



PPC401GF

**Embedded Controller
User's Manual**

13H6948 00002

First Edition (October 1997)

This edition of *IBM PC401GF Embedded Controller User's Manual* applies to the IBM PPC401GF 32-bit embedded controller, 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
Overview	1-1
PPC401GF Overview	1-1
PowerPC Architecture	1-2
The PPC401GF as a PowerPC Implementation	1-3
PPC401GF Features	1-3
RISC Core	1-4
Execution Unit (EXU)	1-4
Storage Control	1-5
Instruction Cache Unit (ICU)	1-6
Data Cache Unit (DCU)	1-6
External Memory and Peripheral Interfaces	1-6
Interrupt Controller Interface	1-7
Debug Support	1-7
Data Types	1-7
Register Set Summary	1-8
General Purpose Registers	1-8
Special Purpose Registers	1-8
Machine State Register	1-8
Condition Register	1-8
Device Control Registers	1-8
Addressing Modes	1-8
Clock Generation and Power Management	1-9
Programming Model	2-1
Memory Organization and Addressing	2-1
Physical Address Map	2-3
Storage Attributes	2-3
Mapping Multiple Storage Attributes to Common Memory Locations	2-4
Registers	2-4
General Purpose Registers	2-4
Special Purpose Registers	2-5
Count Register (CTR)	2-6
Link Register (LR)	2-7
Fixed Point Exception Register (XER)	2-8
Special Purpose Register General (SPRG0-SPRG3)	2-10
Processor Version Register (PVR)	2-10
Condition Register (CR)	2-11
CR Fields after Compare Instructions	2-12
The CR0 Field	2-13
The Time Base	2-14
Machine State Register	2-15
Device Control Registers	2-16

Data Types and Alignment	2-17
Alignment for Storage Reference and Cache Control Instructions	2-17
Alignment and Endian Operation	2-18
Summary of Instructions Causing Alignment Exceptions	2-18
Byte Ordering	2-19
Structure Mapping Examples	2-20
Big-Endian Mapping	2-20
Little Endian Mapping	2-21
PowerPC Byte Ordering	2-21
PowerPC Endian Mode	2-21
Byte Ordering in PowerPC Little Endian Mode	2-22
Control of PowerPC Endian Mode	2-24
Addressing in PowerPC Little Endian Mode	2-25
Little Endian Mode Alignment Requirements	2-25
Switching Endian Modes	2-26
Direct Memory Access in PowerPC Little Endian Mode	2-26
Endian Storage Attribute	2-27
Fetching Instructions from Little Endian Storage Regions	2-27
Accessing Data in Little Endian Storage Regions	2-28
Control of the Endian Storage Attribute	2-29
PowerPC Byte-Reverse Instructions	2-29
Instruction Processing	2-32
Branching Control	2-32
AA Field on Unconditional Branches	2-33
AA Field on Conditional Branches	2-33
BI Field on Conditional Branches	2-33
BO Field on Conditional Branches	2-33
Branch Prediction	2-35
Speculative Accesses	2-36
Speculative Accesses in the PPC401GF	2-36
Pre-fetch Distance Down an Unresolved Branch Path	2-36
Pre-fetch of Branches to Count Register and Branches to Link Register	2-37
Preventing Inappropriate Speculative Accesses	2-37
Fetching Past an Interrupt-causing or Interrupt-returning Instruction	2-38
Fetching Past tw or twi Instructions	2-38
Fetching Past an Unconditional Branch	2-39
Suggested Locations of Memory-Mapped Hardware	2-39
Summary	2-39
Privileged Mode Operation	2-40
MSR Bits and Exception Handling	2-40
Privileged Instructions	2-41
Privileged SPRs	2-41
Privileged DCRs	2-42
Synchronization	2-42
Context Synchronization	2-42
Execution Synchronization	2-45
Storage Synchronization	2-46

Instruction Set	2-46
Instructions Specific to IBM PowerPC Embedded Controllers	2-47
Storage Reference Instructions	2-48
Arithmetic and Logical Instructions	2-48
Compare Instructions	2-49
Branch Instructions	2-49
Condition Register Logical Instructions	2-50
Rotate and Shift Instructions	2-50
Cache Control Instructions	2-51
Interrupt Control Instructions	2-51
Processor Management Instructions	2-52
Extended Mnemonics	2-52
Memory and Peripheral Interface	3-1
Terminology	3-3
Bus States	3-3
Idle State	3-3
Address State	3-3
Wait/Data State	3-4
Recovery State	3-4
Hold State	3-5
Signals	3-5
ABus28:29 (Address Bits 28:29)	3-5
ALE (Address Latch Enable)	3-5
B0:31 (Multiplexed Address Bus Bits 0:31)	3-6
BE0[A31]:BE3 (Byte Enables)	3-9
BLast (Burst Last)	3-9
BootClkSpd (Boot Clock Speed)	3-9
BootW (Boot ROM Width Select)	3-9
BusError (Bus Error Input)	3-10
BusReq (Bus Request)	3-10
BusWidth0:1 (Bus Width)	3-10
HoldAck (Hold Acknowledge)	3-10
HoldReq (Hold Request)	3-10
Ready	3-10
Reset	3-11
W/R (Write/Read)	3-11
External Memory Locations	3-11
Bus Width after Reset	3-14
Access Priorities	3-14
Controlling the Bus Speed	3-15
Connecting to the PPC401GF Bus	3-15
Bus Operations and Timings	3-17
Read and Write Transfers	3-18
Timings	3-19
Data Alignment and Bus Transfers	3-28
External Bus Master Interface	3-29
Shared External Bus Protocol	3-29

External Bus Master Interface Timings	3-30
Reset and Initialization	4-1
Core, Chip, and System Resets	4-1
Reset Sequence	4-1
Processor State After Reset	4-2
Register Contents After A Reset	4-2
Bus Control Unit Behavior During Reset	4-5
PPC401GF Initial Processor Sequencing	4-6
Initialization Requirements	4-6
BRCR Initialization	4-7
Initialization Code Example	4-7
Exceptions, Interrupts, and Timers	5-1
Interrupts and Exceptions	5-2
Architectural Definitions and Behavior	5-2
Behavior of the PPC401GF 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)	5-11
Data Exception Address Register (DEAR)	5-13
Critical Interrupt Exception	5-13
Machine Check Exceptions	5-14
Bus Error Status Register 0 (BESR0)	5-15
Bus Error Address Register (BEAR)	5-16
Instruction Machine Check Handling	5-17
Data Machine Check Handling	5-18
Data Storage Exceptions	5-19
External Interrupt Exception	5-20
Input/Output Configuration Register (IOCR)	5-20
External Interrupt Exception Handling	5-21
Alignment Exception	5-22
Program Exceptions	5-23
System Call Exception	5-24
Programmable Interval Timer (PIT) Exception	5-25
Fixed Interval Timer (FIT) Exception	5-26
Watchdog Timer Exception	5-26
Debug Exception Handling	5-27
Timer Facilities	5-28
Time Base	5-29
Comparison with PowerPC Architecture Time Base	5-30
Programmable Interval Timer (PIT)	5-32
Fixed Interval Timer (FIT)	5-34

Watchdog Timer	5-34
Timer Status Register (TSR)	5-36
Timer Control Register (TCR)	5-37
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
Sequential Cache Operations	6-16
Debugging	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	7-4
Debug Control Register (DBCR)	7-5
Note on the DAC Compare Size Field (DBCR[D1S])	7-6
Debug Status Register (DBSR)	7-6
Data Address Compare Register (DAC1)	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 Chain	7-13
Storage Control	8-1
Physical Address Map	8-1
Storage Attributes	8-3
Data Cache Write-through Register (DCWR)	8-4
Data Cache Cacheability Register (DCCR)	8-6
Instruction Cache Cacheability Register (ICCR)	8-8
Storage Guarded Register (SGR)	8-10
Storage Little-Endian Register (SLER)	8-12
Clock Generation and Power Management	9-1
Power Management Control Register (PMCR0)	9-2
Clock Generation	9-3
Clock Generation and Bus Speed	9-3
Power Management	9-5
Doze Mode	9-6
Nap Mode	9-7
Deep Sleep Mode	9-7
Instruction Set	10-1
Instruction Set Portability	10-1
Instruction Formats	10-2
Pseudocode	10-2
Register Usage	10-5
Alphabetical Instruction Listing	10-5
add	10-6
addc	10-7
adde	10-8
addi	10-9
addic	10-10
addic.	10-11
addis	10-12
addme	10-13
addze	10-14
and	10-15
andc	10-16
andi.	10-17
andis.	10-18
b	10-19
bc	10-20
bcctr	10-28
bclr	10-32
cmp	10-37
cmpi	10-38
cmpl	10-39
cmpli	10-40
cntlzw	10-41

crand	10-42
crandc	10-43
creqv	10-44
crnand	10-45
crnor	10-46
cror	10-47
crorc	10-48
crxor	10-49
dcbf	10-50
dcbi	10-51
dcbst	10-52
dcbt	10-53
dcbtst	10-55
dcbz	10-57
dccci	10-59
dcread	10-61
divw	10-63
divwu	10-64
eieio	10-65
eqv	10-66
extsb	10-67
extsh	10-68
icbi	10-69
icbt	10-71
iccci	10-73
icread	10-74
isync	10-76
lbz	10-78
lbzu	10-79
lbzux	10-80
lbzx	10-81
lha	10-82
lhau	10-83
lhaux	10-84
lhax	10-85
lhbrx	10-86
lhz	10-87
lhzu	10-88
lhzux	10-89
lhzx	10-90
lmw	10-91
lswi	10-92
lswx	10-94
lwarx	10-96
lwbrx	10-98
lwz	10-99
lwzu	10-100

lwzux	10-101
lwzx	10-102
mcrf	10-103
mcrxr	10-104
mfcrr	10-105
mfdcr	10-106
mfmsr	10-108
mfspr	10-109
mtcrf	10-111
mtdcr	10-113
mtmsr	10-115
mtspr	10-116
mulhw	10-118
mulhwu	10-119
mulli	10-120
mullw	10-121
nand	10-122
neg	10-123
nor	10-124
or	10-125
orc	10-126
ori	10-127
oris	10-128
rfci	10-129
rfi	10-130
rlwimi	10-131
rlwinm	10-132
rlwnm	10-135
sc	10-136
slw	10-137
sraw	10-138
srawi	10-139
srw	10-140
stb	10-141
stbu	10-142
stbux	10-143
stbx	10-144
sth	10-145
sthbrx	10-146
sthu	10-147
sthux	10-148
sthx	10-149
stmw	10-150
stswi	10-151
stswx	10-152
stw	10-154
stwbrx	10-155

stwcx	10-156
stwu	10-158
stwux	10-159
stwx	10-160
subf	10-161
subfc	10-162
subfe	10-164
subfic	10-165
subfme	10-166
subfze	10-167
sync	10-168
tw	10-169
twi	10-173
wrtree	10-177
wrtreei	10-178
xor	10-179
xori	10-180
xoris	10-181
Register Summary	11-1
Reserved Registers	11-1
Reserved Fields	11-1
General Purpose Registers	11-2
Machine State Register and Condition Register	11-2
Special Purpose Registers	11-2
Device Control Registers	11-4
Alphabetical Register Listing	11-5
Signal Descriptions	12-1
Pin Description Nomenclature	12-2
I/O Signal Descriptions	12-3
Signals Ordered by Pin Number	12-9
Instruction Summary	A-1
Instruction Set and Extended Mnemonics – Alphabetical	A-1
Instructions Sorted by Opcode	A-39
Instruction Formats	A-47
Instruction Fields	A-47
Instruction Format Diagrams	A-49
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
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.	PPC401GF Block Diagram	1-4
Figure 1-2.	Clock Unit	1-9
Figure 2-1.	Physical Address Map	2-2
Figure 2-2.	General Purpose Register (R0-R31)	2-5
Figure 2-3.	Count Register (CTR)	2-7
Figure 2-4.	Link Register (LR)	2-7
Figure 2-5.	Fixed Point Exception Register (XER)	2-8
Figure 2-6.	Special Purpose Register General (SPRG0-SPRG3)	2-10
Figure 2-7.	Processor Version Register (PVR)	2-11
Figure 2-8.	Condition Register (CR)	2-12
Figure 2-9.	Machine State Register (MSR)	2-15
Figure 2-10.	PPC401GF Data Types	2-17
Figure 2-11.	Normal Word Load or Store (Big Endian Storage Region)	2-30
Figure 2-12.	Byte-reverse Word Load or Store (Little Endian Storage Region)	2-30
Figure 2-13.	Byte-reverse Word Load or Store (Big Endian Storage Region)	2-31
Figure 2-14.	Normal Word Load or Store (Little Endian Storage Region)	2-31
Figure 2-15.	PPC401GF Instruction Queue	2-32
Figure 3-1.	PPC401GF Bus Signals	3-2
Figure 3-2.	Mapping of Address Bits to Bus Bits	3-6
Figure 3-3.	Physical Address Map	3-12
Figure 3-4.	Bus Region Control Register (BRCR0–BRCR7)	3-13
Figure 3-5.	Connecting to the PPC401GF Bus	3-16
Figure 3-6.	Address and Control Signal Connections	3-17
Figure 3-7.	Single Read and Write Word Transfer Timings	3-20
Figure 3-8.	Burst Read of a Word; Burst Cache Line Flush Timings	3-21
Figure 3-9.	Burst Read and Write Word Transfer Timings	3-22
Figure 3-10.	Cache Line Read; Continuous Burst Timings	3-23
Figure 3-11.	Cache Line Flush Burst Timings	3-24
Figure 3-12.	Burst Read/Write Unaligned Halfword Transfer Timings	3-25
Figure 3-13.	Three-byte Burst Reads	3-26
Figure 3-14.	String Read Bus Request Across Word Boundary	3-27
Figure 3-15.	External Bus Interface Timings	3-30
Figure 5-1.	Machine State Register (MSR)	5-7
Figure 5-2.	Save/Restore Register 0 (SRR0)	5-9
Figure 5-3.	Save/Restore Register 1 (SRR1)	5-9
Figure 5-4.	Save/Restore Register 2 (SRR2)	5-10
Figure 5-5.	Save/Restore Register 1 (SRR1)	5-10
Figure 5-6.	Exception Vector Prefix Register (EVPR)	5-11

Figure 5-7.	Exception Syndrome Register (ESR)	5-11
Figure 5-8.	Data Exception Address Register (DEAR)	5-13
Figure 5-9.	Bus Error Status Register 0 (BESR0)	5-15
Figure 5-10.	Bus Address Error Register (BEAR)	5-16
Figure 5-11.	Input/Output Configuration Register (IOCR)	5-21
Figure 5-12.	Relationship of Timer Facilities to the Base Clock	5-29
Figure 5-13.	Time Base Register (TBHI, TBHU)	5-30
Figure 5-14.	Time Base Register (TBLO, TBLU)	5-30
Figure 5-15.	Time Base Comparison	5-32
Figure 5-16.	Programmable Interval Timer (PIT)	5-33
Figure 5-17.	Watchdog Timer State Machine	5-35
Figure 5-18.	Timer Status Register (TSR)	5-37
Figure 5-19.	Timer Control Register (TCR)	5-38
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 (DAC1)	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.	Physical Address Map	8-2
Figure 8-2.	Data Cache Write-through Register (DCWR)	8-4
Figure 8-3.	Data Cache Cacheability Register (DCCR)	8-6
Figure 8-4.	Instruction Cache Cacheability Register (ICCR)	8-8
Figure 8-5.	Storage Guarded Register (SGR)	8-10
Figure 8-6.	Storage Little-Endian Register (SLER)	8-13
Figure 9-1.	Clock Unit	9-1
Figure 9-2.	Power Management Control Register (PMCR0)	9-2
Figure 9-3.	PMCR0 Settings and PPC401GF Operation	9-6
Figure 11-1.	Bus Address Error Register (BEAR)	11-6
Figure 11-2.	Bus Error Status Register 0 (BESR0)	11-7
Figure 11-3.	Bus Region Control Register (BRCR0–BRCR7)	11-8
Figure 11-4.	Cache Debug Control Register (CDBCR)	11-9
Figure 11-5.	Condition Register (CR)	11-10
Figure 11-6.	Count Register (CTR)	11-11
Figure 11-7.	Data Address Compare Register (DAC1)	11-12
Figure 11-8.	Debug Control Register (DBCR)	11-13

Figure 11-9.	Debug Status Register (DBSR)	11-15
Figure 11-10.	Data Cache Cacheability Register (DCCR)	11-17
Figure 11-11.	Data Cache Write-through Register (DCWR)	11-19
Figure 11-12.	Data Exception Address Register (DEAR)	11-21
Figure 11-13.	Exception Syndrome Register (ESR)	11-22
Figure 11-14.	Exception Vector Prefix Register (EVPR)	11-23
Figure 11-15.	General Purpose Register (R0-R31)	11-24
Figure 11-16.	Instruction Address Compare Register (IAC1)	11-25
Figure 11-17.	Instruction Cache Cacheability Register (ICCR)	11-26
Figure 11-18.	Instruction Cache Debug Data Register (ICDBDR)	11-28
Figure 11-19.	Input/Output Configuration Register (IOCR)	11-29
Figure 11-20.	Link Register (LR)	11-30
Figure 11-21.	Machine State Register (MSR)	11-31
Figure 11-22.	Programmable Interval Timer (PIT)	11-33
Figure 11-23.	Power Management Control Register (PMCR0)	11-34
Figure 11-24.	Processor Version Register (PVR)	11-35
Figure 11-25.	Storage Guarded Register (SGR)	11-36
Figure 11-26.	Storage Little-Endian Register (SLER)	11-38
Figure 11-27.	Special Purpose Register General (SPRG0-SPRG3)	11-40
Figure 11-28.	Save/Restore Register 0 (SRR0)	11-41
Figure 11-29.	Save/Restore Register 1 (SRR1)	11-42
Figure 11-30.	Save/Restore Register 2 (SRR2)	11-43
Figure 11-31.	Save/Restore Register 1 (SRR1)	11-44
Figure 11-32.	Time Base High Register (TBHI)	11-45
Figure 11-33.	Time Base High User-mode (TBHU)	11-46
Figure 11-34.	Time Base Low Register (TBLO)	11-47
Figure 11-35.	Time Base Low User-mode (TBLU)	11-48
Figure 11-36.	Timer Control Register (TCR)	11-49
Figure 11-37.	Timer Status Register (TSR)	11-50
Figure 11-38.	Fixed Point Exception Register (XER)	11-51
Figure 12-1.	I/O Diagram	12-1
Figure A-1.	Instruction Format	A-50
Figure A-2.	B Instruction Format	A-50
Figure A-3.	SC Instruction Format	A-50
Figure A-4.	D Instruction Format	A-50
Figure A-5.	X Instruction Format	A-51
Figure A-6.	XL Instruction Format	A-52
Figure A-7.	XFX Instruction Format	A-52
Figure A-8.	XO Instruction Format	A-52
Figure A-9.	M Instruction Format	A-52

Tables

Table 2-1.	PPC401GF SPRs	2-6
Table 2-2.	XER-Updating Arithmetic Instructions	2-9
Table 2-3.	PPC401GF DCRs	2-16
Table 2-4.	Alignment Exception Summary	2-18
Table 2-5.	Bits of the BO Field	2-34
Table 2-6.	Conditional Branch BO Field	2-34
Table 2-7.	Example Memory Mapping	2-39
Table 2-8.	Instruction Execution Privileges and Operating Modes	2-40
Table 2-9.	Privileged Instructions	2-41
Table 2-10.	PPC401GF Instruction Set Functional Summary	2-47
Table 2-11.	Instructions Specific to IBM PowerPC Embedded Controllers	2-47
Table 2-12.	Storage Reference Instructions	2-48
Table 2-13.	Arithmetic and Logical Instructions	2-49
Table 2-14.	Compare Instructions	2-49
Table 2-15.	Branch Instructions	2-50
Table 2-16.	Condition Register Logical Instructions	2-50
Table 2-17.	Rotate and Shift Instructions	2-50
Table 2-18.	Cache Control Instructions	2-51
Table 2-19.	Interrupt Control Instructions	2-51
Table 2-20.	Processor Management Instructions	2-52
Table 3-1.	Transfer Size of a Bus Access	3-6
Table 3-2.	Beats and Bytes Transferred Depending on Operation and Burst Size	3-7
Table 3-3.	B0:31 Usage During the Wait/Data State	3-8
Table 3-4.	BE0:BE3 and Bus Width	3-9
Table 3-5.	BootClkSpd and Bus Speed	3-9
Table 3-6.	BusWidth0:1 and Bus Width	3-10
Table 3-7.	MemClk:Internal Clock Frequency Ratios	3-15
Table 3-8.	Pins for Address Output and Data Transfers	3-16
Table 3-9.	Byte Load/Store Transactions	3-28
Table 3-10.	Halfword Load/Store Transactions	3-28
Table 3-11.	Three-byte Load/Store Transactions	3-28
Table 3-12.	Single-word Load/Store Transactions	3-29
Table 3-13.	Four-word Load/Store Transactions (Cache Line Fills/Flushes)	3-29
Table 4-1.	Processor Configuration After Reset	4-2
Table 4-2.	Contents of the Machine State Register After Reset	4-3
Table 4-3.	Contents of Special Purpose Registers After Reset	4-3
Table 4-4.	Contents of Device Control Registers After Reset	4-5
Table 5-1.	Exception-handling Priorities	5-4

Table 5-2.	Exception Vector Offsets	5-6
Table 5-3.	ESR Alteration by Various Exceptions	5-13
Table 5-4.	Register Settings during Critical Interrupt Exceptions	5-14
Table 5-5.	Register Settings during Machine Check—Instruction Exceptions	5-17
Table 5-6.	Register Settings during Machine Check—Data Exceptions	5-18
Table 5-7.	Register Settings during Data Storage Exceptions	5-20
Table 5-8.	Register Settings during External Interrupt Exceptions	5-22
Table 5-9.	Alignment Exception Summary	5-22
Table 5-10.	Register Settings during Alignment Error Exceptions	5-23
Table 5-11.	ESR Usage for Program Exceptions	5-23
Table 5-12.	Register Settings during Program Exceptions	5-24
Table 5-13.	Register Settings during System Call Exceptions	5-24
Table 5-14.	Register Settings during Programmable Interval Timer Exceptions	5-25
Table 5-15.	Register Settings during Fixed Interval Timer Exceptions	5-26
Table 5-16.	Register Settings during Watchdog Timer Exceptions	5-27
Table 5-17.	SRR2 during Debug Exceptions	5-27
Table 5-18.	Register Settings during Debug Exceptions	5-28
Table 5-19.	FIT Controls	5-34
Table 5-20.	Watchdog Timer Controls	5-34
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.	PPC401GF Sleep Modes	9-5
Table 10-1.	Instructions in the IBM PowerPC Embedded Environment	10-1
Table 10-2.	Operator Precedence	10-5
Table 10-3.	Extended Mnemonics for addi	10-9
Table 10-4.	Extended Mnemonics for addic	10-10
Table 10-5.	Extended Mnemonics for addic.	10-11
Table 10-6.	Extended Mnemonics for addis	10-12
Table 10-7.	Extended Mnemonics for bc, bca, bcl, bcla	10-21
Table 10-8.	Extended Mnemonics for bcctr, bcctrl	10-29
Table 10-9.	Extended Mnemonics for bclr, bclrl	10-33
Table 10-10.	Extended Mnemonics for cmp	10-37
Table 10-11.	Extended Mnemonics for cmpi	10-38
Table 10-12.	Extended Mnemonics for cmpl	10-39
Table 10-13.	Extended Mnemonics for cmpli	10-40
Table 10-14.	Extended Mnemonics for creqv	10-44
Table 10-15.	Extended Mnemonics for crnor	10-46

Table 10-16.	Extended Mnemonics for cror	10-47
Table 10-17.	Extended Mnemonics for crxor	10-49
Table 10-18.	Data Cache Array Tag Information	10-61
Table 10-19.	Instruction Cache Array Tag Information	10-74
Table 10-20.	Extended Mnemonics for mfdcr	10-107
Table 10-21.	Extended Mnemonics for mfspr	10-110
Table 10-22.	Extended Mnemonics for mtrcrf	10-112
Table 10-23.	Extended Mnemonics for mtdcr	10-114
Table 10-24.	Extended Mnemonics for mtspr	10-117
Table 10-25.	Extended Mnemonics for nor, nor.	10-124
Table 10-26.	Extended Mnemonics for or, or.	10-125
Table 10-27.	Extended Mnemonics for ori	10-127
Table 10-28.	Extended Mnemonics for rlwimi, rlwimi.	10-131
Table 10-29.	Extended Mnemonics for rlwinm, rlwinm.	10-132
Table 10-30.	Extended Mnemonics for rlwnm, rlwnm.	10-135
Table 10-31.	Extended Mnemonics for subf, subf., subfo, subfo.	10-161
Table 10-32.	Extended Mnemonics for subfc, subfc., subfco, subfco.	10-163
Table 10-33.	Extended Mnemonics for tw	10-170
Table 10-34.	Extended Mnemonics for twi	10-174
Table 11-1.	PPC401GF General Purpose Registers	11-2
Table 11-2.	Special Purpose Registers	11-3
Table 11-3.	PPC401GF Device Control Registers	11-4
Table 11-4.	ICU Tag Information	11-28
Table 12-1.	Pin Description Nomenclature	12-2
Table 12-2.	PPC401GF I/O Signal Descriptions	12-3
Table 12-3.	Signals Ordered by Pin Number	12-9
Table A-1.	PPC401GF Instruction Syntax Summary	A-2
Table A-2.	PPC401GF Instructions by Opcode	A-39
Table B-1.	PPC401GF 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 PPC401GF	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™ 401GF (PPC401GF™) 32-bit RISC embedded controller.

The PPC401GF RISC embedded controller features

- PowerPC Architecture™
- Single-cycle execution for most instructions
- On-chip 4KB instruction cache and 2KB 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 PPC401GF. The audience should understand embedded system design, operating systems, RISC processing, and design for testability.

How to Use This Book

This book describes the PPC401GF 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	Memory and Peripheral Interface
Chapter 4	Reset and Initialization
Chapter 5	Exceptions, Interrupts, and Timers
Chapter 6	Cache Operations
Chapter 7	Debugging
Chapter 8	Storage Control
Chapter 9	Clock Generation and Power Management
Chapter 10	Instruction Set
Chapter 11	Register Summary
Chapter 12	Signal Descriptions

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 10.3, “Pseudocode,” on p. 10-2, which describes the notational conventions used in instruction descriptions.

<u>Active_Low</u>	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)

and application and technical notes. To obtain copies of these publications, contact your IBM Microelectronics representative.

The IBM PowerPC 401GF 32-bit RISC embedded controller (PPC401GF) is an implementation of the PowerPC Architecture. After a brief overview of the features of the PPC401GF, this chapter discusses the layered organization of the PowerPC Architecture. The chapter then discusses how the PPC401GF implements a variation of the PowerPC Architecture that has been optimized for embedded control applications. PPC401GF compliance with the PowerPC Architecture is discussed. Finally, the major functional units, instruction types, and register types of the PPC401GF are discussed, along with a block diagram to illustrate principal external interfaces and internal flow of data and control signals.

1.1 PPC401GF Overview

The PPC401GF 32-bit RISC embedded controller offers high performance with low power consumption. The PPC401GF RISC CPU executes at sustained speeds approaching one cycle per instruction. On-chip caches reduce chip count and design complexity in systems, while improving system throughput. Features of the PPC401GF include:

- PowerPC RISC fixed-point CPU and PowerPC User Instruction Set Architecture
 - Thirty-two 32-bit general purpose registers
 - Branch prediction
 - Single-cycle execution for most instructions
 - Hardware multiplier and divider for faster integer arithmetic
 - Enhanced string and multiple-word handling
- Interfaces to memory and peripherals
 - 32-bit multiplexed data and address bus
 - Addressing for 4 gigabytes of external memory and MMIO
 - Support for direct connection of byte, halfword, and fullword device controllers
 - External bus clock frequencies of 1X, 1/2X, 1/3X, or 1/4X the CPU core frequency

- Storage attribute control
 - WIGE (write-back/write through, inhibited, compressed, guarded, Endian) storage attribute control for thirty-two 128MB regions
 - True Little Endian operation and PowerPC Endian modes
- Separate instruction cache and write-back/write-thru data cache, both two-way set-associative
- Minimized interrupt latency
- Advanced power management support
- Individually programmable on-chip interfaces for:
 - Bus control regions
 - External interrupts
- Flexible interface to external bus masters

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.

The first two levels of standards represent the instruction set and facilities available to the application programmer. The third level includes features such as system-level instructions which 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.

1.2.1 The PPC401GF as a PowerPC Implementation

The PPC401GF implements the PowerPC User Instruction Set Architecture, user-level registers, programming model, data types, and addressing modes for 32-bit fixed-point operations. This PowerPC Architecture specifies the instruction set and registers that should be provided to support user-level programs. The PPC401GF is fully compliant 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. Both of these kinds of operations are trapped and can be emulated in software.

Most of the architected features of the PPC401GF 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. In addition to these standard features, the PPC401GF provides a number of optimizations and extensions to these levels of the architecture. The full architecture of the PPC401GF is defined by the PowerPC Embedded Environment specifications, together with the common PowerPC User Instruction Set Architecture.

The primary differences between the standard PowerPC Architecture and the embedded variation of it are the following:

- A simplified memory management mechanism with enhancements for embedded applications
- An enhanced, dual-level interrupt structure
- An architected Device Control Register (DCR) address space for integrated system control functions, such as the Bus Region Control Registers (BRCRs)
- The addition of several instructions to support these modified and extended resources

Finally, some of the specific implementation features of the PPC401GF are beyond the scope of the PowerPC Architecture. These features are included to enhance performance, integrate functionality, and/or reduce system complexity in embedded control applications.

1.3 PPC401GF Features

The PPC401GF consists of a pipelined processor core. 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.

Figure 1-1 illustrates the logical organization of the PPC401GF.

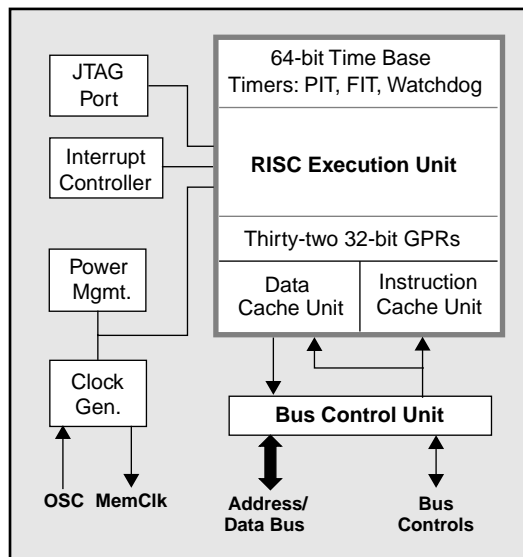


Figure 1-1. PPC401GF Block Diagram

1.3.1 RISC Core

The RISC core comprises four tightly-coupled functional units: the execution unit (EXU), the storage control unit (SCU), the data cache unit (DCU), and the instruction cache unit (ICU). Each cache unit consists of a data array, tag array, and control logic for cache management and addressing. The SCU provides protection functions and storage attribute control for embedded applications. The EXU consists of general purpose registers (GPRs), special purpose registers (SPRs), ALU and multiplier, timers, instruction decode, and the control logic required to manage instruction execution and EXU data flow.

1.3.1.1 Execution Unit (EXU)

The EXU handles instruction fetching, decoding and execution, queue management, and branch prediction. The instruction cache unit passes instructions to the queue in the EXU or, in the event of a cache miss, requests a fetch from external memory through the bus control unit (BCU). To minimize overhead in handling cache misses and noncacheable instructions, a bypass is available from the BCU, which interfaces to the external memory being accessed, to the EXU.

Data transfers to and from the EXU are handled through the bank of 32 GPRs, each 32 bits wide. Load and store instructions move data operands between the GPRs and the data cache unit. In cases of noncacheable data or cache misses, the DCU passes the address

for the data read or write to the BCU. To minimize overhead in handling cache misses and noncacheable operands, a bypass is available from the BCU, which interfaces to the external memory being accessed, to the EXU.

In addition to 32 GPRs, the EXU contains status and special purpose registers that can be read or written by executing programs. Some registers are accessible only while the processor is in privileged mode, while others can also be accessed in user mode.

A robust set of timer facilities is integrated into the EXU. Four timer facilities are provided:

- A 64-bit time base
- A 32-bit count-down programmable interval timer with auto-reload
- A fixed interval timer with four selectable intervals
- A watchdog timer with four intervals and built-in reset

The frequency of the time base and other timer facilities is derived from a clock input which connects to the processor clock. The clock input may be either a crystal or an oscillator.

Two-level exception priority logic in the EXU combines and prioritizes exception sources. When an enabled exception is detected, an interrupt occurs and the processor suspends the current instruction stream and begins executing an exception handling routine.

Exceptions are categorized as either critical or noncritical. Noncritical exceptions include those caused by instruction execution, asynchronous external exceptions, and programmable and fixed interval timer exceptions. Critical exceptions include debug exceptions, machine checks, a critical-external-exception input, and the watchdog timer exception. Critical exceptions have higher priority and are not automatically disabled when an interrupt due to a noncritical exception occurs.

Debug facilities in the PPC401GF are divided between the RISC core and the a JTAG/debug unit external to the core. The Joint Test Action Group (JTAG)/debug unit contains a standard JTAG state machine, together with boundary scan logic and other resources accessible through the an external JTAG interface. See Chapter 7, “Debugging,” for detailed descriptions of the debug facilities and the JTAG interface.

1.3.1.2 Storage Control

The PPC401GF does not support address translation.

The PPC401GF has a 32-bit address bus and a 4 gigabyte (GB) address space, which is presented as a flat address space.

Up to 4GB of external memory can be attached.

The PPC401GF provides storage attribute controls for memory management. See Chapter 8, “Storage Control,” for more information about the storage control attributes.

1.3.1.3 Instruction Cache Unit (ICU)

An instruction cache minimizes access latency for frequently executed instructions. Instruction lines from cacheable memory regions can be prefetched into the ICU. The ICU buffers each four-word line before placing the line into the cache during each fill.

The ICU contains a two-way set-associative 2KB cache memory. Each of the two sets is organized as 64 lines of 16 bytes each.

A separate bypass path handles cache-inhibited instructions to improve performance during line fill operations.

1.3.1.4 Data Cache Unit (DCU)

The DCU manages data transfers between external memory and the general-purpose registers (GPRs) in the execution unit to minimize access latency for frequently used data in external memory. The DCU features byte-writeability to improve the performance of stores to byte and halfword devices.

The DCU contains a two-way set-associative 1KB cache memory. Each of the two cache sets is organized as 32 lines of 16 bytes each.

DCU operations employ either a write-back or write-through strategy to update cached data and maintain coherency with external 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 whenever it is necessary to free up locations for incoming lines. The write-through strategy updates both the cache and external memory during store operations.

A separate bypass path handles non-cacheable loads to improve performance during line fill operations.

1.3.2 External Memory and Peripheral Interfaces

The BCU provides a 32-bit device-paced external data bus interface supporting connection of 8-, 16-, and 32-bit memory and I/O devices.

Eight Bus Region Control Registers (BRCR0–BRCR7) configure the properties of sixteen 256MB bus regions. The configurable properties include device width, hold cycle timings, sequential or target-word-first cache line fills, and four-beat or continuous bursts. Note that each BRCR maps its configurable properties to two bus regions. If address bit 0 (A0) contains 0, the referenced bus region is in the lower half of the address space. If A0 = 1, the referenced bus region is in the upper half.

Several registers provide control of the storage attributes for physical memory. 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 region 0 is associated with the lowest 128MB region, bit 1 the next-lowest 128MB region, and so on.

The PPC401GF provides a shared external bus protocol that allows external bus masters to take control of the PPC401GF external bus. Section 3.11, “External Bus Master Interface,” on p. 3-29, describes this interface in detail.

1.3.3 Interrupt Controller Interface

The PPC401GF provides an interface to an interrupt controller that is logically outside the PPC401GF 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.

1.3.4 Debug Support

Debug is supported by the JTAG port. The IEEE 1149.1 Test Access Port, commonly called JTAG (Joint Test Action Group), is described in IEEE standards document 1149.1, *IEEE Standard Test Access Port and Boundary Scan Architecture*. The standard provides a method for accessing internal facilities on a chip using a four or five signal interface. The JTAG port was originally designed to support scan-based board testing. The JTAG boundary-scan register allows testing of circuitry external to the chip, primarily the board interconnect. Alternatively, the JTAG bypass register can be selected when no other test data register needs to be accessed during a board-level test operation.

The PPC401GF 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.

1.3.5 Data Types

PPC401GF operands are bytes, halfwords, or words. Multiple words or strings of bytes can be transferred using the load/store multiple/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 may be selected to be either Big Endian (the lowest memory address of an operand contains its most significant byte) or 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 PPC401GF also supports true Little Endian addressing and data types. See Section 2.4, “Byte Ordering,” on p. 2-19, for more information about PowerPC Endian modes and true Little Endian operation.

1.3.6 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 device control registers (DCRs).

Chapter 11, “Register Summary,” provides a register diagram and a register field description table for each register.

1.3.6.1 General Purpose Registers

The PPC401GF contains 32 GPRs; each register contains 32 bits. The contents of these registers can be transferred from memory using load instructions and stored to memory using store instructions. GPRs are specified as operands in many PPC401GF instructions.

1.3.6.2 Special Purpose Registers

SPRs contain status and control for resources within the RISC core. Only the Count Register (CTR), Fixed-point Exception Register (XER), Link Register (LR), Time Base High User-mode (TBHU), and the Time Base Low User-mode (TBLU) can be accessed by user-mode programs. Access to all other SPRs is privileged. SPRs are accessed using **mf spr** and **mt spr** instructions.

1.3.6.3 Machine State Register

The PPC401GF contains a 32-bit Machine State Register (MSR). The contents of a GPR can be written to the MSR using the **mt msr** instruction, and the MSR contents can be read into a GPR using the **mf msr** instruction.

1.3.6.4 Condition Register

The PPC401GF contains a 32-bit Condition Register (CR). Instructions are provided to perform logical operations on CR bits and to test CR bits.

1.3.6.5 Device Control Registers

DCRs, which are outside the RISC core, contain status and controls for bus and peripheral controllers and devices. DCRs are accessed using **mt dcr** and **mf dcr** instructions. Access to all DCRs is restricted to supervisor-mode programs.

1.3.7 Addressing Modes

The PPC401GF 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 the “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.

1.4 Clock Generation and Power Management

The PPC401GF clock unit integrates clock generation and power management. The clock unit, which uses a crystal or oscillator input, generates duty cycle-corrected internal clocks having a frequency that is 2X, 1X, 1/2X, or 1/4X the reference clock frequency. The clock unit also provides a duty cycle-corrected MemClk output signal for external bus reference. The MemClk frequency can be 1X, 1/2X, 1/3X, or 1/4X the internal chip clock frequency.

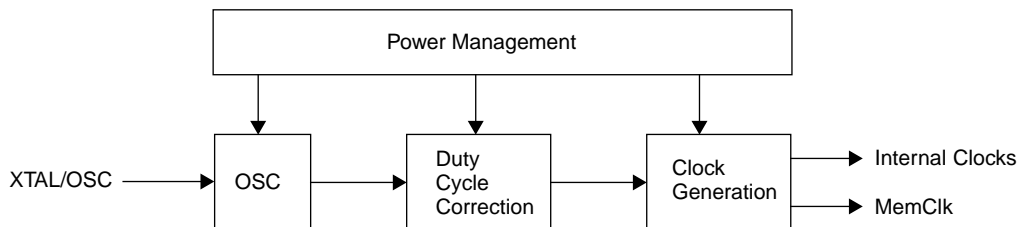


Figure 1-2. Clock Unit

Chapter 9, “Clock Generation and Power Management,” describes the clock unit and power management in detail.

The clock unit greatly reduces standby power consumption when programmed to invoke one of the three available sleep modes: doze, nap, and deep sleep. In doze mode, clock generation continues, but clock distribution is suspended. Nap mode saves even more power: all clock generation circuitry, except for the on-chip oscillator, shuts down. Deep sleep mode saves the most power: all on-chip circuits shut down, reducing power consumption to an extremely low level. Normal processing resumes from any sleep mode when an interrupt, external bus request, or reset occurs. (The processor wait state also provides power savings; in wait mode, clock generation and distribution occur as in normal chip operation, but instruction fetching and execution are suspended.)

The deeper the level of sleep, the longer the wake-up latency. This power control granularity provides the system designer with the flexibility to balance low power requirements against wake-up latency requirements. (Note that the processor wait state provides some power savings without wake-up latency.)

2

Programming Model

The programming model of the PPC401GF embedded controller describes how the following features and operations of the PPC401GF appear to programmers:

- Memory organization and addressing, starting on p. 2-1
- Registers, starting on p. 2-4
- Data types and alignment, starting on p. 2-17
- Byte ordering, starting on p. 2-19
- Instruction processing, starting on p. 2-32
- Branching control, starting on p. 2-32
- Speculative accesses, starting on p. 2-36
- Privileged mode operation, starting on p. 2-40
- Synchronization, starting on p. 2-42
- Instruction set, starting on p. 2-46

2.1 Memory Organization and Addressing

The PowerPC Architecture defines a 32-bit, 4 gigabyte (GB) flat address space for instructions and data. The PPC401GF provides a 32-bit address bus to support attachment of memory to the entire 4GB address space.

Bus Regions	Bus Region Control Registers	Storage Attribute Control Register Bits	Storage Attribute Control Regions
0xFFFF FFFF	BRCR7	31	0xF800 0000
0xF000 0000		30	0xF000 0000
0xEFFF FFFF	BRCR6	29	0xE800 0000
0xE000 0000		28	0xE000 0000
0xDFFF FFFF	BRCR5	27	0xD800 0000
0xD000 0000		26	0xD000 0000
0xCFFF FFFF	BRCR4	25	0xC800 0000
0xC000 0000		24	0xC000 0000
0xBFFF FFFF	BRCR3	23	0xB800 0000
0xB000 0000		22	0xB000 0000
0xAFFF FFFF	BRCR2	21	0xA800 0000
0xA000 0000		20	0xA000 0000
0x9FFF FFFF	BRCR1	19	0x9800 0000
0x9000 0000		18	0x9000 0000
0x8FFF FFFF	BRCR0	17	0x8800 0000
0x8000 0000		16	0x8000 0000
0x7FFF FFFF	BRCR7	15	0x7800 0000
0x7000 0000		14	0x7000 0000
0x6FFF FFFF	BRCR6	13	0x6800 0000
0x6000 0000		12	0x6000 0000
0x5FFF FFFF	BRCR5	11	0x5800 0000
0x5000 0000		10	0x5000 0000
0x4FFF FFFF	BRCR4	9	0x4800 0000
0x4000 0000		8	0x4000 0000
0x3FFF FFFF	BRCR3	7	0x3800 0000
0x3000 0000		6	0x3000 0000
0x2FFF FFFF	BRCR2	5	0x2800 0000
0x2000 0000		4	0x2000 0000
0x1FFF FFFF	BRCR1	3	0x1800 0000
0x1000 0000		2	0x1000 0000
0x0FFF FFFF	BRCR0	1	0x0800 0000
0x0000 0000		0	0x0000 0000

Figure 2-1. Physical Address Map

2.1.1 Physical Address Map

Figure 2-1 shows the memory map of the PPC401GF, which is divided into 16 bus regions and 32 storage attribute control (SAC) regions.

Each of the eight Bus Region Control Registers (BRCR0–BRCR7) controls two 256MB non-contiguous bus regions that share the following programmable characteristics:

- Transfer hold cycles
- Maximum burst; four beats or continuous
- Target word-first or sequential line fills
- Bus width

Each 128MB SAC region is associated with a bit in each of the 32-bit storage attribute control registers described in Section 2.1.2.

2.1.2 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 PPC401GF provides control mechanisms for the WIGE attributes. Because the PPC401GF 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 11, “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.

2.1.3 Mapping Multiple Storage Attributes to Common Memory Locations

As noted in Section 2.1.1, each BRCCR controls two bus regions. In the lower of the two regions, address bit 0 (A0) contains a 0. In the higher region, A0 = 1. Note that each bus region is associated with four SAC regions, two “low” (A0 = 0) and two “high” (A0 = 1).

An external device controlled by a BRCCR can ignore A0 or A4, or both address bits, to support double- or quad-mapping of storage attributes to an external memory location. One, two, or four effective addresses (addresses for instruction fetches and data loads/stores) can refer to one byte of external memory, depending on whether any address bits are ignored. If the external address recognizes all bits in an effective address, one effective address refers to one byte of external memory. If A0 *or* A4 is ignored, two effective addresses refer to one byte of external memory. If A0 *and* A4 are ignored, four effective addresses refer to one byte of external memory. SAC regions are independent of the bus regions. Each address is associated with only one set of storage attributes. However, each physical memory location can be associated with as many as four addresses.

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 11, “Register Summary.”

All registers in the PPC401GF 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 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 PPC401GF 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 11-1 on p. 11-2 for the numbering of the GPRs.

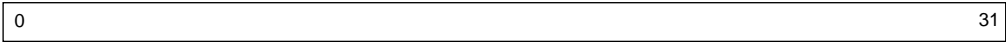


Figure 2-2. General Purpose Register (R0-R31)



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 11-2 on p. 11-3 shows the mnemonic, name, and number for each SPR. Table 2-1 lists the PPC401GF SPRs by function and points to the pages where the SPRs are described more fully.

Table 2-1. PPC401GF SPRs

Function	Register				Access	Page	
Branch Control	CTR				User	2-6	
	LR				User	2-7	
Debug	CDBCR				Privileged	6-9	
	DAC1				Privileged	7-8	
	DBCR				Privileged	7-5	
	DBSR				Privileged	7-6	
	IAC1				Privileged	7-10	
	ICDBDR				Privileged	6-10	
Fixed-point Exception	XER				User	2-8	
General-purpose	SPRG0	SPRG1	SPRG2	SPRG3	Privileged	2-10	
Interrupts and Exceptions	DEAR				Privileged	5-13	
	ESR				Privileged	5-11	
	EVPR				Privileged	5-10	
	SRR0				SRR1	Privileged	5-8
	SRR2				SRR3	Privileged	5-9
Processor Version	PVR				Privileged, read-only	2-10	
Storage Attributes	DCCR				Privileged	8-6	
	DCWR				Privileged	8-4	
	ICCR				Privileged	8-8	
	SGR				Privileged	8-10	
	SLER				Privileged	8-12	
Timer Facilities	TBHI	TBLO			Privileged	5-29	
	TBHU	TBLU			User read-only	5-29	
	PIT	Privileged			5-32		
	TCR	Privileged			5-37		
	TSR	Privileged			5-36		

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. 2-41. 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.

0	31
---	----

Figure 2-3. 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.

0	31
---	----

Figure 2-4. 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.

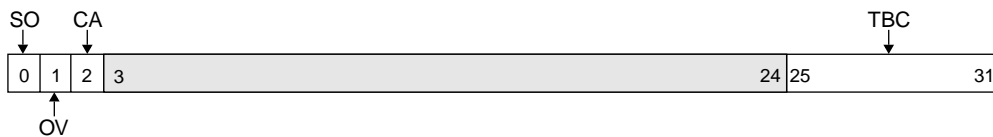


Figure 2-5. Fixed Point Exception Register (XER)

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. 2-9); 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. 2-9); 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. 2-9); 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. 2-41 for more information.



Figure 2-6. 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. 2-41 for more information.

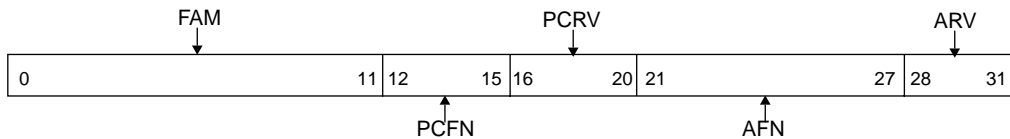


Figure 2-7. Processor Version Register (PVR)

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.	.
16:20	PCRV	Processor Core Revision. Identifies a revision of the processor core defined by the PFN field.	0x0 for PPC401GF
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 2-8. The CR reflects the results of some operations (as indicated in the instruction descriptions in Chapter 10, “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. 2-41 for more information.

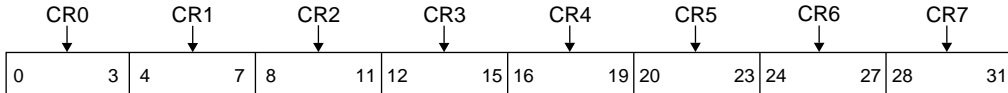


Figure 2-8. Condition Register (CR)

Figure 2-8. 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. 10-37 through p. 10-40) 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 PPC401GF 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. 5-29, for detailed differences between the PPC401GF 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.

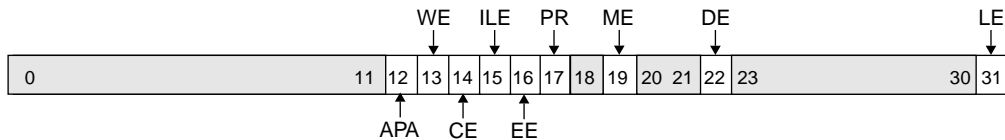


Figure 2-9. Machine State Register (MSR)

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 PPC401GF 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 2-9. 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).

DCRs control the use of the bus regions, store status and address data for bus errors, and control interrupts. Table 11-3 on p. 11-4 shows the mnemonic, name, and number for each DCR.

All DCRs are privileged for both read and write. See Section 2.8, “Privileged Mode Operation,” on p. 2-40, for more information.

Table 2-3. PPC401GF DCRs

Function	Register				Page
Bus Error	BEAR	BESR			5-16
Bus Region Control	BRCR0	BRCR1	BRCR2	BRCR3	3-11
	BRCR4	BRCR5	BRCR6	BRCR7	
Bus Configuration	IOCR				5-20
Power Management	PMCR0				9-2

2.3 Data Types and Alignment

PPC401GF data types consist of bytes (eight bits), halfwords (two bytes), words (four bytes), and strings (one or more bytes containing character data). Figure 2-10 shows the byte, halfword, and word data types and their bit and byte definitions.

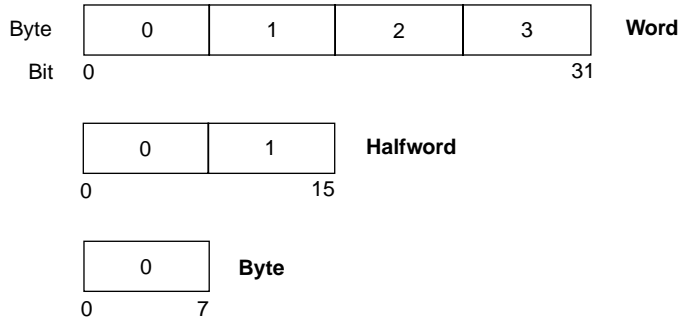


Figure 2-10. PPC401GF Data Types

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 2-12, "Storage Reference Instructions," on p. 2-48) move data to and from storage. The data cache control instructions (see Table 2-18, "Cache Control Instructions," on p. 2-51) 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 10, "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 PPC401GF (see Section 2.4, "Byte Ordering," on p. 2-19 for information about endian operation).

Table 2-4, “Alignment Exception Summary,” on p. 2-18, 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 PPC401GF 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 PPC401GF is operating as a big endian processor (MSR[LE] = 0), EA misalignments do not cause alignment exceptions except as summarized in Table 2-4,.

When the PPC401GF 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 2-4 summarizes the instructions that cause alignment exceptions and the conditions under which the alignment exceptions occur.

Table 2-4. Alignment Exception Summary

PPC401GF 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 2-4. Alignment Exception Summary (cont.)

PPC401GF 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 PPC401GF 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. 2-27.

When the PPC401GF 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. 2-20) 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 PPC401GF 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. 2-26, 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 PPC401GF 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 PPC401GF 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 PPC401GF 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 PPC401GF 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 PPC401GF generates alignment exceptions when attempting to execute any of the following instruction types, if the PPC401GF 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. 5-22, 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 PPC401GF. 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 PPC401GF 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 PPC401GF 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 PPC401GF 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 PPC401GF 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 PPC401GF 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 PPC401GF, 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. 2-20, 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. 5-22, 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 PPC401GF 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 2-11 through 2-14 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 2-11 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 2-11. 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 2-12.

MSB		LSB		
11	12	13	14	GPR

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

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

Figure 2-13 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.

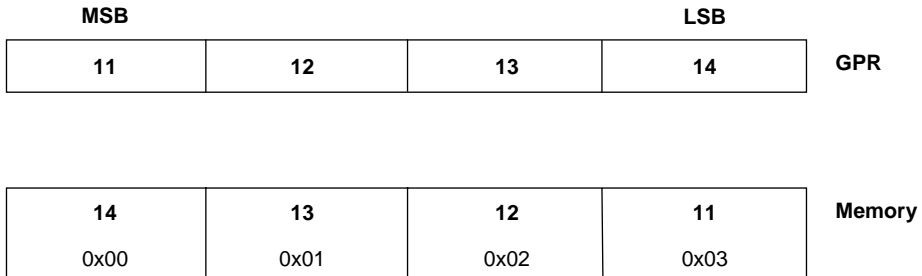


Figure 2-13. Byte-reverse Word Load or Store (Big Endian Storage Region)

Note that the results are identical to the results of a normal load/store in a little endian storage region, as illustrated in Figure 2-14.

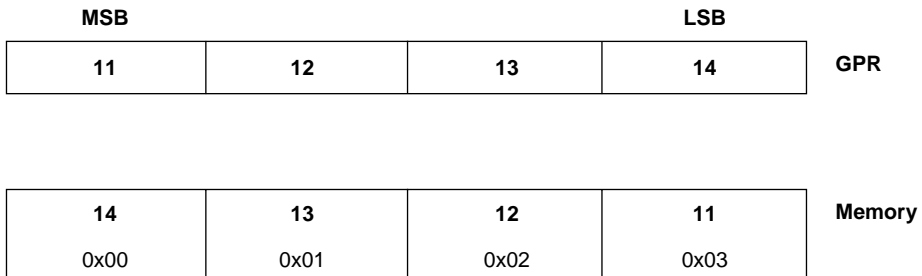


Figure 2-14. Normal Word Load or Store (Little Endian Storage Region)

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 2-15, 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.

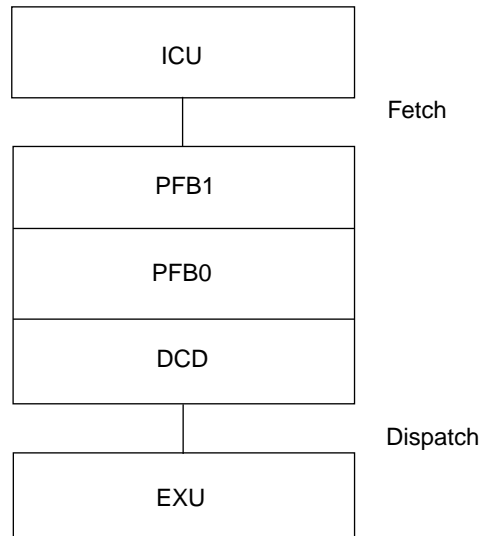


Figure 2-15. PPC401GF Instruction Queue

2.6 Branching Control

The PPC401GF, 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. 2-35.

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 2-5 summarizes the usage of the bits of the BO field. BO[4] is further discussed in Section 2.6.5.

Table 2-5. 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 2-6 lists specific BO field contents, and the resulting actions. In Table 2-6, z represents a mandatory value of zero, and y is a branch prediction option discussed in Section 2.6.5.

Table 2-6. 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 PPC401GF 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 PPC401GF 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 2-6 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 PPC401GF

The PPC401GF does not perform out-of-order execution, nor does the PPC401GF perform speculative loads.

The PPC401GF 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 PPC401GF 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 2-7 shows two address regions of the PPC401GF. 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 2-7. 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**, **rfci**, or **sc** instructions by putting appropriate instructions before or after the **rfi**, **rfci**, 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 2-8. 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 MSR[PR] = 1, the processor is in user mode (problem state); when MSR[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 MSR[PR] = 1 causes a privileged violation program exception (see Section 5.9, “Program Exceptions,” on p. 5-23). The PPC401GF 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 MSR[PR] bit is saved in the SRR1/SRR3 (along with all the other MSR bits) upon any interrupt, and the MSR[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 2-9. 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 PPC401GF 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 PPC401GF, these include most interrupts and the **isync**, **rfci**, **rfi**, 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 PPC401GF, one must use either an **eieio** instruction or a **sync** instruction. Note that for the PPC401GF, the **eieio** and **sync** instructions are implemented identically. See Section 2.9.3 on p. 2-46.

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 psuedocode 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. Consider the following instruction sequence:

```
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 PPC401GF provides three execution synchronizing operations: the **ieio**, **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 PPC401GF 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 **ieio** 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 **ieio**.

2.9.3 Storage Synchronization

The **sync** instruction guarantees that all previous storage references complete with respect to the PPC401GF 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 **eieio** instruction guarantees the order of storage accesses. All storage accesses that precede **eieio** 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
eieio	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 . eieio was necessary, because the read and write addresses are different, but affect each other

The PPC401GF implements both **sync** and **eieio** identically, in the manner described above for **sync**. In the PowerPC Architecture, **sync** can function across all processors in a multiprocessor environment; **eieio** functions only within its executing processor. The PPC401GF is a uniprocessor; in this implementation, **sync** does not guarantee memory ordering across multiprocessors.

2.10 Instruction Set

The PPC401GF instruction set contains instructions defined in the PowerPC Architecture and instructions specific to the IBM PowerPC 400 family of embedded controllers.

Chapter 10, "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 2-10.

Table 2-10 summarizes the PPC401GF instruction set functions by categories. Instructions within each category are described in subsequent sections.

Table 2-10. PPC401GF 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 2-11 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 2-11. 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 2-12 shows the storage reference instructions in the PPC401GF.

Table 2-12. Storage Reference Instructions

Loads					Stores			
Byte	Halfword Algebraic	Halfword	Multiple and String	Word	Byte	Halfword	Multiple and String	Word
lbz	lha	lhrx	lmw	lwarx	stb	sth	stmw	stw
lbzu	lhau	lhz	lswi	lwbrx	stbu	sthbrx	stswi	stwbrx
lbzux	lhaux	lhzu	lswx	lwz	stbux	sthu	stswx	stwu
lbzx	lhax	lhzux		lwzu	stbx	sthux		stwux
		lhzx		lwzux		sthx		stwx
				lwzx				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 2-13 lists the arithmetic and logical instructions in the PPC401GF.

Table 2-13. Arithmetic and Logical Instructions

Arithmetic						Logical		
add	addi	divw	mulhw	subf	subfc	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 2-14 lists the comparison instructions in the PPC401GF.

Table 2-14. 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 2-15 lists the branch instructions in the PPC401GF.

Table 2-15. 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 2-16 lists the condition register logical instructions in the PPC401GF.

Table 2-16. 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 2-17 lists the rotate and shift instructions in the PPC401GF.

Table 2-17. 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 2-18 lists the cache control instructions in the PPC401GF.

Table 2-18. 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 2-19 lists the interrupt control instructions in the PPC401GF.

Table 2-19. 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 PPC401GF, and provide traps, system calls, and synchronization controls.

Table 2-20 lists the processor management instructions in the PPC401GF.

Table 2-20. 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 10, “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.

3

Memory and Peripheral Interface

The PPC401GF memory and peripheral interface is a device-paced, multiplexed bus interface. The Bus Control Unit (BCU) is the internal structure controlling the interface between the PPC401GF and memory and peripherals. The BCU controls the interfaces between storage (addresses external to the PPC401GF) and users of that storage (the processor core and external bus masters).

The BCU controls address output and data transfers on the multiplexed external address/data bus, B0:B31. To enhance data transfer bandwidth, the BCU supports both four-beat burst access and continuous burst access, up to 16 byte transfers or four word transfers.

All PPC401GF external bus operations are synchronous to the MemClk output, simplifying control logic for external memory attached to the bus. The external bus interface can be programmed to operate at 1X, 1/2X, 1/3X or 1/4X the internal clock speed. Bus width is programmable to support attachment to 8-, 16-, and 32-bit memory and peripherals. Further details of peripheral attachment are given in Section 3.8, "Connecting to the PPC401GF Bus," on p. 3 -15.

Figure 3-1 illustrates the chip I/O signals for the external bus. Chapter 12, “Signal Descriptions,” contains a complete logic symbol and tables describing the chip I/O signals.

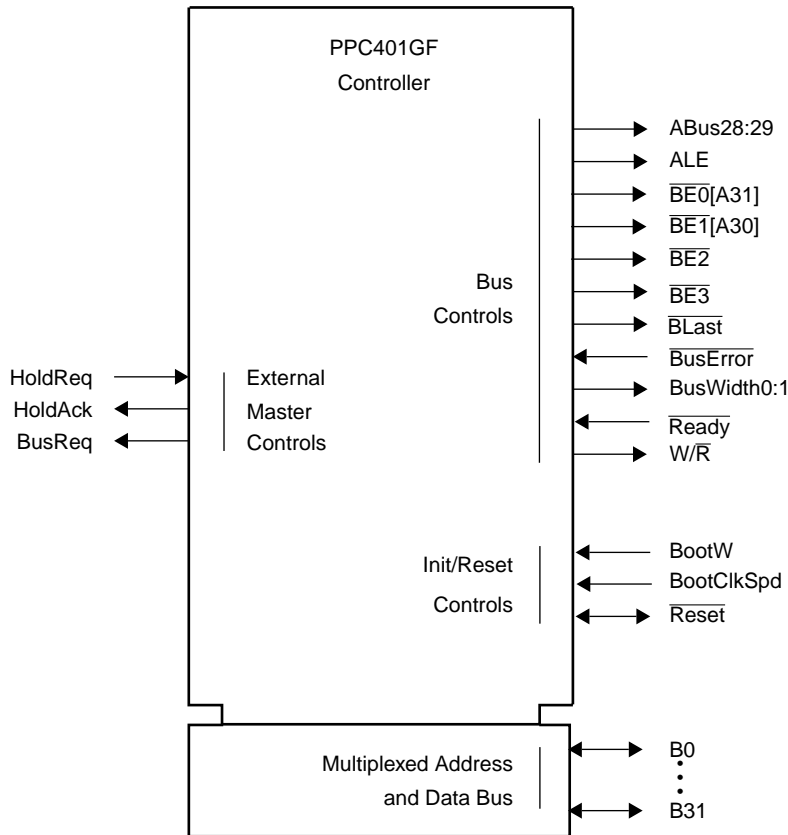


Figure 3-1. PPC401GF Bus Signals

The PPC401GF bus signals are grouped within functional categories, as shown in Figure 3-1. The signals grouped as “bus controls,” “external master,” and “multiplexed address and data bus” comprise the memory and peripheral interface.

The bus signals have the same meanings, whether or not the PPC401GF is the bus master.

3.1 Terminology

In descriptions of bus states, signals, and operations, the following terms are used:

Bus request	A request, resulting from a load/store instruction or instruction prefetch, by the PPC401GF or external bus master to transfer data on the bus.
Bus access	Bus activity, resulting from a bus request, that begins with the assertion of the ALE signal and ends with the deassertion of the $\overline{\text{BLast}}$ signal. A bus access comprises from one to 16 data transfers. The number of data transfers in a bus access is determined by the size of the data item associated with the access, the alignment of its address, and the programmed bus width.
Data transfer	A read (for loads and instruction prefetches) or write (for stores) operation in which data is transferred from the PPC401GF to a external device, or from an external device to the PPC401GF.

Sometimes the term “bus transaction” is used to refer to all bus activity resulting from a bus request.

3.2 Bus States

The external bus has five states:

- Idle
- Address
- Wait/Data
- Recovery
- Hold

3.2.1 Idle State

The bus is in the Idle state when $\overline{\text{Reset}}$ is asserted or when no bus transactions are in progress. The bus remains in the Idle state until the BCU receives a bus request from either cache controller or an external bus master.

The bus can enter the Idle state from the Recovery state (see Section 3.2.4) or the Hold state (see Section 3.2.5).

3.2.2 Address State

When a BCU receives a bus request, it initiates a bus access and the bus enters the Address state. In this state, the multiplexed address/data signals transfer the address of the data to be read or written.

The single clock cycle of the Address state is referred to as the address cycle. During this cycle, the PPC401GF:

- Outputs a valid address on B0:31.

Note that B0:31 are pin names. In the Address state, these signals are called A0:31 for convenience. Figure 3-2 on p. 3 -6 illustrates the mapping of the address bits to the bus pins.

- Activates the ALE signal; an external device can use ALE to latch the address.

Typically, A0:27 are latched; A28:31 can be latched to provide size information for burst transfers. Section 3.3.3, “B0:31 (Multiplexed Address Bus Bits 0:31),” on p. 3 -6, describes how these bits are used to provide size information and shows which pins that are carried these bits.

- Outputs $\overline{BE0}[A31]:\overline{BE1}[A30]$ (which alternatively carry byte enables or the two least significant address bits), $\overline{BE2}:\overline{BE3}$, Abus28:29 (which contain the next two least significant address bits), BusWidth0:1, and W/ \overline{R} .

The bus can enter the Address state from the Idle state (see Section 3.2.1) or the Recovery state (see Section 3.2.4).

3.2.3 Wait/Data State

The Wait/Data state follows the Address state. In the Wait/Data state, the multiplexed address/data signals B0:31 transfer the data to be read or written.

If \overline{Ready} is sampled active, the bus is in the Data state. If \overline{Ready} is sampled inactive, the bus is in the Wait state.

The bus is device-paced; the processor waits for memory or an I/O device to make \overline{Ready} active to indicate that the data is ready for a read or write transfer. In burst mode, two to 16 data items (bytes, halfwords, or words) are read from or written to consecutive memory locations. The number of data items transferred depends on the bus width.

The \overline{Ready} signal is asserted at the end of each transfer. \overline{Ready} and \overline{BLast} are both asserted at the end of each access.

3.2.4 Recovery State

The Recovery state, which follows the Wait/Data state, allows devices attached to the bus to recover to a stable state.

When \overline{Ready} and \overline{BLast} are both sampled active during the same cycle, the bus enters the Recovery state. In the Recovery state, the multiplexed address/data signals (B0:31) retain the same values as in the most recent Wait/Data state for write accesses; for read accesses, B0:31 are hi-Z.

After the number of transfer recovery cycles specified by $BRCRn[TR]$, the state of the bus depends on whether a bus request is pending and, if so, whether the HoldReq signal is asserted. The bus enters the Idle state if no bus request is pending. If a bus request is pending and the HoldReq signal is not asserted, the bus enters the Address state. If a bus request is pending and HoldReq is asserted, the bus enters the Hold state.

3.2.5 Hold State

The bus is in the Hold state (an external bus master controls the bus) while the HoldAck signal is asserted. In the Hold state, the multiplexed address/data signals (B0:31) and all bus control signals are hi-Z.

After the HoldAck signal is deasserted, the bus enters the Idle state.

Section 3.11, “External Bus Master Interface,” on p. 3 -29, describes bus mastering in more detail.

3.3 Signals

The external bus signals communicate between the PPC401GF and external memory controllers and peripherals.

Chapter 12, “Signal Descriptions,” contains a table of signal descriptions ordered by signal name and a table of signals ordered by pin number.

3.3.1 ABus28:29 (Address Bits 28:29)

ABus28:29 outputs a partial unmultiplexed word address during the address cycle. This partial address is incremented with each assertion of \overline{Ready} during burst transfers on a 32-bit bus.

3.3.2 ALE (Address Latch Enable)

ALE is asserted at the beginning of the address cycle. ALE goes inactive before the data cycle begins.

ALE indicates the beginning of a bus access.

For an early implementation of the PPC401GF (Part No. PPC401GF-MA50C2), the polarity of the ALE signal was reversed and this signal was active low (\overline{ALE}). For all other PPC401GF chips, ALE is active high.

3.3.3 B0:31 (Multiplexed Address Bus Bits 0:31)

These are the multiplexed bus bits 0:31. These bits carry 32-bit physical addresses during the Address state. In the Address state, for convenience, these bits are referred to as A0:31. The mapping of A0:31 to B0:31 is shown in Figure 3-2.

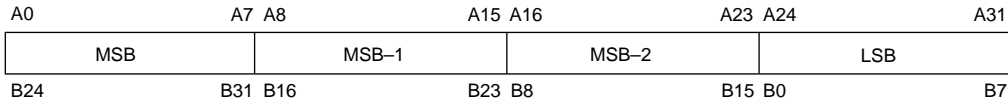


Figure 3-2. Mapping of Address Bits to Bus Bits

When the bus is in the Wait/Data state, B0:31 carry 8-, 16-, or 32-bit data, depending on the programmed bus width. In the Data state, read or write data is present on one or more contiguous bytes. During write operations, unused pins are driven to determinate values. During reads, unused pins are hi-Z. Determinate values are driven on B0:31 during the Idle state. Note that B4:7 (A28:31) are treated as size information.

For a four-beat burst ($BRCRn[MB] = 0$), size information is output on B6:7 (A30:31); for a continuous burst (up to 16 transfers, $BRCRn[MB] = 1$), size information is output on B4:7 (A28:31).

Table 3-1 illustrates how the value of $BRCRn[MB]$ and A28:31 determine the number of transfers (transfer size) in a bus access. Note that the transfer size does not necessarily equal a byte count. The number of bytes moved in a bus transaction equals the number of transfers times the bus width.

In Table 3-1, “xx” represents the address bits A28:29, which are not considered when $BRCRn[MB] = 0$. Because only two address bits ($\overline{BE0}/A31$ and $\overline{BE1}/A30$) are available when $BRCRn[MB] = 0$, the maximum transfer size is four.

Table 3-1. Transfer Size of a Bus Access

Transfer Size	B4:7 (A28:31)	
	$BRCRn[MB] = 0$	$BRCRn[MB] = 1$
1	xx00	0000
2	xx01	0001
2	xx10	0010
4	xx11	0011
8	N/A	0111
16		1111

The type of operation also affects the transfer size (the number of beats) and the amount of data transferred during a bus transaction. During cache line fills and flushes, more data is transferred than during loads/stores.

Table 3-2 shows how the value of $BRCRn[MB]$, the type of operation, and the resulting number of beats and transferred bytes.

Table 3-2. Beats and Bytes Transferred Depending on Operation and Burst Size

Bus Width		Loads/Stores BRCRn[MB] = 0 or BRCRn[MB] = 1	Cache Line Fills/Flushes	
Beats/Bytes Transferred			BRCRn[MB] = 0	BRCRn[MB] = 1
8-bit	Beats	4 maximum	4	16
	Bytes transferred	4 maximum	4	16
16-bit	Beats	2 maximum	4	8
	Bytes transferred	4 maximum	8	16
32-bit	Beats	1	4	4
	Bytes transferred	4 maximum	16	16

Table 3-3 illustrates the how B0:31 are used during writes in the Wait/Data state, for all bus widths. For reads, B0:31 are hi-Z and the external device drives these pins. In the table, shaded areas denote bytes that are ignored in the transfer; the labels of the ignored bytes indicate the data actually transferred. *An:m* indicates address information; *Dn* indicates a data byte.

Table 3-3. B0:31 Usage During the Wait/Data State

Bus Width	Transfer	B24:31	B16:23	B8:15	B0:7
8-Bit	Any transfer	A0:7	A8:16	A16:23	D0
16-Bit	Byte 0	A0:7	A8:16	D0	D0
	Byte 1	A0:7	A8:16	D1	D1
	Half Word 0	A0:7	A8:16	D1	D0
32-Bit	Byte 0	D3	D3	D3	D0
	Byte 1	D1	D1	D1	D1
	Byte 2	D2	D2	D2	D2
	Byte 3	D3	D3	D3	D3
	Half Word 0	D1	D0	D1	D0
	Half Word 1	D2	D2	D1	D1
	Half Word 2	D3	D2	D3	D2
	3-byte 0	D3	D2	D1	D0
	3-byte 1	D3	D2	D1	D0
	Word	D3	D2	D1	D0

If the programmed bus width is 8 or 16 bits, the BCU continues to drive the address on the unused B0:31 pins.

If the programmed bus width is 32 bits, the BCU duplicates the data being driven on write cycles (for transfers of one or two bytes) on the unused B0:31 pins. If the transfer is three bytes, all four bytes in the register are written, but the excluded byte is ignored.

3.3.4 $\overline{\text{BE0}}[\text{A31}]:\overline{\text{BE3}}$ (Byte Enables)

The byte enable outputs, $\overline{\text{BE0}}[\text{A31}]:\overline{\text{BE3}}$, select the bytes written in a memory write access or read during a memory read access. The usage of these lines varies, depending on the bus width programmed into the $\text{BRCRn}[\text{BW}]$ field associated with the bus control region. Table 3-4 illustrates the byte enable usage for different $\text{BRCRn}[\text{BW}]$.

Table 3-4. $\overline{\text{BE0}}:\overline{\text{BE3}}$ and Bus Width

$\text{BRCRn}[\text{BW}]$	$\overline{\text{BE0}}[\text{A31}]$ Usage	$\overline{\text{BE1}}[\text{A30}]$ Usage	$\overline{\text{BE2}}$ Usage	$\overline{\text{BE3}}$ Usage
10 (32-bit)	Enables B0:B7	Enables B8:B15	Enables B16:B23	Enables B24:B31
01 (16-bit)	Enables B0:B7	Becomes A30	Unused (high)	Enables B8:B15
00 (8-bit)	Becomes A31	Becomes A30	Unused (high)	Unused (high)

3.3.5 $\overline{\text{BLast}}$ (Burst Last)

This output signal indicates the last transfer of a bus access, whether burst or non-burst. $\overline{\text{BLast}}$ remains active as long as $\overline{\text{Ready}}$ is deasserted to introduce wait states on the external bus. $\overline{\text{BLast}}$ goes inactive when the last transfer of an external bus transaction is finished.

When $\overline{\text{BLast}}$ and $\overline{\text{Ready}}$ both go inactive, the bus leaves the Wait/Data state, ending a bus access.

3.3.6 BootClkSpd (Boot Clock Speed)

BootClkSpeed is sampled during reset to determine the internal clock speed and the bus clock speed.

Table 3-5. BootClkSpd and Bus Speed

BootClkSpd	Internal Clock	Bus Clock (MemClk)
0	1X input clock frequency	1X internal clock frequency
1	1/4X input clock frequency	1/4X internal clock frequency

The internal clock frequency is controlled by the Power Management Control Register (PMCR0); the Input/Output Configuration Register (IOCR) controls the bus clock frequency.

3.3.7 BootW (Boot ROM Width Select)

BootW is sampled, during and after the $\overline{\text{Reset}}$ pin is active, to determine the width of the boot ROM.

If BootW is tied to logic 0, an 8-bit boot bus width is assumed. If BootW is tied to logic 1, the boot bus width is assumed to be 32 bits. To obtain a 16-bit boot bus width, BootW is tied to the $\overline{\text{Reset}}$ pin.

3.3.8 $\overline{\text{BusError}}$ (Bus Error Input)

A logic 0, input to the $\overline{\text{BusError}}$ pin by an external device, signals that an error occurred on the external bus transaction. $\overline{\text{BusError}}$ is sampled only while $\overline{\text{Ready}}$ is asserted.

3.3.9 BusReq (Bus Request)

While HoldAck is active, BusReq is active when the PPC401GF has a bus operation pending and needs to regain control of the bus from an external bus master. BusReq is also active during bus operations.

3.3.10 BusWidth0:1 (Bus Width)

BusWidth0:1 indicate the width of a bus transaction.

Table 3-6. BusWidth0:1 and Bus Width

BusWidth0:1	Bus Width
00	8-bit
01	16-bit
10	32-bit
11	Reserved

3.3.11 HoldAck (Hold Acknowledge)

HoldAck outputs a logic 1 when the PPC401GF relinquishes the bus to an external bus master. The external bus master uses the HoldReq pin to request use of the bus.

3.3.12 HoldReq (Hold Request)

External bus masters place a logic 1 on the HoldReq pin to request the 401GF bus. When the HoldAck pin is logic 1, the PPC401GF has relinquished its bus to an external master. The external bus master deasserts HoldReq to relinquish the bus to the PPC401GF, which deasserts HoldAck during the following cycle.

3.3.13 $\overline{\text{Ready}}$

The assertion of the $\overline{\text{Ready}}$ signal indicates the end of a data transfer. If the transfer is a write, $\overline{\text{Ready}}$ indicates that the external device has the data. On a read, $\overline{\text{Ready}}$ indicates that the external device has provided the data and the PPC401GF should accept the data.

An external device can delay the presentation of Ready to insert externally-generated (device-paced) wait states into bus transactions.

Data is sampled on the rising edge of MemClk while $\overline{\text{Ready}}$ is asserted.

3.3.14 $\overline{\text{Reset}}$

The $\overline{\text{Reset}}$ signal can be used as an input and as an output.

As an input, the $\overline{\text{Reset}}$ pin is driven low for 1 ms to perform a system reset. The $\overline{\text{Reset}}$ signal, used as an output, can be driven by software or an external debug tool.

When a system reset occurs, the $\overline{\text{Reset}}$ pin outputs logic 0 for up to 2048 internal clock cycles.

3.3.15 W/\overline{R} (Write/Read)

When the PPC401GF is bus master, W/\overline{R} is an output that is low when data is read from memory or a peripheral and high when data is written to memory or a peripheral. W/\overline{R} is asserted during the Address state and remains asserted during the Wait/Data state.

3.4 External Memory Locations

The 4GB address space is divided into 16 bus control regions, each containing 256MB, as illustrated in Figure 3-3. Pairs of bus control regions (where $A0 = 0$ or 1) are associated with a Bus Region Control Register (BRCR0–BRCR7). Each BRCR controls certain characteristics of two separate 256MB blocks of external memory. Because only address bits $A1:30$ are considered, one of the two bus control regions associated with a BRCR is in “low” memory (the address range $0x0000\ 0000$ – $0x7FFF\ FFFF$); the other is in “high” memory (the address range $0x8000\ 0000$ – $0xFFFF\ FFFF$).

The PPC401GF provides real-mode storage attribute controls. Chapter 8, “Storage Control,” describes the storage attributes in more detail.

Storage attribute control registers (DCCR, DCWR, ICCR, SGR, and SLER) control the storage attributes, which control data and instruction accesses. Chapter 11, “Register Summary,” contains bit descriptions for these registers.

Each storage attribute control register contains 32 bits; each bit controls one of 32 storage attribute regions shown in Figure 3-3. Bit 0 of each register controls the lowest-order region, with ascending bits controlling ascending regions in memory. Each region, selected by address bits $A0:A4$, contains 128MB. The storage attributes in each storage attribute region are set independently.

Bus Regions	Bus Region Control Registers	Storage Attribute Control Register Bits	Storage Attribute Control Regions
0xFFFF FFFF	BRCR7	31	0xF800 0000
0xF000 0000		30	0xF000 0000
0xEFFF FFFF	BRCR6	29	0xE800 0000
0xE000 0000		28	0xE000 0000
0xDFFF FFFF	BRCR5	27	0xD800 0000
0xD000 0000		26	0xD000 0000
0xCFFF FFFF	BRCR4	25	0xC800 0000
0xC000 0000		24	0xC000 0000
0xBFFF FFFF	BRCR3	23	0xB800 0000
0xB000 0000		22	0xB000 0000
0xAFFF FFFF	BRCR2	21	0xA800 0000
0xA000 0000		20	0xA000 0000
0x9FFF FFFF	BRCR1	19	0x9800 0000
0x9000 0000		18	0x9000 0000
0x8FFF FFFF	BRCR0	17	0x8800 0000
0x8000 0000		16	0x8000 0000
0x7FFF FFFF	BRCR7	15	0x7800 0000
0x7000 0000		14	0x7000 0000
0x6FFF FFFF	BRCR6	13	0x6800 0000
0x6000 0000		12	0x6000 0000
0x5FFF FFFF	BRCR5	11	0x5800 0000
0x5000 0000		10	0x5000 0000
0x4FFF FFFF	BRCR4	9	0x4800 0000
0x4000 0000		8	0x4000 0000
0x3FFF FFFF	BRCR3	7	0x3800 0000
0x3000 0000		6	0x3000 0000
0x2FFF FFFF	BRCR2	5	0x2800 0000
0x2000 0000		4	0x2000 0000
0x1FFF FFFF	BRCR1	3	0x1800 0000
0x1000 0000		2	0x1000 0000
0x0FFF FFFF	BRCR0	1	0x0800 0000
0x0000 0000		0	0x0000 0000

Figure 3-3. Physical Address Map

If a region contains addresses that are used, the associated BRCR must be programmed.

Figure 3-4 describes the BRCR fields.

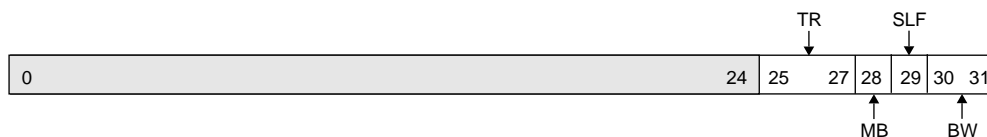


Figure 3-4. Bus Region Control Register (BRCR0–BRCR7)

0:24		Reserved	
25:27	TR	Transfer Recovery 000 Reserved 001 1-cycle 010 2-cycles 011 3-cycles 100 4-cycles 101 5-cycles 110 6-cycles 111 7-cycles	System reset value = 111
28	MB	Maximum Burst 0 Bursts have a maximum burst of four beats 1 Bursts have a maximum burst of 16 beats	System reset value = 0
29	SLF	Sequential Line Fills 0 Line fills are target word first 1 Line Fills are sequential	System reset value = 1
30:31	BW	Bus Width 00 8-bit bus 01 16-bit bus 10 32-bit bus 11 Reserved	System reset value = BootW

The TR field specifies the number of cycles during which the bus remains in the Recovery state following a bus access.

The MB field specifies whether bursts are limited to four or 16 beats. If $BRCRn[MB] = 0$ the maximum burst is four beats, and the maximum bus access size is one word. If $BRCRn[MB] = 1$, the maximum burst is 16 beats, and the maximum bus access size is four words. Note that in accesses to misaligned data items, bursts do not cross word boundaries.

The SLF field specifies how cache lines are filled. If $\text{BRCRn[SLF]} = 1$, line fills are sequential; the first word transferred is the first word of the cache line, whether or not the target address is the first word of the cache line. If $\text{BRCRn[SLF]} = 0$, line fills are target word first; the first word transferred is at the target address. The following sequential addresses to the end of the cache line are filled, then the first word of the cache line, then sequentially from the first word until the entire line is transferred.

The BW field specifies the bus width.

3.5 Bus Width after Reset

In general, the configuration of bank registers to allow the PPC401GF to support various device widths is a software task. At reset, however, enough configuration must be defined by hardware to allow boot code to run, typically from a small ROM in high memory. A default configuration, defined for all BRCRs, supports the slowest possible memory. The device width for this default configuration is determined at reset by the BootW input signal.

BootW is sampled while the $\overline{\text{Reset}}$ signal is active and again after $\overline{\text{Reset}}$ is deactivated to determine the width of the boot ROM. For an 8-bit boot ROM width, the BootW pin should be tied to 0. For a 32-bit boot ROM width, this pin should be tied to 1. For a 16-bit boot ROM width, the BootW pin should be tied to the $\overline{\text{Reset}}$ pin.

3.6 Access Priorities

Because the BCU provides a single pathway to external memory and peripherals, requests for BCU services can conflict with each other. For example, the fetcher, using the instruction cache (ICU), may attempt to read instructions from external memory at the same time the data cache writes data to external memory. Simultaneously, a bus master may request ownership of the bus.

The arbitration priority for access to BCU resources, from highest to lowest, follows:

1. External bus master
2. DCU, high-priority (cache-inhibited DCU reads; DCU line fills; stalled DCU writes; stalled DCU line flushes)

Only one data-side request can be presented to the BCU at a time.

3. ICU, high-priority (cache-inhibited ICU reads; ICU line fills)

Only one instruction-side request can be presented to the BCU at a time.

4. DCU, low priority (DCU stores; line flushes without CPU stalls)

3.7 Controlling the Bus Speed

The bus speed (MemClk output frequency) is controlled by the IOCR[EBS] field, as shown in Table 3-7.

Table 3-7. MemClk:Internal Clock Frequency Ratios

IOCR[EBS] Value	MemClk:Internal Clock Frequency Ratio
00	1:1
01	1:2
10	1:3
11	1:4

The internal clock frequency is controlled by the Power Management Control Register (PMCR0), which sets the internal chip frequency to a ratio of the clock input (crystal or oscillator). See Chapter 9, “Clock Generation and Power Management,” for more information about clock frequencies.

Care must be taken when changing PMCR0[CSC] or IOCR[EBS] while the PPC401GF is running. It is possible to change PMCR0[CSC] or IOCR[EBS] to values that result in bus speeds that are outside the operating speed range of the bus. To avoid this, pay attention to the order of operations. For example, to double the internal clock frequency while retaining the current bus speed, lower the bus speed using IOCR[EBS]. Then, raise the internal clock frequency and the bus speed using PMCR0[CSC].

When the bus speed changes, devices on the bus need time to adjust to the bus speed change. During this time, external bus activity is unreliable. Instructions that change PMCR0[CSC] and IOCR[EBS] must run out of the instruction cache. See Section 9.3, “Clock Generation and Bus Speed,” on p. 9-3, for an example of code that properly changes bus and internal clock speeds and allows enough time for external bus activity to resume.

3.8 Connecting to the PPC401GF Bus

Figure 3-5 shows the byte and bit connections for attaching memory or peripheral devices to the PPC401GF pins B0:31. Connect 8-bit devices to B0:7. For multibyte devices, connect the least significant bytes (LSB) and most significant bytes (MSB) as indicated to preserve the order from most significant bit (msb) to least significant bit (lsb) within each byte.

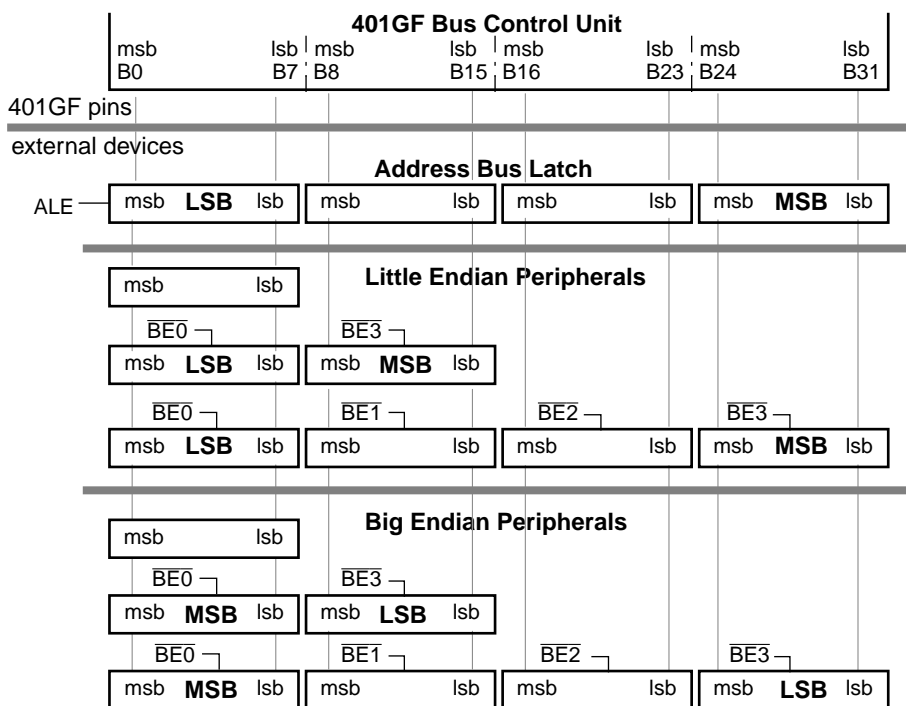


Figure 3-5. Connecting to the PPC401GF Bus

Table 3-8 shows the PPC401GF bus pins used for address output and data transfers. Address bits output by the PPC401GF are numbered A0:31, with A0 as the msb.

Table 3-8. Pins for Address Output and Data Transfers

Word address output by PPC401GF (see note)	A0 (msb):A7	A8:15	A16:23	A24:A27
PPC401GF pins used to output address	B24 (msb):B31	B16:23	B8:15	B0:B3
Data bytes transferred	Byte 0	Byte 1	Byte 2	Byte 3
PPC401GF pins used by data transfer	B0:7	B8:15	B16:23	B24:31

Note: Address bits A28:31 are provided on pins ABus28:29, $\overline{BE1}$ [A30], and $\overline{BE0}$ [A31], respectively.

Connect $\overline{BE0}$ and $\overline{BE1}$ to the lsb and second lsb address inputs of an external 8-bit device. Connect $\overline{BE1}$ to the second lsb address input of an external 16-bit device. Connect ABus28:29 to the fourth and third lsb address inputs of an external 32-bit device.

Figure 3-6 provides a simplified view of how the bus address and control signals might be connected in a system.

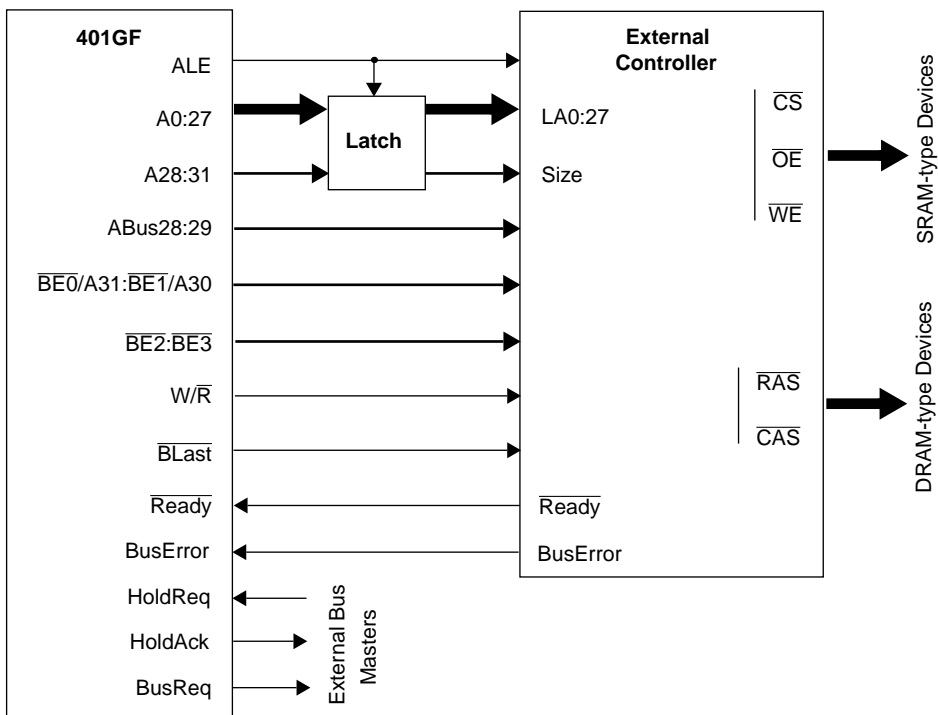


Figure 3-6. Address and Control Signal Connections

3.9 Bus Operations and Timings

Bus accesses either read data from or write data to external devices, such as controllers.

The transfers that comprise a bus access are device-paced. That is, the PPC401GF waits for an external device to indicate, using the $\overline{\text{Ready}}$ signal, that it has data available (for reads) or has received data (for writes).

The assertion of the $\overline{\text{Ready}}$ signal indicates the end of a data transfer. If the transfer is a write, $\overline{\text{Ready}}$ indicates that the external device has the data. On a read, $\overline{\text{Ready}}$ indicates that the external device has provided the data and the PPC401GF should accept the data.

An external device can delay the presentation of $\overline{\text{Ready}}$ to insert externally-generated (device-paced) wait states into bus transactions.

Data is sampled on the rising edge of MemClk while $\overline{\text{Ready}}$ is asserted.

Sampling of $\overline{\text{Ready}}$ occurs once per cycle during the Wait/Data state until $\overline{\text{Ready}}$ is active when sampled.

If $\overline{\text{Ready}}$ is asserted, it indicates:

- During a read access, that the data on the bus is valid for the PPC401GF to read.
- During a write access, that data out on the bus by the PPC401GF was written to its target.

The $\overline{\text{Ready}}$ input is synchronous.

3.9.1 Read and Write Transfers

Read and write transfers share many similarities. The major differences are:

- For reads, the data is presented by the external device; $\overline{\text{Ready}}$ indicates that the external device is presenting the data on the bus.
- For writes, the PPC401GF presents data at the beginning of the Wait/Data state; $\overline{\text{Ready}}$ indicates that the data was written to the external device.
- For reads, W/\overline{R} is active low; otherwise, the transfer is a write.

Keeping these differences in mind, the following description applies to both read and write transfers. Figure 3-7 through Figure 3-14 illustrate the signal usage and timings for a variety of read and write transfers.

As described in Section 3.1, “Terminology,” on p. 3 -3, a bus access is an event comprised of one or more read data transfers or write data transfers. In a bus access, the following sequence occurs:

1. The PPC401GF activates ALE and drives address (A0:27, Abus28:29), address/control ($\overline{\text{BE0}}/\text{A31}$, $\overline{\text{BE1}}/\text{A30}$) and control signals (A28:31, $\overline{\text{BE2}}:\overline{\text{BE3}}$, BusWidth0:1, W/\overline{R}) appropriately for the bus transaction. This is the Address state.
2. The bus enters the Wait/Data state. If $\overline{\text{Ready}}$ is sampled active, the bus is in the Data state. If $\overline{\text{Ready}}$ is sampled inactive, the bus is in the Wait state. The bus remains in the Wait state until the external device drives $\overline{\text{Ready}}$ active.
3. In the Data state, either the PPC401GF (for a write transfer) or the external device (for a read transfer) presents a data item on the bus. The maximum size of the data item is determined by BusWidth0:1; the bus request determines the actual size of the data item to be transferred. Note that the data transfer uses all available data signals. The byte enable signals ($\overline{\text{BE0}}/\text{A31}$, $\overline{\text{BE1}}/\text{A30}$, $\overline{\text{BE2}}:\overline{\text{BE3}}$) determine which data signals actually carry the data to be transferred.

During write transfers, any unused data signals are driven with address or duplicated data. During read transfers, the external device drives the bus.

After the data item is transferred, the PPC401GF samples $\overline{\text{BLast}}$ and $\overline{\text{Ready}}$ to determine whether the bus access is finished.

- If $\overline{\text{BLast}}$ and $\overline{\text{Ready}}$ are both active low during the same cycle, the bus access is finished.
- If $\overline{\text{Ready}}$ is active and $\overline{\text{BLast}}$ is not, another data transfer is pending. The bus calculates the address of the next target data item and either presents a data item or perceives that the external device has presented data. Then, that data item is transferred.

Data transfers continue until all of the data items associated with the access are transferred.

4. The PPC401GF enters the Recovery state. The bus remains in the Recovery state for the number of cycles specified by $\text{BRCRn}[\text{TR}]$, and then enters the Idle state or the Address state.

3.9.2 Timings

The timing diagrams in Figure 3-7 through Figure 3-14 illustrate typical bus transactions.

Figure 3-7 shows the timings for single-beat read and write word accesses on a 32-bit bus. In these accesses, the bus has no Wait state and one Recovery state ($BRCRn[TR] = 001$).

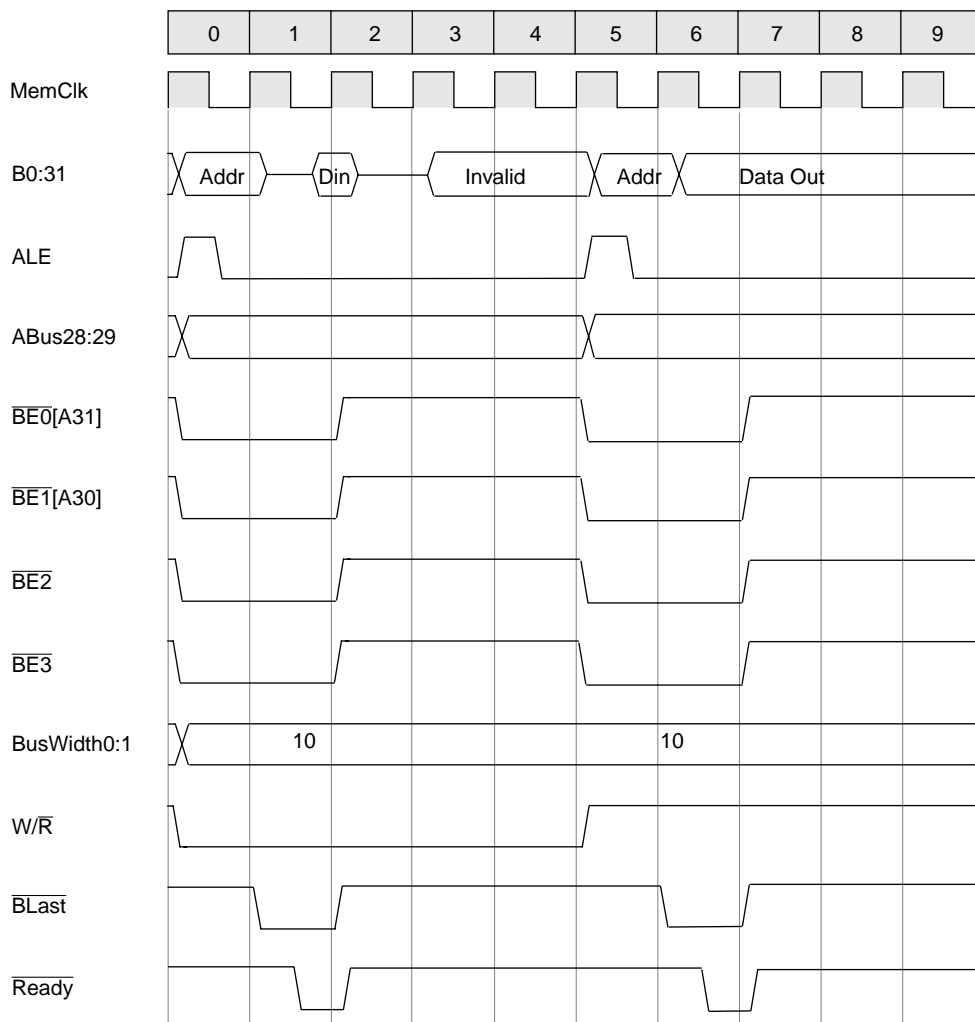


Figure 3-7. Single Read and Write Word Transfer Timings

Figure 3-8 shows the timings for a burst read of a word and a write of four words (cache line flush). The read access reads a word on a 16-bit bus; this is performed as a burst of two halfwords. Note that $\overline{\text{BLast}}$ and $\overline{\text{Ready}}$ are both active low after the two halfwords are transferred (in cycle 3).

Note that after the read access, the BusWidth0:1 signal changes. The write access flushes a cache line on a 32-bit bus; this is a burst of four words.

In these accesses, the bus has no Wait states and one Recovery state ($\text{BRCRn[TR]} = 001$).

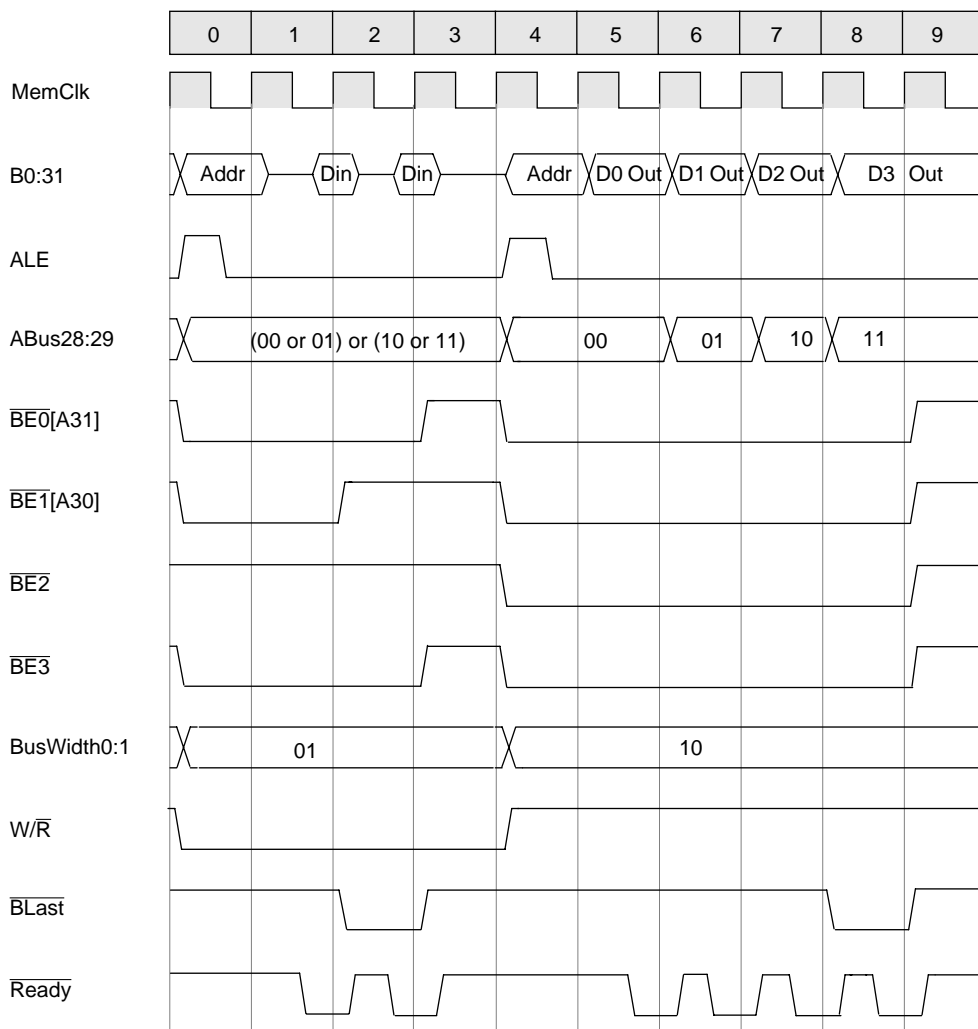


Figure 3-8. Burst Read of a Word; Burst Cache Line Flush Timings

Figure 3-9 shows the timings for a four-beat burst read of a word and a four-beat burst write of a word on an 8-bit bus.

In these accesses, the bus has no Wait states and one Recovery state ($\text{BRCRn}[\text{TR}] = 001$).

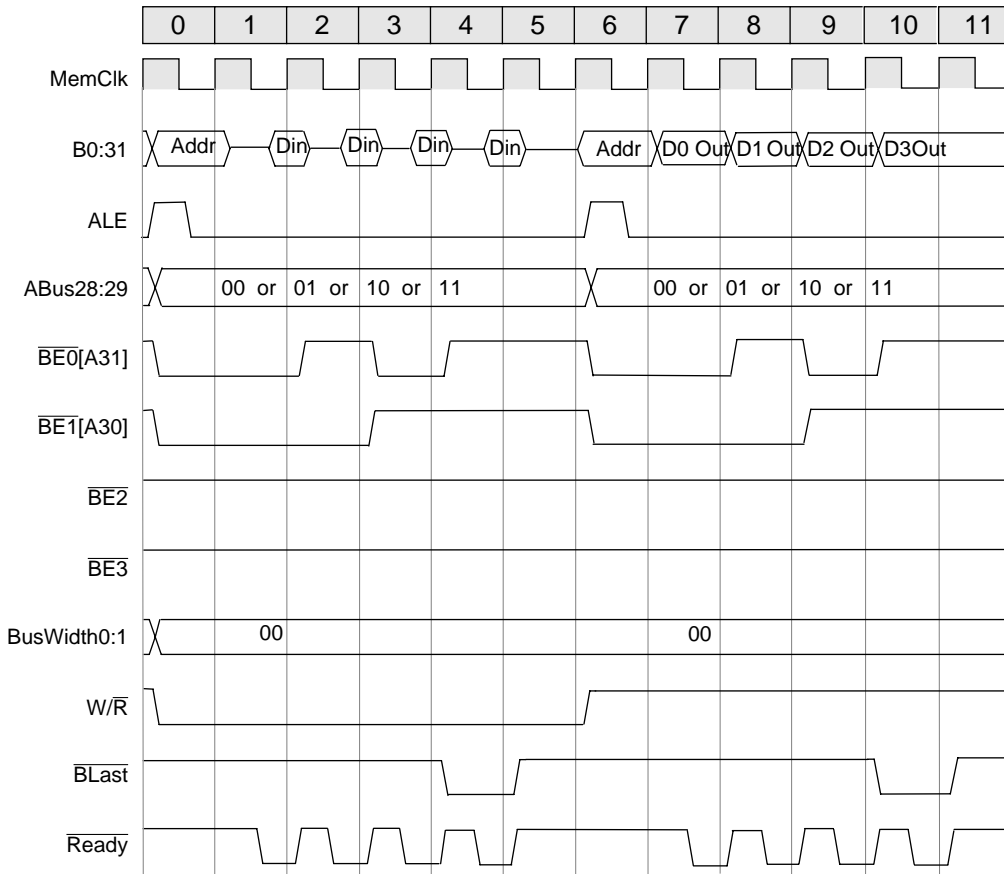


Figure 3-9. Burst Read and Write Word Transfer Timings

Figure 3-10 shows the timings for a continuous-burst read of a cache line (four words, transferred as 16 bytes) on an 8-bit bus.

In this access, the bus has no Wait states and one Recovery state ($\text{BR}CRn[\text{TR}] = 001$).

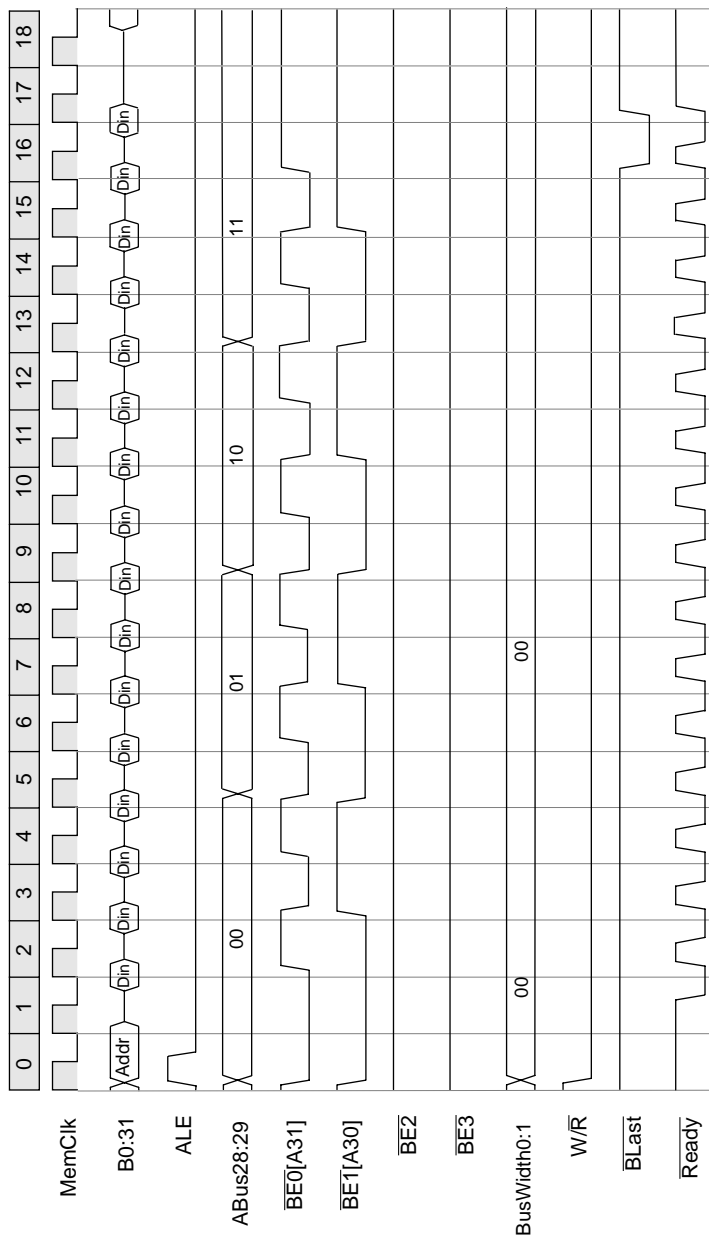


Figure 3-10. Cache Line Read; Continuous Burst Timings

Figure 3-11 shows the timings for a burst write of four words (a cache line flush) on a 32-bit bus.

In this transfer, the Wait state is 2-1-1-1; there is one Recovery state ($BRCRn[TR] = 001$).

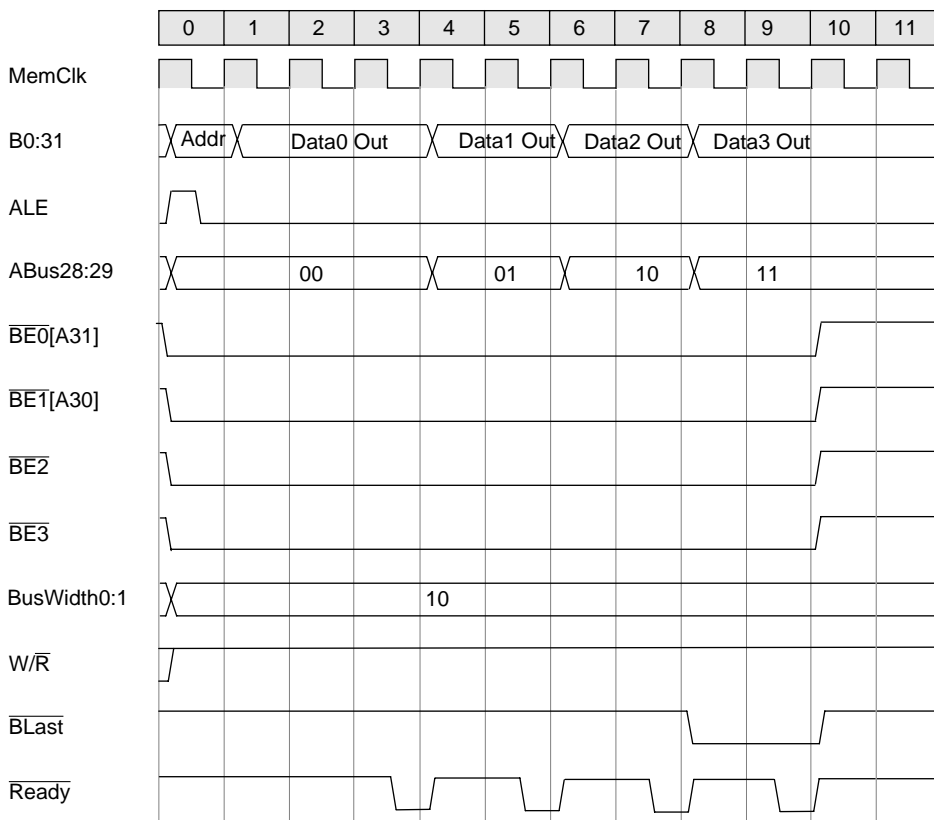


Figure 3-11. Cache Line Flush Burst Timings

Figure 3-12 shows the timings for a burst read of an unaligned halfword and a burst write of an unaligned halfword on a 16-bit bus.

The diagram illustrates the transfer of the bytes of a halfword whose alignment is one byte off the halfword boundary. Note that the transfer is a burst, despite the misalignment.

In these transfers, the Wait state is 1,0; there are two Recovery states ($BRCRn[TR] = 010$).

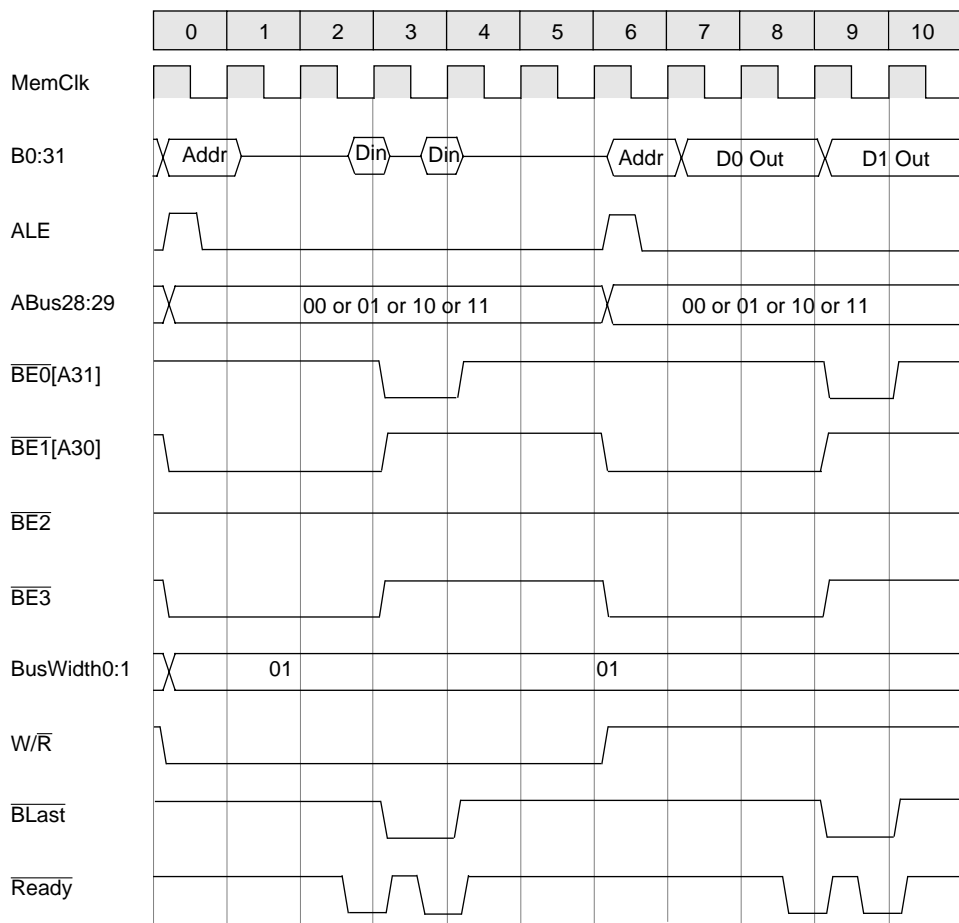


Figure 3-12. Burst Read/Write Unaligned Halfword Transfer Timings

Figure 3-13 shows the timing for a three-byte read and a three-byte write; both start at word addresses.

In the first access, the three bytes are read in two transfers over a 16-bit bus. In the first transfer, both bytes are read and treated as part of the three-byte access. In the second transfer, the low-order byte is read; the high-order byte is ignored.

In the second access, the three bytes are written in one transfer over a 32-bit bus.

In these transfers, the Wait state is 1, 0; there is one recovery state.

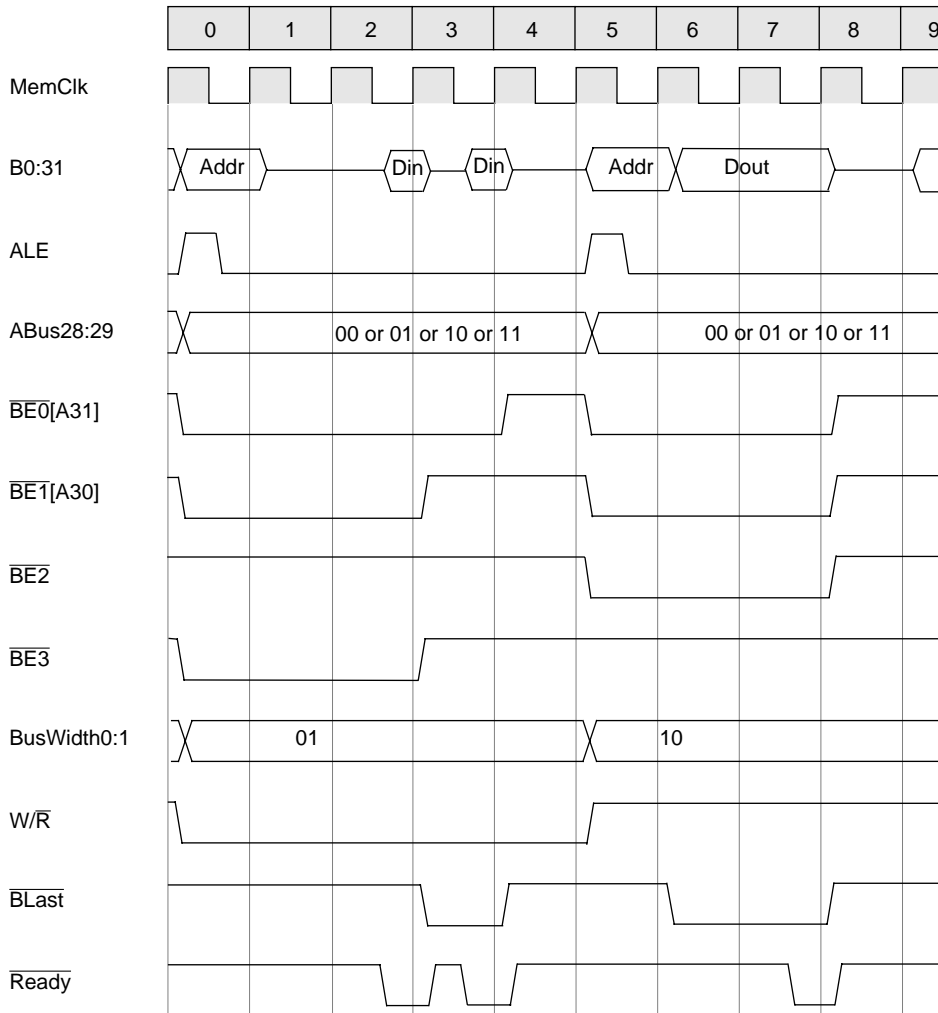


Figure 3-13. Three-byte Burst Reads

Figure 3-14 shows the timings for a read transaction on a 32-bit bus. The read, in this case, is of an eight-byte string that is unaligned with regard to word boundaries.

The address of the string points to byte 3 of a word. That byte is read during the first access; the other bytes are ignored. In the second access, the address is on a word boundary, and the entire word is read. In the third access, the address is also on a word boundary. However, only three bytes are read, starting at byte 0; byte 3 is ignored.

In this transaction, the bus has no Wait states; there is one Recovery state.

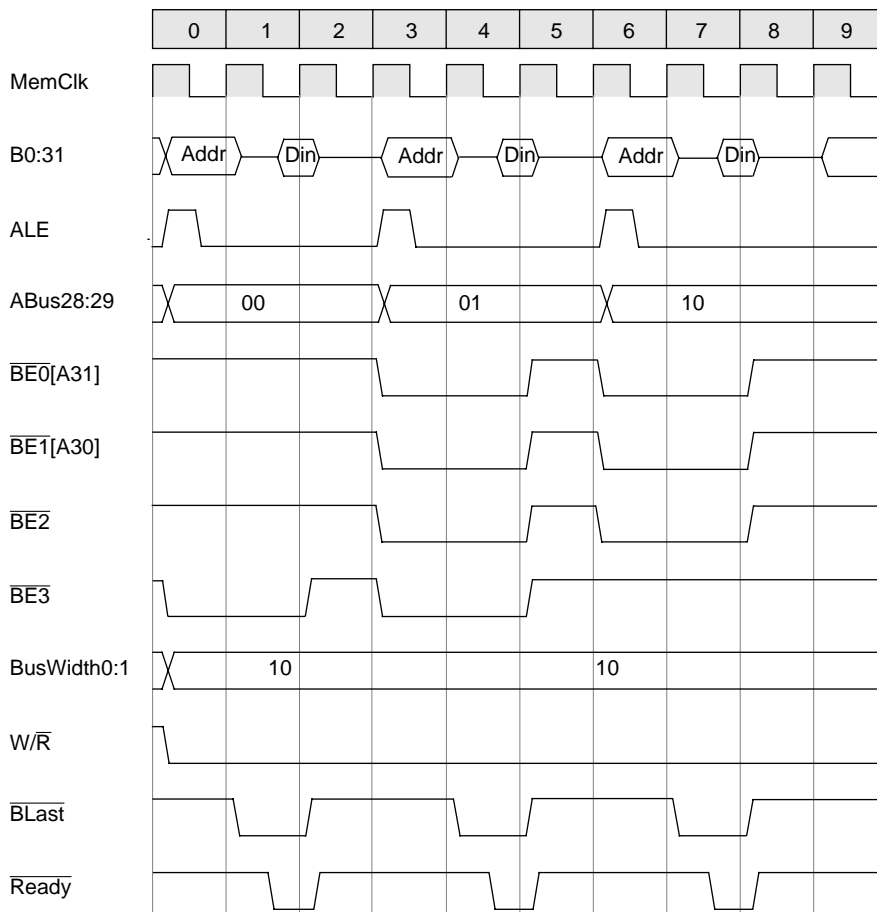


Figure 3-14. String Read Bus Request Across Word Boundary

3.10 Data Alignment and Bus Transfers

The 401GF BCU supports aligned and unaligned bus transfers. Alignment rules for loads and stores are based on address offsets from word boundaries. Table 3-9 through Table 3-13 describe the bus accesses resulting from aligned and unaligned loads and stores for various bus widths.

Table 3-9. Byte Load/Store Transactions

Address Offset from Word Boundary (in Bytes)	8-Bit Bus BRCR[BW] = 00	16-Bit Bus BRCR[BW] = 01	32-Bit Bus BRCR[BW] = 10
N/A	Byte access	Byte access	Byte access

Table 3-10. Halfword Load/Store Transactions

Address Offset from Word Boundary (in Bytes)	8-Bit Bus BRCR[BW] = 00	16-Bit Bus BRCR[BW] = 01	32-Bit Bus BRCR[BW] = 10
+0 (aligned)	2-byte burst	Halfword access	Halfword access
+1	2-byte burst	2-byte burst	Halfword access
+2 (aligned)	2-byte burst	Halfword access	Halfword access
+3	Byte access, byte access	Byte access, byte access	Byte access, byte access

Table 3-11. Three-byte Load/Store Transactions

Address Offset from Word Boundary (in Bytes)	8-Bit Bus BRCR[BW] = 00	16-Bit Bus BRCR[BW] = 01	32-Bit Bus BRCR[BW] = 10
+0 (aligned)	3-byte burst	Burst of one halfword and one byte	3-byte access
+1	3-byte burst	Burst of one byte and one halfword	3-byte access
+2	Not supported		
+3			

Table 3-12. Single-word Load/Store Transactions

Address Offset from Natural Boundary (in Bytes)	8-Bit Bus BRCR[BW] = 00	16-Bit Bus BRCR[BW] = 01	32-Bit Bus BRCR[BW] = 10
+0	4-byte burst	2-halfword burst	Word access
+1	3-byte burst, byte access	Burst of one byte and one halfword; byte access	3-byte access; byte access
+2	2-byte burst, 2-byte burst	Halfword access, halfword access	Halfword access, halfword access
+3	Byte access, 3-byte burst	Byte access; burst of one byte and one halfword	Byte access; 3-byte access

Table 3-13. Four-word Load/Store Transactions (Cache Line Fills/Flushes)

Address Offset from Natural Boundary (in Bytes)	8-Bit Bus BRCR[BW] = 00	16-Bit Bus BRCR[BW] = 01	32-Bit Bus BRCR[BW] = 10
+0	If BRCRn[MB] = 1: 16-byte burst If BRCRn[MB] = 0: Four accesses, each a 4-byte burst	If BRCRn[MB] = 1: 8-halfword burst If BRCRn[MB] = 0: Two accesses, each a 4-halfword burst	4-word burst
+1	Not supported		
+2			
+3			

3.11 External Bus Master Interface

The PPC401GF supports a shared external bus protocol that allows external bus masters to take control of the PPC401GF external bus. To relinquish the external bus, the BCU enters a Hold state. In the Hold state, most bus outputs are floated to a hi-Z value. The tri-stated signals are ABus28:29, ALE, B0:31, $\overline{BE0}[A31]$, $\overline{BE1}[A30]$, $\overline{BE2}$, $\overline{BE3}$, \overline{BLast} , and W/\overline{R} .

3.11.1 Shared External Bus Protocol

Three bus signals support the shared external bus protocol: HoldReq, HoldAck, and BusReq.

To request control of the bus, the external bus master asserts the HoldReq input. If the BCU is idle, the PPC401GF asserts the HoldAck output, to indicate that it has relinquished the bus, in the cycle following the cycle during which HoldReq is asserted. If the BCU is performing a processor operation when HoldReq is asserted, the BCU will assert HoldAck in the cycle following the last cycle of the current BCU operation. While HoldAck is asserted, the BCU is in the Hold state and the chip floats the tri-stated external bus signals. These signals remain at a hi-Z value until the PPC401GF deasserts HoldAck, which always occurs during the cycle following the cycle in which the HoldReq signal is deasserted.

If the PPC401GF has a bus operation pending while the external bus master controls the bus, the PPC401GF asserts the BusReq output to request control of the bus. An external arbiter can decide whether to return control of the bus to the BCU. The BusReq signal is asserted whenever the BCU is busy with an operation or one or more processor bus requests are pending.

To relinquish the bus, the external bus master deasserts the HoldReq input.

Figure 3-15 illustrates the external bus master interface timings.

Note: The bus master interface is *synchronous*. Signals must be presented with timing appropriate for the MemClk output from the PPC401GF. Refer to *PPC401GF Data Sheet* for setup and hold times.

3.11.2 External Bus Master Interface Timings

Figure 3-15 shows the timing for the Hold, HoldAck, and BusReq signals.

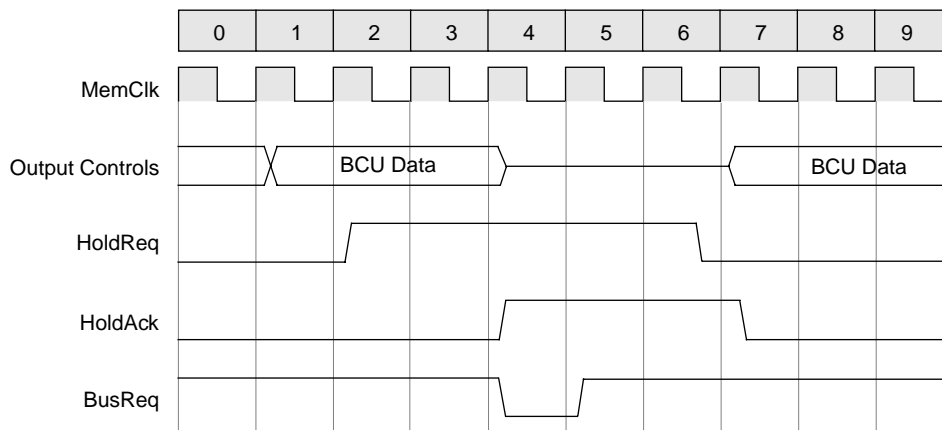


Figure 3-15. External Bus Interface Timings

4

Reset and Initialization

This chapter describes the three types of processor resets, the initial state of the processor after each type of reset, and the initialization code required to begin executing application code. Initialization of external system components or system-specific chip facilities may need to be performed in addition to the basic initialization code described in this chapter.

4.1 Core, Chip, and System Resets

The three different kinds of processor resets that can be performed are described below. Each type of reset may be generated by a debug tool, by the watchdog timer, or by a specific sequence of code. An external **Reset** signal also can initiate a system reset.

Although the PPC401GF performs the three kinds of processor resets, the effects of core and chip resets are identical. To determine which of the three types of reset occurred, the most-recent reset (MRR) field of the Debug Status Register (DBSR) can be examined.

Core reset	Resets the processor core, including the data and instruction caches.
Chip reset	Resets the processor core, including the data and instruction caches. This type of reset is provided in the IBM PowerPC 400 Series Embedded controllers as a means of resetting on-chip peripherals, and is provided on the PPC401GF for compatibility.
System reset	Resets the entire chip. The reset signal is driven active by the PPC401GF during system reset.

4.2 Reset Sequence

Systems must provide a power-on reset (POR) function to reset the PPC401GF before its first use. The POR must drive the **Reset** signal for at least 1 ms to ensure that the internal oscillator and clock generation circuitry can lock and provide clocks to the core and to external devices using MemClk output.

During POR, the PPC401GF drives **Reset** low for up to 2048 CPU clock cycles (2048 input clock cycles if the BootClkSpd pin is 0 or 8192 input clock cycles if BootClkSpd is 1). For system resets following POR, the PPC401GF drives **Reset** low for exactly 2048 CPU cycles. External circuitry attached to the **Reset** pin must tolerate that pin being driven by the PPC401GF.

$\overline{\text{Reset}}$ is an open-drain line, typically connected to a pull-up resistor. Systems with circuits that control $\overline{\text{Reset}}$ externally to the processor should not drive $\overline{\text{Reset}}$ inactive within the first 2048 clock cycles of a reset sequence, including system resets initiated internally. Otherwise, the system reset circuit contends with the processor on the $\overline{\text{Reset}}$ pin.

4.3 Processor State After Reset

Table 4-1 describes the processor configuration after a core, chip or system reset.

Table 4-1. Processor Configuration After Reset

Chip Resource	Core and Chip Reset Configuration	System Reset Configuration
Write-through (W) storage attribute	Unchanged	Unchanged (undefined at power-up)
Cacheability (I) storage attribute	Inhibited	Inhibited
Guarded (G) storage attribute	Guarded	Guarded
Endian (E) storage attribute	Big-Endian	Big-Endian
Watchdog Timer Reset	Disabled	Disabled
Wait state	Disabled	Disabled
Interrupts (external and critical)	Disabled	Disabled
Processor mode	Supervisor mode	Supervisor mode
Bus regions 0–7		
Bus Width	Unchanged	Set by BootW signal during most recent System Reset
Transfer Recovery	Unchanged	Seven cycles
Line Fills	Unchanged	Sequential line fills
Burst Mode	Unchanged	Maximum burst of four beats

4.4 Register Contents After A Reset

After a reset, the contents of the registers control the initial processor state. The initial register contents vary with the type of reset that occurred.

Chapter 11, “Register Summary,” contains descriptions of the registers referred to in Table 4-2 through Table 4-4.

After all resets, all fields of the Machine State Register (MSR) contain zeros. Table 4-2 shows how this affects chip operation.

Table 4-2. Contents of the Machine State Register After Reset

Register	Field	Core Reset	Chip Reset	System Reset	Comment
MSR	WE	0	0	0	Wait state disabled
	CE	0	0	0	Critical interrupts disabled
	ILE	0	0	0	Big Endian interrupt handlers
	EE	0	0	0	External interrupts disabled
	PR	0	0	0	Supervisor mode
	ME	0	0	0	Machine check exceptions disabled
	DE	0	0	0	Debug interrupts disabled
	LE	0	0	0	Big Endian

In general, the contents of special purpose registers (SPRs) are undefined after a core, chip, or system reset. Some SPRs retain the contents they had before a reset occurred.

Table 4-3 shows the contents of SPRs that are defined or unchanged after chip, core, and system resets.

Table 4-3. Contents of Special Purpose Registers After Reset

Register	Bits or Fields	Core Reset	Chip Reset	System Reset	Comment
DBCR	0:31	0x00000000	0x00000000	0x00000000	Debug mode and events disabled
DBSR	MRR	01	10	11	Most recent reset
DCCR	S0:S31	0x00000000	0x00000000	0x00000000	Data cache disabled
DCWR	W0:W31	0x00000000	0x00000000	0x00000000	Write-through is undefined at POR
ESR	0:31	0x00000000	0x00000000	0x00000000	No exception syndromes
ICCR	S0:S31	0x00000000	0x00000000	0x00000000	Instruction cache disabled
PVR	FAM MEM CORE CHIP	0x002 0x1 0x00 0x00	0x002 0x1 0x00 0x00	0x002 0x1 0x00 0x00	Processor version
SGR	G0:G31	0xFFFFFFFF	0xFFFFFFFF	0xFFFFFFFF	Storage is guarded
SLER	S0:S31	0x00000000	0x00000000	0x00000000	Storage is Big Endian
TCR	WRC	00	00	00	Watchdog timer reset disabled

Table 4-3. Contents of Special Purpose Registers After Reset (cont.)

Register	Bits or Fields	Core Reset	Chip Reset	System Reset	Comment
TSR	WRS	Copy of TCR[WRC]	Copy of TCR[WRC]	Copy of TCR[WRC]	Watchdog reset status
	PIS	Undefined	Undefined	Undefined	After POR
	FIS	Unchanged	Unchanged	Unchanged	If reset not caused by watchdog timer

Most Device Control Registers (DCRs) remain unchanged after core and chip resets. Except for the Bus Address Error Register (BEAR), all DCRs are initialized after a system reset. Table 4-4 shows the contents of the DCRs after chip, core, and system resets.

Table 4-4. Contents of Device Control Registers After Reset

Register	Bits or Fields	Core Reset	Chip Reset	System Reset	Comment
BEAR	0:31				Set on bus error
BESR0	IET DRWS DET	000 0 000	000 0 000	000 0 000	No data bus error
BRCR0–7	TR MB SLF BW	Unchanged Unchanged Unchanged Unchanged	Unchanged Unchanged Unchanged Unchanged	111 0 1 BootW	Transfer recovery is seven cycles Maximum burst of four beats Sequential line fills See Section 3.3.7, “BootW (Boot ROM Width Select),” on p. 3 -9 for more information
IOCR	CIL EIL MCA EBS	Unchanged Unchanged Unchanged Unchanged	Unchanged Unchanged Unchanged Unchanged	0 0 BootClkSpd BootClkSpd, BootClkSpd	CritInt pin is active low ExtInt pin is active low If BootClkSpd is 0, MemClk is aligned; otherwise MemClk is unaligned If BootClkSpd = 0, MemClk frequency is 1X the internal chip clock frequency If BootClkSpd = 1, MemClk frequency is 1/4X the internal chip clock frequency
PMCR0	DS NAP CDE TDE CSC	Unchanged Unchanged Unchanged Unchanged Unchanged	Unchanged Unchanged Unchanged Unchanged Unchanged	0 0 0 0 BootClkSpd, 1	Clock generation is enabled Oscillator and clock generation are enabled Internal chip clocks are running Timer clocks are running If BootClkSpd = 0, internal chip clocks run at 1X the clock input frequency If BootClkSpd = 1, internal chip clocks run at 1/4X the clock input frequency

4.5 Bus Control Unit Behavior During Reset

If $\overline{\text{Reset}}$ is asserted while the SysClk input operates, the PPC401GF enters the system reset state.

The assertion of $\overline{\text{Reset}}$ with SysClk non-switching puts all I/Os in the state defined in the PPC401GF data sheet. If SysClk remains non-switching, the PPC401GF does not enter the system reset state, with the exception of JTAG interface logic, which goes to the reset or idle state even though the SysClk remains off.

4.6 PPC401GF Initial Processor Sequencing

After any reset, the processor fetches the word at address 0xFFFF FFFC and attempts to execute it. Because the only memory configured immediately after reset is the upper 256MB bus region (0xF0000000–0xFFFFFFFF), the instruction at 0xFFFFFFFFFC must be a branch instruction. The branch must be to initialization code in the upper 256MB bus region.

The system must provide memory in the upper 256MB bus region. This memory must be either non-volatile or initialized by some mechanism, external to the processor, before a reset. The bus width (8-, 16-, or 32-bit) is controlled by the BootW signal.

There are no processor restrictions on when the initial bus region configurations can be modified after a reset. Memory devices in the system, however, may impose such restrictions.

4.7 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 minimum configuration tasks required to prepare the PPC401GF 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 in the following order to configure the processor resources:

- Initialize the SGR appropriately for guarded or unguarded storage; because all storage is initially guarded, this can improve performance because access to unguarded storage is generally faster.
- Invalidate the instruction cache
- Invalidate the data cache
- Configure the storage control attribute registers as appropriate for the bus regions:
 - Initialize the DCWR to select copy-back or write-through caching.
 - Initialize the DCCR and ICCR to configure cacheability.
 - Initialize the SLER to configure the storage Endianness.

- Program the BAlter CPU or bus clock speed
- Initialize processor registers as required by the system.
- 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.

4.7.1 BRCR Initialization

As shown in Table 4-1, hardware initializes BRCR0–BRCR7 in the same state. These registers need to be appropriately configured before code accesses memory and peripherals in the bus regions. Note that the boot region, BRCR7, is configured properly by hardware (see Section 4.6, “PPC401GF Initial Processor Sequencing,” on p. 4-6, for details). Note also that all other bus regions are configured identically. The bus width for all bus regions is initially controlled by the BootW pin, which the user configures to meet the requirements of the boot region. It is likely that other bus regions require a different bus width.

Some memory devices cannot be reprogrammed while they are being accessed. In such cases, code to perform the bus region reconfiguration must be contained entirely within the instruction cache or in another configured bank.

If software is reprogramming between states of a given BRCR that are valid for the attached hardware, no special precautions are required.

4.7.2 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 are executed. The example is presented in pseudo-code, function calls are named similarly to PPC401GF mnemonics where appropriate. Specific implementations may require different ordering of these sections to ensure proper operation.

```

/* _____ */
/*   PPC401GF Initialization Pseudo Code   */
/* _____ */
@0xFFFFFFFFC:          /* initial instruction fetch from 0xFFFFFFFFC */
    ba(init_code);      /* branch from initial address to initialization code*/

@init_code:

/* _____ */
/* Clear guarded attribute for performance and cacheability. */
/* _____ */

    mtspr(SGR, 0);

```

```

/*_____ */
/* 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(DCCR, d_cache_cacheability); /* enable D-cache */
isync;

/*_____ */
/* Program the BCRs */
/*_____ */

/*_____ */
/* Reconfigure bus region 7 if needed */
/*_____ */

{
    mtdcr(BRCR7, bus_region_7_configuration);
}

/*_____ */
/* Configure bus regions 0–6 if they are used */
/*_____ */

mtdcr(BRCR1, bus_region_0_configuration);
mtdcr(BRCR1, bus_region_1_configuration);

```

```

mtdcr(BRCR2, bus_region_2_configuration);
mtdcr(BRCR3, bus_region_3_configuration);
mtdcr(BRCR4, bus_region_4_configuration);
mtdcr(BRCR5, bus_region_5_configuration);
mtdcr(BRCR6, bus_region_6_configuration);

/* _____ */
/* Load operating system or application code, including exception handlers, into
/* memory. */
/*
/* The example assumes that the system and/or application code is loaded
/* immediately after the cache is initialized.
/* The example assumes that the source and destination regions do not overlap.
/* _____ */

while (not_done) /* repeat until all code has been loaded. */
{
    lmw(4, &code); /* load 4 word into 4 registers */
    stmw(4, &new_location); /* store 4 words to d-cache */
    dcbst(&new_location); /* store cache block to physical memory */
    icbi(&new_location); /* clear any obsolete code from i-cache */
    inc(&code); /* increment the code address by 4 words */
    inc(&new_location); /* increment the new_location addr by 4 words */
}
    sync(); /* allow all stores to complete */

/* _____ */
/* Store the last block (may have been unaligned) */
/* _____ */

dcbst(&new_location); /* store cache block to physical memory */
sync(); /* allow store to complete */
icbi(&new_location); /* clear any obsolete code from i-cache */
isync(); /* clear any obsolete code from queue */

/* _____ */
/* Establish machine state and on-chip facilities. This order MUST be followed.
/* _____ */

/* Configure: watchdog timer */
/* external interrupt polarity */
/*
mtdcr(IOCR,io_configuration); /* configure I/O */

```

```

/*_____*/
/* Alter CPU clock or bus clock speed at this time, if desired */
/*_____*/

/*_____*/
/* Set the exception vector prefix */
/*_____*/

mtspr(EVPR, prefix_addr);          /* initialize exception vector prefix */

/*_____*/
/* Initialize and configure timer facilities */
/*_____*/

mtspr(TBLO, 0);                    /* reset time base low first to avoid ripple */
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(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 Exceptions in the MSR */
/*_____*/
/*
/* Exceptions must be enabled immediately after timer facilities to avoid missing a
/* timer exception.
/*_____*/
/* The MSR also controls the user/supervisor mode and the wait state.
/* These must be initialized by the operating system or application code.
/*_____*/

mtmsr(machine_state);

/*_____*/
/* Initialize non-processor facilities should be performed at this time */
/*_____*/

/*_____*/
/* branch to operating system or application code */
/*_____*/
ba(&code_load_address);

```


5

Exceptions, Interrupts, and Timers

PPC401GF exceptions are generated by signals from external peripherals, instructions, the internal timer facility, debug events, and error conditions. Two external interrupt pins are provided in the PPC401GF, 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. 5-2.

Table 5-2, “Exception Vector Offsets,” on p. 5-6 lists the exceptions which are handled by the PPC401GF 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. Their polarity is controlled by the Input/Output Configuration Register (IOCR). Section 5.7.1, “Input/Output Configuration Register (IOCR),” on p. 5-20, describes the IOCR.

Section 5.5, “Machine Check Exceptions,” on p. 5-14, describes the device control registers (DCRs) related to machine checks: the Bus Address Error Register (BEAR) and the Bus Error Status Register 0 (BESR0).

Section 5.15, “Timer Facilities,” on p. 5-28, describes the timer facilities of the PPC401GF, 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 PPC401GF 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 PPC401GF, 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 PPC401GF 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 n 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 PPC401GF 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 PPC401GF, 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 pin	Critical	Active level on the CritInt pin
6	Watchdog timer—first time-out	Critical	Posting of an enabled first time-out of the watchdog timer in the TSR

Table 5-1. Exception-handling Priorities (cont.)

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[IJXE] = 1.
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. 5-22.
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 PPC401GF processes exceptions as non-critical and critical. Six exceptions are defined as *non-critical*: program exception, system call exception, alignment exception, an active level on the external interrupt (ExtInt) pin, 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 level on the critical interrupt (CritInt) pin, 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 PPC401GF 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	5-18
	Machine check—instruction	—	Critical	5-17
0x0500	External interrupt	Asynchronous precise	Non-critical	5-20
0x0600	Alignment	Synchronous precise	Non-critical	5-22
0x0700	Program	Synchronous precise	Non-critical	5-23
0x0C00	System Call	Synchronous precise	Non-critical	5-24
0x1000	PIT	Asynchronous precise	Non-critical	5-25
0x1010	FIT	Asynchronous precise	Non-critical	5-26
0x1020	Watchdog timer	Asynchronous precise	Critical	5-26

Table 5-2. Exception Vector Offsets (cont.)

Offset	Exception Type	Exception Class	Category	Page
0x2000	Debug exception—IC, BT, TIE, IA1, DR1, DW1	Synchronous precise	Critical	5-27
	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 PPC401GF. 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.

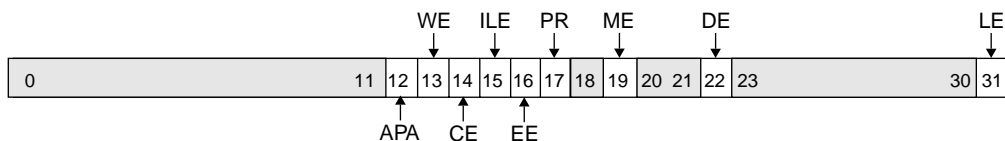


Figure 5-1. Machine State Register (MSR)

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.

Figure 5-1. Machine State Register (MSR) (cont.)

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.	
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.



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.
30:31		Reserved

Figure 5-2 shows the bit definitions for SRR1.

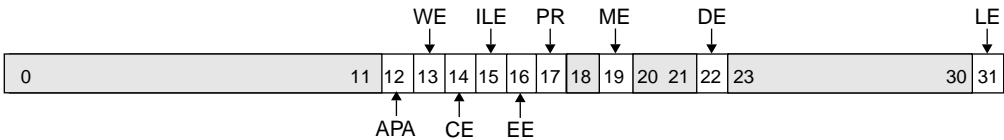


Figure 5-3. Save/Restore Register 1 (SRR1)

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 **rftci** 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 **mfspr** instruction. The contents of GPRs can be written to SRR2 and SRR3 using the **mtspr** instruction.

Figure 5-4 shows the SRR2 bit definitions.

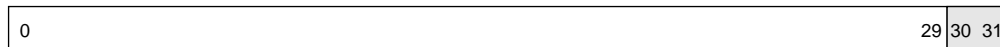


Figure 5-4. Save/Restore Register 2 (SRR2)

0:29		SRR2 receives an instruction address when a critical interrupt is taken; the Program Counter is restored from SRR2 when rftci executes.
30:31		Reserved

Figure 5-5 shows the bit definitions for SRR3.

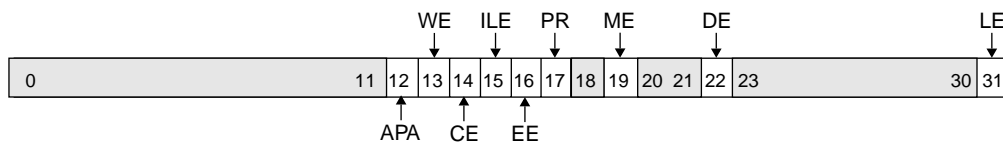


Figure 5-5. Save/Restore Register 1 (SRR1)

0:31	SRR3 receives a copy of the MSR when a critical interrupt is taken; the MSR is restored from SRR3 when rftci 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 **mf spr** instruction. The contents of a GPR can be written to EVPR using the **mt spr** instruction.

Figure 5-6 shows the EVPR bit definitions.

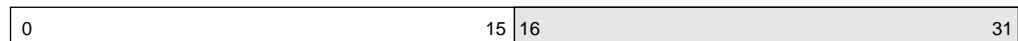


Figure 5-6. Exception Vector Prefix Register (EVPR)

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 5.5.3, “Instruction Machine Check Handling,” on p. 5-17, describes instruction machine checks. Section 5.6, “Data Storage Exceptions,” on p. 5-19, describes data storage exceptions. Section 5.9, “Program Exceptions,” on p. 5-23, 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 5.5.3, “Instruction Machine Check Handling,” on p. 5-17 describes why such resets are recommended.

The contents of the ESR can be written to a GPR using the **mfspir** instruction. The contents of a GPR can be written to the ESR using the **mtspir** instruction

Figure 5-7 shows the ESR bit definitions

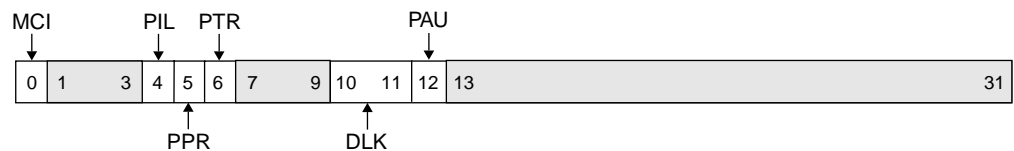


Figure 5-7. Exception Syndrome Register (ESR)

0	MCI	Machine check—instruction 0 Instruction machine check did not occur. 1 Instruction machine check occurred.
1:3		Reserved

Figure 5-7. Exception Syndrome Register (ESR)

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.
13:31		Reserved

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 (MSR[ME] = 1) set ESR[MCI] and clear the data storage and program exception bits.

If a machine check—instruction exception occurs but is disabled (MSR[ME] = 0), it sets ESR[MCI] but leaves the data storage and program exception bits alone. If a machine check—instruction exception occurs while MSR[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), ESR[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 **mf spr** instruction. The contents of a GPR can be written to the DEAR using the **mts pr** instruction.

Figure 5-8 shows the DEAR bit definitions.

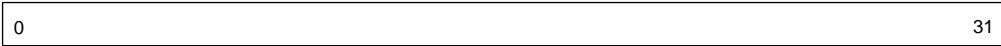


Figure 5-8. Data Exception Address Register (DEAR)

0:31	Address of Data Error (synchronous)
------	-------------------------------------

5.4 Critical Interrupt Exception

An external source requests a critical interrupt by driving the critical interrupt pin (CrittInt) 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. 5-26.

After detecting a critical interrupt, if no synchronous precise exceptions are outstanding, the PPC401GF 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 **rfti** 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 ← 0 ME ← unchanged ILE ← unchanged LE ← 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.

Two registers are used to handle bus errors resulting in machine checks: the BESR0 and the BEAR.

On instruction-side errors (bus errors on fetching), the address of the error is placed in the BEAR and the description of the error is placed in BESR0[IET], assuming that the registers are not locked by a previous machine check—data exception. Instruction-side error information placed in the BEAR and BESR0 because of bus errors during instruction fetching transactions is not locked in these registers, and is overwritten if a subsequent error occurs.

On data-side errors (bus errors on transactions involving the DCU), the address of the error is placed in the BEAR and the description of the error is placed in BESR0[DET], assuming that the registers are not locked by a previous machine check—data exception. This

information is locked (cannot be overwritten by subsequent error events) until software clears the BESR0.

When an instruction-side machine check interrupt occurs, the PPC401GF stores the address of the excepting instruction in SRR2. When a data-side machine check occurs, the PPC401GF 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 Bus Error Status Register 0 (BESR0)

The Bus Error Status Register 0 (BESR0) is a 32-bit DCR; its fields identify whether a machine check is instruction- or data-side. Software should clear the BESR0 before executing an **rfci** instruction. Clearing is performed by writing a word to the BESR0, using an **mtbcr** instruction, that has 1 in bit positions to be cleared and 0 in all other bit positions. The data written to the BESR0 is not direct data, but a mask; a 1 clears the bit and 0 has no effect.

The contents of the BESR0 can be written to a GPR using the **mtbcr** instruction.

Figure 5-9 shows the BESR0 bit definitions.

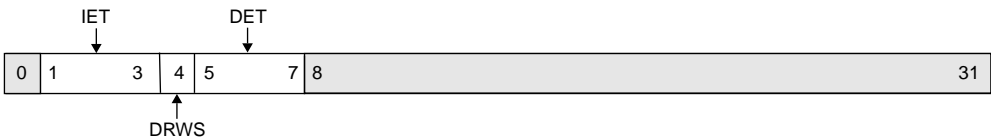


Figure 5-9. Bus Error Status Register 0 (BESR0)

0		Reserved
---	--	----------

Figure 5-9. Bus Error Status Register 0 (BESR0) (cont.)

1:3	IET	Instruction side Error Type 000 No Error 001 Reserved 010 Reserved 011 Reserved 100 Reserved 101 Reserved 110 Active level on the bus error input signal 111 Reserved	Reset value = 000
4	DRWS	Data Read/Write Status 0 Data side write error 1 Data side read error	Reset value = 0
5:7	DET	Data side Error Type 000 No Error 001 Reserved 010 Reserved 011 Reserved 100 Reserved 101 Reserved 110 Active level on the bus error input signal 111 Reserved	Reset value = 000
8:31		Reserved	

5.5.2 Bus Error Address Register (BEAR)

The BEAR is a 32-bit DCR that contains the address of the access on which a bus access error occurred. After the BEAR is loaded with an address, its contents are locked until the BESR0 is cleared.

The contents of the BEAR can be loaded into a GPR using a **mfdcr** instruction. The BEAR is read-only.

Figure 5-10 shows the BEAR bit definitions.

0	31
---	----

Figure 5-10. Bus Address Error Register (BEAR)

0:31	Address of Bus Error (asynchronous)
------	-------------------------------------

5.5.3 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] and BESR0[IET]. 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 ← 0 ILE ← unchanged LE ← ILE
PC	EVPR[0:15] 0x0200
ESR	MCI ← 1

Table 5-5. Register Settings during Machine Check—Instruction Exceptions (cont.)

BESR0	IET ← Written with a value specifying the type of machine check—instruction. See Figure 5-9 on p. 5-15 for more information.
-------	--

5.5.4 Data Machine Check Handling

When a machine check occurs on an data access, a machine check—data exception occurs. The address that caused the machine check is loaded into the BEAR. The first data-side error locks its address into the BEAR. If more data-side errors occur before the first is handled, their addresses are lost. Note that only a bus error can cause a machine check—data exception.

If enabled by MSR[ME], the processor reports the machine check—data exception by vectoring to the machine check handler (EVPR[0:15] || 0x0200), setting BESR0[DET]. Taking the vector automatically clears MSR[ME].

Programming Note: When a bus error occurs, the BEAR and BESR0 are set, along with a hidden register. If MSR[ME] = 1, the processor vectors to the machine check handler, and the hidden register is cleared.

After the code determines that a machine check—data exception occurred, the code should issue a **sync** instruction to ensure that a fill or flush with errors has completed. The code should also clear BESR0[DET] to unlock the BEAR and BESR0.

When MSR[ME] is again set to 1 using an **mtmsr** instruction or **rftci** instruction, the processor vectors to the machine check handler again, if any bus errors occurred for the flush or fill occurring after the hidden register was reset in response to the first machine check vector.

The second machine check exception is caused only by the behavior of the hidden register. In such a case, the interrupt handler finds that BESR0[DET] is not set, nor is ESR[MCI] set.

A significant difference between instruction- and data-side error handling must be appreciated in order to write a working interrupt handler for data-side machine checks. On the instruction-side, a valid/invalid flag accompanies an instruction fetched from memory. If a machine check—instruction exception occurs while obtaining that instruction, an (erroneous) instruction enters the instruction queue, but the instruction is always recognized as invalid. On the data-side, no valid/invalid flag is associated with data items. If a machine check—data exception occurs, invalid data can enter the data cache, but the data is not recognized as invalid. The exception handler for the machine check—data exception must take corrective action, such as invalidating the cache line, to protect against the use of invalid data.

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

Table 5-6. Register Settings during Machine Check—Data Exceptions (cont.)

MSR	WE, PR, CE, EE, ME, DE \leftarrow 0 ILE \leftarrow unchanged LE \leftarrow ILE
PC	EVPR[0:15] 0x0200
ESR	MCI \leftarrow 0
BEAR	Written with the address that caused the machine check
BESR0	DET \leftarrow Written with a value specifying the type of machine check—data. See Figure 5-9 on p. 5-15 for more information. DWRS \leftarrow 0 if write error DWRS \leftarrow 1 if read error

5.6 Data Storage Exceptions

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 PPC401GF 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 \leftarrow 0 CE, ME, DE \leftarrow unchanged ILE \leftarrow unchanged LE \leftarrow 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 pin (ExtInt). 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. 5-25 and Section 5.12, “Fixed Interval Timer (FIT) Exception,” on p. 5-26.

5.7.1 Input/Output Configuration Register (IOCR)

The IOCR is a DCR that allows users to program the external (critical and non-critical) interrupt pins as active positive or active negative polarity. The IOCR[CIL] bit (for the critical interrupt pin) and the IOCR[EIL] bit (for the non-critical interrupt pin) set the polarity of the external interrupt pins.

The contents of the IOCR can be written to a GPR using the **mfdcr** instruction. The contents of a GPR can be written to the IOCR using a **mtdcr** instruction.

Figure 5-11 shows the IOCR bit definitions.

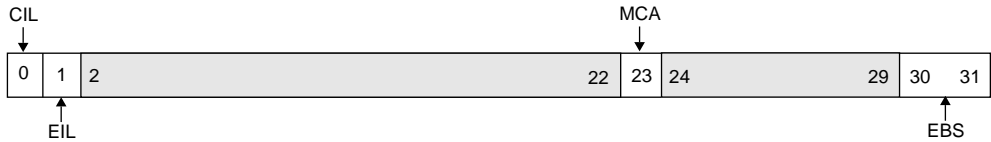


Figure 5-11. Input/Output Configuration Register (IOCR)

0	CIL	Critical Interrupt Level 0 Active low 1 Active high	System reset value = 0
1	EIL	External Interrupt Level 0 Active low 1 Active high	System reset value = 0
2:22		Reserved	
23	MCA	MemClk Alignment 0 MemClk is aligned. 1 MemClk is unaligned.	System reset value = BootClkSpeed input
24:29		Reserved	
2:25		Reserved	
26:27	RDM	Real-Time Debug Mode 00 Trace status outputs disabled 01 Program status and bus status 10 Program status and trace output 11 Reserved	
28:29		Reserved	
30:31	EBS	External Bus Speed 00 MemClk frequency is equal to the internal clock frequency. 01 MemClk frequency is 1/2 the internal clock frequency. 10 MemClk frequency is 1/3 the internal clock frequency. 11 MemClk frequency is 1/4 the internal clock frequency.	System reset value = BootClkSpeed input applied to both bits

5.7.2 External Interrupt Exception Handling

When MSR[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, MSR[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 MSR[LE] = 1 (in PowerPC Little Endian mode). Table 5-9 summarizes the instructions and conditions causing alignment exceptions.

Table 5-9. Alignment Exception Summary

PPC401GF 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
MSR[LE] = 1	dcbz	EA in non-cacheable or write-through storage
	lha, lhau, lhaux, lhax, lhbrx, lhz, lhz, lhz, lhzux, lhzx, sth, sthbrx, sthu, sthux, sthx	EA not halfword-aligned
	dcread, lwarx, lwbrx, lwz, lwz, lwz, 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 \leftarrow 0 CE, ME, DE \leftarrow unchanged ILE \leftarrow unchanged LE \leftarrow 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 PPC401GF 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 PPC401GF writes the address of the trap instruction into SRR0. See **tw** on p. 10-169 and **twi** on p. 10-173 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 PPC401GF 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

Table 5-13. Register Settings during System Call Exceptions (cont.)

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 PPC401GF timer facilities, see Section 5.15, “Timer Facilities,” on p. 5-28. The PIT is described in Section 5.15.2, “Programmable Interval Timer (PIT),” on p. 5-32.

If the PIT exception is enabled by TCR[PIE] and MSR[EE], the PPC401GF 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 PPC401GF 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 PPC401GF timer facilities, see Section 5.15, “Timer Facilities,” on p. 5-28. The FIT is described in Section 5.15.3, “Fixed Interval Timer (FIT),” on p. 5-34.

If the FIT exception is enabled by TCR[FIE] and MSR[EE], the PPC401GF 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→1 transition of the appropriate time-base bit.) The PPC401GF 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 ← 0 CE, ME, DE ← unchanged ILE ← unchanged LE ← ILE
PC	EVPR[0:15] 0x1010
TSR	FIS ← 1

5.13 Watchdog Timer Exception

For a general description of the PPC401GF timer facilities, see Section 5.15, “Timer Facilities,” on p. 5-28. The watchdog timer (WDT) is described in Section 5.15.4, “Watchdog Timer,” on p. 5-34.

If the WDT exception is enabled by TCR[WIE] and MSR[CE], the PPC401GF 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→1 transition of the appropriate time-base bit while TSR[ENW] = 1 and TSR[WIS] = 0.) The PPC401GF 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 PPC401GF 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

Table 5-17. SRR2 during Debug Exceptions

Debug Event	Address Saved in SRR2
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 PPC401GF 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 5-12 shows the relationship of these facilities and the base clock.

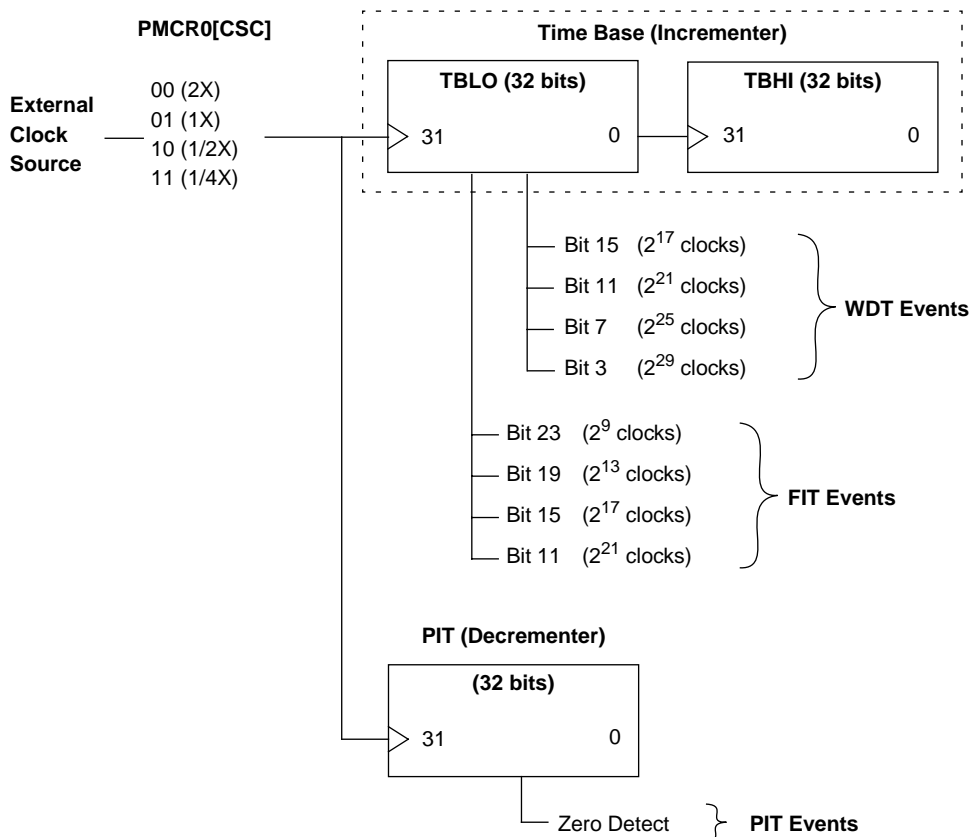


Figure 5-12. Relationship of Timer Facilities to the Base Clock

5.15.1 Time Base

The PPC401GF 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 **mfsprr** 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 **mfsprr** instructions. An **mtspr** to these registers is boundedly undefined.

The period of the 64-bit Time Base is approximately 23397 years for a 25 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 **mtspr**, have the same effect as transitions caused by normal incrementing of the time base.

Figure 5-13 illustrates TBHI(TBHU); Figure 5-13 illustrates TBLO(TBLU).

0	31
---	----

Figure 5-13. 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 5-14. 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 PPC401GF 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 **mfspr** by only one bit. The PowerPC Architecture allows an implementation to ignore this bit and handle **mftb** as **mfspr**. 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 **mtspr** with SPR numbers 0x11C and 0x11D.

For the PPC401GF:

- The PPC401GF 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 PPC401GF also allows read access (in privileged mode only) using the same register number as a write access.
- The PPC401GF provides user-mode and privileged-mode read access to the time base using **mfspr** with SPR numbers 0x3CD and 0x3CC. The PPC401GF provides privileged-mode-only read access via **mfspr** instructions with SPR numbers 0x3DD and 0x3DC. The PPC401GF does not implement **mftb**; an attempt to use it results in a program exception caused by the “illegal” opcode.
- The PPC401GF provides privileged-mode write access to the time base using **mtspr** instructions with SPR numbers 0x3DD and 0x3DC.

Figure 5-15 summarizes the differences between the time bases.:

Figure 5-15. Time Base Comparison

PowerPC Architecture				PPC401GF		
	Access Instructions	Register Number	Access Restrictions	Access Instructions	Register Number	Access Restrictions
Upper 32 bits	mftbu RT <i>Extended mnemonic for mftb RT,TBU</i>	0x10D	Read-Only	mftbhu 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	mftbhi RT <i>Extended mnemonic for mfspr RT,TBHI</i> mttbhi RS <i>Extended mnemonic for mtspr TBHI,RS</i>	0x3DC	Privileged; Read Privileged; Write
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 (TCR[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 ($TCR[ARE] = 0$), a decrement from a PIT count of simply causes a PIT content of 0.

- 2. $TSR[PIS]$ is set to 1.
- 3. If enabled ($TCR[PIE] = 1$ and $MSR[EE] = 1$), a PIT interrupt is taken. See Section 5.11, “Programmable Interval Timer (PIT) Exception,” on p. 5-25, for details of register behavior during a PIT interrupt.

The interrupt handler should use software to reset the $TSR[PIS]$ bit. This is done by using **mtspr** to write a word to the TSR having a 1 in $TSR[PIS]$ 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. 5-36). 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 5-16 illustrates the PIT.



Figure 5-16. 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 (33 Mhz Clock)
0, 0	23	2^9 clocks	15.52 μ sec
0, 1	19	2^{13} clocks	248.2 μ sec
1, 0	15	2^{17} clocks	3.972 msec
1, 1	11	2^{21} clocks	63.55 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. 5-26, 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	3.972 msec
0,1	11	2^{21} clocks	63.55 msec
1,0	7	2^{25} clocks	1.017 sec
1,1	3	2^{29} clocks	16.27 sec

If a WDT timeout occurs while $\text{TSR}[\text{WIS}] = 0$ and $\text{TSR}[\text{ENW}] = 1$, a WDT interrupt occurs if the interrupt is enabled by $\text{TCR}[\text{WIE}]$ and $\text{MSR}[\text{CE}]$. Section 5.13 describes register behavior during a WDT interrupt.

The interrupt handler must use software to reset the $\text{TSR}[\text{WIS}]$ bit. This is done by using **mtspr** to write a word to the TSR having a 1 in $\text{TSR}[\text{WIS}]$ 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 $\text{TSR}[\text{WIS}] = 1$ and $\text{TSR}[\text{ENW}] = 1$, a hardware reset occurs if enabled by a non-zero value of $\text{TCR}[\text{WRC}]$. The assumption is that $\text{TSR}[\text{WIS}]$ 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 $\text{TCR}[\text{WRC}]$ is set to a non-zero value, it cannot be reset by software. This prevents errant software from disabling the WDT reset capability.

Figure 5-17 describes the watchdog state machine. In the figure, numbers in parentheses refer to descriptions of operating modes that follow the table.

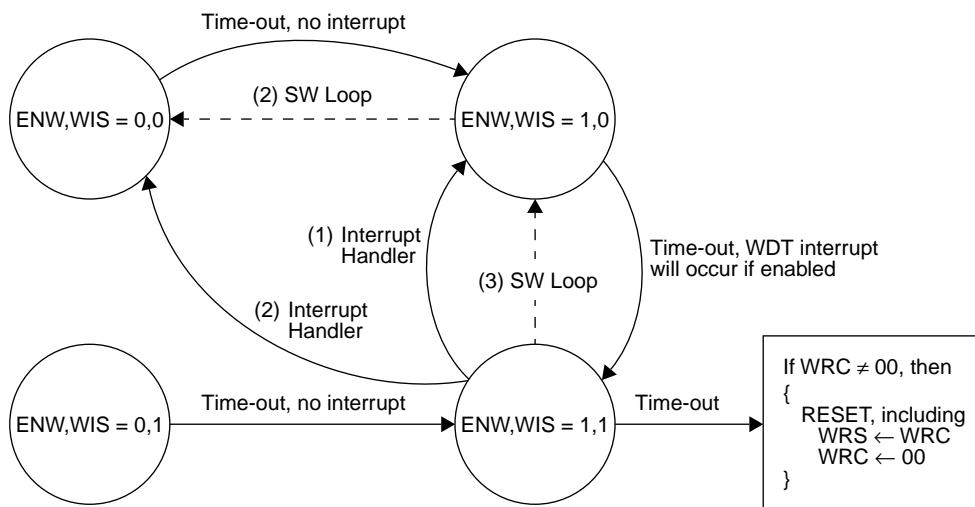


Figure 5-17. Watchdog Timer State Machine

Enable Next WDT $\text{TSR}[\text{ENW}]$	WDT Status $\text{TSR}[\text{WIS}]$	Action when timer interval expires
0	0	Set enable next watchdog ($\text{TSR}[\text{ENW}] = 1$).
0	1	Set $\text{TSR}[\text{ENW}] = 1$.
1	0	Set the watchdog interrupt status ($\text{TSR}[\text{WIS}] = 1$). If $\text{TCR}[\text{WIE}] = 1$ and $\text{MSR}[\text{CE}] = 1$, then interrupt.

Figure 5-17. Watchdog Timer State Machine

Enable Next WDT TSR[ENW]	WDT Status TSR[WIS]	Action when timer interval expires
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 **mfspir** 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.

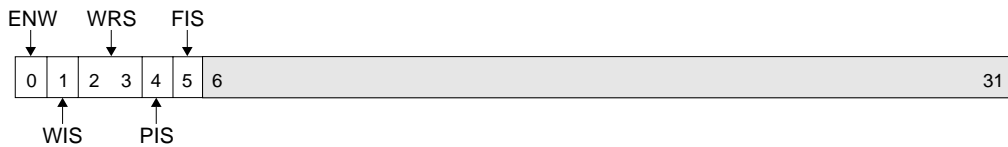


Figure 5-18. Timer Status Register (TSR)

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. 5-34.
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

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.

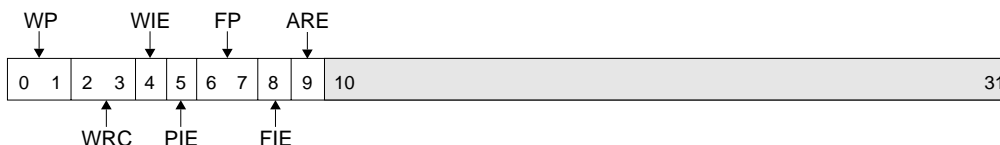


Figure 5-19. Timer Control Register (TCR)

5

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	

6

Cache Operations

The PPC401GF 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.

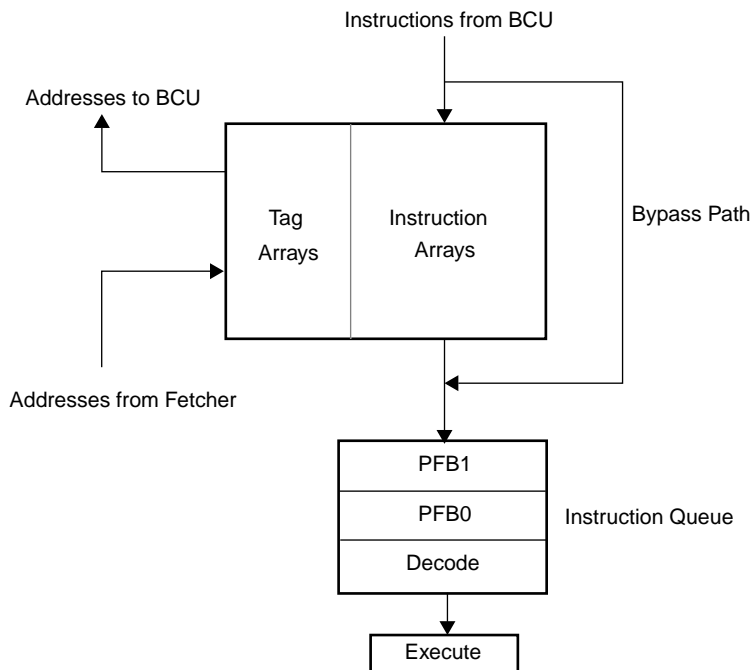


Figure 6-2. Instruction Flow

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. Loading order is controlled by a Bus Region Control Register (BRCR0–BRCR7) field, $\text{BRCR}_n[\text{SLF}]$. 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 from the BCU, 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 8.2.3, “Instruction Cache Cacheability Register (ICCR),” on p. 8-8). If ICCR[Sn] = 1, caching is enabled for the specified region.

The performance of the PPC401GF is significantly lower while executing in cache-inhibited regions.

6

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. Loading order is controlled by BRCR_n[SLF]. 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 from the BCU, 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 8.2.1, “Data Cache Write-through Register (DCWR),” on p. 8-4). 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 8.2.2, “Data Cache Cacheability Register (DCCR),” on p. 8-6). If DCCR[S_n] = 1, caching is enabled for the specified region.

The performance of the PPC401GF 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 10, “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 PPC401GF, 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 PPC401GF 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 PPC401GF, 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.

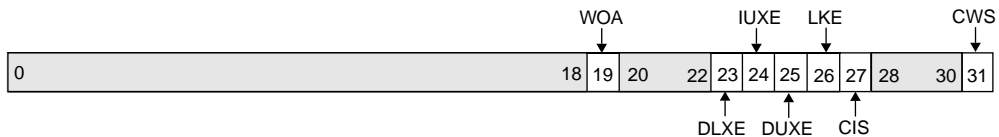


Figure 6-4. Cache Debug Control Register (CDBCR)

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 **mtcdbcr** extended mnemonic is not context- or execution-synchronizing. Software must place appropriate synchronization instructions before and after an **mtcdbcr** 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, 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
- 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 external bus 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 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. The priority signal tells the external bus that the DCU requires immediate service, and is valid only when the data cache is requesting access to the external bus. 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 external bus.	1
dcbf	0	Any cache hit.	0
dcbf/dcbst	0	Load non-cache when requesting access to the external bus.	1
dcbf/dcbst	0	Any line fill when dcbf/dcbst requests access to the external bus.	1

Table 6-1. Priority Changes With Different Data Cache Operations

Instruction	Priority	Next Instruction	Priority
dcbf/dcbst	0	dcbf/dcbst when the first dcbf/dcbst requests access to the external bus.	1
dcbt	0	Any cache hit.	0
dcbt	0	Any other cache operation when a line fill was not accepted on the external bus.	1
dcbi/dccci/dcbz	0	N/A	N/A

6.6.3 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.”

The PPC401GF debug facilities include 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 the 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.

Logic analyzers from Hewlett Packard and Tektronix support a PPC401GF disassembler.

7.2 Debug Modes

There are two debug modes in the PPC401GF; 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, on a processor board.

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 PPC401GF 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 the external $\overline{\text{Halt}}$ 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 $\overline{\text{Halt}}$ 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	The external $\overline{\text{Reset}}$ 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 (DAC1). 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.

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.

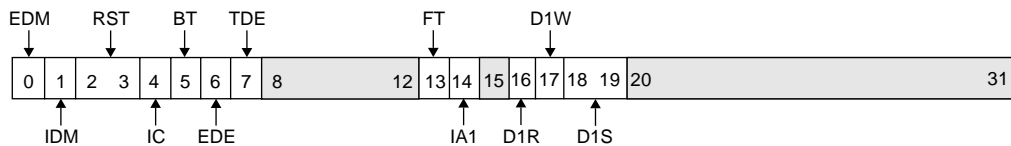


Figure 7-1. Debug Control Register (DBCR)

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.	
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	

Figure 7-1. Debug Control Register (DBCR) (cont.)

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 Exact address compare 01 Ignore the least significant bit (lsb) Byte within halfword address compare 10 Ignore the two lsbs Byte within word address compare 11 Ignore the four lsbs 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.

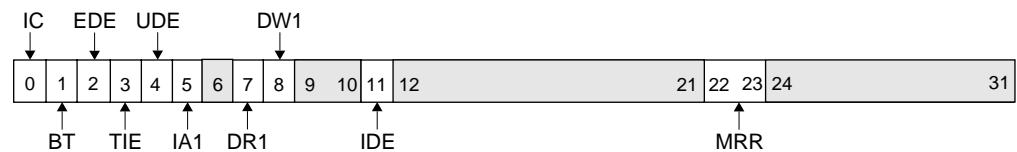


Figure 7-2. Debug Status Register (DBSR)

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
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

Figure 7-2. Debug Status Register (DBSR) (cont.)

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 (DAC1)

The PPC401GF 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 (DAC1)

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 PPC401GF 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 PPC401GF.

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 PPC401GF 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 PPC401GF processor provides a JTAG debug port to support hardware and software test and debug. 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. It does not implement the optional $\overline{\text{TRST}}$ 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. At 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. 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 PPC401GF to ensure signal integrity.

Note that position 14 does not contain a pin.

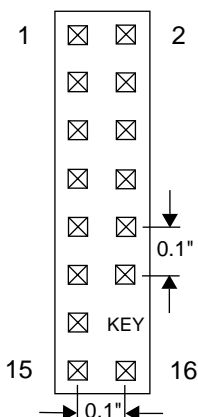


Figure 7-5. JTAG Connector (top view) Physical Layout

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		NC	Reserved
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 PPC401GF 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 3	0011	Private
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 Chain

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 the 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 BSDL for the PPC401GF, which follows, is useful to printed circuit board design and test engineers for the purposes described in the IEEE 1149.1 standard. The BSDL, and conformance of the chip to the IEEE 1149.1 standard, were verified by TestBench, an IBM EDA tool.

```
-- TestBench generated BSDL May 21 09:10 1996
-- TestBench verified chip compliance May 21 10:35 1996
-- TestBench is a registered trademark of IBM
-- Boundary Scan Description Language (BSDL) for
-- PowerPC 401 GF Embedded Controller
--
```

```
entity PPC401GF is
  generic (PHYSICAL_PIN_MAP : string := "TQFP_80");
```

```
port (
  ABus28:      out      bit;
  ABus29:      out      bit;
  ALE:         out      bit;
  B0:          inout    bit;
  B1:          inout    bit;
  B2:          inout    bit;
  B3:          inout    bit;
  B4:          inout    bit;
  B5:          inout    bit;
  B6:          inout    bit;
  B7:          inout    bit;
  B8:          inout    bit;
  B9:          inout    bit;
```

```

B10:      inout  bit;
B11:      inout  bit;
B12:      inout  bit;
B13:      inout  bit;
B14:      inout  bit;
B15:      inout  bit;
B16:      inout  bit;
B17:      inout  bit;
B18:      inout  bit;
B19:      inout  bit;
B20:      inout  bit;
B21:      inout  bit;
B22:      inout  bit;
B23:      inout  bit;
B24:      inout  bit;
B25:      inout  bit;
B26:      inout  bit;
B27:      inout  bit;
B28:      inout  bit;
B29:      inout  bit;
B30:      inout  bit;
B31:      inout  bit;
BE0_A31:  out    bit;
BE1_A30:  out    bit;
BE2:      out    bit;
BE3:      out    bit;
BLast:    out    bit;
BootClkSpd: in    bit;
BootXtal: in    bit;
BootW:    in    bit;
BusError: in    bit;
BusReq:   out    bit;
BusWidth0: out    bit;
BusWidth1: out    bit;
CritInt:  in     bit;
ExtInt:   in     bit;
GND1:     linkage bit;
GND2:     linkage bit;
GND3:     linkage bit;
GND4:     linkage bit;
GND5:     linkage bit;
GND6:     linkage bit;
GND7:     linkage bit;
GND8:     linkage bit;
GND9:     linkage bit;

```

```

Halt:          in      bit;
HoldAck:       out     bit;
HoldReq:       in      bit;
MemClk:        out     bit;
Ready:         in      bit;
Reset:         inout   bit;
TCK:           in      bit;
TDI:           in      bit;
TDO:           out     bit;
TestA:         in      bit;
TMS:           in      bit;
VDD1:          linkage bit;
VDD2:          linkage bit;
VDD3:          linkage bit;
VDD4:          linkage bit;
VDD5:          linkage bit;
VDD6:          linkage bit;
VDD7:          linkage bit;
VDD8:          linkage bit;
W_R:           out     bit;
XTAL1:         in      bit;
XTAL2:         out     bit
);

use STD_1149_1_1994.all;

attribute COMPONENT_CONFORMANCE of PPC401GF : entity is
  "STD_1149_1_1993";

attribute PIN_MAP of PPC401GF : entity is PHYSICAL_PIN_MAP;

constant TQFP_80: PIN_MAP_STRING :=
  " TDO:          1 " &
  " HoldAck:      2 " &
  " XTAL1:        3 " &
  " XTAL2:        4 " &
  " MemClk:       5 " &
  " VDD1:         6 " &
  " BootXtal:     7 " &
  " TDI:          8 " &
  " BootW:        9 " &
  " GND1:        10 " &
  " HoldReq:     11 " &
  " Ready:       12 " &
  " BusError:    13 " &

```



```

" ExtInt:          14 " &
" CritInt:         15 " &
" Halt:           16 " &
" TMS:            17 " &
" TCK:            18 " &
" BootClkSpd:     19 " &
" TestA:          20 " &
" BLast:          21 " &
" BusReq:         22 " &
" W_R:            23 " &
" BE0_A31:        24 " &
" BE1_A30:        25 " &
" VDD2:           26 " &
" BE2:            27 " &
" BE3:            28 " &
" B7:             29 " &
" GND2:           30 " &
" VDD3:           31 " &
" GND3:           32 " &
" B6:             33 " &
" B5:             34 " &
" B4:             35 " &
" B3:             36 " &
" B2:             37 " &
" B1:             38 " &
" B0:             39 " &
" B15:            40 " &
" B14:            41 " &
" B13:            42 " &
" GND4:           43 " &
" VDD4:           44 " &
" B12:            45 " &
" B11:            46 " &
" B10:            47 " &
" B9:             48 " &
" B8:             49 " &
" GND5:           50 " &
" VDD5:           51 " &
" B23:            52 " &
" B22:            53 " &
" B21:            54 " &
" B20:            55 " &
" B19:            56 " &
" VDD6:           57 " &
" GND6:           58 " &

```

```

" B18:          59 " &
" B17:          60 " &
" B16:          61 " &
" B31:          62 " &
" B30:          63 " &
" B29:          64 " &
" B28:          65 " &
" B27:          66 " &
" B26:          67 " &
" B25:          68 " &
" GND7:         69 " &
" VDD7:         70 " &
" GND8:         71 " &
" B24:          72 " &
" Reset:        73 " &
" VDD8:         74 " &
" GND9:         75 " &
" ALE:          76 " &
" ABus28:       77 " &
" ABus29:       78 " &
" BusWidth0:    79 " &
" BusWidth1:    80 " ;

```

```

attribute TAP_SCAN_CLOCK of TCK: signal is (10.0e6,BOTH);
attribute TAP_SCAN_IN   of TDI: signal is true;
attribute TAP_SCAN_MODE of TMS: signal is true;
attribute TAP_SCAN_OUT  of TDO: signal is true;
-- TAP_SCAN_RESET not present

```

```

attribute COMPLIANCE_PATTERNS of PPC401GF: entity is
    "(XTAL1,TestA,Reset) (001)";

```

```

attribute INSTRUCTION_LENGTH of PPC401GF: entity is 4;

```

```

attribute INSTRUCTION_OPCODE of PPC401GF: entity is
    "EXTEST(0000),SAMPLE(0001),"&
    "JTAG3(0011),JTAG5(0101),JTAG7(0111),JTAGB(1011),"&
    "BYPASS(1111)";

```

```

attribute INSTRUCTION_CAPTURE of PPC401GF: entity is "0001";

```

```

attribute INSTRUCTION_PRIVATE of PPC401GF: entity is
    "JTAG3,JTAG5,JTAG7,JTAGB";

```

```

attribute BOUNDARY_LENGTH of PPC401GF: entity is 57;

```

```

-- cell type portname      function safe cntl dis res
attribute BOUNDARY_REGISTER of PPC401GF: entity is
" 0 (BC_1, *,              control, 0          ),"&
" 1 (BC_1, *,              control, 0          ),"&
" 2 (BC_1, BusWidth1,      output3, 0,      0, 0, Z),"&
" 3 (BC_1, BusWidth0,      output3, 0,      0, 0, Z),"&
" 4 (BC_1, ABus29,         output3, 0,      0, 0, Z),"&
" 5 (BC_1, ABus28,         output3, 0,      0, 0, Z),"&
" 6 (BC_1, ALE,            output3, 0,      0, 0, Z),"&
" 7 (BC_7, B24,            bidir, 0,      1, 0, Z),"&
" 8 (BC_7, B25,            bidir, 0,      1, 0, Z),"&
" 9 (BC_7, B26,            bidir, 0,      1, 0, Z),"&
"10 (BC_7, B27,            bidir, 0,      1, 0, Z),"&
"11 (BC_7, B28,            bidir, 0,      1, 0, Z),"&
"12 (BC_7, B29,            bidir, 0,      1, 0, Z),"&
"13 (BC_7, B30,            bidir, 0,      1, 0, Z),"&
"14 (BC_7, B31,            bidir, 0,      1, 0, Z),"&
"15 (BC_7, B16,            bidir, 0,      1, 0, Z),"&
"16 (BC_7, B17,            bidir, 0,      1, 0, Z),"&
"17 (BC_7, B18,            bidir, 0,      1, 0, Z),"&
"18 (BC_7, B19,            bidir, 0,      1, 0, Z),"&
"19 (BC_7, B20,            bidir, 0,      1, 0, Z),"&
"20 (BC_7, B21,            bidir, 0,      1, 0, Z),"&
"21 (BC_7, B22,            bidir, 0,      1, 0, Z),"&
"22 (BC_7, B23,            bidir, 0,      1, 0, Z),"&
"23 (BC_7, B8,             bidir, 0,      1, 0, Z),"&
"24 (BC_7, B9,             bidir, 0,      1, 0, Z),"&
"25 (BC_7, B10,            bidir, 0,      1, 0, Z),"&
"26 (BC_7, B11,            bidir, 0,      1, 0, Z),"&
"27 (BC_7, B12,            bidir, 0,      1, 0, Z),"&
"28 (BC_7, B13,            bidir, 0,      1, 0, Z),"&
"29 (BC_7, B14,            bidir, 0,      1, 0, Z),"&
"30 (BC_7, B15,            bidir, 0,      1, 0, Z),"&
"31 (BC_7, B0,             bidir, 0,      1, 0, Z),"&
"32 (BC_7, B1,             bidir, 0,      1, 0, Z),"&
"33 (BC_7, B2,             bidir, 0,      1, 0, Z),"&
"34 (BC_7, B3,             bidir, 0,      1, 0, Z),"&
"35 (BC_7, B4,             bidir, 0,      1, 0, Z),"&
"36 (BC_7, B5,             bidir, 0,      1, 0, Z),"&
"37 (BC_7, B6,             bidir, 0,      1, 0, Z),"&
"38 (BC_7, B7,             bidir, 0,      1, 0, Z),"&
"39 (BC_1, BE3,            output3, 0,      0, 0, Z),"&
"40 (BC_1, BE2,            output3, 0,      0, 0, Z),"&
"41 (BC_1, BE1_A30,        output3, 0,      0, 0, Z),"&

```

```

" 42 (BC_1, BE0_A31,          output3, 0,    0, 0, Z),"&
" 43 (BC_1, W_R,             output3, 0,    0, 0, Z),"&
" 44 (BC_1, BusReq,          output3, 0,    0, 0, Z),"&
" 45 (BC_1, BLast,           output3, 0,    0, 0, Z),"&
" 46 (BC_1, BootClkSpd,      input,   X              ),"&
" 47 (BC_1, Halt,            input,   X              ),"&
" 48 (BC_1, CritInt,         input,   X              ),"&
" 49 (BC_1, ExtInt,          input,   X              ),"&
" 50 (BC_1, BusError,        input,   X              ),"&
" 51 (BC_1, Ready,           input,   X              ),"&
" 52 (BC_1, HoldReq,         input,   X              ),"&
" 53 (BC_1, BootW,           input,   X              ),"&
" 54 (BC_1, BootXtal,        input,   X              ),"&
" 55 (BC_1, MemClk,          output3, 0,    0, 0, Z),"&
" 56 (BC_1, HoldAck,         output3, 0,    0, 0, Z)";

```

```
end PPC401GF;
```

8

Storage Control

The PPC401GF has a 32-bit address bus and a 4 gigabyte (GB) address space, which is presented as a flat address space.

The PPC401GF does not support address translation.

Up to 4GB of external memory can be attached.

8.1 Physical Address Map

Figure 8-1 shows the memory map of the PPC401GF, which is divided into bus regions and storage attribute control regions.

The eight Bus Region Control Registers (BRCR0–BRCR7) each control two 256MB bus regions that share the following programmable characteristics:

- Transfer hold cycles
- Maximum burst; four beats or continuous
- Target word-first or sequential line fills
- Bus width

The storage control attribute registers, described in the rest of this chapter, control storage attributes. Each storage control attribute register contains 32 bits. Each bit is associated with only one 128MB storage attribute control region.

Bus Regions	Bus Region Control Registers	Storage Attribute Control Register Bits	Storage Attribute Control Regions
0xFFFF FFFF	BRCR7	31	0xF800 0000
0xF000 0000		30	0xF000 0000
0xEFFF FFFF	BRCR6	29	0xE800 0000
0xE000 0000		28	0xE000 0000
0xDFFF FFFF	BRCR5	27	0xD800 0000
0xD000 0000		26	0xD000 0000
0xCFFF FFFF	BRCR4	25	0xC800 0000
0xC000 0000		24	0xC000 0000
0xBFFF FFFF	BRCR3	23	0xB800 0000
0xB000 0000		22	0xB000 0000
0xAFFF FFFF	BRCR2	21	0xA800 0000
0xA000 0000		20	0xA000 0000
0x9FFF FFFF	BRCR1	19	0x9800 0000
0x9000 0000		18	0x9000 0000
0x8FFF FFFF	BRCR0	17	0x8800 0000
0x8000 0000		16	0x8000 0000
0x7FFF FFFF	BRCR7	15	0x7800 0000
0x7000 0000		14	0x7000 0000
0x6FFF FFFF	BRCR6	13	0x6800 0000
0x6000 0000		12	0x6000 0000
0x5FFF FFFF	BRCR5	11	0x5800 0000
0x5000 0000		10	0x5000 0000
0x4FFF FFFF	BRCR4	9	0x4800 0000
0x4000 0000		8	0x4000 0000
0x3FFF FFFF	BRCR3	7	0x3800 0000
0x3000 0000		6	0x3000 0000
0x2FFF FFFF	BRCR2	5	0x2800 0000
0x2000 0000		4	0x2000 0000
0x1FFF FFFF	BRCR1	3	0x1800 0000
0x1000 0000		2	0x1000 0000
0x0FFF FFFF	BRCR0	1	0x0800 0000
0x0000 0000		0	0x0000 0000

Figure 8-1. Physical Address Map

8.2 Storage Attributes

The PowerPC Architecture defines storage attributes to control several aspects of storage behavior. The attributes supported by the PPC401GF 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 PPC401GF 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.

An external device controlled by a BRCCR can ignore A0 or A4, or both address bits, to support double- or quad-mapping of storage attributes to an external memory location. One, two, or four effective addresses (addresses for instruction fetches and data loads/stores) can refer to one byte of external memory, depending on whether any address bits are ignored. If the external address recognizes all bits in an effective address, one effective address refers to one byte of external memory. If A0 *or* A4 is ignored, two effective addresses refer to one byte of external memory. If A0 *and* A4 are ignored, four effective addresses refer to one byte of external memory. regions are independent of the bus regions. Each address is associated with only one set of storage attributes. However, as many as four addresses can be associated with one physical memory location.

The storage attributes in each storage attribute control region are set independently.

Figure 8-2 through Figure 8-6 provide bit descriptions for the SAC registers.

8.2.1 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.

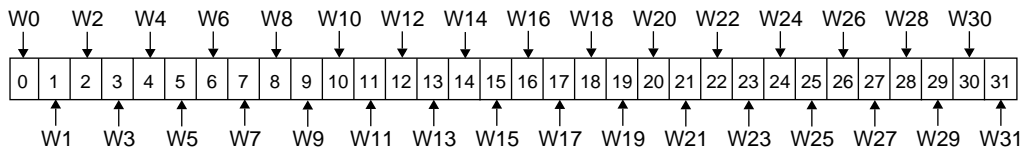


Figure 8-2. Data Cache Write-through Register (DCWR)

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 8-2. Data Cache Write-through Register (DCWR) (cont.)

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

8.2.2 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.

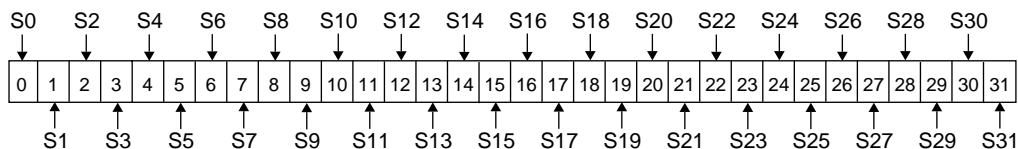


Figure 8-3. Data Cache Cacheability Register (DCCR)

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 8-3. 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

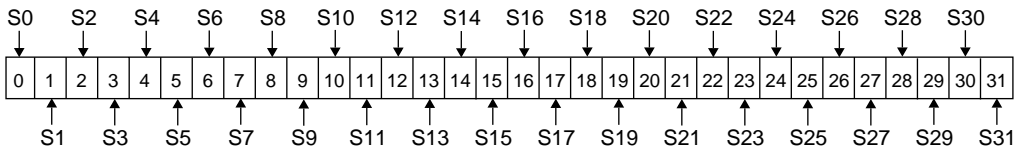
Figure 8-3. Data Cache Cacheability Register (DCCR) (cont.)

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

8.2.3 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 8-4. 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 8-4. 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 8-4. 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

8.2.4 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 PPC401GF 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.

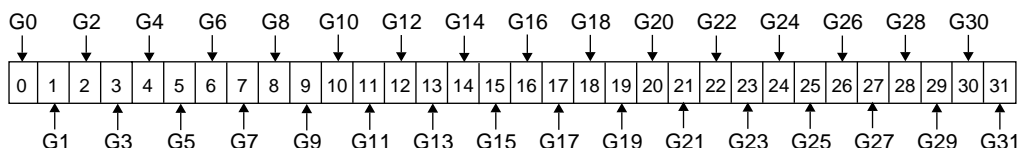


Figure 8-5. Storage Guarded Register (SGR)

0	G0	0 Normal 1 Guarded	0x0000 0000–0x07FF FFFF
---	----	-----------------------	-------------------------

Figure 8-5. 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 8-5. 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

8.2.5 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. 2-19, 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).

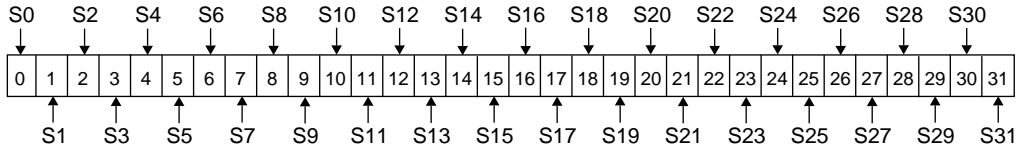


Figure 8-6. Storage Little-Endian Register (SLER)

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 8-6. 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

Clock Generation and Power Management

The PPC401GF incorporates sophisticated power management using techniques acquired during IBM's long history of microcontroller design.

The PPC401GF clock unit integrates clock generation and power management. The clock unit, which uses a crystal or oscillator input, generates duty cycle-corrected internal clocks having a frequency that is 2X, 1X, 1/2X, or 1/4X the reference clock frequency. The clock unit also provides a duty cycle-corrected MemClk output signal for external bus reference. The MemClk frequency can be 1X, 1/2X, 1/3X, or 1/4X the internal chip clock frequency.

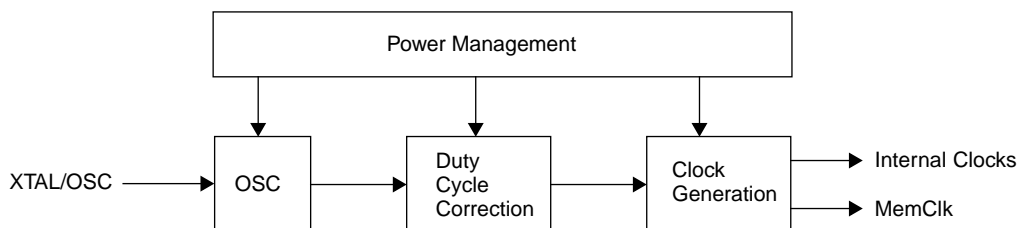


Figure 9-1. Clock Unit

The clock unit greatly reduces standby power consumption when programmed to invoke one of the three available sleep modes: doze, nap, and deep sleep. (The processor wait state also provides power savings; in wait mode, clock generation and distribution occur as in normal chip operation, but instruction fetching and execution are suspended.) Normal processing resumes from any sleep mode when an interrupt, external bus request, or reset occurs. In doze mode, clock generation continues, but clock distribution is suspended. Nap mode saves even more power: all clock generation circuitry, except for the on-chip oscillator, shuts down. Deep sleep mode saves the most power: all on-chip circuits shut down, reducing power consumption to an extremely low level.

The deeper the level of sleep, the longer the wake-up latency. This power control granularity provides the system designer with the flexibility to balance low power requirements against wake-up latency requirements. (Note that the processor wait state provides some power savings without wake-up latency.)

In addition to the low-power sleep modes provided by the clock unit, many logic and circuit design techniques were used to minimize power consumption during normal operation.

9.1 Power Management Control Register (PMCR0)

The Power Management Control Register (PMCR0) is a Device Control Register (DCR) that controls clock generation, power mode, timer clocks, and internal clock frequency relative to the clock input.

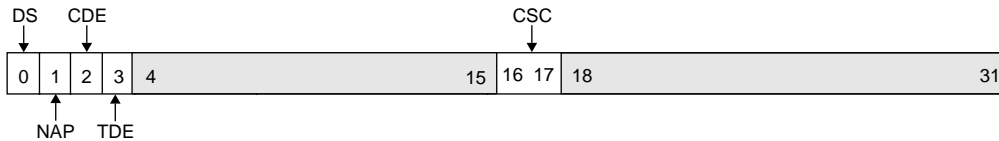


Figure 9-2. Power Management Control Register (PMCR0)

0	DS	Deep sleep 0 Clock generation is enabled. 1 Clock generation and the oscillator are disabled.	System reset value = 0
1	NAP	Nap mode 0 Oscillator and clock generation are enabled. 1 Oscillator is enabled and clock generation is disabled.	System reset value = 0
2	CDE	Chip doze enable 0 Internal chip clocks are running. 1 Internal chip clocks are off.	System reset value = 0
3	TDE	Timer doze enable 0 Timer clocks are running. 1 Timer clocks are off.	System reset value = 0
4:15		Reserved	
16:17	CSC	Chip Internal Clock Speed Control 00 Internal clock frequencies are twice crystal or external oscillator frequency. 01 Internal clock frequencies equal the crystal or external oscillator frequency. 10 Internal chip clock frequencies are one-half crystal or external oscillator frequency. 11 Internal chip clock frequencies are one-quarter crystal or external oscillator frequency.	System reset value = BootCikSpd, 1
18:31		Reserved	

9.2 Clock Generation

The input to the clock unit can be derived from a crystal (placed across the XTAL1 and XTAL2 pins, using the on-chip oscillator), or an external oscillator (with XTAL2 left floating and the external oscillator tied to XTAL1). From the clock input, the PPC401GF can generate internal chip clock frequencies of 2X, 1X, 1/2X, or 1/4X the clock input frequency, depending on the PMCR0[CSC] setting.

9.3 Clock Generation and Bus Speed

Two registers control the internal chip clock and external bus frequencies and MemClk alignment. The PMCR0[CSC] field controls the internal chip clock frequency as described in the preceding section. The Input/Output Configuration Register (IOCR) controls the external bus frequency and MemClk alignment. Depending on the IOCR[EBS] setting, the external bus frequency is 1X, or 1/2X, 1/3X, or 1/4X the internal chip clock frequency. IOCR[MCA] controls MemClk alignment with respect to the internal chip clock.

Engineering Note: To simplify the delay-locked loop design, MemClk is aligned with respect to the internal chip clock only when the MemClk frequency is 25 MHz. When the MemClk frequency is 25 MHz, set IOCR[MCA] = 1 to enable MemClk alignment.

For MemClk frequencies less than 25 MHz, MemClk alignment must be disabled by setting IOCR[MCA] = 0. This causes MemClk to be skewed slightly with respect to the internal chip clock.

The I/O timings in the PPC401GF data sheet specify timings with MemClk alignment enabled and disabled.

Care must be taken when changing PMCR0[CSC] or IOCR[EBS] while the PPC401GF is running. It is possible to change PMCR0[CSC] or IOCR[EBS] to values that result in bus speeds that are outside the operating range of the bus. To prevent this, pay attention to the order of operations. For example, to double the internal clock frequency while retaining the current bus speed, disable all external and critical interrupts, disable MemClk alignment, and lower the bus speed using IOCR[EBS]. Then, raise the internal clock frequency and the bus speed using PMCR0[CSC]. MemClk alignment must remain disabled if the MemClk frequency is less than 25 MHz.

When the bus speed changes, devices on the bus need time to adjust to the bus speed change. During this time, external bus activity is unreliable.

Instructions that change PMCR0[CSC] and IOCR[EBS] must run out of the cache to avoid bus activity while the MemClk frequency changes. External and critical interrupts should be disabled.

The following code example illustrates how to double the internal chip clock frequency and maintain the original MemClk frequency. In the example, the frequencies of the input clock, the internal chip clock, and MemClk are assumed to be 25 MHz initially. The code runs out of the cache and allows time for the bus to settle after the bus speed changes.

```

        bl      clock_stuff      # Put address of clock_stuff into LR
clock_stuff:
        mflr    r3               # Move address of clock_stuff to R3
        addi    r3, r3, 0x0020   # R3 contains the address of
                                # change_speed (the first instruction to
                                # be touched into the cache)
        addi    r4, r0, 0x0004   # Put 4 into CTR
        mtctr   r4

clock_stuff_loop:
        icbt    r0, r3           # icbt at change_speed
        addi    r3, r3, 0x0010   # Increment address by one line
        bdnz    clock_stuff_loop # Touch 4 lines into the cache
        isync                   # Ensure completion

change_speed:
        mfdcr   r4, iocr         # Turn off MemClk alignment
        ori     r4, r4, 0x0100   # (required when MemClk frequency is
        mtdcr   iocr, r4        # less than 25 MHz)

        mfdcr   iocr, r4        # Set MemClk frequency to 1/2X
        ori     r4, r4, 0x001    # internal chip clock
        mtdcr   iocr, r4

        mfdcr   r4, pmcr0       # Set internal chip clock to 2X input
        addis   r2, r0, 0xffff   # clock
        ori     r2, r0, 0x3fff
        and     r4, r2, r4
        mtdcr   pmcr0, r4

        addis   r4, r0, 0x0004   # Spin to regain DLL lock
        mtctr   r4

clock_stuff_spin:
        bdnz    clock_stuff_spin

```

9.4 Power Management

To support low-power applications, the PPC401GF provides three sleep modes for various power savings, and a wait mode. The sleep modes are, from the lowest power savings to the highest: doze mode, nap mode and deep sleep mode.

The wait mode is simply the PPC401GF in the wait state (when the Machine State Register wait state enable bit is set (MSR[WE] = 1)). Clock generation and distribution continue, but instruction processing is suspended. In the wait state, power consumption is reduced significantly.

Table 9-1 shows how the three sleep modes affect the chip.

Table 9-1. PPC401GF Sleep Modes

Sleep Mode	Clock Generation	Oscillator	MemClk	Chip Clock	Timer Clock	Typical Wake-up Latency
Doze mode (chip)	On	On	Active	Off	Running	2 clock cycles
Doze mode (timer)	On	On	Active	Running	Off	2 clock cycles
Doze mode (chip and timer)	On	On	Active	Off	Off	2 clock cycles
Nap mode	Off	On	Inactive	Off	Off	35 clock cycles
Deep sleep mode	Off	Off	Inactive	Off	Off	1.5 ms

When the processor core is in the wait state and the caches and the BCU are idle, the power management unit determines whether the PPC401GF can enter one of the sleep modes, depending on the bit settings in PMCR0. However, the timer doze mode does not depend upon processor quiescence; the timers shut down when PMCR0[TSE] is set, regardless of processor activity.

Figure 9-3 illustrates how the PMCR0 settings affect PPC401GF operation and how IOCR[EBS] affects the MemClk frequency.

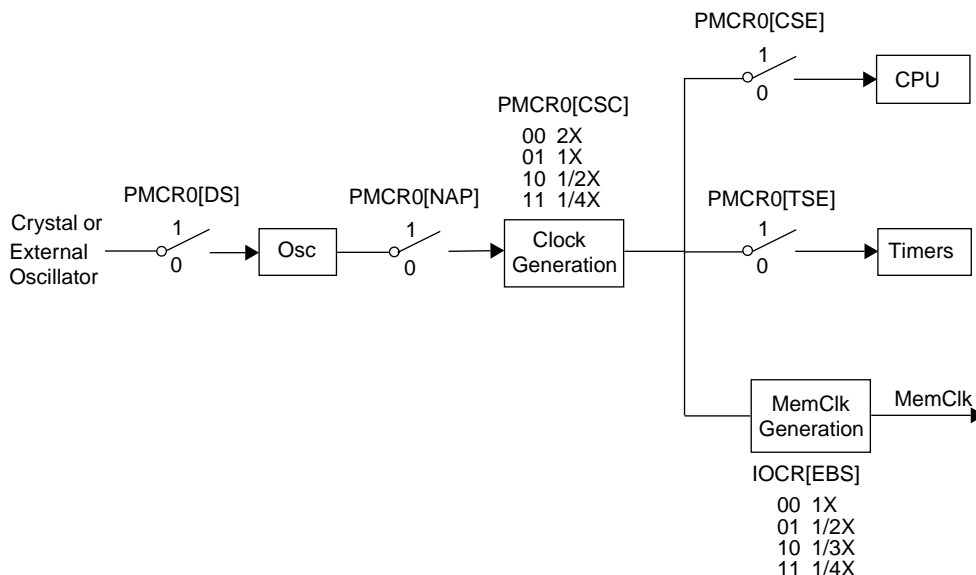


Figure 9-3. PMCR0 Settings and PPC401GF Operation

9.4.1 Doze Mode

In doze mode, clock generation continues, but clock distribution to the internal chip clocks or the timer clocks, or both, shuts down. Wake-up latency is low, and power savings are moderate.

If the processor is in the wait state, the caches and the BCU are idle, and PMCR0[DS] = 0 and PMCR0[NAP] = 0: the internal chip clocks are shut down if PMCR0[CDE] = 1. This is referred to as “chip doze.” The timer clocks are shut down if PMCR0[TDE] = 1 (“timer doze”). If PMCR0[CDE] = 1 and PMCR0[TDE] = 1, both sets of clocks shut down (“double doze”).

When the internal chip clocks are shut down and the timer clocks are still running, instruction processing halts. Timer-driven events (timer interrupts and reset), however, can still occur. When the timer clocks shut down, instruction processing continues, but the timers are unavailable. When both sets of clocks shut down, instruction processing and timer availability stop.

The MemClk signal remains active.

The 401GF wakes up from chip doze when a reset or interrupt occurs or an external bus request or debug halt is received. Waking up from timer doze requires a reset or an **mtbcr** instruction to set PMCR0[TDE] = 0. To wake up from double doze, the events required to

wake from chip doze and timer doze mode must occur, with a reset, or with the chip clocks waking up first to process the **mtdcr** instruction required to exit timer doze.

9.4.2 Nap Mode

In nap mode, all clock generation stops, but the on-chip oscillator continues to run. All internal chip clocks and timer clocks are shut down.

Because clock generation and distribution stops, the MemClk signal is forced high.

If the processor is in the wait state and the caches and the BIU are idle, the PPC401GF enters nap mode if $PMCR0[DS] = 0$ and $PMCR0[NAP] = 1$.

The 401GF wakes up from nap mode when a reset or interrupt occurs, or an external bus request or debug halt is received.

9.4.3 Deep Sleep Mode

In deep sleep mode, all clock activity stops, including clock generation, clock distribution, and the on-chip oscillator. The MemClk signal is forced high.

If the processor is in the wait state and the caches and the BIU are idle, the PPC401GF enters deep sleep mode if $PMCR0[DS] = 1$.

The 401GF wakes up from deep sleep mode when a reset or interrupt occurs, or an external bus request or debug halt is received.

10

Instruction Set

Descriptions of the PPC401GF 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.

10.1 Instruction Set Portability

To support embedded real-time applications, the instruction sets of the PPC401GF 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 10-1 lists instructions in the IBM PowerPC Embedded Environment that are implemented in the PPC401GF.

Table 10-1. Instructions in the IBM PowerPC Embedded Environment

dccci	mfdcr
dcread	mtdcr
iccci	rfci
icbt	wrttee
icread	wrtteei

10.2 Instruction Formats

For more detailed information about instruction formats, including a summary of instruction field usage and instruction format diagrams for the PPC401GF, see Section A.3, “Instruction Formats,” on p. A-47.

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 PPC401GF executes all invalid instruction forms without causing an illegal instruction exception.

10.3 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.
nb	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 10-2. 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

10.4 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. 2-11. For discussion of XER, see Section 2.2.2.3, “Fixed Point Exception Register (XER),” on p. 2-8.

10.5 Alphabetical Instruction Listing

The following pages list the instructions available in the PPC401GF 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

Add Carrying

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) ← (RA) + (RB) + XER[CA]
if (RA) + (RB) + XER[CA] u > 232 - 1 then
    XER[CA] ← 1
else
    XER[CA] ← 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.

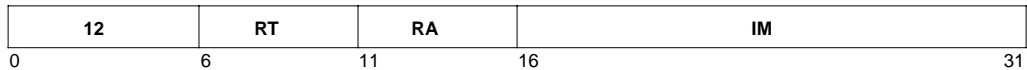
Table 10-3. Extended Mnemonics for addi

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



```
(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

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 10-4. 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.

Add Immediate Carrying and Record

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.

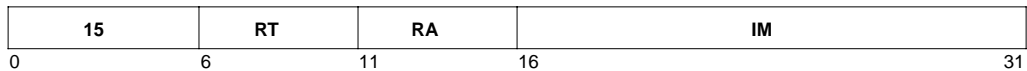
Table 10-5. Extended Mnemonics for addic.

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 10-6. 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



$(RT) \leftarrow (RA) + XER[CA] + (-1)$
 if $(RA) + XER[CA] + 0xFFFF\ FFFF \geq 2^{32} - 1$ then
 $XER[CA] \leftarrow 1$
 else
 $XER[CA] \leftarrow 0$

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

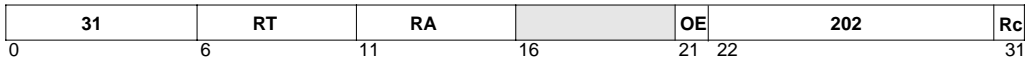
Architecture Note

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

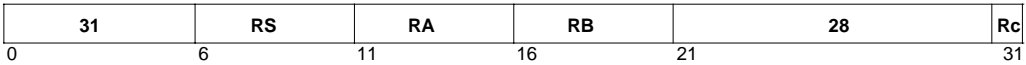
Invalid Instruction Forms

- 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

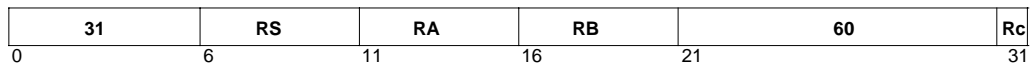
- RA
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1

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

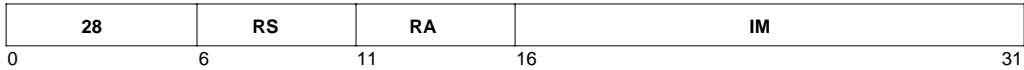
Architecture Note

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.**

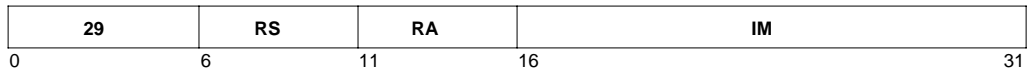
Architecture Note

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



```
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
         $BD \leftarrow target_{16:29}$ 
         $NIA \leftarrow EXTS(BD \parallel ^00)$ 
    else
         $BD \leftarrow (target - CIA)_{16:29}$ 
         $NIA \leftarrow CIA + EXTS(BD \parallel ^00)$ 
    else
         $NIA \leftarrow CIA + 4$ 
    if  $LK = 1$  then
         $(LR) \leftarrow CIA + 4$ 
     $PC \leftarrow NIA$ 

```

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 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 10-7. 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.
bdnzf	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	
bdnzfa		<i>Extended mnemonic for</i> bca 8,cr_bit,target	
bdnzfl		<i>Extended mnemonic for</i> bcl 8,cr_bit,target	(LR) \leftarrow CIA + 4.
bdnzfla		<i>Extended mnemonic for</i> bcla 8,cr_bit,target	(LR) \leftarrow CIA + 4.

bc

Branch Conditional

Table 10-7. 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) ← CIA + 4.
bdzla		<i>Extended mnemonic for</i> bcla 18,0,target	(LR) ← 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) ← CIA + 4.
bdzfla		<i>Extended mnemonic for</i> bcla 2,cr_bit,target	(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	
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.

Table 10-7. 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 10-7. 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
bgtla		<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.
bltla		<i>Extended mnemonic for</i> bcla 12,4*cr_field+0,target	(LR) ← CIA + 4.

Table 10-7. 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 10-7. 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	
bnu a		<i>Extended mnemonic for</i> bca 4,4*cr_field+3,target	
bnu l		<i>Extended mnemonic for</i> bcl 4,4*cr_field+3,target	(LR) ← CIA + 4.
bnu la		<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	
bso a		Extended mnemonic for bca 12,4*cr_field+3,target	
bso l		<i>Extended mnemonic for</i> bcl 12,4*cr_field+3,target	(LR) ← CIA + 4.
bso la		<i>Extended mnemonic for</i> bcla 12,4*cr_field+3,target	(LR) ← CIA + 4.

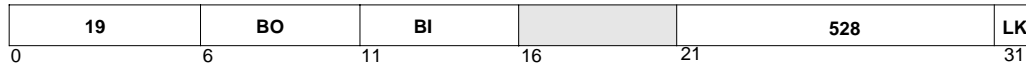
Table 10-7. Extended Mnemonics for bc, bca, bcl, bcla (cont.)

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.

10

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 10-8. 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 10-8. 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.

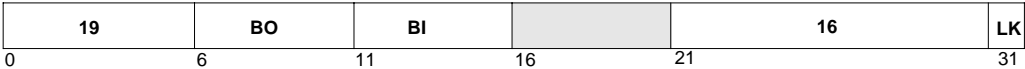
Table 10-8. Extended Mnemonics for bcctr, bcctrl (cont.)

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</i> bcctr 4,4*cr_field+3	
bnuctrl		<i>Extended mnemonic for</i> bcctrl 4,4*cr_field+3	(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</i> bcctr 12,4*cr_field+3	
bsoctrl		<i>Extended mnemonic for</i> bcctrl 12,4*cr_field+3	(LR) \leftarrow CIA + 4.
btctr	cr_bit	Branch if CR _{cr_bit} = 1 to address in CTR. <i>Extended mnemonic for</i> bcctr 12,cr_bit	
btctrl		<i>Extended mnemonic for</i> bcctrl 12,cr_bit	(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</i> bcctr 12,4*cr_field+3	
bunctrl		<i>Extended mnemonic for</i> bcctrl 12,4*cr_field+3	(LR) \leftarrow CIA + 4.

bclr

Branch Conditional to Link Register

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



```
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
     $NIA \leftarrow LR_{0:29} || ^20$ 
else
     $NIA \leftarrow CIA + 4$ 
if  $LK = 1$  then
     $(LR) \leftarrow CIA + 4$ 
 $PC \leftarrow NIA$ 
```

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 BO_2 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 10-9. 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 10-9. 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 10-9. 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	
btlrl		<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	
bnsrlrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+3	(LR) ← CIA + 4.

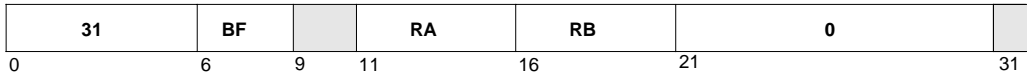
bclr

Branch Conditional to Link Register

Table 10-9. 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 PPC401GF, use of the extended mnemonic **cmpw BF,RA,RB** is recommended.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

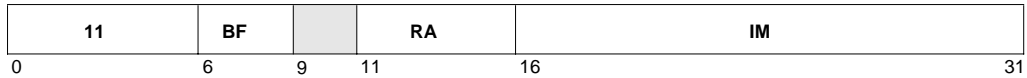
Table 10-10. Extended Mnemonics for cmp

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

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 PPC401GF, 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 10-11. 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  $c_2 \leftarrow 1$ 
 $c_3 \leftarrow \text{XER[SO]}$ 
 $n \leftarrow \text{BF}$ 
 $\text{CR[CRn]} \leftarrow c_{0:3}$ 

```

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 PPC401GF, use of the extended mnemonic **cmplw BF,RA,RB** is recommended.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

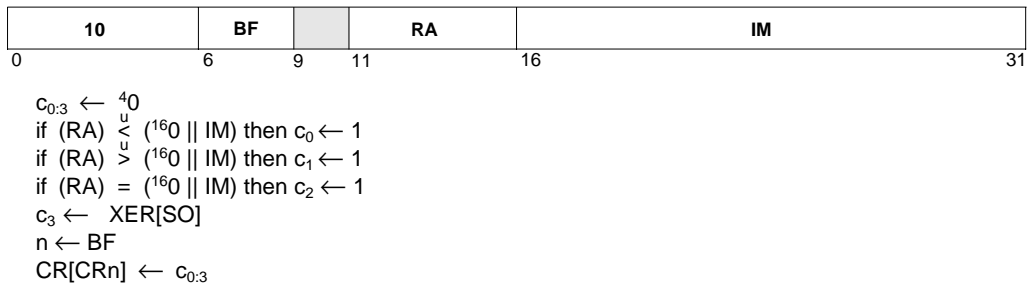
Table 10-12. Extended Mnemonics for cmpl

Mnemonic	Operands	Function	Other Registers Changed
cmplw	[BF,] RA, RB	Compare Logical Word; use CR0 if BF is omitted. <i>Extended mnemonic for</i> cmpl BF,0,RA,RB	

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 PPC401GF, use of the extended mnemonic **cmplwi BF,RA,IM** is recommended.

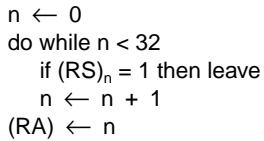
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 10-13. 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.

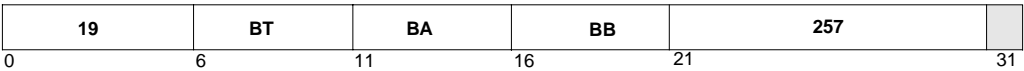
- RA
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1

- 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

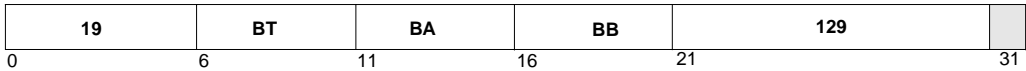
Invalid Instruction Forms

- 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

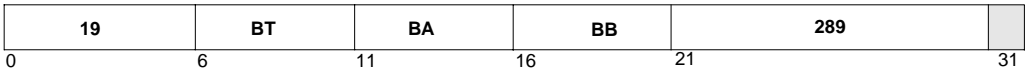
Architecture Note

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

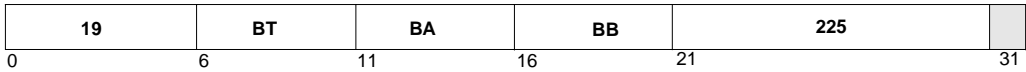
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 10-14. 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

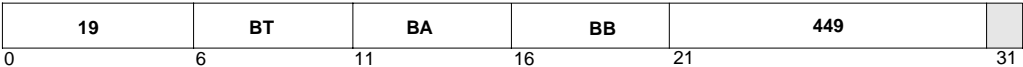
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Table 10-15. 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.

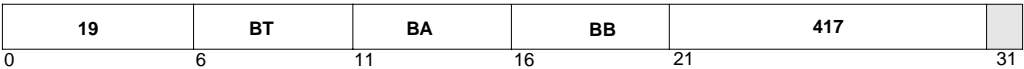
Table 10-16. Extended Mnemonics for cror

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

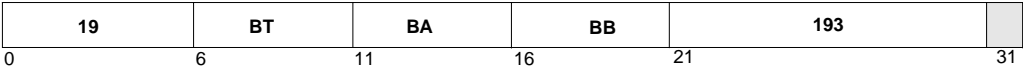
Invalid Instruction Forms

- 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.

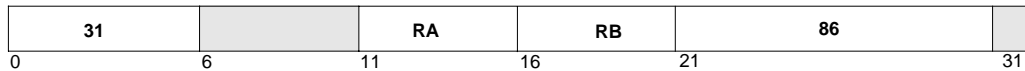
Table 10-17. Extended Mnemonics for crxor

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

Invalid Instruction Forms

- Reserved fields

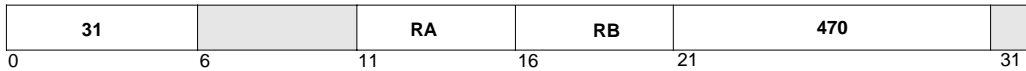
Exceptions

This instruction is considered a “load” with respect to data storage exceptions (for the PPC401GF, these are cache line locking exceptions). See Section 5.6, “Data Storage Exceptions,” on p. 5-19, 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

$EA \leftarrow (RA \ll 0) + (RB)$
 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.

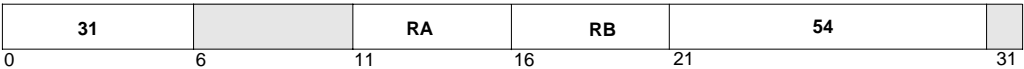
Architecture Note

This instruction is part of the PowerPC Operating Environment Architecture.

dcbst

Data Cache Block Store

dcbst RA, RB



$$EA \leftarrow (RA \neq 0) + (RB)$$
$$DCBST(EA)$$

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

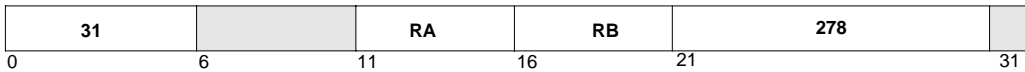
- Reserved fields

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

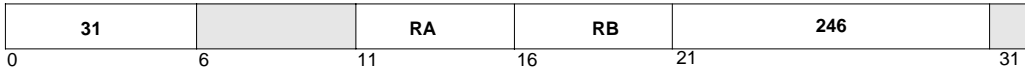
Data Cache Block Touch

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

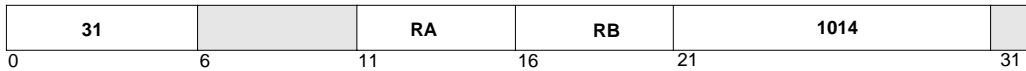
Data Cache Block Touch for Store

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

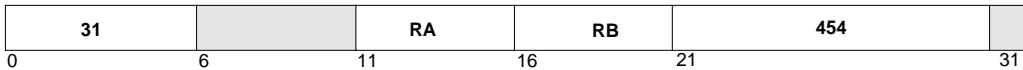
An alignment exception occurs if the EA is marked as non-cacheable or as write-through.

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 PPC401GF, these are cache line locking exceptions). See Section 5.6, “Data Storage Exceptions,” on p. 5-19, 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 10-18. 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

$\text{dividend} = (\text{quotient} \times \text{divisor}) + \text{remainder}$

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 PPC401GF, 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

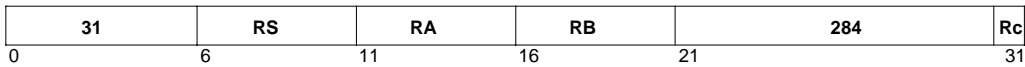
Equivalent

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

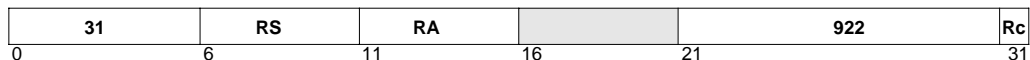
- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Extend Sign Halfword

extsh	RA, RS	Rc=0
extsh.	RA, RS	Rc=1


$$(RA) \leftarrow \text{EXTS}(RS)_{16:31}$$

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.

Registers Altered

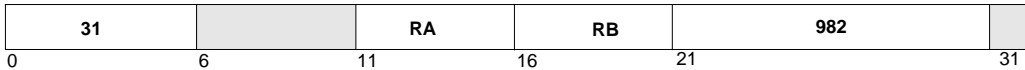
- RA
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1

Invalid Instruction Forms

- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

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

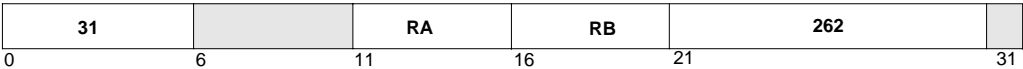
This instruction is considered a “load” with respect to data storage exceptions (for the PPC401GF, these are cache line locking exceptions). See Section 5.6, “Data Storage Exceptions,” on p. 5-19, for more information.

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

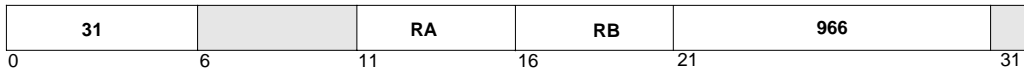
Instruction Cache Block Touch

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

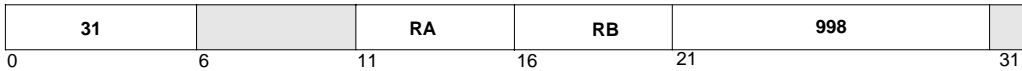
This instruction is considered a “load” with respect to data address compare (DAC) debug exceptions, but will not cause DAC debug events.

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

$$EA \leftarrow (RA[0] + (RB$$

$$\text{if } ((CDBCR[CIS] = 0) \wedge (CDBCR[CWS] = 0)) \text{ then } (ICDBDR \leftarrow (\text{i-cache data, way A})$$

$$\text{if } ((CDBCR[CIS] = 0) \wedge (CDBCR[CWS] = 1)) \text{ then } (ICDBDR \leftarrow (\text{i-cache data, way B})$$

$$\text{if } ((CDBCR[CIS] = 1) \wedge (CDBCR[CWS] = 0)) \text{ then } (ICDBDR \leftarrow (\text{i-cache tag, way A})$$

$$\text{if } ((CDBCR[CIS] = 1) \wedge (CDBCR[CWS] = 1)) \text{ then } (ICDBDR \leftarrow (\text{i-cache tag, way B})$$

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 10-19. 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.

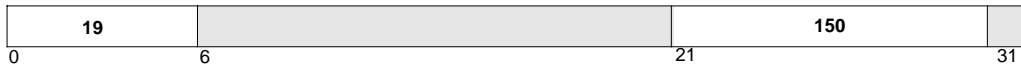
Architecture Note

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

Invalid Instruction Forms

- Reserved fields

Programming Note

See the discussion of context synchronizing instructions in Section 2.9.1 on p. 2-42.

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
```

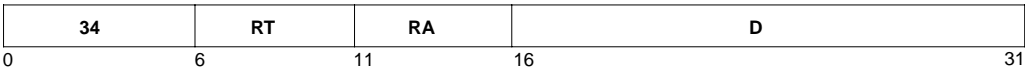
Architecture Note

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.

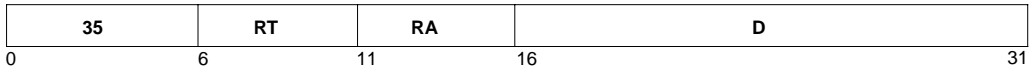
The byte at the EA is extended to 32 bits by concatenating 24 0-bits to its left. The result is placed into register RT.

Registers Altered

- RT

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Ibzu RT, D(RA)

$$EA \leftarrow (RA|0) + \text{EXTS}(D)$$

$$(RA) \leftarrow EA$$

$$(RT) \leftarrow {}^{24}0 \parallel \text{MS}(EA,1)$$

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

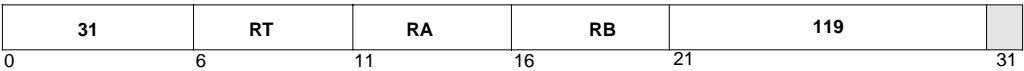
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

l**bzux**

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

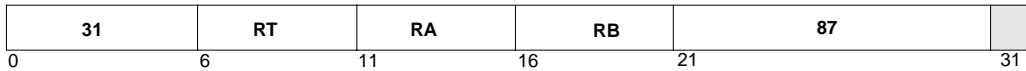
Invalid Instruction Forms

- 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.

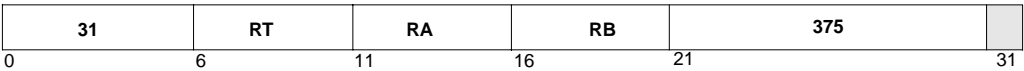
Architecture Note

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

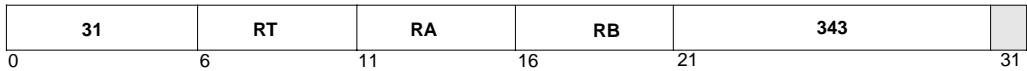
- Reserved fields
- RA = RT
- RA = 0

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

$$EA \leftarrow (RA|0) + (RB)$$

$$(RT) \leftarrow \text{EXTS}(\text{MS}(EA, 2))$$

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.

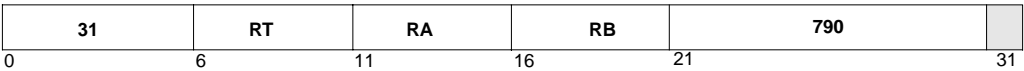
Architecture Note

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

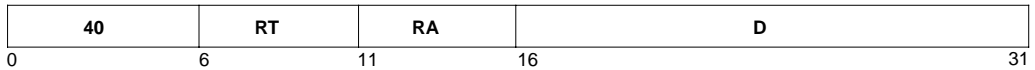
Exceptions

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.

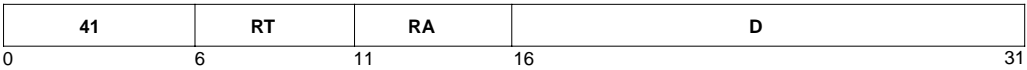
Architecture Note

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

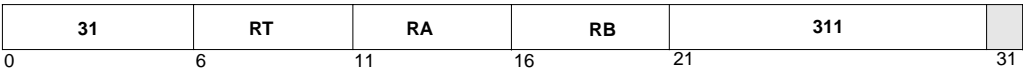
Exceptions

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.

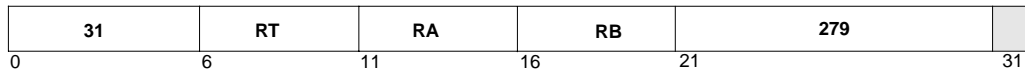
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

lhzx

Load Halfword and Zero Indexed

lhzx RT, RA, RB



$$\begin{aligned} EA &\leftarrow (RA \ll 0) + (RB) \\ (RT) &\leftarrow {}^{160} \parallel MS(EA, 2) \end{aligned}$$

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

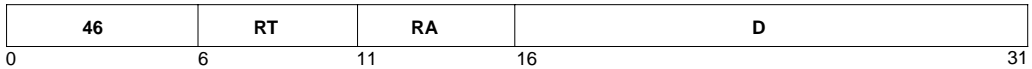
Exceptions

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.

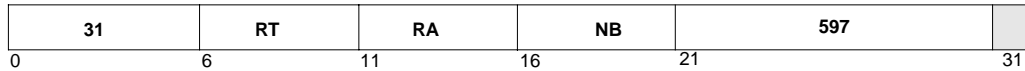
Architecture Note

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

```

10

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.

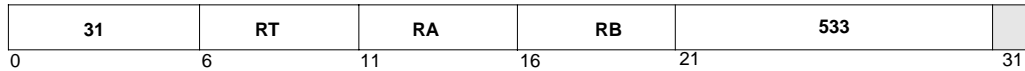
Architecture Note

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

```

10

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 PPC401GF 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.

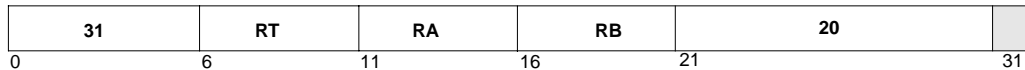
Architecture Note

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

10

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.

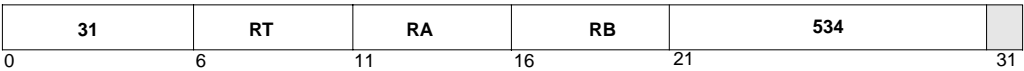
Architecture Note

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

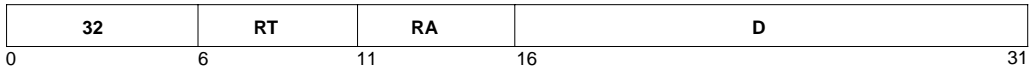
Exceptions

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.

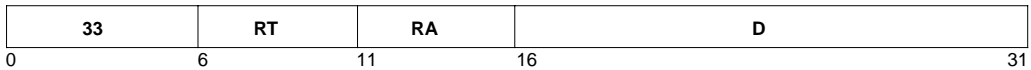
Architecture Note

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

- $RA = RT$
- $RA = 0$

10

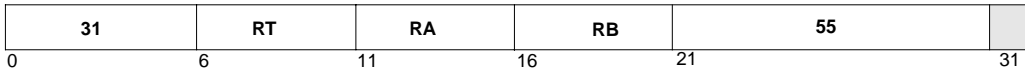
Exceptions

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.

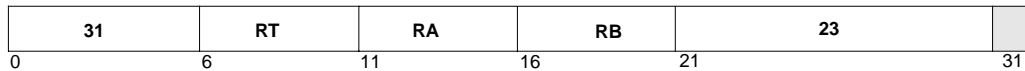
Architecture Note

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

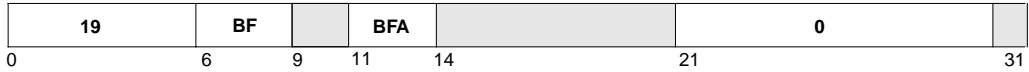
Exceptions

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

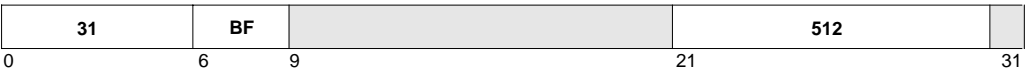
Architecture Note

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

If instruction bit 31 contains 1, the contents of $\text{CR}[\text{CR}_0]$ are undefined.

10

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

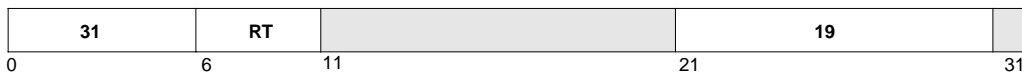
- Reserved fields

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

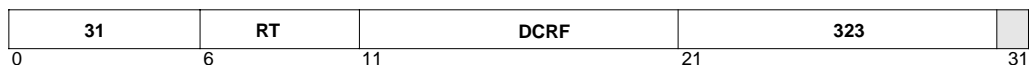
Architecture Note

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. See Table 11-3, “PPC401GF Device Control Registers,” on p. 11-4 for a list of DCR mnemonics and corresponding DCRN and DCRF values.

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.

10

The DCR number (DCRN) specified in the assembler language coding of the **mfdcr** instruction refers to a DCR number (see Table 11-3, “PPC401GF Device Control Registers,” on p. 11-4 for a list of DCRN values). 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.

Table 10-20. Extended Mnemonics for mfdcr

Mnemonic	Operands	Function	Other Registers Changed
mfbear mfbesr0 mfbrcr0 mfbrcr1 mfbrcr2 mfbrcr3 mfbrcr4 mfbrcr5 mfbrcr6 mfbrcr7 mfiocr mfpmc0	RT	Move from device control register DCRN. <i>Extended mnemonic for</i> mfdcr RT,DCRN See Table 11-3 on p. 11-4 for listing of valid DCRN values.	

mfmsr

Move From Machine State Register

mfmsr RT



(RT) ← (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 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 11-2, “Special Purpose Registers,” on p. 11-3 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

Execution of this instruction is privileged if instruction bit 11 contains 1. See Section 2.8.3, “Privileged SPRs,” on p. 2-41 for more information.

The SPR number (SPRN) specified in the assembler language coding of the **mfspr** instruction refers to an SPR number (see Table 11-2, “Special Purpose Registers,” on p. 11-3 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. 2-41 for information about privileged SPRs.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

Move From Special Purpose Register

Table 10-21. Extended Mnemonics for mfspr

Mnemonic	Operands	Function	Other Registers Changed
mfcdbcr mfctr mfdac1 mfdbscr mfdbsrs mfdccr mfdcwr mfdear mfesr mfevpr mfiacl mficcr mficdbdr mflr mfplr mfpr 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> mfspr RT,SPRN See Table 11-2, "Special Purpose Registers," on p. 11-3 for a list of valid SPRN values.	

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

mtcrf

Move to Condition Register Fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

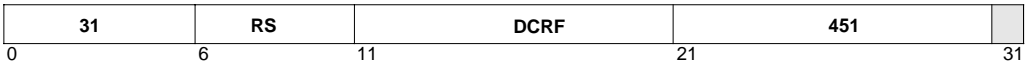
Table 10-22. Extended Mnemonics for mtcrrf

Mnemonic	Operands	Function	Other Registers Changed
mtcr	RS	Move to Condition Register. <i>Extended mnemonic for mtcrrf 0xFF,RS</i>	

mtdcr

Move To Device Control Register

mtdcr DCRN, RS



DCRN ← DCRF_{5:9} || DCRF_{0:4}
(DCR(DCRN)) ← (RS)

The contents of register RS are placed into the DCR specified by the DCRF field. See Table 11-3, “PPC401GF Device Control Registers,” on p. 11-4 for a list of DCR mnemonics and corresponding DCRN and DCRF values.

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.

The DCR number (DCRN) specified in the assembler language coding of the **mtdcr** instruction refers to a DCR number (see Table 11-3, “PPC401GF Device Control Registers,” on p. 11-4 for a list of DCRN values). The assembler handles the unusual register number encoding to generate the DCRF field.

mtdcr

Move To Device Control Register

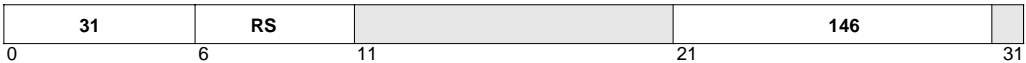
Architecture Note

Programs using this instruction are not portable to PowerPC implementations that do not implement the IBM PowerPC Embedded Environment.

Table 10-23. Extended Mnemonics for mtdcr

Mnemonic	Operands	Function	Other Registers Changed
mtbear mtbesr0 mtbrcr0 mtbrcr1 mtbrcr2 mtbrcr3 mtbrcr4 mtbrcr5 mtbrcr6 mtbrcr7 mtiocr mtpmcr0	RT	Move to device control register DCRN. <i>Extended mnemonic for mtdcr RT,DCRN</i> See Table 11-3, "PPC401GF Device Control Registers," on p. 11-4 for a list of valid DCRN values.	

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 11-2, “Special Purpose Registers,” on p. 11-3 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

Execution of this instruction is privileged if instruction bit 11 is a 1. See Section 2.8.3 (Privileged SPRs) on p. 2-41 for more information.

The SPR number (SPRN) specified in the assembler language coding of the **mtspr** instruction refers to an SPR number (see Table 11-2, “Special Purpose Registers,” on p. 11-3 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. 2-41 for information about privileged SPRs.

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

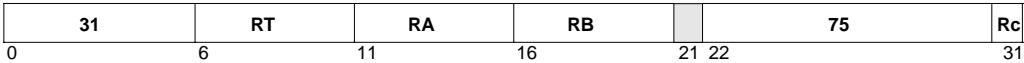
Table 10-24. Extended Mnemonics for mtspr

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 11-2, “Special Purpose Registers,” on p. 11-3 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

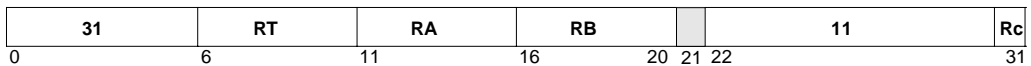
This instruction is part of the PowerPC User Instruction Set Architecture.

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.

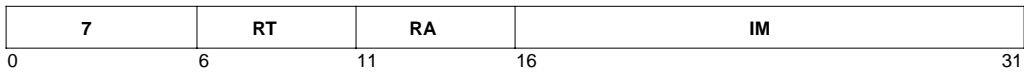
Architecture Note

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

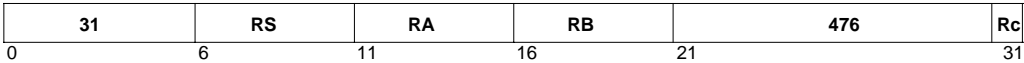
NAND

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.

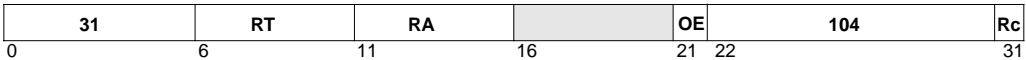
Registers Altered

- 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

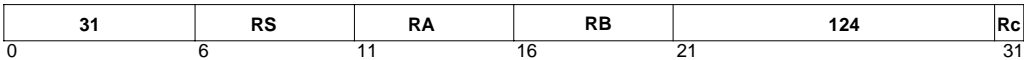
- Reserved fields

Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

nor
NOR

nor RA, RS, RB Rc=0
nor. RA, RS, RB 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 10-25. 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.

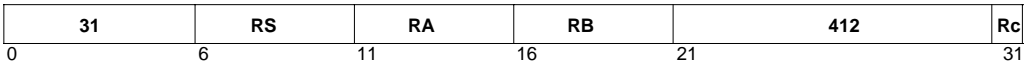
Table 10-26. Extended Mnemonics for or, or.

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.

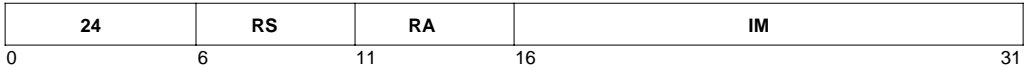
Registers Altered

- 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.

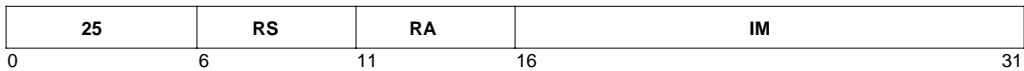
Table 10-27. Extended Mnemonics for ori

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.

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 10-28. 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 10-29. 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 rlwinm RA,RS,0,n,31</i>	
clrlwi.		<i>Extended mnemonic for rlwinm. RA,RS,0,n,31</i>	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 rlwinm RA,RS,n,b-n,31-n</i>	
clrlslwi.		<i>Extended mnemonic for rlwinm. RA,RS,n,b-n,31-n</i>	CR[CR0]

Table 10-29. 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 10-29. 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.

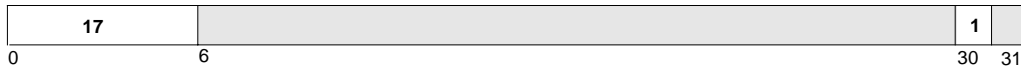
Table 10-30. Extended Mnemonics for rlwnm, rlwnm.

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.

Registers Altered

- SRR0
- SRR1
- MSR[WE, EE, PR, LE]

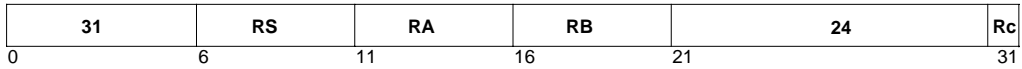
Invalid Instruction Forms

- 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 ← (RB)27:31
r ← ROTL((RS), n)
if (RB)26 = 0 then
    m ← MASK(0, 31 – n)
else
    m ← 320
(RA) ← r ∧ 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.

Registers Altered

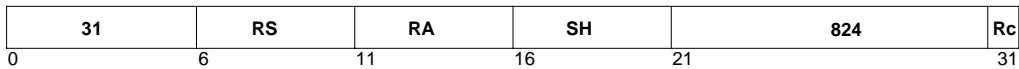
- RA
- XER[CA]
- CR[CR0]_{LT, GT, EQ, SO} if Rc contains 1

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

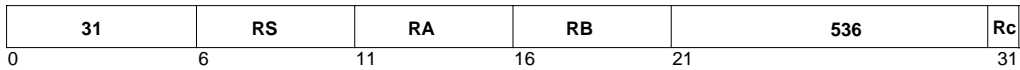
srw.

RA, RS, RB

RA, RS, RB

Rc=0

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.

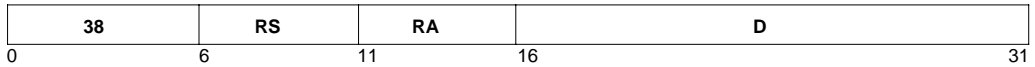
Registers Altered

- 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)



$EA \leftarrow (RA|0) + EXTS(D)$
 $MS(EA, 1) \leftarrow (RS)_{24:31}$

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

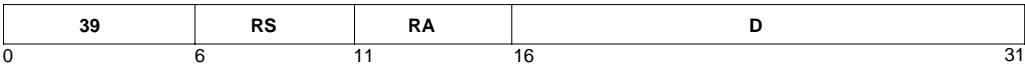
Architecture Note

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

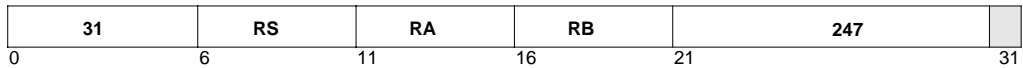
Invalid Instruction Forms

RA = 0

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
- $RA = 0$

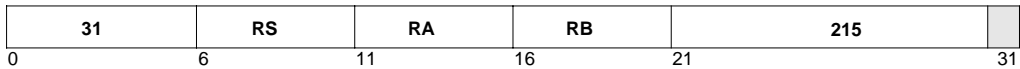
Architecture Note

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.

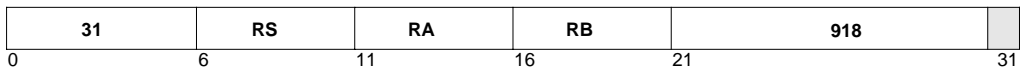
Architecture Note

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

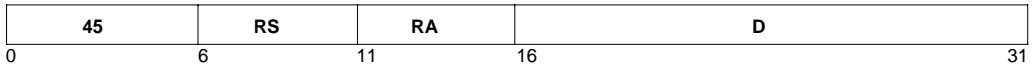
- Reserved fields

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)

$$EA \leftarrow (RA|0) + \text{EXTS}(D)$$

$$MS(EA, 2) \leftarrow (RS)_{16:31}$$

$$(RA) \leftarrow EA$$

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.

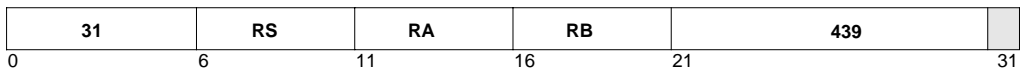
Architecture Note

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

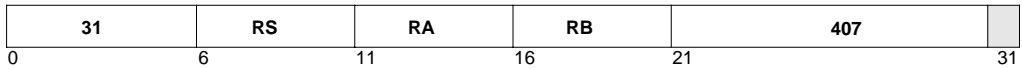
- Reserved fields
- RA = 0

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.

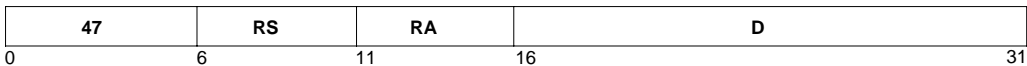
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stmw

Store Multiple Word

stmw RS, D(RA)



$EA \leftarrow (RA \ll 0) + \text{EXTS}(D)$

$r \leftarrow RS$

do while $r \leq 31$

$MS(EA, 4) \leftarrow (GPR(r))$

$r \leftarrow r + 1$

$EA \leftarrow EA + 4$

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

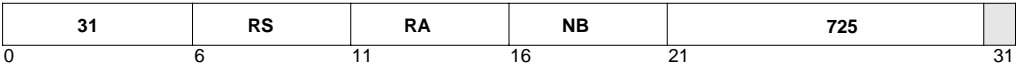
Exceptions

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.

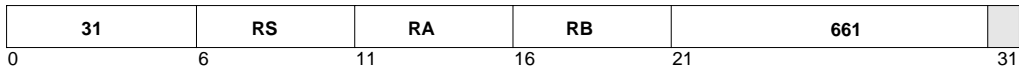
Architecture Note

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 PPC401GF 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.

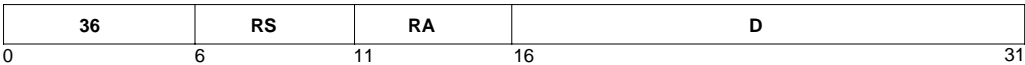
Architecture Note

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

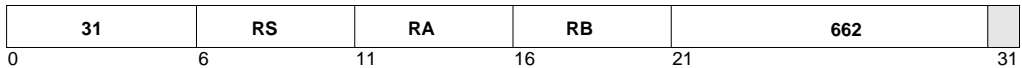
- None

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

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

```
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

10

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.

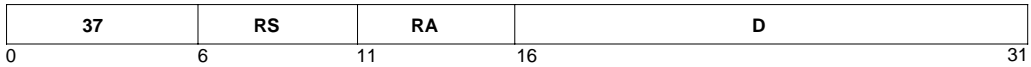
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

stwu

Store Word with Update

stwu RS, D(RA)



$EA \leftarrow (RA|0) + \text{EXTS}(D)$
 $MS(EA, 4) \leftarrow (RS)$
 $(RA) \leftarrow EA$

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 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

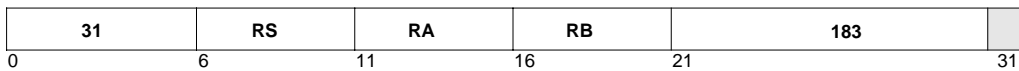
Exceptions

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.

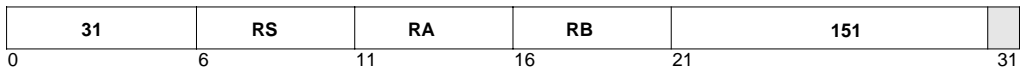
Architecture Note

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 10-31. 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 10-32. 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.

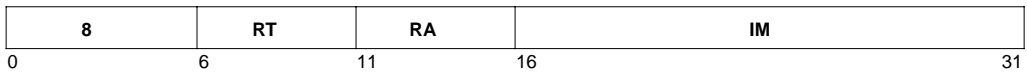
Registers Altered

- 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]

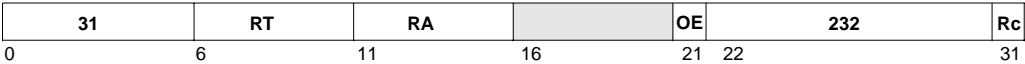
Architecture Note

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

Subtract from Zero Extended

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

31	RT	RA		OE	200	Rc
0	6	11	16	21	22	31

```

(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

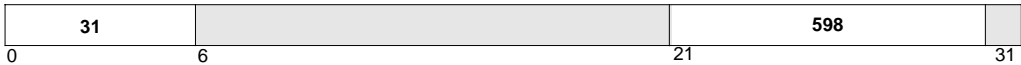
Architecture Note

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

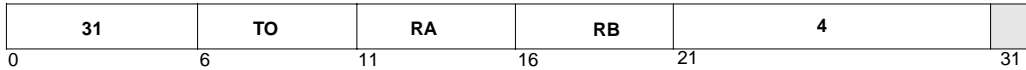
Programming Note

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 PPC401GF, 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. 5-23, 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. 5-27, for more information.

tw

Trap Word

```
(SRR2) ← address of tw instruction
(SRR3) ← (MSR)
(DBSR[TIE]) ← 1
(MSR[WE, EE, PR, CE, DE]) ← 0
(MSR[LE]) ← (MSR[ILE])
PC ← EVPR0:15 || 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. 5-23 for more information.

```
(SRR0) ← address of tw instruction
(SRR1) ← (MSR)
(ESR[PTR]) ← 1
(DBSR[TIE, IDE]) ← 1, 1
(MSR[WE, EE, PR]) ← 0
(MSR[LE]) ← (MSR[ILE])
PC ← EVPR0:15 || 0x0700
```

If instruction bit 31 contains 1, the contents of CR[CR0] are undefined.

Registers Altered

- None

10

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 10-33. Extended Mnemonics for tw

Mnemonic	Operands	Function	Other Registers Changed
trap		Trap unconditionally. <i>Extended mnemonic for tw 31,0,0</i>	

Table 10-33. Extended Mnemonics for tw (cont.)

Mnemonic	Operands	Function	Other Registers Changed
tweq	RA, RB	Trap if (RA) equal to (RB). <i>Extended mnemonic for tw 4,RA,RB</i>	
twge	RA, RB	Trap if (RA) greater than or equal to (RB). <i>Extended mnemonic for tw 12,RA,RB</i>	
twgt	RA, RB	Trap if (RA) greater than (RB). <i>Extended mnemonic for tw 8,RA,RB</i>	
twle	RA, RB	Trap if (RA) less than or equal to (RB). <i>Extended mnemonic for tw 20,RA,RB</i>	
twlge	RA, RB	Trap if (RA) logically greater than or equal to (RB). <i>Extended mnemonic for tw 5,RA,RB</i>	
twlgt	RA, RB	Trap if (RA) logically greater than (RB). <i>Extended mnemonic for tw 1,RA,RB</i>	
twlle	RA, RB	Trap if (RA) logically less than or equal to (RB). <i>Extended mnemonic for tw 6,RA,RB</i>	
twllt	RA, RB	Trap if (RA) logically less than (RB). <i>Extended mnemonic for tw 2,RA,RB</i>	
twlng	RA, RB	Trap if (RA) logically not greater than (RB). <i>Extended mnemonic for tw 6,RA,RB</i>	
twlnl	RA, RB	Trap if (RA) logically not less than (RB). <i>Extended mnemonic for tw 5,RA,RB</i>	
twlt	RA, RB	Trap if (RA) less than (RB). <i>Extended mnemonic for tw 16,RA,RB</i>	

tw

Trap Word

Table 10-33. 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. 5-23 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 PPC401GF.

(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. 5-27, 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. 5-23 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 10-34. 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 10-34. 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>	
twllel	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>	

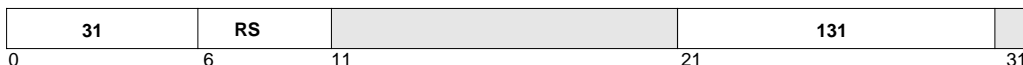
twi

Trap Word Immediate

Table 10-34. 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



$MSR[EE] \leftarrow (RS)_{16}$

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



MSR[EE] ← 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.

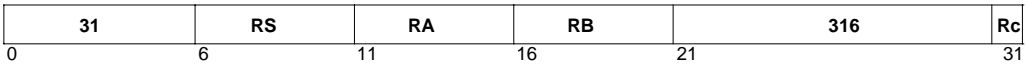
This instruction is used to provide atomic update of MSR[EE]. Typical usage is:

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	Rc=0
xor.	RA, RS, RB	Rc=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

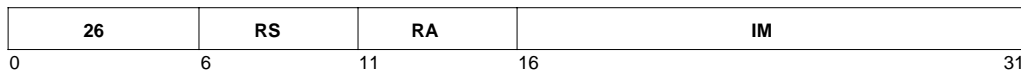
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

xori

XOR Immediate

xori RA, RS, IM



$$(RA) \leftarrow (RS) \oplus (^{16}0 \parallel IM)$$

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

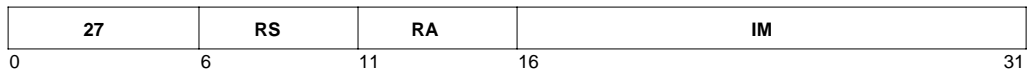
Architecture Note

This instruction is part of the PowerPC User Instruction Set Architecture.

xoris

XOR Immediate Shifted

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 PPC401GF are architected as 32-bits. Table 11-1 through Table 11-3 define the addressing required to access the registers. The pages following these tables define the bit usage within each register.

11.1 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.

11.2 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.

11.3 General Purpose Registers

The PPC401GF 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 11-1. PPC401GF General Purpose Registers

Mnemonic	Register Name	GPR Number		Access
		Decimal	Hex	
R0–R31	General Purpose Register 0–31	0–31	0x0–0x1F	Read/Write

11.4 Machine State Register and Condition Register

Because these registers are accessed using special instructions, they do not require addressing.

11.5 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.

Table 11-2 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. 2-41.

Table 11-2 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 11-2. 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 11-2. 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

11.6 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, such as the Bus Control Unit.

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. 2-40.

All DCR numbers not listed are reserved, and should be neither read nor written.

Table 11-3 shows the mnemonic, name, and number for each DCR. The columns labeled “DCRN” lists the register numbers used as operands in assembler language coding of the **mfdcr** and **mtdcr** instructions. The column labeled “DCRF” lists the corresponding fields contained in the *machine code* of **mfdcr** and **mtdcr**. A DCRN field contains two five-bit subfields of the DCRF field; the subfields are *reversed* in the machine code for the **mfdcr** and **mtdcr** instructions ($\text{DCRN} \leftarrow \text{DCRF}_{5:9} \parallel \text{DCRF}_{0:4}$) for compatibility with the POWER Architecture. Note that the assembler handles the special coding transparently.

Table 11-3. PPC401GF Device Control Registers

Mnemonic	Register Name	DCRN		DCRF	Access
		Decimal	Hex		
BEAR	Bus Error Address Register	144	0x090	0x204	Read Only
BESR0	Bus Error Status Register	145	0x091	0x224	Read/Clear
BRCR0	Bus Region Control Register 0	128	0x080	0x004	Read/Write

Table 11-3. PPC401GF Device Control Registers (cont.)

Mnemonic	Register Name	DCRN		DCRF	Access
		Decimal	Hex		
BRCR1	Bus Region Control Register 1	129	0x081	0x024	Read/Write
BRCR2	Bus Region Control Register 2	130	0x082	0x044	Read/Write
BRCR3	Bus Region Control Register 3	131	0x083	0x064	Read/Write
BRCR4	Bus Region Control Register 4	132	0x084	0x084	Read/Write
BRCR5	Bus Region Control Register 5	133	0x085	0x0A4	Read/Write
BRCR6	Bus Region Control Register 6	134	0x086	0x0C4	Read/Write
BRCR7	Bus Region Control Register 7	135	0x087	0x0E4	Read/Write
IOCR	Input/Output Configuration Register	160	0x0A0	0x005	Read/Write
PMCR0	Power Management Control Register	161	0x0A1	0x025	Read/Write

11.7 Alphabetical Register Listing

The following pages list the registers available in the PPC401GF. For each register, the following information is supplied:

- Register name and mnemonic
- Register type (DCR, 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

BEAR

DCR 0x90 Read-Only

See Section 5.5.2 on p. 5-16.



Figure 11-1. Bus Address Error Register (BEAR)

0:31	Address of Bus Error (asynchronous)
------	-------------------------------------

DCR 0x91 Read/Clear

See also Section 5.5.1 on p. 5-15.

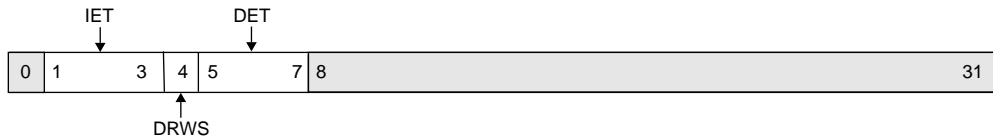


Figure 11-2. Bus Error Status Register 0 (BESR0)

0		Reserved
1:3	IET	Instruction side Error Type 000 No Error 001 Reserved 010 Reserved 011 Reserved 100 Reserved 101 Reserved 110 Active level on the bus error input signal 111 Reserved Reset value = 000
4	DRWS	Data Read/Write Status 0 Data side write error 1 Data side read error Reset value = 0
5:7	DET	Data side Error Type 000 No Error 001 Reserved 010 Reserved 011 Reserved 100 Reserved 101 Reserved 110 Active level on the bus error input signal 111 Reserved Reset value = 000
8:31		Reserved

BRCR0–BRCR7

DCR 0x80–0x87

See also Section 3.7 on p. 3-15.

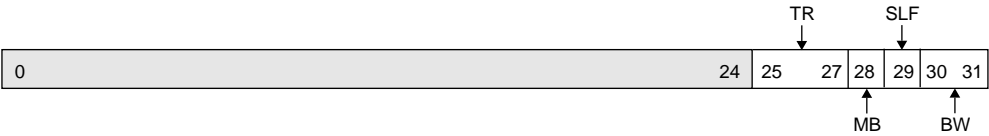


Figure 11-3. Bus Region Control Register (BRCR0–BRCR7)

0:24		Reserved	
25:27	TR	Transfer Recovery 000 Reserved 001 1-cycle 010 2-cycles 011 3-cycles 100 4-cycles 101 5-cycles 110 6-cycles 111 7-cycles	System reset value = 111
28	MB	Maximum Burst 0 Bursts have a maximum burst of four beats 1 Bursts have a maximum burst of 16 beats	System reset value = 0
29	SLF	Sequential Line Fills 0 Line fills are target word first 1 Line Fills are sequential	System reset value = 1
30:31	BW	Bus Width 00 8-bit bus 01 16-bit bus 10 32-bit bus 11 Reserved	System reset value = BootW

SPR 0x3D7

See also Section 6.4 on p. 6-9.

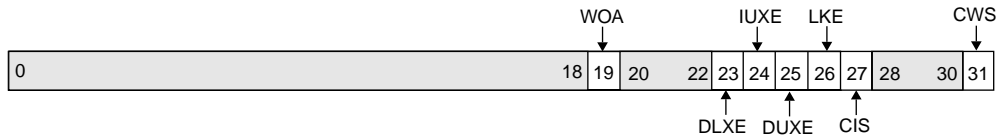


Figure 11-4. Cache Debug Control Register (CDBCR)

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.
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. 2-11.

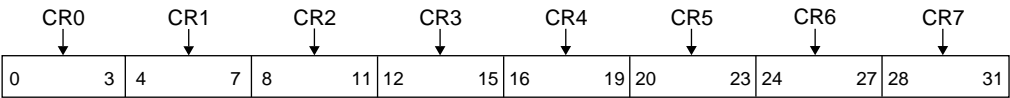


Figure 11-5. Condition Register (CR)

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. 2-6.

0	31
---	----

Figure 11-6. 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 11-7. Data Address Compare Register (DAC1)

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.

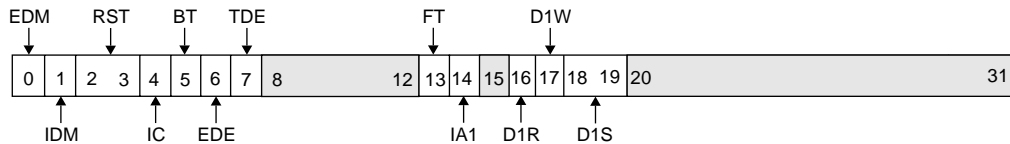


Figure 11-8. Debug Control Register (DBCR)

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.	
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 11-8. 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. 7-6.

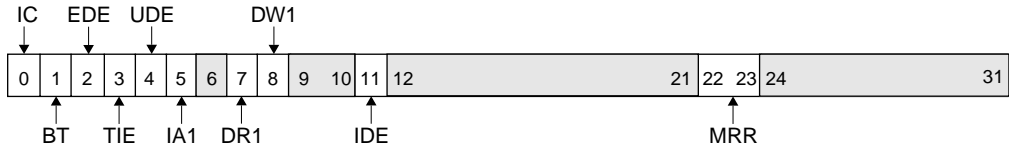


Figure 11-9. Debug Status Register (DBSR)

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 11-9. 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 8.2.1 on p. 8-4.

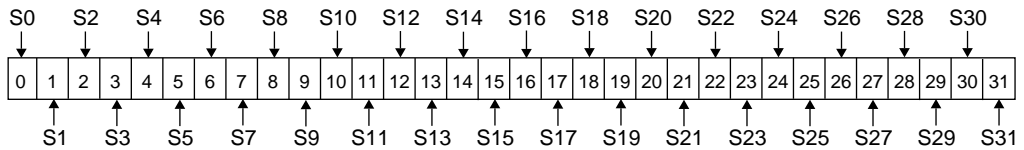


Figure 11-10. Data Cache Cacheability Register (DCCR)

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

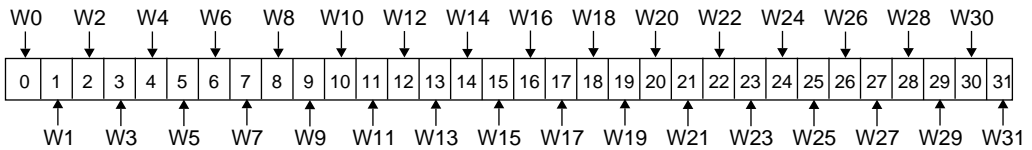
DCCR (cont.)

Figure 11-10. 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 8.2.1 on p. 8-4.

**Figure 11-11. 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 11-11. 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. 5-13.



Figure 11-12. Data Exception Address Register (DEAR)

0:31	Address of Data Error (synchronous)
------	-------------------------------------

ESR

SPR 0x3D4

See also Section 5.3.5 on p. 5-11.

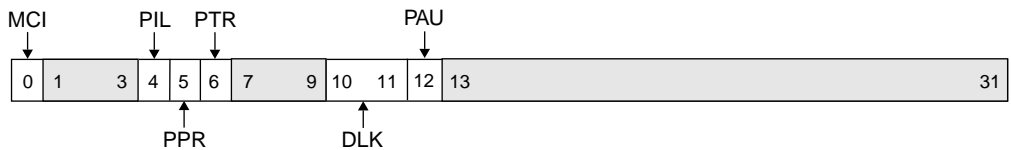


Figure 11-13. Exception Syndrome Register (ESR)

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.
13:31		Reserved

SPR 0x3D6

See also Section 5.3.4 on p. 5-10.

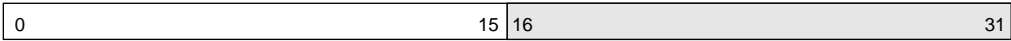


Figure 11-14. Exception Vector Prefix Register (EVPR)

0:15		Exception Vector Prefix
16:31		Reserved

GPR

See also Section 2.2.1 on p. 2-4.



Figure 11-15. General Purpose Register (R0-R31)



SPR 0x3F4

See also Section 7.6.4 on p. 7-10.

0	29	30	31
---	----	----	----

Figure 11-16. 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 8.2.3 on p. 8-8.

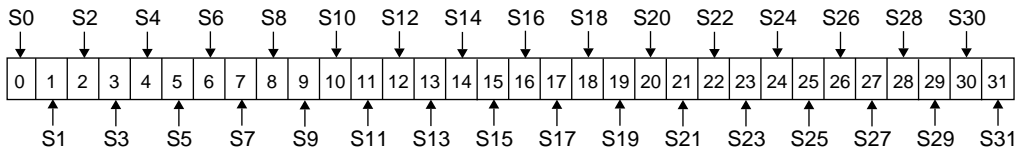


Figure 11-17. Instruction Cache Cacheability Register (ICCR)

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 11-17. 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 11-18. Instruction Cache Debug Data Register (ICDBDR)

0:31		Instruction cache information	See icread , p. 10-74.
------	--	-------------------------------	-------------------------------

Table 11-4. 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

DCR 0xA0

See also Section 5.7.1 on p. 5-20.

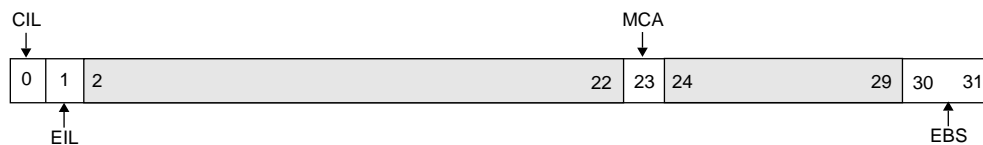


Figure 11-19. Input/Output Configuration Register (IOCR)

0	CIL	Critical Interrupt Level 0 Active low 1 Active high	System reset value = 0
1	EIL	External Interrupt Level 0 Active low 1 Active high	System reset value = 0
2:22		Reserved	
23	MCA	MemClk Alignment 0 MemClk is aligned. 1 MemClk is unaligned.	System reset value = BootClkSpeed input
24:29		Reserved	
30:31	EBS	External Bus Speed 00 MemClk frequency is equal to the internal clock frequency. 01 MemClk frequency is 1/2 the internal clock frequency. 10 MemClk frequency is 1/3 the internal clock frequency. 11 MemClk frequency is 1/4 the internal clock frequency.	System reset value = BootClkSpeed input applied to both bits

LR

SPR 0x008

See also Section 2.2.2.2 on p. 2-7.

0	31
---	----

Figure 11-20. Link Register (LR)

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

See also Section 5.3.1 on p. 5-7.

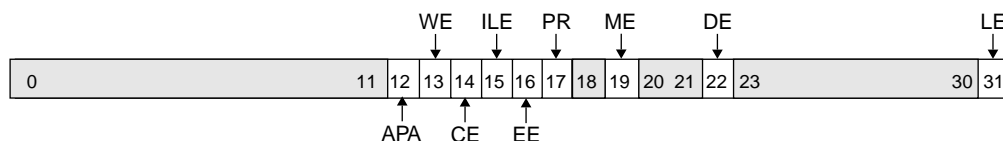


Figure 11-21. Machine State Register (MSR)

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.	

MSR (cont.)

Figure 11-21. 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.

SPR 0x3DB

See also Section 5.15.2 on p. 5-32.

0	31
---	----

Figure 11-22. Programmable Interval Timer (PIT)

0:31		Programmed interval remaining	Number of clocks remaining until the PIT event
------	--	-------------------------------	--

PMCR0

DCR 0x0A1

See also Section 9.1 on p. 9-2.

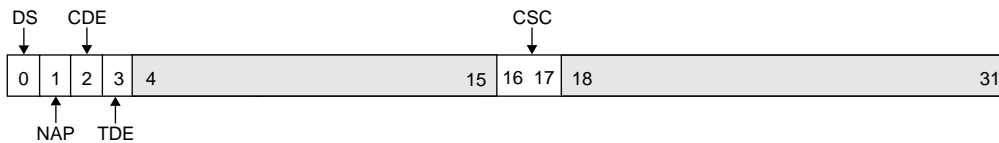


Figure 11-23. Power Management Control Register (PMCR0)

0	DS	Deep sleep 0 Clock generation is enabled. 1 Clock generation and the oscillator are disabled.	System reset value = 0
1	NAP	Nap mode 0 Oscillator and clock generation are enabled. 1 Oscillator is enabled and clock generation is disabled.	System reset value = 0
2	CDE	Chip doze enable 0 Internal chip clocks are running. 1 Internal chip clocks are off.	System reset value = 0
3	TDE	Timer doze enable 0 Timer clocks are running. 1 Timer clocks are off.	System reset value = 0
4:15		Reserved	
16:17	CSC	Chip Internal Clock Speed Control 00 Internal clock frequencies are twice crystal or external oscillator frequency. 01 Internal clock frequencies equal the crystal or external oscillator frequency. 10 Internal chip clock frequencies are one-half crystal or external oscillator frequency. 11 Internal chip clock frequencies are one-quarter crystal or external oscillator frequency.	System reset value = BootClkSpd, 1
18:31		Reserved	

SPR 0x11F Read-Only

See also Section 2.2.2.5 on p. 2-10.

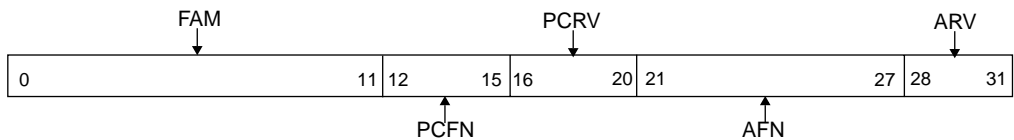


Figure 11-24. Processor Version Register (PVR)

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.	.
16:20	PCRV	Processor Core Revision. Identifies a revision of the processor core defined by the PFN field.	0x0 for PPC401GF
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 8.2.4 on p. 8-10.

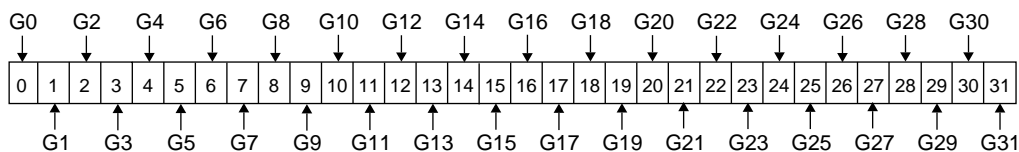


Figure 11-25. Storage Guarded Register (SGR)

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 11-25. 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 8.2.5 on p. 8-12.

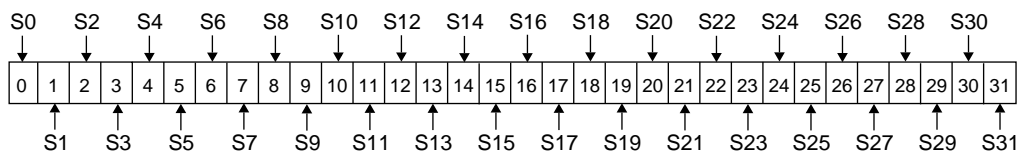


Figure 11-26. Storage Little-Endian Register (SLER)

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 11-26. 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. 2-10.

0	31
---	----

Figure 11-27. 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 11-28. 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.

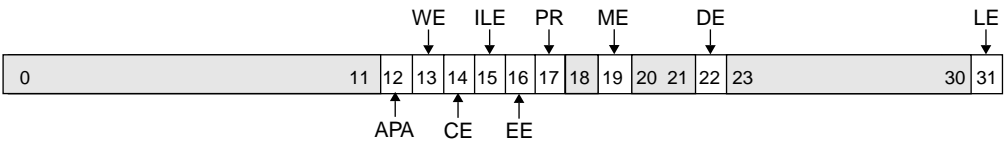


Figure 11-29. Save/Restore Register 1 (SRR1)

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 11-30. 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.

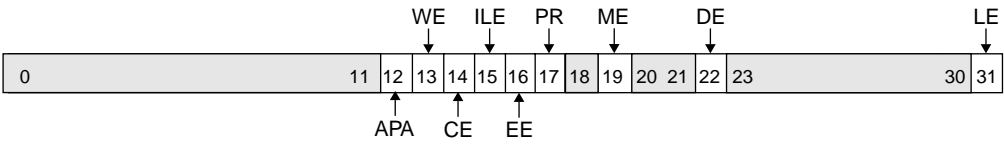


Figure 11-31. Save/Restore Register 1 (SRR1)

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. 5-29.



Figure 11-32. 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. 5-29.

0	31
---	----

Figure 11-33. 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. 5-29.



Figure 11-34. 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. 5-29.

0	31
---	----

Figure 11-35. 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. 5-37.

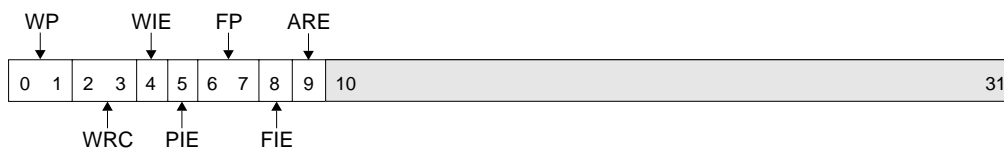


Figure 11-36. Timer Control Register (TCR)

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. 5-36.

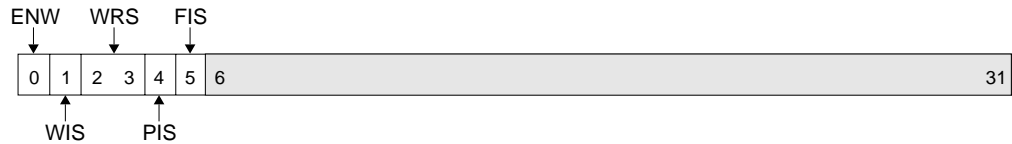


Figure 11-37. Timer Status Register (TSR)

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. 5-34.
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. 2-8.

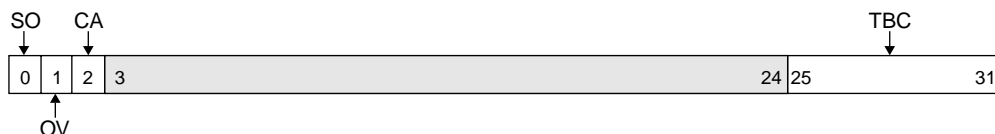


Figure 11-38. Fixed Point Exception Register (XER)

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. 2-9); 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. 2-9); 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. 2-9); 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

12

Signal Descriptions

Figure 12-1 shows the PPC401GF signals grouped by function. Table 12-1 lists the pin description nomenclature used in Table 12-2, which lists the PPC401GF signals, ordered by signal name. Table 12-3 lists the PPC401GF signals, ordered by pin number.

Active-low signals are shown with overbars: $\overline{BE2}$. Multiplexed signals are alphabetized under the first (unmultiplexed) signal names on the same pins.

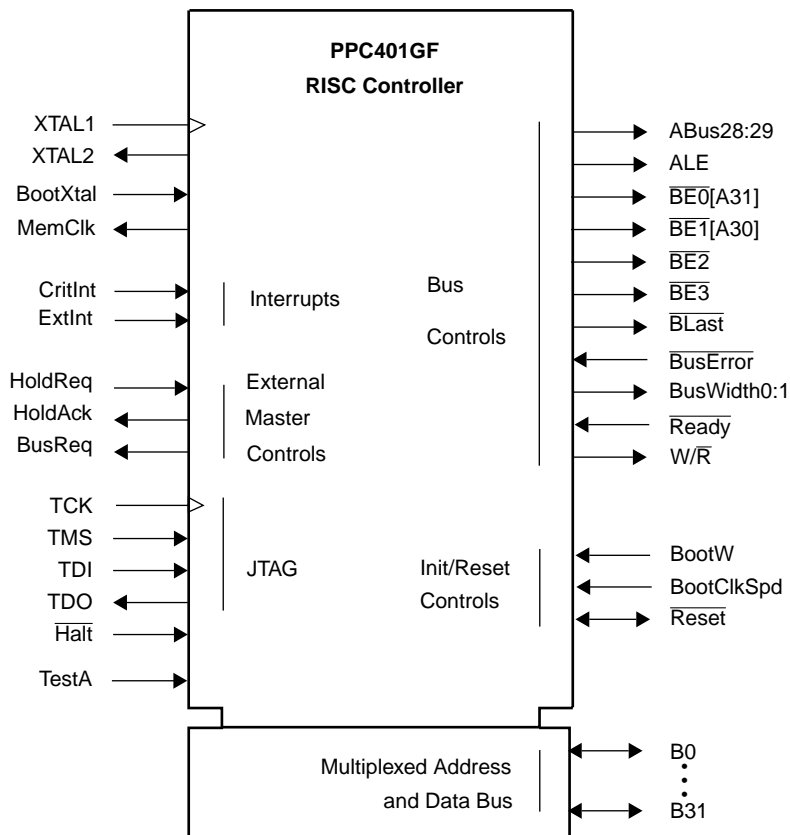


Figure 12-1. I/O Diagram

12.1 Pin Description Nomenclature

Table 12-1 describes the pin description nomenclature used in Table 12-2.

Table 12-1. Pin Description Nomenclature

Symbol	Description
I	Input pin only
O	Output pin only
I/O	Pin can be either an input or output
-	Pin must be connected as described.
S(...)	Synchronous. Inputs must meet setup and hold times relative to MemClk for proper operation. S(E) Edge sensitive input S(L) Level sensitive input
T(...)	Synchronous to TCK. S(E) Edge sensitive input S(L) Level sensitive input
A(...)	Asynchronous. Inputs may be asynchronous relative to the MemClk S(E) Edge sensitive input S(L) Level sensitive input
R(...)	While the processor's $\overline{\text{Reset}}$ pin is asserted, the pin: R(0) is driven low R(1) is driven high R(H) is pulled up to V_{DD} R(Q) is a valid output R(X) is driven to unknown state R(Z) floats
H(...)	While the processor is in the Hold state, the pin: H(1) is driven high H(0) is driven low H(Q) maintains previous state or continues to be a valid output H(Z) floats

12.2 I/O Signal Descriptions

Table 12-2 describes the PPC401GF I/O signals.

Table 12-2. PPC401GF I/O Signal Descriptions

Signal Name	Pin	I/O Type	Function
ABus28	77	O	Address bits A28:29. ABus28:29 output a partial unmultiplexed word address during the Address state. The partial word address is incremented, with each assertion of Ready, during burst transfers.
ABus29	78	R(Z) H(Z)	
ALE	76	O R(1) H(Z)	Address Latch Enable. The PPC401GF asserts this signal high at the beginning of the cycle when an address is output on the multiplexed bus. The signal goes inactive before the beginning of the Data state. For an early implementation of the 401GF (Part No. PPC401GF-MA50C2), the polarity of the ALE signal was reversed and this signal was active low ($\overline{\text{ALE}}$). For all other PPC401GF chips, ALE is active high.
B0	39	I/O S(L) R(Z) H(Z)	Multiplexed bus bit 0. This bus, B0:31, carries 32-bit physical addresses and 8-, 16-, and 32-bit data to and from memory. During the Wait/Data state, read or write data is present on one or more contiguous bytes. During write operations, unused pins continue to drive address; during read operations, unused pins are hi-Z. The determinate values are also driven on the bus during the Idle state.
B1	38	I/O	Multiplexed bus bit 1
B2	37	I/O	Multiplexed bus bit 2
B3	36	I/O	Multiplexed bus bit 3
B4	35	I/O S(L) R(Z) H(Z)	Multiplexed bus bit 4. Depending on burst mode configuration, transfer size information is output on B6:7 ($\text{BRCRn}[\text{MB}] = 0$) for a four-beat burst, or on B4:7 ($\text{BRCRn}[\text{MB}] = 1$) for continuous burst (up to 16 transfers).
B5	34	I/O	Multiplexed bus bit 5. See description of B4 above.
B6	33	I/O	Multiplexed bus bit 6. See description of B4 above.
B7	29	I/O	Multiplexed bus bit 7. See description of B4 above.
B8	49	I/O	Multiplexed bus bit 8
B9	48	I/O	Multiplexed bus bit 9
B10	47	I/O	Multiplexed bus bit 10

Table 12-2. PPC401GF I/O Signal Descriptions (cont.)

Signal Name	Pin	I/O Type	Function
B11	46	I/O	Multiplexed bus bit 11
B12	45	I/O	Multiplexed bus bit 12
B13	42	I/O	Multiplexed bus bit 13
B14	41	I/O	Multiplexed bus bit 14
B15	40	I/O	Multiplexed bus bit 15
B16	61	I/O	Multiplexed bus bit 16
B17	60	I/O	Multiplexed bus bit 17
B18	59	I/O	Multiplexed bus bit 18
B19	56	I/O	Multiplexed bus bit 19
B20	55	I/O	Multiplexed bus bit 20
B21	54	I/O	Multiplexed bus bit 21
B22	53	I/O	Multiplexed bus bit 22
B23	52	I/O	Multiplexed bus bit 23
B24	72	I/O	Multiplexed bus bit 24
B25	68	I/O	Multiplexed bus bit 25
B26	67	I/O	Multiplexed bus bit 26
B27	66	I/O	Multiplexed bus bit 27
B28	65	I/O	Multiplexed bus bit 28
B29	64	I/O	Multiplexed bus bit 29
B30	63	I/O	Multiplexed bus bit 30
B31	62	I/O	Multiplexed bus bit 31

Table 12-2. PPC401GF I/O Signal Descriptions (cont.)

Signal Name	Pin	I/O Type	Function																				
BE0[A31]	24	O R(1) H(Z)	<p>Byte Enable 0[Address 31]. BE0:3 select which (of up to four) bytes on the bus participate in the current bus access. Byte enable encoding depends on the bus width of the memory region accessed.</p> <table><thead><tr><th></th><th>BE0</th><th>BE1</th><th>BE2</th><th>BE3</th></tr></thead><tbody><tr><td>32-Bit Bus Accesses</td><td>Enables B0:7</td><td>Enables B8:15</td><td>Enables B16:23</td><td>Enables B24:31</td></tr><tr><td>16-Bit Bus Accesses</td><td>Enables B0:7</td><td>Becomes addr A30</td><td>Unused (high)</td><td>Enables B8:15</td></tr><tr><td>8-Bit Bus Accesses</td><td>Becomes addr A31</td><td>Becomes addr A31</td><td>Unused (high)</td><td>Unused (high)</td></tr></tbody></table> <p>The PPC401GF asserts byte enables during the Address state. For unaligned burst transfers, these signals switch with each assertion of Ready.</p>		BE0	BE1	BE2	BE3	32-Bit Bus Accesses	Enables B0:7	Enables B8:15	Enables B16:23	Enables B24:31	16-Bit Bus Accesses	Enables B0:7	Becomes addr A30	Unused (high)	Enables B8:15	8-Bit Bus Accesses	Becomes addr A31	Becomes addr A31	Unused (high)	Unused (high)
	BE0	BE1	BE2	BE3																			
32-Bit Bus Accesses	Enables B0:7	Enables B8:15	Enables B16:23	Enables B24:31																			
16-Bit Bus Accesses	Enables B0:7	Becomes addr A30	Unused (high)	Enables B8:15																			
8-Bit Bus Accesses	Becomes addr A31	Becomes addr A31	Unused (high)	Unused (high)																			
BE1[A30]	25	O	Byte Enable 1[Address 30]																				
BE2	27	O	Byte Enable 2																				
BE3	28	O	Byte Enable 3																				
BLast	21	O R(1) H(Z)	Burst Last. Indicates the last transfer of a memory access, whether burst or nonburst. BLast stays active as long as the Ready signal is not asserted to introduce wait states on the external bus. BLast goes inactive when the last transfer of an external bus transaction is done.																				
BootClkSpd	19	I S(L)	<p>Boot Clock Speed. BootClkSpd is sampled during reset to determine the internal clock speed and the bus clock speed.</p> <table><thead><tr><th>BootClkSpd</th><th>Internal Clock</th><th>MemClk</th></tr></thead><tbody><tr><td>0</td><td>1X of input clock</td><td>1X of internal clock</td></tr><tr><td>1</td><td>1/4X of input clock</td><td>1/4X of internal clock</td></tr></tbody></table>	BootClkSpd	Internal Clock	MemClk	0	1X of input clock	1X of internal clock	1	1/4X of input clock	1/4X of internal clock											
BootClkSpd	Internal Clock	MemClk																					
0	1X of input clock	1X of internal clock																					
1	1/4X of input clock	1/4X of internal clock																					
BootXtal	7	I S(L)	Boot Crystal. BootXtal indicates whether an oscillator or a crystal is connected to the XTAL1 input pin. For chips with the part number PPC401GF-MC50C2, tie BootXtal high.																				

Table 12-2. PPC401GF I/O Signal Descriptions (cont.)

Signal Name	Pin	I/O Type	Function
BootW	9	I S(L)	Boot-up ROM Width Select. BootW is sampled before and after the $\overline{\text{Reset}}$ pin is active to determine the width of the boot-up ROM. If this pin is tied to logic 0 when sampled on reset, an 8-bit boot width is assumed. If BootW is tied to logic 1, a 32-bit boot width is assumed. For 16-bit boot widths, this pin should be tied to the $\overline{\text{Reset}}$ pin.
$\overline{\text{BusError}}$	13	I S(L)	Bus Error Input. A logic 0 input to the $\overline{\text{BusError}}$ pin by an external device signals to the PPC401GF that an error occurred on the bus transaction. $\overline{\text{BusError}}$ is only sampled when $\overline{\text{Ready}}$ is asserted.
BusReq	22	O R(Q) H(Q)	Bus Request. While HoldAck is active, BusReq is active when the PPC401GF has a bus operation pending and needs to regain control of the bus. BusReq is also active during bus operations.
BusWidth0	79	O R(Z) H(Z)	BusWidth0:1 indicates the width of a bus transaction on the PPC401GF external multiplexed bus: 00 = 8 bit, 01 = 16 bit, 10 = 32 bit, 11 = reserved.
BusWidth1	80		
CritInt	15	I A(L)	Critical Interrupt. CritInt is a critical interrupt input to the 401GF and users may program the polarity to be active high or active low. The IOCR contains the bit necessary to program the polarity.
ExtInt	14	I A(L)	External interrupt. ExtInt is an interrupt input to the PPC401GF and users may program the polarity to be active high or active low. The IOCR contains the bit necessary to program the polarity.
GND	10	—	Ground. All ground pins must be used.
	30	—	Ground. All ground pins must be used.
	32	—	Ground. All ground pins must be used.
	43	—	Ground. All ground pins must be used.
	50	—	Ground. All ground pins must be used.
	58	—	Ground. All ground pins must be used.
	69	—	Ground. All ground pins must be used.
	71	—	Ground. All ground pins must be used.
	75	—	Ground. All ground pins must be used.

Table 12-2. PPC401GF I/O Signal Descriptions (cont.)

Signal Name	Pin	I/O Type	Function
Halt	16	I A(L)	Halt from external debugger, active low.
HoldAck	2	O R(Q) H(1)	Hold Acknowledge. HoldAck outputs a logic 1 when the PPC401GF relinquishes its external bus to an external bus master. The external bus master uses the HoldReq pin to request use of the PPC401GF bus.
HoldReq	11	I S(L)	Hold Request. External bus masters can request the PPC401GF bus by placing a logic1 on this pin. When the PPC401GF HoldAck pin is logic 1, the PPC401GF has relinquished its external bus to the external master. The external bus master relinquishes the bus to the PPC401GF by deasserting HoldReq. The PPC401GF then deasserts HoldAck during the following cycle.
MemClk	5	O R(Q) H(Q)	Memory Clock. Clock output from the internal PPC401GF clock generator. MemClk is a clock output at the bus frequency. The bus can be configured to run at 1x, 1/2x, 1/3x or 1/4x the internal clock frequency.
Ready	12	I S(L)	Ready. $\overline{\text{Ready}}$ inserts externally generated (device-paced) wait states into bus transactions. $\overline{\text{Ready}}$ indicates to the PPC401GF that data on the external bus can be sampled or removed.
Reset	73	I/O R(L)	Reset. As an input, the Reset pin is driven low for 1 ms to perform a system reset. The Reset signal, used as an output, can be driven by software or an external debug tool. When a system reset occurs, the Reset pin outputs logic 0 for up to 2048 internal clock cycles.
TCK	18	I	JTAG Test Clock Input. TCK is the clock source for the PPC401GF test access port (TAP). The maximum clock rate into the TCK pin is one half of the processor MemClk clock rate.
TDI	8	I T(L)	Test Data In. The TDI is used to input serial data into the TAP. When the TAP enables the use of the TDI pin, the TDI pin is sampled on the rising edge of TCK and this data is input to the selected TAP shift register.
TDO	1	O R(Z) H(Q) T(L)	Test Data Output. TDO is used to transmit data from the PPC401GF TAP. Data from the selected TAP shift register is shifted out on TDO.

Table 12-2. PPC401GF I/O Signal Descriptions (cont.)

Signal Name	Pin	I/O Type	Function
TestA	20	I	Reserved for manufacturing test. Tied low for normal operation.
TMS	17	I T(L)	Test Mode Select. The TMS 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.
V _{DD}	6	—	Power. All power pins must be connected to 3.3V supply.
	26	—	Power. All power pins must be connected to 3.3V supply.
	31	—	Power. All power pins must be connected to 3.3V supply.
	44	—	Power. All power pins must be connected to 3.3V supply.
	51	—	Power. All power pins must be connected to 3.3V supply.
	57	—	Power. All power pins must be connected to 3.3V supply.
V _{DD}	70	—	Power. All power pins must be connected to 3.3V supply.
	74	—	Power. All power pins must be connected to 3.3V supply.
W/ \overline{R}	23	O R(0) H(Z)	Write/Read. When the PPC401GF is bus master, W/ \overline{R} is an output (asserted only in address cycle) that is low when data is read from memory or a peripheral and high when data is written to memory or a peripheral.
XTAL1	3	—	Crystal 1 input. XTAL1 is the processor clock input. This pin can be connected to a oscillator or a crystal.
XTAL2	4	—	Crystal 2 output. XTAL2 is connected to the crystal. If XTAL1 is connected to an oscillator instead of a crystal, this pin should be left unconnected.

12.3 Signals Ordered by Pin Number

Table 12-3 describes the PPC401GF signals ordered by pin number.

Table 12-3. Signals Ordered by Pin Number

Pin	Signal Names	Pin	Signal Names	Pin	Signal Names	Pin	Signal Names
1	TDO	21	BLast	41	B14	61	B16
2	HoldAck	22	BusReq	42	B13	62	B31
3	XTAL1	23	W/R	43	GND	63	B30
4	XTAL2	24	BE0[A31]	44	V _{DD}	64	B29
5	MemClk	25	BE1[A30]	45	B12	65	B28
6	V _{DD}	26	V _{DD}	46	B11	66	B27
7	BootXtal	27	BE2	47	B10	67	B26
8	TDI	28	BE3	48	B9	68	B25
9	BootW	29	B7	49	B8	69	GND
10	GND	30	GND	50	GND	70	V _{DD}
11	HoldReq	31	V _{DD}	51	V _{DD}	71	GND
12	Ready	32	GND	52	B23	72	B24
13	BusError	33	B6	53	B22	73	Reset
14	ExtInt	34	B5	54	B21	74	V _{DD}
15	CritInt	35	B4	55	B20	75	GND
16	Halt	36	B3	56	B19	76	ALE
17	TMS	37	B2	57	V _{DD}	77	ABus28
18	TCK	38	B1	58	GND	78	ABus29
19	BootClkSpd	39	B0	59	B18	79	BusWidth0
20	TestA	40	B15	60	B17	80	BusWidth1



Instruction Summary

This appendix contains PPC401GF instructions summarized alphabetically and by opcode.

- On p. A-1, Section A.1 lists all PPC401GF mnemonics, including extended mnemonics, alphabetically. A short functional description is included for each mnemonic.
- On p. A-39, Section A.2 lists all PPC401GF instructions, sorted by primary and secondary opcodes. Extended mnemonics are not included in the opcode list.
- On p. A-47, Section A.3 illustrates the PPC401GF instruction forms (allowed arrangements of fields within instructions).

A.1 Instruction Set and Extended Mnemonics – Alphabetical

Table A-1 summarizes the PPC401GF 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 10 (Instruction Set) which is also alphabetized under the root form. Chapter 10 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. PPC401GF Instruction Syntax Summary

Mnemonic	Operands	Function	Other Registers Changed	Page
add	RT, RA, RB	Add (RA) to (RB). Place result in RT.		10-6
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].		10-7
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].		10-8
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.		10-9
addic	RT, RA, IM	Add EXTS(IM) to (RA 0). Place result in RT. Place carry-out in XER[CA].		10-10
addic.	RT, RA, IM	Add EXTS(IM) to (RA 0). Place result in RT. Place carry-out in XER[CA].	CR[CR0]	10-11
addis	RT, RA, IM	Add (IM ¹⁶ 0) to (RA 0). Place result in RT.		10-12
addme	RT, RA	Add XER[CA], (RA), (-1). Place result in RT. Place carry-out in XER[CA].		10-13
addme.			CR[CR0]	
addmeo			XER[SO, OV]	
addmeo.			CR[CR0] XER[SO, OV]	

Table A-1. PPC401GF 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].		10-14
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.		10-15
and.			CR[CR0]	
andc	RA, RS, RB	AND (RS) with \neg (RB). Place result in RA.		10-16
andc.			CR[CR0]	
andi.	RA, RS, IM	AND (RS) with $^{16}0 \parallel \text{IM}$. Place result in RA.	CR[CR0]	10-17
andis.	RA, RS, IM	AND (RS) with $(\text{IM} \parallel ^{16}0)$. Place result in RA.	CR[CR0]	10-18
b	target	Branch unconditional relative. $\text{LI} \leftarrow (\text{target} - \text{CIA})_{6:29}$ $\text{NIA} \leftarrow \text{CIA} + \text{EXTS}(\text{LI} \parallel ^{20})$		10-19
ba		Branch unconditional absolute. $\text{LI} \leftarrow \text{target}_{6:29}$ $\text{NIA} \leftarrow \text{EXTS}(\text{LI} \parallel ^{20})$		
bl		Branch unconditional relative. $\text{LI} \leftarrow (\text{target} - \text{CIA})_{6:29}$ $\text{NIA} \leftarrow \text{CIA} + \text{EXTS}(\text{LI} \parallel ^{20})$	$(\text{LR}) \leftarrow \text{CIA} + 4.$	
bla		Branch unconditional absolute. $\text{LI} \leftarrow \text{target}_{6:29}$ $\text{NIA} \leftarrow \text{EXTS}(\text{LI} \parallel ^{20})$	$(\text{LR}) \leftarrow \text{CIA} + 4.$	

Table A-1. PPC401GF 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 $BO_2 = 0$.	10-20
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 $BO_2 = 0$.	10-28
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 $BO_2 = 0$.	10-32
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		10-28
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		10-20
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. PPC401GF 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		10-32
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		10-20
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		10-32
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		10-20
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. PPC401GF 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		10-32
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		10-20
bdza		<i>Extended mnemonic for</i> bca 18,0,target		
bdzli		<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		10-32
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		10-20
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. PPC401GF 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		10-32
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		10-20
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		10-32
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		10-20
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. PPC401GF 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		10-28
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		10-32
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		10-20
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		10-28
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		10-32
bfldr		<i>Extended mnemonic for</i> bclrl 4,cr_bit	(LR) ← CIA + 4.	

Table A-1. PPC401GF 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		10-20
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.	
bgeia		<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		10-28
bgectrl		<i>Extended mnemonic for</i> bcctrl 4,4*cr_field+0	(LR) ← CIA + 4.	
bgeir	[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		10-32
bgeirl		<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		10-20
bgt a		<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. PPC401GF 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		10-28
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		10-32
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		10-20
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		10-28
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		10-32
blelrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+1	(LR) ← CIA + 4.	

Table A-1. PPC401GF 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		10-32
blrl		<i>Extended mnemonic for</i> bclrl 20,0	(LR) ← 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		10-20
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		10-28
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		10-32
bltlrl		<i>Extended mnemonic for</i> bclrl 12,4*cr_field+0	(LR) ← CIA + 4.	

Table A-1. PPC401GF 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		10-20
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.	
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		10-28
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		10-32
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		10-20
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. PPC401GF 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		10-28
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		10-32
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		10-20
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		10-28
bnctrl		<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		10-32
bnllrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+0	(LR) ← CIA + 4.	

Table A-1. PPC401GF 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		10-20
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		10-28
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		10-32
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		10-20
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. PPC401GF 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</i> bcctr 4,4*cr_field+3		10-28
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		10-32
bnulrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+3	(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		10-20
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		10-28
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		10-32
bsolrl		<i>Extended mnemonic for</i> bclrl 12,4*cr_field+3	(LR) ← CIA + 4.	

Table A-1. PPC401GF 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		10-20
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		10-28
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		10-32
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		10-20
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		10-28
bunctrl		<i>Extended mnemonic for</i> bcctrl 12,4*cr_field+3	(LR) ← CIA + 4.	

Table A-1. PPC401GF 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		10-32
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		10-132
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		10-132
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		10-132
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 $n = BF$.		10-37
cmpi	BF, 0, RA, IM	Compare (RA) to EXTS(IM), signed. Results in CR[CRn], where $n = BF$.		10-38
cmpl	BF, 0, RA, RB	Compare (RA) to (RB), unsigned. Results in CR[CRn], where $n = BF$.		10-39
cmpli	BF, 0, RA, IM	Compare (RA) to (${}^{16}0 \parallel IM$), unsigned. Results in CR[CRn], where $n = BF$.		10-40

Table A-1. PPC401GF 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		10-39
cmplwi	[BF,] RA, IM	Compare Logical Word Immediate. Use CR0 if BF is omitted. <i>Extended mnemonic for</i> cmpli BF,0,RA,IM		10-40
cmpw	[BF,] RA, RB	Compare Word. Use CR0 if BF is omitted. <i>Extended mnemonic for</i> cmp BF,0,RA,RB		10-37
cmpwi	[BF,] RA, IM	Compare Word Immediate. Use CR0 if BF is omitted. <i>Extended mnemonic for</i> cmpi BF,0,RA,IM		10-38
cntlzw	RA, RS	Count leading zeros in RS. Place result in RA.		10-41
cntlzw.			CR[CR0]	
crand	BT, BA, BB	AND bit (CR _{BA}) with (CR _{BB}). Place result in CR _{BT} .		10-42
crandc	BT, BA, BB	AND bit (CR _{BA}) with \neg (CR _{BB}). Place result in CR _{BT} .		10-43
crclr	bx	Condition register clear. <i>Extended mnemonic for</i> crxor bx,bx,bx		10-49
creqv	BT, BA, BB	Equivalence of bit CR _{BA} with CR _{BB} . $CR_{BT} \leftarrow \neg(CR_{BA} \oplus CR_{BB})$		10-44
crmove	bx, by	Condition register move. <i>Extended mnemonic for</i> cror bx,by,by		10-47
crnand	BT, BA, BB	NAND bit (CR _{BA}) with (CR _{BB}). Place result in CR _{BT} .		10-45
crnor	BT, BA, BB	NOR bit (CR _{BA}) with (CR _{BB}). Place result in CR _{BT} .		10-46
crnot	bx, by	Condition register not. <i>Extended mnemonic for</i> crnor bx,by,by		10-46

Table A-1. PPC401GF 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} .		10-47
crorc	BT, BA, BB	OR bit (CR_{BA}) with $\neg(CR_{BB})$. Place result in CR_{BT} .		10-48
crset	bx	Condition register set. <i>Extended mnemonic for creqv bx,bx,bx</i>		10-44
crxor	BT, BA, BB	XOR bit (CR_{BA}) with (CR_{BB}). Place result in CR_{BT} .		10-49
dcba	RA, RB	Speculatively establish the data cache block which contains the effective address ($RA 0$) + (RB).		10-50
dcbf	RA, RB	Flush (store, then invalidate) the data cache block which contains the effective address ($RA 0$) + (RB).		10-50
dcbi	RA, RB	Invalidate the data cache block which contains the effective address ($RA 0$) + (RB).		10-51
dcbst	RA, RB	Store the data cache block which contains the effective address ($RA 0$) + (RB).		10-52
dcbt	RA, RB	Load the data cache block which contains the effective address ($RA 0$) + (RB).		10-53
dcbtst	RA, RB	Load the data cache block which contains the effective address ($RA 0$) + (RB).		10-55
dcbz	RA, RB	Zero the data cache block which contains the effective address ($RA 0$) + (RB).		10-57
dccci	RA, RB	Invalidate the data cache congruence class associated with the effective address ($RA 0$) + (RB).		10-59
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.		10-61

Table A-1. PPC401GF Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
divw	RT, RA, RB	Divide (RA) by (RB), signed. Place result in RT.		10-63
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.		10-64
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.		10-65
eqv	RA, RS, RB	Equivalence of (RS) with (RB). $(RA) \leftarrow \neg((RS) \oplus (RB))$		10-66
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		10-132
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		10-132
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.		10-67
extsb.			CR[CR0]	

Table A-1. PPC401GF 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.		10-68
extsh.			CR[CR0]	
icbi	RA, RB	Invalidate the instruction cache block which contains the effective address (RA 0) + (RB).		10-69
icbt	RA, RB	Load the instruction cache block which contains the effective address (RA 0) + (RB).		10-71
iccci	RA, RB	Invalidate instruction cache congruence class associated with the effective address (RA 0) + (RB).		10-73
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.		10-74
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		10-131
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		10-131
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.		10-76
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		10-9

Table A-1. PPC401GF 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).		10-78
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.		10-79
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.		10-80
lbzx	RT, RA, RB	Load byte from EA = (RA 0) + (RB) and pad left with zeroes, (RT) \leftarrow $^{24}0$ MS(EA,1).		10-81
lha	RT, D(RA)	Load halfword from EA = (RA 0) + EXTS(D) and sign extend, (RT) \leftarrow EXTS(MS(EA,2)).		10-82
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.		10-83
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.		10-84
lhax	RT, RA, RB	Load halfword from EA = (RA 0) + (RB) and sign extend, (RT) \leftarrow EXTS(MS(EA,2)).		10-85
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).		10-86
lhz	RT, D(RA)	Load halfword from EA = (RA 0) + EXTS(D) and pad left with zeroes, (RT) \leftarrow $^{16}0$ MS(EA,2).		10-87

Table A-1. PPC401GF Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
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.		10-88
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.		10-89
lhzx	RT, RA, RB	Load halfword from EA = (RA 0) + (RB) and pad left with zeroes, (RT) \leftarrow $^{16}0 \parallel$ MS(EA,2).		10-90
li	RT, IM	Load immediate. (RT) \leftarrow EXTS(IM) <i>Extended mnemonic for</i> addi RT,0,value		10-9
lis	RT, IM	Load immediate shifted. (RT) \leftarrow (IM \parallel $^{16}0$) <i>Extended mnemonic for</i> addis RT,0,value		10-12
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).		10-91
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} .		10-92

Table A-1. PPC401GF 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.		10-94
lwarx	RT, RA, RB	Load word from $EA = (RA 0) + (RB)$ and place in RT, $(RT) \leftarrow MS(EA, 4)$. Set the Reservation bit.		10-96
lwbx	RT, RA, RB	Load word from $EA = (RA 0) + (RB)$ then reverse byte order, $(RT) \leftarrow MS(EA+3, 1) \parallel MS(EA+2, 1) \parallel MS(EA+1, 1) \parallel MS(EA, 1)$.		10-98
lwz	RT, D(RA)	Load word from $EA = (RA 0) + EXTS(D)$ and place in RT, $(RT) \leftarrow MS(EA, 4)$.		10-99
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, $(RA) \leftarrow EA$.		10-100
lwzux	RT, RA, RB	Load word from $EA = (RA 0) + (RB)$ and place in RT, $(RT) \leftarrow MS(EA, 4)$. Update the base address, $(RA) \leftarrow EA$.		10-101
lwzx	RT, RA, RB	Load word from $EA = (RA 0) + (RB)$ and place in RT, $(RT) \leftarrow MS(EA, 4)$.		10-102
mcrf	BF, BFA	Move CR field, $(CR[CRn]) \leftarrow (CR[CRm])$ where $m \leftarrow BFA$ and $n \leftarrow BF$.		10-103
mcrxr	BF	Move XER[0:3] into field CRn, where $n \leftarrow BF$. $CR[CRn] \leftarrow (XER[SO, OV, CA])$. $(XER[SO, OV, CA]) \leftarrow {}^3_0$.		10-104

Table A-1. PPC401GF Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
mfcrr	RT	Move from CR to RT, (RT) \leftarrow (CR).		10-105
mfbrar mfbrar0 mfbrar1 mfbrar2 mfbrar3 mfbrar4 mfbrar5 mfbrar6 mfbrar7 mficrr mfpmcr0	RT	Move from device control register DCRN. <i>Extended mnemonic for</i> mfdr RT,DCRN See Table 11-3 on p. 11-4 for listing of valid DCRN values.		10-106
mfdr	RT, DCRN	Move from DCR to RT, (RT) \leftarrow (DCR(DCRN)).		10-106
mfmsr	RT	Move from MSR to RT, (RT) \leftarrow (MSR).		10-108

Table A-1. PPC401GF Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
mfcdbcr mfctr mfdac1 mfdbsr mfdccr mfdcwr mfdear mfesr mfevpr mfiac1 mficcr mficdbdr mflr mfpit mfpvr mfsgsr mfspgr0 mfspgr1 mfspgr2 mfspgr3 mfssr0 mfssr1 mfssr2 mfssr3 mftbhi mftbhu mftblo mftblu mftcr mftsr mfxer	RT	Move from special purpose register (SPR) SPRN. <i>Extended mnemonic for</i> mfssr RT,SPRN See Table 11-2 on p. 11-3 for listing of valid SPRN values.		10-109
mfssr	RT, SPRN	Move from SPR to RT, (RT) \leftarrow (SPR(SPRN)).		10-109
mr	RT, RS	Move register. (RT) \leftarrow (RS) <i>Extended mnemonic for</i> or RT,RS,RS		10-125
mr.		<i>Extended mnemonic for</i> or. RT,RS,RS	CR[CR0]	
mtrcr	RS	Move to Condition Register. <i>Extended mnemonic for</i> mtrcrf 0xFF,RS		10-111

Table A-1. PPC401GF Instruction Syntax Summary (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, $\text{mask} \leftarrow {}^4(\text{FXM}_0) \parallel {}^4(\text{FXM}_1) \parallel \dots \parallel {}^4(\text{FXM}_6) \parallel {}^4(\text{FXM}_7)$. $(\text{CR}) \leftarrow ((\text{RS}) \wedge \text{mask}) \vee (\text{CR}) \wedge \neg \text{mask}$.		10-111
mtbear mtbesr0 mtbrcr0 mtbrcr1 mtbrcr2 mtbrcr3 mtbrcr4 mtbrcr5 mtbrcr6 mtbrcr7 mtiocr mtpmcr0	RS	Move to device control register DCRN. <i>Extended mnemonic for</i> mtdcr DCRN,RS See Table 11-3 on p. 11-4 for listing of valid DCRN values.		10-113
mtdcr	DCRN, RS	Move to DCR from RS, $(\text{DCR}(\text{DCRN})) \leftarrow (\text{RS})$.		10-113
mtmsr	RS	Move to MSR from RS, $(\text{MSR}) \leftarrow (\text{RS})$.		10-115

Table A-1. PPC401GF Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
mtcdbc mtctr mtdac1 mtdbc mtdbsr mtdccr mtdcwr mtesr mtcvpr mtiac1 mticcr mticdbdr mtlr mtpit mtpvr mtsg mtsprg0 mtsprg1 mtsprg2 mtsprg3 mtsrr0 mtsrr1 mtsrr2 mtsrr3 mttbhi mttblo mttcr mttsr mtxer	RS	Move to SPR SPRN. <i>Extended mnemonic for</i> mtspr SPRN,RS See Table 11-2 on p. 11-3 for listing of valid SPRN values.		10-116
mtspr	SPRN, RS	Move to SPR from RS, (SPR(SPRN)) \leftarrow (RS).		10-116
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) $\leftarrow \text{prod}_{0:31}$.		10-118
mulhw.			CR[CR0]	
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) $\leftarrow \text{prod}_{0:31}$.		10-119
mulhwu.			CR[CR0]	
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) $\leftarrow \text{prod}_{16:47}$		10-120

Table A-1. PPC401GF 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). $(\text{RT}) \leftarrow \text{prod}_{32:63}$.		10-121
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.		10-122
nand.			CR[CR0]	
neg	RT, RA	Negative (two's complement) of RA. $(\text{RT}) \leftarrow \neg(\text{RA}) + 1$		10-123
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		10-127
nor	RA, RS, RB	NOR (RS) with (RB). Place result in RA.		10-124
nor.			CR[CR0]	
not	RA, RS	Complement register. $(\text{RA}) \leftarrow \neg(\text{RS})$ <i>Extended mnemonic for</i> nor RA,RS,RS		10-124
not.		<i>Extended mnemonic for</i> nor. RA,RS,RS	CR[CR0]	
or	RA, RS, RB	OR (RS) with (RB). Place result in RA.		10-125
or.			CR[CR0]	
orc	RA, RS, RB	OR (RS) with $\neg(\text{RB})$. Place result in RA.		10-126
orc.			CR[CR0]	
ori	RA, RS, IM	OR (RS) with ($^{16}0 \parallel \text{IM}$). Place result in RA.		10-127
oris	RA, RS, IM	OR (RS) with ($\text{IM} \parallel ^{16}0$). Place result in RA.		10-128

Table A-1. PPC401GF Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
rfci		Return from critical interrupt (PC) \leftarrow (SRR2). (MSR) \leftarrow (SRR3).		10-129
rfi		Return from interrupt. (PC) \leftarrow (SRR0). (MSR) \leftarrow (SRR1).		10-130
rlwimi	RA, RS, SH, MB, ME	Rotate left word immediate, then insert according to mask. $r \leftarrow \text{ROTL}((RS), SH)$ $m \leftarrow \text{MASK}(MB, ME)$ $(RA) \leftarrow (r \wedge m) \vee ((RA) \wedge \neg m)$		10-131
rlwimi.			CR[CR0]	
rlwinm	RA, RS, SH, MB, ME	Rotate left word immediate, then AND with mask. $r \leftarrow \text{ROTL}((RS), SH)$ $m \leftarrow \text{MASK}(MB, ME)$ $(RA) \leftarrow (r \wedge m)$		10-132
rlwinm.			CR[CR0]	
rlwnm	RA, RS, RB, MB, ME	Rotate left word, then AND with mask. $r \leftarrow \text{ROTL}((RS), (RB)_{27:31})$ $m \leftarrow \text{MASK}(MB, ME)$ $(RA) \leftarrow (r \wedge m)$		10-135
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		10-135
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		10-132
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		10-132
rotrwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,32-n,0,31	CR[CR0]	

Table A-1. PPC401GF Instruction Syntax Summary (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
sc		System call exception is generated. $(SRR1) \leftarrow (MSR)$ $(SRR0) \leftarrow (PC)$ $PC \leftarrow EVPR_{0:15} \parallel x'0C00'$ $(MSR[WE, PR, EE, PE, DR, IR]) \leftarrow 0$ $(MSR[LE]) \leftarrow (MSR[ILE])$		10-136
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 $m \leftarrow {}^{32}0$. $(RA) \leftarrow r \wedge m$.		10-137
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		10-132
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 $m \leftarrow {}^{32}0$. $s \leftarrow (RS)_0$. $(RA) \leftarrow (r \wedge m) \vee ({}^{32}s \wedge \neg m)$. $XER[CA] \leftarrow s \wedge ((r \wedge \neg m) \neq 0)$.		10-138
sraw.			CR[CR0]	
srawi	RA, RS, SH	Shift right algebraic (RS) by SH. $n \leftarrow SH$. $r \leftarrow ROTL((RS), 32 - n)$. $m \leftarrow MASK(n, 31)$. $s \leftarrow (RS)_0$. $(RA) \leftarrow (r \wedge m) \vee ({}^{32}s \wedge \neg m)$. $XER[CA] \leftarrow s \wedge ((r \wedge \neg m) \neq 0)$.		10-139
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 $m \leftarrow {}^{32}0$. $(RA) \leftarrow r \wedge m$.		10-140
srw.			CR[CR0]	

Table A-1. PPC401GF 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		10-132
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 $EA = (RA 0) + EXTS(D)$.		10-141
stbu	RS, D(RA)	Store byte $(RS)_{24:31}$ in memory at $EA = (RA 0) + EXTS(D)$. Update the base address, $(RA) \leftarrow EA$.		10-142
stbux	RS, RA, RB	Store byte $(RS)_{24:31}$ in memory at $EA = (RA 0) + (RB)$. Update the base address, $(RA) \leftarrow EA$.		10-143
stbx	RS, RA, RB	Store byte $(RS)_{24:31}$ in memory at $EA = (RA 0) + (RB)$.		10-144
sth	RS, D(RA)	Store halfword $(RS)_{16:31}$ in memory at $EA = (RA 0) + EXTS(D)$.		10-145
sthbrx	RS, RA, RB	Store halfword $(RS)_{16:31}$ byte-reversed in memory at $EA = (RA 0) + (RB)$. $MS(EA, 2) \leftarrow (RS)_{24:31} \parallel (RS)_{16:23}$		10-146
sthu	RS, D(RA)	Store halfword $(RS)_{16:31}$ in memory at $EA = (RA 0) + EXTS(D)$. Update the base address, $(RA) \leftarrow EA$.		10-147
sthux	RS, RA, RB	Store halfword $(RS)_{16:31}$ in memory at $EA = (RA 0) + (RB)$. Update the base address, $(RA) \leftarrow EA$.		10-148
sthx	RS, RA, RB	Store halfword $(RS)_{16:31}$ in memory at $EA = (RA 0) + (RB)$.		10-149
stmw	RS, D(RA)	Store consecutive words from RS through GPR(31) in memory starting at $EA = (RA 0) + EXTS(D)$.		10-150

Table A-1. PPC401GF 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).		10-151
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).		10-152
stw	RS, D(RA)	Store word (RS) in memory at EA = (RA 0) + EXTSD(D).		10-154
stwbrx	RS, RA, RB	Store word (RS) byte-reversed in memory at EA = (RA 0) + (RB). $MS(EA, 4) \leftarrow (RS)_{24:31} \parallel (RS)_{16:23} \parallel (RS)_{8:15} \parallel (RS)_{0:7}$		10-155
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 $(CR[CR0]) \leftarrow {}^20 \parallel 0 \parallel XER_{so}$.		10-156
stwu	RS, D(RA)	Store word (RS) in memory at EA = (RA 0) + EXTSD(D). Update the base address, (RA) \leftarrow EA.		10-158
stwux	RS, RA, RB	Store word (RS) in memory at EA = (RA 0) + (RB). Update the base address, (RA) \leftarrow EA.		10-159
stwx	RS, RA, RB	Store word (RS) in memory at EA = (RA 0) + (RB).		10-160

Table A-1. PPC401GF 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>		10-161
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>		10-162
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). $(RT) \leftarrow \neg(RA) + (RB) + 1.$		10-161
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].		10-162
subfc.			CR[CR0]	
subfco			XER[SO, OV]	
subfco.			CR[CR0] XER[SO, OV]	

Table A-1. PPC401GF 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].		10-164
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].		10-165
subfme	RT, RA, RB	Subtract (RA) from (–1) with carry-in. (RT) $\leftarrow \neg(RA) + (–1) + XER[CA]$. Place carry-out in XER[CA].		10-166
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].		10-167
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		10-9
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		10-10
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]	10-11
subis	RT, RA, IM	Subtract (IM ¹⁶ 0) from (RA 0). Place result in RT. <i>Extended mnemonic for</i> addis RT,RA,–IM		10-12

Table A-1. PPC401GF 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.		10-168

Table A-1. PPC401GF 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.		10-169

Table A-1. PPC401GF 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. PPC401GF 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]).		10-177
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-50) in bits 0:5. Some instructions also have a secondary opcode field (shown as field XO in Figure A-1 through Figure A-9). PPC401GF 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-47 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. PPC401GF Instructions by Opcode

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
3		D	twi	TO, RA, IM	
7		D	mulli	RT, RA, IM	10-120
8		D	subfic	RT, RA, IM	10-165
10		D	cmpli	BF, 0, RA, IM	10-40

A

Table A-2. PPC401GF Instructions by Opcode (cont.)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
11		D	cmpi	BF, 0, RA, IM	10-38
12		D	addic	RT, RA, IM	10-10
13		D	addic.	RT, RA, IM	10-11
14		D	addi	RT, RA, IM	10-9
15		D	addis	RT, RA, IM	10-12
16		B	bc	BO, BI, target	10-20
			bca		
			bcl		
			bcla		
17		SC	sc		10-136
18		I	b	target	10-19
			ba		
			bl		
			bla		
19	0	XL	mcrf	BF, BFA	10-103
19	16	XL	bclr	BO, BI	10-32
			bclrl		
19	33	XL	crnor	BT, BA, BB	10-46
19	50	XL	rfi		10-130
19	51	XL	rfdi		10-129
19	129	XL	crandc	BT, BA, BB	10-43
19	150	XL	isync		10-76
19	193	XL	crxor	BT, BA, BB	10-49
19	225	XL	crnand	BT, BA, BB	10-45
19	257	XL	crand	BT, BA, BB	10-42
19	289	XL	creqv	BT, BA, BB	10-44
19	417	XL	crorc	BT, BA, BB	10-48
19	449	XL	cror	BT, BA, BB	10-47

Table A-2. PPC401GF Instructions by Opcode (cont.)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
19	528	XL	bcctr	BO, BI	10-28
			bcctrl		
20		M	rlwimi	RA, RS, SH, MB, ME	10-131
			rlwimi.		
21		M	rlwinm	RA, RS, SH, MB, ME	10-132
			rlwinm.		
23		M	rlwnm	RA, RS, RB, MB, ME	10-135
			rlwnm.		
24		D	ori	RA, RS, IM	10-127
25		D	oris	RA, RS, IM	10-128
26		D	xori	RA, RS, IM	
27		D	xoris	RA, RS, IM	
28		D	andi.	RA, RS, IM	10-17
29		D	andis.	RA, RS, IM	10-18
31	0	X	cmp	BF, 0, RA, RB	10-37
31	4	X	tw	TO, RA, RB	
31	8 (520)	XO	subfc	RT, RA, RB	10-162
			subfc.		
			subfco		
			subfco.		
31	10 (522)	XO	addc	RT, RA, RB	10-7
			addc.		
			addco		
			addco.		
31	11 (523)	XO	mulhwu	RT, RA, RB	10-119
			mulhwu.		
31	19	X	mfcrr	RT	10-105
31	20	X	lwarx	RT, RA, RB	10-96
31	23	X	lwzx	RT, RA, RB	10-102

Table A-2. PPC401GF Instructions by Opcode (cont.)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
31	24	X	slw	RA, RS, RB	10-137
			slw.		
31	26	X	cntlzw	RA, RS	10-41
			cntlzw.		
31	28	X	and	RA, RS, RB	10-15
			and.		
31	32	X	cmpl	BF, 0, RA, RB	10-39
31	40 (552)	XO	subf	RT, RA, RB	10-161
			subf.		
			subfo		
			subfo.		
31	54	X	dcbst	RA, RB	10-52
31	55	X	lwzux	RT, RA, RB	10-101
31	60	X	andc	RA, RS, RB	10-16
			andc.		
31	75 (587)	XO	mulhw	RT, RA, RB	10-118
			mulhw.		
31	83	X	mfmsr	RT	10-108
31	86	X	dcbf	RA, RB	10-50
31	87	X	lbzx	RT, RA, RB	10-81
31	104 (616)	XO	neg	RT, RA	10-123
			neg.		
			nego		
			nego.		
31	119	X	lbzux	RT, RA, RB	10-80
31	124	X	nor	RA, RS, RB	10-124
			nor.		
31	131	X	wrttee	RS	

Table A-2. PPC401GF Instructions by Opcode (cont.)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
31	136 (648)	XO	subfe	RT, RA, RB	10-164
			subfe.		
			subfeo		
			subfeo.		
31	138 (650)	XO	adde	RT, RA, RB	10-8
			adde.		
			addeo		
			addeo.		
31	144	XFX	mtcrf	FXM, RS	10-111
31	146	X	mtmsr	RS	10-115
31	150	X	stwcx.	RS, RA, RB	10-156
31	151	X	stwx	RS, RA, RB	10-160
31	163	X	wrtnei	E	
31	183	X	stwux	RS, RA, RB	10-159
31	200 (712)	XO	subfze	RT, RA, RB	10-167
			subfze.		
			subfzeo		
			subfzeo.		
31	202 (714)	XO	addze	RT, RA	10-14
			addze.		
			addzeo		
			addzeo.		
31	215	X	stbx	RS, RA, RB	10-144
31	232 (744)	XO	subfme	RT, RA, RB	10-166
			subfme.		
			subfmeo		
			subfmeo.		

Table A-2. PPC401GF Instructions by Opcode (cont.)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
31	234 (746)	XO	addme	RT, RA	10-13
			addme.		
			addmeo		
			addmeo.		
31	235 (747)	XO	mullw	RT, RA, RB	10-121
			mullw.		
			mullwo		
			mullwo.		
31	246	X	dcbtst	RA,RB	10-55
31	247	X	stbux	RS, RA, RB	10-143
31	262	X	icbt	RA, RB	10-71
31	266 (778)	XO	add	RT, RA, RB	10-6
			add.		
			addo		
			addo.		
31	278	X	dcbt	RA, RB	10-53
31	279	X	lhzx	RT, RA, RB	10-90
31	284	X	eqv	RA, RS, RB	10-66
			eqv.		
31	311	X	lhzux	RT, RA, RB	10-89
31	316	X	xor	RA, RS, RB	
			xor.		
31	323	XFX	mfdcr	RT, DCRN	10-106
31	339	XFX	mfspr	RT, SPRN	10-109
31	343	X	lhax	RT, RA, RB	10-85
31	375	X	lhaux	RT, RA, RB	10-84
31	407	X	sthx	RS, RA, RB	10-149
31	412	X	orc	RA, RS, RB	10-126
			orc.		

Table A-2. PPC401GF Instructions by Opcode (cont.)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
31	439	X	sthux	RS, RA, RB	10-148
31	444	X	or	RA, RS, RB	10-125
			or.		
31	451	XFX	mtdcr	DCRN, RS	10-113
31	454	X	dccci	RA, RB	10-59
31	459 (971)	XO	divwu	RT, RA, RB	10-64
			divwu.		
			divwuo		
			divwuo.		
31	467	XFX	mtspr	SPRN, RS	10-116
31	470	X	dcbi	RA, RB	10-51
31	476	X	nand	RA, RS, RB	10-122
			nand.		
31	486	X	dcread	RT, RA, RB	10-61
31	491 (1003)	XO	divw	RT, RA, RB	10-63
			divw.		
			divwo		
			divwo.		
31	512	X	mcrxr	BF	10-104
31	533	X	lswx	RT, RA, RB	10-94
31	534	X	lwbrx	RT, RA, RB	10-98
31	536	X	srw	RA, RS, RB	10-140
			srw.		
31	597	X	lswi	RT, RA, NB	10-92
31	598	X	sync		10-168
31	661	X	stswx	RS, RA, RB	10-152
31	662	X	stwbrx	RS, RA, RB	10-155
31	725	X	stswi	RS, RA, NB	10-151
31	790	X	lhbrx	RT, RA, RB	10-86

Table A-2. PPC401GF Instructions by Opcode (cont.)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
31	792	X	sraw	RA, RS, RB	10-138
			sraw.		
31	824	X	srawi	RA, RS, SH	10-139
			srawi.		
31	854	X	eieio		10-65
31	918	X	sthbrx	RS, RA, RB	10-146
31	922	X	extsh	RA, RS	10-68
			extsh.		
31	954	X	extsb	RA, RS	10-67
			extsb.		
31	966	X	iccci	RA, RB	10-73
31	982	X	icbi	RA, RB	10-69
31	998	X	icread	RA, RB	10-74
31	1014	X	dcbz	RA, RB	10-57
31	TBD	X	dcba	RA, RB	10-50
32		D	lwz	RT, D(RA)	10-99
33		D	lwzu	RT, D(RA)	10-100
34		D	lbz	RT, D(RA)	10-78
35		D	lbzu	RT, D(RA)	10-79
36		D	stw	RS, D(RA)	10-154
37		D	stwu	RS, D(RA)	10-158
38		D	stb	RS, D(RA)	10-141
39		D	stbu	RS, D(RA)	10-142
40		D	lhz	RT, D(RA)	10-87
41		D	lhzu	RT, D(RA)	10-88
42		D	lha	RT, D(RA)	10-82
43		D	lhau	RT, D(RA)	10-83
44		D	sth	RS, D(RA)	10-145
45		D	sthu	RS, D(RA)	10-147

Table A-2. PPC401GF Instructions by Opcode (cont.)

Primary Opcode	Secondary Opcode	Form	Mnemonic	Operands	Page
46		D	lmw	RT, D(RA)	10-91
47		D	stmw	RS, D(RA)	10-150

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 PPC401GF executes all invalid instruction forms without causing an illegal instruction exception.

A.3.1 Instruction Fields

PPC401GF 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. 2-11, 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.3.2 Instruction Format Diagrams

The “Forms” shown in Figure A-1 through Figure A-9 are valid combinations of instruction fields for the PPC401GF. Table A-2 on p. A-39 indicates which “Form” is utilized by each PPC401GF 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															30	31

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

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		XO	/
OPCD	BF	//	BFA		//	///		XO	/
OPCD	BF	//	///		U	/	XO		Rc
OPCD	BF	//	///		///		XO		/
OPCD	TO		RA		RB		XO		/
OPCD	BT		///		///		XO		Rc
OPCD	///		RA		RB		XO		/
OPCD	///		///		RB		XO		/
OPCD	///		///		///		XO		/
OPCD	///		///		E	//	XO		/
0	6	11	16	21	31				

Figure A-5. X Instruction Format

A

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	/	XO	/
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

A

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

B

Instructions By Category

B.1 Instruction Set Summary Categories

Chapter 10, “Instruction Set,” contains detailed descriptions of the instructions, their operands, and notation.

Table B.1 summarizes the instruction categories in the PPC401GF instruction set. The instructions within each category are listed in subsequent tables.

Table B-1. PPC401GF 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 PPC401GF 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).		10-59
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.		10-61
icbt	RA, RB	Load the instruction cache block which contains the effective address (RA 0) + (RB).		10-71
iccci	RA, RB	Invalidate instruction cache congruence class associated with the effective address (RA 0) + (RB).		10-73
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.		10-74
mfdcr	RT, DCRN	Move from DCR to RT, (RT) \leftarrow (DCR(DCRN)).		10-106
mtdcr	DCRN, RS	Move to DCR from RS, (DCR(DCRN)) \leftarrow (RS).		10-113
rfei		Return from critical interrupt (PC) \leftarrow (SRR2). (MSR) \leftarrow (SRR3).		10-129
wrttee	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).		10-51
dccci	RA, RB	Invalidate the data cache congruence class associated with the effective address (RA 0) + (RB).		10-59
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.		10-61
icbt	RA, RB	Load the instruction cache block which contains the effective address (RA 0) + (RB).		10-71
iccci	RA, RB	Invalidate instruction cache congruence class associated with the effective address (RA 0) + (RB).		10-73
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.		10-74
mfdcr	RT, DCRN	Move from DCR to RT, (RT) ← (DCR(DCRN)).		10-106
mfmsr	RT	Move from MSR to RT, (RT) ← (MSR).		10-108
mfspr	RT, SPRN	Move from SPR to RT, (RT) ← (SPR(SPRN)). Privileged for all SPRs except LR, CTR, TBHU, TBLU, and XER.		10-109
mtdcr	DCRN, RS	Move to DCR from RS, (DCR(DCRN)) ← (RS).		10-113
mtmsr	RS	Move to MSR from RS, (MSR) ← (RS).		10-115

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.		10-116
rfci		Return from critical interrupt (PC) \leftarrow (SRR2). (MSR) \leftarrow (SRR3).		10-129
rfi		Return from interrupt. (PC) \leftarrow (SRR0). (MSR) \leftarrow (SRR1).		10-130
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 PPC401GF.

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 PPC401GF

Mnemonic	Operands	Function	Other Registers Changed	Page
bctr		Branch unconditionally, to address in CTR. <i>Extended mnemonic for bcctr 20,0</i>		10-28
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>		10-20
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>		10-32
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>		10-20
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 PPC401GF (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		10-32
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		10-20
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		10-32
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		10-20
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 PPC401GF (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		10-32
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		10-20
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		10-32
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		10-20
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 PPC401GF (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		10-32
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		10-20
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		10-28
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		10-32
beqlrl		<i>Extended mnemonic for</i> bclrl 12,4*cr_field+2	(LR) ← CIA + 4.	

Table B-4. Extended Mnemonics for PPC401GF (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		10-20
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		10-28
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		10-32
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		10-20
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) ← CIA + 4.	
bgeia		<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		10-28
bgectrl		<i>Extended mnemonic for</i> bcctrl 4,4*cr_field+0	(LR) ← CIA + 4.	

Table B-4. Extended Mnemonics for PPC401GF (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		10-32
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		10-20
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		10-28
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		10-32
bgtlrl		<i>Extended mnemonic for</i> bclrl 12,4*cr_field+1	(LR) ← CIA + 4.	

Table B-4. Extended Mnemonics for PPC401GF (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		10-20
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		10-28
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		10-32
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		10-32
blrl		<i>Extended mnemonic for</i> bclrl 20,0	(LR) ← CIA + 4.	

Table B-4. Extended Mnemonics for PPC401GF (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		10-20
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		10-28
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		10-32
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		10-20
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 PPC401GF (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		10-28
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		10-32
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		10-20
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		10-28
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		10-32
bnglrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+1	(LR) ← CIA + 4.	

Table B-4. Extended Mnemonics for PPC401GF (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		10-20
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		10-28
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		10-32
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		10-20
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 PPC401GF (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		10-28
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		10-32
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		10-20
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		10-28
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		10-32
bnulrl		<i>Extended mnemonic for</i> bclrl 4,4*cr_field+3	(LR) ← CIA + 4.	

Table B-4. Extended Mnemonics for PPC401GF (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		10-20
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		10-28
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		10-32
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		10-20
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 PPC401GF (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		10-28
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		10-32
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		10-20
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		10-28
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		10-32
bunlrl		<i>Extended mnemonic for</i> bclrl 12,4*cr_field+3	(LR) ← CIA + 4.	

Table B-4. Extended Mnemonics for PPC401GF (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		10-132
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		10-132
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		10-132
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		10-39
cmplwi	[BF,] RA, IM	Compare Logical Word Immediate. Use CR0 if BF is omitted. <i>Extended mnemonic for</i> cmpli BF,0,RA,IM		10-40
cmpw	[BF,] RA, RB	Compare Word. Use CR0 if BF is omitted. <i>Extended mnemonic for</i> cmp BF,0,RA,RB		10-37
cmpwi	[BF,] RA, IM	Compare Word Immediate. Use CR0 if BF is omitted. <i>Extended mnemonic for</i> cmpi BF,0,RA,IM		10-38
crclr	bx	Condition register clear. <i>Extended mnemonic for</i> crxor bx,bx,bx		10-49

Table B-4. Extended Mnemonics for PPC401GF (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
crmove	bx, by	Condition register move. <i>Extended mnemonic for</i> cror bx,by,by		10-47
crnot	bx, by	Condition register not. <i>Extended mnemonic for</i> crnor bx,by,by		10-46
crset	bx	Condition register set. <i>Extended mnemonic for</i> creqv bx,bx,bx		10-44
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		10-132
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		10-132
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		10-131
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		10-131
insrwi.		<i>Extended mnemonic for</i> rlwimi. RA,RS,32-b-n,b,b+n-1	CR[CR0]	

Table B-4. Extended Mnemonics for PPC401GF (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		10-9
li	RT, IM	Load immediate. $(RT) \leftarrow \text{EXTS}(IM)$ <i>Extended mnemonic for</i> addi RT,0,value		10-9
lis	RT, IM	Load immediate shifted. $(RT) \leftarrow (IM \parallel 160)$ <i>Extended mnemonic for</i> addis RT,0,value		10-12
mbear mbesr0 mbrcr0 mbrcr1 mbrcr2 mbrcr3 mbrcr4 mbrcr5 mbrcr6 mbrcr7 mfiocr mfpocr0	RT	Move from device control register DCRN. <i>Extended mnemonic for</i> mfdcr RT,DCRN See Table 11-3 on p. 11-4 for listing of valid DCRN values.		10-106

Table B-4. Extended Mnemonics for PPC401GF (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
mfcdbcr mfctr mfdac1 mfdbsr mfdccr mfdcwr mfdear mfesr mfevpr mfiac1 mficcr mficdbdr mflr mfpit mfpvr mfsgr 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> mfspr RT,SPRN See Table 11-2 on p. 11-3 for listing of valid SPRN values.		10-109
mr	RT, RS	Move register. (RT) ← (RS) <i>Extended mnemonic for</i> or RT,RS,RS		10-125
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		10-111

Table B-4. Extended Mnemonics for PPC401GF (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
mtbear mtbesr0 mtbcr0 mtbcr1 mtbcr2 mtbcr3 mtbcr4 mtbcr5 mtbcr6 mtbcr7 mtiocr mtpmcr0	RS	Move to device control register DCRN. <i>Extended mnemonic for</i> mtdcr DCRN,RS See Table 11-3 on p. 11-4 for listing of valid DCRN values.		10-113
mtcdbcr mtctr mtdac1 mtdbcr mtdbsr mtdccr mtdcwr mtesr mtevpr mtiac1 mticcr mticdbdr mtlr mtpit mtpvr mtsgr mtsprg0 mtsprg1 mtsprg2 mtsprg3 mtsrr0 mtsrr1 mtsrr2 mtsrr3 mttbhi mttblo mttcr mttsr mtxer	RS	Move to SPR SPRN. <i>Extended mnemonic for</i> mtspr SPRN,RS See Table 11-2 on p. 11-3 for listing of valid SPRN values.		10-116
nop		Preferred no-op, triggers optimizations based on no-ops. <i>Extended mnemonic for</i> ori 0,0,0		10-127

Table B-4. Extended Mnemonics for PPC401GF (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
not	RA, RS	Complement register. $(RA) \leftarrow \neg(RS)$ <i>Extended mnemonic for</i> nor RA,RS,RS		10-124
not.		<i>Extended mnemonic for</i> nor. RA,RS,RS	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		10-135
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		10-132
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		10-132
rotrwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,32-n,0,31	CR[CR0]	
slwi	RA, RS, n	Shift left immediate. ($n < 32$) $(RA)_{0:31-n} \leftarrow (RS)_{n:31}$ $(RA)_{32-n:31} \leftarrow {}^n0$ <i>Extended mnemonic for</i> rlwinm RA,RS,n,0,31-n		10-132
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		10-132
srwi.		<i>Extended mnemonic for</i> rlwinm. RA,RS,32-n,n,31	CR[CR0]	

Table B-4. Extended Mnemonics for PPC401GF (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>		10-161
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>		10-162
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>		10-9
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>		10-10
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]	10-11
subis	RT, RA, IM	Subtract (IM ¹⁶ 0) from (RA 0). Place result in RT. <i>Extended mnemonic for addis RT,RA,-IM</i>		10-12

Table B-4. Extended Mnemonics for PPC401GF (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 PPC401GF 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 PPC401GF.

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 $^{24}0$ MS(EA,1).		10-78
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.		10-79
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.		10-80
lbzx	RT, RA, RB	Load byte from EA = (RA 0) + (RB) and pad left with zeroes, (RT) \leftarrow $^{24}0$ MS(EA,1).		10-81
lha	RT, D(RA)	Load halfword from EA = (RA 0) + EXTS(D) and sign extend, (RT) \leftarrow EXTS(MS(EA,2)).		10-82
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.		10-83
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.		10-84
lhax	RT, RA, RB	Load halfword from EA = (RA 0) + (RB) and sign extend, (RT) \leftarrow EXTS(MS(EA,2)).		10-85

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, $(RT) \leftarrow {}^{16}0 \parallel MS(EA+1,1) \parallel MS(EA,1)$.		10-86
lhz	RT, D(RA)	Load halfword from EA = (RA 0) + EXTS(D) and pad left with zeroes, $(RT) \leftarrow {}^{16}0 \parallel MS(EA,2)$.		10-87
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, $(RA) \leftarrow EA$.		10-88
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, $(RA) \leftarrow EA$.		10-89
lhzx	RT, RA, RB	Load halfword from EA = (RA 0) + (RB) and pad left with zeroes, $(RT) \leftarrow {}^{16}0 \parallel MS(EA,2)$.		10-90
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).		10-91
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} .		10-92
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.		10-94

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.		10-96
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).		10-98
lwz	RT, D(RA)	Load word from EA = (RA 0) + EXTS(D) and place in RT, (RT) \leftarrow MS(EA,4).		10-99
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.		10-100
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.		10-101
lwzx	RT, RA, RB	Load word from EA = (RA 0) + (RB) and place in RT, (RT) \leftarrow MS(EA,4).		10-102
stb	RS, D(RA)	Store byte (RS) _{24:31} in memory at EA = (RA 0) + EXTS(D).		10-141
stbu	RS, D(RA)	Store byte (RS) _{24:31} in memory at EA = (RA 0) + EXTS(D). Update the base address, (RA) \leftarrow EA.		10-142
stbux	RS, RA, RB	Store byte (RS) _{24:31} in memory at EA = (RA 0) + (RB). Update the base address, (RA) \leftarrow EA.		10-143
stbx	RS, RA, RB	Store byte (RS) _{24:31} in memory at EA = (RA 0) + (RB).		10-144
sth	RS, D(RA)	Store halfword (RS) _{16:31} in memory at EA = (RA 0) + EXTS(D).		10-145

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}		10-146
sthu	RS, D(RA)	Store halfword (RS) _{16:31} in memory at EA = (RA 0) + EXTS(D). Update the base address, (RA) \leftarrow EA.		10-147
sthux	RS, RA, RB	Store halfword (RS) _{16:31} in memory at EA = (RA 0) + (RB). Update the base address, (RA) \leftarrow EA.		10-148
sthx	RS, RA, RB	Store halfword (RS) _{16:31} in memory at EA = (RA 0) + (RB).		10-149
stmw	RS, D(RA)	Store consecutive words from RS through GPR(31) in memory starting at EA = (RA 0) + EXTS(D).		10-150
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).		10-151
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).		10-152
stw	RS, D(RA)	Store word (RS) in memory at EA = (RA 0) + EXTS(D).		10-154
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}		10-155

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 $(CR[CR0]) \leftarrow {}^20 \parallel 0 \parallel XER_{so}.$		10-156
stwu	RS, D(RA)	Store word (RS) in memory at $EA = (RA 0) + EXTS(D).$ Update the base address, $(RA) \leftarrow EA.$		10-158
stwux	RS, RA, RB	Store word (RS) in memory at $EA = (RA 0) + (RB).$ Update the base address, $(RA) \leftarrow EA.$		10-159
stwx	RS, RA, RB	Store word (RS) in memory at $EA = (RA 0) + (RB).$		10-160

B.6 Arithmetic and Logical Instructions

Table B-6 shows the set of arithmetic and logical instructions supported by the PPC401GF. 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.		10-6
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].		10-7
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].		10-8
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.		10-9
addic	RT, RA, IM	Add EXTS(IM) to (RA 0). Place result in RT. Place carry-out in XER[CA].		10-10
addic.	RT, RA, IM	Add EXTS(IM) to (RA 0). Place result in RT. Place carry-out in XER[CA].	CR[CR0]	10-11
addis	RT, RA, IM	Add (IM ¹⁶ 0) to (RA 0). Place result in RT.		10-12
addme	RT, RA	Add XER[CA], (RA), (-1). Place result in RT. Place carry-out in XER[CA].		10-13
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].		10-14
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.		10-15
and.			CR[CR0]	
andc	RA, RS, RB	AND (RS) with \neg (RB). Place result in RA.		10-16
andc.			CR[CR0]	
andi.	RA, RS, IM	AND (RS) with ($^{16}0 \parallel$ IM). Place result in RA.	CR[CR0]	10-17
andis.	RA, RS, IM	AND (RS) with (IM \parallel $^{16}0$). Place result in RA.	CR[CR0]	10-18
cntlzw	RA, RS	Count leading zeros in RS. Place result in RA.		10-41
cntlzw.			CR[CR0]	
divw	RT, RA, RB	Divide (RA) by (RB), signed. Place result in RT.		10-63
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.		10-64
divwu.			CR[CR0]	
divwuo			XER[SO, OV]	
divwuo.			CR[CR0] XER[SO, OV]	
eqv	RA, RS, RB	Equivalence of (RS) with (RB). (RA) $\leftarrow \neg((RS) \oplus (RB))$		10-66
eqv.			CR[CR0]	
extsb	RA, RS	Extend the sign of byte (RS) _{24:31} . Place the result in RA.		10-67
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.		10-68
extsh.			CR[CR0]	
mulhw	RT, RA, RB	Multiply (RA) and (RB), signed. Place hi-order result in RT. $\text{prod}_{0:63} \leftarrow (\text{RA}) \times (\text{RB})$ (signed). (RT) $\leftarrow \text{prod}_{0:31}$.		10-118
mulhw.			CR[CR0]	
mulhwu	RT, RA, RB	Multiply (RA) and (RB), unsigned. Place hi-order result in RT. $\text{prod}_{0:63} \leftarrow (\text{RA}) \times (\text{RB})$ (unsigned). (RT) $\leftarrow \text{prod}_{0:31}$.		10-119
mulhwu.			CR[CR0]	
mulli	RT, RA, IM	Multiply (RA) and IM, signed. Place lo-order result in RT. $\text{prod}_{0:47} \leftarrow (\text{RA}) \times \text{IM}$ (signed) (RT) $\leftarrow \text{prod}_{16:47}$		10-120
mulw	RT, RA, RB	Multiply (RA) and (RB), signed. Place lo-order result in RT. $\text{prod}_{0:63} \leftarrow (\text{RA}) \times (\text{RB})$ (signed). (RT) $\leftarrow \text{prod}_{32:63}$.		10-121
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.		10-122
nand.			CR[CR0]	
neg	RT, RA	Negative (two's complement) of RA. (RT) $\leftarrow \neg(\text{RA}) + 1$		10-123
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.		10-124
nor.			CR[CR0]	
or	RA, RS, RB	OR (RS) with (RB). Place result in RA.		10-125
or.			CR[CR0]	
orc	RA, RS, RB	OR (RS) with $\neg(\text{RB})$. Place result in RA.		10-126
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.		10-127
oris	RA, RS, IM	OR (RS) with (IM ¹⁶ 0). Place result in RA.		10-128
subf	RT, RA, RB	Subtract (RA) from (RB). (RT) $\leftarrow \neg(RA) + (RB) + 1$.		10-161
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].		10-162
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].		10-164
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].		10-165
subfme	RT, RA, RB	Subtract (RA) from (−1) with carry-in. (RT) $\leftarrow \neg(RA) + (−1) + XER[CA]$. Place carry-out in XER[CA].		10-166
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].		10-167
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} .		10-42
crandc	BT, BA, BB	AND bit (CR _{BA}) with ¬(CR _{BB}). Place result in CR _{BT} .		10-43
creqv	BT, BA, BB	Equivalence of bit CR _{BA} with CR _{BB} . $CR_{BT} \leftarrow \neg(CR_{BA} \oplus CR_{BB})$		10-44
crnand	BT, BA, BB	NAND bit (CR _{BA}) with (CR _{BB}). Place result in CR _{BT} .		10-45
crnor	BT, BA, BB	NOR bit (CR _{BA}) with (CR _{BB}). Place result in CR _{BT} .		10-46
cror	BT, BA, BB	OR bit (CR _{BA}) with (CR _{BB}). Place result in CR _{BT} .		10-47
crorc	BT, BA, BB	OR bit (CR _{BA}) with ¬(CR _{BB}). Place result in CR _{BT} .		10-48
crxor	BT, BA, BB	XOR bit (CR _{BA}) with (CR _{BB}). Place result in CR _{BT} .		10-49
mcrf	BF, BFA	Move CR field, (CR[CR _n]) ← (CR[CR _m]) where m ← BFA and n ← BF.		10-103

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. $LI \leftarrow (target - CIA)_{6:29}$ $NIA \leftarrow CIA + EXTS(LI \parallel ^20)$		10-19
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 $BO_2 = 0.$	10-20
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 $BO_2 = 0.$	10-28
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, $NIA \leftarrow LR_{0:29} \parallel 2^0$.	CTR if $BO_2 = 0$.	10-32
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 PPC401GF.

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.		10-37
cmpi	BF, 0, RA, IM	Compare (RA) to EXTS(IM), signed. Results in CR[CRn], where n = BF.		10-38
cmpl	BF, 0, RA, RB	Compare (RA) to (RB), unsigned. Results in CR[CRn], where n = BF.		10-39
cmpli	BF, 0, RA, IM	Compare (RA) to ($1^60 \parallel IM$), unsigned. Results in CR[CRn], where n = BF.		10-40

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 PPC401GF 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. $r \leftarrow \text{ROTL}((RS), SH)$ $m \leftarrow \text{MASK}(MB, ME)$ $(RA) \leftarrow (r \wedge m) \vee ((RA) \wedge \neg m)$		10-131
rlwimi.			CR[CR0]	
rlwinm	RA, RS, SH, MB, ME	Rotate left word immediate, then AND with mask. $r \leftarrow \text{ROTL}((RS), SH)$ $m \leftarrow \text{MASK}(MB, ME)$ $(RA) \leftarrow (r \wedge m)$		10-132
rlwinm.			CR[CR0]	
rlwnm	RA, RS, RB, MB, ME	Rotate left word, then AND with mask. $r \leftarrow \text{ROTL}((RS), (RB)_{27:31})$ $m \leftarrow \text{MASK}(MB, ME)$ $(RA) \leftarrow (r \wedge m)$		10-135
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 $m \leftarrow {}^{32}0$. $(RA) \leftarrow r \wedge m$.		10-137
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 $m \leftarrow {}^{32}0$. $s \leftarrow (RS)_0$. $(RA) \leftarrow (r \wedge m) \vee ({}^{32}s \wedge \neg m)$. $\text{XER}[CA] \leftarrow s \wedge ((r \wedge \neg m) \neq 0)$.		10-138
sraw.			CR[CR0]	
srawi	RA, RS, SH	Shift right algebraic (RS) by SH. $n \leftarrow SH$. $r \leftarrow \text{ROTL}((RS), 32 - n)$. $m \leftarrow \text{MASK}(n, 31)$. $s \leftarrow (RS)_0$. $(RA) \leftarrow (r \wedge m) \vee ({}^{32}s \wedge \neg m)$. $\text{XER}[CA] \leftarrow s \wedge ((r \wedge \neg m) \neq 0)$.		10-139
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 $m \leftarrow {}^{32}0$. $(RA) \leftarrow r \wedge m$.		10-140
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 $(RA 0) + (RB)$.		10-50
dcbf	RA, RB	Flush (store, then invalidate) the data cache block which contains the effective address $(RA 0) + (RB)$.		10-50
dcbi	RA, RB	Invalidate the data cache block which contains the effective address $(RA 0) + (RB)$.		10-51
dcbst	RA, RB	Store the data cache block which contains the effective address $(RA 0) + (RB)$.		10-52
dcbt	RA, RB	Load the data cache block which contains the effective address $(RA 0) + (RB)$.		10-53
dcbtst	RA, RB	Load the data cache block which contains the effective address $(RA 0) + (RB)$.		10-55
dcbz	RA, RB	Zero the data cache block which contains the effective address $(RA 0) + (RB)$.		10-57
dccci	RA, RB	Invalidate the data cache congruence class associated with the effective address $(RA 0) + (RB)$.		10-59

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.		10-61
icbi	RA, RB	Invalidate the instruction cache block which contains the effective address (RA 0) + (RB).		10-69
icbt	RA, RB	Load the instruction cache block which contains the effective address (RA 0) + (RB).		10-71
iccci	RA, RB	Invalidate instruction cache congruence class associated with the effective address (RA 0) + (RB).		10-73
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.		10-74

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).		10-108
mtmsr	RS	Move to MSR from RS, (MSR) \leftarrow (RS).		10-115
rftci		Return from critical interrupt (PC) \leftarrow (SRR2). (MSR) \leftarrow (SRR3).		10-129

Table B-12. Interrupt Control Instructions (cont.)

Mnemonic	Operands	Function	Other Registers Changed	Page
rfi		Return from interrupt. (PC) \leftarrow (SRR0). (MSR) \leftarrow (SRR1).		10-130
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 PPC401GF; 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.		10-65
isync		Synchronize execution context by flushing the prefetch queue.		10-76
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.		10-104
mfcrr	RT	Move from CR to RT, (RT) \leftarrow (CR).		10-105
mfdcr	RT, DCRN	Move from DCR to RT, (RT) \leftarrow (DCR(DCRN)).		10-106
mfspr	RT, SPRN	Move from SPR to RT, (RT) \leftarrow (SPR(SPRN)).		10-109

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, $\text{mask} \leftarrow {}^4(\text{FXM}_0) \parallel {}^4(\text{FXM}_1) \parallel \dots \parallel {}^4(\text{FXM}_6) \parallel {}^4(\text{FXM}_7)$. $(\text{CR}) \leftarrow ((\text{RS}) \wedge \text{mask}) \vee (\text{CR}) \wedge \neg \text{mask}$.		10-111
mtdcr	DCRN, RS	Move to DCR from RS, $(\text{DCR}(\text{DCRN})) \leftarrow (\text{RS})$.		10-113
mtspr	SPRN, RS	Move to SPR from RS, $(\text{SPR}(\text{SPRN})) \leftarrow (\text{RS})$.		10-116
sc		System call exception is generated. $(\text{SRR1}) \leftarrow (\text{MSR})$ $(\text{SRR0}) \leftarrow (\text{PC})$ $\text{PC} \leftarrow \text{EVPR}_{0:15} \parallel \text{x'0C00'}$ $(\text{MSR}[\text{WE}, \text{PR}, \text{EE}, \text{PE}, \text{DR}, \text{IR}]) \leftarrow 0$ $(\text{MSR}[\text{LE}]) \leftarrow (\text{MSR}[\text{ILE}])$		10-136
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.		10-168
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 PPC401GF, 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. 2-35, 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:

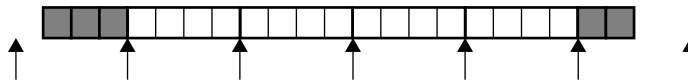
- A BKNT executes in one clock cycle; by definition, a BKNT does not have address or condition dependencies.
- Branches known taken (BKT) by definition have no condition dependencies, but may have address dependencies. A BKT without address dependencies executes in two clock cycles; a BKT having address 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 PPC401GF 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. 2-32, 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

Numerics

401 Core 1-1, 1-3

A

add 10-6
add. 10-6
addc 10-7
addc. 10-7
addco 10-7
addco. 10-7
adde 10-8
adde. 10-8
addeo 10-8
addeo. 10-8
addi 10-9
addic 10-10
addic. 10-11
addis 10-12
addme 10-13
addme. 10-13
addmeo 10-13
addmeo. 10-13
addo 10-6
addo. 10-6
Address state 3-3
addressing 2-1
addressing modes 1-8
addze 10-14
addze. 10-14
addzeo 10-14
addzeo. 10-14
alignment 2-17
alignment and little endian operation 2-18
alignment error 5-22
alignment exceptions
 summary 2-18
and 10-15
and. 10-15
andc 10-16
andc. 10-16
andi. 10-17
andis. 10-18
architecture, PowerPC 1-2
arithmetic compare 2-12

B

b 10-19
ba 10-19
Bank Register Initialization 4-7
bc 10-20
bca 10-20
bcctr 10-28
bcctrl 10-28
bcl 10-20
bcla 10-20
bclr 10-32
bclrl 10-32
bctr 10-29
bctrl 10-29
bdnz 10-21
bdnza 10-21
bdnzf 10-21
bdnzfa 10-21
bdnzfl 10-21
bdnzfla 10-21
bdnzflr 10-33
bdnzflrl 10-33
bdnzl 10-21
bdnzla 10-21
bdnzlr 10-33
bdnzlrl 10-33
bdnzt 10-21
bdnzta 10-21
bdnztl 10-21
bdnztla 10-21
bdnztlr 10-33
bdnztlrl 10-33
bdz 10-22
bdza 10-22
bdzf 10-22
bdzfa 10-22
bdzfl 10-22
bdzfla 10-22
bdzflr 10-34
bdzflrl 10-34
bdzl 10-22
bdzla 10-22
bdzlr 10-33
bdzlrl 10-33
bdzt 10-22
bdzta 10-22
bdztl 10-22

bdztla 10-22
 bdztlr 10-34
 bdztlrl 10-34
 BEAR 5-16, 11-6
 beq 10-23
 beqa 10-23
 beqctr 10-29
 beqctrl 10-29
 beql 10-23
 beqlr 10-34
 beqlrl 10-34
 BCSR 5-15, 11-7
 bf 10-23
 bfa 10-23
 bfctr 10-29
 bfctrl 10-29
 bfl 10-23
 bfla 10-23
 bflr 10-34
 bflrl 10-34
 B-form A-50
 bge 10-23
 bgea 10-23
 bgectrl 10-29
 bgel 10-23
 bgela 10-23
 bgelr 10-34
 bgelrl 10-34
 bgrctr 10-29
 bgt 10-24
 bgta 10-24
 bgtctr 10-29
 bgctrl 10-29
 bgtl 10-24
 bgtla 10-24
 bgtlr 10-34
 bgtrl 10-34
 bl 10-19
 bla 10-19
 ble 10-24
 blea 10-24
 blectr 10-30
 blectrl 10-30
 blel 10-24
 blela 10-24
 blelr 10-35
 blelrl 10-35

blr 10-33
 blrl 10-33
 blt 10-24
 blta 10-24
 bltctr 10-30
 bltctrl 10-30
 bltl 10-24
 bltla 10-24
 bltlr 10-35
 bltrl 10-35
 bne 10-25
 bnea 10-25
 bnctr 10-30
 bnctrl 10-30
 bnel 10-25
 bnela 10-25
 bnelr 10-35
 bnelrl 10-35
 bng 10-25
 bnga 10-25
 bngctr 10-30
 bngctrl 10-30
 bngl 10-25
 bngla 10-25
 bnglr 10-35
 bnglrl 10-35
 bnl 10-25
 bnla 10-25
 bnlctr 10-30
 bnctrl 10-30
 bnll 10-25
 bnlla 10-25
 bnllr 10-35
 bnllrl 10-35
 bns 10-26
 bnsa 10-26
 bnsctr 10-30
 bnsctrl 10-30
 bnsl 10-26
 bnsla 10-26
 bnslr 10-35
 bnsrl 10-35
 bnu 10-26
 bnua 10-26
 bnuctr 10-31
 bnuctrl 10-31
 bnul 10-26

- bnula 10-26
- bnulr 10-36
- bnulrl 10-36
- branch prediction 2-35, A-1, B-4
- branching
 - control 2-32
 - speculative accesses 2-36
- branching control
 - AA field on conditional branches 2-33
 - AA field on unconditional branches 2-33
 - BI field on conditional branches 2-33
 - BO field on conditional branches 2-33
 - branch prediction 2-35
- BRCR0–BRCR7 11-8
- bso 10-26
- bsoa 10-26
- bsoctr 10-31
- bsoctrl 10-31
- bsol 10-26
- bsola 10-26
- bsolr 10-36
- bsolrl 10-36
- bt 10-27
- bta 10-27
- btctr 10-31
- btctrl 10-31
- btl 10-27
- btla 10-27
- btlr 10-36
- bttrl 10-36
- bun 10-27
- buna 10-27
- bunctr 10-31
- bunctrl 10-31
- bunl 10-27
- bunla 10-27
- bunlr 10-36
- bunlrl 10-36
- bus states 3-3
 - Address state 3-3
 - Hold state 3-5
 - Idle state 3-3
 - Recovery state 3-4
 - Wait/Data state 3-4
- bus terminology
 - bus access 3-3
 - bus request 3-3

- data transfer 3-3
- byte ordering
 - storage attribute 8-12

C

- cache
 - data
 - cacheability control 6-6, 8-6
 - 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, 8-8
 - 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
 - data 8-6
- cacheability
 - instruction 8-8
- CDBCR 6-9, 11-9
- clrlslwi 10-132
- clrlslwi. 10-132
- clrlwi 10-132
- clrlwi. 10-132
- clrrwi 10-133
- clrrwi. 10-133
- cmp 10-37
- cmpi 10-38
- cmpl 10-39
- cmpli 10-40
- cmplw 10-39
- cmplwi 10-40
- cmpw 10-37
- cmpwi 10-38
- cntlzw 10-41
- cntlzw. 10-41
- compare

- arithmetic 2-12
- logical 2-12
- condition register 2-11
- condition register (CR) 1-8
- context synchronization 2-42
- CR 11-10
- crand 10-42
- crandc 10-43
- crclr 10-49
- creqv 10-44
- critical exceptions, defined 5-5
- critical interrupt pin 5-13
- crmove 10-47
- crnand 10-45
- crnor 10-46
- crnot 10-46
- cror 10-47
- crorc 10-48
- crset 10-44
- crxor 10-49
- CTR 2-6, 11-11

D

- DAC1 7-8, 11-12
- Data Address Compare Register (DAC1) 7-8
- data alignment 2-17
- 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 cache unit (DCU) 1-6
- data storage exception 5-19
- data types 1-7, 2-17
- DBCR 7-5, 11-13
- DBSR 7-6
- dcba 10-50
- dcbf 6-8, 6-13, 10-50
- dcbi 6-8, 10-51
- dcbst 6-8, 10-52
- dcbt 6-8, 10-53
- dcbtst 6-8, 10-55
- dcbz 6-9, 6-13, 10-57

- dccci 6-9, 10-59
- DCCR 8-6, 11-17
- dcread 6-9, 10-61
- 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 11-19
- DEAR 5-13, 11-21
- Debug Control Register (DBCR) 7-5
- debug exceptions 5-27
 - branch taken 5-27
 - DAC 5-27
 - IAC 5-27
 - instruction completion 5-27
 - non-critical exceptions 5-27
 - TRAP 5-27
 - unconditional 5-27
- 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 7-4
- device control registers 2-16
- device control registers (DCRs) 1-8
- D-form A-50
- divw 10-63
- divw. 10-63
- divwo 10-63
- divwo. 10-63
- divwu 10-64
- divwu. 10-64
- divwuo 10-64
- divwuo. 10-64
- DRAM
 - behavior during reset 4-5

E

- eieio 10-65
- eqv 10-66
- eqv. 10-66
- ESR 11-22
- EVPR 5-10, 11-23
- exception-handling registers, general 5-7
- exceptions
 - exceptions
 - alignment exception summary 2-18
 - FIT 5-26
 - handling
 - MSR bits 2-40
 - PIT 5-25
 - registers during alignment error 5-23
 - registers during critical interrupt 5-14
 - registers during debug exceptions 5-28
 - registers during external interrupts 5-22
 - registers during FIT interrupt 5-26
 - registers during machine check 5-17, 5-18
 - registers during PIT interrupt 5-25
 - registers during program exceptions 5-24
 - registers during system call 5-24
 - registers during watchdog interrupt 5-27
 - SRR0-SRR1 (non-critical) 5-8
 - SRR2-SRR3 (critical) 5-9
- execution synchronization 2-45
- execution unit 1-4
- extended memonics
 - beqlr 10-34
- extended menmonics
 - blectrl 10-30
 - bnlctrl 10-30
- extended mnemonicid
 - bn gla 10-25
- extended mnemonics 2-52
 - alphabetical B-4
 - bctr 10-29
 - bctrl 10-29
 - bdnz 10-21
 - bdnza 10-21
 - bdnzf 10-21
 - bdnzfa 10-21
 - bdnzfkr 10-33
 - bdnzfl 10-21
 - bdnzfla 10-21
 - bdnzflrl 10-33
 - bdnzl 10-21
 - bdnzla 10-21
 - bdnzlr 10-33
 - bdnzlrl 10-33
 - bdnzt 10-21
 - bdnzta 10-21
 - bdnztl 10-21
 - bdnztla 10-21
 - bdnztlr 10-33
 - bdnztlrl 10-33
 - bdz 10-22
 - bdza 10-22
 - bdzf 10-22
 - bdzfa 10-22
 - bdzfl 10-22
 - bdzfla 10-22
 - bdzflr 10-34
 - bdzflrl 10-34
 - bdzl 10-22
 - bdzla 10-22
 - bdzlr 10-33
 - bdzlr 10-33
 - bdzt 10-22
 - bdzta 10-22
 - bdztl 10-22
 - bdztl 10-22
 - bdztlr 10-34
 - bdztlrl 10-34
 - beq 10-23
 - beqa 10-23
 - beqctr 10-29
 - beqctrl 10-29
 - beql 10-23
 - beqlrl 10-34
 - bf 10-23
 - bfa 10-23
 - bfctr 10-29
 - bfctrl 10-29
 - bfl 10-23
 - bfla 10-23
 - bflr 10-34
 - bflrl 10-34
 - bge 10-23
 - bgea 10-23
 - bgctr 10-29
 - bgctrl 10-29

bgel	10-23	bnlla	10-25
bgela	10-23	bnllr	10-35
bgelr	10-34	bnllrl	10-35
bgelrl	10-34	bns	10-26
bgt	10-24	bnsa	10-26
bgta	10-24	bnsctr	10-30
bgtctr	10-29	bnsctrl	10-30
bgtctrl	10-29	bnsi	10-26
bgtl	10-24	bnsia	10-26
bgtla	10-24	bnsir	10-35
bgtlr	10-34	bnsirl	10-35
bgtlrl	10-34	bnu	10-26
ble	10-24	bnuu	10-26
blea	10-24	bnuctr	10-31
blectr	10-30	bnuctrl	10-31
blel	10-24	bnul	10-26
blela	10-24	bnula	10-26
blelr	10-35	bnulr	10-36
blelrl	10-35	bnulrl	10-36
blr	10-33	bsalr	10-36
blrl	10-33	bso	10-26
blt	10-24	bsoa	10-26
blta	10-24	bsoctr	10-31
bltctr	10-30	bsoctrl	10-31
bltctrl	10-30	bsol	10-26
bltl	10-24	bsola	10-26
bltla	10-24	bsolrl	10-36
bltlr	10-35	bt	10-27
bltlrl	10-35	bta	10-27
bne	10-25	btctr	10-31
bnea	10-25	btctrl	10-31
bnectrl	10-30	btl	10-27
bnel	10-25	btla	10-27
bnela	10-25	btlr	10-36
bnelr	10-35	btlrl	10-36
bnelrl	10-35	bun	10-27
bng	10-25	buna	10-27
bnga	10-25	bunctr	10-31
bngctr	10-30	bunctrl	10-31
bngctrl	10-30	bunl	10-27
bngl	10-25	bunla	10-27
bnglr	10-35	bunlr	10-36
bnglrl	10-35	bunlrl	10-36
bnl	10-25	clrlslwi	10-132
bnla	10-25	clrlslwi.	10-132
bnlctr	10-30	clrlwi	10-132
bnll	10-25	clrlwi.	10-132

clrrwi	10-133	la	10-9
clrrwi.	10-133	li	10-9
cmplw	10-39	lis	10-12
cmplwi	10-40	mfcdbcn	10-110
cmpw	10-37	mfctr	10-110
cmpwi	10-38	mfdac	10-110
crclr	10-49	mfdbsl	10-110
crmove	10-47	mfdbsr	10-110
crnot	10-46	mfdbsrs	10-110
crset	10-44	mfdccr	10-110
extlwi	10-133	mfdcwr	10-110
extlwi.	10-133	mfdear	10-110
extrwi	10-133	mfesr	10-110
extrwi.	10-133	mfevpr	10-110
for addi	10-9	mflac1	10-110
for addic	10-10	mflccr	10-110
for addic.	10-11	mflcdbdr	10-110
for addis	10-12	mflr	10-110
for bc, bca, bcl, bcla	10-21	mfplt	10-110
for bcctr, bcctrl	10-29	mfpvr	10-110
for bclr, bclrl	10-33	mfsgr	10-110
for cmp	10-37	mfsler	10-110
for cmpi	10-38	mfsprg0	10-110
for cmpl	10-39	mfsprg1	10-110
for cmpli	10-40	mfsprg2	10-110
for creqv	10-44	mfsprg3	10-110
for crnor	10-46	mftcr	10-110
for cror	10-47	mftsr	10-110
for crxor	10-49	mfxer	10-110
for mfdc	10-107, 10-114	mptvr	10-117
for mfspr	10-110	mr	10-125
for mtrcf	10-112	mr.	10-125
for mtspr	10-117	mtcdbc	10-117
for nor, nor.	10-124	mtcr	10-112
for or, or.	10-125	mtctr	10-117
for ori	10-127	mtdac1	10-117
for rlwimi, rlwimi.	10-131	mtdbc	10-117
for rlwinm, rlwinm.	10-132	mtdbsr	10-117
for rlwnm, rlwnm.	10-135	mtdccr	10-117
for subf, subf., subfo, subfo.	10-161	mtdcwr	10-117
for subfc, subfc., subfco, subfco.	10-163	mtesr	10-117
for tw	10-170	mtevpr	10-117
for twi	10-174	mtiac1	10-117
inslwi	10-131	mticcr	10-117
inslwi.	10-131	mticdbdr	10-117
insrwi	10-131	mtlr	10-117
insrwi.	10-131	mtpit	10-117

mtsear	10-117	twle	10-171
mtsgsr	10-117	twlei	10-175
mtsler	10-117	twlgei	10-175
mtsprg1	10-117	twlgt	10-171
mtsprg2	10-117	twlgti	10-175
mtsprg3	10-117	twlle	10-171
mtsrr0	10-117	twllei	10-175
mtsrr1	10-117	twllt	10-171
mtsrr2	10-117	twllti	10-175
mtsrr3	10-117	twlng	10-171
mttcr	10-117	twlngi	10-175
mttsprg0	10-117	twlnl	10-171
mttsr	10-117	twlnli	10-175
mttxer	10-117	twlt	10-171
nop	10-127	twlti	10-175
not	10-124	twne	10-172
not.	10-124	twnei	10-175
rotlw	10-135	twng	10-172
rotlw.	10-135	twngi	10-176
rotlwi	10-133	twnl	10-172
rotlwi.	10-133	twnli	10-176
rotrwi	10-133		
rotrwi.	10-133	external bus master	3-29
slwi	10-134	arbitration	3-29
slwi.	10-134	interface	3-29
srwi	10-134	synchronous interface	3-30
srwi.	10-134	external bus states	3-3
sub	10-161	Address state	3-3
sub.	10-161	Hold state	3-5
subc	10-163	Idle state	3-3
subc.	10-163	Recovery state	3-4
subco	10-163	Wait/Data state	3-4
subco.	10-163	external interfaces	
subi	10-9	memory and peripherals	1-6
subic	10-10	external interrupts	5-20
subic.	10-11	DMA	5-20
subis	10-12	external interrupt pins	5-20
subo	10-161	JTAG port	5-20
subo.	10-161	serial port	5-20
trap	10-170	extlwi	10-133
tweq	10-171	extlwi.	10-133
tweqi	10-174	extrwi	10-133
twge	10-171	extrwi.	10-133
twgei	10-174	extsb	10-67
twgle	10-171	extsb.	10-67
twgt	10-171	extsh	10-68
twgti	10-175	extsh.	10-68

F

features, 401 Core 1-3
features, PPC401GF 1-3
FIT 5-26, 5-34
fixed interval timer 5-26, 5-34

G

general exception-handling registers 5-7
general purpose registers (GPRs) 1-8
GPR0-GPR31 2-4, 11-24
guarded storage
 architectural overview 2-37
 SGR 8-10
 speculative accesses to 2-37

H

Hold state 3-5

I

IAC1 7-10, 11-25
icbi 6-7, 6-14, 10-69
icbt 6-7, 10-71
iccci 6-8, 10-73
ICCR 8-8, 11-26
ICDBDR 6-10, 11-28
icread 6-8, 10-74
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
Idle state 3-3
I-form A-50
initialization 4-6
 code example 4-7
 requirements 4-6
inslwi 10-131
inslwi. 10-131
insrwi 10-131
insrwi. 10-131
instruction
 add 10-6
 add. 10-6
 addc 10-7

addc. 10-7
addco 10-7
addco. 10-7
adde 10-8
adde. 10-8
addeo 10-8
addeo. 10-8
addi 10-9
addic 10-10
addic. 10-11
addis 10-12
addme 10-13
addme. 10-13
addmeo 10-13
addmeo. 10-13
addo 10-6
addo. 10-6
addze 10-14
addze. 10-14
addzeo 10-14
addzeo. 10-14
and 10-15
and. 10-15
andc 10-16
andc. 10-16
andi. 10-17
andis. 10-18
b 10-19
ba 10-19
bc 10-20
bca 10-20
bcctr 10-28
bcctrl 10-28
bcl 10-20
bcla 10-20
bclr 10-32
bclrl 10-32
bl 10-19
bla 10-19
cmp 10-37
cmpi 10-38
cmpl 10-39
cmpli 10-40
cntlzw 10-41
cntlzw. 10-41
crand 10-42
crandc 10-43

creqv	10-44	lbzx	10-81
crnand	10-45	lha	10-82
crnor	10-46	lhau	10-83
cror	10-47	lhaux	10-84
crorc	10-48	lhax	10-85
crxor	10-49	lhbrx	10-86
dcba	10-50	lhz	10-87
dcbf	10-50	lhzu	10-88
dcbi	10-51	lhzux	10-89
dcbst	10-52	lhzx	10-90
dcbt	10-53	lmw	10-91
dcbtst	10-55	lswi	10-92
dcbz	10-57	lswx	10-94
dccci	10-59	lwarx	10-96
dcread	10-61	lwbrx	10-98
divw	10-63	lwz	10-99
divw.	10-63	lwzu	10-100
divwo	10-63	lwzux	10-101
divwo.	10-63	lwzx	10-102
divwu	10-64	mcrf	10-103
divwu.	10-64	mcrxr	10-104
divwuo	10-64	mfcrr	10-105
divwuo.	10-64	mfdcr	10-106
eieio	10-65	mfmsr	10-108
eqv	10-66	mfspr	10-109
eqv.	10-66	mftb	5-31, 10-111
extsb	10-67	mtcrf	10-111
extsb.	10-67	mtdcr	10-113
extsh	10-68	mtspr	10-116
extsh.	10-68	mulhw	10-118
formats		mulhw.	10-118
B-form	A-50	mulhwu	10-119
D-form	A-50	mulhwu.	10-119
I-form	A-50	mulli	10-120
SC-form	A-50	mullw	10-121
X-form	A-51	mullw.	10-121
XFX-form	A-52	mullwo	10-121
XL-form	A-52	mullwo.	10-121
XO-form	A-52	nand	10-122
icbi	10-69	nand.	10-122
icbt	10-71	neg	10-123
iccci	10-73	neg.	10-123
icread	10-74	nego	10-123
isync	10-76	nego.	10-123
lbz	10-78	nor	10-124
lbzu	10-79	nor.	10-124
lbzux	10-80	or	10-125

- or. 10-125
- orc 10-126
- orc. 10-126
- ori 10-127
- oris 10-128
- rfci 10-129
- rfi 10-130
- rlwimi 10-131
- rlwimi. 10-131
- rlwinm 10-132
- rlwinm. 10-132
- rlwnm 10-135
- rlwnm. 10-135
- sc 10-136
- slw 10-137
- slw. 10-137
- sraw 10-138
- sraw. 10-138
- srawi 10-139
- srawi. 10-139
- srw 10-140
- srw. 10-140
- stb 10-141
- stbu 10-142
- stbux 10-143
- stbx 10-144
- sth 10-145
- sthbrx 10-146
- sthu 10-147
- sthux 10-148
- sthx 10-149
- stmw 10-150
- stswi 10-151
- stswx 10-152
- stw 10-154
- stwbrx 10-155
- stwcx. 10-156
- stwu 10-158
- stwux 10-159
- stwx 10-160
- subf 10-161
- subf. 10-161
- subfc 10-162
- subfc. 10-162
- subfco 10-162
- subfco. 10-162
- subfe 10-164
- subfe. 10-164
- subfeo 10-164
- subfeo. 10-164
- subfic 10-165
- subfme 10-166
- subfme. 10-166
- subfmeo 10-166
- subfmeo. 10-166
- subfo 10-161
- subfo. 10-161
- subfze 10-167
- subfze. 10-167
- subfzeo 10-167
- subfzeo. 10-167
- sync 10-168
- tw 10-169
- twi 10-173
- wrtree 10-177
- wrtreei 10-178
- xor 10-179
- xori 10-180
- 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 cache unit 1-6
- instruction cache unit (ICU) 1-6
- instruction fields A-47
- instruction formats 10-2, A-47
 - B-form A-50
 - D-form A-50
 - diagrams A-49
 - I-Form A-50
 - M-form A-52
 - SC-form A-50
 - X-form A-51
 - XFX-form A-52
 - XL-form A-52
 - XO-form A-52
- instruction forms A-47, A-49
- instruction queue 2-32
- instruction set
 - brief summaries by category 2-46

- instruction set portability 10-1
- instruction set summary
 - arithmetic and logical 2-48
 - branch 2-49
 - cache control 2-51
 - compare 2-49
 - CR logical 2-50
 - interrupt control 2-51
 - processor management 2-52
 - rotate and shift 2-50
- 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 2-18
 - 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 2-17
 - categories B-1
 - comparison B-37
 - condition register logical B-35
 - extended mnemonics B-4
 - format diagrams A-49
 - formats A-47
 - forms A-47, A-49
 - interrupt control B-40
 - opcodes A-39
 - privileged 2-41, B-3
 - processor management B-41
 - rotate and shift B-38
 - specific to PowerPC Embedded Controllers B-1
 - storage reference B-26
 - alignment 2-17
- interfaces
 - interrupt controller 1-7
 - memory and peripherals 1-6
 - interrupt controller interface 1-7
- interrupts
 - internal 5-20
- interrupts and exceptions 5-2, 5-3
 - alignment error 5-22
 - architectural definitions and behavior 5-2
 - asynchronous, defined 5-2
 - critical 5-5, 5-6
 - critical interrupt pin 5-13
 - data machine check 5-18
 - data storage 5-19
 - debug 5-27
 - exception, defined 5-2
 - external interrupts 5-20
 - fixed interval timer 5-26
 - imprecise, defined 5-2
 - instruction machine check 5-3
 - interrupt, defined 5-2
 - machine check, defined 5-3
 - machine check—instruction 5-17
 - non-critical 5-5
 - precise, defined 5-2
 - program 5-23
 - programmable interval timer 5-25
 - synchronous, defined 5-2
 - system call 5-24
 - watchdog timer 5-26
- IOCR 5-20, 11-29
- isync 10-76

L

- la 10-9
- lbz 10-78
- lbzu 10-79
- lbzux 10-80
- lbzx 10-81
- lha 10-82
- lhau 10-83
- lhaux 10-84
- lhax 10-85
- lhbrx 10-86
- lhz 10-87
- lhzu 10-88
- lhzux 10-89
- lhzx 10-90
- li 10-9
- line locking, cache 6-12
- lis 10-12

- little endian operation and alignment 2-18
- lmw 10-91
- logical compare 2-12
- LR 2-7, 11-30
- lswi 10-92
- lswx 10-94
- lwarx 10-96
- lwbrx 10-98
- lwz 10-99
- lwzu 10-100
- lwzux 10-101
- lwzx 10-102

M

- machine state register (MSR) 1-8
- mcrf 10-103
- mcrxr 10-104
- memory and peripheral interfaces 1-6
- memory interface
 - bus width after reset 3-14
 - external bus master 3-29
 - SRAM
 - WBE signal usage 3-9
- memory map 2-1, 2-3, 8-1
 - addressing 2-3, 8-1
 - storage attributes 2-3, 8-3
- memory organization 2-1
- mfcdbcn 10-110
- mfcr 10-105
- mfctr 10-110
- mfdac 10-110
- mfdbcl 10-110
- mfdbsr 10-110
- mfdbsrs 10-110
- mfdccr 10-110
- mfdcr 10-106
- mfdcwr 10-110
- mfdear 10-110
- mfesr 10-110
- mfevpr 10-110
- mflac1 10-110
- mflccr 10-110
- mflcldbdr 10-110
- mflr 10-110
- mfmsr 10-108
- M-form A-52
- mflpt 10-110

- mfpvr 10-110
- mfsgsr 10-110
- mfslcr 10-110
- mfsprr 10-109
- mfsprrg0 10-110
- mfsprrg1 10-110
- mfsprrg2 10-110
- mfsprrg3 10-110
- mftb 5-31, 10-111
- mftcr 10-110
- mftsr 10-110
- mfxer 10-110
- mptvr 10-117
- mr 10-125
- mr. 10-125
- MSR 2-15, 5-7, 11-31
- MSR bits and exception handling 2-40
- mtcdbcrr 10-117
- mtcr 10-112
- mtcrf 10-111
- mtctr 10-117
- mtdac1 10-117
- mtdbcr 10-117
- mtdbcsr 10-117
- mtdccr 10-117
- mtdcr 10-113
- mtdcwr 10-117
- mtdear 10-117
- mtesr 10-117
- mtcvpr 10-117
- mtiac1 10-117
- mticcr 10-117
- mticdbdr 10-117
- mtlr 10-117
- mtlpit 10-117
- mtsgr 10-117
- mtsler 10-117
- mtspr 10-116
- mtsprg1 10-117
- mtsprg2 10-117
- mtsprg3 10-117
- mtsrr0 10-117
- mtsrr1 10-117
- mtsrr2 10-117
- mtsrr3 10-117
- mttcr 10-117
- mttsprg0 10-117

- mttsr 10-117
- mttxer 10-117
- mulhw 10-118
- mulhw. 10-118
- mulhwu 10-119
- mulhwu. 10-119
- mulli 10-120
- mullw 10-121
- mullw. 10-121
- mullwo 10-121
- mullwo. 10-121

N

- nand 10-122
- nand. 10-122
- neg 10-123
- neg. 10-123
- nego 10-123
- nego. 10-123
- non-critical exceptions, defined 5-5
- nop 10-127
- nor 10-124
- nor. 10-124
- not 10-124
- not. 10-124
- notation xxiv, 10-2, A-47

O

- opcodes A-39
- 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 10-125
- or. 10-125
- orc 10-126
- orc. 10-126
- ori 10-127
- oris 10-128
- overview, 401 Core 1-1

P

- physical address map 2-3, 8-1
- PIT 5-25, 5-32, 11-33
- PMCR0 11-34
- portability, instruction set 10-1
- PowerPC architecture 1-2
- PowerPC Endian mode and alignment 2-18
- PPC401GF 1-3
- pre-fetch
 - branches to Count Register 2-37
 - branches to Link Register 2-37
 - queue 2-32
- primary opcodes A-39
- privileged DCRs 2-42
- privileged instructions 2-41
- privileged mode 2-40
- privileged operation 2-40
- privileged SPRs 2-41
- problem state 2-40
- program exception 5-23
- programmable interval timer 5-25, 5-32
- pseudocode 10-2
- PVR 2-10, 11-35

Q

- queue 2-32

R

- R0-R31 2-4, 11-24
- Recovery state 3-4
- register set summary 1-8
- registers
 - BEAR 5-16, 11-6
 - BESR 5-15, 11-7
 - BRCR0–BRCR7 11-8
 - CDBCR 6-9, 11-9
 - CR 1-8, 11-2, 11-10
 - CTR 2-6, 11-11
 - DAC1 7-8, 11-12
 - DBCR 7-5, 11-13
 - DBSR 7-6
 - DCCR 11-17
 - DCR numbering 11-4
 - DCWR 11-19
 - DEAR 5-13, 11-21
 - device control 1-8
 - during alignment error 5-23

- during critical interrupt 5-14
- during debug exceptions 5-28
- during external interrupts 5-22
- during FIT interrupt 5-26
- during machine check 5-17, 5-18
- during PIT interrupt 5-25
- during program exceptions 5-24
- during system call 5-24
- during watchdog interrupt 5-27
- ESR 11-22
- EVPR 5-10, 11-23
- general purpose 1-8
- GPR 11-2
- GPR0-GPR31 2-4, 11-24
- IAC1 7-10, 11-25
- ICCR 11-26
- ICBDR 6-10, 11-28
- IOCR 5-20, 11-29
- LR 2-7, 11-30
- MSR 1-8, 2-15, 5-7, 11-2, 11-31
- PIT 5-32, 11-33
- PMCR0 11-34
- PVR 2-10, 11-35
- R0-R31 2-4, 11-24
- reserved 11-1
- reserved fields 11-1
- SGR 11-36
- SLER 11-38
- special purpose 1-8
- SPR numbering 11-2
- SPRG0-SPRG3 2-10, 11-40
- SRR0 5-8, 11-41
- SRR0-SRR1 (non-critical) 5-8
- SRR1 5-8, 11-42
- SRR2 5-9, 11-43
- SRR2-SRR3 (critical) 5-9
- SRR3 5-9, 11-44
- summary 1-8
- TBHI 11-45
- TBHU 11-46
- TBLO 11-47
- TBLU 11-48
- TCR 5-34, 5-37, 11-49
- TSR 5-34, 5-36, 11-50
- XER 2-8, 11-51
- registers, device control 2-16
- registers, special purpose 2-5
- registers, summary 2-4
- reservation bit 10-96, 10-156
- reserved fields 11-1
- reserved registers 11-1
- reset
 - chip 4-1
 - core 4-1
 - DRAM controller behavior 4-5
 - processor initialization 4-6
 - processor state after 4-2
 - register contents after 4-2
 - system 4-1
 - types of 4-1
- rfci 10-129
- rfi 10-130
- rlwimi 10-131
- rlwimi. 10-131
- rlwinm 10-132
- rlwinm. 10-132
- rlwnm 10-135
- rlwnm. 10-135
- rotlw 10-135
- rotlw. 10-135
- rotlwi 10-133
- rotlwi. 10-133
- rotrwi 10-133
- rotrwi. 10-133
- xtended mnemonics
 - bnctr 10-30

S

- sc 10-136
- SC-form A-50
- secondary opcodes A-39
- SGR 8-10, 11-36
- signals
 - by signal name 12-3
 - ordered by pin number 12-9
- SLER 8-12, 11-38
- slw 10-137
- slw. 10-137
- slwi 10-134
- slwi. 10-134
- special purpose registers 2-5
- special purpose registers (SPRs) 1-8
- speculative accesses
 - 2-36

- speculative fetching 2-36
 - guarded storage 2-37
 - on the 401Core 2-36
 - on the 401GF 2-36
- SPRG0-SPRG3 2-10, 11-40
- sraw 10-138
- sraw. 10-138
- srawi 10-139
- srawi. 10-139
- SRR0 5-8, 11-41
- SRR1 5-8, 11-42
- SRR2 5-9, 11-43
- SRR3 5-9, 11-44
- srw 10-140
- srw. 10-140
- srwi 10-134
- srwi. 10-134
- stb 10-141
- stbu 10-142
- stbux 10-143
- stbx 10-144
- sth 10-145
- sthbrx 10-146
- sthu 10-147
- sthux 10-148
- sthx 10-149
- stmw 10-150
- storage attribute regions 2-3, 8-3
- storage attributes
 - byte ordering 8-12
 - cacheability (data) 8-6
 - control 8-3
 - control registers 8-3
 - guarded storage 8-10
- storage control 1-5
- storage synchronization 2-46
- stswi 10-151
- stswx 10-152
- stw 10-154
- stwbrx 10-155
- stwcx. 10-156
- stwu 10-158
- stwux 10-159
- stwx 10-160
- sub 10-161
- sub. 10-161
- subc 10-163

- subc. 10-163
- subco 10-163
- subco. 10-163
- subf 10-161
- subf. 10-161
- subfc 10-162
- subfc. 10-162
- subfco 10-162
- subfco. 10-162
- subfe 10-164
- subfe. 10-164
- subfeo 10-164
- subfeo. 10-164
- subfic 10-165
- subfme 10-166
- subfme. 10-166
- subfmeo 10-166
- subfmeo. 10-166
- subfo 10-161
- subfo. 10-161
- subfze 10-167
- subfze. 10-167
- subfzeo 10-167
- subfzeo. 10-167
- subi 10-9
- subic 10-10
- subic. 10-11
- subis 10-12
- subo 10-161
- subo. 10-161
- supervisor state 2-40
- sync 10-168
- synchronization
 - architectural references 2-42
 - context 2-42
 - execution 2-45
 - storage 2-46
- system call 5-24

T

- TBHI 11-45
- TBHU 11-46
- TBLO 11-47
- TBLU 11-48
- TCR 5-37, 11-49
- time base 5-29
- timer facilities
 - overview 5-28

- timers
 - FIT 5-34
 - fixed interval timer 5-34
 - PIT 5-32
 - programmable interval timer 5-32
 - TCR 5-37
 - time base
 - comparison with PowerPC 5-30
 - timer control register 5-37
 - timer status register 5-36
 - TSR 5-36
 - watchdog 5-34
- 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 10-170
- true Little Endian
 - storage attribute control 8-12
- TSR 5-36, 11-50
- tw 10-169
- tweq 10-171
- tweqi 10-174
- twge 10-171
- twgei 10-174
- twgle 10-171
- twgt 10-171
- twgti 10-175
- twi 10-173
- twle 10-171
- twlei 10-175
- twlgei 10-175
- twlgt 10-171
- twlgti 10-175
- twlle 10-171
- twllei 10-175
- twllt 10-171
- twllti 10-175
- twlng 10-171
- twlngi 10-175
- twlnl 10-171
- twlnli 10-175
- twlt 10-171
- twlti 10-175

- twne 10-172
- twnei 10-175
- twng 10-172
- twngi 10-176
- twnl 10-172
- twnli 10-176
- types 2-17

U

- unit, execution 1-4
- units
 - data cache 1-6
- units, instruction cache 1-6
- user mode 2-40

W

- Wait/Data state 3-4
- watchdog timer 5-26, 5-34
- wtee 10-177
- wteei 10-178

X

- XER 2-8, 11-51
- X-form A-51
- XFX-form A-52
- XL-form A-52
- XO-form A-52
- xor 10-179
- xori 10-180

