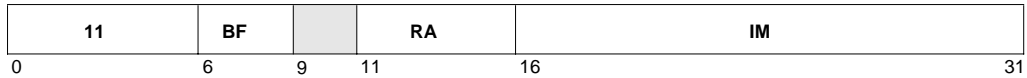
Table 9-10. Extended Mnemonics for cmp

Mnemonic	Operands	Function	Other Registers Changed
cmpw	[BF,] RA, RB	Compare Word; use CR0 if BF is omitted. <i>Extended mnemonic for cmp BF,0,RA,RB</i>	

cmpi

Compare Immediate

cmpi BF, 0, RA, IM



```
c0:3 ← 40
if (RA) < EXTS(IM) then c0 ← 1
if (RA) > EXTS(IM) then c1 ← 1
if (RA) = EXTS(IM) then c2 ← 1
c3 ← XER[SO]
n ← BF
CR[CRn] ← c0:3
```

The IM field is sign-extended to 32 bits. The contents of register RA are compared with the extended IM field, using a 32-bit signed compare.

The CR field specified by the BF field is updated to reflect the results of the compare and the value of XER[SO] is placed into the same CR field.

Registers Altered

- CR[CR_n] where *n* is specified by the BF field

9

Invalid Instruction Forms

- Reserved fields

Programming Note

The PowerPC Architecture defines this instruction as **cmpi BF,L,RA,IM**, where L selects operand size for 64-bit PowerPC implementations. For all 32-bit PowerPC implementations, L = 0 is required (L = 1 is an invalid form); hence for PPC401A1, use of the extended mnemonic **cmpwi BF,RA,IM** is recommended.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-11. Extended Mnemonics for cmpi

Mnemonic	Operands	Function	Other Registers Changed
cmpwi	[BF,] RA, IM	Compare Word Immediate; use CR0 if BF is omitted. <i>Extended mnemonic for cmpi BF,0,RA,IM</i>	

cmpl BF, 0, RA, RB

31	BF		RA	RB	32	
0	6	9	11	16	21	31

```

 $c_{0:3} \leftarrow {}^40$ 
if (RA)  $\overset{u}{<}$  (RB) then  $c_0 \leftarrow 1$ 
if (RA)  $\overset{u}{>}$  (RB) then  $c_1 \leftarrow 1$ 
if (RA) = (RB) then  $c_2 \leftarrow 1$ 
 $c_3 \leftarrow \text{XER}[\text{SO}]$ 
 $n \leftarrow \text{BF}$ 
 $\text{CR}[\text{CRn}] \leftarrow c_{0:3}$ 

```

The contents of register RA are compared with the contents of register RB, using a 32-bit unsigned compare.

The CR field specified by the BF field is updated to reflect the results of the compare and the value of XER[SO] is placed into the same CR field.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- CR[CRn] where n is specified by the BF field

Invalid Instruction Forms

- Reserved fields

Programming Notes

The PowerPC Architecture defines this instruction as **cmpl BF,L,RA,RB**, where L selects operand size for 64-bit PowerPC implementations. For all 32-bit PowerPC implementations, L = 0 is required (L = 1 is an invalid form); hence for PPC401A1, use of the extended mnemonic **cmplw BF,RA,RB** is recommended.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

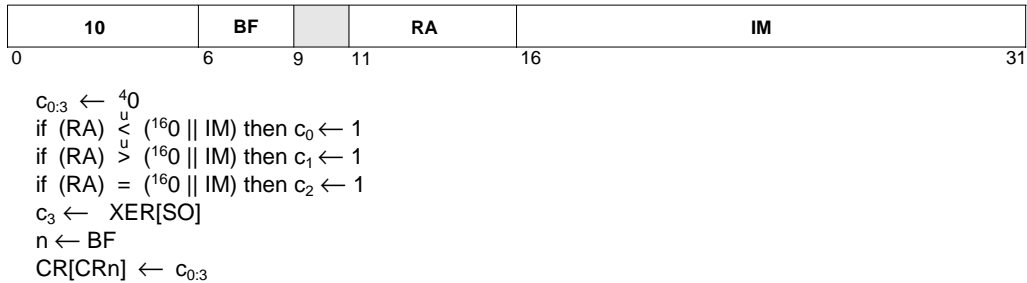
Table 9-12. Extended Mnemonics for cmpl

Mnemonic	Operands	Function	Other Registers Changed
cmplw	[BF,] RA, RB	Compare Logical Word; use CR0 if BF is omitted. <i>Extended mnemonic for cmpl BF,0,RA,RB</i>	

cmpli

Compare Logical Immediate

cmpli BF, 0, RA, IM



The IM field is extended to 32 bits by concatenating 16 0-bits to its left. The contents of register RA are compared with IM using a 32-bit unsigned compare.

The CR field specified by the BF field is updated to reflect the results of the compare and the value of XER[SO] is placed into the same CR field.

Registers Altered

- CR[CRn] where n is specified by the BF field

Invalid Instruction Forms

- Reserved fields

Programming Note

The PowerPC Architecture defines this instruction as **cmpli BF,L,RA,IM**, where L selects operand size for 64-bit PowerPC implementations. For all 32-bit PowerPC implementations, L = 0 is required (L = 1 is an invalid form); hence for PPC401A1, use of the extended mnemonic **cmplwi BF,RA,IM** is recommended.

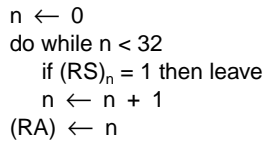
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-13. Extended Mnemonics for cmpli

Mnemonic	Operands	Function	Other Registers Changed
cmplwi	[BF,] RA, IM	Compare Logical Word Immediate; use CR0 if BF is omitted. <i>Extended mnemonic for cmpli BF,0,RA,IM</i>	

Count Leading Zeros Word

$$R_c = 0$$
$$R_c = 1$$


The count ranges from 0 through 32, inclusive.

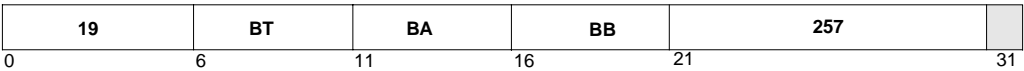
- RA
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1

- Reserved fields

crand

Condition Register AND

crand BT, BA, BB



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

The CR bit specified by the BA field is ANDed with the CR bit specified by the BB field; the result is placed into the CR bit specified by the BT field.

Registers Altered

- CR

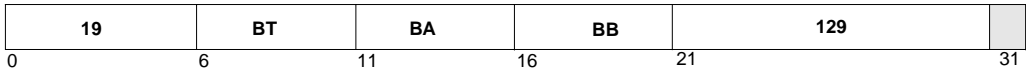
Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

crandc BT, BA, BB



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

The CR bit specified by the BA field is ANDed with the ones complement of the CR bit specified by the BB field; the result is placed into the CR bit specified by the BT field.

Registers Altered

- CR

Invalid Instruction Forms

- Reserved fields

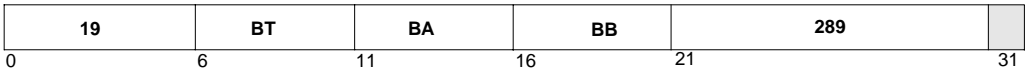
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

creqv

Condition Register Equivalent

creqv BT, BA, BB



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

The CR bit specified by the BA field is XORed with the CR bit specified by the BB field; the ones complement of the result is placed into the CR bit specified by the BT field.

Registers Altered

- CR

Invalid Instruction Forms

- Reserved fields

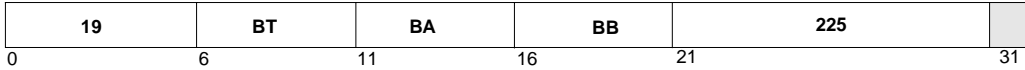
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-14. Extended Mnemonics for creqv

Mnemonic	Operands	Function	Other Registers Changed
crset	bx	Condition register set. <i>Extended mnemonic for creqv bx,bx,bx</i>	

crnand BT, BA, BB



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

The CR bit specified by the BA field is ANDed with the CR bit specified by the BB field; the ones complement of the result is placed into the CR bit specified by the BT field.

Registers Altered

- CR

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

crnor

Condition Register NOR

crnor BT, BA, BB



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

The CR bit specified by the BA field is ORed with the CR bit specified by the BB field; the ones complement of the result is placed into the CR bit specified by the BT field.

Registers Altered

- CR

Invalid Instruction Forms

- Reserved fields

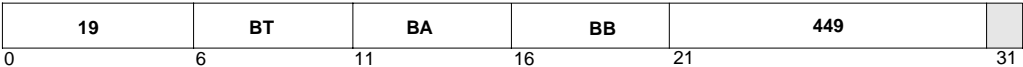
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-15. Extended Mnemonics for crnor

Mnemonic	Operands	Function	Other Registers Changed
crnot	bx, by	Condition register not. <i>Extended mnemonic for crnor bx,by,by</i>	

cror BT, BA, BB



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

The CR bit specified by the BA field is ORed with the CR bit specified by the BB field; the result is placed into the CR bit specified by the BT field.

Registers Altered

- CR

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

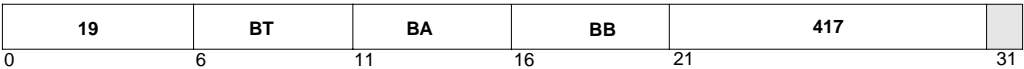
Table 9-16. Extended Mnemonics for cror

Mnemonic	Operands	Function	Other Registers Changed
crmove	bx, by	Condition register move. <i>Extended mnemonic for cror bx,by,by</i>	

crorc

Condition Register OR with Complement

crorc BT, BA, BB



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

The condition register (CR) bit specified by the BA field is ORed with the ones complement of the CR bit specified by the BB field; the result is placed into the CR bit specified by the BT field.

Registers Altered

- CR

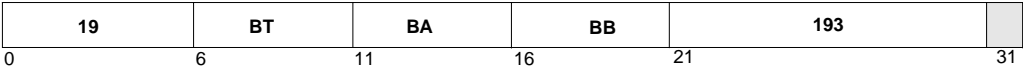
Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

crxor BT, BA, BB



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

The CR bit specified by the BA field is XORed with the CR bit specified by the BB field; the result is placed into the CR bit specified by the BT field.

Registers Altered

- CR

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

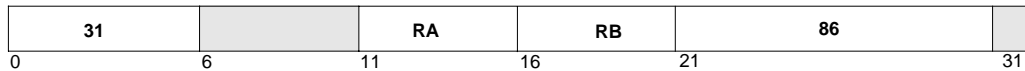
Table 9-17. Extended Mnemonics for crxor

Mnemonic	Operands	Function	Other Registers Changed
crclr	bx	Condition register clear. <i>Extended mnemonic for crxor bx,bx,bx</i>	

dcba

Data Cache Block Flush

dcbf RA, RB



$EA \leftarrow (RA|0) + (RB)$
DCBF(EA)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

If the data block corresponding to the EA is in the data cache and marked as modified (stored into), the data block is copied back to main storage and then marked invalid in the data cache. If the data block is not marked as modified, it is simply marked invalid in the data cache. The operation is performed whether or not the EA is marked as cacheable.

If the data block at the EA is not in the data cache, no operation is performed.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

9

Invalid Instruction Forms

- Reserved fields

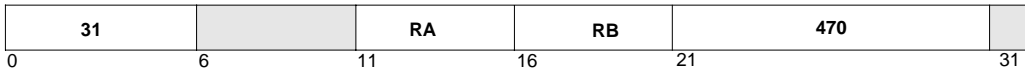
Exceptions

This instruction is considered a “load” with respect to data storage exceptions (for the PPC401A1, these are cache line locking exceptions). See Section 5.6, “Data Storage Exceptions,” on p. 5-15, for more information.

This instruction is considered a “store” with respect to data address compare (DAC) debug exceptions. See Section 7.6.3.1, “Data Address Compare (DAC) Applied to Cache Instructions,” on p. 7-9, for more information.

Architecture Note

This instruction is part of the PowerPC Virtual Environment Architecture.

dcbi RA, RB

$EA \leftarrow (RA[0] + (RB))$
 DCBI(EA)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

If the data block at the EA is in the data cache, the data block is marked invalid, regardless of whether or not the EA is marked as cacheable. If modified data existed in the data block prior to the operation of this instruction, that data is lost.

If the data block at the EA is not in the data cache, no operation is performed.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Notes

Execution of this instruction is privileged.

Exceptions

This instruction is considered a “store” with respect to data address compare (DAC) debug exceptions. See Section 7.6.3.1, “Data Address Compare (DAC) Applied to Cache Instructions,” on p. 7-9, for more information.

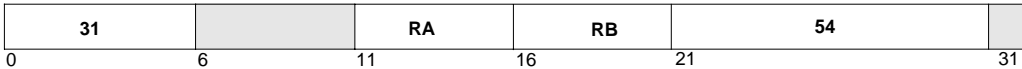
Architecture Note

This instruction is part of the PowerPC Operating Environment Architecture.

dcbst

Data Cache Block Store

dcbst RA, RB



$EA \leftarrow (RA \ll 0) + (RB)$
DCBST(EA)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0, and is the contents of register RA otherwise.

If the data block at the EA is in the data cache and marked as modified, the data block is copied back to main storage and marked as unmodified in the data cache.

If the data block at the EA is in the data cache, and is not marked as modified, or if the data block at the EA is not in the data cache, no operation is performed.

The operation specified by this instruction is performed whether or not the EA is marked as cacheable.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

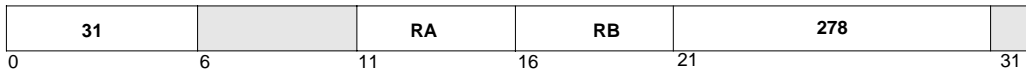
- Reserved fields

Exceptions

This instruction is considered a “store” with respect to data address compare (DAC) debug exceptions. See Section 7.6.3.1, “Data Address Compare (DAC) Applied to Cache Instructions,” on p. 7-9, for more information.

Architecture Note

This instruction is part of the PowerPC Virtual Environment Architecture.

dcbt RA, RB

$EA \leftarrow (RA|0) + (RB)$
 DCBT(EA)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

If the data block at the EA is not in the data cache and the EA is marked as cacheable, the block is read from main storage into the data cache.

If the data block at the EA is in the data cache, or if the EA is marked as non-cacheable, no operation is performed.

This instruction is not allowed to cause data storage exceptions. If execution of the instruction would cause such an exception, then no operation is performed, and no exception occurs.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Notes

The **dcbt** instruction allows a program to begin a cache block fetch from main storage before the program needs the data. The program can later load data from the cache into registers without incurring the latency of a cache miss.

dcbt

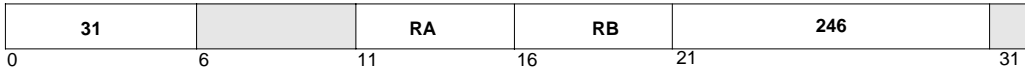
Data Cache Block Touch

Exceptions

This instruction is considered a “load” with respect to data address compare (DAC) debug exceptions. See Section 7.6.3.1, “Data Address Compare (DAC) Applied to Cache Instructions,” on p. 7-9, for more Information.

Architecture Note

This instruction is part of the PowerPC Virtual Environment Architecture.

dcbtst RA, RB

$$EA \leftarrow (RA|0) + (RB)$$

$$DCBTST(EA)$$

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

If the data block at the EA is not in the data cache and the EA address is marked as cacheable, the data block is loaded into the data cache.

If the EA is marked as non-cacheable, or if the data block at the EA is in the data cache, no operation is performed.

This instruction is not allowed to cause data storage exceptions. If execution of the instruction would cause such an exception, then no operation is performed, and no exception occurs.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Notes

The **dcbtst** instruction allows a program to begin a cache block fetch from main storage before the program needs the data. The program can later store data from GPRs into the cache block, without incurring the latency of a cache miss.

Architecturally, **dcbtst** brings data into the cache in “Exclusive” mode, which allows the program to alter the cached data. “Exclusive” mode is part of the MESI protocol for multi-processor systems, and is not implemented. The implementation of the **dcbtst** instruction is identical to the implementation of the **dcbt** instruction.

dcbtst

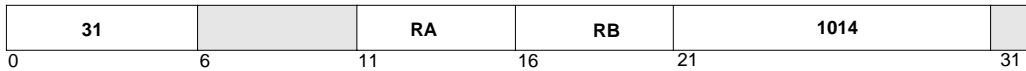
Data Cache Block Touch for Store

Exceptions

This instruction is considered a “load” with respect to data address compare (DAC) debug exceptions. See Section 7.6.3.1, “Data Address Compare (DAC) Applied to Cache Instructions,” on p. 7-9, for more Information.

Architecture Note

This instruction is part of the PowerPC Virtual Environment Architecture.

dcbz RA, RB

$EA \leftarrow (RA \ll 0) + (RB)$
 DCBZ(EA)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

If the data block at the EA is in the data cache and the EA is marked as cacheable and non-write-through, the data in the cache block is set to 0.

If the data block at the EA is not in the data cache and the EA is marked as cacheable and non-write-through, a cache block is established and set to 0. Note that nothing is read from main storage, as described in the programming note.

If the data block at the EA is marked as either write-through or as non-cacheable, an alignment exception occurs.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Notes

Because the **dcbz** instruction can establish an address in the data cache without copying the contents of that address from main storage, the address established may be invalid with respect to the storage subsystem. A subsequent operation may cause the address to be copied back to main storage, for example, to make room for a new cache block; a machine check exception could occur under these circumstances.

If **dcbz** is attempted to an EA which is marked as non-cacheable, the software alignment exception handler should emulate the instruction by storing zeros to the block in main storage. If a data block corresponding to the EA exists in the cache, but the EA is non-cacheable, stores (including **dcbz**) to that address are considered programming errors (the cache block should previously have been flushed).

dcbz

Data Cache Block Set to Zero

If the EA is marked as write-through, the software alignment exception handler should emulate the instruction by storing zeros to the block in main storage. An EA that is marked as write-through required should also be marked as cacheable; when **dcbz** is attempted to such an address, the alignment exception handler should maintain coherency of cache and memory.

Exceptions

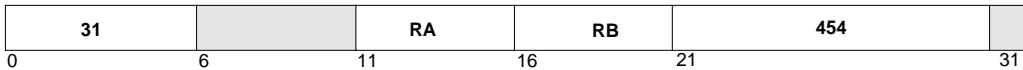
An alignment exception occurs if the EA is marked as non-cacheable or as write-through.

This instruction is considered a “store” with respect to data address compare (DAC) debug exceptions. See Section 7.6.3.1, “Data Address Compare (DAC) Applied to Cache Instructions,” on p. 7-9, for more information.

This instruction is considered a “store” with respect to data storage exceptions (for the PPC401A1, these are cache line locking exceptions). See Section 5.6, “Data Storage Exceptions,” on p. 5-15, for more information.

Architecture Note

This instruction is part of the PowerPC Virtual Environment Architecture.

dccci RA, RB

$EA \leftarrow (RA|0) + (RB)$
 DCCCI(EA)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

Both cache lines in the congruence class specified by $EA_{23:27}$ are invalidated, whether or not they match the EA. If modified data existed in the cache congruence class before the operation of this instruction, that data is lost.

The operation specified by this instruction is performed whether or not the EA is marked as cacheable.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Note

Execution of this instruction is privileged.

This instruction is intended for use in the power-on reset routine to invalidate the entire data cache tag array before enabling the data cache. A series of **dccci** instruction should be executed, one for each congruence class. Cacheability can then be enabled.

Exceptions

This instruction will not cause data address compare (DAC) debug exceptions. See Section 7.6.3.1, “Data Address Compare (DAC) Applied to Cache Instructions,” on p. 7-9, for more Information.

dccci

Data Cache Congruence Class Invalidate

Architecture Note

Programs using this instruction are not portable to PowerPC implementations that do not implement the IBM PowerPC Embedded Environment.

dcread RT, RA, RB

31	RT	RA	RB	486	
0	6	11	16	21	31

$$EA \leftarrow (RA[0] + (RB))$$

if ((CDBCR[CIS] = 0) \wedge (CDBCR[CWS] = 0)) then (RT) \leftarrow (d-cache data, way A)

if ((CDBCR[CIS] = 0) \wedge (CDBCR[CWS] = 1)) then (RT) \leftarrow (d-cache data, way B)

if ((CDBCR[CIS] = 1) \wedge (CDBCR[CWS] = 0)) then (RT) \leftarrow (d-cache tag, way A)

if ((CDBCR[CIS] = 1) \wedge (CDBCR[CWS] = 1)) then (RT) \leftarrow (d-cache tag, way B)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

This instruction is a debugging tool for reading the data cache entries for the congruence class specified by $EA_{23:27}$. The cache information is read into register RT.

If (CDBCR[CIS] = 0), the information is a word of data cache data from the addressed congruence class. The word is specified by $EA_{28:29}$; $EA_{0:22}$ are ignored. If $EA_{30:31}$ are not 00, an alignment exception occurs. If (CDBCR[CWS] = 0), the data is from the A-way, otherwise the data are from the B-way.

If (CDBCR[CIS] = 1), the information is a cache tag from the addressed congruence class; $EA_{0:22}$ and $EA_{28:31}$ are ignored. If (CDBCR[CWS] = 0), the tag is from the A-way, otherwise the tag is from the B-way. Data cache tag information is placed into register RT as follows:

Table 9-18. Data Cache Array Tag Information

0:22	TAG	Cache Tag
23:24		Reserved
25	LK	Cache Line Lock 0 Unlocked 1 Locked
26	D	Cache Line Dirty 0 Not dirty 1 Dirty
27	V	Cache Line Valid 0 Not valid 1 Valid
28:30		Reserved
31	LRU	Least Recently Used (LRU) 0 A-way LRU 1 B-way LRU

dcread

Data Cache Read

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields

Programming Note

Execution of this instruction is privileged.

Exceptions

If EA is not word-aligned, an alignment exception occurs.

This instruction is considered a “load” with respect to data address compare (DAC) debug exceptions. See Section 7.6.3.1, “Data Address Compare (DAC) Applied to Cache Instructions,” on p. 7-9, for more Information.

Architecture Note

Programs using this instruction are not portable to PowerPC implementations that do not implement the IBM PowerPC Embedded Environment.

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

31	RT	RA	RB	OE	491	Rc
0	6	11	16	21 22		31

$(RT) \leftarrow (RA) \div (RB)$

The contents of register RA are divided by the contents of register RB. The quotient is placed into register RT.

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

$\text{dividend} = (\text{quotient} \times \text{divisor}) + \text{remainder}$

where the remainder has the same sign as the dividend and its magnitude is less than that of the divisor.

If an attempt is made to perform $(0x8000\ 0000 \div -1)$ or $(n \div 0)$, the contents of register RT are undefined; if the Rc field also contains 1, the contents of CR[CR0]_{LT,GT,EQ} are undefined. Either invalid division operation sets XER[OV, SO] to 1 if the OE field contains 1.

Registers Altered

- RT
- CR[CR0]_{LT,GT,EQ,SO} if Rc contains 1
- XER[OV, SO] if OE contains 1

Programming Note

The 32-bit remainder can be calculated using the following sequence of instructions:

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

The sequence does not calculate correct results for the invalid divide operations.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

divwu

Divide Word Unsigned

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

31	RT	RA	RB	OE	459	Rc
0	6	11	16	21 22		31

$(RT) \leftarrow (RA) \div (RB)$

The contents of register RA are divided by the contents of register RB. The quotient is placed into register RT.

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

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

If an attempt is made to perform $(n \div 0)$, the contents of register RT are undefined; if the Rc also contains 1, the contents of CR[CR0]_{LT,GT,EQ} are also undefined. The invalid division operation also sets XER[OV, SO] to 1 if the OE field contains 1.

Registers Altered

- RT
- CR[CR0]_{LT,GT,EQ,SO} if Rc contains 1
- XER[OV, SO] if OE contains 1

Programming Note

The 32-bit remainder can be calculated using the following sequence of instructions

divwu	RT,RA,RB	# RT = quotient
mullw	RT,RT,RB	# RT = quotient \times divisor
subf	RT,RT,RA	# RT = remainder

This sequence does not calculate the correct result if the divisor is zero.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

eieio



The **eieio** instruction ensures that all loads and stores preceding an **eieio** instruction complete with respect to main storage before any loads and stores following the **eieio** instruction access main storage.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Note

Architecturally, **eieio** orders storage access, not instruction completion. Therefore, non-storage operations after **eieio** could complete before storage operations that were before **eieio**. The **sync** instruction guarantees ordering of both instruction completion and storage access. For the PPC401A1, the **eieio** instruction is implemented to behave as a **sync** instruction. To write code which is portable between various PowerPC implementations, programmers should use the mnemonic which corresponds to the desired behavior.

Architecture Note

This instruction is part of the PowerPC Virtual Environment Architecture.

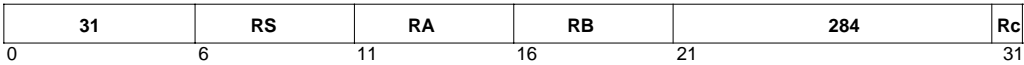
eqv

Equivalent

eqv
eqv.

RA, RS, RB
RA, RS, RB

Rc=0
Rc=1



$(RA) \leftarrow \neg((RS) \oplus (RB))$

The contents of register RS are XORed with the contents of register RB; the ones complement of the result is placed into register RA.

Registers Altered

- RA
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

extsb RA, RS Rc=0
extsb. RA, RS Rc=1



$(RA) \leftarrow \text{EXTS}(RS)_{24:31}$

The least significant byte of register RS is sign-extended to 32 bits by replicating bit 24 of the register into bits 0 through 23 of the result. The result is placed into register RA.

Registers Altered

- RA
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1

Invalid Instruction Forms

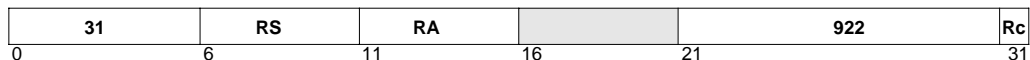
- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Extend Sign Halfword

Rc=0
Rc=1

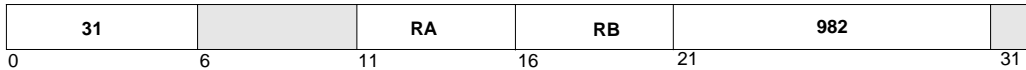


The least significant halfword of register RS is sign-extended to 32 bits by replicating bit 16 of the register into bits 0 through 15 of the result. The result is placed into register RA.

- RA
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1

- Reserved fields

This instruction is part of the PowerPC User Instruction Set Architecture.

icbi RA, RB

$EA \leftarrow (RA|0) + (RB)$
 ICBI(EA)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

If the instruction block at the EA is in the instruction cache, the cache block is marked invalid.

If the instruction block at the EA is not in the instruction cache, no additional operation is performed.

The operation specified by this instruction is performed whether or not the EA is marked as cacheable in the ICCR.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Note

Cacheability for the EA of the operand of instruction cache operations is determined by the ICCR, not the DCCR.

icbi

Instruction Cache Block Invalidate

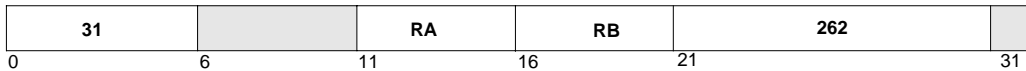
Exceptions

This instruction is considered a “load” with respect to data storage exceptions (for the PPC401A1, these are cache line locking exceptions). See Section 5.6, “Data Storage Exceptions,” on p. 5-15, for more information.

This instruction is considered a “load” with respect to data address compare (DAC) debug exceptions, but will not cause DAC debug events.

Architecture Note

This instruction is part of the PowerPC Virtual Environment Architecture.

icbt RA, RB

$EA \leftarrow (RA \ll 0) + (RB)$
 $ICBT(EA)$

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

If the instruction block at the EA is not in the instruction cache, and is marked as cacheable, the instruction block is loaded into the instruction cache.

If the instruction block at the EA is in the instruction cache, or if the EA is marked as non-cacheable, no operation is performed.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Notes

Execution of this instruction is privileged.

This instruction allows a program to begin a cache block fetch from main storage before the program needs the instruction. The program can later branch to the instruction address and fetch the instruction from the cache without incurring the latency of a cache miss.

Cacheability for the effective address of the operand of instruction cache operations is determined by the ICCR, not the DCCR.

icbt

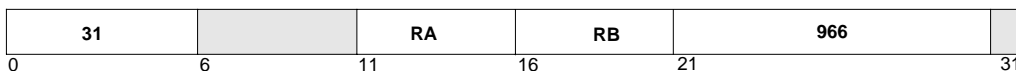
Instruction Cache Block Touch

Exceptions

This instruction is considered a “load” with respect to data address compare (DAC) debug exceptions, but will not cause DAC debug events.

Architecture Note

Programs using this instruction are not portable to PowerPC implementations that do not implement the IBM PowerPC Embedded Environment.

iccci RA, RB

$$EA \leftarrow (RA[0] + (RB))$$

$$ICCCI(EA)$$

An effective address is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

Both cache lines in the congruence class specified by $EA_{22:27}$ are invalidated, whether or not they match the effective address.

The operation specified by this instruction is performed whether or not the effective address is marked cacheable.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Notes

Execution of this instruction is privileged.

This instruction is intended for use in the power-on reset routine to invalidate the entire cache tag array before enabling the cache. A series of **iccci** instructions should be executed, one for each congruence class. Cacheability can then be enabled.

Cacheability for the effective address of the operand of instruction cache operations is determined by the ICCR, not the DCCR.

Exceptions

This instruction is considered a “load” with respect to data address compare (DAC) debug exceptions, but will not cause DAC debug events.

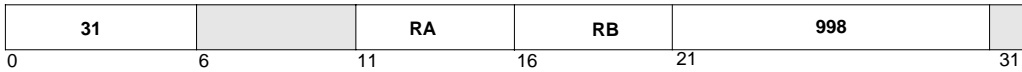
Architecture Note

Programs using this instruction are not portable to PowerPC implementations that do not implement the IBM PowerPC Embedded Environment.

icread

Instruction Cache Read

icread RA, RB



$$EA \leftarrow (RA \ll 0) + (RB)$$

if ((CDBCR[CIS] = 0) \wedge (CDBCR[CWS] = 0)) then (ICDBDR \leftarrow (i-cache data, way A))

if ((CDBCR[CIS] = 0) \wedge (CDBCR[CWS] = 1)) then (ICDBDR \leftarrow (i-cache data, way B))

if ((CDBCR[CIS] = 1) \wedge (CDBCR[CWS] = 0)) then (ICDBDR \leftarrow (i-cache tag, way A))

if ((CDBCR[CIS] = 1) \wedge (CDBCR[CWS] = 1)) then (ICDBDR \leftarrow (i-cache tag, way B))

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

This instruction is a debugging tool for reading the instruction cache entries for the congruence class specified by $EA_{22:27}$. The cache information is read into the Instruction Cache Debug Data Register (ICDBDR), from where it can be read into a GPR using the extended mnemonic **mficbdr**.

If CDBCR[CIS] = 0, the information is a word of instruction cache data from the addressed line. The word is specified by $EA_{28:29}$ ($EA_{0:21}$ and $EA_{30:31}$ are ignored). If CDBCR[CWS] = 0, the data is from the A-way, otherwise from the B-way.

If (CDBCR[CIS] = 1), the information is a cache tag from the addressed congruence class ($EA_{0:21}$ and $EA_{28:31}$ are ignored). If (CDBCR[CWS] = 0), the tag is from the A-way, otherwise from the B-way. Instruction cache tag information is placed in the ICDBDR as follows.

Table 9-19. Instruction Cache Array Tag Information

0:22	TAG	Cache Tag
23:24		Reserved
25	LK	Cache Line Lock 0 Unlocked 1 Locked
26		Reserved
27	V	Cache Line Valid 0 Not valid 1 Valid
28:30		Reserved
31	LRU	Least Recently Used (LRU) 0 A-way LRU 1 B-way LRU

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- ICDBDR

Invalid Instruction Forms

- Reserved fields

Programming Note

Execution of this instruction is privileged.

The instruction pipeline does not automatically wait for data from **icread** to arrive at the ICDBDR before attempting to use the contents of the ICDBDR. Therefore, insert an **isync** instruction between **icread** and **mficdbdr**.

icread r5,r6	# read cache information
isync	# ensure completion of icread
mficdbdr r7	# move information to GPR

Cacheability for the EA of the operand of instruction cache operations is determined by the ICCR, not the DCCR.

Exceptions

This instruction is considered a “load” with respect to data address compare (DAC) debug exceptions, but will not cause DAC debug events.

Architecture Note

Programs using this instruction are not portable to PowerPC implementations that do not implement the IBM PowerPC Embedded Environment.

isync

Instruction Synchronize

isync



The **isync** instruction is a context synchronizing instruction.

The **isync** instruction provides an ordering function for the effects of all instructions executed by the processor. Executing **isync** insures that all instructions preceding the **isync** instruction have completed before the **isync** instruction completes, except that storage accesses caused by those instructions need not have completed. No subsequent instructions are initiated by the processor until after the **isync** instruction completes. Finally, execution of **isync** causes the processor to discard any prefetched instructions, with the effect that subsequent instructions are fetched and executed in the context established by the instructions preceding the **isync** instruction.

The **isync** instruction has no effect on caches.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

9

Invalid Instruction Forms

- Reserved fields

Programming Note

See the discussion of context synchronizing instructions in Section 2.9.1 on p. 2-40.

The following code example illustrates the necessary steps for self-modifying code. This example assumes that `addr1` is both data and instruction cacheable.

```
stw      regN, addr1    # the data in regN is to become an instruction at addr1
dcbst    addr1          # forces data from the data cache to memory
sync     # wait until the data actually reaches the memory
icbi     addr1          # the previous value at addr1 might already be in
                        # the instruction cache; invalidate in the cache
isync    # the previous value at addr1 might already have been
                        # pre-fetched into the queue; invalidate the queue
                        # so that the instruction must be re-fetched
```

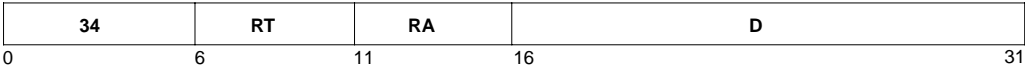
Architecture Note

This instruction is part of the PowerPC Virtual Environment Architecture.

lbz

Load Byte and Zero

lbz RT, D(RA)



$EA \leftarrow (RA|0) + EXTS(D)$
 $(RT) \leftarrow {}^{24}0 \parallel MS(EA,1)$

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The byte at the EA is extended to 32 bits by concatenating 24 0-bits to its left. The result is placed into register RT.

Registers Altered

- RT

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Ibzu RT, D(RA)

35	RT	RA	D
0	6	11	16
			31

$EA \leftarrow (RA|0) + \text{EXTS}(D)$
 $(RA) \leftarrow EA$
 $(RT) \leftarrow {}^{24}0 \parallel \text{MS}(EA,1)$

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise. The EA is placed into register RA.

The byte at the EA is extended to 32 bits by concatenating 24 0-bits to its left. The result is placed into register RT.

Registers Altered

- RA
- RT

Invalid Instruction Forms

- RA=RT
- RA=0

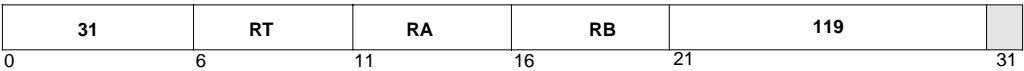
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lbzux

Load Byte and Zero with Update Indexed

lbzux RT, RA, RB



```
EA ← (RA|0) + (RB)
(RA) ← EA
(RT) ← 240 || MS(EA,1)
```

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise. The EA is placed into register RA.

The byte at the EA is extended to 32 bits by concatenating 24 0-bits to its left. The result is placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RA
- RT

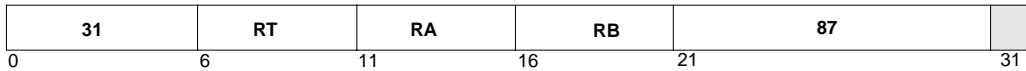
Invalid Instruction Forms

- Reserved fields
- RA=RT
- RA=0

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lbzx RT,RA, RB



$EA \leftarrow (RA|0) + (RB)$
 $(RT) \leftarrow {}^{24}0 \parallel MS(EA,1)$

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The byte at the EA is extended to 32 bits by concatenating 24 0-bits to its left. The result is placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lha

Load Halfword Algebraic

lha RT, D(RA)



$EA \leftarrow (RA|0) + EXTS(D)$
 $(RT) \leftarrow EXTS(MS(EA,2))$

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The halfword at the EA is sign-extended to 32 bits and placed into register RT.

Registers Altered

- RT

Exceptions

An alignment exception occurs if MSR[LE] = 1 and the EA is not halfword-aligned.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lhau RT, D(RA)

43	RT	RA	D
0	6	11	16
			31

$EA \leftarrow (RA|0) + EXTS(D)$
 $(RA) \leftarrow EA$
 $(RT) \leftarrow EXTS(MS(EA,2))$

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 when the RA field is 0 and is the contents of register RA otherwise. The EA is placed into register RA.

The halfword at the EA is sign-extended to 32 bits and placed into register RT.

Registers Altered

- RA
- RT

Invalid Instruction Forms

- $RA = RT$
- $RA = 0$

Exceptions

An alignment exception occurs if $MSR[LE] = 1$ and the EA is not halfword-aligned.

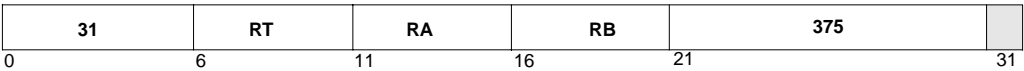
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lhaux

Load Halfword Algebraic with Update Indexed

lhaux RT, RA, RB



```
EA ← (RA|0) + (RB)
(RA) ← EA
(RT) ← EXTS(MS(EA,2))
```

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise. The EA is placed into register RA.

The halfword at the EA is sign-extended to 32 bits and placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RA
- RT

Invalid Instruction Forms

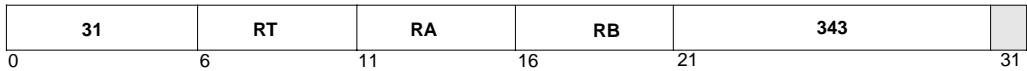
- Reserved fields
- RA = RT
- RA = 0

Exceptions

An alignment exception occurs if MSR[LE] = 1 and the EA is not halfword-aligned.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lhax RT, RA, RB

$$EA \leftarrow (RA \ll 0) + (RB)$$

$$(RT) \leftarrow \text{EXTS}(\text{MS}(EA, 2))$$

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The halfword at the EA is sign-extended to 32 bits and placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields

Exceptions

An alignment exception occurs if MSR[LE] = 1 and the EA is not halfword-aligned.

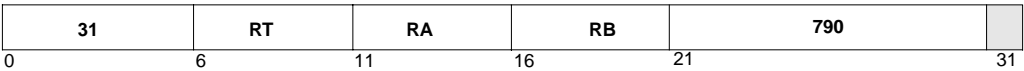
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lhbrx

Load Halfword Byte-Reverse Indexed

lhbrx RT, RA, RB



$EA \leftarrow (RA[0] + (RB))$
 $(RT) \leftarrow {}^{16}0 \parallel MS(EA + 1, 1) \parallel MS(EA, 1)$

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The halfword at the EA is byte-reversed. The resulting halfword is extended to 32 bits by concatenating 16 0-bits to its left. The result is placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields

9

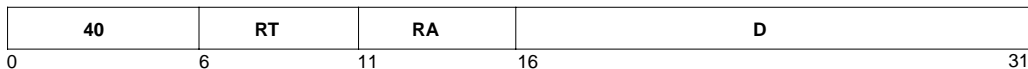
Exceptions

An alignment exception occurs if MSR[LE] = 1 and the EA is not halfword-aligned.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lhz RT, D(RA)



$$EA \leftarrow (RA|0) + \text{EXTS}(D)$$

$$(RT) \leftarrow {}^{16}0 \parallel \text{MS}(EA, 2)$$

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The halfword at the EA is extended to 32 bits by concatenating 16 0-bits to its left. The result is placed into register RT.

Registers Altered

- RT

Exceptions

An alignment exception occurs if MSR[LE] = 1 and the EA is not halfword-aligned.

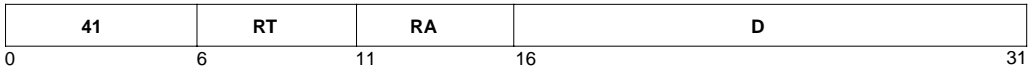
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lhzu

Load Halfword and Zero with Update

lhzu RT, D(RA)



$EA \leftarrow (RA|0) + EXTS(D)$
 $(RA) \leftarrow EA$
 $(RT) \leftarrow {}^{16}0 \parallel MS(EA,2)$

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise. The EA is placed into register RA.

The halfword at the EA is extended to 32 bits by concatenating 16 0-bits to its left. The result is placed into register RT.

Registers Altered

- RA
- RT

Invalid Instruction Forms

- RA = RT
- RA = 0

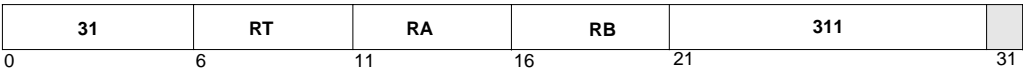
Exceptions

An alignment exception occurs if MSR[LE] = 1 and the EA is not halfword-aligned.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lhzux RT, RA, RB



```
EA ← (RA|0) + (RB)
(RA) ← EA
(RT) ← 160 || MS(EA,2)
```

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise. The EA is placed into register RA.

The halfword at the EA is extended to 32 bits by concatenating 16 0-bits to its left. The result is placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RA
- RT

Invalid Instruction Forms

- Reserved fields
- RA = RT
- RA = 0

Exceptions

An alignment exception occurs if MSR[LE] = 1 and the EA is not halfword-aligned.

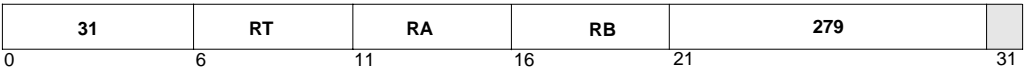
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lhzx

Load Halfword and Zero Indexed

lhzx RT, RA, RB



$EA \leftarrow (RA \ll 0) + (RB)$
 $(RT) \leftarrow {}^{160} || MS(EA,2)$

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The halfword at the EA is extended to 32 bits by concatenating 16 0-bits to its left. The result is placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields

9

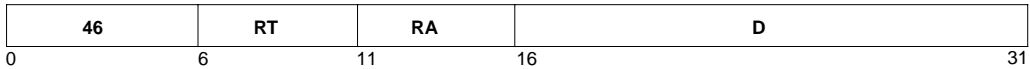
Exceptions

An alignment exception occurs if MSR[LE] = 1 and the EA is not halfword-aligned.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

l_{mw} RT, D(RA)



```

EA ← (RA|0) + EXTS(D)
r ← RT
do while r ≤ 31
    if ((r ≠ RA) ∨ (r = 31)) then
        (GPR(r)) ← MS(EA,4)
    r ← r + 1
    EA ← EA + 4

```

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field in the instruction to 32 bits. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

A series of consecutive words starting at the EA are loaded into a set of consecutive GPRs, starting with register RT and continuing to and including GPR(31). Register RA is not altered by this instruction (unless RA is GPR(31), which is an invalid form of this instruction). The word which would have been placed into register RA is discarded.

Registers Altered

- RT through GPR(31).

Invalid Instruction Forms

- RA is in the range of registers to be loaded, including the case RA = RT = 0.

Exceptions

If MSR[LE] = 1, an alignment exception occurs.

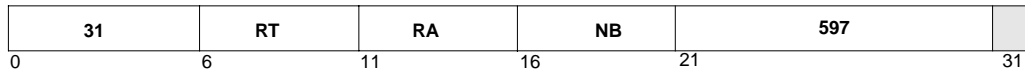
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lswi

Load String Word Immediate

lswi RT, RA, NB



```

EA ← (RA|0)
if NB = 0 then
    CNT ← 32
else
    CNT ← NB
n ← CNT
RFINAL ← ((RT + CEIL(CNT/4) - 1) % 32)
r ← RT - 1
i ← 0
do while n > 0
    if i = 0 then
        r ← r + 1
        if r = 32 then
            r ← 0
        if ((r ≠ RA) ∨ (r = RFINAL)) then
            (GPR(r)) ← 0
        if ((r ≠ RA) ∨ (r = RFINAL)) then
            (GPR(r)i:i+7) ← MS(EA,1)
    i ← i + 8
    if i = 32 then
        i ← 0
    EA ← EA + 1
    n ← n - 1

```

9

An effective address (EA) is determined by the RA field. If the RA field contains 0, the EA is 0. Otherwise, the EA is the contents of register RA.

The NB field specifies the byte count CNT. If the NB field contains 0, the byte count is CNT = 32. Otherwise, the byte count is CNT = NB.

A series of CNT consecutive bytes in main storage, starting at the EA, are loaded into CEIL(CNT/4) consecutive GPRs, four bytes per GPR, until the byte count is exhausted. Bytes are loaded into GPRs; the byte at the lowest address is loaded into the most significant byte. Bits to the right of the last byte loaded into the last GPR are set to 0.

The set of loaded GPRs starts at register RT, continues consecutively through GPR(31), wraps to register 0, loading until the byte count is exhausted, which occurs in register R_{FINAL}. Register RA is not altered (unless RA = R_{FINAL}, an invalid form of this instruction). Bytes which would have been loaded into register RA are discarded.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT and subsequent GPRs as described above.

Invalid Instruction Forms

- Reserved fields
- RA is in the range of registers to be loaded
- RA = RT = 0

Exceptions

If MSR[LE] = 1, an alignment exception occurs.

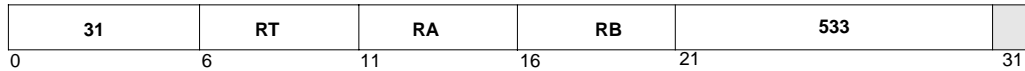
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lswx

Load String Word Indexed

lswx RT, RA, RB



```

EA ← (RA[0] + (RB)
CNT ← XER[TBC]
n ← CNT
RFINAL ← ((RT + CEIL(CNT/4) – 1) % 32)
r ← RT – 1
i ← 0
do while n > 0
    if i = 0 then
        r ← r + 1
        if r = 32 then
            r ← 0
        if (((r ≠ RA) ∧ (r ≠ RB)) ∨ (r = RFINAL)) then
            (GPR(r)) ← 0
        if (((r ≠ RA) ∧ (r ≠ RB)) ∨ (r = RFINAL)) then
            (GPR(r)i:i+7) ← MS(EA,1)
        i ← i + 8
    if i = 32 then
        i ← 0
    EA ← EA + 1
    n ← n – 1

```

9

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

A byte count CNT is obtained from XER[TBC].

A series of CNT consecutive bytes in main storage, starting at the EA, are loaded into CEIL(CNT/4) consecutive GPRs, four bytes per GPR, until the byte count is exhausted. Bytes are loaded into GPRs; the byte having the lowest address is loaded into the most significant byte. Bits to the right of the last byte loaded in the last GPR used are set to 0.

The set of consecutive GPRs loaded starts at register RT, continues through GPR(31), and wraps to register 0, loading until the byte count is exhausted, which occurs in register R_{FINAL}. Register RA is not altered (unless RA = R_{FINAL}, which is an invalid form of this instruction). Register RB is not altered (unless RB = R_{FINAL}, which is an invalid form of this instruction). Bytes which would have been loaded into registers RA or RB are discarded.

If XER[TBC] is 0, the byte count is 0 and the contents of register RT are undefined.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT and subsequent GPRs as described above.

Invalid Instruction Forms

- Reserved fields
- RA or RB is in the range of registers to be loaded.
- RA = RT = 0

Programming Note

If XER[TBC] = 0, the contents of register RT are unchanged and **lswx** is treated as a no-op.

The PowerPC Architecture states that, if XER[TBC] = 0 and if the EA is such that a precise data exception would normally occur (if not for the zero length), **lswx** is treated as a no-op and the precise exception will not occur. Data storage exceptions and alignment exceptions are examples of precise data exceptions.

However, the PowerPC Architecture makes no statement regarding imprecise exceptions related to **lswx** with XER[TBC] = 0. The PPC401A1 generates an imprecise exception (machine check) on this instruction when all of the following conditions are true:

- The address is cacheable
- The address is passed to the data cache
- The address misses in the data cache (resulting in a line fill request)
- The address encounters some form of bus error

Exceptions

If MSR[LE] = 1, an alignment exception occurs.

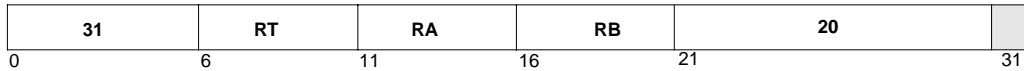
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lwarx

Load Word and Reserve Indexed

lwarx RT, RA, RB



$EA \leftarrow (RA|0) + (RB)$
 $RESERVE \leftarrow 1$
 $(RT) \leftarrow MS(EA,4)$

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The word at the EA is placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Execution of the **lwarx** instruction sets the reservation bit.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields

Programming Note

lwarx and the **stwcx.** instruction should be paired in a loop, as shown in the following example, to create the effect of an atomic operation to a memory area used as a semaphore between asynchronous processes. Only **lwarx** can set the reservation bit to 1. **stwcx.** sets the reservation bit to 0 upon its completion, whether or not **stwcx.** sent (RS) to memory. CR[CR0]_{EQ} must be examined to determine whether (RS) was sent to memory.

```
loop: lwarx          # read the semaphore from memory; set reservation
      "alter"       # change the semaphore bits in register as required
      stwcx.        # attempt to store semaphore; reset reservation
      bne loop      # an asynchronous process has intervened; try again
```

If the asynchronous process in the code example had paired **lwarx** with a store other than **stwcx.**, the reservation bit would not have been cleared in the asynchronous process, and the code example would have overwritten the semaphore.

Exceptions

An alignment exception occurs if the EA is not word-aligned.

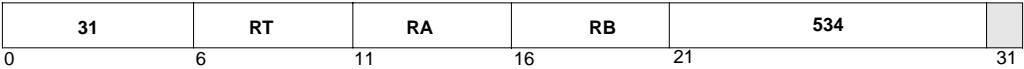
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lwbrx

Load Word Byte-Reverse Indexed

lwbrx RT, RA, RB



$EA \leftarrow (RA|0) + (RB)$
 $(RT) \leftarrow MS(EA+3,1) \parallel MS(EA+2,1) \parallel MS(EA+1,1) \parallel MS(EA,1)$

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The word at the EA is byte-reversed: the least significant byte becomes the most significant byte, the next least significant byte becomes the next most significant byte, and so on. The resulting word is placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields

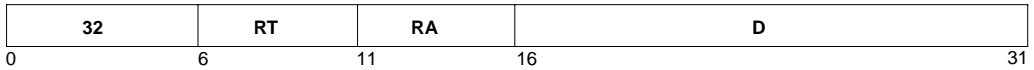
Exceptions

An alignment exception occurs if MSR[LE] = 1 and the EA is not word-aligned.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lwz RT, D(RA)



$$EA \leftarrow (RA|0) + \text{EXTS}(D)$$

$$(RT) \leftarrow \text{MS}(EA, 4)$$

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The word at the EA is placed into register RT.

Registers Altered

- RT

Exceptions

An alignment exception occurs if MSR[LE] = 1 and the EA is not word-aligned.

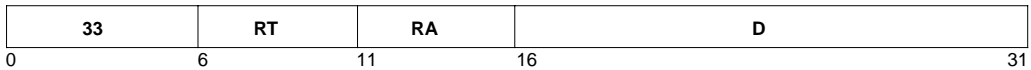
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lwzu

Load Word and Zero with Update

lwzu RT, D(RA)



$EA \leftarrow (RA|0) + EXTS(D)$
 $(RA) \leftarrow EA$
 $(RT) \leftarrow MS(EA,4)$

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise. The EA is placed into register RA.

The word at the EA is placed into register RT.

Registers Altered

- RA
- RT

Invalid Instruction Forms

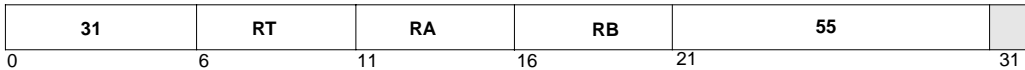
- $RA = RT$
- $RA = 0$

Exceptions

An alignment exception occurs if $MSR[LE] = 1$ and the EA is not word-aligned.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lwzux RT, RA, RB

$EA \leftarrow (RA|0) + (RB)$
 $(RA) \leftarrow EA$
 $(RT) \leftarrow MS(EA,4)$

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise. The EA is placed into register RA.

The word at the EA is placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RA
- RT

Invalid Instruction Forms

- Reserved fields
- $RA = RT$
- $RA = 0$

Exceptions

An alignment exception occurs if $MSR[LE] = 1$ and the EA is not word-aligned.

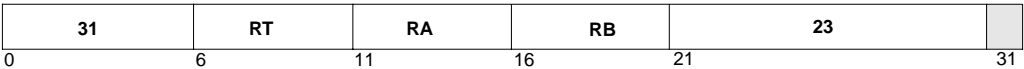
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lwzx

Load Word and Zero Indexed

lwzx RT, RA, RB



$EA \leftarrow (RA|0) + (RB)$
 $(RT) \leftarrow MS(EA,4)$

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

The word at the EA is placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields

9

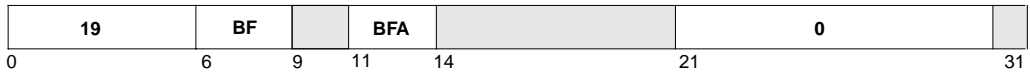
Exceptions

An alignment exception occurs if MSR[LE] = 1 and the EA is not word-aligned.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

mcrf BF, BFA

 $m \leftarrow \text{BFA}$ $n \leftarrow \text{BF}$ $(\text{CR}[\text{CR}_n]) \leftarrow (\text{CR}[\text{CR}_m])$

The contents of the CR field specified by the BFA field are placed into the CR field specified by the BF field.

Registers Altered

- $\text{CR}[\text{CR}_n]$ where n is specified by the BF field.

Invalid Instruction Forms

- Reserved fields

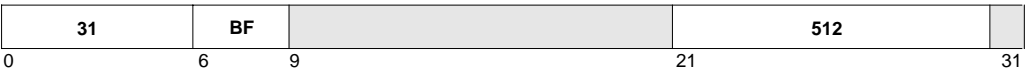
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

mcrxr

Move to Condition Register from XER

mcrxr **BF**



$n \leftarrow \text{BF}$
 $(\text{CR}[\text{CR}_n]) \leftarrow \text{XER}_{0:3}$
 $\text{XER}_{0:3} \leftarrow 0$

The contents of $\text{XER}_{0:3}$ are placed into the CR field specified by the BF field. $\text{XER}_{0:3}$ are then set to 0.

This transfer is positional, by bit number, so the mnemonics associated with each bit are changed. See the following table for clarification.

Bit	XER Usage	CR Usage
0	SO	LT
1	OV	GT
2	CA	EQ
3	Reserved	SO

9 If instruction bit 31 contains 1, the contents of $\text{CR}[\text{CR}_0]$ are undefined.

Registers Altered

- $\text{CR}[\text{CR}_n]$ where n is specified by the BF field.
- $\text{XER}[\text{SO}, \text{OV}, \text{CA}]$

Invalid Instruction Forms

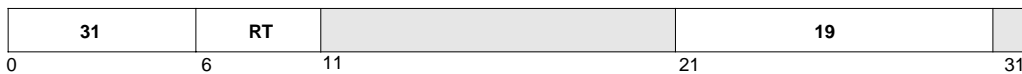
- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

mfcrr

RT



$$(RT) \leftarrow (CR)$$

The contents of the CR are placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields

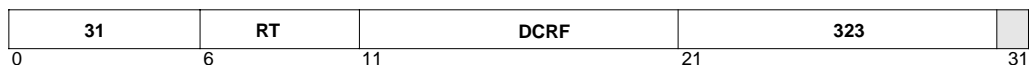
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

mfdcr

Move from Device Control Register

mfdcr RT, DCRN



$$\text{DCRN} \leftarrow \text{DCRF}_{5:9} \parallel \text{DCRF}_{0:4}$$

$$(\text{RT}) \leftarrow (\text{DCR}(\text{DCRN}))$$

The contents of the DCR specified by the DCRF field are placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields
- Invalid DCRF values

Programming Note

Execution of this instruction is privileged.

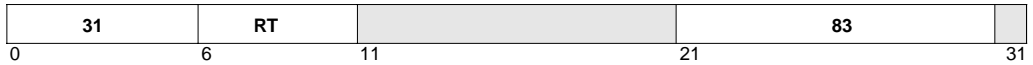
9

The DCR number (DCRN) specified in the assembler language coding of the **mfdcr** instruction refers to a DCR number. The assembler handles the unusual register number encoding to generate the DCRF field.

Architecture Note

Programs using this instruction are not portable to PowerPC implementations that do not implement the IBM PowerPC Embedded Environment.

mfmsr RT

 $(RT) \leftarrow (MSR)$

The contents of the MSR are placed into register RT.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields

Programming Note

Execution of this instruction is privileged.

Architecture Note

This instruction is part of the PowerPC Operating Environment Architecture.

mfspr

Move From Special Purpose Register

mfspr RT, SPRN



$SPRN \leftarrow SPRF_{5:9} \parallel SPRF_{0:4}$
 $(RT) \leftarrow (SPR(SPRN))$

The contents of the SPR specified by the SPRF field are placed into register RT. See Table 10-2, “Special Purpose Registers,” on p. 10-3 for a listing of SPR mnemonics and corresponding SPRN and SPRF values.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RT

Invalid Instruction Forms

- Reserved fields
- Invalid SPRF values

Programming Note

9

Execution of this instruction is privileged if instruction bit 11 contains 1. See Section 2.8.3, “Privileged SPRs,” on p. 2-39 for more information.

The SPR number (SPRN) specified in the assembler language coding of the **mfspr** instruction refers to an SPR number (see Table 10-2, “Special Purpose Registers,” on p. 10-3 for a list of SPRN values). The assembler handles the unusual register number encoding to generate the SPRF field. Also, see Section 2.8.3, “Privileged SPRs,” on p. 2-39 for information about privileged SPRs.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

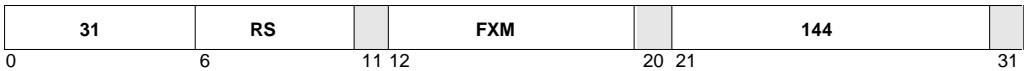
Table 9-20. Extended Mnemonics for mfspir

Mnemonic	Operands	Function	Other Registers Changed
mfcdbcr mfctr mfdac1 mfdbcr mfdbsr mfdbsrs mfdccr mfdcwr mfdear mfesr mfevpr mfiac1 mficcr mficdbdr mflr mfpit mfpvr mfsgr mfsler mfsprg0 mfsprg1 mfsprg2 mfsprg3 mfsrr0 mfsrr1 mfsrr2 mfsrr3 mftbhi mftbhu mftblo mftblu mftcr mftsrr mfxer	RT	Move from special purpose register SPRN. <i>Extended mnemonic for</i> mfspir RT,SPRN See Table 10-2, "Special Purpose Registers," on p. 10-3 for a list of valid SPRN values.	

mftb

Move to Condition Register Fields

mtcrf FXM, RS



$$\text{mask} \leftarrow {}^4(\text{FXM}_0) \parallel {}^4(\text{FXM}_1) \parallel \dots \parallel {}^4(\text{FXM}_6) \parallel {}^4(\text{FXM}_7)$$
$$(\text{CR}) \leftarrow ((\text{RS}) \wedge \text{mask}) \vee ((\text{CR}) \wedge \neg \text{mask})$$

Some or all of the contents of register RS are placed into the CR as specified by the FXM field.

Each bit in the FXM field controls the copying of 4 bits in register RS into the corresponding bits in the CR. The correspondence between the bits in the FXM field and the bit copying operation is shown in the following table:

FXM Bit Number	Bits Controlled
0	0:3
1	4:7
2	8:11
3	12:15
4	16:19
5	20:23
6	24:27
7	28:31

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- CR

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

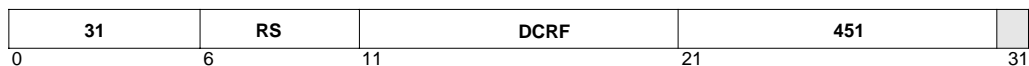
Table 9-21. Extended Mnemonics for mtcrf

Mnemonic	Operands	Function	Other Registers Changed
mtcr	RS	Move to Condition Register. <i>Extended mnemonic for mtcrf 0xFF,RS</i>	

mtdcr

Move To Device Control Register

mtdcr DCRN, RS



$$\text{DCRN} \leftarrow \text{DCRF}_{5:9} \parallel \text{DCRF}_{0:4}$$

$$(\text{DCR}(\text{DCRN})) \leftarrow (\text{RS})$$

The contents of register RS are placed into the DCR specified by the DCRF field.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- DCR(DCRN)

Invalid Instruction Forms

- Reserved fields
- Invalid DCRF values

Programming Note

Execution of this instruction is privileged.

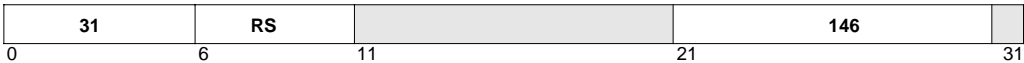
9

The DCR number (DCRN) specified in the assembler language coding of the **mtdcr** instruction refers to a DCR number. The assembler handles the unusual register number encoding to generate the DCRF field.

Architecture Note

Programs using this instruction are not portable to PowerPC implementations that do not implement the IBM PowerPC Embedded Environment.

mtmsr RS



(MSR) ← (RS)

The contents of register RS are placed into the MSR.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- MSR

Invalid Instruction Forms

- Reserved fields

Programming Note

The **mtmsr** instruction is privileged and execution synchronizing.

Architecture Note

This instruction is part of the PowerPC Operating Environment Architecture.

mtspr

Move To Special Purpose Register

mtspr SPRN, RS



$$\text{SPRN} \leftarrow \text{SPRF}_{5:9} \parallel \text{SPRF}_{0:4}$$
$$(\text{SPR}(\text{SPRN})) \leftarrow (\text{RS})$$

The contents of register RS are placed into register RT. See Table 10-2, “Special Purpose Registers,” on p. 10-3 for a listing of SPR mnemonics and corresponding SPRN and SPRF values.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- SPR(SPRN)

Invalid Instruction Forms

- Reserved fields
- Invalid SPRF values

Programming Note

9

Execution of this instruction is privileged if instruction bit 11 is a 1. See Section 2.8.3 (Privileged SPRs) on p. 2-39 for more information.

The SPR number (SPRN) specified in the assembler language coding of the **mtspr** instruction refers to an SPR number (see Table 10-2, “Special Purpose Registers,” on p. 10-3 for a list of SPRN values). The assembler handles the unusual register number encoding to generate the SPRF field. Also, see Section 2.8.3, “Privileged SPRs,” on p. 2-39 for information about privileged SPRs.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

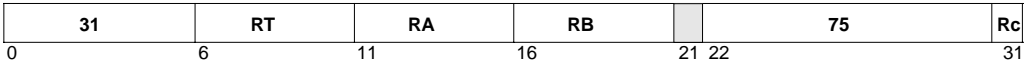
Table 9-22. Extended Mnemonics for mtspr

Mnemonic	Operands	Function	Other Registers Changed
mtcdbcr mtctr mtdac1 mtdbcr mtdbsr mtdccr mtdcwr mtdear mtesr mtevpr mtiac1 mticcr mticdbdr mtlr mtpit mtpvr mtsgr mtsler mtsprg0 mtsprg1 mtsprg2 mtsprg3 mtsrr0 mtsrr1 mtsrr2 mtsrr3 mttbhi mttblo mttcr mttsr mtxer	RS	<p>Move to special purpose register SPRN.</p> <p><i>Extended mnemonic for</i> mtspr SPRN,RS</p> <p>See Table 10-2, “Special Purpose Registers,” on p. 10-3 for a list of valid SPRN values.</p>	

mulhw

Multiply High Word

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



$$\text{prod}_{0:63} \leftarrow (\text{RA}) \times (\text{RB}) \text{ (signed)}$$
$$(\text{RT}) \leftarrow \text{prod}_{0:31}$$

The 64-bit signed product of registers RA and RB is formed. The most significant 32 bits of the result is placed into register RT.

Registers Altered

- RT
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1

Programming Note

The most significant 32 bits of the product, unlike the least significant 32 bits, may differ depending on whether the registers RA and RB are interpreted as signed or unsigned quantities. The **mulhw** instruction generates the correct result when these operands are interpreted as signed quantities. The **mulhwu** instruction generates the correct result when these operands are interpreted as unsigned quantities.

Architecture Note

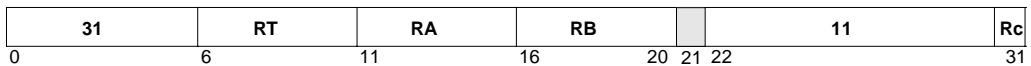
This instruction is part of the PowerPC User Instruction Set Architecture.

mulhwu

Multiply High Word Unsigned

mulhwu	RT, RA, RB
mulhwu.	RT, RA, RB

Rc=0
Rc=1


$$\begin{aligned} \text{prod}_{0:63} &\leftarrow (\text{RA}) \times (\text{RB}) \text{ (unsigned)} \\ (\text{RT}) &\leftarrow \text{prod}_{0:31} \end{aligned}$$

The 64-bit unsigned product of registers RA and RB is formed. The most significant 32 bits of the result are placed into register RT.

Registers Altered

- RT
- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1

Programming Note

The most significant 32 bits of the product, unlike the least significant 32 bits, may differ depending on whether the registers RA and RB are interpreted as signed or unsigned quantities. The **mulhw** instruction generates the correct result when these operands are interpreted as signed quantities. The **mulhbw** instruction generates the correct result when these operands are interpreted as unsigned quantities.

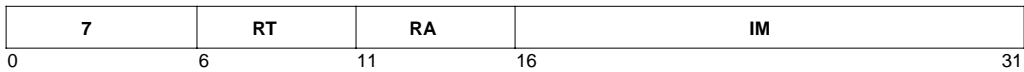
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

mulli

Multiply Low Immediate

mulli RT, RA, IM



$$\text{prod}_{0:47} \leftarrow (\text{RA}) \times \text{EXTS}(\text{IM}) \text{ (signed)}$$
$$(\text{RT}) \leftarrow \text{prod}_{16:47}$$

The 48-bit product of register RA and the sign-extended IM field is formed. Both register RA and the IM field are interpreted as signed quantities. The least significant 32 bits of the product are placed into register RT.

Registers Altered

- RT

Programming Note

The least significant 32 bits of the product are correct, regardless of whether register RA and field IM are interpreted as signed or unsigned numbers.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

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

31	RT	RA	RB	OE	235	Rc
0	6	11	16	21	22	31

$\text{prod}_{0:63} \leftarrow (\text{RA}) \times (\text{RB}) \text{ (signed)}$
 $(\text{RT}) \leftarrow \text{prod}_{32:63}$

The 64-bit signed product of register RA and register RB is formed. The least significant 32 bits of the result is placed into register RT.

If the signed product cannot be represented in 32 bits and OE=1, XER[SO, OV] are set to 1.

Registers Altered

- RT
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1
- XER[SO, OV] if OE=1

Programming Note

The least significant 32 bits of the product are correct, regardless of whether register RA and register RB are interpreted as signed or unsigned numbers. The overflow indication is correct only if the operands are regarded as signed numbers.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

nand

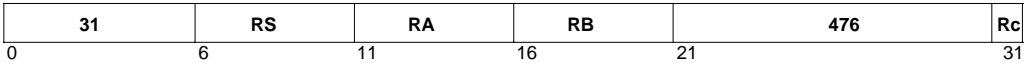
NAND

nand

nand.

RA, RS, RB
RA, RS, RB

Rc=0
Rc=1



$(RA) \leftarrow \neg((RS) \wedge (RB))$

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

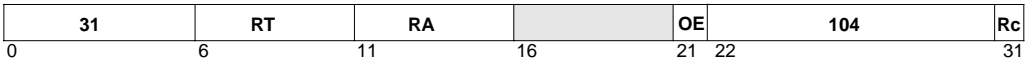
Registers Altered

- RA
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

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



$(RT) \leftarrow \neg(RA) + 1$

The twos complement of the contents of register RA are placed into register RT.

Registers Altered

- RT
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1
- XER[CA, SO, OV] if OE=1

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

nor

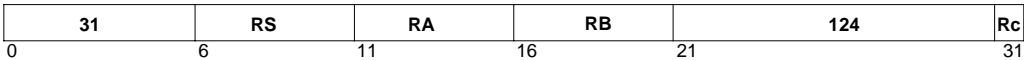
NOR

nor

nor.

RA, RS, RB
RA, RS, RB

Rc=0
Rc=1



$(RA) \leftarrow \neg((RS) \vee (RB))$

The contents of register RS is ORed with the contents of register RB; the ones complement of the result is placed into register RA.

Registers Altered

- RA
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1

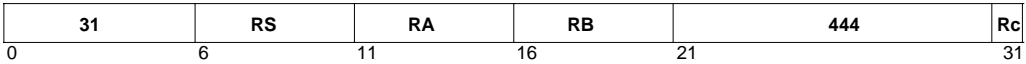
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-23. Extended Mnemonics for nor, nor.

Mnemonic	Operands	Function	Other Registers Changed
not	RA, RS	Complement register. (RA) ← ¬(RS) <i>Extended mnemonic for nor RA,RS,RS</i>	
not.		<i>Extended mnemonic for nor. RA,RS,RS</i>	CR[CR0]

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; the result is placed into register RA.

Registers Altered

- RA
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

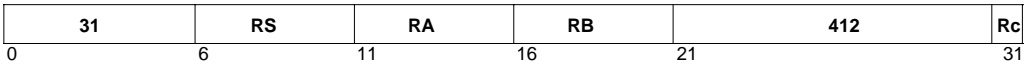
Table 9-24. Extended Mnemonics for or, or.

Mnemonic	Operands	Function	Other Registers Changed
mr	RT, RS	Move register. (RT) ← (RS) <i>Extended mnemonic for or RT,RS,RS</i>	
mr.		<i>Extended mnemonic for or. RT,RS,RS</i>	CR[CR0]

orc

OR with Complement

orc RA, RS, RB Rc=0
orc. RA, RS, RB Rc=1



$$(RA) \leftarrow (RS) \vee \neg(RB)$$

The contents of register RS is ORed with the ones complement of the contents of register RB; the result is placed into register RA.

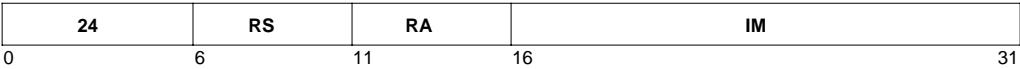
Registers Altered

- RA
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

ori RA, RS, IM



$(RA) \leftarrow (RS) \vee (^{16}0 \parallel IM)$

The IM field is extended to 32 bits by concatenating 16 0-bits on the left. Register RS is ORed with the extended IM field; the result is placed into register RA.

Registers Altered

- RA

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

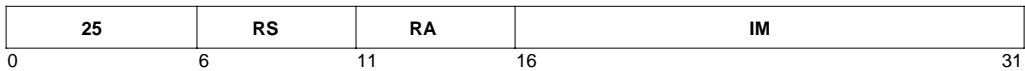
Table 9-25. Extended Mnemonics for ori

Mnemonic	Operands	Function	Other Registers Changed
nop		Preferred no-op; triggers optimizations based on no-ops. <i>Extended mnemonic for ori 0,0,0</i>	

oris

OR Immediate Shifted

oris RA, RS, IM



$(RA) \leftarrow (RS) \vee (IM \parallel ^{16}0)$

The IM Field is extended to 32 bits by concatenating 16 0-bits on the right. Register RS is ORed with the extended IM field and the result is placed into register RA.

Registers Altered

- RA

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

rfci

(PC) \leftarrow (SRR2)

(MSR) \leftarrow (SRR3)

The program counter (PC) is restored with the contents of SRR2 and the MSR is restored with the contents of SRR3.

Instruction execution returns to the address contained in the PC.

Registers Altered

- MSR

Programming Note

Execution of this instruction is privileged and context-synchronizing.

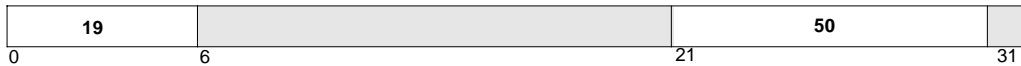
Architecture Note

Programs using this instruction are not portable to PowerPC implementations that do not implement the IBM PowerPC Embedded Environment.

rfi

Return From Interrupt

rfi



$(PC) \leftarrow (SRR0)$
 $(MSR) \leftarrow (SRR1)$

The program counter (PC) is restored with the contents of SRR0 and the MSR is restored with the contents of SRR1.

Instruction execution returns to the address contained in the PC.

Registers Altered

- MSR

Invalid Instruction Forms

- Reserved fields

Programming Note

Execution of this instruction is privileged and context-synchronizing.

9

Architecture Note

This instruction is part of the PowerPC Operating Environment Architecture.

rlwimi RA, RS, SH, MB, ME Rc=0
rlwimi. RA, RS, SH, MB, ME Rc=1

20	RS	RA	SH	MB	ME	Rc
0	6	11	16	21	26	31

$r \leftarrow \text{ROTL}((RS), SH)$
 $m \leftarrow \text{MASK}(MB, ME)$
 $(RA) \leftarrow (r \wedge m) \vee ((RA) \wedge \neg m)$

The contents of register RS are rotated left by the number of bit positions specified in the SH field. A mask is generated, having 1-bits starting at the bit position specified in the MB field and ending in the bit position specified by the ME field, with 0-bits elsewhere.

If the starting point of the mask is at a higher bit position than the ending point, the 1-bits portion of the mask wraps from the highest bit position back around to the lowest. The rotated data is inserted into register RA, in positions corresponding to the bit positions in the mask that contain a 1-bit.

Registers Altered

- RA
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-26. Extended Mnemonics for rlwimi, rlwimi.

Mnemonic	Operands	Function	Other Registers Changed
inslwi	RA, RS, n, b	Insert from left immediate. ($n > 0$) $(RA)_{b:b+n-1} \leftarrow (RS)_{0:n-1}$ <i>Extended mnemonic for</i> rlwimi RA,RS,32-b,b,b+n-1	
inslwi.		<i>Extended mnemonic for</i> rlwimi. RA,RS,32-b,b,b+n-1	CR[CR0]
insrwi	RA, RS, n, b	Insert from right immediate. ($n > 0$) $(RA)_{b:b+n-1} \leftarrow (RS)_{32-n:31}$ <i>Extended mnemonic for</i> rlwimi RA,RS,32-b-n,b,b+n-1	
insrwi.		<i>Extended mnemonic for</i> rlwimi. RA,RS,32-b-n,b,b+n-1	CR[CR0]

rlwinm

Rotate Left Word Immediate then AND with Mask

rlwinm RA, RS, SH, MB, ME Rc=0
rlwinm. RA, RS, SH, MB, ME Rc=1

21	RS	RA	SH	MB	ME	Rc
0	6	11	16	21	26	31

$r \leftarrow \text{ROTL}((RS), SH)$
 $m \leftarrow \text{MASK}(MB, ME)$
 $(RA) \leftarrow r \wedge m$

The contents of register RS are rotated left by the number of bit positions specified in the SH field. A mask is generated, having 1-bits starting at the bit position specified in the MB field and ending in the bit position specified by the ME field with 0-bits elsewhere.

If the starting point of the mask is at a higher bit position than the ending point, the 1-bits portion of the mask wraps from the highest bit position back around to the lowest. The rotated data is ANDed with the generated mask; the result is placed into register RA.

Registers Altered

- RA
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-27. Extended Mnemonics for rlwinm, rlwinm.

Mnemonic	Operands	Function	Other Registers Changed
clrlwi	RA, RS, n	Clear left immediate. ($n < 32$) $(RA)_{0:n-1} \leftarrow {}^n0$ <i>Extended mnemonic for</i> rlwinm RA,RS,0,n,31	
clrlwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,0,n,31	CR[CR0]
clrlslwi	RA, RS, b, n	Clear left and shift left immediate. $(n \leq b < 32)$ $(RA)_{b-n:31-n} \leftarrow (RS)_{b:31}$ $(RA)_{32-n:31} \leftarrow {}^n0$ $(RA)_{0:b-n-1} \leftarrow {}^{b-n}0$ <i>Extended mnemonic for</i> rlwinm RA,RS,n,b-n,31-n	
clrlslwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,n,b-n,31-n	CR[CR0]

Table 9-27. Extended Mnemonics for rlwinm, rlwinm. (cont.)

Mnemonic	Operands	Function	Other Registers Changed
clrrwi	RA, RS, n	Clear right immediate. ($n < 32$) $(RA)_{32-n:31} \leftarrow {}^n0$ <i>Extended mnemonic for</i> rlwinm RA,RS,0,0,31-n	
clrrwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,0,0,31-n	CR[CR0]
extlwi	RA, RS, n, b	Extract and left justify immediate. ($n > 0$) $(RA)_{0:n-1} \leftarrow (RS)_{b:b+n-1}$ $(RA)_{n:31} \leftarrow {}^{32-n}0$ <i>Extended mnemonic for</i> rlwinm RA,RS,b,0,n-1	
extlwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,b,0,n-1	CR[CR0]
extrwi	RA, RS, n, b	Extract and right justify immediate. $(n > 0)$ $(RA)_{32-n:31} \leftarrow (RS)_{b:b+n-1}$ $(RA)_{0:31-n} \leftarrow {}^{32-n}0$ <i>Extended mnemonic for</i> rlwinm RA,RS,b+n,32-n,31	
extrwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,b+n,32-n,31	CR[CR0]
rotlwi	RA, RS, n	Rotate left immediate. $(RA) \leftarrow \text{ROTL}((RS), n)$ <i>Extended mnemonic for</i> rlwinm RA,RS,n,0,31	
rotlwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,n,0,31	CR[CR0]
rotrwi	RA, RS, n	Rotate right immediate. $(RA) \leftarrow \text{ROTL}((RS), 32-n)$ <i>Extended mnemonic for</i> rlwinm RA,RS,32-n,0,31	
rotrwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,32-n,0,31	CR[CR0]

rlwinm

Rotate Left Word Immediate then AND with Mask

Table 9-27. Extended Mnemonics for rlwinm, rlwinm. (cont.)

Mnemonic	Operands	Function	Other Registers Changed
slwi	RA, RS, n	Shift left immediate. ($n < 32$) $(RA)_{0:31-n} \leftarrow (RS)_{n:31}$ $(RA)_{32-n:31} \leftarrow {}^n0$ <i>Extended mnemonic for</i> rlwinm RA,RS,n,0,31-n	
slwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,n,0,31-n	CR[CR0]
srwi	RA, RS, n	Shift right immediate. ($n < 32$) $(RA)_{n:31} \leftarrow (RS)_{0:31-n}$ $(RA)_{0:n-1} \leftarrow {}^n0$ <i>Extended mnemonic for</i> rlwinm RA,RS,32-n,n,31	
srwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,32-n,n,31	CR[CR0]

rlwnm RA, RS, RB, MB, ME Rc=0
rlwnm. RA, RS, RB, MB, ME Rc=1

23	RS	RA	RB	MB	ME	Rc
0	6	11	16	21	26	31

$r \leftarrow \text{ROTL}((RS), (RB)_{27:31})$
 $m \leftarrow \text{MASK}(MB, ME)$
 $(RA) \leftarrow r \wedge m$

The contents of register RS are rotated left by the number of bit positions specified by the contents of register RB_{27:31}. A mask is generated, having 1-bits starting at the bit position specified in the MB field and ending in the bit position specified by the ME field with 0-bits elsewhere.

If the starting point of the mask is at a higher bit position than the ending point, the ones portion of the mask wraps from the highest bit position back to the lowest. The rotated data is ANDed with the generated mask and the result is placed into register RA.

Registers Altered

- RA
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

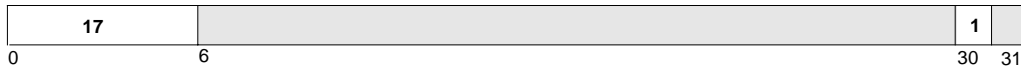
Table 9-28. Extended Mnemonics for rlwnm, rlwnm.

Mnemonic	Operands	Function	Other Registers Changed
rotlw	RA, RS, RB	Rotate left. $(RA) \leftarrow \text{ROTL}((RS), (RB)_{27:31})$ <i>Extended mnemonic for</i> rlwnm RA,RS,RB,0,31	
rotlw.		<i>Extended mnemonic for</i> rlwnm. RA,RS,RB,0,31	CR[CR0]

SC

System Call

sc



$(SRR1) \leftarrow (MSR)$
 $(SRR0) \leftarrow (PC)$
 $PC \leftarrow EVPR_{0:15} \parallel 0x0C00$
 $(MSR[WE, EE, PR]) \leftarrow 0$
 $(MSR[LE]) \leftarrow (MSR[ILE])$

A system call exception is generated. The contents of the MSR are copied into SRR1 and (4 + address of **sc** instruction) is placed into SRR0.

The program counter (PC) is then loaded with the exception vector address. The exception vector address is calculated by concatenating the high halfword of the Exception Vector Prefix Register (EVPR) to the left of 0x0C00.

The MSR[WE, EE, PR] bits are set to 0, and MSR[ILE] is copied to MSR[LE].

Program execution continues at the new address in the PC.

The **sc** instruction is context synchronizing.

9

Registers Altered

- SRR0
- SRR1
- MSR[WE, EE, PR, LE]

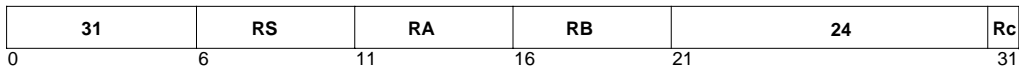
Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

slw RA, RS, RB Rc=0
slw. RA, RS, RB Rc=1



$n \leftarrow (RB)_{27:31}$
 $r \leftarrow ROTL((RS), n)$
if $(RB)_{26} = 0$ then
 $m \leftarrow MASK(0, 31 - n)$
else
 $m \leftarrow {}^{32}0$
 $(RA) \leftarrow r \wedge m$

The contents of register RS are shifted left by the number of bits specified by the contents of register RB_{27:31}. Bits shifted left out of the most significant bit are lost, and 0-bits fill vacated bit positions on the right. The result is placed into register RA.

If bit 26 of register RB contains a one, register RA is set to zero.

Registers Altered

- RA
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

sraw

Shift Right Algebraic Word

sraw RA, RS, RB Rc=0
sraw. RA, RS, RB Rc=1

31	RS	RA	RB	792	Rc
0	6	11	16	21	31

```
n ← (RB)27:31
r ← ROTL((RS), 32 – n)
if (RB)26 = 0 then
    m ← MASK(n, 31)
else
    m ← 320
s ← (RS)0
(RA) ← (r ∧ m) ∨ (32s ∧ ¬m)
XER[CA] ← s ∧ ((r ∧ ¬m) ≠ 0)
```

The contents of register RS are shifted right by the number of bits specified the contents of register RB_{27:31}. Bits shifted out of the least significant bit are lost. Register RS₀ is replicated to fill the vacated positions on the left. The result is placed into register RA.

If register RS contains a negative number and any 1-bits were shifted out of the least significant bit position, XER[CA] is set to 1; otherwise, it is set to 0.

If bit 26 of register RB contains 1, register RA and XER[CA] are set to bit 0 of register RS.

9

Registers Altered

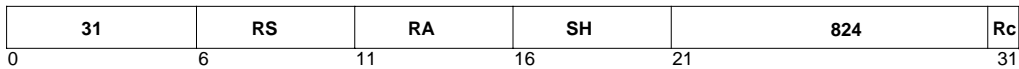
- RA
- XER[CA]
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

srawiRA, RS, SHRc=0

srawi. RA, RS, SHRc=1



```
n ← SH
r ← ROTL((RS), 32 – n)
m ← MASK(n, 31)
s ← (RS)0
(RA) ← (r ∧ m) ∨ (32s ∧ ¬m)
XER[CA] ← s ∧ ((r ∧ ¬m)≠0)
```

The contents of register RS are shifted right by the number of bits specified in the SH field. Bits shifted out of the least significant bit are lost. Bit RS₀ is replicated to fill the vacated positions on the left. The result is placed into register RA.

If register RS contains a negative number and any 1-bits were shifted out of the least significant bit position, XER[CA] is set to 1; otherwise, it is set to 0.

Registers Altered

- RA
- XER[CA]
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1

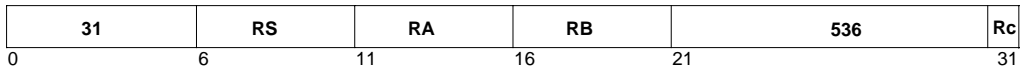
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

SrW

Shift Right Word

srw RA, RS, RB Rc=0
srw. RA, RS, RB Rc=1



```
n ← (RB)27:31
r ← ROTL((RS), 32 – n)
if (RB)26 = 0 then
    m ← MASK(n, 31)
else
    m ← 320
(RA) ← r ∧ m
```

The contents of register RS are shifted right by the number of bits specified the contents of register RB_{27:31}. Bits shifted right out of the least significant bit are lost, and 0-bits fill the vacated bit positions on the left. The result is placed into register RA.

If bit 26 of register RB contains a one, register RA is set to 0.

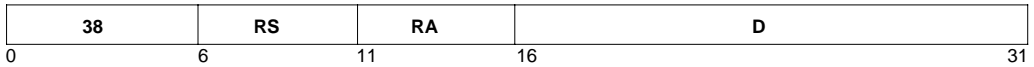
Registers Altered

- RA
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stb RS, D(RA)



$EA \leftarrow (RA|0) + \text{EXTS}(D)$
 $MS(EA, 1) \leftarrow (RS)_{24:31}$

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

The least significant byte of register RS is stored into the byte at the EA.

Registers Altered

- None

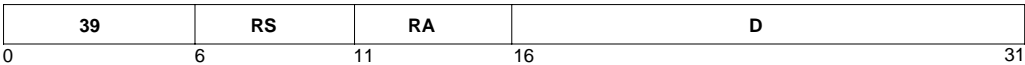
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stbu

Store Byte with Update

stbu RS, D(RA)



$$EA \leftarrow (RA|0) + EXTS(D)$$
$$MS(EA, 1) \leftarrow (RS)_{24:31}$$
$$(RA) \leftarrow EA$$

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

The least significant byte of register RS is stored into the byte at the EA.

The EA is placed into register RA.

Registers Altered

- RA

Invalid Instruction Forms

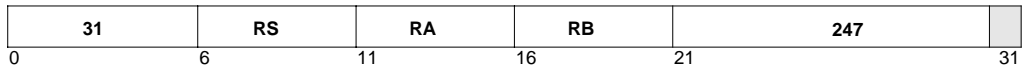
RA = 0

9

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stbux RS, RA, RB



$EA \leftarrow (RA|0) + (RB)$
 $MS(EA, 1) \leftarrow (RS)_{24:31}$
 $(RA) \leftarrow EA$

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

The least significant byte of register RS is stored into the byte at the EA.

The EA is placed into register RA.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RA

Invalid Instruction Forms

- Reserved fields
- $RA = 0$

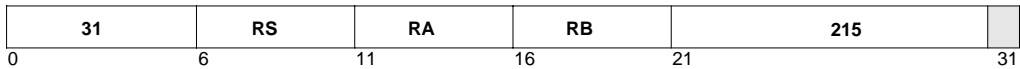
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stbx

Store Byte Indexed

stbx RS, RA, RB



$EA \leftarrow (RA|0) + (RB)$
 $MS(EA, 1) \leftarrow (RS)_{24:31}$

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

The least significant byte of register RS is stored into the byte at the EA.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

sth RS, D(RA)



$$EA \leftarrow (RA|0) + \text{EXTS}(D)$$

$$MS(EA, 2) \leftarrow (RS)_{16:31}$$

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 when the RA field is 0 and is the contents of register RA otherwise.

The least significant halfword of register RS is stored into the halfword at the EA in main storage.

Registers Altered

- None

Exceptions

An alignment exception occurs if MSR[LE] = 1 and the EA is not halfword-aligned.

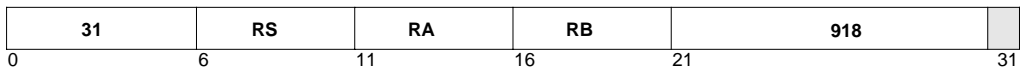
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

sthbrx

Store Halfword Byte-Reverse Indexed

sthbrx RS, RA, RB



$EA \leftarrow (RA|0) + (RB)$
 $MS(EA, 2) \leftarrow (RS)_{24:31} \parallel (RS)_{16:23}$

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

The least significant halfword of register RS is byte-reversed. The result is stored into the halfword at the EA.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

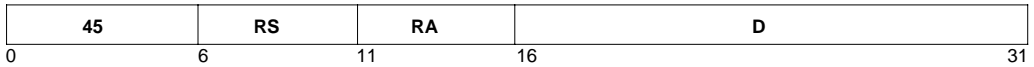
9

Exceptions

An alignment exception occurs if MSR[LE] = 1 and the EA is not halfword-aligned.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

sthu RS, D(RA)

$$EA \leftarrow (RA|0) + \text{EXTS}(D)$$

$$\text{MS}(EA, 2) \leftarrow (RS)_{16:31}$$

$$(RA) \leftarrow EA$$

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

The least significant halfword of register RS is stored into the halfword at the EA.

The EA is placed into register RA.

Registers Altered

- RA

Invalid Instruction Forms

- RA = 0

Exceptions

An alignment exception occurs if MSR[LE] = 1 and the EA is not halfword-aligned.

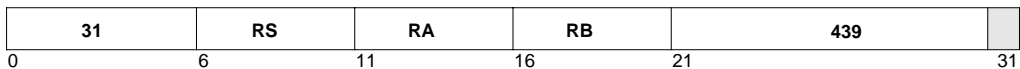
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

sthux

Store Halfword with Update Indexed

sthux RS, RA, RB



```
EA ← (RA|0) + (RB)
MS(EA, 2) ← (RS)16:31
(RA) ← EA
```

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

The least significant halfword of register RS is stored into the halfword at the EA.

The EA is placed into register RA.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RA

Invalid Instruction Forms

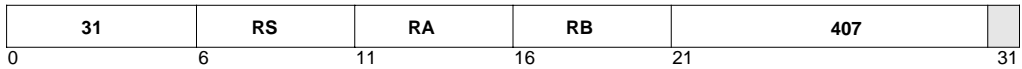
- Reserved fields
- RA = 0

Exceptions

An alignment exception occurs if MSR[LE] = 1 and the EA is not halfword-aligned.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

sthx RS, RA, RB

$$EA \leftarrow (RA|0) + (RB)$$

$$MS(EA, 2) \leftarrow (RS)_{16:31}$$

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

The least significant halfword of register RS is stored into the halfword at the EA.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Exceptions

An alignment exception occurs if MSR[LE] = 1 and the EA is not halfword-aligned.

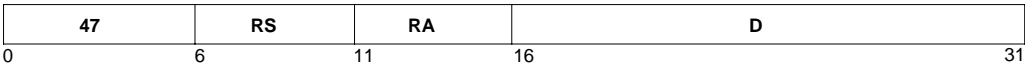
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stmw

Store Multiple Word

stmw RS, D(RA)



```
EA ← (RA|0) + EXTS(D)
r ← RS
do while r ≤ 31
    MS(EA, 4) ← (GPR(r))
    r ← r + 1
    EA ← EA + 4
```

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

The contents of a series of consecutive registers, starting with register RS and continuing through GPR(31), are stored into consecutive words starting at the EA.

Registers Altered

- None

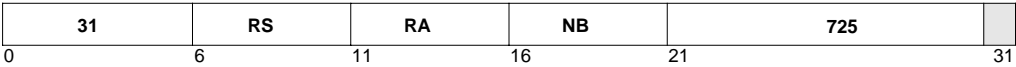
Exceptions

An alignment exception occurs if MSR[LE] = 1.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stswi RS, RA, NB



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

An effective address (EA) is determined by the RA field. If the RA field contains 0, the EA is 0; otherwise, the EA is the contents of register RA.

A byte count is determined by the NB field. If the NB field contains 0, the byte count is 32; otherwise, the byte count is the contents of the NB field.

The contents of a series of consecutive GPRs (starting with register RS, continuing through GPR(31), wrapping to GPR(0), and continuing to the final byte count) are stored, starting at the EA. The bytes in each GPR are accessed starting with the most significant byte. The byte count determines the number of transferred bytes.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Exceptions

An alignment exception occurs if MSR[LE] = 1.

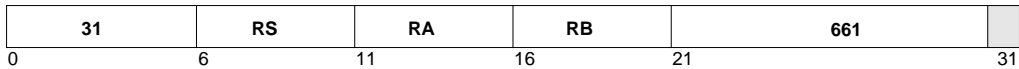
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stswx

Store String Word Indexed

stswx RS, RA, RB



```
EA ← (RA|0) + (RB)
n ← XER[TBC]
r ← RS - 1
i ← 0
do while n > 0
    if i = 0 then
        r ← r + 1
    if r = 32 then
        r ← 0
    MS(EA, 1) ← (GPR(r)i:i+7)
    i ← i + 8
    if i = 32 then
        i ← 0
    EA ← EA + 1
    n ← n - 1
```

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

A byte count is contained in XER[TBC].

The contents of a series of consecutive GPRs (starting with register RS, continuing through GPR(31), wrapping to GPR(0), and continuing to the final byte count) are stored, starting at the EA. The bytes in each GPR are accessed starting with the most significant byte. The byte count determines the number of transferred bytes.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Note

If XER[TBC] = 0, **stswx** is treated as a no-op.

The PowerPC Architecture states that, if XER[TBC] = 0 and if the EA is such that a precise data exception would normally occur (if not for the zero length), **stswx** is treated as a no-op

and the precise exception will not occur. Data storage exceptions and alignment exceptions are examples of precise data exceptions.

However, the architecture makes no statement regarding imprecise exceptions related to **stswx** with XER[TBC] = 0. The PPC401A1 generates an imprecise exception (machine check) on this instruction when all of the following conditions are true:

- The instruction passes all protection bounds checking
- The address is cacheable
- The address is passed to the data cache
- The address misses in the data cache (resulting in a line fill request)
- The address encounters some form of bus error (non-configured, for example).

Exceptions

An alignment exception occurs if MSR[LE] = 1.

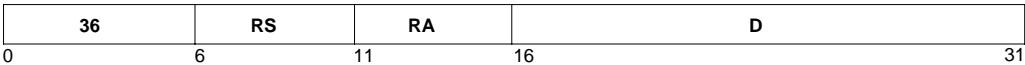
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stw

Store Word

stw RS, D(RA)



$EA \leftarrow (RA|0) + EXTS(D)$
 $MS(EA, 4) \leftarrow (RS)$

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

The contents of register RS are stored at the EA.

Registers Altered

- None

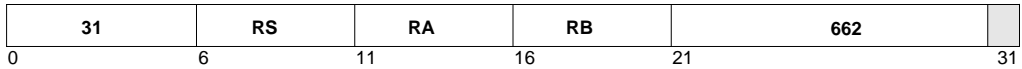
Exceptions

An alignment exception occurs if MSR[LE] = 1 and the EA is not word-aligned.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stwbrx RS, RA, RB



$EA \leftarrow (RA|0) + (RB)$

$MS(EA, 4) \leftarrow (RS)_{24:31} \parallel (RS)_{16:23} \parallel (RS)_{8:15} \parallel (RS)_{0:7}$

An EA is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

The contents of register RS are byte-reversed: the least significant byte becomes the most significant byte, the next least significant byte becomes the next most significant byte, and so on. The result is stored into the word at the EA.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Exceptions

An alignment exception occurs if MSR[LE] = 1 and the EA is not word-aligned.

Invalid Instruction Forms

- Reserved fields

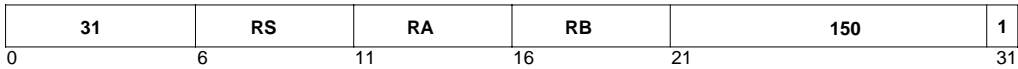
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stwcx.

Store Word Conditional Indexed

stwcx. RS, RA, RB



```
EA ← (RA[0] + (RB))
if RESERVE = 1 then
    MS(EA, 4) ← (RS)
    RESERVE ← 0
    (CR[CR0]) ← 20 || 1 || XERso
else
    (CR[CR0]) ← 20 || 0 || XERso
```

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

If the reservation bit contains 1 when the instruction is executed, the contents of register RS are stored into the word at the EA and the reservation bit is cleared. If the reservation bit contains 0 when the instruction is executed, no store operation is performed.

CR[CR0] is set as follows:

- CR[CR0]_{LT, GT} are cleared
- CR[CR0]_{EQ} is set to the state of the reservation bit at the start of the instruction
- CR[CR0]_{SO} is set to the contents of the XER[SO] bit

Registers Altered

- CR[CR0]_{LT, GT, EQ, SO}

Programming Note

lwarx and the **stwcx.** instruction should be paired in a loop, as shown in the following example, to create the effect of an atomic operation to a memory area used as a semaphore between asynchronous processes. Only **lwarx** can set the reservation bit to 1. **stwcx.** sets the reservation bit to 0 upon its completion, whether or not **stwcx.** sent (RS) to memory. CR[CR0]_{EQ} must be examined to determine whether (RS) was sent to memory.

```
loop: lwarx          # read the semaphore from memory; set reservation
      "alter"       # change the semaphore bits in register as required
      stwcx.        # attempt to store semaphore; reset reservation
      bne loop      # an asynchronous process has intervened; try again
```

If the asynchronous process in the code example had paired **lwarx** with a store other than **stwcx.**, the reservation bit would not have been cleared in the asynchronous process, and the code example would have overwritten the semaphore.

stwcx.

Store Word Conditional Indexed

Exceptions

An alignment exception occurs if the EA is not word-aligned.

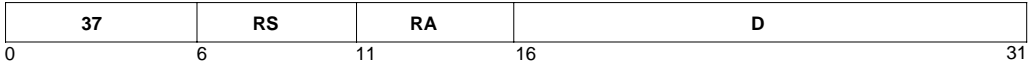
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stwu

Store Word with Update

stwu RS, D(RA)



```
EA ← (RA|0) + EXTS(D)
MS(EA, 4) ← (RS)
(RA) ← EA
```

An effective address (EA) is formed by adding a displacement to a base address. The displacement is obtained by sign-extending the 16-bit D field to 32 bits. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

The contents of register RS are stored into the word at the EA.

The EA is placed into register RA.

Registers Altered

- RA

Invalid Instruction Forms

- RA = 0

9

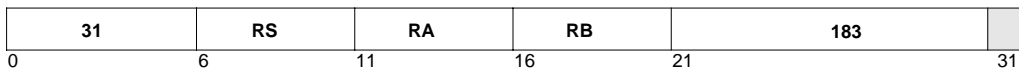
Exceptions

An alignment exception occurs if MSR[LE] = 1 and the EA is not word-aligned.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stwux RS, RA, RB



```
EA ← (RA|0) + (RB)
MS(EA, 4) ← (RS)
(RA) ← EA
```

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

The contents of register RS are stored into the word at the EA.

The EA is placed into register RA.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- RA

Invalid Instruction Forms

- Reserved fields
- RA = 0

Exceptions

An alignment exception occurs if MSR[LE] = 1 and the EA is not word-aligned.

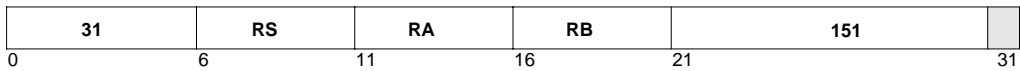
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stwx

Store Word Indexed

stwx RS, RA, RB



$EA \leftarrow (RA|0) + (RB)$
 $MS(EA,4) \leftarrow (RS)$

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 when the RA field is 0, and is the contents of register RA otherwise.

The contents of register RS are stored into the word at the EA.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Exceptions

An alignment exception occurs if MSR[LE] = 1 and the EA is not word-aligned.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

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

$$(RT) \leftarrow \neg(RA) + (RB) + 1$$

The sum of the ones complement of register RA, register RB, and 1 is stored into register RT.

Registers Altered

- RT
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-29. Extended Mnemonics for subf, subf., subfo, subfo.

Mnemonic	Operands	Function	Other Registers Changed
sub	RT, RA, RB	Subtract (RB) from (RA). $(RT) \leftarrow \neg(RB) + (RA) + 1.$ <i>Extended mnemonic for subf RT,RB,RA</i>	
sub.		<i>Extended mnemonic for subf. RT,RB,RA</i>	CR[CR0]
subo		<i>Extended mnemonic for subfo RT,RB,RA</i>	XER[SO, OV]
subo.		<i>Extended mnemonic for subfo. RT,RB,RA</i>	CR[CR0] XER[SO, OV]

subfc

Subtract From Carrying

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

31	RT	RA	RB	OE	8	Rc
0	6	11	16	21 22		31

```
(RT) ← ¬(RA) + (RB) + 1
if ¬(RA) + (RB) + 1 > 232 - 1 then
    XER[CA] ← 1
else
    XER[CA] ← 0
```

The sum of the ones complement of register RA, register RB, and 1 is stored into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the subtract operation.

Registers Altered

- RT
- XER[CA]
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-30. Extended Mnemonics for subfc, subfc., subfco, subfco.

Mnemonic	Operands	Function	Other Registers Changed
subc	RT, RA, RB	Subtract (RB) from (RA). $(RT) \leftarrow \neg(RB) + (RA) + 1$. Place carry-out in XER[CA]. <i>Extended mnemonic for subfc RT,RB,RA</i>	
subc.		<i>Extended mnemonic for subfc. RT,RB,RA</i>	CR[CR0]
subco		<i>Extended mnemonic for subfco RT,RB,RA</i>	XER[SO, OV]
subco.		<i>Extended mnemonic for subfco. RT,RB,RA</i>	CR[CR0] XER[SO, OV]

subfe

Subtract From Extended

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

31	RT	RA	RB	OE	136	Rc
0	6	11	16	21 22		31

```
(RT) ← ¬(RA) + (RB) + XER[CA]
if ¬(RA) + (RB) + XER[CA] u > 232 − 1 then
    XER[CA] ← 1
else
    XER[CA] ← 0
```

The sum of the ones complement of register RA, register RB, and XER[CA] is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the subtract operation.

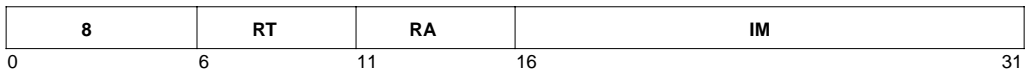
Registers Altered

- RT
- XER[CA]
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1
- XER[SO, OV] if OE contains 1

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

subfic RT, RA, IM



```

(RT) ← ¬(RA) + EXTS(IM) + 1
if ¬(RA) + EXTS(IM) + 1  $\geq$   $2^{32} - 1$  then
    XER[CA] ← 1
else
    XER[CA] ← 0
    
```

The sum of the ones complement of RA, the IM field sign-extended to 32 bits, and 1 is placed into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the subtract operation.

Registers Altered

- RT
- XER[CA]

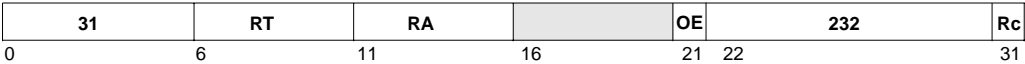
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

subfme

Subtract from Minus One Extended

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



```
(RT) ← ¬(RA) − 1 + XER[CA]
if ¬(RA) + 0xFFFF FFFF + XER[CA] ≥ 232 − 1 then
    XER[CA] ← 1
else
    XER[CA] ← 0
```

The sum of the ones complement of register RA, −1, and XER[CA] is placed into register RT. XER[CA] is set to a value determined by the unsigned magnitude of the result of the subtract operation.

Registers Altered

- RT
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1
- XER[SO, OV] if OE contains 1
- XER[CA]

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

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



```
(RT) ← ¬(RA) + XER[CA]
if ¬(RA) + XER[CA]  $\overset{u}{>} 2^{32} - 1$  then
    XER[CA] ← 1
else
    XER[CA] ← 0
```

The sum of the ones complement of register RA and XER[CA] is stored into register RT.

XER[CA] is set to a value determined by the unsigned magnitude of the result of the subtract operation.

Registers Altered

- RT
- XER[CA]
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1
- XER[SO, OV] if OE contains 1

Invalid Instruction Forms

- Reserved fields

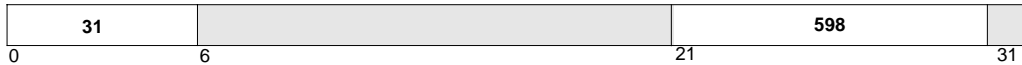
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

sync

Synchronize

sync



Synchronize System

The **sync** instruction guarantees that all instructions initiated by the processor preceding the **sync** instruction will complete before the **sync** instruction completes, and that no subsequent instructions will be initiated by the processor until after **sync** completes. When **sync** completes, all storage accesses initiated by the processor prior to **sync** will have been completed with respect to all mechanisms that access storage.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None.

Invalid Instruction Forms

- Reserved fields

Programming Note

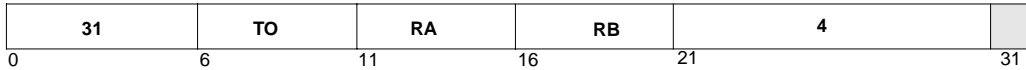
9

Architecturally, the **eieio** instruction orders storage access, not instruction completion. Therefore, non-storage operations that follow **eieio** could complete before storage operations that precede **eieio**. The **sync** instruction guarantees ordering of instruction completion and storage access. For the PPC401A1, the **eieio** instruction is implemented to behave as a **sync** instruction. To write code that is portable between various PowerPC implementations, programmers should use the mnemonic which corresponds to the desired behavior.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

tw TO, RA, RB



if (((RA) < (RB) \wedge TO₀ = 1) \vee
 ((RA) > (RB) \wedge TO₁ = 1) \vee
 ((RA) = (RB) \wedge TO₂ = 1) \vee
 ((RA) \leq (RB) \wedge TO₃ = 1) \vee
 ((RA) \geq (RB) \wedge TO₄ = 1)) then TRAP (see details below)

Register RA is compared with register RB. If any comparison condition selected by the TO field is true, a TRAP occurs. The behavior of a TRAP depends upon the debug mode of the processor, as described below:

- If TRAP is not enabled as a debug event (DBCR[TDE] = 0 or DBCR[EDM,IDM] = 0,0):

TRAP causes a program interrupt. See Section 5.9, “Program Exceptions,” on p. 5-18, for more information.

(SRR0) \leftarrow address of **tw** instruction
 (SRR1) \leftarrow (MSR)
 (ESR[PTR]) \leftarrow 1
 (MSR[WE, EE, PR]) \leftarrow 0
 (MSR[LE]) \leftarrow (MSR[ILE])
 PC \leftarrow EVPR_{0:15} || 0x0700

- If TRAP is enabled as an external debug event (DBCR[TDE] = 1 and DBCR[EDM] = 1):

TRAP goes to the debug stop state, to be handled by an external debugger with hardware control.

(DBSR[TIE]) \leftarrow 1

In addition, if TRAP is also enabled as an internal debug event (DBCR[IDM] = 1) and debug exceptions are disabled (MSR[DE] = 0), then report an imprecise event:
 (DBSR[IDE]) \leftarrow 1

PC \leftarrow address of **tw** instruction

- If TRAP is enabled as an internal debug event and *not* an external debug event (DBCR[TDE] = 1 and DBCR[EDM,IDM] = 0,1) and debug exceptions are enabled (MSR[DE] = 1):

TRAP causes a debug interrupt. See Section 5.14, “Debug Exception Handling,” on p. 5-23, for more information.

tw

Trap Word

(SRR2) \leftarrow address of **tw** instruction
(SRR3) \leftarrow (MSR)
(DBSR[TIE]) \leftarrow 1
(MSR[WE, EE, PR, CE, DE]) \leftarrow 0
(MSR[LE]) \leftarrow (MSR[ILE])
PC \leftarrow EVPR_{0:15} || 0x2000

- If TRAP is enabled as an internal debug event and *not* an external debug event (DBCR[TDE] = 1 and DBCR[EDM, IDM] = 0, 1) and Debug Exceptions are disabled (MSR[DE] = 0):

TRAP reports the debug event as an *imprecise* event and causes a program interrupt. See Section 5.9, “Program Exceptions,” on p. 5-18 for more information.

(SRR0) \leftarrow address of **tw** instruction
(SRR1) \leftarrow (MSR)
(ESR[PTR]) \leftarrow 1
(DBSR[TIE, IDE]) \leftarrow 1, 1
(MSR[WE, EE, PR]) \leftarrow 0
(MSR[LE]) \leftarrow (MSR[ILE])
PC \leftarrow EVPR_{0:15} || 0x0700

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

9

Registers Altered

- None

Invalid Instruction Forms

- Reserved fields

Programming Note

This instruction is inserted into the execution stream by a debugger to implement breakpoints, and is not typically used by application code.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-31. Extended Mnemonics for tw

Mnemonic	Operands	Function	Other Registers Changed
trap		Trap unconditionally. <i>Extended mnemonic for tw 31,0,0</i>	

Table 9-31. Extended Mnemonics for tw (cont.)

Mnemonic	Operands	Function	Other Registers Changed
tw eq	RA, RB	Trap if (RA) equal to (RB). <i>Extended mnemonic for tw 4,RA,RB</i>	
tw ge	RA, RB	Trap if (RA) greater than or equal to (RB). <i>Extended mnemonic for tw 12,RA,RB</i>	
tw gt	RA, RB	Trap if (RA) greater than (RB). <i>Extended mnemonic for tw 8,RA,RB</i>	
tw le	RA, RB	Trap if (RA) less than or equal to (RB). <i>Extended mnemonic for tw 20,RA,RB</i>	
tw lge	RA, RB	Trap if (RA) logically greater than or equal to (RB). <i>Extended mnemonic for tw 5,RA,RB</i>	
tw lgt	RA, RB	Trap if (RA) logically greater than (RB). <i>Extended mnemonic for tw 1,RA,RB</i>	
tw lle	RA, RB	Trap if (RA) logically less than or equal to (RB). <i>Extended mnemonic for tw 6,RA,RB</i>	
tw llt	RA, RB	Trap if (RA) logically less than (RB). <i>Extended mnemonic for tw 2,RA,RB</i>	
tw lng	RA, RB	Trap if (RA) logically not greater than (RB). <i>Extended mnemonic for tw 6,RA,RB</i>	
tw nlt	RA, RB	Trap if (RA) logically not less than (RB). <i>Extended mnemonic for tw 5,RA,RB</i>	
tw lt	RA, RB	Trap if (RA) less than (RB). <i>Extended mnemonic for tw 16,RA,RB</i>	

tw

Trap Word

Table 9-31. Extended Mnemonics for tw (cont.)

Mnemonic	Operands	Function	Other Registers Changed
twne	RA, RB	Trap if (RA) not equal to (RB). <i>Extended mnemonic for tw 24,RA,RB</i>	
twng	RA, RB	Trap if (RA) not greater than (RB). <i>Extended mnemonic for tw 20,RA,RB</i>	
twnl	RA, RB	Trap if (RA) not less than (RB). <i>Extended mnemonic for tw 12,RA,RB</i>	

twi TO, RA, IM

3	TO	RA	IM
0	6	11	16
			31

if (((RA) < EXTS(IM) \wedge TO₀ = 1) \vee
 ((RA) > EXTS(IM) \wedge TO₁ = 1) \vee
 ((RA) = EXTS(IM) \wedge TO₂ = 1) \vee
 ((RA) \wedge EXTS(IM) \wedge TO₃ = 1) \vee
 ((RA) \vee EXTS(IM) \wedge TO₄ = 1)) then TRAP (see details below)

Register RA is compared with the IM field, which has been sign-extended to 32 bits. If any comparison condition selected by the TO field is true, a TRAP occurs. The behavior of a TRAP depends upon the Debug Mode of the processor, as described below:

- If TRAP is not enabled as a debug event (DBCR[TDE] = 0 or DBCR[EDM, IDM] = 0,0):

TRAP causes a Program interrupt. See Section 5.9, “Program Exceptions,” on p. 5-18 for more information.

(SRR0) \leftarrow address of **twi** instruction
 (SRR1) \leftarrow (MSR)
 (ESR[PTR]) \leftarrow 1
 (MSR[WE, EE, PR]) \leftarrow 0
 (MSR[LE]) \leftarrow (MSR[ILE])
 PC \leftarrow EVPR_{0:15} || 0x0700

- If TRAP is enabled as an External debug event (DBCR[TDE] = 1 and DBCR[EDM] = 1):

TRAP goes to the Debug Stop state, to be handled by an external debugger with hardware control over the PPC401A1.

(DBSR[TIE]) \leftarrow 1
 In addition, if TRAP is also enabled as an Internal debug event (DBCR[IDM] = 1)
 and Debug Exceptions are disabled (MSR[DE] = 0), then report an imprecise event:
 (DBSR[IDE]) \leftarrow 1
 PC \leftarrow address of **twi** instruction

- If TRAP is enabled as an Internal debug event and *not* an External debug event (DBCR[TDE] = 1 and DBCR[EDM, IDM] = 0,1) and Debug Exceptions are enabled (MSR[DE] = 1):

TRAP causes a Debug interrupt. See Section 5.14, “Debug Exception Handling,” on p. 5-23, for further information.

twi

Trap Word Immediate

(SRR2) \leftarrow address of **twi** instruction
(SRR3) \leftarrow (MSR)
(DBSR[TIE]) \leftarrow 1
(MSR[WE, EE, PR, CE, DE]) \leftarrow 0
(MSR[LE]) \leftarrow (MSR[ILE])
PC \leftarrow EVPR_{0:15} || 0x2000

- If TRAP is enabled as an Internal debug event and *not* an External debug event (DBCR[TDE] = 1 and DBCR[EDM, IDM] = 0, 1) and Debug Exceptions are disabled (MSR[DE] = 0):

TRAP will report the debug event as an *imprecise* event and will cause a Program interrupt. See Section 5.9, “Program Exceptions,” on p. 5-18 for more information.

(SRR0) \leftarrow address of **twi** instruction
(SRR1) \leftarrow (MSR)
(ESR[PTR]) \leftarrow 1
(DBSR[TIE, IDE]) \leftarrow 1, 1
(MSR[WE, EE, PR]) \leftarrow 0
(MSR[LE]) \leftarrow (MSR[ILE])
PC \leftarrow EVPR_{0:15} || 0x0700

Registers Altered

- None

Programming Note

This instruction is inserted into the execution stream by a debugger to implement breakpoints, and is not typically used by application code.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 9-32. Extended Mnemonics for twi

Mnemonic	Operands	Function	Other Registers Changed
tweqi	RA, IM	Trap if (RA) equal to EXTS(IM). <i>Extended mnemonic for twi 4,RA,IM</i>	
twgei	RA, IM	Trap if (RA) greater than or equal to EXTS(IM). <i>Extended mnemonic for twi 12,RA,IM</i>	

Table 9-32. Extended Mnemonics for twi (cont.)

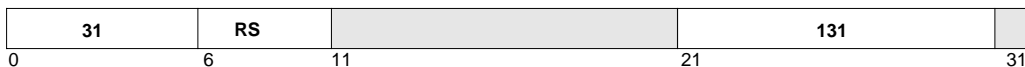
Mnemonic	Operands	Function	Other Registers Changed
twgti	RA, IM	Trap if (RA) greater than EXTS(IM). <i>Extended mnemonic for twi 8,RA,IM</i>	
twlei	RA, IM	Trap if (RA) less than or equal to EXTS(IM). <i>Extended mnemonic for twi 20,RA,IM</i>	
twlgei	RA, IM	Trap if (RA) logically greater than or equal to EXTS(IM). <i>Extended mnemonic for twi 5,RA,IM</i>	
twlgti	RA, IM	Trap if (RA) logically greater than EXTS(IM). <i>Extended mnemonic for twi 1,RA,IM</i>	
twllei	RA, IM	Trap if (RA) logically less than or equal to EXTS(IM). <i>Extended mnemonic for twi 6,RA,IM</i>	
twllti	RA, IM	Trap if (RA) logically less than EXTS(IM). <i>Extended mnemonic for twi 2,RA,IM</i>	
twlngi	RA, IM	Trap if (RA) logically not greater than EXTS(IM). <i>Extended mnemonic for twi 6,RA,IM</i>	
twlnli	RA, IM	Trap if (RA) logically not less than EXTS(IM). <i>Extended mnemonic for twi 5,RA,IM</i>	
twlti	RA, IM	Trap if (RA) less than EXTS(IM). <i>Extended mnemonic for twi 16,RA,IM</i>	
twnei	RA, IM	Trap if (RA) not equal to EXTS(IM). <i>Extended mnemonic for twi 24,RA,IM</i>	

twi

Trap Word Immediate

Table 9-32. Extended Mnemonics for twi (cont.)

Mnemonic	Operands	Function	Other Registers Changed
twngi	RA, IM	Trap if (RA) not greater than EXTS(IM). <i>Extended mnemonic for twi 20,RA,IM</i>	
twnli	RA, IM	Trap if (RA) not less than EXTS(IM). <i>Extended mnemonic for twi 12,RA,IM</i>	

wrtee RS

$$\text{MSR}[\text{EE}] \leftarrow (\text{RS})_{16}$$

The MSR[EE] is set to the value specified by bit 16 of register RS.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- MSR[EE]

Invalid Instruction Forms:

- Reserved fields

Programming Note

Execution of this instruction is privileged.

This instruction is used to provide atomic update of MSR[EE]. Typical usage is:

mfmsr	Rn	#Save EE in Rn[16]
wrteei	0	#Turn off EE
• • • • •		#Code with EE disabled
wrtee	Rn	#Restore EE without affecting other MSR changes that may have occurred during the disabled code

Architecture Note

Programs using this instruction are not portable to PowerPC implementations that do not implement the IBM PowerPC Embedded Environment.

wrteei

Write External Enable Immediate

wrteei E $\text{MSR}[\text{EE}] \leftarrow \text{E}$

The MSR[EE] is set to the value specified by the E field.

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- MSR[EE]

Invalid Instruction Forms:

- Reserved fields

Programming Note

Execution of this instruction is privileged.

This instruction is used to provide atomic update of MSR[EE]. Typical usage is:

9

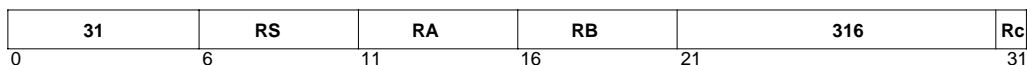
mfmsr	Rn	#Save EE in Rn[16]
wrteei	0	#Turn off EE
• • • • •		#Code with EE disabled
wrtee	Rn	#Restore EE without affecting other MSR changes that may have occurred during the disabled code

Architecture Note

Programs using this instruction are not portable to PowerPC implementations that do not implement the IBM PowerPC Embedded Environment.

xor

RA, RS, RB

 $R_c = 1$ 

$$(RA) \leftarrow (RS) \oplus (RB)$$

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

Registers Altered

- $CR[CR0]_{LT, GT, EQ, SO}$ if Rc contains 1
- RA

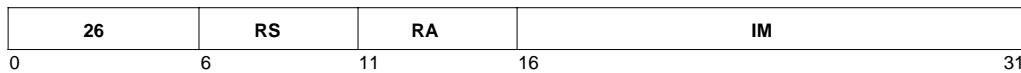
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

xori

XOR Immediate

xori RA, RS, IM



$$(RA) \leftarrow (RS) \oplus (160 \parallel IM)$$

The IM field is extended to 32 bits by concatenating 16 0-bits on the left. The contents of register RS are XORed with the extended IM field; the result is placed into register RA.

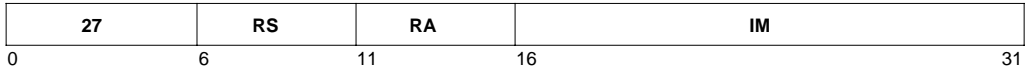
Registers Altered

- RA

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

xoris RA, RS, IM



$$(RA) \leftarrow (RS) \oplus (IM \parallel ^{16}0)$$

The IM field is extended to 32 bits by concatenating 16 0-bits on the right. The contents of register RS are XORed with the extended IM field; the result is placed into register RA.

Registers Altered

- RA

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

All registers contained in the PPC401A1 are architected as 32-bits. Table 10-1 through Table 10-2 defines the addressing required to access the registers. The pages following these tables define the bit usage within each register.

10.1 Reserved Registers

Any register numbers not listed in the tables which follow are *reserved*, and should be neither read nor written. These reserved register numbers may be used for additional functions on future PowerPC Embedded Controllers.

10.2 Reserved Fields

For all registers with fields marked as reserved, the reserved fields should be written as *zero* and read as *undefined*. That is, when writing to a reserved field, write a zero to that field. When reading from a reserved field, ignore that field.

A good coding practice is to perform the initial write to a register with reserved fields as described in the preceding paragraph, and to perform all subsequent writes to the register using a read-modify-write strategy: read the register, alter desired fields with logical instructions, and then write the register.

10.3 General Purpose Registers

The PPC401A1 contains 32 General Purpose Registers (GPRs). The contents of these registers can be loaded from memory using load instructions and stored to memory using store instructions. GPRs are also addressed by all integer instructions.

Table 10-1. PPC401A1 General Purpose Registers

Mnemonic	Register Name	GPR Number		Access
		Decimal	Hex	
R0–R31	General Purpose Register 0–31	0–31	0x0–0x1F	Read/Write

10.4 Machine State Register and Condition Register

Because these registers are accessed using special instructions, they do not require addressing.

10.5 Special Purpose Registers

Special Purpose Registers (SPRs), which are part of the PowerPC Embedded Architecture, are accessed using the **mtspr** and **mfspir** instructions. SPRs control the use of the debug facilities, timers, interrupts, storage control attributes, and other architected processor resources.

10

Table 10-2 shows the mnemonics, names, and numbers of the SPRs. The columns under “SPRN” list the register numbers used as operands in assembler language coding of the **mfspir** and **mtspr** instructions. The column labeled “SPRF” lists the corresponding fields contained in the *machine code* of **mfspir** and **mtspr**. The SPRN field contains two five-bit subfields of the SPRF field; the subfields are *reversed* in the machine code for the **mfspir** and **mtspr** instructions ($\text{SPRN} \leftarrow \text{SPRF}_{5:9} \parallel \text{SPRF}_{0:4}$) for compatibility with the POWER Architecture. Note that the assembler handles the special coding transparently.

The only SPRs that are not privileged are the Count Register (CTR), the Link Register (LR), the Time Base High User-mode (TBHU), the Time Base Low User-mode (TBLU), and the Fixed-point Exception Register (XER). See Section 2.8.3, “Privileged SPRs,” on p. 2-39.

Table 10-2 lists the SPRs, their mnemonics and names, their numbers (SPRN) and the corresponding SPRF numbers, and access. All SPR numbers not listed are reserved, and should be neither read nor written.

Table 10-2. Special Purpose Registers

Mnemonic	Register Name	SPRN		SPRF	Access
		Decimal	Hex		
CDBCR	Cache Debug Control Register	983	0x3D7	0x2FE	Read/Write
CTR	Count Register	9	0x009	0x120	Read/Write
DAC	Data Address Compare	1014	0x3F6	0x2DF	Read/Write
DBCR	Debug Control Register	1010	0x3F2	0x25F	Read/Write
DBSR	Debug Status Register	1008	0x3F0	0x21F	Read/Clear
DCCR	Data Cache Cacheability Register	1018	0x3FA	0x35F	Read/Write
DCWR	Data Cache Write-through Register	954	0x3BA	0x35D	Read/Write
DEAR	Data Error Address Register	981	0x3D5	0x2BE	Read/Write
ESR	Exception Syndrome Register	980	0x3D4	0x29E	Read/Write
EVPR	Exception Vector Prefix Register	982	0x3D6	0x2DE	Read/Write
IAC	Instruction Address Compare	1012	0x3F4	0x29F	Read/Write
ICCR	Instruction Cache Cacheability Register	1019	0x3FB	0x37F	Read/Write
ICDBDR	Instruction Cache Debug Data Register	979	0x3D3	0x27E	Read-only
LR	Link Register	8	0x008	0x100	Read/Write
PIT	Programmable Interval Timer	987	0x3DB	0x37E	Read/Write
PVR	Processor Version Register	287	0x11F	0x3E8	Read-only
SGR	Storage Guarded Register	953	0x3B9	0x33D	Read/Write
SLER	Storage Little Endian Register	955	0x3BB	0x37D	Read/Write
SPRG0	SPR General Purpose Register 0	272	0x110	0x208	Read/Write
SPRG1	SPR General Purpose Register 1	273	0x111	0x228	Read/Write
SPRG2	SPR General Purpose Register 2	274	0x112	0x248	Read/Write
SPRG3	SPR General Purpose Register 3	275	0x113	0x268	Read/Write
SRR0	Save/Restore Register 0	26	0x01A	0x340	Read/Write
SRR1	Save/Restore Register 1	27	0x01B	0x360	Read/Write
SRR2	Save/Restore Register 2	990	0x3DE	0x3DE	Read/Write

Table 10-2. Special Purpose Registers (cont.)

Mnemonic	Register Name	SPRN		SPRF	Access
		Decimal	Hex		
SRR3	Save/Restore Register 3	991	0x3DF	0x3FE	Read/Write
TBHI	Time Base High	988	0x3DC	0x39E	Read/Write
TBHU	Time Base High User-mode	972	0x3CC	0x19E	Read Only
TBLO	Time Base Low	989	0x3DD	0x3BE	Read/Write
TBLU	Time Base Low User-mode	973	0x3CD	0x1BE	Read Only
TCR	Timer Control Register	986	0x3DA	0x35E	Read/Write
TSR	Timer Status Register	984	0x3D8	0x31E	Read/Clear
XER	Fixed Point Exception Register	1	0x001	0x020	Read/Write

10.6 Device Control Registers

Device Control Registers (DCRs), which are architecturally outside of the processor core, are accessed using the **mtdcr** and **mfdcr** instructions. DCRs are used to control, configure, and hold status for various functional units that are not part of the RISC processor core. Although the PPC401A1 does not contain DCRs, the **mtdcr** and **mfdcr** instructions are provided.

10

The **mtdcr** and **mfdcr** instructions are privileged, for all DCR numbers. Therefore, all accesses to DCRs are privileged. See Section 2.8, “Privileged Mode Operation,” on p. 2-38.

All DCR numbers reserved, and should be neither read nor written, unless they are part of a Core+ASIC implementation.

10.7 Alphabetical Register Listing

The following pages list the registers available in the PPC401A1. For each register, the following information is supplied:

- Register name and mnemonic
- Register type (SPR)
- Register number (address)
- A diagram illustrating the register fields (all register fields have mnemonics, unless there is only one field)
- A table describing the register fields, giving field mnemonic, field bit location, field name, and the function associated with various field values

SPR 0x3D7

See also Section 6.4 on p. 6-9.

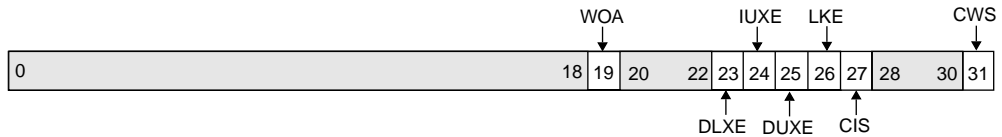


Figure 10-1. Cache Debug Control Register (CDBCR)

0:18		Reserved
19	WOA	Write-on-Allocate 0 All store misses result in a line fill. 1 Store misses do not cause a line fill, but result in a non-cacheable store.
20:22		Reserved
23	DLXE	DCU Lock-out Exception Enable 0 DCU lock-out exception is disabled. 1 DCU lock-out exception is enabled.
24	IUXE	ICU Unlock Exception Enable 0 ICU unlock exception is disabled. 1 ICU unlock exception is enabled.
25	DUXE	DCU Unlock Exception Enable 0 DCU unlock exception is disabled. 1 DCU unlock exception is enabled.
26	LKE	Lock Enable 0 Line locking is disabled. 1 Line locking is enabled.
27	CIS	Cache Information Select 0 Information is cache data. 1 Information is cache tag.
28:30		Reserved
31	CWS	Cache Way Select 0 Cache way is A. 1 Cache way is B.

CR

See also Section 2.2.3 on p. 2-9.

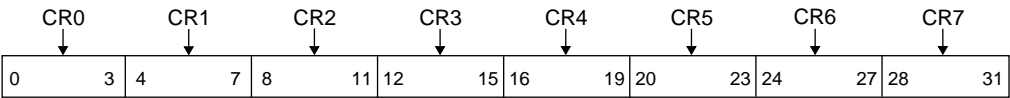


Figure 10-2. Condition Register (CR)			
0:3	CR0	Condition Register Field 0	CR[CRn] _{0:3} indicate less than, greater than, equal to, and summary overflow, respectively.
4:7	CR1	Condition Register Field 1	See the description of CR[CR0].
8:11	CR2	Condition Register Field 2	See the description of CR[CR0].
12:15	CR3	Condition Register Field 3	See the description of CR[CR0].
16:19	CR4	Condition Register Field 4	See the description of CR[CR0].
20:23	CR5	Condition Register Field 5	See the description of CR[CR0].
24:27	CR6	Condition Register Field 6	See the description of CR[CR0].
28:31	CR7	Condition Register Field 7	See the description of CR[CR0].

SPR 0x009

See also Section 2.2.2.1 on p. 2-4.

0	31
---	----

Figure 10-3. Count Register (CTR)

0:31		Count	Used as count for branch conditional with decrement instructions, or as address for branch-to-counter instructions
------	--	-------	--

DAC1

SPR 0x3F6

See also Section 7.6.3 on p. 7-8.



Figure 10-4. Data Address Compare Register (DAC)

0:31		Data Address Compare (DAC) byte address	DBCR[D1S] determines byte, halfword, or word usage.
------	--	---	---

SPR 0x3F2

See also Section 7.6.1 on p. 7-5.

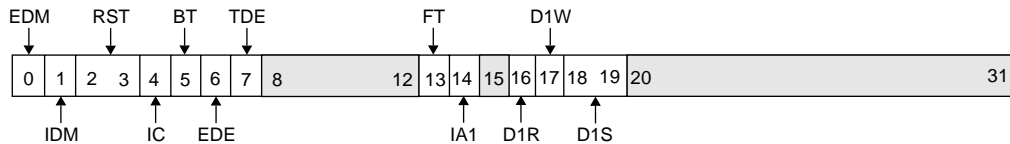


Figure 10-5. Debug Control Register (DBCR)

0	EDM	External Debug Mode 0 Disable 1 Enable	
1	IDM	Internal Debug Mode 0 Disable 1 Enable	
2:3	RST	Reset 00 No action 01 Core reset 10 Chip reset 11 System reset Attention: Writing 01, 10, or 11 to this field causes a processor reset request.	
4	IC	Instruction Completion Debug Event 0 Disable 1 Enable	Instruction completion does not cause a debug event if MSR[DE] = 0 in internal debug mode
5	BT	Branch Taken Debug Event 0 Disable 1 Enable	Branch taken does not cause a debug event if MSR[DE] = 0 in internal debug mode
6	EDE	Exception Debug Event 0 Disable 1 Enable	Critical exceptions do not cause debug events if MSR[DE] = 0 in internal debug mode
7	TDE	TRAP Debug Event 0 Disable 1 Enable	
8:12		Reserved	
13	FT	Freeze timers on debug event 0 Free-run timers 1 Freeze timers	
14	IA1	Instruction Address Compare 1 Enable 0 Disable 1 Enable	

DBCR (cont.)

Figure 10-5. Debug Control Register (DBCR) (cont.)

15		Reserved
16	D1R	DAC Read Enable 0 Disable 1 Enable
17	D1W	DAC Write Enable 0 Disable 1 Enable
18:19	D1S	DAC Size 00 Compare all bits 01 Ignore the least significant bit (lsb) 10 Ignore the two lsbs 11 Ignore the four lsbs Exact address compare Byte within halfword address compare Byte within word address compare Quadword address compare
20:31		Reserved

SPR 0x3F0 Read/Clear

See also Section 7.6.2 on p. 7-7.

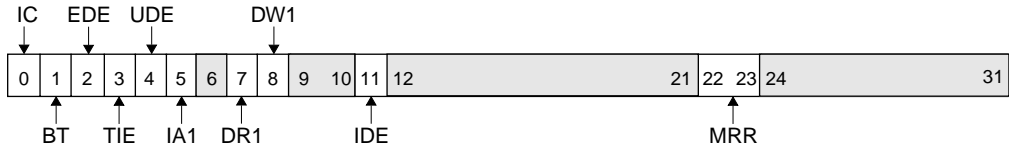


Figure 10-6. Debug Status Register (DBSR)

0	IC	Instruction Completion Debug Event 0 Event didn't occur 1 Event occurred
1	BT	Branch Taken Debug Event 0 Event didn't occur 1 Event occurred
2	EDE	Exception Debug Event 0 Event didn't occur 1 Event occurred
3	TIE	TRAP Instruction Debug Event 0 Event didn't occur 1 Event occurred
4	UDE	Unconditional Debug Event 0 Event didn't occur 1 Event occurred
5	IA1	IAC1 Debug Event 0 Event didn't occur 1 Event occurred
6		Reserved
7	DR1	DAC Read Debug Event 0 Event didn't occur 1 Event occurred
8	DW1	DAC Write Debug Event 0 Event didn't occur 1 Event occurred
9:10		Reserved
11	IDE	Imprecise Debug Event 0 Event didn't occur 1 Event occurred
12:21		Reserved

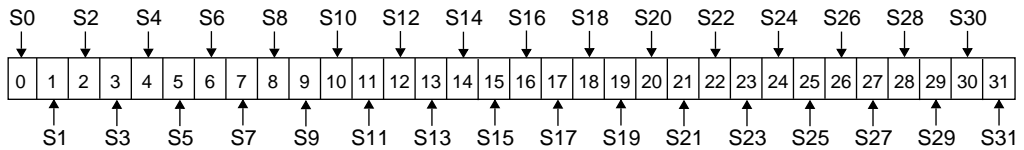
DBSR (cont.)

Figure 10-6. Debug Status Register (DBSR) (cont.)

22:23	MRR	Most Recent Reset 00 No reset has occurred since last cleared by software. 01 Core reset 10 Chip reset 11 System reset	This field is set to a value, indicating the type of reset, when a reset occurs.
24:31		Reserved	

SPR 0x3FA

See Section 8.1.1 on p. 8-2.

**Figure 10-7. Data Cache Cacheability Register (DCCR)**

0	S0	0 Noncacheable 1 Cacheable	0x0000 0000–0x07FF FFFF
1	S1	0 Noncacheable 1 Cacheable	0x0800 0000–0x0FFF FFFF
2	S2	0 Noncacheable 1 Cacheable	0x1000 0000–0x17FF FFFF
3	S3	0 Noncacheable 1 Cacheable	0x1800 0000–0x1FFF FFFF
4	S4	0 Noncacheable 1 Cacheable	0x2000 0000–0x27FF FFFF
5	S5	0 Noncacheable 1 Cacheable	0x2800 0000–0x2FFF FFFF
6	S6	0 Noncacheable 1 Cacheable	0x3000 0000–0x37FF FFFF
7	S7	0 Noncacheable 1 Cacheable	0x3800 0000–0x3FFF FFFF
8	S8	0 Noncacheable 1 Cacheable	0x4000 0000–0x47FF FFFF
9	S9	0 Noncacheable 1 Cacheable	0x4800 0000–0x4FFF FFFF
10	S10	0 Noncacheable 1 Cacheable	0x5000 0000–0x57FF FFFF
11	S11	0 Noncacheable 1 Cacheable	0x5800 0000–0x5FFF FFFF
12	S12	0 Noncacheable 1 Cacheable	0x6000 0000–0x67FF FFFF
13	S13	0 Noncacheable 1 Cacheable	0x6800 0000–0x6FFF FFFF

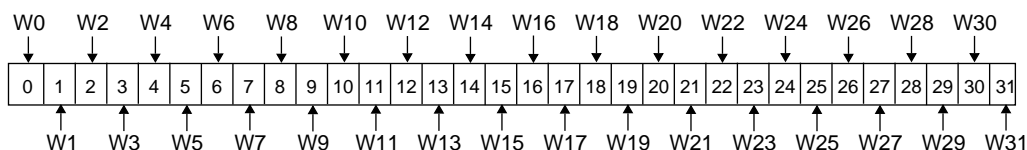
DCCR (cont.)

Figure 10-7. Data Cache Cacheability Register (DCCR) (cont.)

14	S14	0 Noncacheable 1 Cacheable	0x7000 0000–0x77FF FFFF
15	S15	0 Noncacheable 1 Cacheable	0x7800 0000–0x7FFF FFFF
16	S16	0 Noncacheable 1 Cacheable	0x8000 0000–0x87FF FFFF
17	S17	0 Noncacheable 1 Cacheable	0x8800 0000–0x8FFF FFFF
18	S18	0 Noncacheable 1 Cacheable	0x9000 0000–0x97FF FFFF
19	S19	0 Noncacheable 1 Cacheable	0x9800 0000–0x9FFF FFFF
20	S20	0 Noncacheable 1 Cacheable	0xA000 0000–0xA7FF FFFF
21	S21	0 Noncacheable 1 Cacheable	0xA800 0000–0xAFFF FFFF
22	S22	0 Noncacheable 1 Cacheable	0xB000 0000–0xB7FF FFFF
23	S23	0 Noncacheable 1 Cacheable	0xB800 0000–0xBFFF FFFF
24	S24	0 Noncacheable 1 Cacheable	0xC000 0000–0xC7FF FFFF
25	S25	0 Noncacheable 1 Cacheable	0xC800 0000–0xCFFF FFFF
26	S26	0 Noncacheable 1 Cacheable	0xD000 0000–0xD7FF FFFF
27	S27	0 Noncacheable 1 Cacheable	0xD800 0000–0xDFFF FFFF
28	S28	0 Noncacheable 1 Cacheable	0xE000 0000–0xE7FF FFFF
29	S29	0 Noncacheable 1 Cacheable	0xE800 0000–0xEFFF FFFF
30	S30	0 Noncacheable 1 Cacheable	0xF000 0000–0xF7FF FFFF
31	S31	0 Noncacheable 1 Cacheable	0xF800 0000–0xFFFF FFFF

SPR 0x3BA

See Section 8.1.1 on p. 8-2.

**Figure 10-8. Data Cache Write-through Register (DCWR)**

0	W0	0 Write-back 1 Write-through	0x0000 0000–0x07FF FFFF
1	W1	0 Write-back 1 Write-through	0x0800 0000–0x0FFF FFFF
2	W2	0 Write-back 1 Write-through	0x1000 0000–0x17FF FFFF
3	W3	0 Write-back 1 Write-through	0x1800 0000–0x1FFF FFFF
4	W4	0 Write-back 1 Write-through	0x2000 0000–0x27FF FFFF
5	W5	0 Write-back 1 Write-through	0x2800 0000–0x2FFF FFFF
6	W6	0 Write-back 1 Write-through	0x3000 0000–0x37FF FFFF
7	W7	0 Write-back 1 Write-through	0x3800 0000–0x3FFF FFFF
8	W8	0 Write-back 1 Write-through	0x4000 0000–0x47FF FFFF
9	W9	0 Write-back 1 Write-through	0x4800 0000–0x4FFF FFFF
10	W10	0 Write-back 1 Write-through	0x5000 0000–0x57FF FFFF
11	W11	0 Write-back 1 Write-through	0x5800 0000–0x5FFF FFFF
12	W12	0 Write-back 1 Write-through	0x6000 0000–0x67FF FFFF
13	W13	0 Write-back 1 Write-through	0x6800 0000–0x6FFF FFFF

DCWR (cont.)

Figure 10-8. Data Cache Write-through Register (DCWR) (cont.)

14	W14	0 Write-back 1 Write-through	0x7000 0000–0x77FF FFFF
15	W15	0 Write-back 1 Write-through	0x7800 0000–0x7FFF FFFF
16	W16	0 Write-back 1 Write-through	0x8000 0000–0x87FF FFFF
17	W17	0 Write-back 1 Write-through	0x8800 0000–0x8FFF FFFF
18	W18	0 Write-back 1 Write-through	0x9000 0000–0x97FF FFFF
19	W19	0 Write-back 1 Write-through	0x9800 0000–0x9FFF FFFF
20	W20	0 Write-back 1 Write-through	0xA000 0000–0xA7FF FFFF
21	W21	0 Write-back 1 Write-through	0xA800 0000–0xAFFF FFFF
22	W22	0 Write-back 1 Write-through	0xB000 0000–0xB7FF FFFF
23	W23	0 Write-back 1 Write-through	0xB800 0000–0xBFFF FFFF
24	W24	0 Write-back 1 Write-through	0xC000 0000–0xC7FF FFFF
25	W25	0 Write-back 1 Write-through	0xC800 0000–0xCFFF FFFF
26	W26	0 Write-back 1 Write-through	0xD000 0000–0xD7FF FFFF
27	W27	0 Write-back 1 Write-through	0xD800 0000–0xDFFF FFFF
28	W28	0 Write-back 1 Write-through	0xE000 0000–0xE7FF FFFF
29	W29	0 Write-back 1 Write-through	0xE800 0000–0xEFFF FFFF
30	W30	0 Write-back 1 Write-through	0xF000 0000–0xF7FF FFFF
31	W31	0 Write-back 1 Write-through	0xF800 0000–0xFFFF FFFF

SPR 0x3D5

See also Section 5.3.6 on p. 5-12.

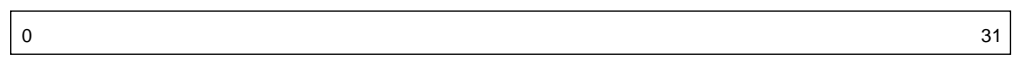


Figure 10-9. Data Exception Address Register (DEAR)

0:31	Address of Data Error (synchronous)
------	-------------------------------------

ESR

SPR 0x3D4

See also Section 5.3.5 on p. 5-10.

Figure 10-10. Exception Syndrome Register (ESR)

0	MCI	Machine check—instruction 0 Instruction machine check did not occur. 1 Instruction machine check occurred.
1:3		Reserved
4	PIL	Program exception—illegal 0 Illegal Instruction error did not occur. 1 Illegal Instruction error occurred.
5	PPR	Program exception—privileged 0 Privileged instruction error did not occur. 1 Privileged instruction error occurred.
6	PTR	Program exception—trap 0 Trap with successful compare did not occur. 1 Trap with successful compare occurred.
7:9		Reserved
10:11	DLK	Data Storage exception— lock fault 00 No lock exception 01 dcbf unlock exception 10 icbi unlock exception 11 dcbz lock-out exception
12	PAU	Program exception—auxiliary processor unavailable 0 Auxiliary processor unavailable exception did not occur. 1 Auxiliary processor unavailable exception occurred.

SPR 0x3D6

See also Section 5.3.4 on p. 5-10.

0	15	16	31
---	----	----	----

Figure 10-11. Exception Vector Prefix Register (EVPR)

0:15		Exception Vector Prefix
16:31		Reserved

GPR

See also Section 2.2.1 on p. 2-2.



Figure 10-12. General Purpose Register (R0-R31)



SPR 0x3F4

See also Section 7.6.4 on p. 7-10.

0	29	30	31
---	----	----	----

Figure 10-13. Instruction Address Compare Register (IAC1)			
0:29		Instruction Address Compare word address	Omit two low-order bits of complete address.
30:31		Reserved	

ICCR

SPR 0x3FB

See Section 8.1.3 on p. 8-6.

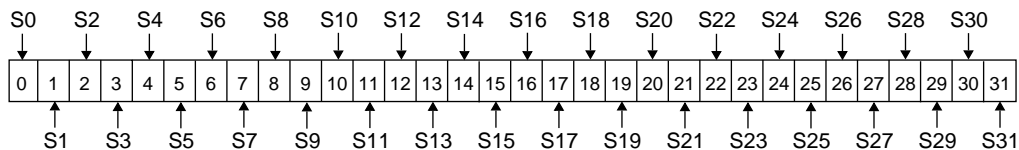


Figure 10-14. Instruction Cache Cacheability Register (ICCR)

0	S0	0 Noncacheable 1 Cacheable	0x0000 0000–0x07FF FFFF
1	S1	0 Noncacheable 1 Cacheable	0x0800 0000–0x0FFF FFFF
2	S2	0 Noncacheable 1 Cacheable	0x1000 0000–0x17FF FFFF
3	S3	0 Noncacheable 1 Cacheable	0x1800 0000–0x1FFF FFFF
4	S4	0 Noncacheable 1 Cacheable	0x2000 0000–0x27FF FFFF
5	S5	0 Noncacheable 1 Cacheable	0x2800 0000–0x2FFF FFFF
6	S6	0 Noncacheable 1 Cacheable	0x3000 0000–0x37FF FFFF
7	S7	0 Noncacheable 1 Cacheable	0x3800 0000–0x3FFF FFFF
8	S8	0 Noncacheable 1 Cacheable	0x4000 0000–0x47FF FFFF
9	S9	0 Noncacheable 1 Cacheable	0x4800 0000–0x4FFF FFFF
10	S10	0 Noncacheable 1 Cacheable	0x5000 0000–0x57FF FFFF
11	S11	0 Noncacheable 1 Cacheable	0x5800 0000–0x5FFF FFFF
12	S12	0 Noncacheable 1 Cacheable	0x6000 0000–0x67FF FFFF
13	S13	0 Noncacheable 1 Cacheable	0x6800 0000–0x6FFF FFFF

Figure 10-14. Instruction Cache Cacheability Register (ICCR) (cont.)

14	S14	0 Noncacheable 1 Cacheable	0x7000 0000–0x77FF FFFF
15	S15	0 Noncacheable 1 Cacheable	0x7800 0000–0x7FFF FFFF
16	S16	0 Noncacheable 1 Cacheable	0x8000 0000–0x87FF FFFF
17	S17	0 Noncacheable 1 Cacheable	0x8800 0000–0x8FFF FFFF
18	S18	0 Noncacheable 1 Cacheable	0x9000 0000–0x97FF FFFF
19	S19	0 Noncacheable 1 Cacheable	0x9800 0000–0x9FFF FFFF
20	S20	0 Noncacheable 1 Cacheable	0xA000 0000–0xA7FF FFFF
21	S21	0 Noncacheable 1 Cacheable	0xA800 0000–0xAFFF FFFF
22	S22	0 Noncacheable 1 Cacheable	0xB000 0000–0xB7FF FFFF
23	S23	0 Noncacheable 1 Cacheable	0xB800 0000–0xBFFF FFFF
24	S24	0 Noncacheable 1 Cacheable	0xC000 0000–0xC7FF FFFF
25	S25	0 Noncacheable 1 Cacheable	0xC800 0000–0xCFFF FFFF
26	S26	0 Noncacheable 1 Cacheable	0xD000 0000–0xD7FF FFFF
27	S27	0 Noncacheable 1 Cacheable	0xD800 0000–0xDFFF FFFF
28	S28	0 Noncacheable 1 Cacheable	0xE000 0000–0xE7FF FFFF
29	S29	0 Noncacheable 1 Cacheable	0xE800 0000–0xEFFF FFFF
30	S30	0 Noncacheable 1 Cacheable	0xF000 0000–0xF7FF FFFF
31	S31	0 Noncacheable 1 Cacheable	0xF800 0000–0xFFFF FFFF

ICDBDR

SPR 0x3D3 Read-Only

See also Section 6.4 on p. 6-9.

0	31
---	----

Figure 10-15. Instruction Cache Debug Data Register (ICDBDR)

0:31		Instruction cache information	See icread , p. 9-74.
------	--	-------------------------------	------------------------------

Table 10-3. ICU Tag Information

0:22	TAG	Cache Tag
23:24		Reserved
25	LK	Cache Line Lock 0 Unlocked 1 Locked
26		Reserved
27	V	Cache Line Valid 0 Not valid 1 Valid
28:30		Reserved
31	LRU	Least Recently Used (LRU) 0 A-way LRU 1 B-way LRU

SPR 0x008

See also Section 2.2.2.2 on p. 2-5.

0	31
---	----

Figure 10-16. Link Register (LR)

0:31		Link Registers contents	If (LR) represents an instruction address, LR _{0:31} should be zero.
------	--	-------------------------	---

MSR

See also Section 5.3.1 on p. 5-7.

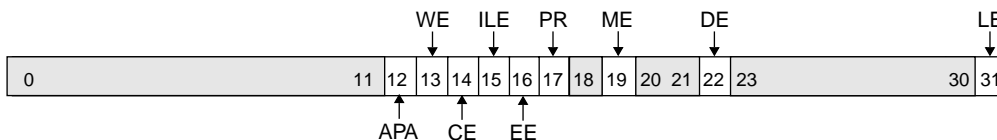


Figure 10-17. Machine State Register (MSR)

0:12		Reserved	
12	APA	Auxiliary Processor Available 0 Auxiliary processor not available. 1 Auxiliary processor available.	
13	WE	Wait State Enable 0 The processor is not in the wait state. 1 The processor is in the wait state.	If MSR[WE] = 1, the processor remains in the wait state until an exception is taken, a reset occurs, or an external debug tool clears WE.
14	CE	Critical Interrupt Enable 0 Critical interrupts are disabled. 1 Critical interrupts are enabled.	Controls the critical interrupt input and watchdog timer first time-out interrupts.
15	ILE	Interrupt Little Endian 0 Interrupt handlers execute in big endian mode. 1 Interrupt handlers execute in PowerPC little endian mode.	Copied to MSR(LE) when an interrupt is taken.
16	EE	External Interrupt Enable 0 Asynchronous exceptions are disabled. 1 Asynchronous exceptions are enabled.	Controls the non-critical external interrupt input, Programmable Interval Timer, and Fixed Interval Timer interrupts.
17	PR	Problem State 0 Supervisor State (all instructions allowed) 1 Problem State (some instructions not allowed)	
18		Reserved	
19	ME	Machine Check Enable 0 Machine check exceptions are disabled 1 Machine check exceptions are enabled.	
20:21		Reserved	
22	DE	Debug Exception Enable 0 Debug exceptions are disabled. 1 Debug exceptions are enabled.	

Figure 10-17. Machine State Register (MSR) (cont.)

23:30		Reserved
31	LE	Little Endian 0 Processor executes in big endian mode. 1 Processor executes in PowerPC little endian mode.

PIT

SPR 0x3DB

See also Section 5.15.2 on p. 5-28.

0	31
---	----

Figure 10-18. Programmable Interval Timer (PIT)

0:31		Programmed interval remaining	Number of clocks remaining until the PIT event
------	--	-------------------------------	--

SPR 0x11F Read-Only

See also Section 2.2.2.5 on p. 2-8.

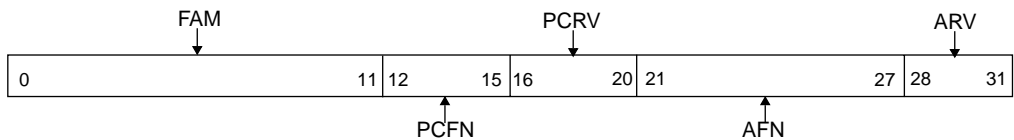


Figure 10-19. Processor Version Register (PVR)

0:11	FAM	Processor Family. Identifies a PowerPC family, such as 4xx or 6xx.	0x002 for the 4xx family.
12:15	PCFN	Processor Core Function. Identifies a specific processor core implementation.	1 for PPC401A1.
16:20	PCRV	Processor Core Revision. Identifies a revision of the processor core defined by the PFN field.	.
21:27	AFN	ASIC Function. An assigned identifier for an ASIC containing a PowerPC 400 Series processor core.	
28:31	ARV	ASIC Revision. An assigned identifier for a revision of the ASIC defined by the AFN field.	

SGR

SPR 0x3B9

See Section 8.1.4 on p. 8-8.

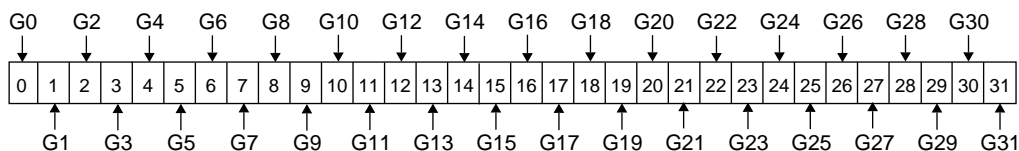


Figure 10-20. Storage Guarded Register (SGR)

0	G0	0 Normal 1 Guarded	0x0000 0000–0x07FF FFFF
1	G1	0 Normal 1 Guarded	0x0800 0000–0x0FFF FFFF
2	G2	0 Normal 1 Guarded	0x1000 0000–0x17FF FFFF
3	G3	0 Normal 1 Guarded	0x1800 0000–0x1FFF FFFF
4	G4	0 Normal 1 Guarded	0x2000 0000–0x27FF FFFF
5	G5	0 Normal 1 Guarded	0x2800 0000–0x2FFF FFFF
6	G6	0 Normal 1 Guarded	0x3000 0000–0x37FF FFFF
7	G7	0 Normal 1 Guarded	0x3800 0000–0x3FFF FFFF
8	G8	0 Normal 1 Guarded	0x4000 0000–0x47FF FFFF
9	G9	0 Normal 1 Guarded	0x4800 0000–0x4FFF FFFF
10	G10	0 Normal 1 Guarded	0x5000 0000–0x57FF FFFF
11	G11	0 Normal 1 Guarded	0x5800 0000–0x5FFF FFFF
12	G12	0 Normal 1 Guarded	0x6000 0000–0x67FF FFFF
13	G13	0 Normal 1 Guarded	0x6800 0000–0x6FFF FFFF

Figure 10-20. Storage Guarded Register (SGR) (cont.)

14	G14	0 Normal 1 Guarded	0x7000 0000–0x77FF FFFF
15	G15	0 Normal 1 Guarded	0x7800 0000–0x7FFF FFFF
16	G16	0 Normal 1 Guarded	0x8000 0000–0x87FF FFFF
17	G17	0 Normal 1 Guarded	0x8800 0000–0x8FFF FFFF
18	G18	0 Normal 1 Guarded	0x9000 0000–0x97FF FFFF
19	G19	0 Normal 1 Guarded	0x9800 0000–0x9FFF FFFF
20	G20	0 Normal 1 Guarded	0xA000 0000–0xA7FF FFFF
21	G21	0 Normal 1 Guarded	0xA800 0000–0xAFFF FFFF
22	G22	0 Normal 1 Guarded	0xB000 0000–0xB7FF FFFF
23	G23	0 Normal 1 Guarded	0xB800 0000–0xBFFF FFFF
24	G24	0 Normal 1 Guarded	0xC000 0000–0xC7FF FFFF
25	G25	0 Normal 1 Guarded	0xC800 0000–0xCFFF FFFF
26	G26	0 Normal 1 Guarded	0xD000 0000–0xD7FF FFFF
27	G27	0 Normal 1 Guarded	0xD800 0000–0xDFFF FFFF
28	G28	0 Normal 1 Guarded	0xE000 0000–0xE7FF FFFF
29	G29	0 Normal 1 Guarded	0xE800 0000–0xEFFF FFFF
30	G30	0 Normal 1 Guarded	0xF000 0000–0xF7FF FFFF
31	G31	0 Normal 1 Guarded	0xF800 0000–0xFFFF FFFF

SLER

SPR 0x3BB

See Section 8.1.5 on p. 8-10.

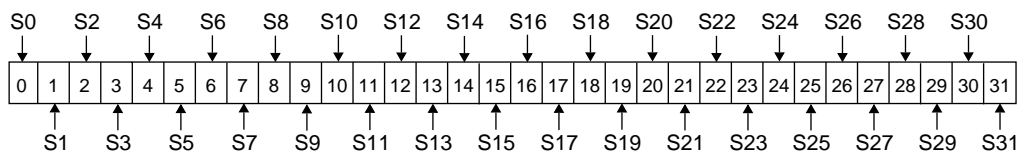


Figure 10-21. Storage Little-Endian Register (SLER)

0	S0	0 Big endian or PowerPC little endian 1 True little endian	0x0000 0000–0x07FF FFFF
1	S1	0 Big endian or PowerPC little endian 1 True little endian	0x0800 0000–0x0FFF FFFF
2	S2	0 Big endian or PowerPC little endian 1 True little endian	0x1000 0000–0x17FF FFFF
3	S3	0 Big endian or PowerPC little endian 1 True little endian	0x1800 0000–0x1FFF FFFF
4	S4	0 Big endian or PowerPC little endian 1 True little endian	0x2000 0000–0x27FF FFFF
5	S5	0 Big endian or PowerPC little endian 1 True little endian	0x2800 0000–0x2FFF FFFF
6	S6	0 Big endian or PowerPC little endian 1 True little endian	0x3000 0000–0x37FF FFFF
7	S7	0 Big endian or PowerPC little endian 1 True little endian	0x3800 0000–0x3FFF FFFF
8	S8	0 Big endian or PowerPC little endian 1 True little endian	0x4000 0000–0x47FF FFFF
9	S9	0 Big endian or PowerPC little endian 1 True little endian	0x4800 0000–0x4FFF FFFF
10	S10	0 Big endian or PowerPC little endian 1 True little endian	0x5000 0000–0x57FF FFFF
11	S11	0 Big endian or PowerPC little endian 1 True little endian	0x5800 0000–0x5FFF FFFF
12	S12	0 Big endian or PowerPC little endian 1 True little endian	0x6000 0000–0x67FF FFFF
13	S13	0 Big endian or PowerPC little endian 1 True little endian	0x6800 0000–0x6FFF FFFF

Figure 10-21. Storage Little-Endian Register (SLER) (cont.)

14	S14	0 Big endian or PowerPC little endian 1 True little endian	0x7000 0000–0x77FF FFFF
15	S15	0 Big endian or PowerPC little endian 1 True little endian	0x7800 0000–0x7FFF FFFF
16	S16	0 Big endian or PowerPC little endian 1 True little endian	0x8000 0000–0x87FF FFFF
17	S17	0 Big endian or PowerPC little endian 1 True little endian	0x8800 0000–0x8FFF FFFF
18	S18	0 Big endian or PowerPC little endian 1 True little endian	0x9000 0000–0x97FF FFFF
19	S19	0 Big endian or PowerPC little endian 1 True little endian	0x9800 0000–0x9FFF FFFF
20	S20	0 Big endian or PowerPC little endian 1 True little endian	0xA000 0000–0xA7FF FFFF
21	S21	0 Big endian or PowerPC little endian 1 True little endian	0xA800 0000–0xAFFF FFFF
22	S22	0 Big endian or PowerPC little endian 1 True little endian	0xB000 0000–0xB7FF FFFF
23	S23	0 Big endian or PowerPC little endian 1 True little endian	0xB800 0000–0xBFFF FFFF
24	S24	0 Big endian or PowerPC little endian 1 True little endian	0xC000 0000–0xC7FF FFFF
25	S25	0 Big endian or PowerPC little endian 1 True little endian	0xC800 0000–0xCFFF FFFF
26	S26	0 Big endian or PowerPC little endian 1 True little endian	0xD000 0000–0xD7FF FFFF
27	S27	0 Big endian or PowerPC little endian 1 True little endian	0xD800 0000–0xDFFF FFFF
28	S28	0 Big endian or PowerPC little endian 1 True little endian	0xE000 0000–0xE7FF FFFF
29	S29	0 Big endian or PowerPC little endian 1 True little endian	0xE800 0000–0xEFFF FFFF
30	S30	0 Big endian or PowerPC little endian 1 True little endian	0xF000 0000–0xF7FF FFFF
31	S31	0 Big endian or PowerPC little endian 1 True little endian	0xF800 0000–0xFFFF FFFF

SPRG0–SPRG3

SPR 0x110-0x113

See also Section 2.2.2.4 on p. 2-8.

0	31
---	----

Figure 10-22. Special Purpose Register General (SPRG0-SPRG3)

0-31		General data	Privileged user-specified; no hardware usage.
------	--	--------------	---

SPR 0x01A

See also Section 5.3.2 on p. 5-8.

0	29	30	31
---	----	----	----

Figure 10-23. Save/Restore Register 0 (SRR0)

0:29		SRR0 receives an instruction address when a non-critical interrupt is taken; the Program Counter is restored from SRR0 when rfi executes.
30:31		Reserved

SRR1

SPR 0x01B

See also Section 5.3.2 on p. 5-8.

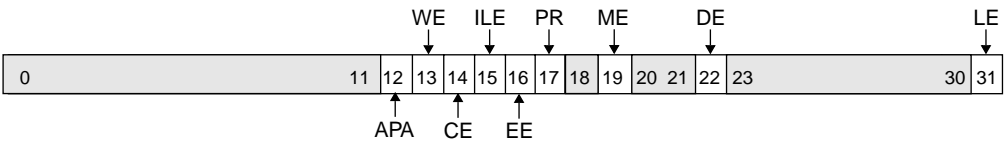


Figure 10-24. Save/Restore Register 1 (SRR1)

0:31	SRR1 receives a copy of the MSR when a non-critical interrupt is taken; the MSR is restored from SRR1 when rfi executes.
------	---

SPR 0x3DE

See also Section 5.3.3 on p. 5-9.

0	29	30	31
---	----	----	----

Figure 10-25. Save/Restore Register 2 (SRR2)

0:29		SRR2 receives an instruction address when a critical interrupt is taken; the Program Counter is restored from SRR2 when rfci executes.
30:31		Reserved

SRR3

SPR 0x3DF

See also Section 5.3.3 on p. 5-9.

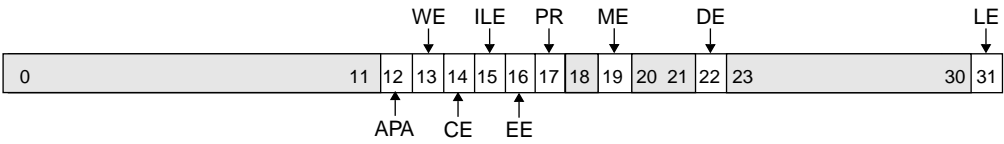


Figure 10-26. Save/Restore Register 1 (SRR1)

0:31	SRR3 receives a copy of the MSR when a critical interrupt is taken; the MSR is restored from SRR3 when rfci executes.
------	--

SPR 0x3DC

See also Section 5.15.1 on p. 5-25.



Figure 10-27. Time Base High Register (TBHI)

0:31		Time High	Current count, high-order.
------	--	-----------	----------------------------

TBHU

SPR 0x3CC Read-Only

See also Section 5.15.1 on p. 5-25.

0	31
---	----

Figure 10-28. Time Base High User-mode (TBHU)

0:31		Time High	Current count, high-order. Note: TBHU is a read-only access vehicle to the time base register TBHI. It is not possible for the contents of TBHU and TBHI to differ.
------	--	-----------	---

SPR 0x3DD

See also Section 5.15.1 on p. 5-25.



Figure 10-29. Time Base Low Register (TBLO)

0:31		Time Low	Current count, low-order.
------	--	----------	---------------------------

TBLU

SPR 0x3CD Read-Only

See also Section 5.15.1 on p. 5-25.

0	31
---	----

Figure 10-30. Time Base Low User-mode (TBLU)

0:31		Time Low	Current count, low-order. Note: TBLU is a read-only access vehicle to the time base register TBLO. It is not possible for the contents of TBLU and TBLO to differ.
------	--	----------	--

SPR 0x3DA

See Section 5.15.6 on p. 5-33.

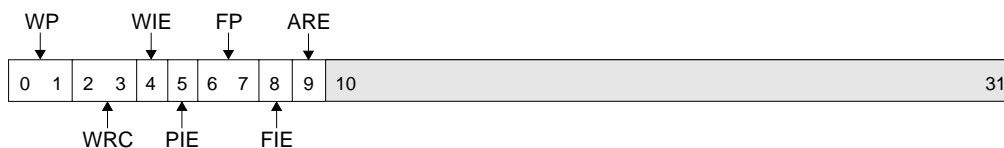


Figure 10-31. Timer Control Register (TCR)

0:1	WP	Watchdog Period 00 2^{17} clocks 01 2^{21} clocks 10 2^{25} clocks 11 2^{29} clocks	
2:3	WRC	Watchdog Reset Control 00 No Watchdog reset will occur. 01 Core reset will be forced by the Watchdog. 10 Chip reset will be forced by the Watchdog. 11 System reset will be forced by the Watchdog.	TCR[WRC] resets to 00. This field can be set by software, but cannot be cleared by software, except by a software-induced reset.
4	WIE	Watchdog Interrupt Enable 0 Disable WDT interrupt. 1 Enable WDT interrupt.	
5	PIE	PIT Interrupt Enable 0 Disable PIT interrupt. 1 Enable PIT interrupt.	
6:7	FP	FIT Period 00 2^9 clocks 01 2^{13} clocks 10 2^{17} clocks 11 2^{21} clocks	
8	FIE	FIT Interrupt Enable 0 Disable FIT interrupt. 1 Enable FIT interrupt.	
9	ARE	Auto Reload Enable 0 Disable auto reload. 1 Enable auto reload.	Disables on reset.
10:31		Reserved	

TSR

SPR 0x3D8 Read/Clear

See Section 5.15.5 on p. 5-32.

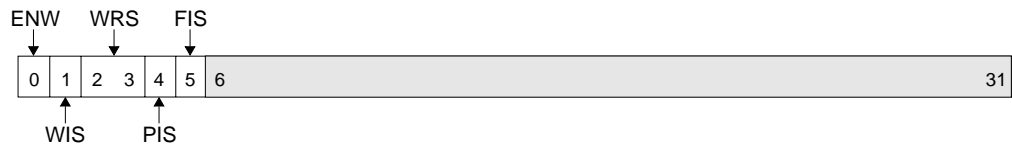


Figure 10-32. Timer Status Register (TSR)

0	ENW	Enable Next Watchdog 0 Action on next Watchdog event is to set TSR[0]. 1 Action on next Watchdog event is governed by TSR[1]. See Section 5.15.4 on p. 5-30.
1	WIS	Watchdog Interrupt Status 0 No Watchdog interrupt is pending. 1 Watchdog interrupt is pending.
2:3	WRS	Watchdog Reset Status 00 No Watchdog reset has occurred. 01 Core reset was forced by the Watchdog. 10 Chip reset was forced by the Watchdog. 11 System reset was forced by the Watchdog.
4	PIS	PIT Interrupt Status 0 No PIT interrupt is pending. 1 PIT interrupt is pending.
5	FIS	FIT Interrupt Status 0 No FIT interrupt is pending. 1 FIT interrupt is pending.
6:31		Reserved

SPR 0x001

See Section 2.2.2.3 on p. 2-6.

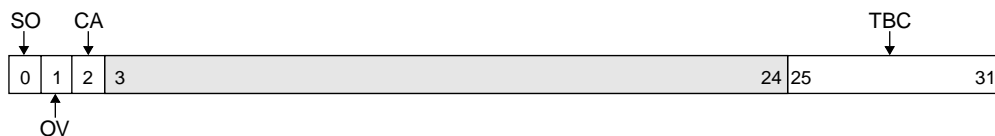


Figure 10-33. Fixed Point Exception Register (XER)

0	SO	Summary Overflow 0 No overflow has occurred. 1 Overflow has occurred.	Can be <i>set</i> by mtspr or using arithmetic instructions with the “OE” option (see Table 2-2 on p. 2-7); can be <i>reset</i> by mtspr or by mcrxr .
1	OV	Overflow 0 No overflow has occurred. 0 Overflow has occurred.	Can be <i>set</i> by mtspr or arithmetic instructions with the “OE” option (see Table 2-2 on p. 2-7); can be <i>reset</i> by mtspr , by mcrxr , or by arithmetic instructions with the “OE” option.
2	CA	Carry 0 Carry has not occurred. 1 Carry has occurred.	Can be <i>set</i> by mtspr or arithmetic instructions that update the CA field (see Table 2-2 on p. 2-7); can be <i>reset</i> by mtspr , by mcrxr , or by arithmetic instructions that update the CA field.
3:24		Reserved	
25:31	TBC	Transfer Byte Count	Used by lswx and stswx ; written by mtspr

Table 11-2 lists the PPC401A1 signals in alphabetical order. The table also lists the hard macro name, I/O type, termination if unused, and timing, and provides a page reference.

11.1 Signal Naming Conventions

Signal names used in this chapter follow the format defined below:

PREFIX1_prefix2SigName1[_sigName2][_ NEG][*(m:n)*]

where:

- PREFIX1 is an *uppercase* prefix identifying the source of the signal, either by unit name (such as CPU, DCU) or by the interface type (such as DCR, LSSD).
- _prefix2 is a *lowercase* prefix identifying the destination of the signal, either by unit name (e.g. cpu, dcu) or by the interface type (e.g. dcr, lssd).
- SigName1 is a *mixed case* name reflecting the primary function of the signal.
- [sigName2] (optional) is a *mixed case* name reflecting the secondary function of the signal, and is separated from SigName1 by an underscore.
- [_ NEG] (optional) denotes that a signal is active low. Unless so denoted, all signals are active high.
- [*(m:n)*] (optional) indicates a bussed signal. This notation is for reading convenience only; the actual core signal names must be unbussed, 1-bit signal names. "Bussed" signals appear on the macro symbol in an expanded format, each including the base name and a one or two digit suffix (depending on whether the highest bit position must be represented by one or two digits) identifying that signal's bit position on the bus.

Actual PPC401A1 hard macro signal names do not follow the format above. These names must be unbussed, and contain only uppercase letters and numbers. Hard macro signal names are limited to seven characters. Other characters, such as underscores, are illegal. These restrictions make the hard core macro compatible with a variety of vendor development tools for chip design, simulation, synthesis, timing, and so on. The hard macro

signal names appear in the signal descriptions for each in PPC401A1 interface, which are organized by the signal names that follow the naming conventions.

Such signal names are used throughout this chapter to ease reading of the PPC401A1 signal names and more clearly identify function. These signal names also appear at the test mode matrix (TMM) boundary, an unsynthesized “soft” core that provides a test wrapper around the PPC401A1. These soft cores need not follow the naming restrictions enforced for the hard cores.

Table 11-1 defines the prefixes used in the signal names. The second column in the table identifies whether the prefix refers to a logic unit that resides *on* the PPC401A1 or *off* of the PPC401A1. For example, the DCU unit, which resides on the PPC401A1, is an on-core unit. The CPM unit, which does not reside on the PPC401A1, is an off-core unit, is not part of the PPC401A1.

Table 11-1. Signal Name Prefix Definitions

Prefix1 (prefix2)	Definition	On/Off Core
BIU (biu)	Bus Interface Unit	Off
CORE (core)	Generic PPC401xx core identifier	On
CPM (cpm)	Clock and power management	Off
CPU (core)	Core CPU	On
DBG (dbg)	Debug unit	On
DCR (dcr)	Device Control Register	Off
DCU (dcu)	Data cache unit	On
EIC (eic)	External interrupt controller	Off
ICU (icu)	Instruction cache unit	On
JTAG (jtag)	JTAG	On
LSSD (lssd)	Level-sensitive scan design	Off
TIE (tie)	TIE (statically, to GND or V_{DD})	Off
TST	Test/TMM mode control	Off
XXX (xxx)	Unspecified ASIC unit	Off

Table 11-2. Ford Signal I/O Sorted Alphabetically by Signal Name

Signal	Core Name	I/O	If Unused	Page
BIU_cpuDCUBusy	CPBSYDK	I	0	3-57
BIU_dbgHoldAck	DBHLDK	I	0	3-109
BIU_dcuAccept	DKACCPT	I	0	3-54
BIU_dcuBusErr	DKBERR	I	0	3-57
BIU_dcuDValid	DKDVAL	I	0	3-54
BIU_dcuRdDBus(0:31)	DKDInn	I	0x00000000	3-55
BIU_dcuWdAddr(0:1)	DKWADRn	I	0b00	3-55
BIU_dcuWrDAck	DKWDACK	I	0	3-56
BIU_icuAccept	IKACCPT	I	0	3-32
BIU_icuBusErr	IKBERR	I	0	3-34
BIU_icuDValid	IKDVAL	I	0	3-32
BIU_icuRdDBus(0:31)	IKDInn	I	0x00000000	3-33
BIU_icuWdAddr(0:1)	IKWADRn	I	0b00	3-33
CORE_issdScanout(0:2)	LSSDSOn	O	No connect	3-18
CPM_cpuAllowCoreClk	CMALWCC	I	1	3-8
CPM_cpuAllowTimerClk	CMALWTC	I	1	3-8
CPM_cpuCoreClkOff	CMCLOFF	I	0	3-8
CPM_cpuSystemClock	CMSYSCK	I	Required	3-7
CPU_biuMsrWait	CPMSRWE	O	No connect	3-11
CPU_biuStopAck	CPSTPAK	O	No connect	3-11
CPU_cpmCoreSleepReq	CMSLPRQ	O	No connect	3-9
CPU_cpmMsrCE	CMMSRCE	O	No connect	3-9
CPU_cpmMsrEE	CMMSREE	O	No connect	3-9
CPU_cpmTimerIntrp	CMTMINT	O	No connect	3-9
CPU_cpmTimerReset	CMTMRST	O	No connect	3-9
CPU_dcrABus(0:9)	DCRADRn	O	No connect	3-89
CPU_dcrDBus(0:31)	DCRDOnn	O	No connect or wrap to DBusIn	3-89
CPU_dcrMFDCR	DCRMVFR	O	No connect	3-88
CPU_dcrMTDCR	DCRMVTO	O	No connect	3-89
CPU_iosSysErrorOut	CPERROR	O	No connect	3-11
DBG_biuDMAOverride	DBDMAOV	O	No connect	3-53
DBG_extReqChipReset	DBRSTCH	O	Required	3-24
DBG_extReqCoreReset	DBRSTCO	O	Required	3-24

Table 11-2. Ford Signal I/O Sorted Alphabetically by Signal Name

Signal	Core Name	I/O	If Unused	Page
DBG_extReqSystemReset	DBRSTSY	O	Required	3-25
DCR_cpuDAck	DCRDACK	I	0	3-89
DCR_cpuDBus(0:31)	DCRDInn	I	0x00000000 or wrapped DBusOut	3-90
DCU_biuABus(0:31)	DKADRnn	O	No connect	3-48
DCU_biuAbort	DKABORT	O	No connect	3-53
DCU_biuBytesNLine	DKBYTEX	O	No connect	3-49
DCU_biuDTags(0:3)	DKDTAGn	O	No connect	3-50
DCU_biuGuardedMem	DKGUARD	O	No connect	3-51
DCU_biuPriority	DKHIPRI	O	No connect	3-52
DCU_biuRNW	DKRDNRW	O	No connect	3-48
DCU_biuRequest	DKREQST	O	No connect	3-47
DCU_biuWrDBus(0:31)	DKDONn	O	No connect	3-53
EIC_cpuCritical	EIINTCR	I	0	3-101
EIC_cpuNonCritical	EIINTNC	I	0	3-100
EXT_chipReset	XRSTCHP	I	Required	3-26
EXT_coreReset	XRSTCOR	I	Required	3-25
EXT_jtagBScanTDO	JTBNDISI	I	0	3-105
EXT_sysReset	XRSTSYS	I	Required	3-26
ICU_biuABus(0:29)	IKADRnn	O	No connect	3-30
ICU_biuAbort	IKABORT	O	No connect	3-31
ICU_biuGuardedMem	IKGUARD	O	No connect	3-31
ICU_biuRequest	IKREQST	O	No connect	3-29
ICU_biuWordNLine	IKWORDX	O	No connect	3-30
IOS_cacheArrayTest	XARYTST	I	0	3-18
IOS_dbgDebugHalt	DBHALT	I	0	3-109
IOS_gprScanGate	XGPRSG	I	1	3-18
IOS_jtagResetIn	JTRSTIN	I	Required	3-105
IOS_jtagTestClock(TCK)	JTTCLK	I	See IEEE 1149.1	3-104
IOS_jtagTestDataIn(TDI)	JTTDI	I	1	3-105
IOS_jtagTestModeSelect(TMS)	JTTMS	I	1	3-105
JTAG_extCaptureDR	JTCAPDR	O	No connect	3-106
JTAG_extExttest	JTEXTST	O	No connect	3-106
JTAG_extShiftDR	JTSHFDR	O	No connect	3-106
JTAG_extUpdateDR	JTUPDDR	O	No connect	3-107
JTAG_iosTestDataOut(TDO)	JTTDO	O	No connect	3-106

Table 11-2. Ford Signal I/O Sorted Alphabetically by Signal Name

Signal	Core Name	I/O	If Unused	Page
JTAG_iosTestDataOutEn	JTTDOEN	O	No connect	3-106
LSSD_coreScanin(0:2)	LSSDSIn	I	0b000	3-18
LSSD_scanAClk	LSSDA	I	0	3-16
LSSD_scanCClk	LSSDC	I	1	3-17
TIE_cpuPVR(0:31)	CPPVRnn	I	Required	3-11
TMM_LSSDScanBClk	TMLSSDB	I	0	3-16
TMM_LSSDTest	TMLSSDT	I	Required	3-16
TMM_coreTest	TMCORET	I	Required	3-15
TMM_scanIn	TMSCANI	I	0	3-16
TMM_scanOut	TMSCANO	O	No connect	3-16

Table 11-3. Ford Signal I/O Sorted Alphabetically by Hard Macro Name

Core Name	Signal	I/O	If Unused	Page
CMALWCC	CPM_cpuAllowCoreClk	I	1	3-8
CMALWTC	CPM_cpuAllowTimerClk	I	1	3-8
CMCLOFF	CPM_cpuCoreClkOff	I	0	3-8
CMMSRCE	CPU_cpmMsrCE	O	No connect	3-9
CMMSREE	CPU_cpmMsrEE	O	No connect	3-9
CMSLPRQ	CPU_cpmCoreSleepReq	O	No connect	3-9
CMSYSCK	CPM_cpuSystemClock	I	Required	3-9
CMTMINT	CPU_cpmTimerIntrp	O	No connect	3-9
CMTMRST	CPU_cpmTimerReset	O	No connect	3-9
CPBSYDK	BIU_cpuDCUBusy	I	0	3-57
CPERROR	CPU_iosSysErrorOut	O	No connect	3-11
CPMSRWE	CPU_biuMsrWait	O	No connect	3-11
CPPVRnn	TIE_cpuPVR(0:31)	I	Required	3-11
CPSTPAK	CPU_biuStopAck	O	No connect	3-11
DBDMAOV	DBG_biuDMAOverride	O	No connect	3-53
DBHALT	IOS_dbgDebugHalt	I	0	3-109
DBHLDK	BIU_dbgHoldAck	I	0	3-109
DBRSTCH	DBG_extReqChipReset	O	Required	3-24
DBRSTCO	DBG_extReqCoreReset	O	Required	3-24
DBRSTSY	DBG_extReqSystemReset	O	Required	3-25
DCRADRN	CPU_dcrABus(0:9)	O	No connect	3-89
DCRDACK	DCR_cpuDAck	I	0	3-89

Table 11-3. Ford Signal I/O Sorted Alphabetically by Hard Macro Name

Core Name	Signal	I/O	If Unused	Page
DCRDInn	DCR_cpuDBus(0:31)	I	0x0000 0000 or wrapped DBusOut	3-89
DCRDOnn	CPU_dcrDBus(0:31)	O	No connect, or wrap to DBusIn	3-90
DCRMVFR	CPU_dcrMFDCR	O	No connect	3-88
DCRMVTO	CPU_dcrMTDCR	O	No connect	3-89
DKABORT	DCU_biuAbort	O	No connect	3-53
DKACCPT	BIU_dcuAccept	I	0	3-54
DKADRnn	DCU_biuABus(0:31)	O	No connect	3-48
DKBERR	BIU_dcuBusErr	I	0	3-57
DKBYTEX	DCU_biuBytesNLine	O	No connect	3-49
DKDIInn	BIU_dcuRdDBus(0:31)	I	0x0000 0000	3-55
DKDOnn	DCU_biuWrDBus(0:31)	O	No connect	3-53
DKDTAGn	DCU_biuDTags(0:3)	O	No connect	3-50
DKDVAL	BIU_dcuDValid	I	0	3-54
DKGUARD	DCU_biuGuardedMem	O	No connect	3-51
DKHIPRI	DCU_biuPriority	O	No connect	3-52
DKRDNRW	DCU_biuRNW	O	No connect	3-48
DKREQST	DCU_biuRequest	O	No connect	3-47
DKWADRn	BIU_dcuWdAddr(0:1)	I	0b00	3-55
DKWDACK	BIU_dcuWrDAck	I	0	3-56
EIINTCR	EIC_cpuCritical	I	0	3-101
EIINTNC	EIC_cpuNonCritical	I	0	3-100
IKABORT	ICU_biuAbort	O	No connect	3-31
IKACCPT	BIU_icuAccept	I	0	3-32
IKADRnn	ICU_biuABus(0:29)	O	No connect	3-30
IKBERR	BIU_icuBusErr	I	0	3-34
IKDIInn	BIU_icuRdDBus(0:31)	I	0x0000 0000	3-33
IKDVAL	BIU_icuDValid	I	0	3-32
IKGUARD	ICU_biuGuardedMem	O	No connect	3-31
IKREQST	ICU_biuRequest	O	No connect	3-29
IKWADRn	BIU_icuWdAddr(0:1)	I	0b00	3-33
IKWORDX	ICU_biuWordNLine	O	No connect	3-30
JTBNSI	EXT_jtagBScanTDO	I	0	3-106
JTCAPDR	JTAG_extCaptureDR	O	No connect	3-106
JTEXTST	JTAG_extExtest	O	No connect	3-106

Table 11-3. Ford Signal I/O Sorted Alphabetically by Hard Macro Name

Core Name	Signal	I/O	If Unused	Page
JTRSTIN	IOS_jtagResetIn	I	Required	3-105
JTSHFDR	JTAG_extShiftDR	O	No connect	3-106
JTCLK	IOS_jtagTestClock(TCK)	I	See IEEE 1149.1	3-104
JTTDI	IOS_jtagTestDataIn(TDI)	I	1	3-105
JTTDO	JTAG_iosTestDataOut(TDO)	O	No connect	3-106
JTTDOEN	JTAG_iosTestDataOutEn	O	No connect	3-106
JTTMS	IOS_jtagTestModeSelect(TMS)	I	1	3-105
JTUPDDR	JTAG_extUpdateDR	O	No connect	3-107
LSSDA	LSSD_scanAClk	I	0	3-16
LSSDC	LSSD_scanCClk	I	1	3-17
LSSDSIn	LSSD_coreScanin(0:2)	I	0b000	3-18
LSSDSOn	CORE_IssdScanout(0:2)	O	No connect	3-18
TMCORET	TMM_coreTest	I	Required	3-15
TMLSSDB	TMM_LSSDScanBClk	I	0	3-16
TMLSSDT	TMM_LSSDTest	I	Required	3-16
TMSCANI	TMM_scanIn	I	0	3-16
TMSCANO	TMM_scanOut	O	No connect	3-16
XARYTST	IOS_cacheArrayTest	I	0	3-18
XGPRSG	IOS_gprScanGate	I	1	3-18
XRSTCHP	EXT_chipReset	I	Required	3-26
XRSTCOR	EXT_coreReset	I	Required	3-25
XRSTSYS	EXT_sysReset	I	Required	3-26



Instruction Summary

This appendix contains PPC401A1 instructions summarized alphabetically and by opcode.

- On p. A-1, Section A.1 lists all PPC401A1 mnemonics, including extended mnemonics, alphabetically. A short functional description is included for each mnemonic.
- On p. A-38, Section A.2 lists all PPC401A1 instructions, sorted by primary and secondary opcodes. Extended mnemonics are not included in the opcode list.
- On p. A-46, Section A.3 illustrates the PPC401A1 instruction forms (allowed arrangements of fields within instructions).

A.1 Instruction Set and Extended Mnemonics – Alphabetical

Table A-1 summarizes the PPC401A1 instruction set, including required extended mnemonics. All mnemonics are listed alphabetically, without regard to whether the mnemonic is realized in hardware or software. When an instruction supports multiple hardware mnemonics (for example, **b**, **ba**, **bl**, **bla** are all forms of **b**), the instruction is alphabetized under the root form. The hardware instructions are described in detail in Chapter 9 (Instruction Set) which is also alphabetized under the root form. Chapter 9 also describes the instruction operands and notation.

Note the following for every Branch Conditional mnemonic:

Bit 4 of the BO field provides a hint about the most likely outcome of a conditional branch (see Section 2.6.5 for a full discussion of Branch Prediction). Assemblers should set $BO_4 = 0$ unless a specific reason exists otherwise. In the BO field values specified in the table below, $BO_4 = 0$ has always been assumed. The assembler must allow the programmer to specify Branch Prediction. To do this, the assembler will support a suffix to every conditional branch mnemonic, as follows:

- + Predict branch to be taken.
- Predict branch not to be taken.

As specific examples, **bc** also could be coded as **bc+** or **bc–**, and **bne** also could be coded **bne+** or **bne–**. These alternate codings set $BO_4 = 1$ only if the requested prediction differs from the Standard Prediction (see Section 2.6.5).

Table A-1. PPC401A1 Instruction Syntax Summary

Mnemonic	Operands	Function	Other Registers Changed	Page
add	RT, RA, RB	Add (RA) to (RB). Place result in RT.		9-6
add.			CR[CR0]	
addo			XER[SO, OV]	
addo.			CR[CR0] XER[SO, OV]	
addc	RT, RA, RB	Add (RA) to (RB). Place result in RT. Place carry-out in XER[CA].		9-7
addc.			CR[CR0]	
addco			XER[SO, OV]	
addco.			CR[CR0] XER[SO, OV]	
adde	RT, RA, RB	Add XER[CA], (RA), (RB). Place result in RT. Place carry-out in XER[CA].		9-8
adde.			CR[CR0]	
addeo			XER[SO, OV]	
addeo.			CR[CR0] XER[SO, OV]	
addi	RT, RA, IM	Add EXTS(IM) to (RA 0). Place result in RT.		9-9
addic	RT, RA, IM	Add EXTS(IM) to (RA 0). Place result in RT. Place carry-out in XER[CA].		9-10
addic.	RT, RA, IM	Add EXTS(IM) to (RA 0). Place result in RT. Place carry-out in XER[CA].	CR[CR0]	9-11
addis	RT, RA, IM	Add (IM ¹⁶ 0) to (RA 0). Place result in RT.		9-12
addme	RT, RA	Add XER[CA], (RA), (-1). Place result in RT. Place carry-out in XER[CA].		9-13
addme.			CR[CR0]	
addmeo			XER[SO, OV]	
addmeo.			CR[CR0] XER[SO, OV]	

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
addze	RT, RA	Add XER[CA] to (RA). Place result in RT. Place carry-out in XER[CA].		9-14
addze.			CR[CR0]	
addzeo			XER[SO, OV]	
addzeo.			CR[CR0] XER[SO, OV]	
and	RA, RS, RB	AND (RS) with (RB). Place result in RA.		9-15
and.			CR[CR0]	
andc	RA, RS, RB	AND (RS) with \neg (RB). Place result in RA.		9-16
andc.			CR[CR0]	
andi.	RA, RS, IM	AND (RS) with ($^{16}0 \parallel$ IM). Place result in RA.	CR[CR0]	9-17
andis.	RA, RS, IM	AND (RS) with (IM \parallel $^{16}0$). Place result in RA.	CR[CR0]	9-18
b	target	Branch unconditional relative. $LI \leftarrow (target - CIA)_{6:29}$ $NIA \leftarrow CIA + EXTS(LI \parallel ^20)$		9-19
ba		Branch unconditional absolute. $LI \leftarrow target_{6:29}$ $NIA \leftarrow EXTS(LI \parallel ^20)$		
bl		Branch unconditional relative. $LI \leftarrow (target - CIA)_{6:29}$ $NIA \leftarrow CIA + EXTS(LI \parallel ^20)$	(LR) $\leftarrow CIA + 4.$	
bla		Branch unconditional absolute. $LI \leftarrow target_{6:29}$ $NIA \leftarrow EXTS(LI \parallel ^20)$	(LR) $\leftarrow CIA + 4.$	

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
bc	BO, BI, target	Branch conditional relative. $BD \leftarrow (target - CIA)_{16:29}$ $NIA \leftarrow CIA + EXTS(BD \parallel ^20)$	CTR if $BO_2 = 0$.	9-20
bca		Branch conditional absolute. $BD \leftarrow target_{16:29}$ $NIA \leftarrow EXTS(BD \parallel ^20)$	CTR if $BO_2 = 0$.	
bcl		Branch conditional relative. $BD \leftarrow (target - CIA)_{16:29}$ $NIA \leftarrow CIA + EXTS(BD \parallel ^20)$	CTR if $BO_2 = 0$. (LR) $\leftarrow CIA + 4$.	
bcla		Branch conditional absolute. $BD \leftarrow target_{16:29}$ $NIA \leftarrow EXTS(BD \parallel ^20)$	CTR if $BO_2 = 0$. (LR) $\leftarrow CIA + 4$.	
bcctr	BO, BI	Branch conditional to address in CTR. Using (CTR) at exit from instruction, $NIA \leftarrow CTR_{0:29} \parallel ^20$.	CTR if $BO_2 = 0$.	9-28
bcctrl			CTR if $BO_2 = 0$. (LR) $\leftarrow CIA + 4$.	
bclr	BO, BI	Branch conditional to address in LR. Using (LR) at entry to instruction, $NIA \leftarrow LR_{0:29} \parallel ^20$.	CTR if $BO_2 = 0$.	9-32
bctrl			CTR if $BO_2 = 0$. (LR) $\leftarrow CIA + 4$.	
bctr		Branch unconditionally, to address in CTR. <i>Extended mnemonic for</i> bcctr 20,0		9-28
bctrl		<i>Extended mnemonic for</i> bcctrl 20,0	(LR) $\leftarrow CIA + 4$.	
bdnz	target	Decrement CTR. Branch if $CTR \neq 0$. <i>Extended mnemonic for</i> bc 16,0,target		9-20
bdnza		<i>Extended mnemonic for</i> bca 16,0,target		
bdnzl		<i>Extended mnemonic for</i> bcl 16,0,target	(LR) $\leftarrow CIA + 4$.	
bdnzla		<i>Extended mnemonic for</i> bcla 16,0,target	(LR) $\leftarrow CIA + 4$.	

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
bdnzlr		Decrement CTR. Branch if CTR \neq 0, to address in LR. <i>Extended mnemonic for</i> bclr 16,0		9-32
bdnzlrl		<i>Extended mnemonic for</i> bclrl 16,0	(LR) \leftarrow CIA + 4.	
bdnzf	cr_bit, target	Decrement CTR. Branch if CTR \neq 0 AND CR _{cr_bit} = 0. <i>Extended mnemonic for</i> bc 0,cr_bit,target		9-20
bdnzfa		<i>Extended mnemonic for</i> bca 0,cr_bit,target		
bdnzfl		<i>Extended mnemonic for</i> bcl 0,cr_bit,target	(LR) \leftarrow CIA + 4.	
bdnzfla		<i>Extended mnemonic for</i> bcla 0,cr_bit,target	(LR) \leftarrow CIA + 4.	
bdnzflr	cr_bit	Decrement CTR. Branch if CTR \neq 0 AND CR _{cr_bit} = 0, to address in LR. <i>Extended mnemonic for</i> bclr 0,cr_bit		9-32
bdnzflrl		<i>Extended mnemonic for</i> bclrl 0,cr_bit	(LR) \leftarrow CIA + 4.	
bdnzt	cr_bit, target	Decrement CTR. Branch if CTR \neq 0 AND CR _{cr_bit} = 1. <i>Extended mnemonic for</i> bc 8,cr_bit,target		9-20
bdnzta		<i>Extended mnemonic for</i> bca 8,cr_bit,target		
bdnztl		<i>Extended mnemonic for</i> bcl 8,cr_bit,target	(LR) \leftarrow CIA + 4.	
bdnztla		<i>Extended mnemonic for</i> bcla 8,cr_bit,target	(LR) \leftarrow CIA + 4.	

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
bdnztlr	cr_bit	Decrement CTR. Branch if CTR \neq 0 AND CR _{cr_bit} = 1, to address in LR. <i>Extended mnemonic for</i> bclr 8,cr_bit		9-32
bdnztlrl		<i>Extended mnemonic for</i> bclrl 8,cr_bit	(LR) \leftarrow CIA + 4.	
bdz	target	Decrement CTR. Branch if CTR = 0. <i>Extended mnemonic for</i> bc 18,0,target		9-20
bdza		<i>Extended mnemonic for</i> bca 18,0,target		
bdzl		<i>Extended mnemonic for</i> bcl 18,0,target	(LR) \leftarrow CIA + 4.	
bdzla		<i>Extended mnemonic for</i> bcla 18,0,target	(LR) \leftarrow CIA + 4.	
bdzlr		Decrement CTR. Branch if CTR = 0, to address in LR. <i>Extended mnemonic for</i> bclr 18,0		9-32
bdzlrl		<i>Extended mnemonic for</i> bclrl 18,0	(LR) \leftarrow CIA + 4.	
bdzf	cr_bit, target	Decrement CTR. Branch if CTR = 0 AND CR _{cr_bit} = 0. <i>Extended mnemonic for</i> bc 2,cr_bit,target		9-20
bdzfa		<i>Extended mnemonic for</i> bca 2,cr_bit,target		
bdzfl		<i>Extended mnemonic for</i> bcl 2,cr_bit,target	(LR) \leftarrow CIA + 4.	
bdzfla		<i>Extended mnemonic for</i> bcla 2,cr_bit,target	(LR) \leftarrow CIA + 4.	

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
bdzflr	cr_bit	Decrement CTR. Branch if CTR = 0 AND CR _{cr_bit} = 0 to address in LR. <i>Extended mnemonic for</i> bclr 2,cr_bit		9-32
bdzflrl		<i>Extended mnemonic for</i> bclrl 2,cr_bit	(LR) ← CIA + 4.	
bdzt	cr_bit, target	Decrement CTR. Branch if CTR = 0 AND CR _{cr_bit} = 1. <i>Extended mnemonic for</i> bc 10,cr_bit,target		9-20
bdzta		<i>Extended mnemonic for</i> bca 10,cr_bit,target		
bdztl		<i>Extended mnemonic for</i> bcl 10,cr_bit,target	(LR) ← CIA + 4.	
bdztla		<i>Extended mnemonic for</i> bcla 10,cr_bit,target	(LR) ← CIA + 4.	
bdztlr	cr_bit	Decrement CTR. Branch if CTR = 0 AND CR _{cr_bit} = 1, to address in LR. <i>Extended mnemonic for</i> bclr 10,cr_bit		9-32
bdztlrl		<i>Extended mnemonic for</i> bclrl 10,cr_bit	(LR) ← CIA + 4.	
beq	[cr_field], target	Branch if equal. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 12,4*cr_field+2,target		9-20
beqa		<i>Extended mnemonic for</i> bca 12,4*cr_field+2,target		
beql		<i>Extended mnemonic for</i> bcl 12,4*cr_field+2,target	(LR) ← CIA + 4.	
beqla		<i>Extended mnemonic for</i> bcla 12,4*cr_field+2,target	(LR) ← CIA + 4.	

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
beqctr	[cr_field]	Branch if equal, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bcctr 12,4*cr_field+2		9-28
beqctrl		<i>Extended mnemonic for</i> bcctrl 12,4*cr_field+2	(LR) ← CIA + 4.	
beqlr	[cr_field]	Branch if equal, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 12,4*cr_field+2		9-32
beqlrl		<i>Extended mnemonic for</i> bclrl 12,4*cr_field+2	(LR) ← CIA + 4.	
bf	cr_bit, target	Branch if CR _{cr_bit} = 0. <i>Extended mnemonic for</i> bc 4,cr_bit,target		9-20
bfa		<i>Extended mnemonic for</i> bca 4,cr_bit,target		
bfl		<i>Extended mnemonic for</i> bcl 4,cr_bit,target	(LR) ← CIA + 4.	
bfla		<i>Extended mnemonic for</i> bcla 4,cr_bit,target	(LR) ← CIA + 4.	
bfctr	cr_bit	Branch if CR _{cr_bit} = 0, to address in CTR. <i>Extended mnemonic for</i> bcctr 4,cr_bit		9-28
bfctrl		<i>Extended mnemonic for</i> bcctrl 4,cr_bit	(LR) ← CIA + 4.	
bfldr	cr_bit	Branch if CR _{cr_bit} = 0, to address in LR. <i>Extended mnemonic for</i> bclr 4,cr_bit		9-32
bfldr		<i>Extended mnemonic for</i> bclrl 4,cr_bit	(LR) ← CIA + 4.	

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
bge	[cr_field], target	Branch if greater than or equal. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 4,4*cr_field+0,target		9-20
bgea		<i>Extended mnemonic for</i> bca 4,4*cr_field+0,target		
bgel		<i>Extended mnemonic for</i> bcl 4,4*cr_field+0,target	(LR) ← CIA + 4.	
bgeLa		<i>Extended mnemonic for</i> bcla 4,4*cr_field+0,target	(LR) ← CIA + 4.	
bgectr	[cr_field]	Branch if greater than or equal, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bcctr 4,4*cr_field+0		9-28
bgectrl		<i>Extended mnemonic for</i> bcctrl 4,4*cr_field+0	(LR) ← CIA + 4.	
bgeLr	[cr_field]	Branch if greater than or equal, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 4,4*cr_field+0		9-32
bgeLrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+0	(LR) ← CIA + 4.	
bgt	[cr_field], target	Branch if greater than. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 12,4*cr_field+1,target		9-20
bgtA		<i>Extended mnemonic for</i> bca 12,4*cr_field+1,target		
bgtl		<i>Extended mnemonic for</i> bcl 12,4*cr_field+1,target	(LR) ← CIA + 4.	
bgtla		<i>Extended mnemonic for</i> bcla 12,4*cr_field+1,target	(LR) ← CIA + 4.	

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
bgtctr	[cr_field]	Branch if greater than, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bcctr 12,4*cr_field+1		9-28
bgtctrl		<i>Extended mnemonic for</i> bcctrl 12,4*cr_field+1	(LR) ← CIA + 4.	
bgtlr	[cr_field]	Branch if greater than, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 12,4*cr_field+1		9-32
bgtlrl		<i>Extended mnemonic for</i> bclrl 12,4*cr_field+1	(LR) ← CIA + 4.	
ble	[cr_field], target	Branch if less than or equal. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 4,4*cr_field+1,target		9-20
blea		<i>Extended mnemonic for</i> bca 4,4*cr_field+1,target		
blel		<i>Extended mnemonic for</i> bcl 4,4*cr_field+1,target	(LR) ← CIA + 4.	
blela		<i>Extended mnemonic for</i> bcla 4,4*cr_field+1,target	(LR) ← CIA + 4.	
blectr	[cr_field]	Branch if less than or equal, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bcctr 4,4*cr_field+1		9-28
blectrl		<i>Extended mnemonic for</i> bcctrl 4,4*cr_field+1	(LR) ← CIA + 4.	
blelr	[cr_field]	Branch if less than or equal, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 4,4*cr_field+1		9-32
blelrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+1	(LR) ← CIA + 4.	

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
blr		Branch unconditionally, to address in LR. <i>Extended mnemonic for</i> bclr 20,0		9-32
blrl		<i>Extended mnemonic for</i> bclrl 20,0	(LR) \leftarrow CIA + 4.	
blt	[cr_field], target	Branch if less than. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 12,4*cr_field+0,target		9-20
blta		<i>Extended mnemonic for</i> bca 12,4*cr_field+0,target		
bltl		<i>Extended mnemonic for</i> bcl 12,4*cr_field+0,target	(LR) \leftarrow CIA + 4.	
bltla		<i>Extended mnemonic for</i> bcla 12,4*cr_field+0,target	(LR) \leftarrow CIA + 4.	
bltctr	[cr_field]	Branch if less than, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bcctr 12,4*cr_field+0		9-28
bltctrl		<i>Extended mnemonic for</i> bcctrl 12,4*cr_field+0	(LR) \leftarrow CIA + 4.	
bltlr	[cr_field]	Branch if less than, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 12,4*cr_field+0		9-32
bltlrl		<i>Extended mnemonic for</i> bclrl 12,4*cr_field+0	(LR) \leftarrow CIA + 4.	

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
bne	[cr_field], target	Branch if not equal. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 4,4*cr_field+2,target		9-20
bnea		<i>Extended mnemonic for</i> bca 4,4*cr_field+2,target		
bnel		<i>Extended mnemonic for</i> bcl 4,4*cr_field+2,target	(LR) ← CIA + 4.	
bnela		<i>Extended mnemonic for</i> bcla 4,4*cr_field+2,target	(LR) ← CIA + 4.	
bnectr	[cr_field]	Branch if not equal, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bcctr 4,4*cr_field+2		9-28
bnectl		<i>Extended mnemonic for</i> bcctrl 4,4*cr_field+2	(LR) ← CIA + 4.	
bnelr	[cr_field]	Branch if not equal, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 4,4*cr_field+2		9-32
bnelrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+2	(LR) ← CIA + 4.	
bng	[cr_field], target	Branch if not greater than. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 4,4*cr_field+1,target		9-20
bnga		<i>Extended mnemonic for</i> bca 4,4*cr_field+1,target		
bngl		<i>Extended mnemonic for</i> bcl 4,4*cr_field+1,target	(LR) ← CIA + 4.	
bngla		<i>Extended mnemonic for</i> bcla 4,4*cr_field+1,target	(LR) ← CIA + 4.	

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
bngctr	[cr_field]	Branch if not greater than, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bcctr 4,4*cr_field+1		9-28
bngctrl		<i>Extended mnemonic for</i> bcctrl 4,4*cr_field+1	(LR) ← CIA + 4.	
bnglr	[cr_field]	Branch if not greater than, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 4,4*cr_field+1		9-32
bnglrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+1	(LR) ← CIA + 4.	
bnl	[cr_field], target	Branch if not less than. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 4,4*cr_field+0,target		9-20
bnla		<i>Extended mnemonic for</i> bca 4,4*cr_field+0,target		
bnll		<i>Extended mnemonic for</i> bcl 4,4*cr_field+0,target	(LR) ← CIA + 4.	
bnlla		<i>Extended mnemonic for</i> bcla 4,4*cr_field+0,target	(LR) ← CIA + 4.	
bnlctr	[cr_field]	Branch if not less than, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bcctr 4,4*cr_field+0		9-28
bnlctrl		<i>Extended mnemonic for</i> bcctrl 4,4*cr_field+0	(LR) ← CIA + 4.	
bnllr	[cr_field]	Branch if not less than, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 4,4*cr_field+0		9-32
bnllrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+0	(LR) ← CIA + 4.	

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
bns	[cr_field], target	Branch if not summary overflow. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 4,4*cr_field+3,target		9-20
bnsa		<i>Extended mnemonic for</i> bca 4,4*cr_field+3,target		
bnsi		<i>Extended mnemonic for</i> bcl 4,4*cr_field+3,target	(LR) ← CIA + 4.	
bnsia		<i>Extended mnemonic for</i> bcla 4,4*cr_field+3,target	(LR) ← CIA + 4.	
bnsctr	[cr_field]	Branch if not summary overflow, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bcctr 4,4*cr_field+3		9-28
bnsctrl		<i>Extended mnemonic for</i> bcctrl 4,4*cr_field+3	(LR) ← CIA + 4.	
bnslr	[cr_field]	Branch if not summary overflow, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 4,4*cr_field+3		9-32
bnslrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+3	(LR) ← CIA + 4.	
bnu	[cr_field], target	Branch if not unordered. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 4,4*cr_field+3,target		9-20
bnua		<i>Extended mnemonic for</i> bca 4,4*cr_field+3,target		
bnul		<i>Extended mnemonic for</i> bcl 4,4*cr_field+3,target	(LR) ← CIA + 4.	
bnula		<i>Extended mnemonic for</i> bcla 4,4*cr_field+3,target	(LR) ← CIA + 4.	

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
bnuctr	[cr_field]	Branch if not unordered, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for bcctr 4,4*cr_field+3</i>		9-28
bnuctrl		<i>Extended mnemonic for bcctrl 4,4*cr_field+3</i>	(LR) ← CIA + 4.	
bnulr	[cr_field]	Branch if not unordered, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for bclr 4,4*cr_field+3</i>		9-32
bnulrl		<i>Extended mnemonic for bclrl 4,4*cr_field+3</i>	(LR) ← CIA + 4.	
bso	[cr_field], target	Branch if summary overflow. Use CR0 if cr_field is omitted. <i>Extended mnemonic for bc 12,4*cr_field+3,target</i>		9-20
bsoa		<i>Extended mnemonic for bca 12,4*cr_field+3,target</i>		
bsol		<i>Extended mnemonic for bcl 12,4*cr_field+3,target</i>	(LR) ← CIA + 4.	
bsola		<i>Extended mnemonic for bcla 12,4*cr_field+3,target</i>	(LR) ← CIA + 4.	
bsoctr	[cr_field]	Branch if summary overflow, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for bcctr 12,4*cr_field+3</i>		9-28
bsoctrl		<i>Extended mnemonic for bcctrl 12,4*cr_field+3</i>	(LR) ← CIA + 4.	
bsolr	[cr_field]	Branch if summary overflow, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for bclr 12,4*cr_field+3</i>		9-32
bsolrl		<i>Extended mnemonic for bclrl 12,4*cr_field+3</i>	(LR) ← CIA + 4.	

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
bt	cr_bit, target	Branch if CR _{cr_bit} = 1. <i>Extended mnemonic for</i> bc 12,cr_bit,target		9-20
bta		<i>Extended mnemonic for</i> bca 12,cr_bit,target		
btl		<i>Extended mnemonic for</i> bcl 12,cr_bit,target	(LR) ← CIA + 4.	
btla		<i>Extended mnemonic for</i> bcla 12,cr_bit,target	(LR) ← CIA + 4.	
btctr	cr_bit	Branch if CR _{cr_bit} = 1, to address in CTR. <i>Extended mnemonic for</i> bcctr 12,cr_bit		9-28
btctrl		<i>Extended mnemonic for</i> bcctrl 12,cr_bit	(LR) ← CIA + 4.	
btlr	cr_bit	Branch if CR _{cr_bit} = 1, to address in LR. <i>Extended mnemonic for</i> bclr 12,cr_bit		9-32
btlrl		<i>Extended mnemonic for</i> bclrl 12,cr_bit	(LR) ← CIA + 4.	
bun	[cr_field], target	Branch if unordered. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 12,4*cr_field+3,target		9-20
buna		<i>Extended mnemonic for</i> bca 12,4*cr_field+3,target		
bunl		<i>Extended mnemonic for</i> bcl 12,4*cr_field+3,target	(LR) ← CIA + 4.	
bunla		<i>Extended mnemonic for</i> bcla 12,4*cr_field+3,target	(LR) ← CIA + 4.	
bunctr	[cr_field]	Branch if unordered, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bcctr 12,4*cr_field+3		9-28
bunctrl		<i>Extended mnemonic for</i> bcctrl 12,4*cr_field+3	(LR) ← CIA + 4.	

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
bunlr	[cr_field]	Branch if unordered, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 12,4*cr_field+3		9-32
bunlrl		<i>Extended mnemonic for</i> bclrl 12,4*cr_field+3	(LR) \leftarrow CIA + 4.	
clrlwi	RA, RS, n	Clear left immediate. ($n < 32$) $(RA)_{0:n-1} \leftarrow {}^n0$ <i>Extended mnemonic for</i> rlwinm RA,RS,0,n,31		9-130
clrlwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,0,n,31	CR[CR0]	
clrlslwi	RA, RS, b, n	Clear left and shift left immediate. ($n \leq b < 32$) $(RA)_{b-n:31-n} \leftarrow (RS)_{b:31}$ $(RA)_{32-n:31} \leftarrow {}^n0$ $(RA)_{0:b-n-1} \leftarrow {}^{b-n}0$ <i>Extended mnemonic for</i> rlwinm RA,RS,n,b-n,31-n		9-130
clrlslwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,n,b-n,31-n	CR[CR0]	
clrrwi	RA, RS, n	Clear right immediate. ($n < 32$) $(RA)_{32-n:31} \leftarrow {}^n0$ <i>Extended mnemonic for</i> rlwinm RA,RS,0,0,31-n		9-130
clrrwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,0,0,31-n	CR[CR0]	
cmp	BF, 0, RA, RB	Compare (RA) to (RB), signed. Results in CR[CRn], where $n = BF$.		9-37
cmpi	BF, 0, RA, IM	Compare (RA) to EXTS(IM), signed. Results in CR[CRn], where $n = BF$.		9-38
cmpl	BF, 0, RA, RB	Compare (RA) to (RB), unsigned. Results in CR[CRn], where $n = BF$.		9-39
cmpli	BF, 0, RA, IM	Compare (RA) to (${}^{16}0 \parallel IM$), unsigned. Results in CR[CRn], where $n = BF$.		9-40

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
cmplw	[BF,] RA, RB	Compare Logical Word. Use CR0 if BF is omitted. <i>Extended mnemonic for</i> cmpl BF,0,RA,RB		9-39
cmplwi	[BF,] RA, IM	Compare Logical Word Immediate. Use CR0 if BF is omitted. <i>Extended mnemonic for</i> cmpli BF,0,RA,IM		9-40
cmpw	[BF,] RA, RB	Compare Word. Use CR0 if BF is omitted. <i>Extended mnemonic for</i> cmp BF,0,RA,RB		9-37
cmpwi	[BF,] RA, IM	Compare Word Immediate. Use CR0 if BF is omitted. <i>Extended mnemonic for</i> cmpi BF,0,RA,IM		9-38
cntlzw	RA, RS	Count leading zeros in RS. Place result in RA.		9-41
cntlzw.			CR[CR0]	
crand	BT, BA, BB	AND bit (CR _{BA}) with (CR _{BB}). Place result in CR _{BT} .		9-42
crandc	BT, BA, BB	AND bit (CR _{BA}) with \neg (CR _{BB}). Place result in CR _{BT} .		9-43
crclr	bx	Condition register clear. <i>Extended mnemonic for</i> crxor bx,bx,bx		9-49
creqv	BT, BA, BB	Equivalence of bit CR _{BA} with CR _{BB} . $CR_{BT} \leftarrow \neg(CR_{BA} \oplus CR_{BB})$		9-44
crmove	bx, by	Condition register move. <i>Extended mnemonic for</i> cror bx,by,by		9-47
crnand	BT, BA, BB	NAND bit (CR _{BA}) with (CR _{BB}). Place result in CR _{BT} .		9-45
crnor	BT, BA, BB	NOR bit (CR _{BA}) with (CR _{BB}). Place result in CR _{BT} .		9-46
crnot	bx, by	Condition register not. <i>Extended mnemonic for</i> crnor bx,by,by		9-46

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
cror	BT, BA, BB	OR bit (CR_{BA}) with (CR_{BB}). Place result in CR_{BT} .		9-47
crorc	BT, BA, BB	OR bit (CR_{BA}) with $\neg(CR_{BB})$. Place result in CR_{BT} .		9-48
crset	bx	Condition register set. <i>Extended mnemonic for</i> creqv bx,bx,bx		9-44
crxor	BT, BA, BB	XOR bit (CR_{BA}) with (CR_{BB}). Place result in CR_{BT} .		9-49
dcba	RA, RB	Speculatively establish the data cache block which contains the effective address $(RA 0) + (RB)$.		9-50
dcbf	RA, RB	Flush (store, then invalidate) the data cache block which contains the effective address $(RA 0) + (RB)$.		9-50
dcbi	RA, RB	Invalidate the data cache block which contains the effective address $(RA 0) + (RB)$.		9-51
dcbst	RA, RB	Store the data cache block which contains the effective address $(RA 0) + (RB)$.		9-52
dcbt	RA, RB	Load the data cache block which contains the effective address $(RA 0) + (RB)$.		9-53
dcbtst	RA, RB	Load the data cache block which contains the effective address $(RA 0) + (RB)$.		9-55
dcbz	RA, RB	Zero the data cache block which contains the effective address $(RA 0) + (RB)$.		9-57
dccci	RA, RB	Invalidate the data cache congruence class associated with the effective address $(RA 0) + (RB)$.		9-59
dcread	RT, RA, RB	Read either tag or data information from the data cache congruence class associated with the effective address $(RA 0) + (RB)$. Place the results in RT.		9-61

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
divw	RT, RA, RB	Divide (RA) by (RB), signed. Place result in RT.		9-63
divw.			CR[CR0]	
divwo			XER[SO, OV]	
divwo.			CR[CR0] XER[SO, OV]	
divwu	RT, RA, RB	Divide (RA) by (RB), unsigned. Place result in RT.		9-64
divwu.			CR[CR0]	
divwuo			XER[SO, OV]	
divwuo.			CR[CR0] XER[SO, OV]	
eieio		Storage synchronization. All loads and stores that precede the eieio instruction complete before any loads and stores that follow the instruction access main storage. Implemented as sync , which is more restrictive.		9-65
eqv	RA, RS, RB	Equivalence of (RS) with (RB). $(RA) \leftarrow \neg((RS) \oplus (RB))$		9-66
eqv.			CR[CR0]	
extlwi	RA, RS, n, b	Extract and left justify immediate. ($n > 0$) $(RA)_{0:n-1} \leftarrow (RS)_{b:b+n-1}$ $(RA)_{n:31} \leftarrow 32-n_0$ <i>Extended mnemonic for</i> rlwinm RA,RS,b,0,n-1		9-130
extlwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,b,0,n-1	CR[CR0]	
extrwi	RA, RS, n, b	Extract and right justify immediate. ($n > 0$) $(RA)_{32-n:31} \leftarrow (RS)_{b:b+n-1}$ $(RA)_{0:31-n} \leftarrow 32-n_0$ <i>Extended mnemonic for</i> rlwinm RA,RS,b+n,32-n,31		9-130
extrwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,b+n,32-n,31	CR[CR0]	
extsb	RA, RS	Extend the sign of byte (RS) _{24:31} . Place the result in RA.		9-67
extsb.			CR[CR0]	

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
extsh	RA, RS	Extend the sign of halfword (RS) _{16:31} . Place the result in RA.		9-68
extsh.			CR[CR0]	
icbi	RA, RB	Invalidate the instruction cache block which contains the effective address (RA 0) + (RB).		9-69
icbt	RA, RB	Load the instruction cache block which contains the effective address (RA 0) + (RB).		9-71
iccci	RA, RB	Invalidate instruction cache congruence class associated with the effective address (RA 0) + (RB).		9-73
icread	RA, RB	Read either tag or data information from the instruction cache congruence class associated with the effective address (RA 0) + (RB). Place the results in ICDBDR.		9-74
inslwi	RA, RS, n, b	Insert from left immediate. (n > 0) $(RA)_{b:b+n-1} \leftarrow (RS)_{0:n-1}$ <i>Extended mnemonic for</i> rlwimi RA,RS,32-b,b,b+n-1		9-129
inslwi.		<i>Extended mnemonic for</i> rlwimi. RA,RS,32-b,b,b+n-1	CR[CR0]	
insrwi	RA, RS, n, b	Insert from right immediate. (n > 0) $(RA)_{b:b+n-1} \leftarrow (RS)_{32-n:31}$ <i>Extended mnemonic for</i> rlwimi RA,RS,32-b-n,b,b+n-1		9-129
insrwi.		<i>Extended mnemonic for</i> rlwimi. RA,RS,32-b-n,b,b+n-1	CR[CR0]	
isync		Synchronize execution context by flushing the prefetch queue.		9-76
la	RT, D(RA)	Load address. (RA ≠ 0) D is an offset from a base address that is assumed to be (RA). $(RT) \leftarrow (RA) + \text{EXTS}(D)$ <i>Extended mnemonic for</i> addi RT,RA,D		9-9

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
lbz	RT, D(RA)	Load byte from EA = (RA 0) + EXTS(D) and pad left with zeroes, (RT) \leftarrow $^{24}0$ MS(EA,1).		9-78
lbzu	RT, D(RA)	Load byte from EA = (RA 0) + EXTS(D) and pad left with zeroes, (RT) \leftarrow $^{24}0$ MS(EA,1). Update the base address, (RA) \leftarrow EA.		9-79
lbzux	RT, RA, RB	Load byte from EA = (RA 0) + (RB) and pad left with zeroes, (RT) \leftarrow $^{24}0$ MS(EA,1). Update the base address, (RA) \leftarrow EA.		9-80
lbzx	RT, RA, RB	Load byte from EA = (RA 0) + (RB) and pad left with zeroes, (RT) \leftarrow $^{24}0$ MS(EA,1).		9-81
lha	RT, D(RA)	Load halfword from EA = (RA 0) + EXTS(D) and sign extend, (RT) \leftarrow EXTS(MS(EA,2)).		9-82
lhau	RT, D(RA)	Load halfword from EA = (RA 0) + EXTS(D) and sign extend, (RT) \leftarrow EXTS(MS(EA,2)). Update the base address, (RA) \leftarrow EA.		9-83
lhaux	RT, RA, RB	Load halfword from EA = (RA 0) + (RB) and sign extend, (RT) \leftarrow EXTS(MS(EA,2)). Update the base address, (RA) \leftarrow EA.		9-84
lhax	RT, RA, RB	Load halfword from EA = (RA 0) + (RB) and sign extend, (RT) \leftarrow EXTS(MS(EA,2)).		9-85
lhbrx	RT, RA, RB	Load halfword from EA = (RA 0) + (RB) then reverse byte order and pad left with zeroes, (RT) \leftarrow $^{16}0$ MS(EA+1,1) MS(EA,1).		9-86
lhz	RT, D(RA)	Load halfword from EA = (RA 0) + EXTS(D) and pad left with zeroes, (RT) \leftarrow $^{16}0$ MS(EA,2).		9-87

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
lhz	RT, D(RA)	Load halfword from EA = (RA 0) + EXTS(D) and pad left with zeroes, (RT) \leftarrow $^{16}0 \parallel$ MS(EA,2). Update the base address, (RA) \leftarrow EA.		9-88
lhzx	RT, RA, RB	Load halfword from EA = (RA 0) + (RB) and pad left with zeroes, (RT) \leftarrow $^{16}0 \parallel$ MS(EA,2). Update the base address, (RA) \leftarrow EA.		9-89
lhzx	RT, RA, RB	Load halfword from EA = (RA 0) + (RB) and pad left with zeroes, (RT) \leftarrow $^{16}0 \parallel$ MS(EA,2).		9-90
li	RT, IM	Load immediate. (RT) \leftarrow EXTS(IM) <i>Extended mnemonic for</i> addi RT,0,value		9-9
lis	RT, IM	Load immediate shifted. (RT) \leftarrow (IM \parallel $^{16}0$) <i>Extended mnemonic for</i> addis RT,0,value		9-12
lmw	RT, D(RA)	Load multiple words starting from EA = (RA 0) + EXTS(D). Place into consecutive registers, RT through GPR(31). RA is not altered unless RA = GPR(31).		9-91
lswi	RT, RA, NB	Load consecutive bytes from EA=(RA 0). Number of bytes n=32 if NB=0, else n=NB. Stack bytes into words in CEIL(n/4) consecutive registers starting with RT, to R _{FINAL} \leftarrow ((RT + CEIL(n/4) - 1) % 32). GPR(0) is consecutive to GPR(31). RA is not altered unless RA = R _{FINAL} .		9-92

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
lswx	RT, RA, RB	Load consecutive bytes from $EA = (RA 0) + (RB)$. Number of bytes $n = XER[TBC]$. Stack bytes into words in $CEIL(n/4)$ consecutive registers starting with RT, to $R_{FINAL} \leftarrow ((RT + CEIL(n/4) - 1) \% 32)$. GPR(0) is consecutive to GPR(31). RA is not altered unless $RA = R_{FINAL}$. RB is not altered unless $RB = R_{FINAL}$. If $n=0$, content of RT is undefined.		9-94
lwarx	RT, RA, RB	Load word from $EA = (RA 0) + (RB)$ and place in RT, $(RT) \leftarrow MS(EA, 4)$. Set the Reservation bit.		9-96
lwbx	RT, RA, RB	Load word from $EA = (RA 0) + (RB)$ then reverse byte order, $(RT) \leftarrow MS(EA+3, 1) \parallel MS(EA+2, 1) \parallel MS(EA+1, 1) \parallel MS(EA, 1)$.		9-98
lwz	RT, D(RA)	Load word from $EA = (RA 0) + EXTS(D)$ and place in RT, $(RT) \leftarrow MS(EA, 4)$.		9-99
lwzu	RT, D(RA)	Load word from $EA = (RA 0) + EXTS(D)$ and place in RT, $(RT) \leftarrow MS(EA, 4)$. Update the base address, $(RA) \leftarrow EA$.		9-100
lwzux	RT, RA, RB	Load word from $EA = (RA 0) + (RB)$ and place in RT, $(RT) \leftarrow MS(EA, 4)$. Update the base address, $(RA) \leftarrow EA$.		9-101
lwzx	RT, RA, RB	Load word from $EA = (RA 0) + (RB)$ and place in RT, $(RT) \leftarrow MS(EA, 4)$.		9-102
mcrf	BF, BFA	Move CR field, $(CR[CRn]) \leftarrow (CR[CRm])$ where $m \leftarrow BFA$ and $n \leftarrow BF$.		9-103
mcrxr	BF	Move XER[0:3] into field CRn, where $n \leftarrow BF$. $CR[CRn] \leftarrow (XER[SO, OV, CA])$. $(XER[SO, OV, CA]) \leftarrow {}^3_0$.		9-104

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
mfcrr	RT	Move from CR to RT, (RT) \leftarrow (CR).		9-105
mfdcr	RT, DCRN	Move from DCR to RT, (RT) \leftarrow (DCR(DCRN)).		9-106
mfmsr	RT	Move from MSR to RT, (RT) \leftarrow (MSR).		9-107
mfcdbcr mfctr mfdac1 mfdbsr mfdccr mfdcwr mfdear mfesr mfevpr mfiacl mficcr mficdbdr mflr mfpit mfpvr mfsgrr mfsprr0 mfsprr1 mfsprr2 mfsprr3 mfssrr0 mfssrr1 mfssrr2 mfssrr3 mftbhi mftbhu mftblo mftblu mftcr mftsr mfxer	RT	Move from special purpose register (SPR) SPRN. <i>Extended mnemonic for</i> mfspr RT,SPRN See Table 10-2 on p. 10-3 for listing of valid SPRN values.		9-108
mfspr	RT, SPRN	Move from SPR to RT, (RT) \leftarrow (SPR(SPRN)).		9-108

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
mr	RT, RS	Move register. $(RT) \leftarrow (RS)$ <i>Extended mnemonic for or RT,RS,RS</i>		9-123
mr.		<i>Extended mnemonic for or. RT,RS,RS</i>	CR[CR0]	
mtcr	RS	Move to Condition Register. <i>Extended mnemonic for mtcrf 0xFF,RS</i>		9-110
mtcrf	FXM, RS	Move some or all of the contents of RS into CR as specified by FXM field, $\text{mask} \leftarrow {}^4(\text{FXM}_0) \parallel {}^4(\text{FXM}_1) \parallel \dots \parallel$ ${}^4(\text{FXM}_6) \parallel {}^4(\text{FXM}_7)$. $(CR) \leftarrow ((RS) \wedge \text{mask}) \vee (CR) \wedge \neg \text{mask}$.		9-110
mtdcr	DCRN, RS	Move to DCR from RS, $(DCR(DCRN)) \leftarrow (RS)$.		9-112
mtmsr	RS	Move to MSR from RS, $(MSR) \leftarrow (RS)$.		9-113

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
mtcdbc mtctr mtdac1 mtdbc mtdbsr mtdccr mtdcwr mtesr mtevp mtiac1 mticcr mticdbdr mtlr mtpit mtpvr mtsg mtsprg0 mtsprg1 mtsprg2 mtsprg3 mtsrr0 mtsrr1 mtsrr2 mtsrr3 mtbhi mtblo mttcr mttsr mtxer	RS	Move to SPR SPRN. <i>Extended mnemonic for</i> mtspr SPRN,RS See Table 10-2 on p. 10-3 for listing of valid SPRN values.		9-114
mtspr	SPRN, RS	Move to SPR from RS, (SPR(SPRN)) \leftarrow (RS).		9-114
mulhw	RT, RA, RB	Multiply (RA) and (RB), signed. Place high-order result in RT. $\text{prod}_{0:63} \leftarrow (\text{RA}) \times (\text{RB})$ (signed). (RT) $\leftarrow \text{prod}_{0:31}$.	CR[CR0]	9-116
mulhw.				
mulhwu	RT, RA, RB	Multiply (RA) and (RB), unsigned. Place high-order result in RT. $\text{prod}_{0:63} \leftarrow (\text{RA}) \times (\text{RB})$ (unsigned). (RT) $\leftarrow \text{prod}_{0:31}$.	CR[CR0]	9-117
mulhwu.				
mulli	RT, RA, IM	Multiply (RA) and IM, signed. Place low-order result in RT. $\text{prod}_{0:47} \leftarrow (\text{RA}) \times \text{IM}$ (signed) (RT) $\leftarrow \text{prod}_{16:47}$		9-118

A

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
mullw	RT, RA, RB	Multiply (RA) and (RB), signed. Place low-order result in RT. $\text{prod}_{0:63} \leftarrow (\text{RA}) \times (\text{RB})$ (signed). $(\text{RT}) \leftarrow \text{prod}_{32:63}$.		9-119
mullw.			CR[CR0]	
mullwo			XER[SO, OV]	
mullwo.			CR[CR0] XER[SO, OV]	
nand	RA, RS, RB	NAND (RS) with (RB). Place result in RA.		9-120
nand.			CR[CR0]	
neg	RT, RA	Negative (two's complement) of RA. $(\text{RT}) \leftarrow \neg(\text{RA}) + 1$		9-121
neg.			CR[CR0]	
nego			XER[SO, OV]	
nego.			CR[CR0] XER[SO, OV]	
nop		Preferred no-op, triggers optimizations based on no-ops. <i>Extended mnemonic for</i> ori 0,0,0		9-125
nor	RA, RS, RB	NOR (RS) with (RB). Place result in RA.		9-122
nor.			CR[CR0]	
not	RA, RS	Complement register. $(\text{RA}) \leftarrow \neg(\text{RS})$ <i>Extended mnemonic for</i> nor RA,RS,RS		9-122
not.		<i>Extended mnemonic for</i> nor. RA,RS,RS	CR[CR0]	
or	RA, RS, RB	OR (RS) with (RB). Place result in RA.		9-123
or.			CR[CR0]	
orc	RA, RS, RB	OR (RS) with $\neg(\text{RB})$. Place result in RA.		9-124
orc.			CR[CR0]	
ori	RA, RS, IM	OR (RS) with ($^{16}0 \parallel \text{IM}$). Place result in RA.		9-125
oris	RA, RS, IM	OR (RS) with ($\text{IM} \parallel ^{16}0$). Place result in RA.		9-126

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
rfti		Return from critical interrupt (PC) \leftarrow (SRR2). (MSR) \leftarrow (SRR3).		9-127
rfti		Return from interrupt. (PC) \leftarrow (SRR0). (MSR) \leftarrow (SRR1).		9-128
rlwimi	RA, RS, SH, MB, ME	Rotate left word immediate, then insert according to mask. $r \leftarrow \text{ROTL}((RS), SH)$ $m \leftarrow \text{MASK}(MB, ME)$ $(RA) \leftarrow (r \wedge m) \vee ((RA) \wedge \neg m)$		9-129
rlwimi.			CR[CR0]	
rlwinm	RA, RS, SH, MB, ME	Rotate left word immediate, then AND with mask. $r \leftarrow \text{ROTL}((RS), SH)$ $m \leftarrow \text{MASK}(MB, ME)$ $(RA) \leftarrow (r \wedge m)$		9-130
rlwinm.			CR[CR0]	
rlwnm	RA, RS, RB, MB, ME	Rotate left word, then AND with mask. $r \leftarrow \text{ROTL}((RS), (RB)_{27:31})$ $m \leftarrow \text{MASK}(MB, ME)$ $(RA) \leftarrow (r \wedge m)$		9-133
rlwnm.			CR[CR0]	
rotlw	RA, RS, RB	Rotate left. $(RA) \leftarrow \text{ROTL}((RS), (RB)_{27:31})$ <i>Extended mnemonic for</i> rlwnm RA,RS,RB,0,31		9-133
rotlw.		<i>Extended mnemonic for</i> rlwnm. RA,RS,RB,0,31	CR[CR0]	
rotlwi	RA, RS, n	Rotate left immediate. $(RA) \leftarrow \text{ROTL}((RS), n)$ <i>Extended mnemonic for</i> rlwinm RA,RS,n,0,31		9-130
rotlwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,n,0,31	CR[CR0]	
rotrwi	RA, RS, n	Rotate right immediate. $(RA) \leftarrow \text{ROTL}((RS), 32-n)$ <i>Extended mnemonic for</i> rlwinm RA,RS,32-n,0,31		9-130
rotrwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,32-n,0,31	CR[CR0]	

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
sc		System call exception is generated. $(SRR1) \leftarrow (MSR)$ $(SRR0) \leftarrow (PC)$ $PC \leftarrow EVPR_{0:15} \parallel x'0C00'$ $(MSR[WE, PR, EE, PE, DR, IR]) \leftarrow 0$ $(MSR[LE]) \leftarrow (MSR[ILE])$		9-134
slw	RA, RS, RB	Shift left (RS) by $(RB)_{27:31}$. $n \leftarrow (RB)_{27:31}$. $r \leftarrow ROTL((RS), n)$. if $(RB)_{26} = 0$ then $m \leftarrow MASK(0, 31 - n)$ else $m \leftarrow {}^{32}0$. $(RA) \leftarrow r \wedge m$.		9-135
slw.			CR[CR0]	
slwi	RA, RS, n	Shift left immediate. ($n < 32$) $(RA)_{0:31-n} \leftarrow (RS)_{n:31}$ $(RA)_{32-n:31} \leftarrow {}^n0$ <i>Extended mnemonic for</i> rlwinm RA,RS,n,0,31-n		9-130
slwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,n,0,31-n	CR[CR0]	
sraw	RA, RS, RB	Shift right algebraic (RS) by $(RB)_{27:31}$. $n \leftarrow (RB)_{27:31}$. $r \leftarrow ROTL((RS), 32 - n)$. if $(RB)_{26} = 0$ then $m \leftarrow MASK(n, 31)$ else $m \leftarrow {}^{32}0$. $s \leftarrow (RS)_0$. $(RA) \leftarrow (r \wedge m) \vee ({}^{32}s \wedge \neg m)$. $XER[CA] \leftarrow s \wedge ((r \wedge \neg m) \neq 0)$.		9-136
sraw.			CR[CR0]	
srawi	RA, RS, SH	Shift right algebraic (RS) by SH. $n \leftarrow SH$. $r \leftarrow ROTL((RS), 32 - n)$. $m \leftarrow MASK(n, 31)$. $s \leftarrow (RS)_0$. $(RA) \leftarrow (r \wedge m) \vee ({}^{32}s \wedge \neg m)$. $XER[CA] \leftarrow s \wedge ((r \wedge \neg m) \neq 0)$.		9-137
srawi.			CR[CR0]	
srw	RA, RS, RB	Shift right (RS) by $(RB)_{27:31}$. $n \leftarrow (RB)_{27:31}$. $r \leftarrow ROTL((RS), 32 - n)$. if $(RB)_{26} = 0$ then $m \leftarrow MASK(n, 31)$ else $m \leftarrow {}^{32}0$. $(RA) \leftarrow r \wedge m$.		9-138
srw.			CR[CR0]	

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
srwi	RA, RS, n	Shift right immediate. ($n < 32$) $(RA)_{n:31} \leftarrow (RS)_{0:31-n}$ $(RA)_{0:n-1} \leftarrow ^n0$ <i>Extended mnemonic for</i> rlwinm RA,RS,32-n,n,31		9-130
srwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,32-n,n,31	CR[CR0]	
stb	RS, D(RA)	Store byte $(RS)_{24:31}$ in memory at $EA = (RA 0) + EXTS(D)$.		9-139
stbu	RS, D(RA)	Store byte $(RS)_{24:31}$ in memory at $EA = (RA 0) + EXTS(D)$. Update the base address, $(RA) \leftarrow EA$.		9-140
stbux	RS, RA, RB	Store byte $(RS)_{24:31}$ in memory at $EA = (RA 0) + (RB)$. Update the base address, $(RA) \leftarrow EA$.		9-141
stbx	RS, RA, RB	Store byte $(RS)_{24:31}$ in memory at $EA = (RA 0) + (RB)$.		9-142
sth	RS, D(RA)	Store halfword $(RS)_{16:31}$ in memory at $EA = (RA 0) + EXTS(D)$.		9-143
sthbrx	RS, RA, RB	Store halfword $(RS)_{16:31}$ byte-reversed in memory at $EA = (RA 0) + (RB)$. $MS(EA, 2) \leftarrow (RS)_{24:31} \parallel (RS)_{16:23}$		9-144
sthu	RS, D(RA)	Store halfword $(RS)_{16:31}$ in memory at $EA = (RA 0) + EXTS(D)$. Update the base address, $(RA) \leftarrow EA$.		9-145
sthux	RS, RA, RB	Store halfword $(RS)_{16:31}$ in memory at $EA = (RA 0) + (RB)$. Update the base address, $(RA) \leftarrow EA$.		9-146
sthx	RS, RA, RB	Store halfword $(RS)_{16:31}$ in memory at $EA = (RA 0) + (RB)$.		9-147
stmw	RS, D(RA)	Store consecutive words from RS through GPR(31) in memory starting at $EA = (RA 0) + EXTS(D)$.		9-148

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
stswi	RS, RA, NB	Store consecutive bytes in memory starting at EA=(RA 0). Number of bytes n=32 if NB=0, else n=NB. Bytes are unstacked from CEIL(n/4) consecutive registers starting with RS. GPR(0) is consecutive to GPR(31).		9-149
stswx	RS, RA, RB	Store consecutive bytes in memory starting at EA=(RA 0)+(RB). Number of bytes n=XER[TBC]. Bytes are unstacked from CEIL(n/4) consecutive registers starting with RS. GPR(0) is consecutive to GPR(31).		9-150
stw	RS, D(RA)	Store word (RS) in memory at EA = (RA 0) + EXTS(D).		9-152
stwbrx	RS, RA, RB	Store word (RS) byte-reversed in memory at EA = (RA 0) + (RB). $MS(EA, 4) \leftarrow (RS)_{24:31} \parallel (RS)_{16:23} \parallel (RS)_{8:15} \parallel (RS)_{0:7}$		9-153
stwcx.	RS, RA, RB	Store word (RS) in memory at EA = (RA 0) + (RB) only if reservation bit is set. if RESERVE = 1 then $MS(EA, 4) \leftarrow (RS)$ RESERVE \leftarrow 0 $(CR[CR0]) \leftarrow {}^20 \parallel 1 \parallel XER_{s0}$ else $(CR[CR0]) \leftarrow {}^20 \parallel 0 \parallel XER_{s0}$.		9-154
stwu	RS, D(RA)	Store word (RS) in memory at EA = (RA 0) + EXTS(D). Update the base address, (RA) \leftarrow EA.		9-156
stwux	RS, RA, RB	Store word (RS) in memory at EA = (RA 0) + (RB). Update the base address, (RA) \leftarrow EA.		9-157
stwx	RS, RA, RB	Store word (RS) in memory at EA = (RA 0) + (RB).		9-158

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
sub	RT, RA, RB	Subtract (RB) from (RA). $(RT) \leftarrow \neg(RB) + (RA) + 1.$ <i>Extended mnemonic for subf RT,RB,RA</i>		9-159
sub.		<i>Extended mnemonic for subf. RT,RB,RA</i>	CR[CR0]	
subo		<i>Extended mnemonic for subfo RT,RB,RA</i>	XER[SO, OV]	
subo.		<i>Extended mnemonic for subfo. RT,RB,RA</i>	CR[CR0] XER[SO, OV]	
subc	RT, RA, RB	Subtract (RB) from (RA). $(RT) \leftarrow \neg(RB) + (RA) + 1.$ Place carry-out in XER[CA]. <i>Extended mnemonic for subfc RT,RB,RA</i>		9-160
subc.		<i>Extended mnemonic for subfc. RT,RB,RA</i>	CR[CR0]	
subco		<i>Extended mnemonic for subfco RT,RB,RA</i>	XER[SO, OV]	
subco.		<i>Extended mnemonic for subfco. RT,RB,RA</i>	CR[CR0] XER[SO, OV]	
subf	RT, RA, RB	Subtract (RA) from (RB). $(RT) \leftarrow \neg(RA) + (RB) + 1.$		9-159
subf.			CR[CR0]	
subfo			XER[SO, OV]	
subfo.			CR[CR0] XER[SO, OV]	
subfc	RT, RA, RB	Subtract (RA) from (RB). $(RT) \leftarrow \neg(RA) + (RB) + 1.$ Place carry-out in XER[CA].		9-160
subfc.			CR[CR0]	
subfco			XER[SO, OV]	
subfco.			CR[CR0] XER[SO, OV]	

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
subfe	RT, RA, RB	Subtract (RA) from (RB) with carry-in. (RT) $\leftarrow \neg(RA) + (RB) + XER[CA]$. Place carry-out in XER[CA].		9-162
subfe.			CR[CR0]	
subfeo			XER[SO, OV]	
subfeo.			CR[CR0] XER[SO, OV]	
subfic	RT, RA, IM	Subtract (RA) from EXTS(IM). (RT) $\leftarrow \neg(RA) + EXTS(IM) + 1$. Place carry-out in XER[CA].		9-163
subfme	RT, RA, RB	Subtract (RA) from (−1) with carry-in. (RT) $\leftarrow \neg(RA) + (−1) + XER[CA]$. Place carry-out in XER[CA].		9-164
subfme.			CR[CR0]	
subfmeo			XER[SO, OV]	
subfmeo.			CR[CR0] XER[SO, OV]	
subfze	RT, RA, RB	Subtract (RA) from zero with carry-in. (RT) $\leftarrow \neg(RA) + XER[CA]$. Place carry-out in XER[CA].		9-165
subfze.			CR[CR0]	
subfzeo			XER[SO, OV]	
subfzeo.			CR[CR0] XER[SO, OV]	
subi	RT, RA, IM	Subtract EXTS(IM) from (RA 0). Place result in RT. <i>Extended mnemonic for</i> addi RT,RA,−IM		9-9
subic	RT, RA, IM	Subtract EXTS(IM) from (RA). Place result in RT. Place carry-out in XER[CA]. <i>Extended mnemonic for</i> addic RT,RA,−IM		9-10
subic.	RT, RA, IM	Subtract EXTS(IM) from (RA). Place result in RT. Place carry-out in XER[CA]. <i>Extended mnemonic for</i> addic. RT,RA,−IM	CR[CR0]	9-11
subis	RT, RA, IM	Subtract (IM ¹⁶ 0) from (RA 0). Place result in RT. <i>Extended mnemonic for</i> addis RT,RA,−IM		9-12

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
sync		Synchronization. All instructions that precede sync complete before any instructions that follow sync begin. When sync completes, all storage accesses initiated prior to sync will have completed.		9-166

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
trap		Trap unconditionally. <i>Extended mnemonic for tw 31,0,0</i>		
tweq	RA, RB	Trap if (RA) equal to (RB). <i>Extended mnemonic for tw 4,RA,RB</i>		
twge		Trap if (RA) greater than or equal to (RB). <i>Extended mnemonic for tw 12,RA,RB</i>		
twgt		Trap if (RA) greater than (RB). <i>Extended mnemonic for tw 8,RA,RB</i>		
twle		Trap if (RA) less than or equal to (RB). <i>Extended mnemonic for tw 20,RA,RB</i>		
twlge		Trap if (RA) logically greater than or equal to (RB). <i>Extended mnemonic for tw 5,RA,RB</i>		
twlgt		Trap if (RA) logically greater than (RB). <i>Extended mnemonic for tw 1,RA,RB</i>		
twlle		Trap if (RA) logically less than or equal to (RB). <i>Extended mnemonic for tw 6,RA,RB</i>		
twllt		Trap if (RA) logically less than (RB). <i>Extended mnemonic for tw 2,RA,RB</i>		
twlng		Trap if (RA) logically not greater than (RB). <i>Extended mnemonic for tw 6,RA,RB</i>		
twlnl		Trap if (RA) logically not less than (RB). <i>Extended mnemonic for tw 5,RA,RB</i>		
twlt		Trap if (RA) less than (RB). <i>Extended mnemonic for tw 16,RA,RB</i>		
twne		Trap if (RA) not equal to (RB). <i>Extended mnemonic for tw 24,RA,RB</i>		
twng		Trap if (RA) not greater than (RB). <i>Extended mnemonic for tw 20,RA,RB</i>		
twnl		Trap if (RA) not less than (RB). <i>Extended mnemonic for tw 12,RA,RB</i>		
tw	TO, RA, RB	Trap exception is generated if, comparing (RA) with (RB), any condition specified by TO is true.		9-167

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
tweqi	RA, IM	Trap if (RA) equal to EXTS(IM). <i>Extended mnemonic for twi 4,RA,IM</i>		
twgei		Trap if (RA) greater than or equal to EXTS(IM). <i>Extended mnemonic for twi 12,RA,IM</i>		
twgti		Trap if (RA) greater than EXTS(IM). <i>Extended mnemonic for twi 8,RA,IM</i>		
twlei		Trap if (RA) less than or equal to EXTS(IM). <i>Extended mnemonic for twi 20,RA,IM</i>		
twlgei		Trap if (RA) logically greater than or equal to EXTS(IM). <i>Extended mnemonic for twi 5,RA,IM</i>		
twlgti		Trap if (RA) logically greater than EXTS(IM). <i>Extended mnemonic for twi 1,RA,IM</i>		
twllei		Trap if (RA) logically less than or equal to EXTS(IM). <i>Extended mnemonic for twi 6,RA,IM</i>		
twllti		Trap if (RA) logically less than EXTS(IM). <i>Extended mnemonic for twi 2,RA,IM</i>		
twlngi		Trap if (RA) logically not greater than EXTS(IM). <i>Extended mnemonic for twi 6,RA,IM</i>		
twlnli		Trap if (RA) logically not less than EXTS(IM). <i>Extended mnemonic for twi 5,RA,IM</i>		
twlti		Trap if (RA) less than EXTS(IM). <i>Extended mnemonic for twi 16,RA,IM</i>		
twnei		Trap if (RA) not equal to EXTS(IM). <i>Extended mnemonic for twi 24,RA,IM</i>		
twngi		Trap if (RA) not greater than EXTS(IM). <i>Extended mnemonic for twi 20,RA,IM</i>		
twnli		Trap if (RA) not less than EXTS(IM). <i>Extended mnemonic for twi 12,RA,IM</i>		
twi	TO, RA, IM	Trap exception is generated if, comparing (RA) with EXTS(IM), any condition specified by TO is true.		

Table A-1. PPC401A1 Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
wrttee	RS	Write value of RS ₁₆ to the External Enable bit (MSR[EE]).		9-175
wrtteei	E	Write value of E to the External Enable bit (MSR[EE]).		
xor	RA, RS, RB	XOR (RS) with (RB). Place result in RA.		
xor.			CR[CR0]	
xori	RA, RS, IM	XOR (RS) with (¹⁶ 0 IM). Place result in RA.		
xoris	RA, RS, IM	XOR (RS) with (IM ¹⁶ 0). Place result in RA.		

A.2 Instructions Sorted by Opcode

All instructions are four bytes long and word aligned. All instructions have a primary opcode field (shown as field OPCODE in Figure A-1 through Figure A-9 beginning on p. A-49) in bits 0:5. Some instructions also have a secondary opcode field (shown as field XO in Figure A-1 through Figure A-9). PPC401A1 instructions sorted by primary and secondary opcode may be found in Table A-2 below.

The “Form” indicated in the table refers to the arrangement of valid field combinations within the four-byte instruction. See Section A.3 (Instruction Formats) on p. A-46 for illustration of the field layouts associated with each form.

Form X has a 10-bit secondary opcode field, while form XO uses only the low-order 9-bits of that field. Form XO uses the high-order secondary opcode bit (the tenth bit) as a variable; therefore, every form XO instruction really consumes two secondary opcodes from the 10-bit secondary-opcode space. The implicitly consumed secondary opcode is listed in parentheses for form XO instructions in the table below.

Table A-2. PPC401A1 Instructions by Opcode

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
3		D	twi	TO, RA, IM	
7		D	mulli	RT, RA, IM	9-118
8		D	subfic	RT, RA, IM	9-163
10		D	cmpli	BF, 0, RA, IM	9-40

Table A-2. PPC401A1 Instructions by Opcode (cont.)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
11		D	cmpi	BF, 0, RA, IM	9-38
12		D	addic	RT, RA, IM	9-10
13		D	addic.	RT, RA, IM	9-11
14		D	addi	RT, RA, IM	9-9
15		D	addis	RT, RA, IM	9-12
16		B	bc	BO, BI, target	9-20
			bca		
			bcl		
			bcla		
17		SC	sc		9-134
18		I	b	target	9-19
			ba		
			bl		
			bla		
19	0	XL	mcrf	BF, BFA	9-103
19	16	XL	bclr	BO, BI	9-32
			bclrl		
19	33	XL	crnor	BT, BA, BB	9-46
19	50	XL	rfi		9-128
19	51	XL	rfci		9-127
19	129	XL	crandc	BT, BA, BB	9-43
19	150	XL	isync		9-76
19	193	XL	crxor	BT, BA, BB	9-49
19	225	XL	crnand	BT, BA, BB	9-45
19	257	XL	crand	BT, BA, BB	9-42
19	289	XL	creqv	BT, BA, BB	9-44
19	417	XL	crorc	BT, BA, BB	9-48
19	449	XL	cror	BT, BA, BB	9-47

Table A-2. PPC401A1 Instructions by Opcode (cont.)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
19	528	XL	bcctr	BO, BI	9-28
			bcctrl		
20		M	rlwimi	RA, RS, SH, MB, ME	9-129
			rlwimi.		
21		M	rlwinm	RA, RS, SH, MB, ME	9-130
			rlwinm.		
23		M	rlwnm	RA, RS, RB, MB, ME	9-133
			rlwnm.		
24		D	ori	RA, RS, IM	9-125
25		D	oris	RA, RS, IM	9-126
26		D	xori	RA, RS, IM	
27		D	xoris	RA, RS, IM	
28		D	andi.	RA, RS, IM	9-17
29		D	andis.	RA, RS, IM	9-18
31	0	X	cmp	BF, 0, RA, RB	9-37
31	4	X	tw	TO, RA, RB	
31	8 (520)	XO	subfc	RT, RA, RB	9-160
			subfc.		
			subfco		
			subfco.		
31	10 (522)	XO	addc	RT, RA, RB	9-7
			addc.		
			addco		
			addco.		
31	11 (523)	XO	mulhwu	RT, RA, RB	9-117
			mulhwu.		
31	19	X	mfcrr	RT	9-105
31	20	X	lwarx	RT, RA, RB	9-96
31	23	X	lwzxx	RT, RA, RB	9-102

Table A-2. PPC401A1 Instructions by Opcode (cont.)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
31	24	X	slw	RA, RS, RB	9-135
			slw.		
31	26	X	cntlzw	RA, RS	9-41
			cntlzw.		
31	28	X	and	RA, RS, RB	9-15
			and.		
31	32	X	cmpl	BF, 0, RA, RB	9-39
31	40 (552)	XO	subf	RT, RA, RB	9-159
			subf.		
			subfo		
			subfo.		
31	54	X	dcbst	RA, RB	9-52
31	55	X	lwzux	RT, RA, RB	9-101
31	60	X	andc	RA, RS, RB	9-16
			andc.		
31	75 (587)	XO	mulhw	RT, RA, RB	9-116
			mulhw.		
31	83	X	mfmsr	RT	9-107
31	86	X	dcbf	RA, RB	9-50
31	87	X	lbzx	RT, RA, RB	9-81
31	104 (616)	XO	neg	RT, RA	9-121
			neg.		
			nego		
			nego.		
31	119	X	lbzux	RT, RA, RB	9-80
31	124	X	nor	RA, RS, RB	9-122
			nor.		
31	131	X	wrttee	RS	

Table A-2. PPC401A1 Instructions by Opcode (cont.)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
31	136 (648)	XO	subfe	RT, RA, RB	9-162
			subfe.		
			subfeo		
			subfeo.		
31	138 (650)	XO	adde	RT, RA, RB	9-8
			adde.		
			addeo		
			addeo.		
31	144	XFX	mtcrf	FXM, RS	9-110
31	146	X	mtmsr	RS	9-113
31	150	X	stwcx.	RS, RA, RB	9-154
31	151	X	stwx	RS, RA, RB	9-158
31	163	X	wrtnei	E	
31	183	X	stwux	RS, RA, RB	9-157
31	200 (712)	XO	subfze	RT, RA, RB	9-165
			subfze.		
			subfzeo		
			subfzeo.		
31	202 (714)	XO	addze	RT, RA	9-14
			addze.		
			addzeo		
			addzeo.		
31	215	X	stbx	RS, RA, RB	9-142
31	232 (744)	XO	subfme	RT, RA, RB	9-164
			subfme.		
			subfmeo		
			subfmeo.		

Table A-2. PPC401A1 Instructions by Opcode (cont.)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
31	234 (746)	XO	addme	RT, RA	9-13
			addme.		
			addmeo		
			addmeo.		
31	235 (747)	XO	mullw	RT, RA, RB	9-119
			mullw.		
			mullwo		
			mullwo.		
31	246	X	dcbtst	RA, RB	9-55
31	247	X	stbux	RS, RA, RB	9-141
31	262	X	icbt	RA, RB	9-71
31	266 (778)	XO	add	RT, RA, RB	9-6
			add.		
			addo		
			addo.		
31	278	X	dcbt	RA, RB	9-53
31	279	X	lhzx	RT, RA, RB	9-90
31	284	X	eqv	RA, RS, RB	9-66
			eqv.		
31	311	X	lhzux	RT, RA, RB	9-89
31	316	X	xor	RA, RS, RB	
			xor.		
31	323	XFX	mfdcr	RT, DCRN	9-106
31	339	XFX	mfspr	RT, SPRN	9-108
31	343	X	lhax	RT, RA, RB	9-85
31	375	X	lhaux	RT, RA, RB	9-84
31	407	X	sthx	RS, RA, RB	9-147
31	412	X	orc	RA, RS, RB	9-124
			orc.		

Table A-2. PPC401A1 Instructions by Opcode (cont.)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
31	439	X	sthux	RS, RA, RB	9-146
31	444	X	or	RA, RS, RB	9-123
			or.		
31	451	XFX	mtdcr	DCRN, RS	9-112
31	454	X	dccci	RA, RB	9-59
31	459 (971)	XO	divwu	RT, RA, RB	9-64
			divwu.		
			divwuo		
			divwuo.		
31	467	XFX	mtspr	SPRN, RS	9-114
31	470	X	dcbi	RA, RB	9-51
31	476	X	nand	RA, RS, RB	9-120
			nand.		
31	486	X	dcread	RT, RA, RB	9-61
31	491 (1003)	XO	divw	RT, RA, RB	9-63
			divw.		
			divwo		
			divwo.		
31	512	X	mcrxr	BF	9-104
31	533	X	lswx	RT, RA, RB	9-94
31	534	X	lwbrx	RT, RA, RB	9-98
31	536	X	srw	RA, RS, RB	9-138
			srw.		
31	597	X	lswi	RT, RA, NB	9-92
31	598	X	sync		9-166
31	661	X	stswx	RS, RA, RB	9-150
31	662	X	stwbrx	RS, RA, RB	9-153
31	725	X	stswi	RS, RA, NB	9-149
31	790	X	lhbrx	RT, RA, RB	9-86

Table A-2. PPC401A1 Instructions by Opcode (cont.)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
31	792	X	sraw	RA, RS, RB	9-136
			sraw.		
31	824	X	srawi	RA, RS, SH	9-137
			srawi.		
31	854	X	eieio		9-65
31	918	X	sthbrx	RS, RA, RB	9-144
31	922	X	extsh	RA, RS	9-68
			extsh.		
31	954	X	extsb	RA, RS	9-67
			extsb.		
31	966	X	iccci	RA, RB	9-73
31	982	X	icbi	RA, RB	9-69
31	998	X	icread	RA, RB	9-74
31	1014	X	dcbz	RA, RB	9-57
31	TBD	X	dcba	RA, RB	9-50
32		D	lwz	RT, D(RA)	9-99
33		D	lwzu	RT, D(RA)	9-100
34		D	lbz	RT, D(RA)	9-78
35		D	lbzu	RT, D(RA)	9-79
36		D	stw	RS, D(RA)	9-152
37		D	stwu	RS, D(RA)	9-156
38		D	stb	RS, D(RA)	9-139
39		D	stbu	RS, D(RA)	9-140
40		D	lhz	RT, D(RA)	9-87
41		D	lhzu	RT, D(RA)	9-88
42		D	lha	RT, D(RA)	9-82
43		D	lhau	RT, D(RA)	9-83
44		D	sth	RS, D(RA)	9-143
45		D	sthu	RS, D(RA)	9-145

Table A-2. PPC401A1 Instructions by Opcode (cont.)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
46		D	lmw	RT, D(RA)	9-91
47		D	stmw	RS, D(RA)	9-148

A.3 Instruction Formats

Instructions are four bytes long. Instruction addresses are always word-aligned.

Instruction bits 0 through 5 always contain the primary opcode. Many instructions have an extended opcode in another field. The remaining instruction bits contain additional fields. All instruction fields belong to one of the following categories:

- **Defined**

These instructions contain values, such as opcodes, that cannot be altered. The instruction format diagrams specify the values of defined fields.

- **Variable**

These fields contain operands, such as general purpose register selectors and immediate values, that may vary from execution to execution. The instruction format diagrams specify the operands in variable fields.

- **Reserved**

Bits in a reserved field should be set to 0. In the instruction format diagrams, reserved fields are shaded.

If any bit in a defined field does not contain the expected value, the instruction is illegal and an illegal instruction exception occurs. If any bit in a reserved field does not contain 0, the instruction form is invalid and its result is architecturally undefined. The PPC401A1 executes all invalid instruction forms without causing an illegal instruction exception.

A.3.1 Instruction Fields

A

PPC401A1 instructions contain various combinations of the following fields, as indicated in the instruction format diagrams. The numbers, enclosed in parentheses, that follow the field names indicate the bit positions; bit fields are indicated by starting and stopping bit positions separated by colons.

AA (30) Absolute address bit.

- 0 The immediate field represents an address relative to the current instruction address (CIA). The effective address (EA) of the branch is either the sum of the LI field sign-extended to 32 bits and the branch instruction address, or the sum of the BD field sign-extended to 32 bits

and the branch instruction address.

- 1 The immediate field represents an absolute address. The EA of the branch is either the LI field or the BD field, sign-extended to 32 bits.

BA (11:15)	Specifies a bit in the condition register (CR) used as a source of a CR-logical instruction.
BB (16:20)	Specifies a bit in the CR used as a source of a CR-logical instruction.
BD (16:29)	An immediate field specifying a 14-bit signed twos complement branch displacement. This field is concatenated on the right with 0b00 and sign-extended to 32 bits.
BF (6:8)	Specifies a field in the CR used as a target in a compare or mcrf instruction.
BFA (11:13)	Specifies a field in the CR used as a source in a mcrf instruction.
BI (11:15)	Specifies a bit in the CR used as a source for the condition of a conditional branch instruction.
BO (6:10)	Specifies options for conditional branch instructions. See Section 2.6.4.
BT (6:10)	Specifies a bit in the CR used as a target as the result of a CR-Logical instruction.
D (16:31)	Specifies a 16-bit signed two's-complement integer displacement for load/store instructions.
DCRN (11:20)	Specifies a device control register (DCR).
FXM (12:19)	Field mask used to identify CR fields to be updated by the mtcrf instruction.
IM (16:31)	An immediate field used to specify a 16-bit value (either signed integer or unsigned).
LI (6:29)	An immediate field specifying a 24-bit signed twos complement branch displacement; this field is concatenated on the right with b'00' and sign-extended to 32 bits.
LK (31)	Link bit. 0 Do not update the link register (LR). 1 Update the LR with the address of the next instruction.
MB (21:25)	Mask begin. Used in rotate-and-mask instructions to specify the beginning bit of a mask.
ME (26:30)	Mask end. Used in rotate-and-mask instructions to specify the ending bit of a mask.

NB (16:20)	Specifies the number of bytes to move in an immediate string load or store.
OPCD (0:5)	Primary opcode. Primary opcodes, in decimal, appear in the instruction format diagrams presented with individual instructions. The OPCD field name does not appear in instruction descriptions.
OE (21)	Enables setting the OV and SO fields in the fixed-point exception register (XER) for extended arithmetic.
RA (11:15)	A GPR used as a source or target.
RB (16:20)	A GPR used as a source.
Rc (31)	Record bit. 0 Do not set the CR. 1 Set the CR to reflect the result of an operation. See Section 2.2.3, “Condition Register (CR),” on p. 2-9, for a further discussion of how the CR bits are set.
RS (6:10)	A GPR used as a source.
RT (6:10)	A GPR used as a target.
SH (16:20)	Specifies a shift amount.
SPRF (11:20)	Specifies a special purpose register (SPR).
TO (6:10)	Specifies the conditions on which to trap, as described under tw and twi instructions.
XO (21:30)	Extended opcode for instructions without an OE field. Extended opcodes, in decimal, appear in the instruction format diagrams presented with individual instructions. The XO field name does not appear in instruction descriptions.
XO (22:30)	Extended opcode for instructions with an OE field. Extended opcodes, in decimal, appear in the instruction format diagrams presented with individual instructions. The XO field name does not appear in instruction descriptions.

A

A.3.2 Instruction Format Diagrams

The “Forms” shown in Figure A-1 through Figure A-9 are valid combinations of instruction fields for the PPC401A1. Table A-2 on p. A-38 indicates which “Form” is utilized by each PPC401A1 opcode. Fields indicated by slashes (/, //, or ///) are reserved. These figures have been adapted from the PowerPC User Instruction Set Architecture.

I-Form

OPCD		LI																								AA	LK
0	6																									30	31

Figure A-1. Instruction Format

B-Form

OPCD	BO	BI	BD																AA	LK	
0	6	11	16																	30	31

Figure A-2. B Instruction Format

SC-Form

OPCD		///		///		///														1	/		
0		6		11		16																3031	

Figure A-3. SC Instruction Format

D-Form

OPCD		RT		RA		D		
OPCD		RT		RA		SI		
OPCD		RS		RA		D		
OPCD		RS		RA		UI		
OPCD		BF	/	L	RA		SI	
OPCD		BF	/	L	RA		UI	
OPCD		TO		RA		SI		
0	6	11	16	31				

Figure A-4. D Instruction Format

A

X-Form

OPCD	RT	RA	RB	XO	Rc
OPCD	RT	RA	RB	XO	/
OPCD	RT	RA	NB	XO	/
OPCD	RT	RA	WS	XO	/
OPCD	RT	///	RB	XO	/
OPCD	RT	///	///	XO	/
OPCD	RS	RA	RB	XO	Rc
OPCD	RS	RA	RB	XO	1
OPCD	RS	RA	RB	XO	/
OPCD	RS	RA	NB	XO	/
OPCD	RS	RA	WS	XO	/
OPCD	RS	RA	SH	XO	Rc
OPCD	RS	RA	///	XO	Rc
OPCD	RS	///	RB	XO	/
OPCD	RS	///	///	XO	/
OPCD	BF	/ L	RA	RB	/
OPCD	BF	//	BFA	//	/
OPCD	BF	//	///	U	/ Rc
OPCD	BF	//	///	///	/
OPCD	TO	RA	RB	XO	/
OPCD	BT	///	///	XO	Rc
OPCD	///	RA	RB	XO	/
OPCD	///	///	RB	XO	/
OPCD	///	///	///	XO	/
OPCD	///	///	E	//	/
0	6	11	16	21	31

Figure A-5. X Instruction Format

XL-Form

OPCD	BT	BA	BB	XO	/
OPCD	BO	BI	///	XO	LK
OPCD	BF	//	BFA	///	XO
OPCD	///	///	///	XO	/
0	6	11	16	21	31

Figure A-6. XL Instruction Format

XFX-Form

OPCD	RT	SPRF	XO	/
OPCD	RT	DCRF	XO	/
OPCD	RT	/	FXM	/
OPCD	RS	SPRF	XO	/
OPCD	RS	DCRF	XO	/
0	6	11	21	31

Figure A-7. XFX Instruction Format

XO-Form

OPCD	RT	RA	RB	O E	XO	Rc
OPCD	RT	RA	RB	/	XO	Rc
OPCD	RT	RA	///	O E	XO	Rc
0	6	11	16	21 22		31

Figure A-8. XO Instruction Format

M-Form

OPCD	RS	RA	RB	MB	ME	Rc
OPCD	RS	RA	SH	MB	ME	Rc
0	6	11	16	21	26	31

Figure A-9. M Instruction Format

A

B

Instructions By Category

B.1 Instruction Set Summary Categories

Chapter 9, “Instruction Set,” contains detailed descriptions of the instructions, their operands, and notation.

Table B.1 summarizes the instruction categories in the PPC401A1 instruction set. The instructions within each category are listed in subsequent tables.

Table B-1. PPC401A1 Instruction Set Functional Summary

Storage Reference	load, store
Arithmetic and Logical	add, subtract, negate, multiply, divide, and, andc, or, orc, xor, nand, nor, xnor, sign extension, count leading zeros
Comparison	compare, compare logical, compare immediate
Branch	branch, branch conditional, branch to LR, branch to CTR
CR Logical	crand, crandc, cror, crorc, crnand, crnor, crxor, crxnor, move CR field
Rotate/Shift	rotate and insert, rotate and mask, shift left, shift right
Cache Control	invalidate, touch, zero, flush, store, read
Interrupt Control	write to external interrupt enable bit, move to/from MSR, return from interrupt, return from critical interrupt
Processor Management	system call, synchronize, trap, move to/from DCRs, move to/from SPRs, move to/from CR

B.2 Instructions Specific to PowerPC Embedded Controllers

To meet the functional requirements of processors for embedded systems and real-time applications, the PowerPC Embedded Controller family defines instructions that are not part of the PowerPC Architecture.

Table B-2 summarizes the PPC401A1 instructions specific to the PowerPC Embedded Controller family.

Table B-2. Instructions Specific to PowerPC Embedded Controllers

Mnemonic	Operands	Function	Other Registers Changed	Page
dccci	RA, RB	Invalidate the data cache congruence class associated with the effective address (RA 0) + (RB).		9-59
dcread	RT, RA, RB	Read either tag or data information from the data cache congruence class associated with the effective address (RA 0) + (RB). Place the results in RT.		9-61
icbt	RA, RB	Load the instruction cache block which contains the effective address (RA 0) + (RB).		9-71
iccci	RA, RB	Invalidate instruction cache congruence class associated with the effective address (RA 0) + (RB).		9-73
icread	RA, RB	Read either tag or data information from the instruction cache congruence class associated with the effective address (RA 0) + (RB). Place the results in ICDBDR.		9-74
mfdcr	RT, DCRN	Move from DCR to RT, (RT) \leftarrow (DCR(DCRN)).		9-106
mtdcr	DCRN, RS	Move to DCR from RS, (DCR(DCRN)) \leftarrow (RS).		9-112
rfci		Return from critical interrupt (PC) \leftarrow (SRR2). (MSR) \leftarrow (SRR3).		9-127
wrtree	RS	Write value of RS ₁₆ to MSR[EE].		
wrtteei	E	Write value of E to MSR[EE].		

B.3 Privileged Instructions

Table B-3 lists instructions that are under control of the MSR[PR] bit. These instructions are not allowed to be executed when MSR[PR] = 1:

Table B-3. Privileged Instructions

Mnemonic	Operands	Function	Other Registers Changed	Page
dcbi	RA, RB	Invalidate the data cache block which contains the effective address (RA 0) + (RB).		9-51
dccci	RA, RB	Invalidate the data cache congruence class associated with the effective address (RA 0) + (RB).		9-59
dcread	RT, RA, RB	Read either tag or data information from the data cache congruence class associated with the effective address (RA 0) + (RB). Place the results in RT.		9-61
icbt	RA, RB	Load the instruction cache block which contains the effective address (RA 0) + (RB).		9-71
iccci	RA, RB	Invalidate instruction cache congruence class associated with the effective address (RA 0) + (RB).		9-73
icread	RA, RB	Read either tag or data information from the instruction cache congruence class associated with the effective address (RA 0) + (RB). Place the results in ICDBDR.		9-74
mfdcr	RT, DCRN	Move from DCR to RT, (RT) ← (DCR(DCRN)).		9-106
mfmsr	RT	Move from MSR to RT, (RT) ← (MSR).		9-107
mfspr	RT, SPRN	Move from SPR to RT, (RT) ← (SPR(SPRN)). Privileged for all SPRs except LR, CTR, TBHU, TBLU, and XER.		9-108
mtdcr	DCRN, RS	Move to DCR from RS, (DCR(DCRN)) ← (RS).		9-112
mtmsr	RS	Move to MSR from RS, (MSR) ← (RS).		9-113

Table B-3. Privileged Instructions (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
mtspr	SPRN, RS	Move to SPR from RS, (SPR(SPRN)) \leftarrow (RS). Privileged for all SPRs except LR, CTR, TBHU, TBLU, and XER.		9-114
rfci		Return from critical interrupt (PC) \leftarrow (SRR2). (MSR) \leftarrow (SRR3).		9-127
rfi		Return from interrupt. (PC) \leftarrow (SRR0). (MSR) \leftarrow (SRR1).		9-128
wrtee	RS	Write value of RS ₁₆ to the External Enable bit (MSR[EE]).		
wrteei	E	Write value of E to the External Enable bit (MSR[EE]).		

B.4 Assembler Extended Mnemonics

In the appendix “Assembler Extended Mnemonics” of the PowerPC Architecture, it is required that a PowerPC assembler support at least a minimal set of extended mnemonics. These mnemonics encode to the opcodes of other instructions; the only benefit of extended mnemonics is improved usability. Code using extended mnemonics can be easier to write and to understand. Table B-4 lists the extended mnemonics required for the PPC401A1.

Note the following for every Branch Conditional mnemonic:

Bit 4 of the BO field provides a hint about the most likely outcome of a conditional branch (see Section 2.6.5 for a full discussion of Branch Prediction). Assemblers should set $BO_4 = 0$ unless a specific reason exists otherwise. In the BO field values specified in the table below, $BO_4 = 0$ has always been assumed. The assembler must allow the programmer to specify Branch Prediction. To do this, the assembler will support a suffix to every conditional branch mnemonic, as follows:

- + Predict branch to be taken.
- Predict branch not to be taken.

As specific examples, **bc** also could be coded as **bc+** or **bc-**, and **bne** also could be coded **bne+** or **bne-**. These alternate codings set $BO_4 = 1$ only if the requested prediction differs from the Standard Prediction (see Section 2.6.5).

Table B-4. Extended Mnemonics for PPC401A1

Mnemonic	Operands	Function	Other Registers Changed	Page
bctr		Branch unconditionally, to address in CTR. <i>Extended mnemonic for bcctr 20,0</i>		9-28
bctrl		<i>Extended mnemonic for bcctrl 20,0</i>	(LR) \leftarrow CIA + 4.	
bdnz	target	Decrement CTR. Branch if CTR \neq 0. <i>Extended mnemonic for bc 16,0,target</i>		9-20
bdnza		<i>Extended mnemonic for bca 16,0,target</i>		
bdnzl		<i>Extended mnemonic for bcl 16,0,target</i>	(LR) \leftarrow CIA + 4.	
bdnzla		<i>Extended mnemonic for bcla 16,0,target</i>	(LR) \leftarrow CIA + 4.	
bdnzlr		Decrement CTR. Branch if CTR \neq 0, to address in LR. <i>Extended mnemonic for bclr 16,0</i>		9-32
bdnzlrl		<i>Extended mnemonic for bclrl 16,0</i>	(LR) \leftarrow CIA + 4.	
bdnzf	cr_bit, target	Decrement CTR. Branch if CTR \neq 0 AND $CR_{cr_bit} = 0$. <i>Extended mnemonic for bc 0,cr_bit,target</i>		9-20
bdnzfa		<i>Extended mnemonic for bca 0,cr_bit,target</i>		
bdnzfl		<i>Extended mnemonic for bcl 0,cr_bit,target</i>	(LR) \leftarrow CIA + 4.	
bdnzfla		<i>Extended mnemonic for bcla 0,cr_bit,target</i>	(LR) \leftarrow CIA + 4.	

Table B-4. Extended Mnemonics for PPC401A1 (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
bdnzflr	cr_bit	Decrement CTR. Branch if CTR \neq 0 AND CR _{cr_bit} = 0, to address in LR. <i>Extended mnemonic for</i> bclr 0,cr_bit		9-32
bdnzflrl		<i>Extended mnemonic for</i> bclrl 0,cr_bit	(LR) \leftarrow CIA + 4.	
bdnzt	cr_bit, target	Decrement CTR. Branch if CTR \neq 0 AND CR _{cr_bit} = 1. <i>Extended mnemonic for</i> bc 8,cr_bit,target		9-20
bdnzta		<i>Extended mnemonic for</i> bca 8,cr_bit,target		
bdnztl		<i>Extended mnemonic for</i> bcl 8,cr_bit,target	(LR) \leftarrow CIA + 4.	
bdnztla		<i>Extended mnemonic for</i> bcla 8,cr_bit,target	(LR) \leftarrow CIA + 4.	
bdnztlr	cr_bit	Decrement CTR. Branch if CTR \neq 0 AND CR _{cr_bit} = 1, to address in LR. <i>Extended mnemonic for</i> bclr 8,cr_bit		9-32
bdnztlrl		<i>Extended mnemonic for</i> bclrl 8,cr_bit	(LR) \leftarrow CIA + 4.	
bdz	target	Decrement CTR. Branch if CTR = 0. <i>Extended mnemonic for</i> bc 18,0,target		9-20
bdza		<i>Extended mnemonic for</i> bca 18,0,target		
bdzl		<i>Extended mnemonic for</i> bcl 18,0,target	(LR) \leftarrow CIA + 4.	
bdzla		<i>Extended mnemonic for</i> bcla 18,0,target	(LR) \leftarrow CIA + 4.	

Table B-4. Extended Mnemonics for PPC401A1 (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
bdzlr		Decrement CTR. Branch if CTR = 0, to address in LR. <i>Extended mnemonic for</i> bclr 18,0		9-32
bdzlrl		<i>Extended mnemonic for</i> bclrl 18,0	(LR) \leftarrow CIA + 4.	
bdzfb	cr_bit, target	Decrement CTR. Branch if CTR = 0 AND CR _{cr_bit} = 0. <i>Extended mnemonic for</i> bc 2,cr_bit,target		9-20
bdzfa		<i>Extended mnemonic for</i> bca 2,cr_bit,target		
bdzfl		<i>Extended mnemonic for</i> bcl 2,cr_bit,target	(LR) \leftarrow CIA + 4.	
bdzfla		<i>Extended mnemonic for</i> bcla 2,cr_bit,target	(LR) \leftarrow CIA + 4.	
bdzflr	cr_bit	Decrement CTR. Branch if CTR = 0 AND CR _{cr_bit} = 0 to address in LR. <i>Extended mnemonic for</i> bclr 2,cr_bit		9-32
bdzflrl		<i>Extended mnemonic for</i> bclrl 2,cr_bit	(LR) \leftarrow CIA + 4.	
bdzt	cr_bit, target	Decrement CTR. Branch if CTR = 0 AND CR _{cr_bit} = 1. <i>Extended mnemonic for</i> bc 10,cr_bit,target		9-20
bdzta		<i>Extended mnemonic for</i> bca 10,cr_bit,target		
bdztl		<i>Extended mnemonic for</i> bcl 10,cr_bit,target	(LR) \leftarrow CIA + 4.	
bdztla		<i>Extended mnemonic for</i> bcla 10,cr_bit,target	(LR) \leftarrow CIA + 4.	

Table B-4. Extended Mnemonics for PPC401A1 (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
bdztlr	cr_bit	Decrement CTR. Branch if CTR = 0 AND CR _{cr_bit} = 1, to address in LR. <i>Extended mnemonic for</i> bclr 10,cr_bit		9-32
bdztlrl		<i>Extended mnemonic for</i> bclrl 10,cr_bit	(LR) ← CIA + 4.	
beq	[cr_field,] target	Branch if equal. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 12,4*cr_field+2,target		9-20
beqa		<i>Extended mnemonic for</i> bca 12,4*cr_field+2,target		
beql		<i>Extended mnemonic for</i> bcl 12,4*cr_field+2,target	(LR) ← CIA + 4.	
beqla		<i>Extended mnemonic for</i> bcla 12,4*cr_field+2,target	(LR) ← CIA + 4.	
beqctr	[cr_field]	Branch if equal, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bcctr 12,4*cr_field+2		9-28
beqctrl		<i>Extended mnemonic for</i> bcctrl 12,4*cr_field+2	(LR) ← CIA + 4.	
beqlr	[cr_field]	Branch if equal, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 12,4*cr_field+2		9-32
beqlrl		<i>Extended mnemonic for</i> bclrl 12,4*cr_field+2	(LR) ← CIA + 4.	

Table B-4. Extended Mnemonics for PPC401A1 (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
bf	cr_bit, target	Branch if CR _{cr_bit} = 0. <i>Extended mnemonic for</i> bc 4,cr_bit,target		9-20
bfa		<i>Extended mnemonic for</i> bca 4,cr_bit,target		
bfl		<i>Extended mnemonic for</i> bcl 4,cr_bit,target	(LR) ← CIA + 4.	
bfla		<i>Extended mnemonic for</i> bcla 4,cr_bit,target	(LR) ← CIA + 4.	
bfctr	cr_bit	Branch if CR _{cr_bit} = 0, to address in CTR. <i>Extended mnemonic for</i> bcctr 4,cr_bit		9-28
bfctrl		<i>Extended mnemonic for</i> bcctrl 4,cr_bit	(LR) ← CIA + 4.	
bflr	cr_bit	Branch if CR _{cr_bit} = 0, to address in LR. <i>Extended mnemonic for</i> bclr 4,cr_bit		9-32
bflrl		<i>Extended mnemonic for</i> bclrl 4,cr_bit	(LR) ← CIA + 4.	
bge	[cr_field,] target	Branch if greater than or equal. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 4,4*cr_field+0,target		9-20
bgea		<i>Extended mnemonic for</i> bca 4,4*cr_field+0,target		
bgel		<i>Extended mnemonic for</i> bcl 4,4*cr_field+0,target	(LR) ← CIA + 4.	
bgeLa		<i>Extended mnemonic for</i> bcla 4,4*cr_field+0,target	(LR) ← CIA + 4.	
bgectr	[cr_field]	Branch if greater than or equal, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bcctr 4,4*cr_field+0		9-28
bgectrl		<i>Extended mnemonic for</i> bcctrl 4,4*cr_field+0	(LR) ← CIA + 4.	

Table B-4. Extended Mnemonics for PPC401A1 (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
bgelr	[cr_field]	Branch if greater than or equal, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 4,4*cr_field+0		9-32
bgelrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+0	(LR) ← CIA + 4.	
bgt	[cr_field,] target	Branch if greater than. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 12,4*cr_field+1,target		9-20
bgta		<i>Extended mnemonic for</i> bca 12,4*cr_field+1,target		
bgtl		<i>Extended mnemonic for</i> bcl 12,4*cr_field+1,target	(LR) ← CIA + 4.	
bgtla		<i>Extended mnemonic for</i> bcla 12,4*cr_field+1,target	(LR) ← CIA + 4.	
bgtctr	[cr_field]	Branch if greater than, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bcctr 12,4*cr_field+1		9-28
bgtctrl		<i>Extended mnemonic for</i> bcctrl 12,4*cr_field+1	(LR) ← CIA + 4.	
bgtlr	[cr_field]	Branch if greater than, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 12,4*cr_field+1		9-32
bgtlrl		<i>Extended mnemonic for</i> bclrl 12,4*cr_field+1	(LR) ← CIA + 4.	

Table B-4. Extended Mnemonics for PPC401A1 (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
ble	[cr_field,] target	Branch if less than or equal. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 4,4*cr_field+1,target		9-20
blea		<i>Extended mnemonic for</i> bca 4,4*cr_field+1,target		
blel		<i>Extended mnemonic for</i> bcl 4,4*cr_field+1,target	(LR) ← CIA + 4.	
blela		<i>Extended mnemonic for</i> bcla 4,4*cr_field+1,target	(LR) ← CIA + 4.	
blectr	[cr_field]	Branch if less than or equal, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bcctr 4,4*cr_field+1		9-28
blectrl		<i>Extended mnemonic for</i> bcctl 4,4*cr_field+1	(LR) ← CIA + 4.	
blelr	[cr_field]	Branch if less than or equal, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 4,4*cr_field+1		9-32
blelrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+1	(LR) ← CIA + 4.	
blr		Branch unconditionally, to address in LR. <i>Extended mnemonic for</i> bclr 20,0		9-32
blrl		<i>Extended mnemonic for</i> bclrl 20,0	(LR) ← CIA + 4.	

Table B-4. Extended Mnemonics for PPC401A1 (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
blt	[cr_field,] target	Branch if less than. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 12,4*cr_field+0,target		9-20
blta		<i>Extended mnemonic for</i> bca 12,4*cr_field+0,target		
bltl		<i>Extended mnemonic for</i> bcl 12,4*cr_field+0,target	(LR) ← CIA + 4.	
bltla		<i>Extended mnemonic for</i> bcla 12,4*cr_field+0,target	(LR) ← CIA + 4.	
bltctr	[cr_field]	Branch if less than, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bcctr 12,4*cr_field+0		9-28
bltctrl		<i>Extended mnemonic for</i> bcctrl 12,4*cr_field+0	(LR) ← CIA + 4.	
bltlr	[cr_field]	Branch if less than, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 12,4*cr_field+0		9-32
bltlrl		<i>Extended mnemonic for</i> bclrl 12,4*cr_field+0	(LR) ← CIA + 4.	
bne	[cr_field,] target	Branch if not equal. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 4,4*cr_field+2,target		9-20
bnea		<i>Extended mnemonic for</i> bca 4,4*cr_field+2,target		
bnel		<i>Extended mnemonic for</i> bcl 4,4*cr_field+2,target	(LR) ← CIA + 4.	
bnela		<i>Extended mnemonic for</i> bcla 4,4*cr_field+2,target	(LR) ← CIA + 4.	

Table B-4. Extended Mnemonics for PPC401A1 (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
bnctr	[cr_field]	Branch if not equal, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bcctr 4,4*cr_field+2		9-28
bnctrl		<i>Extended mnemonic for</i> bcctrl 4,4*cr_field+2	(LR) ← CIA + 4.	
bnelr	[cr_field]	Branch if not equal, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 4,4*cr_field+2		9-32
bnelrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+2	(LR) ← CIA + 4.	
bng	[cr_field,] target	Branch if not greater than. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 4,4*cr_field+1,target		9-20
bnga		<i>Extended mnemonic for</i> bca 4,4*cr_field+1,target		
bngl		<i>Extended mnemonic for</i> bcl 4,4*cr_field+1,target	(LR) ← CIA + 4.	
bngla		<i>Extended mnemonic for</i> bcla 4,4*cr_field+1,target	(LR) ← CIA + 4.	
bngctr	[cr_field]	Branch if not greater than, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bcctr 4,4*cr_field+1		9-28
bngctrl		<i>Extended mnemonic for</i> bcctrl 4,4*cr_field+1	(LR) ← CIA + 4.	
bnglrl	[cr_field]	Branch if not greater than, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclrl 4,4*cr_field+1		9-32
bnglrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+1	(LR) ← CIA + 4.	

Table B-4. Extended Mnemonics for PPC401A1 (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
bnl	[cr_field,] target	Branch if not less than. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 4,4*cr_field+0,target		9-20
bnla		<i>Extended mnemonic for</i> bca 4,4*cr_field+0,target		
bnll		<i>Extended mnemonic for</i> bcl 4,4*cr_field+0,target	(LR) ← CIA + 4.	
bnlla		<i>Extended mnemonic for</i> bcla 4,4*cr_field+0,target	(LR) ← CIA + 4.	
bnlctr	[cr_field]	Branch if not less than, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bcctr 4,4*cr_field+0		9-28
bnlctrl		<i>Extended mnemonic for</i> bcctrl 4,4*cr_field+0	(LR) ← CIA + 4.	
bnllr	[cr_field]	Branch if not less than, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 4,4*cr_field+0		9-32
bnllrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+0	(LR) ← CIA + 4.	
bns	[cr_field,] target	Branch if not summary overflow. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 4,4*cr_field+3,target		9-20
bnsa		<i>Extended mnemonic for</i> bca 4,4*cr_field+3,target		
bnsi		<i>Extended mnemonic for</i> bcl 4,4*cr_field+3,target	(LR) ← CIA + 4.	
bnsia		<i>Extended mnemonic for</i> bcla 4,4*cr_field+3,target	(LR) ← CIA + 4.	

Table B-4. Extended Mnemonics for PPC401A1 (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
bnsctr	[cr_field]	Branch if not summary overflow, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bcctr 4,4*cr_field+3		9-28
bnsctrl		<i>Extended mnemonic for</i> bcctrl 4,4*cr_field+3	(LR) ← CIA + 4.	
bnslr	[cr_field]	Branch if not summary overflow, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 4,4*cr_field+3		9-32
bnsrlr		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+3	(LR) ← CIA + 4.	
bnu	[cr_field,] target	Branch if not unordered. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 4,4*cr_field+3,target		9-20
bnuu		<i>Extended mnemonic for</i> bca 4,4*cr_field+3,target		
bnul		<i>Extended mnemonic for</i> bcl 4,4*cr_field+3,target	(LR) ← CIA + 4.	
bnula		<i>Extended mnemonic for</i> bcla 4,4*cr_field+3,target	(LR) ← CIA + 4.	
bnuctr	[cr_field]	Branch if not unordered, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bcctr 4,4*cr_field+3		9-28
bnuctrl		<i>Extended mnemonic for</i> bcctrl 4,4*cr_field+3	(LR) ← CIA + 4.	
bnulr	[cr_field]	Branch if not unordered, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 4,4*cr_field+3		9-32
bnulrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+3	(LR) ← CIA + 4.	

Table B-4. Extended Mnemonics for PPC401A1 (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
bso	[cr_field,] target	Branch if summary overflow. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 12,4*cr_field+3,target		9-20
bsoa		<i>Extended mnemonic for</i> bca 12,4*cr_field+3,target		
bsol		<i>Extended mnemonic for</i> bcl 12,4*cr_field+3,target	(LR) ← CIA + 4.	
bsola		<i>Extended mnemonic for</i> bcla 12,4*cr_field+3,target	(LR) ← CIA + 4.	
bsoctr	[cr_field]	Branch if summary overflow, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bcctr 12,4*cr_field+3		9-28
bsoctrl		<i>Extended mnemonic for</i> bcctrl 12,4*cr_field+3	(LR) ← CIA + 4.	
bsolr	[cr_field]	Branch if summary overflow, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 12,4*cr_field+3		9-32
bsolrl		<i>Extended mnemonic for</i> bclrl 12,4*cr_field+3	(LR) ← CIA + 4.	
bt	cr_bit, target	Branch if CR _{cr_bit} = 1. <i>Extended mnemonic for</i> bc 12,cr_bit,target		9-20
bta		<i>Extended mnemonic for</i> bca 12,cr_bit,target		
btl		<i>Extended mnemonic for</i> bcl 12,cr_bit,target	(LR) ← CIA + 4.	
btla		<i>Extended mnemonic for</i> bcla 12,cr_bit,target	(LR) ← CIA + 4.	

Table B-4. Extended Mnemonics for PPC401A1 (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
btctr	cr_bit	Branch if CR _{cr_bit} = 1, to address in CTR. <i>Extended mnemonic for</i> bcctr 12,cr_bit		9-28
btctrl		<i>Extended mnemonic for</i> bcctrl 12,cr_bit	(LR) ← CIA + 4.	
btlr	cr_bit	Branch if CR _{cr_bit} = 1, to address in LR. <i>Extended mnemonic for</i> bclr 12,cr_bit		9-32
btlrl		<i>Extended mnemonic for</i> bclrl 12,cr_bit	(LR) ← CIA + 4.	
bun	[cr_field,] target	Branch if unordered. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bc 12,4*cr_field+3,target		9-20
buna		<i>Extended mnemonic for</i> bca 12,4*cr_field+3,target		
bunl		<i>Extended mnemonic for</i> bcl 12,4*cr_field+3,target	(LR) ← CIA + 4.	
bunla		<i>Extended mnemonic for</i> bcla 12,4*cr_field+3,target	(LR) ← CIA + 4.	
bunctr	[cr_field]	Branch if unordered, to address in CTR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bcctr 12,4*cr_field+3		9-28
bunctrl		<i>Extended mnemonic for</i> bcctrl 12,4*cr_field+3	(LR) ← CIA + 4.	
bunlr	[cr_field]	Branch if unordered, to address in LR. Use CR0 if cr_field is omitted. <i>Extended mnemonic for</i> bclr 12,4*cr_field+3		9-32
bunlrl		<i>Extended mnemonic for</i> bclrl 12,4*cr_field+3	(LR) ← CIA + 4.	

Table B-4. Extended Mnemonics for PPC401A1 (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
clrlwi	RA, RS, n	Clear left immediate. ($n < 32$) $(RA)_{0:n-1} \leftarrow {}^n0$ <i>Extended mnemonic for</i> rlwinm RA,RS,0,n,31		9-130
clrlwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,0,n,31	CR[CR0]	
clrlslwi	RA, RS, b, n	Clear left and shift left immediate. $(n \leq b < 32)$ $(RA)_{b-n:31-n} \leftarrow (RS)_{b:31}$ $(RA)_{32-n:31} \leftarrow {}^n0$ $(RA)_{0:b-n-1} \leftarrow {}^{b-n}0$ <i>Extended mnemonic for</i> rlwinm RA,RS,n,b-n,31-n		9-130
clrlslwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,n,b-n,31-n	CR[CR0]	
clrrwi	RA, RS, n	Clear right immediate. ($n < 32$) $(RA)_{32-n:31} \leftarrow {}^n0$ <i>Extended mnemonic for</i> rlwinm RA,RS,0,0,31-n		9-130
clrrwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,0,0,31-n	CR[CR0]	
cmplw	[BF,] RA, RB	Compare Logical Word. Use CR0 if BF is omitted. <i>Extended mnemonic for</i> cmpl BF,0,RA,RB		9-39
cmplwi	[BF,] RA, IM	Compare Logical Word Immediate. Use CR0 if BF is omitted. <i>Extended mnemonic for</i> cmpli BF,0,RA,IM		9-40
cmpw	[BF,] RA, RB	Compare Word. Use CR0 if BF is omitted. <i>Extended mnemonic for</i> cmp BF,0,RA,RB		9-37
cmpwi	[BF,] RA, IM	Compare Word Immediate. Use CR0 if BF is omitted. <i>Extended mnemonic for</i> cmpi BF,0,RA,IM		9-38
crclr	bx	Condition register clear. <i>Extended mnemonic for</i> crxor bx,bx,bx		9-49

Table B-4. Extended Mnemonics for PPC401A1 (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
crmove	bx, by	Condition register move. <i>Extended mnemonic for</i> cror bx,by,by		9-47
crnot	bx, by	Condition register not. <i>Extended mnemonic for</i> crnor bx,by,by		9-46
crset	bx	Condition register set. <i>Extended mnemonic for</i> creqv bx,bx,bx		9-44
extlwi	RA, RS, n, b	Extract and left justify immediate. ($n > 0$) $(RA)_{0:n-1} \leftarrow (RS)_{b:b+n-1}$ $(RA)_{n:31} \leftarrow 32-n0$ <i>Extended mnemonic for</i> rlwinm RA,RS,b,0,n-1		9-130
extlwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,b,0,n-1	CR[CR0]	
extrwi	RA, RS, n, b	Extract and right justify immediate. ($n > 0$) $(RA)_{32-n:31} \leftarrow (RS)_{b:b+n-1}$ $(RA)_{0:31-n} \leftarrow 32-n0$ <i>Extended mnemonic for</i> rlwinm RA,RS,b+n,32-n,31		9-130
extrwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,b+n,32-n,31	CR[CR0]	
inslwi	RA, RS, n, b	Insert from left immediate. ($n > 0$) $(RA)_{b:b+n-1} \leftarrow (RS)_{0:n-1}$ <i>Extended mnemonic for</i> rlwimi RA,RS,32-b,b,b+n-1		9-129
inslwi.		<i>Extended mnemonic for</i> rlwimi. RA,RS,32-b,b,b+n-1	CR[CR0]	
insrwi	RA, RS, n, b	Insert from right immediate. ($n > 0$) $(RA)_{b:b+n-1} \leftarrow (RS)_{32-n:31}$ <i>Extended mnemonic for</i> rlwimi RA,RS,32-b-n,b,b+n-1		9-129
insrwi.		<i>Extended mnemonic for</i> rlwimi. RA,RS,32-b-n,b,b+n-1	CR[CR0]	

Table B-4. Extended Mnemonics for PPC401A1 (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
la	RT, D(RA)	Load address. ($RA \neq 0$) D is an offset from a base address that is assumed to be (RA). $(RT) \leftarrow (RA) + \text{EXTS}(D)$ <i>Extended mnemonic for</i> addi RT,RA,D		9-9
li	RT, IM	Load immediate. $(RT) \leftarrow \text{EXTS}(IM)$ <i>Extended mnemonic for</i> addi RT,0,value		9-9
lis	RT, IM	Load immediate shifted. $(RT) \leftarrow (IM \parallel 160)$ <i>Extended mnemonic for</i> addis RT,0,value		9-12

Table B-4. Extended Mnemonics for PPC401A1 (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
mfcdbcr mfctr mfdac1 mfdbsr mfdccr mfdcwr mfdear mfesr mfevpr mfiac1 mficcr mficdbdr mflr mfpit mfpvr mfsgsr mfsprg0 mfsprg1 mfsprg2 mfsprg3 mfsrr0 mfsrr1 mfsrr2 mfsrr3 mftbhi mftbhu mftblo mftblu mftcr mftsr mfxer	RT	Move from special purpose register (SPR) SPRN. <i>Extended mnemonic for</i> mfsspr RT,SPRN See Table 10-2 on p. 10-3 for listing of valid SPRN values.		9-108
mr	RT, RS	Move register. (RT) ← (RS) <i>Extended mnemonic for</i> or RT,RS,RS		9-123
mr.		<i>Extended mnemonic for</i> or. RT,RS,RS	CR[CR0]	
mtcr	RS	Move to Condition Register. <i>Extended mnemonic for</i> mtcrf 0xFF,RS		9-110

Table B-4. Extended Mnemonics for PPC401A1 (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
mtcdbcr mtctr mtdac1 mtdbcr mtdbsr mtdccr mtdcwr mtesr mtevr mtiac1 mticcr mticdbdr mtlr mtpit mtpvr mtsgr mtsrg0 mtsrg1 mtsrg2 mtsrg3 mtsrr0 mtsrr1 mtsrr2 mtsrr3 mtbhi mtblo mttcr mttsr mtxer	RS	Move to SPR SPRN. <i>Extended mnemonic for</i> mtspr SPRN,RS See Table 10-2 on p. 10-3 for listing of valid SPRN values.		9-114
nop		Preferred no-op, triggers optimizations based on no-ops. <i>Extended mnemonic for</i> ori 0,0,0		9-125
not	RA, RS	Complement register. $(RA) \leftarrow \neg(RS)$ <i>Extended mnemonic for</i> nor RA,RS,RS		9-122
not.		<i>Extended mnemonic for</i> nor. RA,RS,RS	CR[CR0]	

Table B-4. Extended Mnemonics for PPC401A1 (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
rotlw	RA, RS, RB	Rotate left. (RA) \leftarrow ROTL((RS), (RB) _{27:31}) <i>Extended mnemonic for</i> rlwnm RA,RS,RB,0,31		9-133
rotlw.		<i>Extended mnemonic for</i> rlwnm. RA,RS,RB,0,31	CR[CR0]	
rotlwi	RA, RS, n	Rotate left immediate. (RA) \leftarrow ROTL((RS), n) <i>Extended mnemonic for</i> rlwinm RA,RS,n,0,31		9-130
rotlwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,n,0,31	CR[CR0]	
rotrwi	RA, RS, n	Rotate right immediate. (RA) \leftarrow ROTL((RS), 32-n) <i>Extended mnemonic for</i> rlwinm RA,RS,32-n,0,31		9-130
rotrwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,32-n,0,31	CR[CR0]	
slwi	RA, RS, n	Shift left immediate. (n < 32) (RA) _{0:31-n} \leftarrow (RS) _{n:31} (RA) _{32-n:31} \leftarrow ⁿ 0 <i>Extended mnemonic for</i> rlwinm RA,RS,n,0,31-n		9-130
slwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,n,0,31-n	CR[CR0]	
srwi	RA, RS, n	Shift right immediate. (n < 32) (RA) _{n:31} \leftarrow (RS) _{0:31-n} (RA) _{0:n-1} \leftarrow ⁿ 0 <i>Extended mnemonic for</i> rlwinm RA,RS,32-n,n,31		9-130
srwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,32-n,n,31	CR[CR0]	

Table B-4. Extended Mnemonics for PPC401A1 (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
sub	RT, RA, RB	Subtract (RB) from (RA). $(RT) \leftarrow \neg(RB) + (RA) + 1$. <i>Extended mnemonic for subf RT,RB,RA</i>		9-159
sub.		<i>Extended mnemonic for subf. RT,RB,RA</i>	CR[CR0]	
subo		<i>Extended mnemonic for subfo RT,RB,RA</i>	XER[SO, OV]	
subo.		<i>Extended mnemonic for subfo. RT,RB,RA</i>	CR[CR0] XER[SO, OV]	
subc	RT, RA, RB	Subtract (RB) from (RA). $(RT) \leftarrow \neg(RB) + (RA) + 1$. Place carry-out in XER[CA]. <i>Extended mnemonic for subfc RT,RB,RA</i>		9-160
subc.		<i>Extended mnemonic for subfc. RT,RB,RA</i>	CR[CR0]	
subco		<i>Extended mnemonic for subfco RT,RB,RA</i>	XER[SO, OV]	
subco.		<i>Extended mnemonic for subfco. RT,RB,RA</i>	CR[CR0] XER[SO, OV]	
subi	RT, RA, IM	Subtract EXTS(IM) from (RA 0). Place result in RT. <i>Extended mnemonic for addi RT,RA,-IM</i>		9-9
subic	RT, RA, IM	Subtract EXTS(IM) from (RA). Place result in RT. Place carry-out in XER[CA]. <i>Extended mnemonic for addic RT,RA,-IM</i>		9-10
subic.	RT, RA, IM	Subtract EXTS(IM) from (RA). Place result in RT. Place carry-out in XER[CA]. <i>Extended mnemonic for addic. RT,RA,-IM</i>	CR[CR0]	9-11
subis	RT, RA, IM	Subtract (IM ¹⁶ 0) from (RA 0). Place result in RT. <i>Extended mnemonic for addis RT,RA,-IM</i>		9-12

Table B-4. Extended Mnemonics for PPC401A1 (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
tweqi	RA, IM	Trap if (RA) equal to EXTS(IM). <i>Extended mnemonic for twi 4,RA,IM</i>		
twgei		Trap if (RA) greater than or equal to EXTS(IM). <i>Extended mnemonic for twi 12,RA,IM</i>		
twgti		Trap if (RA) greater than EXTS(IM). <i>Extended mnemonic for twi 8,RA,IM</i>		
twlei		Trap if (RA) less than or equal to EXTS(IM). <i>Extended mnemonic for twi 20,RA,IM</i>		
twlgei		Trap if (RA) logically greater than or equal to EXTS(IM). <i>Extended mnemonic for twi 5,RA,IM</i>		
twlgti		Trap if (RA) logically greater than EXTS(IM). <i>Extended mnemonic for twi 1,RA,IM</i>		
twllei		Trap if (RA) logically less than or equal to EXTS(IM). <i>Extended mnemonic for twi 6,RA,IM</i>		
twllti		Trap if (RA) logically less than EXTS(IM). <i>Extended mnemonic for twi 2,RA,IM</i>		
twlngi		Trap if (RA) logically not greater than EXTS(IM). <i>Extended mnemonic for twi 6,RA,IM</i>		
twlnli		Trap if (RA) logically not less than EXTS(IM). <i>Extended mnemonic for twi 5,RA,IM</i>		
twlti		Trap if (RA) less than EXTS(IM). <i>Extended mnemonic for twi 16,RA,IM</i>		
twnei		Trap if (RA) not equal to EXTS(IM). <i>Extended mnemonic for twi 24,RA,IM</i>		
twngi		Trap if (RA) not greater than EXTS(IM). <i>Extended mnemonic for twi 20,RA,IM</i>		
twnli		Trap if (RA) not less than EXTS(IM). <i>Extended mnemonic for twi 12,RA,IM</i>		

B.5 Storage Reference Instructions

The PPC401A1 uses load and store instructions to transfer data between memory and the general purpose registers. Load and store instructions operate on byte, halfword and word data. The Storage Reference instructions also support loading or storing multiple registers, character strings, and byte-reversed data. Table B-5 shows the Storage Reference instructions available for use in the PPC401A1.

Table B-5. Storage Reference Instructions

Mnemonic	Operands	Function	Other Registers Changed	Page
lbz	RT, D(RA)	Load byte from EA = (RA 0) + EXTS(D) and pad left with zeroes, (RT) \leftarrow $^{24}0 \parallel$ MS(EA,1).		9-78
lbzu	RT, D(RA)	Load byte from EA = (RA 0) + EXTS(D) and pad left with zeroes, (RT) \leftarrow $^{24}0 \parallel$ MS(EA,1). Update the base address, (RA) \leftarrow EA.		9-79
lbzux	RT, RA, RB	Load byte from EA = (RA 0) + (RB) and pad left with zeroes, (RT) \leftarrow $^{24}0 \parallel$ MS(EA,1). Update the base address, (RA) \leftarrow EA.		9-80
lbzx	RT, RA, RB	Load byte from EA = (RA 0) + (RB) and pad left with zeroes, (RT) \leftarrow $^{24}0 \parallel$ MS(EA,1).		9-81
lha	RT, D(RA)	Load halfword from EA = (RA 0) + EXTS(D) and sign extend, (RT) \leftarrow EXTS(MS(EA,2)).		9-82
lhau	RT, D(RA)	Load halfword from EA = (RA 0) + EXTS(D) and sign extend, (RT) \leftarrow EXTS(MS(EA,2)). Update the base address, (RA) \leftarrow EA.		9-83
lhaux	RT, RA, RB	Load halfword from EA = (RA 0) + (RB) and sign extend, (RT) \leftarrow EXTS(MS(EA,2)). Update the base address, (RA) \leftarrow EA.		9-84
lhax	RT, RA, RB	Load halfword from EA = (RA 0) + (RB) and sign extend, (RT) \leftarrow EXTS(MS(EA,2)).		9-85

Table B-5. Storage Reference Instructions (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
lhbrx	RT, RA, RB	Load halfword from EA = (RA 0) + (RB) then reverse byte order and pad left with zeroes, $(RT) \leftarrow {}^{16}0 \parallel MS(EA+1,1) \parallel MS(EA,1)$.		9-86
lhz	RT, D(RA)	Load halfword from EA = (RA 0) + EXTS(D) and pad left with zeroes, $(RT) \leftarrow {}^{16}0 \parallel MS(EA,2)$.		9-87
lhzu	RT, D(RA)	Load halfword from EA = (RA 0) + EXTS(D) and pad left with zeroes, $(RT) \leftarrow {}^{16}0 \parallel MS(EA,2)$. Update the base address, $(RA) \leftarrow EA$.		9-88
lhzux	RT, RA, RB	Load halfword from EA = (RA 0) + (RB) and pad left with zeroes, $(RT) \leftarrow {}^{16}0 \parallel MS(EA,2)$. Update the base address, $(RA) \leftarrow EA$.		9-89
lhzx	RT, RA, RB	Load halfword from EA = (RA 0) + (RB) and pad left with zeroes, $(RT) \leftarrow {}^{16}0 \parallel MS(EA,2)$.		9-90
lmw	RT, D(RA)	Load multiple words starting from EA = (RA 0) + EXTS(D). Place into consecutive registers, RT through GPR(31). RA is not altered unless RA = GPR(31).		9-91
lswi	RT, RA, NB	Load consecutive bytes from EA=(RA 0). Number of bytes n=32 if NB=0, else n=NB. Stack bytes into words in CEIL(n/4) consecutive registers starting with RT, to $R_{FINAL} \leftarrow ((RT + CEIL(n/4) - 1) \% 32)$. GPR(0) is consecutive to GPR(31). RA is not altered unless RA = R _{FINAL} .		9-92
lswx	RT, RA, RB	Load consecutive bytes from EA=(RA 0)+(RB). Number of bytes n=XER[TBC]. Stack bytes into words in CEIL(n/4) consecutive registers starting with RT, to $R_{FINAL} \leftarrow ((RT + CEIL(n/4) - 1) \% 32)$. GPR(0) is consecutive to GPR(31). RA is not altered unless RA = R _{FINAL} . RB is not altered unless RB = R _{FINAL} . If n=0, content of RT is undefined.		9-94

Table B-5. Storage Reference Instructions (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
lwarx	RT, RA, RB	Load word from EA = (RA 0) + (RB) and place in RT, (RT) \leftarrow MS(EA,4). Set the Reservation bit.		9-96
lwbx	RT, RA, RB	Load word from EA = (RA 0) + (RB) then reverse byte order, (RT) \leftarrow MS(EA+3,1) MS(EA+2,1) MS(EA+1,1) MS(EA,1).		9-98
lwz	RT, D(RA)	Load word from EA = (RA 0) + EXTS(D) and place in RT, (RT) \leftarrow MS(EA,4).		9-99
lwzu	RT, D(RA)	Load word from EA = (RA 0) + EXTS(D) and place in RT, (RT) \leftarrow MS(EA,4). Update the base address, (RA) \leftarrow EA.		9-100
lwzux	RT, RA, RB	Load word from EA = (RA 0) + (RB) and place in RT, (RT) \leftarrow MS(EA,4). Update the base address, (RA) \leftarrow EA.		9-101
lwzx	RT, RA, RB	Load word from EA = (RA 0) + (RB) and place in RT, (RT) \leftarrow MS(EA,4).		9-102
stb	RS, D(RA)	Store byte (RS) _{24:31} in memory at EA = (RA 0) + EXTS(D).		9-139
stbu	RS, D(RA)	Store byte (RS) _{24:31} in memory at EA = (RA 0) + EXTS(D). Update the base address, (RA) \leftarrow EA.		9-140
stbux	RS, RA, RB	Store byte (RS) _{24:31} in memory at EA = (RA 0) + (RB). Update the base address, (RA) \leftarrow EA.		9-141
stbx	RS, RA, RB	Store byte (RS) _{24:31} in memory at EA = (RA 0) + (RB).		9-142
sth	RS, D(RA)	Store halfword (RS) _{16:31} in memory at EA = (RA 0) + EXTS(D).		9-143

Table B-5. Storage Reference Instructions (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
sthbrx	RS, RA, RB	Store halfword (RS) _{16:31} byte-reversed in memory at EA = (RA 0) + (RB). MS(EA, 2) \leftarrow (RS) _{24:31} (RS) _{16:23}		9-144
sthu	RS, D(RA)	Store halfword (RS) _{16:31} in memory at EA = (RA 0) + EXTS(D). Update the base address, (RA) \leftarrow EA.		9-145
sthux	RS, RA, RB	Store halfword (RS) _{16:31} in memory at EA = (RA 0) + (RB). Update the base address, (RA) \leftarrow EA.		9-146
sthx	RS, RA, RB	Store halfword (RS) _{16:31} in memory at EA = (RA 0) + (RB).		9-147
stmw	RS, D(RA)	Store consecutive words from RS through GPR(31) in memory starting at EA = (RA 0) + EXTS(D).		9-148
stswi	RS, RA, NB	Store consecutive bytes in memory starting at EA=(RA 0). Number of bytes n=32 if NB=0, else n=NB. Bytes are unstacked from CEIL(n/4) consecutive registers starting with RS. GPR(0) is consecutive to GPR(31).		9-149
stswx	RS, RA, RB	Store consecutive bytes in memory starting at EA=(RA 0)+(RB). Number of bytes n=XER[TBC]. Bytes are unstacked from CEIL(n/4) consecutive registers starting with RS. GPR(0) is consecutive to GPR(31).		9-150
stw	RS, D(RA)	Store word (RS) in memory at EA = (RA 0) + EXTS(D).		9-152
stwbrx	RS, RA, RB	Store word (RS) byte-reversed in memory at EA = (RA 0) + (RB). MS(EA, 4) \leftarrow (RS) _{24:31} (RS) _{16:23} (RS) _{8:15} (RS) _{0:7}		9-153

Table B-5. Storage Reference Instructions (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
stwcx.	RS, RA, RB	Store word (RS) in memory at $EA = (RA 0) + (RB)$ only if reservation bit is set. if RESERVE = 1 then $MS(EA, 4) \leftarrow (RS)$ RESERVE $\leftarrow 0$ $(CR[CR0]) \leftarrow {}^20 \parallel 1 \parallel XER_{so}$ else $(CR[CR0]) \leftarrow {}^20 \parallel 0 \parallel XER_{so}.$		9-154
stwu	RS, D(RA)	Store word (RS) in memory at $EA = (RA 0) + EXTS(D).$ Update the base address, $(RA) \leftarrow EA.$		9-156
stwux	RS, RA, RB	Store word (RS) in memory at $EA = (RA 0) + (RB).$ Update the base address, $(RA) \leftarrow EA.$		9-157
stwx	RS, RA, RB	Store word (RS) in memory at $EA = (RA 0) + (RB).$		9-158

B.6 Arithmetic and Logical Instructions

Table B-6 shows the set of arithmetic and logical instructions supported by the PPC401A1. Arithmetic operations are performed on integer or ordinal operands stored in registers. Instructions using two operands are defined in a three operand format where the operation is performed on the operands stored in two registers and the result is placed in a third register. Instructions using one operand are defined in a two operand format where the operation is performed on the operand in one register and the result is placed in another register. Several instructions also have immediate formats in which one operand is coded as

part of the instruction itself. Most arithmetic and logical instructions can optionally set the condition code register based on the outcome of the instruction.

Table B-6. Arithmetic and Logical Instructions

Mnemonic	Operands	Function	Other Registers Changed	Page
add	RT, RA, RB	Add (RA) to (RB). Place result in RT.		9-6
add.			CR[CR0]	
addo			XER[SO, OV]	
addo.			CR[CR0] XER[SO, OV]	
addc	RT, RA, RB	Add (RA) to (RB). Place result in RT. Place carry-out in XER[CA].		9-7
addc.			CR[CR0]	
addco			XER[SO, OV]	
addco.			CR[CR0] XER[SO, OV]	
adde	RT, RA, RB	Add XER[CA], (RA), (RB). Place result in RT. Place carry-out in XER[CA].		9-8
adde.			CR[CR0]	
addeo			XER[SO, OV]	
addeo.			CR[CR0] XER[SO, OV]	
addi	RT, RA, IM	Add EXTS(IM) to (RA 0). Place result in RT.		9-9
addic	RT, RA, IM	Add EXTS(IM) to (RA 0). Place result in RT. Place carry-out in XER[CA].		9-10
addic.	RT, RA, IM	Add EXTS(IM) to (RA 0). Place result in RT. Place carry-out in XER[CA].	CR[CR0]	9-11
addis	RT, RA, IM	Add (IM ¹⁶ 0) to (RA 0). Place result in RT.		9-12
addme	RT, RA	Add XER[CA], (RA), (-1). Place result in RT. Place carry-out in XER[CA].		9-13
addme.			CR[CR0]	
addmeo			XER[SO, OV]	
addmeo.			CR[CR0] XER[SO, OV]	

Table B-6. Arithmetic and Logical Instructions (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
addze	RT, RA	Add XER[CA] to (RA). Place result in RT. Place carry-out in XER[CA].		9-14
addze.			CR[CR0]	
addzeo			XER[SO, OV]	
addzeo.			CR[CR0] XER[SO, OV]	
and	RA, RS, RB	AND (RS) with (RB). Place result in RA.		9-15
and.			CR[CR0]	
andc	RA, RS, RB	AND (RS) with \neg (RB). Place result in RA.		9-16
andc.			CR[CR0]	
andi.	RA, RS, IM	AND (RS) with ($1^60 \parallel$ IM). Place result in RA.	CR[CR0]	9-17
andis.	RA, RS, IM	AND (RS) with (IM \parallel 1^60). Place result in RA.	CR[CR0]	9-18
cntlzw	RA, RS	Count leading zeros in RS. Place result in RA.		9-41
cntlzw.			CR[CR0]	
divw	RT, RA, RB	Divide (RA) by (RB), signed. Place result in RT.		9-63
divw.			CR[CR0]	
divwo			XER[SO, OV]	
divwo.			CR[CR0] XER[SO, OV]	
divwu	RT, RA, RB	Divide (RA) by (RB), unsigned. Place result in RT.		9-64
divwu.			CR[CR0]	
divwuo			XER[SO, OV]	
divwuo.			CR[CR0] XER[SO, OV]	
eqv	RA, RS, RB	Equivalence of (RS) with (RB). (RA) $\leftarrow \neg((RS) \oplus (RB))$		9-66
eqv.			CR[CR0]	
extsb	RA, RS	Extend the sign of byte (RS) _{24:31} . Place the result in RA.		9-67
extsb.			CR[CR0]	

Table B-6. Arithmetic and Logical Instructions (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
extsh	RA, RS	Extend the sign of halfword (RS) _{16:31} . Place the result in RA.		9-68
extsh.			CR[CR0]	
mulhw	RT, RA, RB	Multiply (RA) and (RB), signed. Place hi-order result in RT. $\text{prod}_{0:63} \leftarrow (RA) \times (RB)$ (signed). (RT) $\leftarrow \text{prod}_{0:31}$.		9-116
mulhw.			CR[CR0]	
mulhwu	RT, RA, RB	Multiply (RA) and (RB), unsigned. Place hi-order result in RT. $\text{prod}_{0:63} \leftarrow (RA) \times (RB)$ (unsigned). (RT) $\leftarrow \text{prod}_{0:31}$.		9-117
mulhwu.			CR[CR0]	
mulli	RT, RA, IM	Multiply (RA) and IM, signed. Place lo-order result in RT. $\text{prod}_{0:47} \leftarrow (RA) \times IM$ (signed) (RT) $\leftarrow \text{prod}_{16:47}$		9-118
mulw	RT, RA, RB	Multiply (RA) and (RB), signed. Place lo-order result in RT. $\text{prod}_{0:63} \leftarrow (RA) \times (RB)$ (signed). (RT) $\leftarrow \text{prod}_{32:63}$.		9-119
mulw.			CR[CR0]	
mulwo			XER[SO, OV]	
mulwo.			CR[CR0] XER[SO, OV]	
nand	RA, RS, RB	NAND (RS) with (RB). Place result in RA.		9-120
nand.			CR[CR0]	
neg	RT, RA	Negative (two's complement) of RA. (RT) $\leftarrow \neg(RA) + 1$		9-121
neg.			CR[CR0]	
nego			XER[SO, OV]	
nego.			CR[CR0] XER[SO, OV]	
nor	RA, RS, RB	NOR (RS) with (RB). Place result in RA.		9-122
nor.			CR[CR0]	
or	RA, RS, RB	OR (RS) with (RB). Place result in RA.		9-123
or.			CR[CR0]	
orc	RA, RS, RB	OR (RS) with $\neg(RB)$. Place result in RA.		9-124
orc.			CR[CR0]	

Table B-6. Arithmetic and Logical Instructions (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
ori	RA, RS, IM	OR (RS) with (¹⁶ 0 IM). Place result in RA.		9-125
oris	RA, RS, IM	OR (RS) with (IM ¹⁶ 0). Place result in RA.		9-126
subf	RT, RA, RB	Subtract (RA) from (RB). (RT) $\leftarrow \neg(RA) + (RB) + 1$.		9-159
subf.			CR[CR0]	
subfo			XER[SO, OV]	
subfo.			CR[CR0] XER[SO, OV]	
subfc	RT, RA, RB	Subtract (RA) from (RB). (RT) $\leftarrow \neg(RA) + (RB) + 1$. Place carry-out in XER[CA].		9-160
subfc.			CR[CR0]	
subfco			XER[SO, OV]	
subfco.			CR[CR0] XER[SO, OV]	
subfe	RT, RA, RB	Subtract (RA) from (RB) with carry-in. (RT) $\leftarrow \neg(RA) + (RB) + XER[CA]$. Place carry-out in XER[CA].		9-162
subfe.			CR[CR0]	
subfeo			XER[SO, OV]	
subfeo.			CR[CR0] XER[SO, OV]	
subfic	RT, RA, IM	Subtract (RA) from EXTS(IM). (RT) $\leftarrow \neg(RA) + EXTS(IM) + 1$. Place carry-out in XER[CA].		9-163
subfme	RT, RA, RB	Subtract (RA) from (−1) with carry-in. (RT) $\leftarrow \neg(RA) + (−1) + XER[CA]$. Place carry-out in XER[CA].		9-164
subfme.			CR[CR0]	
subfmeo			XER[SO, OV]	
subfmeo.			CR[CR0] XER[SO, OV]	
subfze	RT, RA, RB	Subtract (RA) from zero with carry-in. (RT) $\leftarrow \neg(RA) + XER[CA]$. Place carry-out in XER[CA].		9-165
subfze.			CR[CR0]	
subfzeo			XER[SO, OV]	
subfzeo.			CR[CR0] XER[SO, OV]	

Table B-6. Arithmetic and Logical Instructions (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
xor	RA, RS, RB	XOR (RS) with (RB). Place result in RA.		
xor.			CR[CR0]	
xori	RA, RS, IM	XOR (RS) with (¹⁶ 0 IM). Place result in RA.		
xoris	RA, RS, IM	XOR (RS) with (IM ¹⁶ 0). Place result in RA.		

B.7 Condition Register Logical Instructions

Condition Register (CR) logical instructions allow the user to combine the results of several comparisons without incurring the overhead of conditional branching. These instructions can significantly improve code performance if multiple conditions are tested prior to making a branch decision. Table B-7 summarizes the CR logical instructions.

Table B-7. Condition Register Logical Instructions

Mnemonic	Operands	Function	Other Registers Changed	Page
crand	BT, BA, BB	AND bit (CR _{BA}) with (CR _{BB}). Place result in CR _{BT} .		9-42
crandc	BT, BA, BB	AND bit (CR _{BA}) with ¬(CR _{BB}). Place result in CR _{BT} .		9-43
creqv	BT, BA, BB	Equivalence of bit CR _{BA} with CR _{BB} . $CR_{BT} \leftarrow \neg(CR_{BA} \oplus CR_{BB})$		9-44
crnand	BT, BA, BB	NAND bit (CR _{BA}) with (CR _{BB}). Place result in CR _{BT} .		9-45
crnor	BT, BA, BB	NOR bit (CR _{BA}) with (CR _{BB}). Place result in CR _{BT} .		9-46
cror	BT, BA, BB	OR bit (CR _{BA}) with (CR _{BB}). Place result in CR _{BT} .		9-47
crorc	BT, BA, BB	OR bit (CR _{BA}) with ¬(CR _{BB}). Place result in CR _{BT} .		9-48
crxor	BT, BA, BB	XOR bit (CR _{BA}) with (CR _{BB}). Place result in CR _{BT} .		9-49
mcrf	BF, BFA	Move CR field, (CR[CR _n]) ← (CR[CR _m]) where m ← BFA and n ← BF.		9-103

B.8 Branch Instructions

The architecture provides conditional and unconditional branches to any storage location. The conditional branch instructions test condition codes set previously and branch accordingly. Conditional branch instructions may decrement and test the Count Register (CTR) as part of determination of the branch condition and may save the return address in the Link Register (LR). The target address for a branch may be a displacement from the current instruction address (CIA), or may be contained in the LR or CTR, or may be an absolute address.

Table B-8. Branch Instructions

Mnemonic	Operands	Function	Other Registers Changed	Page
b	target	Branch unconditional relative. $LI \leftarrow (target - CIA)_{6:29}$ $NIA \leftarrow CIA + EXTS(LI \parallel ^20)$		9-19
ba		Branch unconditional absolute. $LI \leftarrow target_{6:29}$ $NIA \leftarrow EXTS(LI \parallel ^20)$		
bl		Branch unconditional relative. $LI \leftarrow (target - CIA)_{6:29}$ $NIA \leftarrow CIA + EXTS(LI \parallel ^20)$	$(LR) \leftarrow CIA + 4.$	
bla		Branch unconditional absolute. $LI \leftarrow target_{6:29}$ $NIA \leftarrow EXTS(LI \parallel ^20)$	$(LR) \leftarrow CIA + 4.$	
bc	BO, BI, target	Branch conditional relative. $BD \leftarrow (target - CIA)_{16:29}$ $NIA \leftarrow CIA + EXTS(BD \parallel ^20)$	CTR if $BO_2 = 0.$	9-20
bca		Branch conditional absolute. $BD \leftarrow target_{16:29}$ $NIA \leftarrow EXTS(BD \parallel ^20)$	CTR if $BO_2 = 0.$	
bcl		Branch conditional relative. $BD \leftarrow (target - CIA)_{16:29}$ $NIA \leftarrow CIA + EXTS(BD \parallel ^20)$	CTR if $BO_2 = 0.$ $(LR) \leftarrow CIA + 4.$	
bcla		Branch conditional absolute. $BD \leftarrow target_{16:29}$ $NIA \leftarrow EXTS(BD \parallel ^20)$	CTR if $BO_2 = 0.$ $(LR) \leftarrow CIA + 4.$	
bcctr	BO, BI	Branch conditional to address in CTR. Using (CTR) at exit from instruction, $NIA \leftarrow CTR_{0:29} \parallel ^20.$	CTR if $BO_2 = 0.$	9-28
bcctrl			CTR if $BO_2 = 0.$ $(LR) \leftarrow CIA + 4.$	

Table B-8. Branch Instructions (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
bclr	BO, BI	Branch conditional to address in LR. Using (LR) at entry to instruction, $NIA \leftarrow LR_{0:29} \parallel ^20$.	CTR if $BO_2 = 0$.	9-32
bclrl			CTR if $BO_2 = 0$. (LR) $\leftarrow CIA + 4$.	

B.9 Comparison Instructions

Comparison instructions perform arithmetic and logical comparisons between two operands and set one of the eight condition code register fields based on the outcome of the comparison. Table B-9 shows the comparison instructions supported by the PPC401A1.

Table B-9. Comparison Instructions

Mnemonic	Operands	Function	Other Registers Changed	Page
cmp	BF, 0, RA, RB	Compare (RA) to (RB), signed. Results in CR[CRn], where n = BF.		9-37
cmpi	BF, 0, RA, IM	Compare (RA) to EXTS(IM), signed. Results in CR[CRn], where n = BF.		9-38
cmpl	BF, 0, RA, RB	Compare (RA) to (RB), unsigned. Results in CR[CRn], where n = BF.		9-39
cmpli	BF, 0, RA, IM	Compare (RA) to (¹⁶ 0 IM), unsigned. Results in CR[CRn], where n = BF.		9-40

B.10 Rotate and Shift Instructions

Rotate and shift instructions rotate and shift operands which are stored in the general purpose registers. Rotate instructions can also mask rotated operands. Table B-10 shows the PPC401A1 rotate and shift instructions.

Table B-10. Rotate and Shift Instructions

Mnemonic	Operands	Function	Other Registers Changed	Page
rlwimi	RA, RS, SH, MB, ME	Rotate left word immediate, then insert according to mask. $r \leftarrow \text{ROTL}((RS), SH)$ $m \leftarrow \text{MASK}(MB, ME)$ $(RA) \leftarrow (r \wedge m) \vee ((RA) \wedge \neg m)$		9-129
rlwimi.			CR[CR0]	
rlwinm	RA, RS, SH, MB, ME	Rotate left word immediate, then AND with mask. $r \leftarrow \text{ROTL}((RS), SH)$ $m \leftarrow \text{MASK}(MB, ME)$ $(RA) \leftarrow (r \wedge m)$		9-130
rlwinm.			CR[CR0]	
rlwnm	RA, RS, RB, MB, ME	Rotate left word, then AND with mask. $r \leftarrow \text{ROTL}((RS), (RB)_{27:31})$ $m \leftarrow \text{MASK}(MB, ME)$ $(RA) \leftarrow (r \wedge m)$		9-133
rlwnm.			CR[CR0]	
slw	RA, RS, RB	Shift left (RS) by $(RB)_{27:31}$. $n \leftarrow (RB)_{27:31}$. $r \leftarrow \text{ROTL}((RS), n)$. if $(RB)_{26} = 0$ then $m \leftarrow \text{MASK}(0, 31 - n)$ else $m \leftarrow {}^{32}0$. $(RA) \leftarrow r \wedge m$.		9-135
slw.			CR[CR0]	
sraw	RA, RS, RB	Shift right algebraic (RS) by $(RB)_{27:31}$. $n \leftarrow (RB)_{27:31}$. $r \leftarrow \text{ROTL}((RS), 32 - n)$. if $(RB)_{26} = 0$ then $m \leftarrow \text{MASK}(n, 31)$ else $m \leftarrow {}^{32}0$. $s \leftarrow (RS)_0$. $(RA) \leftarrow (r \wedge m) \vee ({}^{32}s \wedge \neg m)$. $\text{XER}[CA] \leftarrow s \wedge ((r \wedge \neg m) \neq 0)$.		9-136
sraw.			CR[CR0]	
srawi	RA, RS, SH	Shift right algebraic (RS) by SH. $n \leftarrow SH$. $r \leftarrow \text{ROTL}((RS), 32 - n)$. $m \leftarrow \text{MASK}(n, 31)$. $s \leftarrow (RS)_0$. $(RA) \leftarrow (r \wedge m) \vee ({}^{32}s \wedge \neg m)$. $\text{XER}[CA] \leftarrow s \wedge ((r \wedge \neg m) \neq 0)$.		9-137
srawi.			CR[CR0]	

Table B-10. Rotate and Shift Instructions (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
srw	RA, RS, RB	Shift right (RS) by $(RB)_{27:31}$. $n \leftarrow (RB)_{27:31}$. $r \leftarrow \text{ROTL}((RS), 32 - n)$. if $(RB)_{26} = 0$ then $m \leftarrow \text{MASK}(n, 31)$ else $m \leftarrow {}^{32}0$. $(RA) \leftarrow r \wedge m$.		9-138
srw.			CR[CR0]	

B.11 Cache Control Instructions

Cache control instructions allow the user to indirectly control the contents of the data and instruction caches. The user may fill, flush, invalidate and zero blocks (16-byte lines) in the data cache. The user may also invalidate congruence classes in both caches and invalidate individual lines in the instruction cache.

Table B-11. Cache Control Instructions

Mnemonic	Operands	Function	Other Registers Changed	Page
dcba	RA, RB	Speculatively establish the data cache block which contains the effective address $(RA 0) + (RB)$.		9-50
dcbf	RA, RB	Flush (store, then invalidate) the data cache block which contains the effective address $(RA 0) + (RB)$.		9-50
dcbi	RA, RB	Invalidate the data cache block which contains the effective address $(RA 0) + (RB)$.		9-51
dcbst	RA, RB	Store the data cache block which contains the effective address $(RA 0) + (RB)$.		9-52
dcbt	RA, RB	Load the data cache block which contains the effective address $(RA 0) + (RB)$.		9-53
dcbtst	RA, RB	Load the data cache block which contains the effective address $(RA 0) + (RB)$.		9-55
dcbz	RA, RB	Zero the data cache block which contains the effective address $(RA 0) + (RB)$.		9-57
dccci	RA, RB	Invalidate the data cache congruence class associated with the effective address $(RA 0) + (RB)$.		9-59

Table B-11. Cache Control Instructions (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
dcread	RT, RA, RB	Read either tag or data information from the data cache congruence class associated with the effective address (RA 0) + (RB). Place the results in RT.		9-61
icbi	RA, RB	Invalidate the instruction cache block which contains the effective address (RA 0) + (RB).		9-69
icbt	RA, RB	Load the instruction cache block which contains the effective address (RA 0) + (RB).		9-71
iccci	RA, RB	Invalidate instruction cache congruence class associated with the effective address (RA 0) + (RB).		9-73
icread	RA, RB	Read either tag or data information from the instruction cache congruence class associated with the effective address (RA 0) + (RB). Place the results in ICDBDR.		9-74

B.12 Interrupt Control Instructions

The interrupt control instructions allow the user to move data between general purpose registers and the machine state register, return from interrupts and enable or disable maskable external interrupts. Table B-12 shows the Interrupt control instruction set.

Table B-12. Interrupt Control Instructions

Mnemonic	Operands	Function	Other Registers Changed	Page
mfmsr	RT	Move from MSR to RT, (RT) \leftarrow (MSR).		9-107
mtmsr	RS	Move to MSR from RS, (MSR) \leftarrow (RS).		9-113
rfci		Return from critical interrupt (PC) \leftarrow (SRR2). (MSR) \leftarrow (SRR3).		9-127

Table B-12. Interrupt Control Instructions (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
rfi		Return from interrupt. (PC) \leftarrow (SRR0). (MSR) \leftarrow (SRR1).		9-128
wrttee	RS	Write value of RS ₁₆ to the External Enable bit (MSR[EE]).		
wrtteei	E	Write value of E to the External Enable bit (MSR[EE]).		

B.13 Processor Management Instructions

The processor management instructions move data between GPRs and SPRs and DCRs in the PPC401A1; these instructions also provide traps, system calls and synchronization controls.

Table B-13. Processor Management Instructions

Mnemonic	Operands	Function	Other Registers Changed	Page
eieio		Storage synchronization. All loads and stores that precede the eieio instruction complete before any loads and stores that follow the instruction access main storage. Implemented as sync , which is more restrictive.		9-65
isync		Synchronize execution context by flushing the prefetch queue.		9-76
mcrxr	BF	Move XER[0:3] into field CR _n , where $n \leftarrow BF$. CR[CR _n] \leftarrow (XER[SO, OV, CA]). (XER[SO, OV, CA]) \leftarrow ³ 0.		9-104
mfcrr	RT	Move from CR to RT, (RT) \leftarrow (CR).		9-105
mfdcr	RT, DCRN	Move from DCR to RT, (RT) \leftarrow (DCR(DCRN)).		9-106
mfsprr	RT, SPRN	Move from SPR to RT, (RT) \leftarrow (SPR(SPRN)).		9-108

Table B-13. Processor Management Instructions (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
mtcrf	FXM, RS	Move some or all of the contents of RS into CR as specified by FXM field, $\text{mask} \leftarrow {}^4(\text{FXM}_0) \parallel {}^4(\text{FXM}_1) \parallel \dots \parallel {}^4(\text{FXM}_6) \parallel {}^4(\text{FXM}_7)$. $(\text{CR}) \leftarrow ((\text{RS}) \wedge \text{mask}) \vee (\text{CR}) \wedge \neg \text{mask}$.		9-110
mtdcr	DCRN, RS	Move to DCR from RS, $(\text{DCR}(\text{DCRN})) \leftarrow (\text{RS})$.		9-112
mtspr	SPRN, RS	Move to SPR from RS, $(\text{SPR}(\text{SPRN})) \leftarrow (\text{RS})$.		9-114
sc		System call exception is generated. $(\text{SRR1}) \leftarrow (\text{MSR})$ $(\text{SRR0}) \leftarrow (\text{PC})$ $\text{PC} \leftarrow \text{EVPR}_{0:15} \parallel \text{x'0C00'}$ $(\text{MSR}[\text{WE}, \text{PR}, \text{EE}, \text{PE}, \text{DR}, \text{IR}]) \leftarrow 0$ $(\text{MSR}[\text{LE}]) \leftarrow (\text{MSR}[\text{ILE}])$		9-134
sync		Synchronization. All instructions that precede sync complete before any instructions that follow sync begin. When sync completes, all storage accesses initiated prior to sync will have completed.		9-166
tw	TO, RA, RB	Trap exception is generated if, comparing (RA) with (RB), any condition specified by TO is true.		
twi	TO, RA, IM	Trap exception is generated if, comparing (RA) with EXTS(IM), any condition specified by TO is true.		



Code Optimization and Instruction Timings

The code optimization guidelines in Section C.1, “Code Optimization Guidelines,” and the information describing instruction timings in Section C.2, “Instruction Timings,” on p. C-4, can help compiler, system, and application programmers produce high-performance code and determine accurate execution times.

C.1 Code Optimization Guidelines

The following guidelines can help to reduce program execution times.

C.1.1 Condition Register Bits for Boolean Variables

Compilers can use Condition Register (CR) bits to store boolean variables, where 0 and 1 represent False and True values, respectively. This generally improves performance, compared to using General Purpose Registers (GPRs) to store boolean variables. Most common operations on boolean variables can be accomplished using the CR Logical instructions.

C.1.2 CR Logical Instruction for Compound Branches

For example, consider the following psuedocode:

if (Var28 || Var29 || Var30 || Var 31) branch to target

Var28–Var31 are boolean variables, maintained as bits in the CR[CR7] field (CR_{28:31}). The value 1 represents True; 0 represents False.

This could be coded with branches as:

bt	28,target
bt	29,target
bt	30,target
bt	31,target

Generally faster, functionally equivalent code, using CR Logical instructions, follows:

crr	2,28,29
crr	2,2,30
crr	2,2,31
bt	2,target

C.1.3 Floating-Point Emulation

Two ways of handling floating-point emulation are available.

The preferred method is a call interface to subroutines in a floating-point emulation run-time library.

Alternatively, code can use the PowerPC floating point instructions. The PPC401A1, an integer processor, does not recognize these instructions and will take an illegal instruction interrupt. The interrupt handler can be written to determine the instruction opcode and execute appropriate (integer-based) library routines to provide the equivalent function.

Because this method adds interrupt context switching time to the execution time of library routines that would have been called directly by the preferred method, it is not preferred. However, this method supports code that contains PowerPC floating-point instructions.

C.1.4 Instruction Cache Usage

Code can be organized, based on the size and structure of the instruction cache arrays, to minimize cache misses.

In the instruction cache arrays, any two addresses in which address bits $A_{22:27}$ (the index) are the same, but which differ in $A_{0:21}$ (the tag), are called congruent. (This describes a two-way set-associative cache.) $A_{28:31}$ define the 16 bytes in a cache line, the smallest object that can be brought into the cache. Only two congruent lines can be in the cache simultaneously; accessing a third congruent line causes the removal from the cache of one of the two lines previously there, provided that at least one of the lines is unlocked.

Moving new code into the cache arrays occurs at the speed of external memory. Much faster execution is possible when all code is available in the cache. Organizing code to uniformly use $A_{22:27}$ minimizes the use of code with congruent addresses.

C.1.5 Data Cache Usage

Data can be organized, based on the size and structure of the data cache arrays, to minimize cache misses.

In the data cache arrays, any two addresses in which $A_{23:27}$ (the index) are the same, but which differ in $A_{0:22}$ (the tag), are called congruent. (This describes a two-way set-associative cache.) $A_{28:31}$ define the 16 bytes in a cache line, the smallest object that can be brought into the cache. Only two congruent lines can be in the cache simultaneously;

accessing a third congruent line causes the removal from the cache of one of the two lines previously there, provided that at least one of the lines is unlocked.

Moving data into and out of the cache arrays occurs at the speed of external memory. Much faster execution is possible when all data is available in the cache. Organizing data to uniformly use $A_{23:27}$ minimizes the use of data with congruent addresses.

C.1.6 CR Dependencies

For CR-setting Arithmetic, Compare, CR-Logical, and Logical instructions, and the CR-setting **mcrf**, **mcrxr**, and **mtcrf** instructions, put an instruction between the CR-setting instruction and a Branch instruction that uses a bit in the CR field set by the CR-setting instruction.

C.1.7 LR and CTR Dependencies

For Branch instructions that use the contents of the Count Register (CTR) or the Link Register (LR) as a target address:

- If the branch is taken, an instruction between a CTR/LR update and the Branch is sufficient to eliminate the wait state.
- Pre-fetching is aborted when a CTR/LR-updating instruction precedes a branch to CTR/LR that is predicted or known taken. When a predicted taken branch is not taken, stopping pre-fetching slows performance. Putting an instruction between a CTR/LR-updating instruction and a branch that uses the contents of the CTR or the LR as a target address allows pre-fetching to continue.

C.1.8 Branch Prediction

Use the Y-bit in branch instructions to force proper branch prediction when there is a more likely prediction than the standard prediction. See Section 2.6.5, “Branch Prediction,” on p. 2-33, for a more information about branch prediction.

C.1.9 Alignment

- For speed, align all accesses on the appropriate operand-size boundary. For example, load/store word operands should be word-aligned, and so on. Hardware does not trap unaligned accesses; instead, two accesses are performed for a load or store of an unaligned operand that crosses a word boundary. Unaligned accesses that do not cross word boundaries are performed in one access.
- Align branch targets that are unlikely to be hit by “fall-through” code on cache line boundaries (such as the address of functions such as **strcpy**), to minimize the number of unused instructions in cache line fills.

- Avoid branches to the last word in a cache line; the instruction following the branch target is delayed for a clock cycle while the cache performs a least recently-used (LRU) update of the cache line containing the branch target.

C.2 Instruction Timings

The following timing descriptions consider only “first order” effects of cache misses in the instruction cache (I-side) and data cache (D-side).

The timing descriptions *do not* provide complete descriptions of the performance penalty associated with cache misses; the timing descriptions do not consider bus contention between the I-side and the D-side, or the time associated with performing line fills or flushes. Unless specifically stated otherwise, the number of cycles apply to systems having zero-wait memory access.

C.2.1 General Rules

Instructions execute in order.

All instructions, assuming cache hits, execute in one cycle, except:

- Halfword multiply instructions execute in 11 clock cycles; word multiply instructions execute in 19.
- Divide instructions execute in 35 clock cycles.
- Aligned load/store instructions that hit in the cache execute in two clock cycles/word. See Section C.1.9, “Alignment,” on p. C-3, for information on execution timings for unaligned load/stores.
- Branches execute in two, three, or four clock cycles, as described in Section C.2.2.

C.2.2 Branches

Branch instructions are decoded, predicted and resolved, or simply resolved in the decode stage of the instruction pipeline. Branches can be known taken or not taken, or can have address or condition dependencies. Branches having address dependencies are never predicted. The directions of conditional branches having no address dependencies are predicted.

Conditional branches may depend on the results of an instruction that is changing the CR or the CTR.

Branches to the LR or the CTR may depend on an instruction in the execute stage is changing the contents of the LR or CTR.

Address dependencies can be considered as one of the following:

- A **bcctr** instruction that is known taken, or unresolved, that immediately follows a **mtctr** instruction, or a branch known not taken (BKNT) that decrements the CTR

- A **bclr** instruction that is known taken, or unresolved, that immediately follows a **mtlr** instruction, or BKNT that updates the LR

Instruction timings for branch instructions follow:

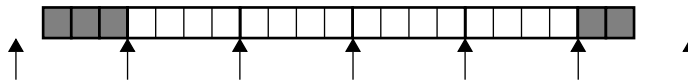
- A BKNT executes in one clock cycle; by definition, a BKNT does not have address or condition dependencies.
- Branches known taken (BKT) by definition have no condition dependencies, but may have address dependencies. A BKT without address dependencies executes in two clock cycles; a BKT having address dependencies executes in three clock cycles.
- Branches predicted not taken (BPNT), which must have condition dependencies, execute in two clock cycles if the prediction is correct; otherwise, four clock cycles are required
- Branches predicted taken (BPT), which must have condition dependencies, execute in two clock cycles, even if the prediction is incorrect.

C.2.3 String Instructions

Calculating execution times for string instructions requires an understanding of data alignment, and of the behavior of the string instructions with respect to alignment.

In the following example, the string contains 21 bytes. The first three bytes do not begin on a word boundary, and the final 2 bytes do not end on a word boundary. The PPC401A1 handles any unaligned leading bytes as a special case, then moves as many bytes as aligned words as possible, and finally handles any unaligned trailing bytes as a special case.

In the following example, arrows indicate word boundaries (the address is an exact multiple of four); shaded boxes represent unaligned bytes.



The execution time of the string instruction is the sum of the:

1. Cycles required to handle unaligned leading bytes; if any, add two clock cycles.

In the example, there are unaligned leading bytes; this transfer adds two clock cycles.

2. Cycles required to handle the number of word-aligned transfers required. Assuming data cache hits, each word-aligned transfer requires two clock cycles.

In the example, there are four aligned words; this transfer requires eight clock cycles.

3. Cycles required to handle unaligned trailing bytes; if any, add two clock cycles.

In the example, there are unaligned trailing bytes; this transfer adds two clock cycles.

A string instruction operating on the example 21-byte string requires twelve clock cycles.

C.2.4 Data Cache Loads and Stores

- Cacheable stores that miss in the D-cache require no extra clock cycles.
- Cacheable loads that miss in the D-cache require four extra clock cycles (three + memory speed).
- Non-cacheable stores require no extra clock cycles.
- Non-cacheable loads require at least three extra clock cycles (two + memory speed).

C.2.5 Instruction Cache Misses

Refer to Section 2.5, “Instruction Processing,” on p. 2-30, for detailed information about the instruction queue and instruction fetching.

In general, for instruction cache misses: when the pre-fetch queue is full, and PFB0 and PFB1 contain non-branching instructions or BKNT branch instructions, and instructions are fetched from cacheable memory, the penalty, in clock cycles, is three + memory speed.

When executing instructions from non-cacheable memory, the penalty, in clock cycles, is two + memory speed.

Index

A

add 9-6
add. 9-6
addc 9-7
addc. 9-7
addco 9-7
addco. 9-7
adde 9-8
adde. 9-8
addeo 9-8
addeo. 9-8
addi 9-9
addic 9-10
addic. 9-11
addis 9-12
addme 9-13
addme. 9-13
addmeo 9-13
addmeo. 9-13
addo 9-6
addo. 9-6
addressing 2-1
addressing modes 1-11
addze 9-14
addze. 9-14
addzeo 9-14
addzeo. 9-14
alignment 2-15
alignment and little endian operation 2-16
alignment error 5-17
alignment exceptions
 summary 2-16
and 9-15
and. 9-15
andc 9-16
andc. 9-16
andi. 9-17
andis. 9-18
architecture, PowerPC 1-2
arithmetic compare 2-10

B

b 9-19
ba 9-19

bc 9-20
bca 9-20
bcctr 9-28
bcctrl 9-28
bcl 9-20
bcla 9-20
bclr 9-32
bclrl 9-32
bctr 9-29
bctrl 9-29
bdnz 9-21
bdnza 9-21
bdnzf 9-21
bdnzfa 9-21
bdnzfl 9-21
bdnzfla 9-21
bdnzflr 9-33
bdnzflrl 9-33
bdnzl 9-21
bdnzla 9-21
bdnzlr 9-33
bdnzlrl 9-33
bdnzt 9-21
bdnzta 9-21
bdnztl 9-21
bdnztla 9-21
bdnztlr 9-33
bdnztlrl 9-33
bdz 9-22
bdza 9-22
bdzf 9-22
bdzfa 9-22
bdzfl 9-22
bdzfla 9-22
bdzflr 9-34
bdzflrl 9-34
bdzl 9-22
bdzla 9-22
bdzlr 9-33
bdzlrl 9-33
bdzt 9-22
bdzta 9-22
bdztl 9-22
bdztla 9-22

bdztlr 9-34
 bdztlrl 9-34
 beq 9-23
 beqa 9-23
 beqctr 9-29
 beqctrl 9-29
 beql 9-23
 beqlr 9-34
 beqlrl 9-34
 bf 9-23
 bfa 9-23
 bfctr 9-29
 bfctrl 9-29
 bfl 9-23
 bfla 9-23
 bflr 9-34
 bflrl 9-34
 B-form A-49
 bge 9-23
 bgea 9-23
 bgectrl 9-29
 bgel 9-23
 bgela 9-23
 bgelr 9-34
 bgelrl 9-34
 bgrctr 9-29
 bgt 9-24
 bgta 9-24
 bgtctr 9-29
 bgtctrl 9-29
 bgtl 9-24
 bgvla 9-24
 bgvlr 9-34
 bgvlrl 9-34
 bl 9-19
 bla 9-19
 ble 9-24
 blea 9-24
 blectr 9-30
 blectrl 9-30
 blel 9-24
 blela 9-24
 blelr 9-35
 blelrl 9-35
 blr 9-33
 blrl 9-33
 blt 9-24

blta 9-24
 bltctr 9-30
 bltctrl 9-30
 bltl 9-24
 blvla 9-24
 blvlr 9-35
 blvlrl 9-35
 bne 9-25
 bnea 9-25
 bnectr 9-30
 bnectrl 9-30
 bnel 9-25
 bnela 9-25
 bnelr 9-35
 bnelrl 9-35
 bng 9-25
 bnga 9-25
 bngctr 9-30
 bngctrl 9-30
 bngl 9-25
 bngla 9-25
 bnglr 9-35
 bnglrl 9-35
 bnl 9-25
 bnla 9-25
 bnctr 9-30
 bnctrl 9-30
 bnll 9-25
 bnlla 9-25
 bnllr 9-35
 bnllrl 9-35
 bns 9-26
 bnsa 9-26
 bnsctr 9-30
 bnsctrl 9-30
 bnsl 9-26
 bnsla 9-26
 bnslr 9-35
 bnsrl 9-35
 bnu 9-26
 bnua 9-26
 bnuctr 9-31
 bnuctrl 9-31
 bnul 9-26
 bnula 9-26
 bnulr 9-36
 bnulrl 9-36

- branch prediction 2-33, A-1, B-4
- branching
 - control 2-30
 - speculative accesses 2-34
- branching control
 - AA field on conditional branches 2-31
 - AA field on unconditional branches 2-31
 - BI field on conditional branches 2-31
 - BO field on conditional branches 2-31
 - branch prediction 2-33
- bso 9-26
- bsoa 9-26
- bsoctr 9-31
- bsoctrl 9-31
- bsol 9-26
- bsola 9-26
- bsolr 9-36
- bsolrl 9-36
- bt 9-27
- bta 9-27
- btctr 9-31
- btctrl 9-31
- btl 9-27
- btla 9-27
- btlr 9-36
- btlrl 9-36
- bun 9-27
- buna 9-27
- bunctr 9-31
- bunctrl 9-31
- bunl 9-27
- bunla 9-27
- bunlr 9-36
- bunlrl 9-36
- byte ordering
 - storage attribute 8-10

C

- cache
 - data
 - cacheability control 6-6
 - coherency 6-7
 - debugging 6-9, 6-11
 - operations 6-5
 - organization 6-4
 - write strategies 6-6
 - debugging 6-9

- instruction
 - cacheability control 6-4, 8-6
 - coherency 6-4
 - debugging 6-9, 6-10
 - operations 6-3
- instructions 6-7
 - DAC debug events 7-9
 - DCU 6-8
 - ICU 6-7
 - line locking 6-12
- cache line locking 6-12
- cacheability
 - instruction 8-6
- CDBCR 6-9, 10-5
- clrlslwi 9-130
- clrlslwi. 9-130
- clrlwi 9-130
- clrlwi. 9-130
- clrrwi 9-131
- clrrwi. 9-131
- cmp 9-37
- cmpi 9-38
- cmpl 9-39
- cmpli 9-40
- cmplw 9-39
- cmplwi 9-40
- cmpw 9-37
- cmpwi 9-38
- cntlzw 9-41
- cntlzw. 9-41
- compare
 - arithmetic 2-10
 - logical 2-10
- condition register 2-9
- condition register (CR) 1-10
- context synchronization 2-40
- CR 10-6
- crand 9-42
- crandc 9-43
- crclr 9-49
- creqv 9-44
- critical exceptions, defined 5-5
- critical interrupt pin 5-13
- crmove 9-47
- crnand 9-45
- crnor 9-46
- crnot 9-46

- cror 9-47
- crorc 9-48
- crset 9-44
- crxor 9-49
- CTR 2-4, 10-7

D

- DAC1 7-8, 10-8
- Data Address Compare Register (DAC1) 7-8
- data alignment 2-15
- data cache
 - cacheability control 6-6
 - coherency 6-7
 - debugging 6-9, 6-11
 - instructions 6-8
 - locking lines 6-12
 - operations 6-5
 - organization 6-4
 - unlocking lines 6-13
 - write strategies 6-6
- data storage exception 5-15
- data types 1-9, 2-15
- DBCR 7-5, 10-9
- DBSR 7-7
- dcba 9-50
- dcbf 6-8, 6-13, 9-50
- dcbi 6-8, 9-51
- dcbst 6-8, 9-52
- dcbt 6-8, 9-53
- dcbtst 6-8, 9-55
- dcbz 6-9, 6-13, 9-57
- dccci 6-9, 9-59
- DCCR 10-13
- dcread 6-9, 9-61
- DCU
 - cacheability control 6-6
 - coherency 6-7
 - debugging 6-11
 - instructions 6-8
 - locking lines 6-12
 - unlocking lines 6-13
 - write strategies 6-6
- DCWR 10-15
- DEAR 5-12, 10-17
- Debug Control Register (DBCR) 7-5
- debug exceptions 5-23
 - branch taken 5-23

- DAC 5-23
- IAC 5-23
- instruction completion 5-23
- non-critical exceptions 5-23
- TRAP 5-23
- unconditional 5-23

- debugging
 - boundary scan chain 7-13
 - debug events 7-4
 - debug interfaces 7-10
 - JTAG test access port 7-11
 - development tools 7-1
 - modes 7-1
 - external 7-2
 - internal 7-1
 - processor control 7-2
 - processor status 7-3
 - registers 7-5
- device control registers 2-14
- device control registers (DCRs) 1-11

- D-form A-49
- divw 9-63
- divw. 9-63
- divwo 9-63
- divwo. 9-63
- divwu 9-64
- divwu. 9-64
- divwuo 9-64
- divwuo. 9-64

E

- eieio 9-65
- eqv 9-66
- eqv. 9-66
- ESR 10-18
- EVPR 5-10, 10-19
- exception-handling registers, general 5-7
- exceptions
 - exceptions
 - alignment exception summary 2-16
 - FIT 5-21
 - handling
 - MSR bits 2-38
 - PIT 5-20
 - registers during alignment error 5-18
 - registers during critical interrupt 5-13
 - registers during debug exceptions 5-23

- registers during external interrupts 5-17
- registers during FIT interrupt 5-21
- registers during machine check 5-15
- registers during PIT interrupt 5-21
- registers during program exceptions 5-19
- registers during system call 5-20
- registers during watchdog interrupt 5-22
- SRR0-SRR1 (non-critical) 5-8
- SRR2-SRR3 (critical) 5-9
- execution synchronization 2-43
- extended memonics
 - beqlr 9-34
- extended menmonics
 - blectrl 9-30
 - bnlctrl 9-30
- extended mnemonicid
 - bngla 9-25
- extended mnemonics 2-50
 - alphabetical B-4
 - bctr 9-29
 - bctrl 9-29
 - bdnz 9-21
 - bdnza 9-21
 - bdnzf 9-21
 - bdnzfa 9-21
 - bdnzfkr 9-33
 - bdnzfl 9-21
 - bdnzfla 9-21
 - bdnzflrl 9-33
 - bdnzl 9-21
 - bdnzla 9-21
 - bdnzlr 9-33
 - bdnzlrl 9-33
 - bdnzt 9-21
 - bdnzta 9-21
 - bdnztl 9-21
 - bdnztla 9-21
 - bdnztlr 9-33
 - bdnztlrl 9-33
 - bdz 9-22
 - bdza 9-22
 - bdzf 9-22
 - bdzfa 9-22
 - bdzfl 9-22
 - bdzfla 9-22
 - bdzflr 9-34
 - bdzflrl 9-34
 - bdzl 9-22
 - bdzla 9-22
 - bdzlr 9-33
 - bdzlr 9-33
 - bdzt 9-22
 - bdzta 9-22
 - bdztl 9-22
 - bdztlr 9-34
 - bdztlr 9-34
 - bdztlrl 9-34
 - beq 9-23
 - beqa 9-23
 - beqctr 9-29
 - beqctrl 9-29
 - beql 9-23
 - beqlrl 9-34
 - bf 9-23
 - bfa 9-23
 - bfctr 9-29
 - bfctrl 9-29
 - bfl 9-23
 - bfla 9-23
 - bflr 9-34
 - bflrl 9-34
 - bge 9-23
 - bgea 9-23
 - bgctr 9-29
 - bgctrl 9-29
 - bge 9-23
 - bgela 9-23
 - bgelr 9-34
 - bgelrl 9-34
 - bgt 9-24
 - bgta 9-24
 - bgctr 9-29
 - bgctrl 9-29
 - bgtl 9-24
 - bgta 9-24
 - bgtlr 9-34
 - bgtlrl 9-34
 - ble 9-24
 - blea 9-24
 - blectr 9-30
 - blel 9-24
 - blela 9-24
 - blelr 9-35
 - blelrl 9-35

blr 9-33
 blrl 9-33
 blt 9-24
 blta 9-24
 bltctr 9-30
 bltctrl 9-30
 btl 9-24
 bltla 9-24
 bltlr 9-35
 bltlrl 9-35
 bne 9-25
 bnea 9-25
 bnectrl 9-30
 bnel 9-25
 bnela 9-25
 bnelr 9-35
 bnelrl 9-35
 bng 9-25
 bnga 9-25
 bngctr 9-30
 bngctrl 9-30
 bngl 9-25
 bnglr 9-35
 bnglrl 9-35
 bnl 9-25
 bnla 9-25
 bnlctr 9-30
 bnll 9-25
 bnlla 9-25
 bnllr 9-35
 bnllrl 9-35
 bns 9-26
 bnsa 9-26
 bnsctr 9-30
 bnsctrl 9-30
 bnsi 9-26
 bnsia 9-26
 bnsir 9-35
 bnsirl 9-35
 bnu 9-26
 bnua 9-26
 bnuctr 9-31
 bnuctrl 9-31
 bnul 9-26
 bnula 9-26
 bnulr 9-36
 bnulrl 9-36

bsalr 9-36
 bso 9-26
 bsoa 9-26
 bsoctr 9-31
 bsoctrl 9-31
 bsol 9-26
 bsola 9-26
 bsolrl 9-36
 bt 9-27
 bta 9-27
 btctr 9-31
 btctrl 9-31
 btl 9-27
 btla 9-27
 btlr 9-36
 btlrl 9-36
 bun 9-27
 buna 9-27
 bunctr 9-31
 bunctrl 9-31
 bunl 9-27
 bunla 9-27
 bunlr 9-36
 bunlrl 9-36
 clrlslwi 9-130
 clrlslwi. 9-130
 clrlwi 9-130
 clrlwi. 9-130
 clrrwi 9-131
 clrrwi. 9-131
 cmplw 9-39
 cmplwi 9-40
 cmpw 9-37
 cmpwi 9-38
 crclr 9-49
 crmove 9-47
 crnot 9-46
 crset 9-44
 extlwi 9-131
 extlwi. 9-131
 extrwi 9-131
 extrwi. 9-131
 for addi 9-9
 for addic 9-10
 for addic. 9-11
 for addis 9-12
 for bc, bca, bcl, bcla 9-21

for bcctr, bcctrl 9-29
 for bclr, bclrl 9-33
 for cmp 9-37
 for cmpi 9-38
 for cmpl 9-39
 for cmpli 9-40
 for creqv 9-44
 for crnor 9-46
 for cror 9-47
 for crxor 9-49
 for mfspr 9-109
 for mtcrrf 9-111
 for mtspr 9-115
 for nor, nor. 9-122
 for or, or. 9-123
 for ori 9-125
 for rlwimi, rlwimi. 9-129
 for rlwinm, rlwinm. 9-130
 for rlwnm, rlwnm. 9-133
 for subf, subf., subfo, subfo. 9-159
 for subfc, subfc., subfco, subfco. 9-161
 for tw 9-168
 for twi 9-172
 inslwi 9-129
 inslwi. 9-129
 insrwi 9-129
 insrwi. 9-129
 la 9-9
 li 9-9
 lis 9-12
 mfcdbcn 9-109
 mfctr 9-109
 mfdac 9-109
 mfdbcl 9-109
 mfdbsr 9-109
 mfdbsrs 9-109
 mfdccr 9-109
 mfdcwr 9-109
 mfdear 9-109
 mfesr 9-109
 mfevpr 9-109
 mflac1 9-109
 mflccr 9-109
 mflcldbdr 9-109
 mflr 9-109
 mfplt 9-109
 mfpvr 9-109

mfsgr 9-109
 mfsler 9-109
 mfsprg0 9-109
 mfsprg1 9-109
 mfsprg2 9-109
 mfsprg3 9-109
 mftcr 9-109
 mftsr 9-109
 mfxer 9-109
 mptvr 9-115
 mr 9-123
 mr. 9-123
 mtcdbcr 9-115
 mtcrr 9-111
 mtctr 9-115
 mtdac1 9-115
 mtdbcr 9-115
 mtdbsr 9-115
 mtdccr 9-115
 mtdcwr 9-115
 mtesr 9-115
 mtevpr 9-115
 mtiac1 9-115
 mticcr 9-115
 mticdbdr 9-115
 mtlr 9-115
 mtpit 9-115
 mtsear 9-115
 mtsgr 9-115
 mtsler 9-115
 mtsprg1 9-115
 mtsprg2 9-115
 mtsprg3 9-115
 mtsrr0 9-115
 mtsrr1 9-115
 mtsrr2 9-115
 mtsrr3 9-115
 mttcr 9-115
 mttspg0 9-115
 mttsr 9-115
 mttxr 9-115
 nop 9-125
 not 9-122
 not. 9-122
 rotlw 9-133
 rotlw. 9-133
 rotlwi 9-131

- rotlwi. 9-131
- rotrwi 9-131
- rotrwi. 9-131
- slwi 9-132
- slwi. 9-132
- srwi 9-132
- srwi. 9-132
- sub 9-159
- sub. 9-159
- subc 9-161
- subc. 9-161
- subco 9-161
- subco. 9-161
- subi 9-9
- subic 9-10
- subic. 9-11
- subis 9-12
- subo 9-159
- subo. 9-159
- trap 9-168
- tweq 9-169
- tweqi 9-172
- twge 9-169
- twgei 9-172
- twgle 9-169
- twgt 9-169
- twgti 9-173
- twle 9-169
- twlei 9-173
- twlgei 9-173
- twlgt 9-169
- twlgti 9-173
- twlle 9-169
- twllei 9-173
- twllt 9-169
- twllti 9-173
- twlng 9-169
- twlngi 9-173
- twlnl 9-169
- twlnli 9-173
- twlt 9-169
- twlti 9-173
- twne 9-170
- twnei 9-173
- twng 9-170
- twngi 9-174
- twnl 9-170

- twnli 9-174
- external interrupts 5-16
 - DMA 5-16
 - external interrupt pins 5-16
 - JTAG port 5-16
 - serial port 5-16
- extlwi 9-131
- extlwi. 9-131
- extrwi 9-131
- extrwi. 9-131
- extsb 9-67
- extsb. 9-67
- extsh 9-68
- extsh. 9-68

F

- FIT 5-21, 5-29
- fixed interval timer 5-21, 5-29

G

- general exception-handling registers 5-7
- general purpose registers (GPRs) 1-10
- GPR0-GPR31 2-2, 10-20
- guarded storage
 - architectural overview 2-35
 - SGR 8-8
 - speculative accesses to 2-35

I

- IAC1 7-10, 10-21
- icbi 6-7, 6-14, 9-69
- icbt 6-7, 9-71
- iccci 6-8, 9-73
- ICCR 8-6, 10-22
- ICDBDR 6-10, 10-24
- icread 6-8, 9-74
- ICU
 - cacheability control 6-4
 - coherency 6-4
 - debugging 6-10
 - instructions 6-7
 - locking lines 6-12
 - operations 6-3
 - unlocking lines 6-13
- I-form A-49
- initialization 4-2
 - code example 4-4

requirements 4-3
 inslwi 9-129
 inslwi. 9-129
 insrwi 9-129
 insrwi. 9-129
 instruction
 add 9-6
 add. 9-6
 addc 9-7
 addc. 9-7
 addco 9-7
 addco. 9-7
 adde 9-8
 adde. 9-8
 addeo 9-8
 addeo. 9-8
 addi 9-9
 addic 9-10
 addic. 9-11
 addis 9-12
 addme 9-13
 addme. 9-13
 addmeo 9-13
 addmeo. 9-13
 addo 9-6
 addo. 9-6
 addze 9-14
 addze. 9-14
 addzeo 9-14
 addzeo. 9-14
 and 9-15
 and. 9-15
 andc 9-16
 andc. 9-16
 andi. 9-17
 andis. 9-18
 b 9-19
 ba 9-19
 bc 9-20
 bca 9-20
 bcctr 9-28
 bcctrl 9-28
 bcl 9-20
 bcla 9-20
 bclr 9-32
 bclrl 9-32
 bl 9-19

bla 9-19
 cmp 9-37
 cmpi 9-38
 cmpl 9-39
 cmpli 9-40
 cntlzw 9-41
 cntlzw. 9-41
 crand 9-42
 crandc 9-43
 creqv 9-44
 crnand 9-45
 crnor 9-46
 cror 9-47
 crorc 9-48
 crxor 9-49
 dcba 9-50
 dcbf 9-50
 dcbi 9-51
 dcbst 9-52
 dcbt 9-53
 dcbtst 9-55
 dcbz 9-57
 dcccci 9-59
 dcread 9-61
 divw 9-63
 divw. 9-63
 divwo 9-63
 divwo. 9-63
 divwu 9-64
 divwu. 9-64
 divwuo 9-64
 divwuo. 9-64
 eieio 9-65
 eqv 9-66
 eqv. 9-66
 extsb 9-67
 extsb. 9-67
 extsh 9-68
 extsh. 9-68
 formats
 B-form A-49
 D-form A-49
 I-form A-49
 SC-form A-49
 X-form A-50
 XFX-form A-51
 XL-form A-51

XO-form A-51

icbi 9-69
icbt 9-71
iccci 9-73
icread 9-74
isync 9-76
lbz 9-78
lbzu 9-79
lbzux 9-80
lbzx 9-81
lha 9-82
lhau 9-83
lhaux 9-84
lhax 9-85
lhrx 9-86
lhz 9-87
lhzu 9-88
lhzux 9-89
lhzx 9-90
lmw 9-91
lswi 9-92
lswx 9-94
lwarx 9-96
lwbrx 9-98
lwz 9-99
lwzu 9-100
lwzux 9-101
lwzx 9-102
mcrf 9-103
mcrxr 9-104
mfcr 9-105
mfdcr 9-106
mfmsr 9-107
mfspr 9-108
mftb 5-27, 9-110
mtcrf 9-110
mtdcr 9-112
mtspr 9-114
mulhw 9-116
mulhw. 9-116
mulhwu 9-117
mulhwu. 9-117
mulli 9-118
mullw 9-119
mullw. 9-119
mullwo 9-119
mullwo. 9-119

nand 9-120
nand. 9-120
neg 9-121
neg. 9-121
nego 9-121
nego. 9-121
nor 9-122
nor. 9-122
or 9-123
or. 9-123
orc 9-124
orc. 9-124
ori 9-125
oris 9-126
rfci 9-127
rfi 9-128
rlwimi 9-129
rlwimi. 9-129
rlwinm 9-130
rlwinm. 9-130
rlwnm 9-133
rlwnm. 9-133
sc 9-134
slw 9-135
slw. 9-135
sraw 9-136
sraw. 9-136
srawi 9-137
srawi. 9-137
srw 9-138
srw. 9-138
stb 9-139
stbu 9-140
stbux 9-141
stbx 9-142
sth 9-143
sthbrx 9-144
sthu 9-145
sthux 9-146
sthx 9-147
stmw 9-148
stswi 9-149
stswx 9-150
stw 9-152
stwbrx 9-153
stwcx. 9-154
stwu 9-156

- stwx 9-157
- subf 9-159
- subf. 9-159
- subfc 9-160
- subfc. 9-160
- subfco 9-160
- subfco. 9-160
- subfe 9-162
- subfe. 9-162
- subfeo 9-162
- subfeo. 9-162
- subfic 9-163
- subfme 9-164
- subfme. 9-164
- subfmeo 9-164
- subfmeo. 9-164
- subfo 9-159
- subfo. 9-159
- subfze 9-165
- subfze. 9-165
- subfzeo 9-165
- subfzeo. 9-165
- sync 9-166
- tw 9-167
- twi 9-171
- wrtee 9-175
- wrteei 9-176
- xor 9-177
- xori 9-178
- instruction cache
 - cacheability control 6-4
 - coherency 6-4
 - debugging 6-9, 6-10
 - instructions 6-7
 - locking lines 6-12
 - operations 6-3
 - unlocking lines 6-13
- instruction fields A-46
- instruction formats 9-2, A-46
 - B-form A-49
 - D-form A-49
 - diagrams A-48
 - I-Form A-49
 - M-form A-51
 - SC-form A-49
 - X-form A-50
 - AFX-form A-51
 - XL-form A-51
 - XO-form A-51
- instruction forms A-46, A-48
- instruction queue 2-30
- instruction set
 - brief summaries by category 2-44
- instruction set portability 9-1
- instruction set summary
 - arithmetic and logical 2-46
 - branch 2-47
 - cache control 2-49
 - compare 2-47
 - CR logical 2-48
 - interrupt control 2-49
 - processor management 2-50
 - rotate and shift 2-48
- instruction timings C-4
 - branches and cr logicals C-4
 - general rules C-4
 - instruction cache misses C-6
 - loads and stores C-6
 - strings C-5
- instructions
 - alignment exceptions, causing 2-16
 - alphabetical, including extended mnemonics A-1
 - arithmetic and logical B-30
 - branch B-36
 - cache 6-7
 - DAC debug events 7-9
 - DCU 6-8
 - ICU 6-7
 - cache control B-39
 - alignment 2-15
 - categories B-1
 - comparison B-37
 - condition register logical B-35
 - extended mnemonics B-4
 - format diagrams A-48
 - formats A-46
 - forms A-46, A-48
 - interrupt control B-40
 - opcodes A-38
 - privileged 2-39, B-3
 - processor management B-41
 - rotate and shift B-38

- specific to PowerPC Embedded Controllers
 - B-1
- storage reference B-26
 - alignment 2-15
- interfaces
 - interrupt controller 1-9
- interrupt controller interface 1-9
- interrupts and exceptions 5-1, 5-3
 - alignment error 5-17
 - architectural definitions and behavior 5-2
 - asynchronous, defined 5-2
 - critical 5-5, 5-6
 - critical interrupt pin 5-13
 - data machine check 5-15
 - data storage 5-15
 - debug 5-23
 - exception, defined 5-1
 - external interrupts 5-16
 - fixed interval timer 5-21
 - imprecise, defined 5-2
 - instruction machine check 5-3
 - interrupt, defined 5-1
 - machine check, defined 5-2
 - machine check—instruction 5-14
 - non-critical 5-5
 - precise, defined 5-2
 - program 5-18
 - programmable interval timer 5-20
 - synchronous, defined 5-2
 - system call 5-20
 - watchdog timer 5-22
- isync 9-76

L

- la 9-9
- lbz 9-78
- lbzu 9-79
- lbzux 9-80
- lbzx 9-81
- lha 9-82
- lhau 9-83
- lhaux 9-84
- lhax 9-85
- lhbz 9-86
- lhz 9-87
- lhzu 9-88
- lhzux 9-89

- lhzx 9-90
- li 9-9
- line locking, cache 6-12
- lis 9-12
- little endian operation and alignment 2-16
- lmw 9-91
- logical compare 2-10
- LR 2-5, 10-25
- lswi 9-92
- lswx 9-94
- lwarx 9-96
- lwbrx 9-98
- lwz 9-99
- lwzu 9-100
- lwzux 9-101
- lwzx 9-102

M

- machine state register (MSR) 1-10
- mcrf 9-103
- mcrxr 9-104
- memory management 1-7
- memory map 2-1
 - storage attributes 2-1, 8-1
- memory organization 2-1
- mfcdbcn 9-109
- mfcr 9-105
- mfctr 9-109
- mfdac 9-109
- mfdtbl 9-109
- mfdbsr 9-109
- mfdbsrs 9-109
- mfdccr 9-109
- mfdcr 9-106
- mfdcwr 9-109
- mfdear 9-109
- mfesr 9-109
- mfevpr 9-109
- mflac1 9-109
- mflccr 9-109
- mflcldbdr 9-109
- mflr 9-109
- mfmsr 9-107
- M-form A-51
- mfplt 9-109
- mfpvr 9-109
- mfsgr 9-109

- mfsler 9-109
- mfspr 9-108
- mfsprg0 9-109
- mfsprg1 9-109
- mfsprg2 9-109
- mfsprg3 9-109
- mftb 5-27, 9-110
- mftcr 9-109
- mftsr 9-109
- mfixer 9-109
- mptvr 9-115
- mr 9-123
- mr. 9-123
- MSR 2-13, 5-7, 10-26
- MSR bits and exception handling 2-38
- mtcdbc 9-115
- mtcr 9-111
- mtcrf 9-110
- mtctr 9-115
- mtdac1 9-115
- mtdbc 9-115
- mtdbsr 9-115
- mtdccr 9-115
- mtdcr 9-112
- mtdcwr 9-115
- mtdear 9-115
- mtesr 9-115
- mtevpr 9-115
- mtiac1 9-115
- mticcr 9-115
- mticdbdr 9-115
- mtlr 9-115
- mtpit 9-115
- mtsgr 9-115
- mtsler 9-115
- mtspr 9-114
- mtsprg1 9-115
- mtsprg2 9-115
- mtsprg3 9-115
- mtsrr0 9-115
- mtsrr1 9-115
- mtsrr2 9-115
- mtsrr3 9-115
- mttcr 9-115
- mttsprg0 9-115
- mttsr 9-115
- mttxer 9-115

- mulhw 9-116
- mulhw. 9-116
- mulhwu 9-117
- mulhwu. 9-117
- mulli 9-118
- mulw 9-119
- mulw. 9-119
- mullwo 9-119
- mullwo. 9-119

N

- nand 9-120
- nand. 9-120
- neg 9-121
- neg. 9-121
- nego 9-121
- nego. 9-121
- non-critical exceptions, defined 5-5
- nop 9-125
- nor 9-122
- nor. 9-122
- not 9-122
- not. 9-122
- notation xxiv, 9-2, A-46

O

- opcodes A-38
- optimization
 - coding guidelines C-1
 - alignment C-3
 - boolean variables C-1
 - branch prediction C-3
 - data cache usage C-2
 - dependency upon CR C-3
 - dependency upon LR and CTR C-3
 - floating point emulation C-2
 - instruction cache usage C-2
- or 9-123
- or. 9-123
- orc 9-124
- orc. 9-124
- ori 9-125
- oris 9-126

P

- PIT 5-20, 5-28, 10-28
- portability, instruction set 9-1
- PowerPC architecture 1-2

PowerPC Endian mode and alignment 2-16

pre-fetch

branches to Count Register 2-35

branches to Link Register 2-35

queue 2-30

primary opcodes A-38

privileged DCRs 2-40

privileged instructions 2-39

privileged mode 2-38

privileged operation 2-38

privileged SPRs 2-39

problem state 2-38

program exception 5-18

programmable interval timer 5-20, 5-28

pseudocode 9-2

PVR 2-8, 10-29

Q

queue 2-30

R

R0-R31 2-2, 10-20

register set summary 1-10

registers

CDBCR 6-9, 10-5

CR 1-10, 10-2, 10-6

CTR 2-4, 10-7

DAC1 7-8, 10-8

DBCR 7-5, 10-9

DBSR 7-7

DCCR 10-13

DCR numbering 10-4

DCWR 10-15

DEAR 5-12, 10-17

device control 1-11

during alignment error 5-18

during critical interrupt 5-13

during debug exceptions 5-23

during external interrupts 5-17

during FIT interrupt 5-21

during machine check 5-15

during PIT interrupt 5-21

during program exceptions 5-19

during system call 5-20

during watchdog interrupt 5-22

ESR 10-18

EVPR 5-10, 10-19

general purpose 1-10

GPR 10-2

GPR0-GPR31 2-2, 10-20

IAC1 7-10, 10-21

ICCR 10-22

ICDBDR 6-10, 10-24

LR 2-5, 10-25

MSR 1-10, 2-13, 5-7, 10-2, 10-26

PIT 5-28, 10-28

PVR 2-8, 10-29

R0-R31 2-2, 10-20

reserved 10-1

reserved fields 10-1

SGR 10-30

SLER 10-32

special purpose 1-10

SPR numbering 10-2

SPRG0-SPRG3 2-8, 10-34

SRR0 5-8, 10-35

SRR0-SRR1 (non-critical) 5-8

SRR1 5-8, 10-36

SRR2 5-9, 10-37

SRR2-SRR3 (critical) 5-9

SRR3 5-9, 10-38

summary 1-10

TBHI 10-39

TBHU 10-40

TBLO 10-41

TBLU 10-42

TCR 5-29, 5-30, 5-33, 10-43

TSR 5-30, 5-32, 10-44

XER 2-6, 10-45

registers, device control 2-14

registers, special purpose 2-3

registers, summary 2-2

reservation bit 9-96, 9-154

reserved fields 10-1

reserved registers 10-1

reset

processor initialization 4-2

processor state after 4-1

rfci 9-127

rfi 9-128

rlwimi 9-129

rlwimi. 9-129

rlwinm 9-130

rlwinm. 9-130

- rlwnm 9-133
- rlwnm. 9-133
- rotlw 9-133
- rotlw. 9-133
- rotlwi 9-131
- rotlwi. 9-131
- rotrwi 9-131
- rotrwi. 9-131
- extended mnemonics
 - bnctr 9-30

S

- sc 9-134
- SC-form A-49
- secondary opcodes A-38
- SGR 8-8, 10-30
- SLER 8-10, 10-32
- slw 9-135
- slw. 9-135
- slwi 9-132
- slwi. 9-132
- special purpose registers 2-3
- special purpose registers (SPRs) 1-10
- speculative accesses
 - 2-34
- speculative fetching 2-34
 - guarded storage 2-35
 - on the 401Core 2-34
 - on the 401GF 2-34
- SPRG0-SPRG3 2-8, 10-34
- sraw 9-136
- sraw. 9-136
- srawi 9-137
- srawi. 9-137
- SRR0 5-8, 10-35
- SRR1 5-8, 10-36
- SRR2 5-9, 10-37
- SRR3 5-9, 10-38
- srw 9-138
- srw. 9-138
- srwi 9-132
- srwi. 9-132
- stb 9-139
- stbu 9-140
- stbux 9-141
- stbx 9-142
- sth 9-143

- sthbrx 9-144
- sthu 9-145
- sthux 9-146
- sthx 9-147
- stmw 9-148
- storage attribute regions 2-1, 8-1
- storage attributes
 - byte ordering 8-10
 - control registers 8-1
 - guarded storage 8-8
- storage synchronization 2-44
- stswi 9-149
- stswx 9-150
- stw 9-152
- stwbrx 9-153
- stwcx. 9-154
- stwu 9-156
- stwux 9-157
- stwx 9-158
- sub 9-159
- sub. 9-159
- subc 9-161
- subc. 9-161
- subco 9-161
- subco. 9-161
- subf 9-159
- subf. 9-159
- subfc 9-160
- subfc. 9-160
- subfco 9-160
- subfco. 9-160
- subfe 9-162
- subfe. 9-162
- subfeo 9-162
- subfeo. 9-162
- subfic 9-163
- subfme 9-164
- subfme. 9-164
- subfmeo 9-164
- subfmeo. 9-164
- subfo 9-159
- subfo. 9-159
- subfze 9-165
- subfze. 9-165
- subfzeo 9-165
- subfzeo. 9-165
- subi 9-9

- subic 9-10
- subic. 9-11
- subis 9-12
- subo 9-159
- subo. 9-159
- supervisor state 2-38
- sync 9-166
- synchronization
 - architectural references 2-40
 - context 2-40
 - execution 2-43
 - storage 2-44
- system call 5-20

T

- TBHI 10-39
- TBHU 10-40
- TBLO 10-41
- TBLU 10-42
- TCR 5-33, 10-43
- time base 5-25
- timer facilities
 - overview 5-24
- timers
 - FIT 5-29
 - fixed interval timer 5-29
 - PIT 5-28
 - programmable interval timer 5-28
 - TCR 5-33
 - time base
 - comparison with PowerPC 5-26
 - timer control register 5-33
 - timer status register 5-32
 - TSR 5-32
 - watchdog 5-30
- timings
 - instruction C-4
 - branches and cr logicals C-4
 - general rules C-4
 - instruction cache misses C-6
 - loads and stores C-6
 - strings C-5
- trap 9-168
- true Little Endian
 - storage attribute control 8-10
- TSR 5-32, 10-44
- tw 9-167

- tw eq 9-169
- tw eqi 9-172
- tw ge 9-169
- tw gei 9-172
- tw gle 9-169
- tw gt 9-169
- tw gti 9-173
- twi 9-171
- tw le 9-169
- tw lei 9-173
- tw lgei 9-173
- tw lgt 9-169
- tw lgti 9-173
- tw lle 9-169
- tw llei 9-173
- tw llt 9-169
- tw llti 9-173
- tw lng 9-169
- tw lngi 9-173
- tw lnl 9-169
- tw lnli 9-173
- tw lt 9-169
- tw lti 9-173
- tw ne 9-170
- tw nei 9-173
- tw ng 9-170
- tw ngi 9-174
- tw nl 9-170
- tw nli 9-174
- types 2-15

U

- user mode 2-38

W

- watchdog timer 5-22, 5-30
- wr tee 9-175
- wr teei 9-176

X

- XER 2-6, 10-45
- X-form A-50
- XFX-form A-51
- XL-form A-51
- XO-form A-51
- xor 9-177
- xori 9-178