POWERPC PROGRAMMING for INTEL PROGRAMMERS

Your In-Depth Reference to the PowerPC Architecture

Analyze the Programmatic Differences in the PowerPC Family of Microprocessors

Features an Expanded and Annotated PowerPC Instruction Set Reference

KIP MCCLANAHAN
Systems Software Designer, Motorola Incorporated
POWERPC PROGRAMMING FOR INTEL PROGRAMMERS

Kip McClanahan
Welcome to the world of IDG Books Worldwide.

IDG Books Worldwide, Inc. is a subsidiary of International Data Group, the world's largest publisher of computer-related information and the leading global provider of information services on information technology. IDG was founded more than 25 years ago and now employs more than 7,500 people worldwide. IDG publishes more than 235 computer publications in 67 countries (see listing below). More than fifty million people read one or more IDG publications each month.

Launched in 1990, IDG Books Worldwide is today the #1 publisher of best-selling computer books in the United States. We are proud to have received 3 awards from the Computer Press Association in recognition of editorial excellence, and our best-selling ...For Dummies™ series has more than 15 million copies in print with translations in 24 languages. IDG Books, through a recent joint venture with IDG's Hi-Tech Beijing, became the first U.S. publisher to publish a computer book in the People's Republic of China. In record time, IDG Books has become the first choice for millions of readers around the world who want to learn how to better manage their businesses.

Our mission is simple: Every IDG book is designed to bring extra value and skill-building instructions to the reader. Our books are written by experts who understand and care about our readers. The knowledge base of our editorial staff comes from years of experience in publishing, education, and journalism — experience which we use to produce books for the '90s. In short, we care about books, so we attract the best people. We devote special attention to details such as audience, interior design, use of icons, and illustrations. And because we use an efficient process of authoring, editing, and desktop publishing our books electronically, we can spend more time ensuring superior content and spend less time on the technicalities of making books.

You can count on our commitment to deliver high-quality books at competitive prices on topics consumers want to read about. At IDG, we value quality, and we have been delivering quality for more than 25 years. You'll find no better book on a subject than an IDG book.

John Kilcullen
President and CEO
IDG Books Worldwide, Inc.
FOR MORE INFORMATION

For general information on IDG Books in the U.S., including information on discounts and premiums, contact IDG Books at 800-434-3422.

For information on where to purchase IDG's books outside the U.S., contact Christina Turner at 415-655-3022.

For information on translations, contact Marc Jeffrey Mikulich, Foreign Rights Manager, at IDG Books Worldwide; fax number: 415-655-3295.

For sales inquiries and special prices for bulk quantities, contact Tony Real at 800-434-3422 or 415-655-3048.

For information on using IDG's books in the classroom and ordering examination copies, contact Jim Kelly at 800-434-2086.

PowerPC Programming for Intel Programmers is distributed in Canada by Macmillan of Canada, a Division of Canada Publishing Corporation; by Computer and Technical Books in Miami, Florida, for South America and the Caribbean; by Longman Singapore in Singapore, Malaysia, Thailand, and Korea; by Toppan Co. Ltd. in Japan; by Asia Computerworld in Hong Kong; by Woodslane Pty. Ltd. in Australia and New Zealand; and by Transword Publishers Ltd. in the U.K. and Europe.
ABOUT THE AUTHOR

Kip McClanahan has been writing low-level code for PCs for over a decade. Before joining Motorola, Kip wrote x86 assembly language device drivers for various operating systems including DOS, NetWare, Windows, and UNIX. Shortly after the PowerPC 601 microprocessor reached first silicon, Kip joined Motorola to work on the development of the Motorola Computer Group's PowerPC-based personal computer systems.

During the initial PowerPC development efforts at Motorola, Kip realized that the transition from x86 assembly language to PowerPC assembly language could be facilitated with a little correlation between the two architectures. This book is the result of that realization.

Kip currently spends his time writing code for Motorola's RISC Software Group. This RISC software team's responsibilities include developing Motorola's PowerPC firmware and extensive PowerPC software development tools as well as porting Microsoft's Windows NT operating system to the PowerPC architecture.

When he's not working, Kip enjoys mountain biking, hiking, cooking, and brewing his own beer in the undeniably awesome town of Austin, Texas.

Kip McClanahan can be reached on the Internet via email at:
kip@io.com

There is a World-Wide-Web site set up for book-related information and PowerPC programming information at:
http://www.io.com/user/kip/PPCPROG.HTML

There is also an Internet mailing list for PowerPC programming-related discussions. To subscribe, send an email message with the word "SUBSCRIBE" as the only body text to the following address:
powerpc-pro-request@io.com
DEDICATION

For my parents: Dick and Gloria McClanahan. As far as light reaches, it will never fall on two more supportive and caring people.
ACkNOWLEDGMENTS

First and foremost, vast thanks to Jill, my honey, who put up with the demands on my time, late nights, bad moods, repeated requests for, “Will you type this in?” and still managed to be supportive throughout. Thank you.

Second and still foremost, let it be known that Rob Hummel is more powerful than a locomotive and able to leap tall buildings in a single bound. Constantly offering great suggestions and astute observations, Rob was fundamental to this effort. If I gain nothing else from this experience, I’ve gained a friend and peer.

Thanks also to the dedicated and professional team at IDG: Trudy Neuhaus, Kate Tolini, Denise Peters, and Amy Pedersen. To say that “It would not have been possible without...” would be an understatement.

Thanks to Carol and Belinda at Waterside Productions; Julie Meyer at Cunningham Communications, a fantastic resource, and thanks to Ian Suhrstedt for work on the World Wide Web page for PowerPC Programming for Intel Programmers.

There have been dozens of people at Motorola who have also made this a better book. In particular, I’d like to thank the following people for the following things: Jerry Young, who did a terrific job of editing, reviewing, and providing the Motorola perspective on many topics; John Southard for his superb technical review and insight (and always willing to listen to my fatigue-induced ranting and raving); Matt Holle for his technical review and hackery; Kumar Ranganathan for his technical review; and Dean Mosley, the man with the all the answers. Also thanks to Keith Diefendorff, Doug McQuaid, Ray Essick, Mike Phillip, and Clara Serrano for their technical input and review as well as Ilona Rossman, Glenn Stephens, David Zack, Nelda Currah, Madeline Brock, Bruce Pape, and Robert Yuan.

And finally, thanks to the Gang: Jon, Heidi, Kenny, Martin; Lee Deshler at IBM Corp. and Deni Connor (who got the ball rolling); as well as the Austin-Based BlackHawk Design Team: Doug, Lan, CW, Gary, C.Bala, Mike, Amy, Richard, Paul, Jayathi, and the rest! And thanks to John and Amelia Amershek as well as Don and Rita Anselmo.
# Contents

## Introduction

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motivating Progress</td>
<td>1</td>
</tr>
<tr>
<td>Seeking Information</td>
<td>2</td>
</tr>
<tr>
<td>Something for Everyone</td>
<td>2</td>
</tr>
<tr>
<td>Architecture vs. Implementation</td>
<td>3</td>
</tr>
<tr>
<td>Conventions</td>
<td>5</td>
</tr>
<tr>
<td>The Tangle of Terminology</td>
<td>5</td>
</tr>
<tr>
<td>Bits, Bytes, and Words</td>
<td>6</td>
</tr>
<tr>
<td>Register and Bit Field Conventions</td>
<td>6</td>
</tr>
<tr>
<td>Number Systems</td>
<td>7</td>
</tr>
<tr>
<td>Operand Order</td>
<td>7</td>
</tr>
<tr>
<td>Ready to Roll</td>
<td>7</td>
</tr>
</tbody>
</table>

## Chapter One — The PowerPC Transition

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Name, Same Game</td>
<td>9</td>
</tr>
<tr>
<td>Searching for Similarities</td>
<td>10</td>
</tr>
<tr>
<td>Preserving Order</td>
<td>10</td>
</tr>
<tr>
<td>Defining RISC Architecture</td>
<td>11</td>
</tr>
<tr>
<td>Common Programming Operations</td>
<td>13</td>
</tr>
<tr>
<td>Making the Transition</td>
<td>23</td>
</tr>
<tr>
<td>Popular MPU Comparison</td>
<td>24</td>
</tr>
<tr>
<td>Architecture Awaits</td>
<td>25</td>
</tr>
</tbody>
</table>

## Chapter Two — Foundations and Architecture

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Intel i486</td>
<td>29</td>
</tr>
<tr>
<td>Cache Unit</td>
<td>30</td>
</tr>
<tr>
<td>Floating-Point Unit</td>
<td>32</td>
</tr>
<tr>
<td>Integer (Fixed-Point) Unit</td>
<td>32</td>
</tr>
<tr>
<td>Segmentation and Paging Units</td>
<td>32</td>
</tr>
<tr>
<td>Meet The PowerPC Architecture</td>
<td>34</td>
</tr>
<tr>
<td>Architecture, Implementation, and Scope</td>
<td>35</td>
</tr>
<tr>
<td>The PowerPC 601</td>
<td>37</td>
</tr>
<tr>
<td>PowerPC Blocks, Bits, and Buses</td>
<td>37</td>
</tr>
<tr>
<td>The Instruction Unit</td>
<td>39</td>
</tr>
<tr>
<td>The Integer Unit</td>
<td>44</td>
</tr>
<tr>
<td>Floating-Point Unit</td>
<td>44</td>
</tr>
<tr>
<td>Memory Management Unit</td>
<td>44</td>
</tr>
<tr>
<td>The Cache Unit</td>
<td>45</td>
</tr>
<tr>
<td>Memory Unit and System Interface Unit</td>
<td>46</td>
</tr>
<tr>
<td>The PowerPC 603</td>
<td>47</td>
</tr>
<tr>
<td>The Instruction Unit</td>
<td>47</td>
</tr>
</tbody>
</table>
The Load/Store Unit .................................................................49
The Cache Unit ........................................................................49
The Completion Unit (Completion Buffer) ..................................49
Power Management Capabilities ...............................................50
The PowerPC 603e ....................................................................51
The PowerPC 604 ......................................................................51
Instruction Unit ........................................................................53
Multiple Integer Units ................................................................54
Floating-Point Unit ....................................................................54
Cache Units ...............................................................................55
Power Management Capabilities ...............................................55
The PowerPC 620 ......................................................................55
Instruction Unit ........................................................................57
Integer and Floating-Point Units ...............................................57
Cache Unit ...............................................................................57
System Interface Unit ...............................................................58
Summary ..................................................................................58

Chapter Three — Of Eggs and Endianners ...................................59
A Brief History of Endianness ....................................................59
Origin of the Terms ....................................................................60
The End-Side Story ....................................................................62
Endianness and Memory ..........................................................63
Byte Addressing within Multibyte Operands ..............................65
A Closer Look ...........................................................................67
PowerPC Endianness ...............................................................68
PowerPC Support for Bi-Endian Memory ....................................68
Switching Endian Modes ...........................................................70
Endian Conversion .....................................................................74
Summary ..................................................................................76

Chapter Four — The PowerPC Programming Model ....................77
Privilege Levels .........................................................................77
Registers ..................................................................................78
i486 Register Set ........................................................................79
    Supervisor-Level Registers ..................................................81
    User-Level Registers ..........................................................82
PowerPC UISA and VEA Register Set ........................................82
    General-Purpose Registers ..................................................83
    Floating-Point Registers .......................................................83
    Condition Register ................................................................85
    Floating-Point Status and Control Register ..........................88
    Integer Exception Register .................................................89
    Link Register ......................................................................91
    Count Register ....................................................................92
    Time Base Register ............................................................92
Contents

OEA and Implementation-Specific Register Set .................................................... 93
  Common OEA Registers .................................................................................. 95
  Machine State Register .................................................................................. 95
  Processor Version Register ............................................................................. 100
  Segment Registers .......................................................................................... 101
  Table Search Description Register .................................................................. 104
  General Special-Purpose Registers ................................................................ 105
  Machine Status Save/Restore Registers .......................................................... 106
  Decrementer Register ...................................................................................... 108
  Data Address Breakpoint Register ................................................................... 109
  External Access Register .................................................................................. 110
  Data Address Register ...................................................................................... 111
  Data Access and Alignment Exception Source Register .................................. 112
PowerPC 601 Register Set ................................................................................... 112
  Real-Time Clock Registers ............................................................................. 113
  Multiply Quotient Register ............................................................................. 115
  Block Address Translation Registers ............................................................. 116
  Checkstop Sources and Enables Register ...................................................... 119
  Debug Modes Register .................................................................................... 122
  Instruction Address Breakpoint Register ....................................................... 123
  Processor Identification Register ..................................................................... 124
PowerPC 603 Register Set ................................................................................... 112
  Block Address Translation Registers ............................................................. 126
  Hardware Implementation Register ................................................................ 128
  Memory Paging and Data Structures ............................................................... 129
PowerPC 604 Register Set ................................................................................... 134
  Machine State Register ................................................................................... 134
  Decrementer Register ...................................................................................... 136
  Hardware Implementation-Dependent Register .............................................. 136
  Instruction Address Breakpoint Register ....................................................... 137
  Performance Monitoring Registers .................................................................. 138
PowerPC 620 Register Set ................................................................................... 144
  Hardware Implementation-Dependent Register .............................................. 146
  Address Space Register ................................................................................... 148
  Bus Status and Control Register ..................................................................... 148
  Performance Monitoring Registers .................................................................. 149
Summary ............................................................................................................. 149

CHAPTER FIVE — ADDRESSING MODES AND OPERAND CONVENTIONS ........151

x86 Conventions ............................................................................................... 152
  Operand Types .................................................................................................. 152
  Operand Movement ........................................................................................... 153
  x86 Effective Address Calculation .................................................................... 154
PowerPC Conventions ....................................................................................... 155
  Three-Operand Format ....................................................................................... 155
  Naming Conventions ........................................................................................ 158
# Table of Contents

String Manipulation ................................................................. 342  
Accessing Data Structures ...................................................... 343  
Reading the Time Base Register ............................................. 347  
Advanced Topics ........................................................................ 348  
  Processor Version Determination ............................................. 348  
  Multiple Word Shifts ............................................................... 349  
  Floating-Point 4 x 4 Matrix Multiply ....................................... 351  
  BAT Register Manipulation .................................................... 354  
  Atomic Memory Accesses .......................................................... 356  
  Simplified vs. Unisimplified Mnemonics .................................... 359  
Summary ..................................................................................... 360

**CHAPTER TWELVE — TECHNIQUES AND TRICKS** ............................................... 361

  Programming Tips .................................................................... 361  
  Interleave Memory Accesses .................................................... 362  
  Interleave Integer Operations .................................................. 362  
  Avoid Load/Store Multiple/String Instructions ............................. 363  
  Exploit Rename Registers ........................................................ 364  
  Don't Serialize Execution ......................................................... 364  
  The Performance Monitoring Facility ........................................ 365  
    Performance Monitor Control Register .................................. 367  
    Performance Monitor Counter Registers ............................... 369  
    The Sampled Address Registers .......................................... 369  
    Performance Monitoring Interrupt ........................................ 371

**APPENDIX A** ........................................................................... 371

**APPENDIX B** ........................................................................... 657

**APPENDIX C** ........................................................................... 677

**APPENDIX D** ........................................................................... 689

**BIBLIOGRAPHY** ...................................................................... 697

**INDEX** .................................................................................. 699
INTRODUCTION

“Progress is a nice word. But change is its motivator…”
— Robert F. Kennedy

Motivating Progress

Undoubtedly, progress is a nice word. But when the pace of progress lags, a dramatic change is required to get our attention and reveal an improved perspective. We’ve just experienced such a change in the world of personal computers: the introduction of personal computer systems based on the new family of PowerPC microprocessors.

My purpose for writing this book is to allow you to be part of that progress; profound change doesn’t happen by itself. The blazing performance, low cost, and cutting edge technology of the RISC-based PowerPC architecture has the potential to outlast and outgrow the x86 family of CISC processors. I’m going to present a careful look at the PowerPC architecture and explain how it corresponds to the Intel world. Armed with this knowledge, you can transition smoothly between the two hardware platforms.

Today’s personal computer systems are high-volume, mass-appeal systems. IBM PC compatibles are based on the Intel (or AMD or Cyrix) x86-compatible CPUs. But it’s the software that drives sales of those popular computer systems. And
software will drive the sales of PowerPC-based machines as well. Achieve an understanding of the PowerPC architecture and you will create well-constructed software — the key to the success of the PowerPC family.

IBM, Motorola, and Apple alone cannot ensure the success of the PowerPC family. It's also up to you and me — the programmers who'll be writing the software that sells the systems. This book will teach you the details of the various PowerPC implementations and empower you to write software for the next generation of microprocessors.

**Seeking Information**

There is simply no single source of detailed PowerPC programming information that spans all the implementations while focusing on what programmers need to know. Moreover, there is no way for one of the largest bases of programmers in the world (Intel x86 and compatibles) to capitalize on their current knowledge when learning the inner workings of the PowerPC RISC microprocessors. Although we’re assured that the PowerPC family of microprocessors is destined to change the course of computing, there isn’t sufficient technical information available yet to potential programmers. This book changes that fact.

Intel x86 programmers have been forced to evolve into some of the most resourceful and skilled programmers in the world — simply due to the limitations of the x86 architecture and the extent to which it has been stretched. Give those creative x86 programmers a powerful new architecture and the tools to utilize it and watch out — something good is bound to result. Within the pages of this book, you'll find those tools and the details of their use.

**Something for Everyone**

As part of the team that wrote the firmware for Motorola's 603- and 604-based personal computer systems (code-named Blackhawk), it was natural for me to try to relate the aspects of the PowerPC architecture to those of the x86. It became clear to me that most Intel programmers would be able to come up to speed more quickly if the explanation of the PowerPC related new features to old, old tricks to new, and so on. I'd have understood much more, in less time, if I'd had this book when I started programming PowerPC processors. I've targeted this book at programmers experienced with programming Intel x86 systems. But this book has something to offer anyone who wants to understand and use PowerPC processors.
Like most books, this book is designed to be read sequentially. However, I seldom read any computer reference book sequentially and you probably don't either. With that in mind, here are a few pointers on how best to navigate the following pages.

I suggest that you skim through Appendix A (the instruction set reference) before anything else. Many of the instructions have C code/PPC (PowerPC) assembly examples that show how each instruction is used. I think you'll find it helpful to see how the code looks and when certain instructions are used in common operations.

If you're familiar with the basic architecture of the PowerPC, you can skip Chapter 2, “Foundations and Architecture,” which features a comparison between the x86 architecture and each implementation of the PowerPC architecture. After that, there is a natural flow of chapters, each building on concepts presented in the preceding one. If you don't have a specific question you need answered immediately, reading through the chapters in sequence will yield the most benefit.

Architecture vs. Implementation

Implementations of the PowerPC architecture can vary within well-defined bounds. Cache size and type, power management capabilities, and number of execution units can all vary between each part. Each PowerPC microprocessor has a different set of features, yet they’re all compatible PowerPC processors. In this section, we’ll examine the IBM architecture that inspired the PowerPC and the details of the PowerPC architecture itself.

In a sense, the PowerPC architecture is the offspring of the POWER architecture. If you haven’t already heard of IBM’s POWER (Performance Optimized With Enhanced RISC) architecture, this is a good place to be introduced. IBM has been involved with RISC since John Cocke developed the tenets of RISC at IBM’s T.J. Watson Research Center in the mid-1970s. Since then, IBM has defined the POWER architecture and developed the RS/6000 workstation based on POWER.

IBM reasoned that if the POWER architecture could be simplified, sped up, and extended to support multiple processors, it would appeal to a wide variety of system designers; and IBM could use the processor itself. Consequently, IBM decided that turning their multichip POWER processor into a single-chip (PowerPC) version would be appropriate for personal computers.

To make the transition from POWER to PowerPC easier, the PowerPC architects decided to make the first PowerPC processor (the 601) a bridge part. In other words, significant POWER architecture features would be
built into the 601 so the software porting effort would be reduced. The POWER-only features would then be eliminated from future PowerPC implementations, making future parts (603, 604, and so on) pure PowerPC architecture implementations. Understanding where the PowerPC architecture (and the 601's unique feature set) came from will help when we take a closer look at individual processor implementations in Chapter 2, "Foundations and Architecture."

Each PowerPC microprocessor is an individual implementation of the PowerPC architecture. The easiest way to understand this is by looking at how a particular processor is completely specified by the PowerPC architecture. There are three main "books" that constitute the PowerPC architecture and a fourth that is specific to each implementation of that architecture:

- Book I specifies the PowerPC user instruction set architecture (UISA), covering the base instruction set, registers, and other facilities available to programmers.
- Book II specifies the PowerPC virtual environment architecture (VEA) and describes the storage model, aliasing, atomicity, cache synchronization and control, and other facilities available to programmers.
- Book III specifies the PowerPC operating environment architecture (OEA). It describes the system (supervisor level) instructions, the machine state register (MSR), logical/virtual/physical translation, segmentation, paging, and other related facilities. Although the PowerPC architecture describes each of these facilities, some are optional. The feature set that each PowerPC processor implements is described in Chapter 2, "Foundations and Architecture."
- Book IV specifies the PowerPC implementation definition for each processor. It covers the unique implementation-dependent aspects of each microprocessor that are beyond the scope of the first three PowerPC architecture books, such as cache organization and control, translation lookaside buffer (TLB) organization and control, and special supervisor-mode instructions.

The first three books were originally bound independently of each other, but the most recent distribution binds them together in a single Morgan Kaufman publication called PowerPC Architecture. A separate Book IV, corresponding to individual PowerPC microprocessors, is published by Motorola, in the form of individual PowerPC user's manuals.
This book spans all four of the PowerPC architecture books (and more) to create a thorough presentation of each processor. Each implementation’s quirks and unique features will be examined from a programmer’s perspective to give you the best possible foundation of PowerPC knowledge.

**CONVENTIONS**

The inherent differences between the x86 and PowerPC architectures leave considerable room for confusion of syntax and semantics. The following sections nail down the notational conventions and terminology that we’ll use throughout the remainder of this book.

**The Tangle of Terminology**

The processor and software terminology in the PowerPC architecture specification varies just enough from the user documentation to make it interesting, as you’ll see. In the interest of making things clear, we need a list for future reference. When we use one of the following terms in this book, we’ll use the user documentation conventions because they parallel the widely available PowerPC user’s manuals.

<table>
<thead>
<tr>
<th>Architecture Specification Terminology</th>
<th>User Documentation Terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt</td>
<td>Exception</td>
</tr>
<tr>
<td>Fixed-point unit (FXU)</td>
<td>Integer unit (IU)</td>
</tr>
<tr>
<td>Extended opcode</td>
<td>Secondary opcode</td>
</tr>
<tr>
<td>Extended mnemonics</td>
<td>Simplified mnemonics</td>
</tr>
<tr>
<td>Effective address</td>
<td>Effective or logical address</td>
</tr>
<tr>
<td>Data storage interrupt (DSI)</td>
<td>Data access exception (DAE)</td>
</tr>
<tr>
<td>Direct store segment</td>
<td>I/O controller interface segment</td>
</tr>
<tr>
<td>Privileged mode (or privileged state)</td>
<td>Supervisor-level privilege</td>
</tr>
<tr>
<td>Problem mode</td>
<td>User-level privilege</td>
</tr>
<tr>
<td>Real address</td>
<td>Physical address</td>
</tr>
<tr>
<td>Storage (locations)</td>
<td>Memory</td>
</tr>
<tr>
<td>Storage (the act of)</td>
<td>Access</td>
</tr>
<tr>
<td>Store in</td>
<td>Write back</td>
</tr>
<tr>
<td>Store through</td>
<td>Write through</td>
</tr>
</tbody>
</table>
Bits, Bytes, and Words

As you would expect, we’ll often talk in terms of memory. When discussing indivisible units of memory, we’ll use terms such as bit, byte, and word. Another terminology convention is required in this case because the number of bits in a word differs between the x86 and PowerPC architecture. The following table summarizes the size of each term as it applies to the x86 family, the PowerPC family, and this book.

<table>
<thead>
<tr>
<th>Memory Unit</th>
<th>x86</th>
<th>PowerPC</th>
<th>This Book</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
<td>A bit...</td>
<td>is a bit...</td>
<td>is a bit.</td>
</tr>
<tr>
<td>byte</td>
<td>8 bits</td>
<td>8 bits</td>
<td>8 bits</td>
</tr>
<tr>
<td>word</td>
<td>16 bits</td>
<td>32 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>half-word (short)</td>
<td>Uncommon</td>
<td>16 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td>long</td>
<td>32 bits</td>
<td>Uncommon</td>
<td>32 bits</td>
</tr>
<tr>
<td>dword</td>
<td>32 bits</td>
<td>64 bits</td>
<td>64 bits</td>
</tr>
</tbody>
</table>

It is important for Intel x86 programmers to note that a word in the PowerPC architecture and in this book, is 32 bits wide. Bits and bytes are constant across all architectures mentioned in this book. When considering bits, bytes, and words, it is important to understand specific bit and byte ordering as well as bit labels (the side of the byte where bit 0 resides) within words. Chapter 3, “Of Eggs and Endians,” which focuses on the differences between big and little endian addressing schemes, covers bit and byte ordering thoroughly.

Register and Bit Field Conventions

Compared to the CISC x86 family, the RISC PowerPC family is a register wonderland. Because the volume of registers and associated bits can be confusing, we need a simple and efficient way to identify them. When discussing registers and bit fields within those registers, we’ll designate them in the following manner: REGISTER[BITFIELD]. For example, if we’re talking about the LT bit in the condition register (CR), it would be written: CR[LT]. Registers and their fields are covered in detail in Chapter 4, “The PowerPC Programming Model.”
Number Systems

Hexadecimal numbers are prefixed with 0x, as in the C programming language. Binary numbers are prefixed with 0b. Decimal numbers have no prefix at all. For example:

- 0x1fa4  16-bit hexadecimal number
- 0b10101111  8-bit binary number
- 65534  16-bit decimal number

Operand Order

Intel programmers are used to operands being evaluated from right to left, as in the following line of 80386 assembly code: movzx EAX,0x12345678. Here, the hexadecimal value 0x12345678 is placed in the EAX register. The Intel ordering scheme was designed to mimic algebraic notation. For example:

```assembly
mov EAX,0x12345678  =  EAX = 0x12345678
```

and

```assembly
add EAX,EBX  =  EAX = EAX + EBX
```

On PowerPC processors, operands can be evaluated from right-to-left or left-to-right, depending on the operation. A good rule to memorize is load to the left. In general, PowerPC instructions that load a value into a destination register are evaluated in Intel right-to-left fashion. Store operations, in contrast, are evaluated in left-to-right fashion. There are some exceptions, and complete details of operand order are discussed in Chapter 5, “Addressing Modes and Operand Conventions.”

Ready to Roll

As you read the remainder of this book, keep the following in mind: For the first time since the introduction of the IBM PC in 1981, we’re poised at the edge of a revolution in the personal computer industry. There are millions of Intel-based systems, millions of users, and billions of dollars worth of software. If PowerPC machines capture even a small share of the personal computer industry, the demand for software will grow exponentially.
You can't dispute the x86 family's current dominance of the personal computer industry; Intel possesses so much momentum that dramatic change won't happen overnight. But with the price/performance ratio of the PowerPC architecture, the backing by industry giants, and no backward compatibility limitations, the PowerPC architecture is positioned to compete with and outlast the x86 architecture. Profound change won't happen by itself. But armed with the knowledge in this book, you can be part of the progress.
"The art of progress is to preserve order amid change and to preserve change amid order."

— Alfred North Whitehead

NEW NAME, SAME GAME

Nearly all programmers have faced the prospect of learning an entirely new programming language. Sometimes it’s a career-based necessity, sometimes it’s just nice to understand another platform, and sometimes it serves as a good example of the kind of transitions we face in the computer industry.

Recently, I was in a situation that required a firm understanding of the Forth programming language — something I lacked. I found a public domain Forth interpreter and bought a book on the Forth language. After a week, I began to feel that I was spinning my wheels.

In desperation, I started looking for parallels between what I knew about programming in general and programming on this new Forth platform. For me, this was the turning point. Little similarities led to much larger parallels and soon I was comfortable writing short Forth programs. Whether it’s Forth, assembly language, or a new computer architecture, given sufficient information and motivation, it’s easy to learn almost anything.
Searching for Similarities

Up until my involvement with PowerPC-based computers, I'd been pro-
gramming x86 platforms almost exclusively. I've worked on other architec-
tures, but always considered x86 machines the home field. So during my
initial experimentation with PowerPC-based systems, I took the time to
explore the architecture thoroughly. Initially, it seemed as though all my x86
knowledge would be no help at all.

Faced with learning a new system architecture, it made sense to start
looking for the little similarities that would help bridge the transition
between x86 and PowerPC platforms. Common ground is a great place to
start. The remainder of this book will point out the common ground and
provide enough information to allow you to program PowerPC-based
machines efficiently.

Preserving Order

From the programmer's perspective (as opposed to that of the microproces-
sor designer), both the x86 and the PowerPC family of processors must do
similar things in a similar fashion. For example, consider the following
functions that must be implemented by both architectures:

- Both must have registers that are operated on by instructions.
- Both must load and store values in memory.
- Both must perform arithmetic operations.

Indeed, both architectures support the listed functions although they do
so in different manners. This is the first place where we'll visualize the dif-
ferences between the x86 and PowerPC architectures.

The following x86 assembly language code fragment loads the EAX reg-
ister with the hexadecimal value 0x1234abcd. That value is then stored in
the memory variable named VariableInMemory.

```
mov EAX, 0x1234abcd ; put immediate value 0x1234abcd
mov VariableInMemory, EAX ; store value in EAX into variable VariableInMemory
```

Brace yourself. The PowerPC assembly language equivalent is coded as
follows:

```
The PowerPC Transition

One of the first questions you might ask is: What are add and or instructions doing in the PowerPC example? In this case, they’re used as simple load instructions. In the example, the `addis` instruction adds the immediate value 0x1234 and zero (represented by `r0`) and shifts the result into the upper 16 bits of register `r3`. Similarly, the `ori` instruction loads the lower 16 bits by oring 0xabcd with the value in `r3`. The result is two add instructions accomplishing the same thing as the x86 `mov` instruction.

Your next question is bound to be, why does it take four instructions on the PowerPC processor and only two on the x86 processor? The answer is rooted in the fundamental characteristic of RISC. RISC architectures are load/store (also known as register/register) architectures. Load/store architectures allow only indirect manipulation of values in memory.

In the x86 code, we moved the value 0x1234abcd directly into the memory location specified by `VariableInMemory`. On a PowerPC RISC microprocessor, if you need to modify a variable, you must do so indirectly: Load the variable’s address into register A, load the value from that address into register B, perform the operation on register B, and store the results back using the address in register A.

In this simple example, you’ve seen one characteristic of RISC architectures that differs from an x86 implementation. Now is a good time to get acquainted with the nature of RISC architectures and the features we can expect to find in the PowerPC architecture.

**Defining RISC Architecture**

If you try to separate the RISC (reduced instruction set computer) and CISC (complex instruction set computer) worlds using criteria of instruction count or functionality, you’ll be lucky to distinguish between the two at all. Despite the name, the instructions on modern RISC processors aren’t necessarily reduced in number or complexity. In fact, RISC and CISC are converging, with characteristics of one architecture commonly appearing in the other.

Nonetheless, there are characteristics unique to each architecture. RISC architectures, such as the PowerPC, sport the following features:
- **General-Purpose Register Architecture**
  The instruction sets of RISC architectures have only explicit operands. That is, only registers and memory locations (specified indirectly using registers) may be used in common operations. In contrast, the x86 assigns most registers a dedicated purpose, such as indexing (ESI, EDI), port access (DX, AX), or counting (ECX).

- **Register/Register Architecture**
  As the name implies, a register/register (also called load/store) architecture allows manipulation of values contained in registers only; it does not allow direct modification of memory locations. To access the contents of memory, you must load the value from memory into a register, perform the desired operation, and then write the value back out to memory.

  There are several important performance benefits in a register/register architecture. In particular, instructions can be fixed-length, speeding instruction fetching and cache fills. Memory accesses and arithmetic operations are not coupled, allowing execution units to work on separate parts of an operation simultaneously. And because the operands for most operations are available in registers, there is generally less need to access main memory — which is always slower than accessing registers.

- **Fixed-Length Instructions**
  Unlike the x86, which has instructions from 1 to 15 bytes long, the PowerPC architecture specifies that each instruction be 4 bytes (32 bits) in length. On PowerPC processors, fetching instructions from memory and branching in programs can be optimized considerably due to the assumptions that can be made concerning 32-bit alignment of code in memory.

- **Three-Operand Instruction Format**
  Many PowerPC instructions require three operands. The idea behind the three-operand (two source, one destination) format is that the destination register can be independent of the two source values. Thus, the source values are left intact and are usable by subsequent operations without requiring reloading.

- **Multiple Execution Units**
  All PowerPC implementations have multiple execution units and, under optimal conditions, are able to dispatch one instruction to each unit during every clock cycle. In other words, more than one instruction can
be dispatched per clock cycle. This fundamental RISC feature is known as *superscalar operation* and is one of the primary advantages of RISC over classic CISC designs. While most superscalar microprocessors have multiple execution units, it is not a requirement.

- **Instruction Pipelining**
  Like nearly all modern performance-oriented CPUs, both the Intel i486 and the PowerPC processors, take advantage of *instruction pipelining*, the technique used to overlap the execution of multiple instructions. The instruction pipeline is discussed in Chapter 2, “Foundations and Architecture” and Chapter 7, “The Sublime Art of Instruction Timing.”

- **Pipelining, Parallelism, and Superscalar Operation**
  A superscalar microprocessor can issue multiple instructions each clock cycle from a conventional instruction stream. While each instruction may or may not be issued to separate pipelines, PowerPC implementations use separate pipelines for each execution unit. By issuing one instruction to each unit every clock cycle, instruction parallelism is achieved.

**Common Programming Operations**

Now let's continue to compare fundamental x86 and PowerPC programming operations. Since we've examined how the two architectures load and store values to registers and memory, we’re ready to examine a few more detailed examples. In this section, we’ll look at three progressively detailed examples: an implementation of the `toupper()` function, a string scanning example, and an array example. Each example will compare different aspects of programming the x86 and PowerPC architectures.

In each example, we’ll first look at a generic C implementation of the particular example. Next, we’ll look at the x86 assembly language implementation and note any distinguishing features. Finally, we’ll examine the PowerPC assembly language implementation. After seeing both assembly implementations, we’ll compare and contrast the two versions, operation by operation.

If the structure of the PowerPC code in the following examples seems unfamiliar, don’t worry — this introductory section is intended to be a starting point. We have the remainder of the book to fill in the details. By the end of this section, you’ll have a general sense of what it is like to program a PowerPC microprocessor.
Throughout this book, I use the C programming language as the high-level language of choice. I'll code a particular operation in C, then show the translation into both x86 and PowerPC assembly language.

An Implementation of toupper()

If we decided to break open a standard C library and investigate the typical implementation of the toupper() function, we'd probably find that it uses a lookup table. In terms of programming efficiency, the lookup table implementation is a better mechanism than the following example. However, using a lookup doesn't demonstrate the differences between the x86 and PowerPC architectures as well as our example does.

In our implementation of the toupper() function, we'll take an ASCII character as our argument and pass back its uppercase equivalent, if one exists. To convert a character to its uppercase equivalent, it must fall into the range of lowercase ASCII characters (0x61–0x7a). If a character outside this range is passed to our function, we'll simply return that character to the calling code.

```c
// C version of our toupper() implementation
/
int toupper(int ch)      // standard C definition of toupper()
{
    if ((ch >= 0x61) && (ch <= 0x7a)) // if the character is in range
        return(ch-0x20);           // convert lower- to uppercase
    else
        return(ch);              // out of range: return same character
}
```

This simple function, if coded correctly in assembly language, should require a subtract operation and a couple of compares to verify that the character falls into the proper range. We'll code the x86 version of toupper() first, then the PowerPC version. Following the code sequences, we'll compare each implementation. You'll be pleasantly surprised by the number of similarities between these two examples.

```
x86 version of toupper()
assumes:
EAX contains the character to convert, with the upper 16 bits zeroed out
returns:
```
The PowerPC Transition

; EAX contains the converted character
; toupperX86:
  cmp eax, 0x61 ; is the value less than 0x61?
  jb x86OutOfRange ; jump if below
  cmp eax, 0x7a ; is it greater than 0x7a?
  ja x86OutOfRange ; jump if above
  sub eax, 0x20 ; perform conversion
x86OutOfRange:
  ret ; return to caller

Fortunately, the assembly language version of `toupper()` closely parallels the C version. We'll assume that the C compiler can be configured to pass the character argument in EAX. The EAX register, containing the character to convert, is checked to verify that its contents are within the range of lowercase ASCII characters. If so, none of the jumps to `x86OutOfRange` are taken and a value of 0x20 is subtracted from EAX. At that point, execution falls through to the return statement where the `toupper()` function concludes.

The following PowerPC version of `toupper()` uses registers for parameter passing. This feature is common to general-purpose register (GPR) architectures, and we'll see it in all PowerPC code examples that require parameter passing.

; PowerPC version of toupper()
; assumes:
; r3 contains the character to be converted with upper 16 bits zeroed
; returns:
; r3 contains the converted character
; toupperPPC:
  cmpl 0,0,r3,0x61 ; is the character less than 0x61?
  blt PPCOutOfRange ; branch less-than
  cmpl 0,0,r3,0x7a ; is it greater than 0x7a?
  bgt PPCOutOfRange ; branch greater than
  addi r3,r3,-0x20 ; perform conversion
  PPCOutOfRange:
  bclr 0x14,0 ; branch always: return

At first glance, the PowerPC version looks quite similar to the x86 version. In fact, with the exception of the instruction format, the two implementations of `toupper()` are nearly identical. Let's take a minute to compare the two sequences.
By convention, certain GPRs have dedicated functionality. However, register usage is not defined at the PowerPC architecture level — or even at the processor implementation level. Register usage may vary depending on the programming language and other system-specific characteristics. For this reason, we'll keep convention usage to a minimum and concentrate specifically on programming the various PowerPC processors. Table 1-1 shows the register usage conventions that are used in the examples found in this book; a register that is labeled volatile does not have to be preserved across a function call.

The PowerPC `cmpi` instruction is basically equivalent to the x86 `cmp` instruction. So — what are all those operands? As described in the definition of `cmpi` in Appendix A, "PowerPC Instruction Set Reference," the first operand determines which field of the condition register (CR) is updated with the results of the compare. In this case, we are going to update the CR0 field. The second parameter is optional; it is always 0 on 32-bit PowerPC implementations and always 1 on 64-bit implementations. Because of the implementation dependency of the second parameter, it is often omitted from the instruction since the assembler would be configured for a specific target processor. The remaining two parameters are equivalent in function to the parameters of a compare operation on an x86; `r3` specifies the register whose contents are compared to the immediate value 0x61 (and 0x7a).

<table>
<thead>
<tr>
<th>Table 1-1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PowerPC Register Usage Conventions</strong></td>
</tr>
<tr>
<td><strong>Register Name</strong></td>
</tr>
<tr>
<td>----------------------------</td>
</tr>
<tr>
<td>GPR 0</td>
</tr>
<tr>
<td>GPR 1</td>
</tr>
<tr>
<td>GPR 2</td>
</tr>
<tr>
<td>GPR 3</td>
</tr>
<tr>
<td>GPR 4-10</td>
</tr>
</tbody>
</table>
The PowerPC branch instructions behave just like their x86 jump counterparts. However, if you have been reading closely, you may wonder how the branch instructions know which field of the condition register to use to determine if the branch is taken. After all, we went out of our way to specify that the compare operation update the CR0 field. The answer to this question introduces an important aspect of the PowerPC instruction set architecture — simplified mnemonics. The blt and bgt instructions are simplified mnemonics and actually represent specific forms of the more complex bc (branch conditional) instruction. The simplified form of these branch instructions assumes that CR0 is being used if no other field is specified. Simplified mnemonics are discussed in length in Chapter 6, "The PowerPC Instruction Set." For now, it is sufficient to understand that the blt and bgt instructions perform the same operation as the x86 jb and jag instructions.

If the character value is in range, we'll fall through both compare instructions and perform the conversion with the addic (add immediate with carry) instruction. We all remember from basic math that adding a negative value is the same as subtracting a positive value and that's exactly what we're doing with the addic instruction. We're adding \(-0x20\) to register r3 and storing the result back into r3.

Finally, we return to the caller after executing the bclr (branch conditional to link register). One of the options available when branching is to automatically update the link register with the address of the instruction following the branch (the return address). This common operation allows a subroutine to return to the caller by simply branching to the link register. The operands used with the bclr instruction simply mean branch always. Upon return to the caller, the converted character is located in the original register r3.

**A String Scanning Example**

If there's one thing we programmers tend to do frequently, it's scan strings. Scanning a string for a character or set of characters is a common operation whenever software needs to interface with people. To determine the length of a NULL-terminated string, it's natural to use the `strlen()` function — which makes it a good second example.

In the following example, the string passed to the `strlen()` function is scanned for the terminating NULL character. When the end of the string is located, the difference between the starting address plus one (explained in
the following examples) and address of the NULL character is returned to
the caller as the length.

```c
// typical C library implementation of strlen()

int strlen(char *s) {  
    char *sTemp = s + 1;  // account for zero-based starting address
    while (*s++ != '\0');   // scan string for NULL termination
    return (s - sTemp);   // difference between start and end address
}
```

Unlike our `toupper()` function, if you were to examine the implementa-
tion of `strlen()` in a typical C library, it would appear very similar to the pre-
ceding code. In the x86 and PowerPC assembly versions that follow, we'll
again see two very similar code sequences.

```
x86 version of strlen()

; assumes:
;   eax contains pointer to NULL-terminated string
; returns:
;   eax contains the length of the string
; the contents of registers edx and ebx are destroyed

strlenX86:
    lea ebx,[eax + 0x01]  ; set ebx equal to string pointer + 1

topOfLoopX86:
    mov edx,eax  ; get a copy of the string pointer
    inc eax  ; increment pointer
    cmp byte ptr [edx],0x00  ; is character a NULL?
    jne topOfLoopX86  ; no, continue scanning
    sub eax,ebx  ; yes, subtract two pointer values to get length
    ret  ; return to caller
```

In the x86 code in the preceding example, the first `lea` instruction
sets the EBX register equal to the string pointer plus one — just like the
first line of the C version. This is necessary because the pointer used in
the while-loop is post-incremented one time too many after the NULL
character is found. In the x86 assembly version, notice that EAX has
already been incremented by the time we test for the NULL character.
This behavior is not an error — the same functionality is present in the
PowerPC version following.
PowerPC version of strlen()

assumes:
- r3 contains a pointer to a null-terminated string
returns:
- r3 contains the length of the string

strlenPPC:
```
addic r10, r3, 1 ; r10 is set equal to r3 + 1
```

topOfLoopPPC:
```
  lbz r12, 0(r3) ; load a character from string into r12
  addic r3, r3, 1 ; increment our pointer: r3
  cmpi 0, r12, 0x00 ; is it a NULL character?
  bne topOfLoopPPC ; branch if not equal to topOfLoopPPC

  subfc r3, r10, r3 ; subtract r3 from r10 with carry
  bclr 0x14, 0 ; return to caller: length is in r3
```

It's time, again, to compare and contrast. The first instruction in the PowerPC code, addic, performs exactly the same function as the lea instruction in the x86 example. General-purpose register r3 contains the address of the string (our string pointer) and addic loads r10 with (r3 + 1). Note that r10 is acting as the character pointer variable sTemp from the C version of strlen().

Once inside the loop, we load the character (byte) pointed to by r3 into r12 in preparation for the compare operation. In the x86 version, we are able to compare the byte in memory with 0x00 (NULL character) — why not do that here? This points out the memory addressing restrictions associated with a register/register architecture: We must address memory indirectly. The PowerPC implementation of strlen() must load the character into a register before it can be operated on by the compare operation. Chapter 5, "Addressing Modes and Operand Conventions," contains a thorough discussion of memory addressing on PowerPC processors.

Once we've found the NULL character, we fall through the branch and subtract r3 from r10 to obtain the length of the string. By convention, the EAX register is used to return values from functions on x86 systems. If you examine the previous two examples, you'll see that EAX is used to hold the return value in both the toupper() and strlen() functions. On PowerPC processors, r3 is used in the same manner. This convention is listed in Table 1-1.
An Array Example

For our final example, let's use another common C construct: the for-loop. For the purposes of our example, we'll assume that all variables are global and both array1 and array2 are 10-element arrays. If we wanted to equate each element of array1 to its corresponding element in array2, we might use the following code fragment:

```c
for (q = 0; q < 10; q++)
    array1[q] = array2[q]; // equate each corresponding element
```

Compiling this into x86 assembly code produces the following listing:

```assembly
ExampleStartX86:
    mov      dword ptr q, 0x00 ; zero counter variable q
XLOOP1:
    cmp      dword ptr q, 0x0a ; compare immediate q to 10
    jb       XLOOP3 ; not at 10 yet, keep going
    jmp      x86Done ; we're done, jump to end
XLOOP2:
    inc      dword ptr q ; increment counter variable q
    jmp      XLOOP1 ; jump to top of loop to continue
XLOOP3:
    mov      eax, q ; get value of q in register EAX
    shl      eax, 0x02 ; multiply EAX by 4; get current offset
    mov      edx, eax ; now get value of q in edx from eax
    mov      eax, array2[edx] ; get value out of array2
    mov      array1[edx], eax ; store value in array1
    jmp      XLOOP2 ; back to top after increment
x86Done:
```

Although it's clearly machine-generated code, this x86 code fragment is readable. Now take a deep breath and examine the assembly code generated for the PowerPC.

```assembly
ExampleStartPowerPC:
    mov      dword ptr q, 0x00 ; zero counter variable q
PowerPC
PowerPC
PowerPC
PowerPC
PowerPC
PowerPC
PowerPC
PowerPC
PowerPC
PowerPC
```

Although it's clearly machine-generated code, this x86 code fragment is readable. Now take a deep breath and examine the assembly code generated for the PowerPC.
contents of registers need not be preserved
returns:
each element of array1 has been equated to the corresponding
element in array2

ExampleStartPPC:

addi r8,r0,0 ; load immediate r8 with zero
TopOfLoop:
slli r9,r8,2 ; multiply count by 4

addis r12,r0,array2@h ; high portion of array2's address
ori r12,r12,array2@l ; or in the low portion of the address
addc r12,r9,r12 ; current position = address(array2) + (count*4)

lwz r10,0(r12) ; load 32-bit value from array2

addis r12,r0,array1@h ; get address of array1 (upper)
ori r12,r12,array1@l ; (lower)
addc r9,r9,r12 ; current position = address(array1) + count*4

stw r10,0(r9) ; store 32-bit value into array1

addic r8,r8,1 ; increment counter
cmpi 0,r8,10 ; are we done with the loop?
blt TopOfLoop ; branch if less than 10
bclr 0x14,0 ; we're done - return

Although it's quite different, it's not necessarily more difficult to understand. Let's go through the PowerPC code step by step.

The first operation in both versions is to zero the counter variable, as shown in the following example. In the x86 version, the memory variable q is zeroed by moving zero into the variable. In the PowerPC version, general-purpose register r8 is used as the counter and it is zeroed using the add immediate instruction we saw in the previous example.

<table>
<thead>
<tr>
<th>x86 Operation</th>
<th>PowerPC Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov  dword ptr q,0x00</td>
<td>addi r8,r0,0</td>
</tr>
</tbody>
</table>

After the counter is cleared, the x86 version tests the counter variable for the limiting condition before proceeding. The PowerPC version does not perform this initial test, but checks for the limiting condition after one complete iteration.

At this point, both examples fall into the main loop. The first operation in either loop is the calculation of the current offset into both arrays. As
shown in the following lines, this is accomplished by multiplying the counter by four and storing the result in a register.

<table>
<thead>
<tr>
<th>x86 Code</th>
<th>PowerPC Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov eax, _q</td>
<td>sli r9, r8, 2</td>
</tr>
<tr>
<td>shl eax, 0x02</td>
<td></td>
</tr>
<tr>
<td>mov edx, eax</td>
<td></td>
</tr>
</tbody>
</table>

In the preceding lines, we see one of the advantages of having a register/register architecture. By using a register-based counter, the PowerPC version avoids having to transfer the count into a register, shift the value left, and finally transfer the value into the EDX register. Either way, at the end of this sequence, the x86 code contains the current array offset in the EDX register; the PowerPC code contains the current array offset in r9.

Next, a value from array2 is loaded into a register so that it may be stored into the equivalent position of array1. The following lines perform that operation.

<table>
<thead>
<tr>
<th>x86 Code</th>
<th>PowerPC Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov eax, array2[edx]</td>
<td>addis r12, r0, array2@h</td>
</tr>
<tr>
<td>ori r12, r12, array2@l</td>
<td>addc r12, r9, r12</td>
</tr>
<tr>
<td>addc r12, r9, r12</td>
<td>lwz r10, 0(r12)</td>
</tr>
</tbody>
</table>

In the x86 sequence, the value from array2 is loaded into the EAX register. Because the x86 can access memory directly, this operation can be performed in one instruction by indexing (with EDX) from the base address of array2. In contrast, the PowerPC version indirectly accesses memory by first loading the address of array2 into r12, adding the current offset to that address, and finally loading r10 with the value from array2.

In the following lines, the value that was loaded in the preceding sequence is stored into array1 at the same relative offset. In the PowerPC version, the value from array2 is stored at an address in array1 equal to the sum of r12 and r9. The memory addressing used for the store operation is analogous to the load operation described previously.

<table>
<thead>
<tr>
<th>x86 Code</th>
<th>PowerPC Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov array1[edx], eax</td>
<td>addis r12, r0, array1@h</td>
</tr>
<tr>
<td>ori r12, r12, array1@l</td>
<td>addc r9, r9, r12</td>
</tr>
<tr>
<td>addc r9, r9, r12</td>
<td>stw r10, 0(r9)</td>
</tr>
</tbody>
</table>
Now that both sequences have transferred one element of $array_2$ into $array_1$, the loop must iterate if necessary.

Finally, the counter is incremented and the loop continues until complete. The following sequences increment the counter in both versions. In the case of the x86 version, the loop jumps back to the initial compare operation and continues. The PowerPC version compares the count value to the limit value and iterates if the limit condition has not yet been reached.

\[
\begin{align*}
\text{x86 Code} & \quad \text{PowerPC Code} \\
\text{inc dword ptr q} & \quad \text{addic r8,r8,1}
\end{align*}
\]

**Example Summary**

By now, we've seen a considerable quantity of PowerPC code. The examples shown demonstrate many general similarities between the x86 and PowerPC architectures. However, there are clearly some differences between the implementation of the x86 and PowerPC examples. Most differences arise primarily out of the syntax and semantic rules of the PowerPC programming model. An understanding of these differences is the first step to being able to program the various PowerPC implementations with the same level of efficiency and understanding as the x86 family of microprocessors.

**Making the Transition**

The following items summarize some of the significant areas of divergence between the two processor families. We'll become familiar with each of these areas during the course of this book.

- **Programming Model**
  The registers available for use in both supervisor and user modes vary dramatically between architectures. The PowerPC has 32 general-purpose registers (GPRs), 32 floating-point registers (FPRs), and a handful of control registers. As you probably know, there are significantly fewer registers on an x86 processor. Additionally, most specific x86 operations presume the use of a subset of those registers.

- **Instruction Mnemonics**
  The instruction sets of the two architectures are significantly different. Both the number of parameters and the order of their interpretation vary between architectures. A complete explanation of the operation of
all PowerPC instructions is contained in Appendix A, “PowerPC Instruction Set Reference.”

- **Vocabulary**
  Because of the differences between architectures and the fact that the PowerPC architecture is relatively new, there are terms that may not yet be part of your vocabulary.

- **New Performance Features**
  The PowerPC implementations have numerous performance-enhancing features, such as improved floating-point and superscalar design.

- **The Expandable PowerPC Architecture**
  The PowerPC architecture is designed to be viable well into the future. PowerPC implementations can exist as either 32- or 64-bit microprocessors; the 60x series are 32-bit microprocessors and the 620 is a 64-bit implementation. The 620 can also be configured to function as a 32-bit processor, but has a unique set of features when running in 64-bit mode. Additionally, the PowerPC architecture has been designed with multiprocessing support in mind. The 601, 604, and 620 PowerPC processors have a rich set of multiprocessing facilities.

- **New Tools and Operating Environments**
  The compilers, assemblers, and operating environments will vary between platforms. Whenever possible, tool vendors try to maintain coherency between platforms, although this is not always possible. Where appropriate, we'll distinguish between how specific tools work on either platform.

**Popular MPU Comparison**
When a new microprocessor is announced it's usually surrounded by a flurry of facts and figures. And by the time you're interested in comparing one processor to another, odds are good that you've lost the original information. Fact sheets don't just show which processor is the fastest, but reveal how the processors compare architecturally. Rapid development cycles ensure that Table 1-2 won't remain current long. But the information shown won't change unless new versions are released without updating the processor name/number.
Architecture Awaits

In the next chapter, we’ll review the x86 architecture by considering the Intel i486. Then we’ll take a look at each of the PowerPC implementations: the 601, 603, 604, and 620. Of course, we’ll be comparing and contrasting with the x86 as we go. Here’s where it starts to get interesting.
## Table 1-2
Microprocessor Comparison Chart

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Company**</th>
<th>Date Introduced</th>
<th>MPU Type</th>
<th>Internal Cache (L1) instr/data (KB)</th>
<th>External Cache (L2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerPC 601</td>
<td>Motorola, IBM</td>
<td>10/1/92 *</td>
<td>RISC</td>
<td>32, unified</td>
<td>Externally controlled</td>
</tr>
<tr>
<td>50 MHz</td>
<td>Motorola, IBM</td>
<td>10/1/92 *</td>
<td>RISC</td>
<td>32, unified</td>
<td>Externally controlled</td>
</tr>
<tr>
<td>66 MHz</td>
<td>Motorola, IBM</td>
<td>10/12/93 *</td>
<td>RISC</td>
<td>32, unified</td>
<td>Externally controlled</td>
</tr>
<tr>
<td>80 MHz</td>
<td>Motorola, IBM</td>
<td>3/30/94 *</td>
<td>RISC</td>
<td>32, unified</td>
<td>Externally controlled</td>
</tr>
<tr>
<td>100 MHz</td>
<td>Motorola, IBM</td>
<td>100 MHz</td>
<td>RISC</td>
<td>32, unified</td>
<td>Externally controlled</td>
</tr>
<tr>
<td>PowerPC 601+</td>
<td>Motorola, IBM</td>
<td>ibd</td>
<td>RISC</td>
<td>32, unified</td>
<td>Externally controlled</td>
</tr>
<tr>
<td>100 MHz</td>
<td>Motorola, IBM</td>
<td>10/18/93 *</td>
<td>RISC</td>
<td>8/8</td>
<td>Externally controlled</td>
</tr>
<tr>
<td>80 MHz</td>
<td>Motorola, IBM</td>
<td>10/18/93 *</td>
<td>RISC</td>
<td>8/8</td>
<td>Externally controlled</td>
</tr>
<tr>
<td>PowerPC 603e</td>
<td>Motorola, IBM</td>
<td>4/19/94*</td>
<td>RISC</td>
<td>16/16</td>
<td>Externally controlled</td>
</tr>
<tr>
<td>80 MHz</td>
<td>Motorola, IBM</td>
<td>4/19/94*</td>
<td>RISC</td>
<td>16/16</td>
<td>Externally controlled</td>
</tr>
<tr>
<td>PowerPC 604</td>
<td>Motorola, IBM</td>
<td>4/19/94*</td>
<td>RISC</td>
<td>16/16</td>
<td>Externally controlled</td>
</tr>
<tr>
<td>PowerPC 620</td>
<td>Motorola, IBM</td>
<td>ibd</td>
<td>RISC</td>
<td>32/32</td>
<td>Internally controlled</td>
</tr>
<tr>
<td>Intel 80486</td>
<td>Intel</td>
<td>6/91</td>
<td>CISC</td>
<td>8/8</td>
<td>Internally controlled</td>
</tr>
<tr>
<td>50 MHz</td>
<td>Intel</td>
<td>6/91</td>
<td>CISC</td>
<td>8/8</td>
<td>Internally controlled</td>
</tr>
<tr>
<td>Intel Pentium</td>
<td>Intel</td>
<td>3/93</td>
<td>CISC</td>
<td>8/8</td>
<td>Internally controlled</td>
</tr>
<tr>
<td>66 MHz</td>
<td>Intel</td>
<td>4Q 94</td>
<td>CISC</td>
<td>8/8</td>
<td>Internally controlled</td>
</tr>
<tr>
<td>90 MHz</td>
<td>Intel</td>
<td>2Q 95</td>
<td>CISC</td>
<td>8/8</td>
<td>Internally controlled</td>
</tr>
<tr>
<td>120 MHz</td>
<td>Intel</td>
<td>16/8</td>
<td>CISC</td>
<td>8/8</td>
<td>Internally controlled</td>
</tr>
<tr>
<td>AMD K5</td>
<td>AMD</td>
<td>3Q 95</td>
<td>CISC/RISC</td>
<td>16/8</td>
<td>Internally controlled</td>
</tr>
<tr>
<td>100 MHz</td>
<td>AMD</td>
<td>5/89</td>
<td>CISC</td>
<td>4/4</td>
<td>Externally controlled</td>
</tr>
<tr>
<td>M68040</td>
<td>AMD</td>
<td>5/89</td>
<td>CISC</td>
<td>4/4</td>
<td>Externally controlled</td>
</tr>
<tr>
<td>25 MHz</td>
<td>DEC</td>
<td>2/92</td>
<td>RISC</td>
<td>16/16</td>
<td>Internally controlled</td>
</tr>
<tr>
<td>Alpha 21064A</td>
<td>DEC</td>
<td>2/92</td>
<td>RISC</td>
<td>16/16</td>
<td>Internally controlled</td>
</tr>
<tr>
<td>300 MHz</td>
<td>DEC</td>
<td>2/94</td>
<td>RISC</td>
<td>8/8</td>
<td>Internally controlled</td>
</tr>
<tr>
<td>MIPS R4400SC</td>
<td>MIPS/SGI</td>
<td>11/92</td>
<td>RISC</td>
<td>16/16</td>
<td>4</td>
</tr>
<tr>
<td>150 MHz</td>
<td>MIPS/SGI</td>
<td>11/92</td>
<td>RISC</td>
<td>16/16</td>
<td>4</td>
</tr>
<tr>
<td>MIPS R10000</td>
<td>MIPS/SGI</td>
<td>ibd</td>
<td>RISC</td>
<td>32/32</td>
<td>Internally controlled</td>
</tr>
<tr>
<td>HP PA7100</td>
<td>MIPS/SGI</td>
<td>ibd</td>
<td>RISC</td>
<td>32/32</td>
<td>Internally controlled</td>
</tr>
<tr>
<td>100 MHz</td>
<td>Hewlett-Packard Co.</td>
<td>2/92</td>
<td>RISC</td>
<td>16K / 16K</td>
<td>1 / 2</td>
</tr>
<tr>
<td>Super Sparc</td>
<td>Sun Microsystems and TI, Inc.</td>
<td>5 / 92</td>
<td>RISC</td>
<td>20K / 16K</td>
<td>Externally controlled</td>
</tr>
<tr>
<td>Ultra Sparc</td>
<td>Sun Microsystems and TI, Inc.</td>
<td>10/94</td>
<td>RISC</td>
<td>16/16</td>
<td>Internally controlled</td>
</tr>
</tbody>
</table>

* For the PowerPC microprocessors, "date introduced" refers to the date of first silicon.
** The MPC601 is being manufactured solely by IBM at their Burlington, Vermont fab. The MPC603 and MPC604 will be manufactured by both Motorola and IBM. Motorola's MOS11 fab in Austin, Texas, will produce both the PowerPC 603 and 604 microprocessors.
<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Number of Registers gp/fp</th>
<th>Max Instr. Issue Rate per Cycle</th>
<th>Number of Execution Units</th>
<th>Number of Pipeline Stages int/fp</th>
<th>Endianness</th>
<th>SPECInt92</th>
<th>SPECfp92</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerPC 601</td>
<td>32/32</td>
<td>3</td>
<td>3</td>
<td>4/5</td>
<td>Big, switchable</td>
<td>53</td>
<td>65</td>
</tr>
<tr>
<td>50 MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80 MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100 MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC 601+</td>
<td>32/32</td>
<td>3</td>
<td>3</td>
<td>4/5</td>
<td>Big, switchable</td>
<td>tbd</td>
<td>tbd</td>
</tr>
<tr>
<td>100 MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC 603</td>
<td>32/32</td>
<td>3</td>
<td>5</td>
<td>3/3</td>
<td>Big, switchable</td>
<td>60</td>
<td>70</td>
</tr>
<tr>
<td>66 MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80 MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC 603</td>
<td>32/32</td>
<td>3</td>
<td>5</td>
<td>3/3</td>
<td>Big, switchable</td>
<td>75</td>
<td>85</td>
</tr>
<tr>
<td>80 MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC 604</td>
<td>32/32</td>
<td>4</td>
<td>6</td>
<td>3/3</td>
<td>tbd</td>
<td>160</td>
<td>165</td>
</tr>
<tr>
<td>100 MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC 620</td>
<td>32/32 (64-bit)</td>
<td>6</td>
<td>tbd</td>
<td>tbd</td>
<td>225</td>
<td>300</td>
<td></td>
</tr>
<tr>
<td>133 MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel 80486</td>
<td>8/8</td>
<td>1</td>
<td>n/a</td>
<td>5/8</td>
<td>Little</td>
<td>27.9</td>
<td>13.1</td>
</tr>
<tr>
<td>50 MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Pentium</td>
<td>8/8</td>
<td>2</td>
<td>3</td>
<td>5/8</td>
<td>Little</td>
<td>67.4</td>
<td>63.6</td>
</tr>
<tr>
<td>66 MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90 MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120 MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMD K5</td>
<td>8/8</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>Little</td>
<td>130</td>
<td>75</td>
</tr>
<tr>
<td>100 MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M68040</td>
<td>16/8</td>
<td>1</td>
<td>n/a</td>
<td>3/6</td>
<td>Big</td>
<td>21</td>
<td>15</td>
</tr>
<tr>
<td>25 MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alpha 21064A</td>
<td>32/32</td>
<td>2</td>
<td>4</td>
<td>7/10</td>
<td>Little</td>
<td>130</td>
<td>184</td>
</tr>
<tr>
<td>200 MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alpha 21164</td>
<td>32/32</td>
<td>4</td>
<td>4</td>
<td>tbd</td>
<td>Little</td>
<td>est. 330</td>
<td>est. 500</td>
</tr>
<tr>
<td>300 MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIPS R4400SC</td>
<td>32/32</td>
<td>1</td>
<td>Super-pipelined</td>
<td>7/10</td>
<td>Big, switchable</td>
<td>88</td>
<td>97</td>
</tr>
<tr>
<td>150 MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIPS R10000</td>
<td>32/32</td>
<td>4</td>
<td>5</td>
<td>tbd</td>
<td>Big, switchable</td>
<td>est. 300</td>
<td>est. 600</td>
</tr>
<tr>
<td>200 MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HP PA7100</td>
<td>32/32</td>
<td>2</td>
<td>3</td>
<td>5/6</td>
<td>Big</td>
<td>81</td>
<td>150</td>
</tr>
<tr>
<td>100 MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sun Sparc</td>
<td>136/32</td>
<td>3</td>
<td>5</td>
<td>4/5</td>
<td>Big</td>
<td>80</td>
<td>100</td>
</tr>
<tr>
<td>60 MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ultra Sparc</td>
<td>tbd</td>
<td>4</td>
<td>9</td>
<td>4/5</td>
<td>Big</td>
<td>est. 275</td>
<td>est. 305</td>
</tr>
<tr>
<td>60 MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Microprocessor</td>
<td>Die Size (mm²)</td>
<td>Number of Transistors (millions)</td>
<td>Technology</td>
<td>Operating Voltage (volts)</td>
<td>Peak Power (watts)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------</td>
<td>----------------------------------</td>
<td>--------------</td>
<td>--------------------------</td>
<td>-------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC 601</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50 MHz</td>
<td>120</td>
<td>2.8</td>
<td>.6 µm CMOS</td>
<td>3.6</td>
<td>6.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 MHz</td>
<td>120</td>
<td>2.8</td>
<td>.6 µm CMOS</td>
<td>3.6</td>
<td>8.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80 MHz</td>
<td>120</td>
<td>2.8</td>
<td>.6 µm CMOS</td>
<td>3.6</td>
<td>10.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100 MHz</td>
<td>74</td>
<td>2.8</td>
<td>.5 µm CMOS</td>
<td>2.5</td>
<td>6.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC 601+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100 MHz</td>
<td>tbd</td>
<td>2.8</td>
<td>.5 µm CMOS</td>
<td>2.5</td>
<td>6.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC 603</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 MHz</td>
<td>85</td>
<td>1.6</td>
<td>.5 µm CMOS</td>
<td>3.3</td>
<td>2.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80 MHz</td>
<td>85</td>
<td>1.6</td>
<td>.5 µm CMOS</td>
<td>3.3</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC 603</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80 MHz</td>
<td>98</td>
<td>2.6</td>
<td>.5 µm CMOS</td>
<td>3.3</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC 604</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100 MHz</td>
<td>196</td>
<td>3.6</td>
<td>.5 µm CMOS</td>
<td>3.3</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC 620</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>133 MHz</td>
<td>331</td>
<td>7</td>
<td>.5 µm CMOS</td>
<td>3.3</td>
<td>30***</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel 80486</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50 MHz</td>
<td></td>
<td>1.2</td>
<td>.8 µm CMOS</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Pentium</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 MHz</td>
<td>295</td>
<td>3.1</td>
<td>.8 µm BiCMOS</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>90 MHz</td>
<td>295</td>
<td>3.1</td>
<td>.8 µm BiCMOS</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>120 MHz</td>
<td>163</td>
<td>3.3</td>
<td>.6 µm BiCMOS</td>
<td>3.3</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMD K5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100 MHz</td>
<td>225</td>
<td>4.3</td>
<td>.5 µm CMOS</td>
<td>3.3</td>
<td>tbd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M68040</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25 MHz</td>
<td>8</td>
<td>1.2</td>
<td>.8</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alpha 21064A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200 MHz</td>
<td>194</td>
<td>1.68</td>
<td>.68 µm CMOS</td>
<td>3.3</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alpha 21164</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>300 MHz</td>
<td>300</td>
<td>9.3</td>
<td>.5 µm CMOS</td>
<td>tbd</td>
<td>50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIPS R4400SC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>150 MHz</td>
<td>186</td>
<td>2.3</td>
<td>.6 µm CMOS</td>
<td>5/3.3</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIPS R100000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200 MHz</td>
<td>298</td>
<td>5.9</td>
<td>.5 µm CMOS</td>
<td>tbd</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HP PA7100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100 MHz</td>
<td>201</td>
<td>.85</td>
<td>.8 µm CMOS</td>
<td>5</td>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Super Sparc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60 MHz</td>
<td>256</td>
<td>3.1</td>
<td>.7 µm BiCMOS</td>
<td>5</td>
<td>14.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ultra Sparc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60 MHz</td>
<td>315</td>
<td>3.8</td>
<td>.5 µm CMOS</td>
<td>tbd</td>
<td>20</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*** PPC620 dissipates 30W at 133 MHz in a worst-case scenario.
Knowing the architecture is everything. There's a direct relationship between understanding a processor's architecture and creating quality code. The x86 platform is a perfect example of a family of compatible processors that requires implementation-specific knowledge to achieve its best performance. Learn how and why the processor becomes inefficient and it becomes a simple task to increase throughput on the system.

The PowerPC family of RISC microprocessors poses similar obstacles for programmers. To achieve optimum performance when switching between the 601, 603, 604, and 620 you must understand the cache, memory management unit, instruction queue, and various execution units — the keys to learning how code flows through the processor. Fortunately, these aspects of each PowerPC chip can be examined from implementation to implementation. Given an understanding of the architecture, the PowerPC world is at your fingertips.

Nothing beats a well-annotated block diagram when you need a quick system overview — that's why you'll find six in
this chapter. The block diagrams for each PowerPC implementation explain the unique features of each and show the PowerPC architecture as a whole.

As we begin to look at the PowerPC microprocessors, keep in mind that we need a perspective that allows us to distinguish architectural similarities and differences. To do so, we'll need to consider where we came from (the Intel x86 family) and where we're going (PowerPC microprocessors, of course). Using familiar x86 concepts to illustrate aspects of the PowerPC architecture will reduce the effort required to learn a new architecture and programming model.

THE INTEL i486

Intel introduced the 8086 in 1978. IBM introduced its PC in 1981. Since then, the x86 family has evolved considerably. It wasn't until the first i486 appeared in 1989 that the x86 family finally acquired the features characteristic of high-performance processors: multiple execution units, burst cycles, and an on-chip (primary) cache.

The i486 is well-understood, popular, and sufficiently powerful. Although ostensibly a CISC design, the i486 includes a number of features that are traditionally found in RISC designs such as each PowerPC family member. This is the primary reason for using the i486 as the basis for the discussions that follow.

Figure 2-1 shows the i486's architecture. It's going to be helpful to compare the units that constitute an i486 compared to the elements of each PowerPC microprocessor. The more analogies we can draw between the two architectures, the easier it will be to gain a working perspective of the PowerPC family. With that in mind, let's pick apart the various components of the i486.

Cache Unit

The i486 contains an 8K unified code/data cache. While fundamentally a write-through-style cache, the i486 has the ability to postpone or reorder pending writes to improve performance. Adherence to a strict write-through policy means all writes to the cache are passed immediately to external memory. Later revisions of the i486 support write-back operation. By
Figure 2-1
The internal architecture of the popular i486 is an excellent basis for comparison to PowerPC microprocessors.
default, the 601 follows a write-back policy, requiring cache values and data status to be maintained until external memory is updated. The 601 allows its write policy to be switched on a per-page or block basis.

**Floating-Point Unit**

The i486’s floating-point unit (FPU) uses the 32-, 64-, and 80-bit formats of the IEEE 754 Floating-Point Standard. The Intel i486 was the first x86 family member to offer an on-chip FPU. Earlier 80x86 chips used a matching 80x87 external math coprocessor. Because floating-point support was optional in PC systems, and given the fact that the i486 integer unit outperforms the FPU, general-purpose PC applications that use floating-point instructions are rare.

In contrast, the PowerPC architecture has put particular emphasis on floating-point support and performance. Appendix C, “Floating Point on the PowerPC,” contains a review of floating-point concepts and a detailed look at floating point on PowerPC microprocessors.

**Integer (Fixed-Point) Unit**

The i486 integer unit (IU) — also call the fixed-point or datapath unit — performs all arithmetic and logical operations required by the processor’s instruction set. The PowerPC architecture also defines a fixed-point unit to perform similar operations. Some PowerPC implementations, such as the 604 and 620, have multiple fixed-point units.

**Segmentation and Paging Units**

The *Intel Hardware Reference Manual* defines a segment as “a protected, independent address space.” The segmentation unit enforces isolation among application programs and thereby isolates the effects of programming errors. Using data structures called *segment descriptors*, the segmentation unit converts the selector:offset address into a 32-bit linear address as shown in Figure 2-2. The linear address is then passed to the paging unit or the cache unit. To optimize segmentation translation, the i486 caches up to six segment descriptors in internal registers.
The i486 paging unit uses data structures called page tables to map a page of linear addresses produced by the segmentation unit to any corresponding page of physical addresses, as shown in Figure 2-2. A page is simply a contiguous 4K chunk of linear address space. For example, you can program the paging unit to make the memory at physical address 0x10000 appear at linear address 0xff800000 by filling in a page table with appropriate values.

Without the functionality of segmentation and paging, most of the advanced operating systems that we use today would not be possible. These mechanisms allow us to use virtual address spaces that can be considerably larger than our computer system’s physical memory size. They also allow the operating system to manipulate data structures that are larger than physical memory by swapping part of the data out to a hard disk. Memory management is discussed in Chapter 8.

It may seem strange to use three different memory addressing schemes (logical, linear, and physical) when we could have one big flat memory model that goes from 0 to 0xffffffff. Of course, even if such a model was
available, we wouldn’t want to pay for the 4GB of RAM such a system would require. And although it would be simple in concept, we’d get frustrated with the limitations of that configuration.

Segmentation and paging abstract and translate memory addresses into many different forms. Bear in mind, however, that when an address is placed on the processor bus, into the cache, or is passed to an external bus master (such as a sound card), it must represent physically present memory. Knowing when addresses are logical, linear, or physical is simplified considerably by knowing where the address is being used in the system.

**Translation Lookaside Buffer**

The *translation lookaside buffer* (TLB) is a component of the paging unit that caches page table entries. By keeping the 32 most recently used page table entries in the TLB, the paging unit can efficiently translate often-used linear addresses. This, in turn, speeds up the process of delivering physical addresses to the components that require them. Note that TLBs are not defined or required by the PowerPC architecture. Rather, TLBs are implemented on a per-processor basis. However, all PowerPC processors discussed in this book use TLBs. Keep the concept of translation lookaside buffers in mind. We’ll see them again when we talk about the PowerPC architecture and each processor implementation.

**MEET THE POWERPC ARCHITECTURE**

Reviewing the x86 architecture is helpful, but there’s quite a bit of new information ahead. Remember that the PowerPC architecture is simply that — a general RISC architecture. Each PowerPC microprocessor is a unique implementation of that architecture. The processors will have many common components and mechanisms, but may also differ in significant ways. For example, the PowerPC architecture is general enough to allow PowerPC implementations to be targeted at markets ranging from embedded applications to desktop computers.

While reading through the descriptions of the processors, you may find it helpful to refer back to Table 1-2, “Microprocessor Comparison Chart.” We’ll work our way through all the processors’ units and examine their dependencies and how they work together. We’ll examine all the similarities, all the differences, and how they affect each particular implementation. The best approach is to start at the beginning with the PowerPC architecture.
Architecture, Implementation, and Scope

The PowerPC architecture is a flexible specification that accommodates a variety of processor implementations. Each PowerPC processor (601, 603, 604, and 620) is an implementation of the PowerPC architecture.

As programmers, we’re often interested only in how each processor works — not the historical details of where each feature is defined. However, understanding the scope of the PowerPC architecture and the differences between architecture and implementation provides a helpful context for the discussions in this chapter and the rest of the book.

Architecture Definitions

To ensure software compatibility between each PowerPC processor implementation, the PowerPC architecture defines the following architectural features:

- **The Instruction Set**
The PowerPC instruction set includes the actual instructions, the forms used for encoding the instructions, and the addressing modes used during memory accesses.

- **The Programming Model**
The programming model defines the register set and the memory conventions, including details regarding the bit and byte ordering, and the conventions for how integer and floating-point data is stored.

- **The Memory Model**
The memory model defines a processor’s memory addressing capabilities and its relationship to caching, byte ordering (big or little endian), aliasing, and coherency.

- **The Exception Model**
The exception model defines the common set of exceptions and the conditions that can generate those exceptions. The exception model defines the exception vectors and a set of registers used when exceptions are taken. The exception model also provides memory space for implementation-specific exceptions.
The Memory Management Model
The memory management model defines how memory is partitioned, configured, and protected (also referred to as the processor's context). The memory management model also specifies how memory translation is performed.

The Time-Keeping Model
The time-keeping model defines the facilities that permit the time of day to be determined and the resources and mechanisms required for supporting time-related exceptions.

Outside the Architecture
An important part of understanding the scope of the PowerPC architecture is knowing what is not defined by the architecture specification. That is, each of the features and facilities described below can vary from processor to processor, as described in this chapter. To remain flexible, the PowerPC architecture does not define the following features:

- System Bus Interface Signals
Although the numerous implementations may have similar interfaces, the PowerPC architecture does not define individual signals or the bus protocol. For example, OEA allows each implementation to determine the signal or signals that trigger some exception conditions.

- Cache Design
The PowerPC architecture does not define the size, the set associativity, or the mechanism used for maintaining coherency for an on-chip (level 1) cache. The architecture supports, but does not require, the use of separate instruction and data caches. Likewise, the architecture does not specify the method by which cache coherency is ensured.

- The Number/Nature of Execution Units
The PowerPC architecture facilitates the design of processors that use pipelining and parallel execution units to maximize instruction throughput. However, the PowerPC architecture does not define the internal hardware details of a processor implementation. For example, one processor may execute load/store operations in the integer unit (PPC 601), while another may execute load/store instructions in a dedicated load/store unit (PPC 603, 604, and 620). Additionally, the architecture
does not prescribe which execution unit is responsible for executing a particular instruction.

- **Other Micro-architecture Issues**
  There are a number of other features that the PowerPC architecture does not define, such as: the details regarding the instruction fetching mechanism, how instructions are decoded and dispatched, and how results are written back to the architectural register set.

### The PowerPC 601

The 601 spearheaded the invasion of PowerPC processors. As the first implementation of the PowerPC architecture, the adoption of the 601 could popularize the PowerPC architecture. This goal would be facilitated if it were easy for system designers to rapidly implement a shipable 601 computer system.

At the time of the 601's launch, IBM was shipping RS/6000 systems based on its POWER architecture. IBM intended the 601 to serve as a bridge chip, spanning the gap between its current POWER systems and the emerging PowerPC architecture. A 601-based RS/6000 could ship quickly if the 601 architecture was similar enough to the POWER architecture. As a result, the IBM/Motorola/Apple consortium included POWER features in the design of the PowerPC 601.

In theory, by including key POWER instructions in the 601’s instruction set, the time required to put out a 601-based system would be greatly reduced. The reality was quite different. Apple beat every other PowerPC system manufacturer to the punch, releasing the PowerMacs in mid-1994. A few months later, IBM released its RS/6000 system based on the 601 and the rest is history in the making.

### PowerPC Blocks, Bits, and Buses

When working with the 601, it takes time to understand the relationships between the different units and “blocks” in the diagrams. Figure 2-3 presents an overview of the 601 microprocessor and shows the interactions between the general functional units.
Each PowerPC microprocessor has its own unique set of execution units. The 601 contains an integer unit (IU), a floating-point unit (FPU), and a branch processing unit (BPU), each of which processes a subset of the processor's instruction set. Additionally, the PowerPC architecture supports both independent execution units and out-of-order instruction issue. In
other words, while the IU and FPU are working on integer and floating-point instructions, the BPU can browse the bottom half of the instruction queue and resolve conditional branches both early and out of order.

The instruction unit comprises several smaller units that work together to direct instruction flow successfully. The integer unit and floating-point unit are the primary execution units. The memory management unit supplies cache tag information and is fed from the integer unit. Add a cache and system interface unit, and we've got the 601 sketched out.

**The Instruction Unit**

The instruction unit comprises the instruction queue, issue (or dispatch) logic, and the branch processing unit (BPU). Figure 2-4 illustrates the relationship between the instruction unit and its components. This is the 601’s control center for instruction traffic. The IU determines the address of the next instruction to be fetched using each of its three components and many concepts that are fundamental to the PowerPC architecture: instruction prefetch, branch prediction, out-of-order operation, and branch folding. Each of these concepts are fully detailed in Chapter 7, “The Sublime Art of Instruction Timing.”

**Out-of-Order Dispatch and Execution**

The PowerPC’s instruction unit allows instructions to be issued — and even completed — in an order that differs from their appearance in the instruction stream. Instructions that have no dependencies can complete regardless of currently pending instructions. This out-of-order instruction dispatch and execution enhances performance.

The instruction unit and the integer unit (discussed in the following section) reconstruct the original program order using instructions in the integer unit’s pipeline as a reference to recall the correct sequence. External to the processor, the code appears to execute in precisely the order it was written — it’d be quite a mess otherwise!

**The Instruction Queue**

The instruction queue (IQ) holds up to eight instructions labeled Q7 through Q0, where Q7 represents the top entry and Q0 is the bottom entry. Instructions are dispatched from lower queue entries and replaced into the
This detailed PowerPC 601 block diagram shows the relationship between the instruction unit and its components.
upper entries. As instructions are dispatched, instruction queue entries move down from Q7 toward Q0.

Integer instructions are issued from only the Q0 entry in the instruction queue. Floating-point and branch instructions are issued from the Q3 through Q0 entries. A detailed analysis of instruction dispatch, branch prediction, and branch folding’s effect on overall performance is presented in Chapter 7, “The Sublime Art of Instruction Timing.”

When the instruction queue has available space, it is refilled from the cache. Refilling the queue can be done in a single processor clock cycle (assuming a cache hit) using a burst read from the cache.

Introduction to the PowerPC Pipeline Concept

Superscalar instruction execution on PowerPC processors relies on instruction pipelining. Instruction pipelining is implemented in hardware and is transparent to the programmer. And understanding of how the PowerPC pipeline works is essential to writing efficient code. Chapter 7, “The Sublime Art of Instruction Timing,” examines the PowerPC pipeline in detail. Until then, a basic understanding of pipelining will suffice for our preliminary discussions.

The most common analogy used to describe the pipeline mechanism is that of a factory assembly line: Each pipeline stage specializes in one aspect of executing the instruction and is therefore quite efficient. Since each stage has only one function, new instructions can be shifted into that stage as soon as the previous job is done. Figure 2-5 shows an example of a simple pipeline and how instructions flow through each stage.

Ideally, an instruction passes through each stage of the pipeline in one clock cycle. In practice, there are many factors that affect the flow of instructions through the pipeline. The idea is that after instruction#1 leaves the first stage of the pipeline (instruction fetch), the execution unit is free to fill the first stage with instruction#2. In this manner, the pipeline increases the instruction throughput (the number of instructions executed per clock cycle) by working on several instructions in tandem.

Let’s take a closer look at how instruction#1 passes through our example pipeline in Figure 2-5.
During clock cycle 1, instruction#1 is fetched from memory (typically the instruction cache).

During clock cycle 2, instruction#1 is passed to the decode stage, where it is analyzed and dispatched to the appropriate execution unit. Also, instruction#2 is fetched into the first stage.

During clock cycle 3, instruction#1 passes into the execution stage. At this point, the processor is ready to perform logical or arithmetic operations or a register load/store. At the same time, instruction#2 is decoded and instruction#3 is fetched.

During clock cycle 4, the results of instruction#1 being executed are written back to the status registers as appropriate. At this point, instruction#1 has completed processing and the pipeline is full. If there are no pipeline stalls, one instruction will execute per clock cycle starting with cycle 5.

The pipeline implementation on each PowerPC processor is considerably more complex than this simple pipeline example. But it's sufficiently explained for now — we’ll need this information for discussions in the following chapters.
Branch Processing Unit

The branch processing unit (BPU) scans the lower portion of the instruction queue (Q3 through Q0), looking for branch instructions. When it finds an unconditional branch or a branch that depends on an available conditional value, the BPU resolves the branch immediately and the next sequential instruction takes the place of the branch. This is called branch folding. Branches that can be folded out of the instruction sequence take zero cycles to complete and do not interrupt instruction dispatch to other execution units.

When a conditional branch depends on unavailable information, the BPU attempts to predict whether the branch will be taken or not. This process is known as branch prediction. The 601 and 603 use static branch prediction to determine the direction that the branch will take. Dynamic branch prediction, used by the 604 and 620, is discussed later on in this chapter.

By default, the 601 and 603 predict that a branch will be taken if the target address displacement is negative. If the target address displacement is positive, the BPU guesses that the branch will not be taken. A bit in each branch opcode allows you to reverse the target address displacement assumption. Using these opcodes, compilers can generate code to make the processor guess more accurately.

After the branch direction has been predicted, the processor then fetches instructions from the target address until the branch is actually resolved. Eventually, sufficient conditional information is available to confirm the BPU’s guess. If the BPU was right, the processor continues execution of the predicted instructions. In effect, the predicted branch instruction (and subsequent instructions) is replaced by instructions from the target address. If the BPU guessed incorrectly, the processor flushes all instructions currently in the instruction queue and begins execution along the proper instruction path.

The BPU is commonly considered an execution unit (like the integer or floating-point units) because it uses its own registers and executes its function — resolving branches — independently of the other units.

Issue Logic

During each processor clock cycle, the issue logic dispatches up to three instructions from the lower half (bottom four entries) of the instruction
queue. The issue logic is closely coupled with the instruction queue and branch prediction unit.

**The Integer Unit**

The *integer unit* (IU) receives all integer, all load/store, and some floating-point instructions from the Q0 position of the instruction unit’s instruction queue. The IU is alternately known as the fixed-point unit.

Just as with the Intel i486 integer unit, the PowerPC 601’s IU is the workhorse of the processor. It is responsible for executing loads and stores (both integer and floating-point), memory management instructions, all integer arithmetic and logic instructions, and special-purpose register instructions.

To execute instructions, the IU uses 32 general-purpose registers (GPRs) and an *internal arithmetic logic unit* (ALU). During instruction execution, the IU uses rename registers (defined in the following section) to minimize pipeline stalls due to GPR contention. The integer unit relies on the memory management unit and cache to satisfy any access to main memory required by instructions.

**Rename Registers (Buffers)**

Rename buffers are used to track the results associated with an executing instruction. When a PowerPC processor’s instruction unit dispatches an instruction, a rename buffer is allocated for use by the instruction as it proceeds through the various stages of execution. If the instruction completes successfully, then the results in the rename buffer are used to update the architectural registers. However, if the instruction is a mispredicted branch or otherwise fails to complete successfully, the rename buffer provides an easy way to disregard the results of executing that instruction without significant overhead.

**Floating-Point Unit**

Like the 601 integer unit, the *floating-point unit* (FPU) receives instructions from the instruction unit’s queue. To ensure that the instruction unit does not wait for lengthy floating-point instructions to complete, the FPU has its own two-instruction queue. By removing floating-point instructions from
the main instruction queue and placing them in the FPU’s smaller queue, integer and branch instructions have time to execute in parallel.

The 601’s floating-point unit uses its thirty-two 64-bit floating-point registers (FPRs) to operate on both single- and double-precision floating-point numbers. Like the i486’s FPU, the 601’s FPU complies with the IEEE 754 standard. And there is hardware support for all IEEE 754 data types.

**Memory Management Unit**

The *memory management unit* (MMU) in the 601 performs roughly the same function as the segmentation and paging units of the i486. In fact, it is capable of quite a bit more.

The 601’s MMU supports not only segment- and page-based address translation, but block-oriented translations as well. It’s helpful to think of the 601’s MMU as simply an address broker. Regardless of whether the address was generated by the instruction unit (for use when fetching or branching) or by the integer unit (for accessing data), the MMU controls address translation and enforces memory protection.

The MMU’s functionality is specified by the PowerPC architecture, which covers the operating environment architecture (OEA) for both 32- and 64-bit implementations. The 601’s MMU does not strictly conform to the PowerPC architecture specification. Here are the areas where the 601 strays from the PowerPC architecture:

- The protection mechanism provided by *block address translation* (BAT) registers differs from the PowerPC OEA architecture definition.
- The SRR1 register’s bit definitions differ from the PowerPC architecture specification in the case of the instruction access exception.
- The 601 does not implement guarded (cache inhibited) memory.
- Some OEA MMU instructions (such as *tlb sync*) are not implemented on the 601.

The functionality of the PowerPC MMU is complex enough to merit a chapter of its own. Chapter 8, “Memory Management,” deals exclusively with the MMU operation and compares MMU features across the PowerPC family of microprocessors.
The Cache Unit

The study of cache design and implementation can fill an entire book. And, in fact, it does. If you need to know the details of cache design, the references in the bibliography at the end of this book can point you in the right direction.

The 601 contains a 32K eight-way set-associative unified code and data cache. As a unified cache, it caches both instructions and data. Other PowerPC implementations employ separate caches for instructions and data. Having separate caches for instructions and data is characteristic of a Harvard Architecture cache model. This term is commonly used in descriptions of PowerPC implementations other than the 601.

In set-associative cache designs, such as that used in the 601, the lower bits of an address are used in a hashing algorithm to determine the set into which the data will be stored. Because the cache contains a limited number of sets, many addresses will hash to the same set.

The number of ways refers to the number of possible places within the appropriate set into which the specific data could be stored. The 601’s eight-way cache stores data from a specific address in main memory to one of eight (and only eight) specific cache locations.

Finally, the 601’s least recently used (LRU) replacement policy identifies which of the eight ways will receive the new data. In particular, the way that has been used the least recently (and is therefore less likely to be needed) is chosen as the location for the new data.

As shown in Figure 2-4, the 601’s cache provides a 256-bit (8-word) bus to the instruction fetch unit and load/store unit. The instruction unit looks in the cache for the address of the next instruction. If the address is in the cache (a cache hit), the instruction queue is filled with as many instructions as will fit, starting from that address.

The 601 provides for software control of cacheability, write-policy, and memory coherency using the system control registers described in Chapter 4, “The PowerPC Programming Model.”

Memory Unit and System Interface Unit

The memory unit (MU) and system interface unit (SIU) are two separate functional units. Because they both deal with external accesses to the bus (main memory), it makes sense to discuss them together.
The 601's memory unit consists of read/write buffers that the processor uses when accessing external memory. Such accesses can be caused by a load or store cache miss or the need to maintain cache coherency. When the 601 needs to access main (external) memory, the MU prioritizes accesses to the system bus.

The system interface unit (or bus interface unit) handles all accesses to main memory as dictated by the memory unit. Addresses and values from the MU's buffers are placed physically on the system bus by the SIU. The 601's SIU, like all other PowerPC implementations in this book, is based on the Motorola M88110 RISC microprocessor's bus design.

System bus accesses are grouped into burst-read (to fill the instruction queue or cache) and burst-write operations, I/O controller interface operations (to talk to I/O devices), and single-beat operations (for accessing non-cacheable memory areas).

**The PowerPC 603**

In this section, we'll focus on the features of the 603 that differ from the 601. Aspects of the 603 that are similar to the 601 won't be discussed again. But you can see from the block diagram in Figure 2-6 that the 603 differs significantly from the 601. Many of the first PowerPC computer systems will be based on the 603 processor.

**The Instruction Unit**

The 603's instruction unit is similar to the 601's. In fact, the differences that are present are generally transparent to the programmer and user alike. Of course, that shouldn't stop you from wanting to understand the operation of each unit.

**The Instruction Queue**

The 603's instruction queue holds up to six instructions at a time, compared to the 601's eight. As instructions are fetched from the 603's instruction cache, they are sorted into branch and non-branch categories. All branch instructions are fed directly to the branch processing unit; all other instructions are placed in the instruction queue to be dispatched to the floating-point or integer unit.
The differences between the 601 and 603 are significant, as shown in this detailed 603 block diagram.

Another difference between the 601 and 603 instruction queue is that the 603 dispatches instructions from the bottom two entries. You may recall that the 601 could dispatch instructions from the bottom four entries.
The Load/Store Unit

The 601 performs load/store operations with its integer unit (and on occasion its FPU). The 603 has a dedicated load/store unit (LSU). All load/store operations and register-to-register moves are handled here — even for floating-point registers.

Load/store instructions may be executed, and access memory, other than in program order. This allows the 603 to minimize pipeline stalls. To aid program-order memory accesses, there are synchronization instructions that enforce strict ordering.

The Cache Unit

Unlike the 601, the 603 has separate caches for instructions and data. Specifically, the 603’s cache subsystem consists of two 8K two-way set-associative instruction and data caches. The 603’s caches use the same LRU replacement policy as the 601.

The functionality and structure of the cache on any PowerPC implementation is transparent to the programmer and the user. Poor code, however, can defeat the best cache. Understanding the cache’s operation is important to taking advantage of and avoiding the pitfalls of specific cache features. In Chapter 9, “The PowerPC Cache,” we’ll examine the various user- and supervisor-level cache management instructions.

Failure to keep needed code and data in the caches can have a negative effect on performance. When the 603 needs to refill its instruction queue, the instruction cache can provide two instructions per cycle to the instruction queue (assuming a cache hit). Accessing main memory to retrieve program code would be much more costly.

The Completion Unit (Completion Buffer)

There is an important distinction between a completed instruction and an instruction that has finished execution. The results of a completed instruction are used to update the main register file and system registers (also known as the architectural registers). The result of an instruction that has simply finished executing and its effect on other registers is undetermined until the completion unit determines that it is appropriate to complete the instruction.
The 603's *completion unit* (CU) can track up to five instructions from the point of dispatch until execution has finished. When instructions execute out of order, the completion unit must reorder the results and make instruction completion sequential. Upon completion of the instructions, the architectural registers are updated.

**Power Management Capabilities**

Perhaps the most distinguishing feature of the 603 is its power management capability. The four power modes are software-selectable using bits in the MSR and HIDO registers. The four modes are defined as follows:

- **Full-Power Mode**
  This is the default power state of the 603. All execution units run at the processor clock speed, but are clocked only when needed. After the full-power mode is set, no hardware or software intervention is necessary. If dynamic power management mode is disabled, the 603 will stay in full-power mode indefinitely. But if power management is enabled, idle units will be put into low-power mode as appropriate.

- **Doze Mode**
  In this mode, most functional units are disabled; only the timebase/decrementer and bus snooping logic remain enabled. Returning to the full-power state requires an external asynchronous or system management interrupt, a decrementer exception, or a hard/soft system reset. The transition from doze mode to full-power mode takes very few clock cycles.

- **Nap Mode**
  In this mode, the bus snooping logic is disabled; otherwise it is the same as doze mode.

- **Sleep Mode**
  In this mode, all internal functional units are disabled and the 603 is consuming the least possible amount of power while enabled. To return to full-power mode, the processor must be reenabled systematically before assertion of one of the above wake-up signals.

The 603 has a dedicated interrupt and interrupt vector specifically for power management — the *system management interrupt* (SMI). Using the SMI or the decrementer interrupt, the 603 can switch into one of the four
power management modes. However, the 603 can not switch modes without returning to full-power mode first.

Most power management transitions are controlled by software, such as an operating system. In some cases, hardware will put the 603 into a power management mode depending on predetermined conditions such as a time delay using the decrementer interrupt.

**The PowerPC 603e**

The PowerPC 603e implementation is one of the latest PowerPC processors released by Motorola, Inc. and IBM Corporation.

The following set of features distinguishes the 603e from the original PowerPC 603 implementation.

- The instruction and data caches have been increased to 16K each.
- A bit was added to the HIDO register that allows clock configuration to be readable by software.
- Support for single-cycle stores has been added. This will result in a 7%-10% performance improvement across most software.
- The performance of cache-inhibited stores and write-through stores has been improved.
- A new adder was included in the system unit.
- A half-clock bus multiplier was added. This will allow higher internal clock speeds.

**The PowerPC 604**

The PowerPC 604 is a dramatic step up from the 601 and 603 processors. We’ll cover only the characteristics of the 604 that differ significantly from the previous implementations. The addition of new execution units (for a total of six), dynamic branch prediction, and speculative execution give the 604 formidable power. A block diagram of the 604 is shown in Figure 2-7. Although still a 32-bit implementation, the 604 closely resembles the internal architecture of the 620, examined in the next section.
Although it's still a 32-bit microprocessor, the 604 shown in this block diagram shares many characteristics of the 620.
**Instruction Unit**

Like the PowerPC 601, the 604’s instruction queue can hold up to eight instructions. During typical program execution, the 604 can fetch, dispatch, and complete up to four instructions per cycle. Each instruction is dispatched in order to the appropriate execution unit, one instruction per unit.

**Branch Processing Unit**

The 604’s BPU is considerably more complex than previous implementations. Don’t be discouraged as you read the following sections on branch prediction and speculative execution — it’s a complex topic and we’re not going to spend much time on it here. Chapter 7 discusses branch prediction (and its effect on execution) in detail.

**Dynamic Branch Prediction**

The PowerPC 604 is the first implementation to use dynamic branch prediction. Compared to static branch prediction, the dynamic form is considerably more complex. The 604 employs three techniques, depending on the branch instruction’s position in the pipeline.

In the fetch stage, the *branch target address cache* (BTAC) is examined for a match for the fetched address. On the 604, the BTAC is a 64-entry cache of target addresses from previously executed branch instructions. The fetched address is sufficient to predict a branch using the BTAC.

In the decode and dispatch stages, the first branch instruction in the code stream is identified and predicted. For conditional branches that depend on unavailable information, the *branch history table* (BHT) is used to predict the outcome. On the 604, the BHT is a 512-entry cache of prediction states. The prediction states are: strong taken, weak taken, weak not taken, and strong not taken.

When a branch instruction is executed, the BHT is updated with information concerning the outcome of the branch. As information is updated in the BHT, it is also added (or removed) from the BTAC. For example, if a branch is predicted to be taken for the next encounter, it is added to the BTAC; if not, it is removed from the BTAC.
Speculative Execution

Speculative execution is a complex technique that is the silicon equivalent of "covering all bases." More precisely, the PowerPC 604 uses speculative execution in conjunction with dynamic branch prediction to follow the code paths of up to four predicted branches.

While speculatively executing along a predicted branch path, the 604 will continue to predict and execute for up to two subsequent branches. Along the way, the 604 saves the machine state when it encounters a branch that requires further prediction. This way, the 604 can return to a previous state with little overhead when the branch is resolved.

Instructions from a mispredicted path are identified and removed from the pipelines. When the actual code path has been determined, the associated code has likely already been speculatively executed and the rename buffers and shadow registers can be used to update the architectural registers.

We'll take a good look at speculative execution in Chapter 7, "The Sublime Art of Instruction Timing," when we compare the various branch prediction schemes.

Multiple Integer Units

The 604 is the first PowerPC implementation to provide multiple integer units. The 604 has two single-cycle integer units and one multiple-cycle IU.

Each of the single-cycle integer units comprises three subunits: a fast adder/comparator subunit, a logic subunit, and a rotator/shifter subunit. However, only one of the three subunits may be executing at a given time. Together, these three subunits handle all one-cycle integer instructions and register-to-register operations.

The complex integer unit consists of a 32-bit integer multiplier/divider and handles all multiple-cycle integer instructions. It is also responsible for special-purpose register manipulation.

Floating-Point Unit

The 604's floating-point unit is very similar to the FPU found in the 601 and 603. Most importantly, the performance of the 604's FPU has been
increased from the previous implementations. In general, both single- and double-precision operations have a three-cycle latency in the FPU.

**Cache Units**

Like the 603, the 604 has separate caches for instructions and data. The 604’s cache subsystem consists of two 16K, four-way set-associative instruction and data caches. The 604’s caches use the same LRU replacement policy as the 601 and 603.

The caches on the 604 have been specifically designed to facilitate multiprocessor implementation. With a full set of cache control instructions and the complete modified/exclusive/shared/invalid (MESI) protocol, the 604 is well suited for multiprocessing. A complete discussion of cache issues is covered in Chapter 9, “The PowerPC Cache.”

**Power Management Capabilities**

Power management on the 604 is quite simple. The 604 implements the equivalent of the 603’s nap mode — most functional units are disabled and the processor consumes typically less than 0.4 watts. Only the timebase/decrementer and external interrupt logic remain enabled. The 604 can enter nap mode via software. But returning to the full-power state requires an external asynchronous or system management interrupt, a decrementer exception, or a hard/soft system reset. As with the 603, the transition from doze mode to full-power mode takes very few clock cycles.

**The PowerPC 620**

Announced in October 1994, the 620 is the first 64-bit PowerPC implementation. Along with that distinction come many differences when compared with the 32-bit 60x implementations. Figure 2-8 shows a block diagram of the 620.

Representing the high end of PowerPC computing, the 620 is designed for servers and fast workstations. Target SPEC ratings are 225 SPECInt92 and 300 SPECfp92 with the chip running at 133MHz. Like the 604, the
Figure 2-8
The 620 is the first 64-bit PowerPC implementation.
620 processor is designed to accommodate multiprocessor configurations by allowing bus-snooping-based cache coherency and four-state MESI cache coherency protocol.

Even though the 620 has 38 additional 64-bit instructions, it is line-for-line software compatible with all 32-bit implementations. In fact, there is no performance penalty when running 32-bit code. By setting the proper bit, the 620 becomes a 32-bit processor.

The power management capabilities for the 620 are identical to the 604. A software-accessible nap mode is fully supported.

**Instruction Unit**

Because the 620 and 604 contain a similar compliment of execution units, the two processors behave similarly in terms of instruction dispatch. The 620 can dispatch up to four instructions per cycle.

Each execution unit contains up to four reservation stations to queue instructions when the execution unit is busy with another instruction. Like the other implementations, the 620 supports in-order dispatch and out-of-order execution.

The 620’s branch history table has been expanded to 2048 entries and the branch target address cache has been bumped up to 256 entries. With this capability, the 620 can speculatively execute up to four unresolved branches.

**Integer and Floating-Point Units**

As you can see in Figure 2-8, the 620 boasts three integer units, a floating-point unit, a load/store unit, and several units that constitute the instruction unit block. The 620 resembles but far outperforms the 604. For example, an \texttt{fdiv} instruction on the 604 takes 32 cycles to complete and only 18 cycles on the 620.

**Cache Unit**

Doubling the size of the 604’s caches, the 620 features 32K, eight-way set-associative data and instruction caches. Additionally, the 620 is the first PowerPC processor to feature built-in support for a level-two cache.
System Interface Unit

The 620’s SIU is similar in design to the 604. A key difference between the two implementations is that the 620 has a 40-bit address bus that is capable of addressing 1 terabyte (1024 gigabytes) of physical memory. Other enhancements to the SIU allow the 620 to move data to and from the processor more efficiently than in previous implementations.

SUMMARY

In perspective, everything fits together pretty nicely; the i486’s structure is comparable to that of the 601. And perhaps the architectural differences are fewer and smaller than you had envisioned.

In the next chapter, we’ll look at the major differences between popular computer architectures of today — endianness. Coming from an x86 background, you’ll have to get used to seeing things in a new light. You should be getting used to that already.
Chapter 3

OF EGGS AND ENDIANS

"All true believers shall open their eggs at the convenient end."
— Jonathan Swift, from Gulliver’s Travels

A BRIEF HISTORY OF ENDIANNESS

In the good old days of 8-bit processors, we had only one addressing scheme. Because 8-bit microprocessors access only 8-bit operands, reading and writing any operand to memory meant generating an address for 1 byte at a time. With the advent of processors with data buses that were 16 bits wide (or wider), programmers have had to contend with two distinctly different memory addressing schemes.

A 16-bit value is stored as two individual bytes. The processor can thus store a 16-bit value with its most significant 8 bits at a specific byte address and its least significant 8 bits at the next higher byte address. Or, the processor can store the least significant 8 bits at a specific byte address and the most significant 8 bits at the next higher byte address. In other words, the two choices are byte-reversed images of each other.
Origin of the Terms

In 1981, Danny Cohen wrote the paper that gave name to both sides of this grand point of computer architecture and memory addressing controversy. He likened the memory addressing dispute to an episode described in Johnathan Swift's *Gulliver's Travels*, where rival factions fought over which was the proper end of an egg to open — the little end or the big end. More than 200 years after Swift, Cohen pointed out a parallel within the computer industry that gave new meaning to Swift’s Endians. An excerpt from Swift’s novel gives us a feeling for the intensity of the controversy:

“It is computed, that eleven thousand persons have, at several times, suffered death, rather than submit to break their Eggs at the smaller End. Many hundred large volumes have been published upon this controversy...”

The *big endians*, those who favor egg-opening at the big end, typically assigned the most significant bit (MSb) to the lowest memory address and the least significant bit (LSb) to the highest. In this manner, the significance of a bit decreases with increasing memory address and a word in memory is addressed with respect to the address of its MSb. Conversely, the *little endians* assign the LSb to the lowest memory address; they start at the little end.

Independent of endianness, the high-order byte always corresponds to the leftmost two digits in a hexadecimal number when it is printed in a form that humans use (not in computer memory). Furthermore the leftmost bit of the high-order byte corresponds to the MSb. For example, in the hexadecimal number 0x87654321, the high-order byte (also known as the most significant byte or MSB) is 0x87 regardless of the method used to store the number in memory. The MSb is the high-order bit corresponding to the high-order digit in byte 0x87. The low-order (least significant) byte and bit correspond to the rightmost byte: 0x21 in the preceding example.

At the time that the terms *big endian* and *little endian* were introduced to the computer industry, there wasn’t a Mac or PC to be found. The first two computers to be recognized as endians were the IBM 370 and PDP-11; the PDP-11 was the first big endian and the IBM 370 was the first little endian. The two configurations shown in Figure 3-1 illustrate the difference between the big endian and little endian storage conventions when storing the 16-bit number 0x1234 at an arbitrary memory address.

Motorola 680x0 addressing is big endian. The Intel x86 addressing scheme is perhaps the most (in)famous little endian. The release and subsequent popularity of Apple’s Macintosh and IBM’s PC enshrined Motorola and Intel as icons of big and little endian schemes, respectively. Recognize
that each endian mode is simply a memory addressing methodology and nothing more. Of course, if you’re used to working with one scheme, it’s disturbing to see bytes in the wrong order; your world has been turned upside down — or at least left to right.

(a) Big endians store the first 8 bits (MSb) first, and the second 8 bits (LSb) second.

```
    MSb  LSb
1  2  3  4
15  0 bit
```

(b) Little endians store the second 8 bits (LSb) first, and the first 8 bits (MSb) second.

```
    LSb  MSb
3  4  1  2
15  0 bit
```

Note that (a) and (b) are byte reversed images of each other.

Figure 3-1
Storing a multibyte operand (0x1234) requires that you choose between two different byte-ordering schemes.

Applied consistently, either endian scheme would work equally well in any particular architecture. The scheme that each of us thinks is better often depends on what we were “brought up with.” Combine this prejudice with the fact that these schemes are fundamentally incompatible with each other (a microprocessor must consistently use only one) and you can see how the big versus little endian controversy can take on the overtones of a holy war!

The endianness of a processor becomes significant when accessing a specific byte of a multibyte operand — bits 8 through 15 of a 32-bit value, for example. Even if you know the base address that the processor uses to access the 32-bit value, without knowing the processor’s storage convention you won’t know where to look for a specific byte.
Cohen argued that to avoid memory addressing anarchy we must all adopt a single endian scheme, whether big or little — it didn’t matter to him. Unfortunately for programmers, no single addressing scheme has become dominant: Intel’s microprocessors use the little endian approach while Motorola’s use the big endian scheme.

But Cohen’s plea for peace may have been heard by the designers of the PowerPC family and other RISC microprocessors. The PowerPC architects opted for a compromise. Each PowerPC processor can be switched into either big or little endian addressing modes. This is an important feature for processors that will host multiple operating systems which may differ in their fundamental endian assumptions.

**The End-Side Story**

When examining the differences between the two endian schemes, it helps to understand what *doesn’t* define the endianness of a computer system. The big and little endian schemes are not defined by how the individual bits in a word are labeled. In Figure 3-2, two big endian words and one little endian word are shown with their individual bits labeled. In addition, the most significant bit (MSb) and least significant bit (LSb) are also indicated.

Notice that the two big endian processor families (the Motorola 68K and the PowerPC processors) use different bit labeling conventions. The 68K calls its MSb bit 31, the PowerPC processors call its MSb bit 0. In fact, the big endian 68K labels its bits the same as the little endian Intel x86.

The point to remember here is that *bit labels don’t define endianness*. The big endian scheme uses the MSb to establish the address of a word in
memory — regardless of how that bit is numbered — while the little endian scheme uses the LSb for the same purpose. Given those two points, you can now see that, regardless of bit numbering, the 68K and PowerPC addressing scheme is the same and that the Intel method differs from both. Let’s look at some specific examples.

![Diagram of memory addressing schemes](Figure 3.2)

**Figure 3.2**

Bit labels within a word are simply a manufacturer’s convention and are independent of endianness.

### Endianness and Memory

Assume that you’ve used a debugger to enter the values shown in Table 3-1 into both an x86 PC and a PowerPC system. You store the 32-bit word 0x00facade at offset 0, the 16-bit short value 0x1357 at offset 7, and the byte string ‘p’, ‘o’, ‘w’, ‘e’, ‘r’ at offset 0x0b. Furthermore, assume that the PowerPC system is running in big endian mode (its default, power-on state).

#### Table 3-1

Data Entered into Both Computers

<table>
<thead>
<tr>
<th>Offset</th>
<th>Data</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x00facade</td>
<td>32-bit word</td>
</tr>
<tr>
<td>0x07</td>
<td>0x1357</td>
<td>16-bit short</td>
</tr>
<tr>
<td>0x0b</td>
<td>‘p’, ‘o’, ‘w’, ‘e’, ‘r’</td>
<td>Five 8-bit bytes</td>
</tr>
</tbody>
</table>
Having put the same values into the same memory locations on each machine, we’ll dump that memory and examine the contents. Our debugger would display the information shown in Figure 3-3.

### Figure 3-3
A view of data stored in big and little endian memory

At first glance, the individual bytes that make up the word and short data elements look byte-reversed between the two endian schemes. The string, however, looks the same. Does the display fit with our definition of endianness? In fact, it does. Let’s make sure we understand why.

Big endians address words in memory with respect to the MSb, stored in the low-address byte of the word. Thus the value 0x00facade is properly stored in bytes 0–3 under the big endian model. Looking at the value stored in little endian memory, however, it’s not immediately clear that the bytes are in the proper order. Little endians address words with respect to the LSb, stored in the lowest byte address. As a result, the individual bytes of a multibyte value are displayed with the most significant bytes to the right. We have to read the word from right (MSb) to left (LSb) starting at offset 3. (This is often referred to as “back-words” ordering.) Thus, 0x00facade is properly positioned at offset 0 in both memory schemes.

### Differing Little Endian Display Methods

Some documentation (such as the PowerPC 601 User’s Manual) depict dumps of little endian memory with the zero offset on the right-hand side of the row to correspond to the position of bit zero. For our purposes, however, we’ll put the address zero offset for displays of both endian types on the left of a 16-byte row of memory to make direct comparisons easier.
Using the same reasoning, we see that 0x1357 is properly positioned in both views of memory. We have to read only two reversed bytes for the little endian case since the data size was 16 bits. Knowing the word size (16, 32, or 64 bits) is a vitally important aspect to reading values in memory under either scheme — for both you and the microprocessor.

The text string “power” is stored as five individual bytes. When storing a single byte, the issue of multibyte ordering doesn’t apply. Thus, both endian schemes store the byte string in the same fashion.

**Byte Addressing within Multibyte Operands**

One of the most confusing aspects regarding the effects of endianness is determining the position of individual bytes within multibyte operands. The byte position depends on the size of the operand. Thus, hard-coded assumptions can cause confusing results.

Figure 3-4 illustrates the relationship between byte position and operand size. The examples assume that each type of processor (big and little endian) is responsible for the storage of the specified items in memory using its native endianness. (In other words, the little endian example shown is not an emulation performed by a big endian processor.)

For each of the operand sizes shown in Figure 3-4, the big endian memory dump is byte-reversed relative to the little endian dump. But the important thing to note is that the Oxab byte (the MSB) in each first operand appears at address Ox00 in every big endian dump — regardless of the operand’s size. In contrast, the byte address of the Oxab byte within each first operand in the little endian dump depends explicitly on the operand’s size. This distinction becomes significant for programmers when attempting to address individual bytes within a word.

For example, to test if a 32-bit value stored in memory is negative, you can AND the most significant byte with 0x80. A non-zero result means that the operand is negative. But determining which byte to test depends on whether the processor is big or little endian.

On a little endian (x86) platform, you must test the byte located 3 bytes higher than the base address of the 32-bit word. As shown in Figure 3-4(b), the byte value Oxab appears at little endian offset Ox03. However, on a big endian (PowerPC) platform, testing the same byte means examining memory at offset Ox00.
(a) Two short (16-bit) operands are stored at offset 0 (0x abcd) and offset 8 (0x 1234).

<table>
<thead>
<tr>
<th>Big endian storage of short in memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>ab cd</td>
</tr>
<tr>
<td>Byte</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Little endian storage of short in memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>cd ab</td>
</tr>
<tr>
<td>Byte</td>
</tr>
</tbody>
</table>

(b) Two word (32-bit) operands are stored at offset 0 (0x abcdef00) and offset 8 (0x 12345678).

<table>
<thead>
<tr>
<th>Big endian storage of word in memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>ab cd ef 00</td>
</tr>
<tr>
<td>Byte</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Little endian storage of word in memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 ef cd ab</td>
</tr>
<tr>
<td>Byte</td>
</tr>
</tbody>
</table>

(c) Two dword (64-bit) operands are stored at offset 0 (0x abcdef0011223344) and offset 8 (0x 12345678aabbccdd).

<table>
<thead>
<tr>
<th>Big endian storage of dword in memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>ab cd ef 00 11 22 33 44 12 34 56 78 aa bb cc dd</td>
</tr>
<tr>
<td>Byte</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Little endian storage of dword in memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>44 33 22 11 00 ef cd ab cd cc bb aa 78 56 34 12</td>
</tr>
<tr>
<td>Byte</td>
</tr>
</tbody>
</table>

**Figure 3-4**
Determining the position of individual bytes within a word requires knowing the endianness of the processor.
It's important to note that if we performed this comparison on the entire 32-bit word (as opposed to a single byte within the word), the distinction between big and little endian disappears. Only when testing a particular byte within a larger memory unit do you need to consider endianness.

The bottom line is that making endian-dependent assumptions concerning the position of bytes (and bits) within words will result in code that is difficult to port between endian schemes.

**A Closer Look**

We've seen how identical values are stored in memory under each of the two endian schemes. Now let's look at the same set of data but interpret it using each endian scheme. Figure 3-5 shows 16 bytes of memory. Table 3-2 summarizes various values that can be read from each using both big and little endian addressing methods.

<table>
<thead>
<tr>
<th>Byte</th>
<th>00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>00 fa ca de 00 00 13 57 00 00 70 6f 77 65 72 'p' o' w' e' r'</td>
</tr>
</tbody>
</table>

**Figure 3-5**

Depending on the endian scheme of the processor, the same data in memory can be interpreted differently.

Understanding how each of the values in Table 3-2 are obtained is the larger part of understanding the differences between big and little endian memory addressing. If you port an application from an x86 platform to a PowerPC platform, understanding the differences in accessing memory using either endian scheme could save you time and help you avoid frustration.

**Table 3-2**

<table>
<thead>
<tr>
<th>Little Endian Memory Decoding</th>
<th>Big Endian Memory Decoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Size</strong></td>
<td><strong>Offset</strong></td>
</tr>
<tr>
<td>byte</td>
<td>0x02</td>
</tr>
<tr>
<td>short</td>
<td>0x02</td>
</tr>
<tr>
<td>short</td>
<td>0x0c</td>
</tr>
<tr>
<td>word</td>
<td>0x08</td>
</tr>
</tbody>
</table>
**PowerPC Endianness**

Each PowerPC processor is able to use either endian scheme when addressing memory. The power-on default state of all PowerPC implementations is to use the big endian memory addressing scheme. But each processor can be switched into little endian mode using the proper instruction sequence. Some of the unconventional aspects of the PowerPC family’s little endian mode are:

- Internal PowerPC registers always contain data in big endian format.
- The physical address used to access data in external memory depends on the size (number of bytes) of the data.
- Even in little endian mode, the PowerPC microprocessors execute big endian code. The instruction addresses are modified to accommodate little endian addressing. This concept is explained in the section “Endian Conversion” later in this chapter.
- Alignment requirements for accessing memory are more strict when using little endian mode.

**PowerPC Support for Bi-Endian Memory**

The values derived for Table 3-2 show the complications that arise when the same memory is interpreted using both the big and little endian schemes. There are situations, however, when both endian schemes must be used to access memory on PowerPC systems. For example, an application program that has established one endian scheme can be forcibly switched into the other scheme during an exception. If the exception handler needs to access some portion of the application’s data, a memory interpretation problem arises. Furthermore, consider the situation where you receive a packet of big endian data from an external source (a network, for example) while running in little endian mode.

Fortunately, if a PowerPC processor is operating in one endian mode and must access memory stored in the other mode, a mode switch is not necessary. There are several PowerPC instructions that allow memory to be interpreted using the other scheme — the “other scheme” being the endian mode that the processor is not currently using.

The need to manipulate other-endian data isn’t unique to PowerPC processors. The i486 bswap (byte swap) instruction is included specifically to enable the processor to operate on data passed to it from big endian comput-
ers. The i486's `bswap` instruction operates only on 32-bit operands. The PowerPC instructions that facilitate other-endian data manipulation are slightly more inclusive, allowing translation of both 16- and 32-bit quantities.

Table 3-3 lists the PowerPC instructions that access memory using the other scheme. When operating in big endian mode, these instructions load and store data using the little endian scheme; when operating in little endian mode, these instructions load and store data using the big endian scheme. Where bit-labels are used in Table 3-3, they refer to the bit labels of a PowerPC 32-bit word as shown in Figure 3-2(b). Further examples of using these instructions are found in Appendix A, “PowerPC Instruction Set Reference.”

**Table 3-3**

<table>
<thead>
<tr>
<th>Name</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Half-Word</td>
<td><code>lhbr x</code> rD,rA,rB</td>
<td>The high-order byte of the half-word in memory is loaded into the low-order byte of rD. The next higher-order byte of the half-word is loaded into the next lower-order byte of rD.</td>
</tr>
<tr>
<td></td>
<td><code>lwbr x</code> rD,rA,rB</td>
<td>MemoryWord is pointed to by the sum (rA+rB). The bytes of MemoryWord are loaded into the bytes of rD in the following manner:</td>
</tr>
<tr>
<td></td>
<td><code>sthbr x</code> rS,rA,rB</td>
<td>The low-order byte of rS is stored into the high-order byte of the half-word in memory. The next lower-order byte of rS is stored in the next higher-order byte of the half-word in memory.</td>
</tr>
<tr>
<td></td>
<td><code>stwbr x</code> rS,rA,rB</td>
<td>This operation is analogous to the <code>lwbr x</code> instruction. MemoryWord is pointed to by the sum (rA+rB). The store performs the following operation:</td>
</tr>
</tbody>
</table>
Don’t worry if the instruction format in Table 3-3 seems foreign now. These instructions will make more sense after you read the discussion of PowerPC registers in Chapter 4, “The PowerPC Programming Model,” and operand conventions and addressing modes in Chapter 5, “Addressing Modes and Operand Conventions.”

**Switching Endian Modes**

By default, PowerPC processors are big endian at power-on or after a hard reset. And only after such a reset can the little endian mode be selected on a PowerPC processor. The method used to select either endian mode on the 601 differs from the other PowerPC implementations. The PowerPC architecture defines two bits in the *machine state register* (MSR) to control the endianness of user- and supervisor-level software independently.

Figure 3-6 shows the bits that affect endianness on the 603, 604, and 620 processors. The *exception little endian mode* (ELE) and the *little endian* (LE) bits switch the endianness. Because the 601 doesn’t implement the ELE or LE bits in the MSR, the endian switching code sequence differs from other PowerPC implementations. The MSR is discussed at length in Chapter 4, “The PowerPC Programming Model.”

---

**Figure 3-6**

Two bits in the MSR determine endianness for the 603, 604, and 620 processors.
IBM/Motorola Terminology Conventions

Recall that IBM and Motorola terminology typically differs when it comes to "exceptions" and "interrupts." Motorola says "exception" where IBM says "interrupt." Thus, Motorola's ELE (exception little endian mode) bit becomes IBM's ILE (interrupt little endian mode) bit. This type of terminology disagreement will hopefully converge in future revisions of IBM/Motorola documentation. For further discussion of the terminology conventions used in this book, refer back to Chapter 1, "The PowerPC Transition."

As mentioned in Chapter 1, "The PowerPC Transition," the 601 does things in a slightly different manner than the other PowerPC implementations. The 601 uses the little mode (LM) bit in the HID0 (hardware implementation dependent 0) register, shown in Figure 3-7, to switch between big and little endian modes. However, switching endian modes on the 601 requires some precautions beyond those necessary for the other processors. Before switching modes on the 601, you'll also have to do the following:

- Flush all caches.
- Disable decrementer and external exceptions.
- Be sure the mode-switching code sequence doesn't cross a protection boundary.

![Figure 3-7](image)

Switching endian modes on the 601 requires setting the appropriate bit in the HID0 register.
Changing the endian mode is a privileged operation that is limited to low-level system software. Code that runs at the user level, such as application software, can’t modify the bits necessary to change the endianness of the system. An attempt to do so results in a protection violation exception. When debugging or writing supervisor-level software, it’s helpful to understand the processes of switching the endianness of a PowerPC system. The following code sequences show how to change the endianness of each PowerPC processor.

The code sequences shown in Listings 3-1, 3-2, and 3-3 show how to put the 601, 603 and 604, and 620 processors into little endian mode, respectively. To switch back to big endian mode, a similar operation must be performed with the MSR bits set accordingly. In particular, the same synchronization must be performed before clearing the MSR[ELE] bit (or MSR[LM] on the 601). Note that the rfi (return from interrupt) instruction does not modify the MSR[ELE] bit. If the PowerPC exception handlers are to be executed in little endian mode, the MSR[ELE] bit must be set independently of the rfi instruction.

**Listing 3-1**

Setting the endianness on the PowerPC 601

```asm
; Setting endian mode on the 601
; this example assumes that: MSR[EE] is zero (0).
; caches have been flushed

mfspr r3, HID0 ; move from special purpose register:
; put hardware implementation dependent
; (HID0) register contents into GPR r3

ori r3, r3, 0x8 ; OR-immediate:
; set HID0[LM] (little endian mode) bit

sync ; three sync instructions must precede
sync ; the move to special purpose register
sync ; instruction

mtspr HID0, r3 ; move to special purpose register:
; put the value of r3 into
; the HID0 register

sync ; three syncs must follow the
sync ; mtspr or mtmsr
```

```
```
Listing 3-2
Setting the endianness on the PowerPC 603 and 604

```assembly
sync          ; complete all prior instructions
mfmsr r3     ; move from special purpose register:
              ; get the current contents of the MSR in r3
ori r3, r3, 1 ; set LE=1 (little endian)
mtspr SRR1, r3 ; move to the system save and restore
              ; register 1 (SRR1)

lwz r3, targetAddr(rA) ; get target address for little
                        ; endian code

mtspr SRR0, r3 ; move to the system save and restore
                ; register 0 (SRR0)

rfi ; return from interrupt: change modes
    ; and branch to target code
```

Listing 3-3
Setting the endianness on the PowerPC 620

```assembly
sync          ; complete all prior instructions
mfmsr r3     ; move from special purpose register:
              ; get MSR value in register r3
ori r3, r4, 0x00010000 ; set ELE bit
andi r3, r3, 0xffffffff ; and immediate:
                        ; reset the MSR[RI] bit to indicate
                        ; non-recoverable interrupt

mtmsr r3     ; load MSR with new value
mtspr SRR1, r3 ; save a copy of the new MSR value
mtspr SRR0, targetAddr ; little endian target address
rfi ; return from interrupt into little
    ; endian mode

...            ; we're in little endian mode when
    ; we get here
```
Endian Conversion

Our examination of endianness wouldn't be complete without discussing how the processors convert between the two addressing modes internally. Although the typical user wouldn't be aware that a conversion was taking place (or even which endian mode was active), programmers should understand the mechanisms involved. We'll examine why conversion is necessary and how it is accomplished within the processor.

If memory operands were indivisible, the concept of byte ordering during operand access would be unnecessary. Because memory operands can also be viewed as a series of smaller addressable units, however, byte ordering becomes a key issue in the context of endianness.

The PowerPC architecture implements bi-endianness not by changing the order in which bytes are stored in memory, but by adjusting the byte address of a memory operand within an aligned doubleword as necessary before accessing memory. In other words, little endian storage on PowerPC processors is an illusion. The terms munging or swizzeling refer to the mechanism used to convert from big endian mode to little endian mode in the address path. For load and store operations, the effective address of an operand is first computed regardless of the endian mode in effect. The resultant address is then modified as shown in Figure 3-8, according to the size of the memory operand. The effective address modification makes it appear to the processor that the aligned memory operand is stored as little endian — in fact, they are stored as big endian, but in different bytes within a doubleword than they would be in big endian mode.

To illustrate address munging, consider the section of little endian memory shown previously in the bottom half of Figure 3-4(c). Let's assume that we interpret the first doubleword of memory shown (bytes 0 through 7) as containing two words: the value 0x12345678 stored at offset 0x00 and the value 0xabcdef00 stored at offset 0x04. The top half of Figure 3-4(c) accurately represents how that data would be stored in the same doubleword on PowerPC processors when operating in little endian mode.

Now we assume that our little endian program wants to retrieve the word value 0x12345678. From its point of view, it generates the effective address 0x00. The PowerPC processor then XORs the lower three bits of this effective address with 0b100 as shown here:

\[
\begin{align*}
\text{0x00 (} &= 1001b\text{) original address} \\
\text{XOR } \text{0x04 (} &= 100b \text{ for 32-bit word accesses) mung value} \\
\hline
\text{0x04 (} &= 1101b\text{) adjusted address}
\end{align*}
\]
The PowerPC processor converts addresses by munging when accessing data in little endian mode.

The processor then performs a big endian access to the word at effective address 0x04, retrieving the value 0x11223344. Thus the memory appears to the program to be in little endian ordering.

Performing the XOR operation on the lower 3 bits of the address makes sense only if the memory operand is aligned on an address equal to a multiple of its size. When executing in little endian mode, the processor may generate an exception when load and store instructions are issued with a misaligned effective address—regardless of whether such access could be performed without exception in big endian mode.

To see why munging misaligned addresses doesn’t work, consider again the section of little endian memory shown in the bottom half of Figure 3-4(c). Let’s assume that we interpret bytes 1 and 2 as a misaligned halfword of the value 0x2233 at address 0x01. Munging this address with
the 0b110 value prescribed for half-word address adjustment results in an address of 0x07. But the big endian misaligned half-word stored at address 0x07 has the value 0x4412.

When the processor is executing in little endian mode, instructions (which, presumably, have been encoded in little endian mode) are fetched as if they were in big endian order. This is not a problem, however, as the PowerPC processors swap a dword (two 32-bit words) worth of instructions at a time, before passing them on to the instruction queue. If this conversion was not applied, the instructions would execute out of program order and your software would tend not to do what you had envisioned.

**SUMMARY**

Understanding both big and little endian addressing methods means that you understand the basic differences between the x86 and PowerPC architecture’s native memory schemes. Chapter 5, “Addressing Modes and Operand Conventions,” covers additional aspects of PowerPC memory addressing and effective address calculation.

In the next chapter, we’ll investigate the PowerPC programming model. Each PowerPC implementation’s programming model has hardware-specific characteristics that make it the least constant aspect among processor models. With a firm grasp on how the programming model changes between PowerPC implementations, you’ll be armed with the ability to create processor implementation-independent code — and that code will take full advantage of each implementation’s features.
"It was an edifice with numberless winding passages and turnings opening into one another, and seeming to have neither beginning nor end."
—Thomas Bulfinch, from Age of Fable

The term *programming model* refers to the set of architectural features that are visible to the programmer. Registers, instruction format, memory organization, addressing modes, and exceptions are all components of a basic programming model. And the PowerPC family's basic programming model is constant across each PowerPC processor implementation. However, within the bounds of the PowerPC architecture, programming models may vary slightly between implementations. In this chapter, we'll examine both the basic programming model and each processor's implementation-specific features, contrasting them to the Intel x86 when appropriate.

**Privilege Levels**

The x86 ring protection scheme consists of four privilege levels — or rings — that range from 3 to 0, increasing in privilege as the ring number decreases. Application programs run in ring 3, the least privileged level, and allow limited access to system...
resources. At the other end of the x86 privilege-level spectrum comes ring 0, in which operating system kernels and device drivers operate. The x86 privilege levels that fall between 3 and 0 vary in terms of usage depending on operating system implementation.

As we proceed through this chapter, the terms supervisor mode and user mode crop up frequently. These terms refer to the current privilege level under which the PowerPC processor is operating. On PowerPC implementations, the supervisor level corresponds to ring 0, and the user level corresponds to rings 1 through 3.

**Registers**

All x86 assembly language programmers are well aware of the limited set of registers available on x86 machines; we've all been in situations where having just one more register would have made the difference between absolutely amazing code and simply functional code. So, we rewrite our code (often a good idea anyway) and fit everything into the registers that are available.

We'll examine the PowerPC register set by separating it into three categories based on where the register is defined within the PowerPC architecture and privilege-level access restrictions. The three register categories are as follows:

- The user instruction set architecture (UISA) and virtual environment architecture (VEA) describe the PowerPC registers that are user-level programs, such as applications. The registers defined by the UISA/VEA are present in the PowerPC implementations discussed in this book. However, as described previously in Chapter 2, the PowerPC architecture affords flexibility within a processor’s implementation of the VEA and OEA register set.

- The operating environment architecture (OEA) defines the supervisor-level register set. These registers are accessible by the operating systems or other privileged software. The transition from UISA (or VEA) to the privilege level of the OEA is accomplished via the PowerPC exception mechanism, as discussed in Chapter 10, "Exceptions and Interrupts.”

- Additionally, a PowerPC microprocessor can have registers that are not defined by the architecture. The 601, 603, 604, and 620 PowerPC processors all have implementation-specific registers. The documentation provided for each implementation describes unique features of the
processor, including specialized registers. Because these registers are typically available only to supervisor-level software, we'll discuss them in conjunction with the OEA-defined register set.

References to the UISA, VEA, and OEA architectures, and the features defined within each, appear frequently in PowerPC documentation. From a programmer’s perspective, separating the PowerPC architecture into the UISA, VEA, and OEA illustrates the aspects of the architecture that are intended for use by software at a particular privilege level.

While reading the sections that follow, bear in mind that the names of registers as they appear in source code are simply that — names. These external representations of the processor’s registers are what we use when referencing registers in our programs; the internal representation is what is actually stored in the opcode.

Both the i486 and PowerPC processors have an internal representation for each register. Even when using assembly language, the internal representation is transparent to the programmer. For example, the following line of PowerPC assembly language contains a reference to two PowerPC registers: the count register (CTR) and general-purpose register r3.

```
mfspr r3, CTR ; load GPR r3 with the value of the CTR
```

We specify the two registers by their external names: r3 and CTR; the assembler will generate an opcode that references both registers by their internal names (GPR3 and SPR9) which the processor understands. Fortunately, we’ll only have to use the external register names.

The register names used on the Intel x86 (such as EAX and EBX) are well known. However, the PowerPC’s register names are less familiar to us at this point. In each of the following sections that describes a PowerPC register (or group of registers), we’ll use source code fragments to show how each register would be named in a code sequence. Soon, the PowerPC register set will be as familiar and accessible as those of the x86 family. But first, a quick review.

### i486 REGISTER SET

The 32-bit Intel i486 register set, shown in Figure 4-1, uses the extended form (EAX, EBX, and so on) for each register name. The 32-bit register set
The i486 programming model can be separated into supervisor- and user-level registers.

is available on i386 and later processors. As mentioned earlier, for purposes of comparison with PowerPC processors, we will assume that our i486 is running in protected mode.

Unlike PowerPC processors, the x86 processor does not have a general-purpose register architecture; many tasks on the x86 require specific registers. For example, EDX and EAX must be used to output byte and word
values to ports, ECX is used as a counter in loops, and ESI and EDI are used for automatic indexing. Operating in protected mode removes some of the restrictions, but we're still faced with limitations.

The i486's register set can be broken into two categories: supervisor-level registers and user-level registers. When running in a lesser privileged ring (anything other than ring 0), the supervisor-level registers are not available. When running at ring 0, both the supervisor- and user-level register sets are accessible.

If you're accustomed to programming in the real-mode DOS environment of x86 processors, you may not be familiar with the supervisor-level registers which are commonly used when running the processor in protected mode. The following lists summarize the i486's supervisor- and user-level registers and their purposes.

### Supervisor-Level Registers

The EFLAGS register indicates the state of the processor. Additionally, EFLAGS controls the operation of features such as single-stepping instructions and the privilege level required to perform I/O operations. On PowerPC processors, the same function is provided by the machine state register (MSR) and condition register (CR).

The global descriptor table register (GDTR) and interrupt descriptor table register (IDTR) memory management registers of the i486 configure the processor's segmentation, paging, and interrupt mechanisms. On PowerPC processors, the equivalent function is provided by the segment registers, block address translation (BAT) registers, and the MSR.

The i486 control registers (CR0–CR3) enable system features such as caching and paging, define paging control structures, and report fault addresses. PowerPC processors have dedicated registers to enable system features (MSR), define address translation mechanisms (BAT and segment registers), and determine the cause of exceptions (SRR0 and SRR1).

The i486 debug registers (DR0–DR7) set and control breakpoints. The debugging facilities on PowerPC processors vary on each implementation. In general, PowerPC processors will have both instruction and data breakpoint registers.

Finally, the test registers (TR3–TR7) of the i486 are used primarily after power-on reset to verify the internal operation of processor components such as the cache and translation lookaside buffer. There are no explicit test
registers for PowerPC processors; however, any of the general-purpose registers could be used in a similar capacity.

**User-Level Registers**

The general registers on the i486 (such as EAX and EBX) are used as instruction operands for both integer and floating-point operations. The PowerPC’s 32 general-purpose registers (GPRs) and 32 floating-point registers (FPRs) are used in an analogous manner. Note that the PowerPC architecture does not define a register that is equivalent to the 486’s EIP (instruction pointer) register. The current instruction address (CIA) is tracked internally to the processor and is not available to software. The EFLAGS register determines the state of the processor; when running at the user privilege level, the EFLAGS register contains bits that are set to indicate the outcome of comparisons and other bit-level operations. PowerPC processors have a condition register (CR) which is used in a similar manner to the user-level EFLAGS register.

The i486’s segment registers separate regions of memory (such as code and data) and are a key component of memory protection. PowerPC processors use their own segmentation mechanism, BAT registers, and paging to provide roughly the same functions. An important point to remember for programmers who are familiar with the 16-bit x86 environment (and associated segmentation headaches) is that the PowerPC does not implement such segmentation and is more “flat-model”-oriented.

**POWERPC UISA AND VEA REGISTER SET**

The registers defined by the PowerPC UISA and VEA are common to all PowerPC implementations discussed in this book. On 64-bit PowerPC processors, the machine state, general-purpose, link, and count registers are expanded from 32 to 64 bits. When a register changes from 32 bits to 64 bits due to implementation size, bit 31 (in 32-bit mode) always corresponds to bit 63 (in 64-bit mode). That is to say, 32-bit registers are right-justified within a 64-bit register. In the register figures that follow, some registers are defined with a special-purpose register (SPR) or time-base register (TBR) number outside
the bit-field definition. These numbers are generated by assemblers and used by the processor in the instruction encoding; they are part of the internal register representation mentioned previously. For example, the link register is SPR8; thus the value 8 appears somewhere with an instruction opcode that refers to the link register. This information is useful when encoding or decoding instructions by hand. In general, assemblers, debuggers, and dissemblers handle the necessary translation — but that should never keep us from understanding what’s going on behind the scenes.

Software running at supervisor level has access to the entire register set of the processor, including the UISA/VEA registers. Supervisor-level registers are defined by the OEA. We’ll examine them in the following section.

The PowerPC UISA/VEA register set, shown in Figure 4-2, comprises seven different registers and register categories.

**General-Purpose Registers**

The PowerPC’s 32 _general-purpose registers_ (GPRs), shown in Figure 4-2, are used by the integer unit to perform both logical and arithmetic operations. Additionally, the PowerPC 603 and 604 processors implement a load/store execution unit that uses GPRs during memory access operations. GPR width corresponds to implementation width; 32-bit PowerPC processors have 32-bit GPRs and 64-bit implementations have 64-bit GPRs.

All PowerPC integer instructions use GPRs for the source and destination values of the operation. GPRs are named in software sequences simply by appending the GPR number to the letter r. For example, the following PowerPC instruction uses three GPRs; r3 is the destination register and r4 and r5 are the source registers.

```
add r3, r4, r5 ; r3=r4+r5 where r3,r4,r5 are integers
```

**Floating-Point Registers**

The PowerPC’s 32 _floating-point registers_ (FPRs) are used exclusively by the floating-point unit to perform floating-point operations. Each FPR is 64 bits wide, independent of processor implementation, as shown in Figure 4-2.
### User Instruction Set Architecture Registers

#### General-Purpose Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPR0</td>
<td>0</td>
<td>31/63</td>
</tr>
<tr>
<td>GPR31</td>
<td>0</td>
<td>31/63</td>
</tr>
</tbody>
</table>

#### Floating-Point Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPR0</td>
<td>0</td>
<td>63/63</td>
</tr>
<tr>
<td>FPR31</td>
<td>0</td>
<td>63/63</td>
</tr>
</tbody>
</table>

#### Condition Register

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR</td>
<td>0</td>
<td>31/31</td>
</tr>
</tbody>
</table>

#### Floating-Point Status and Control Register

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPSCR</td>
<td>0</td>
<td>31/31</td>
</tr>
</tbody>
</table>

#### Integer Exception Register

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>XER</td>
<td>0</td>
<td>31/31</td>
</tr>
</tbody>
</table>

#### Link Register

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR</td>
<td>0</td>
<td>31/63</td>
</tr>
</tbody>
</table>

#### Count Register

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTR</td>
<td>0</td>
<td>31/63</td>
</tr>
</tbody>
</table>

#### User Virtual Environment Architecture

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Offset</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBR268</td>
<td>TBL - Time Base Facility Lower (for Reading)</td>
<td></td>
<td>31/31</td>
</tr>
<tr>
<td>TBR269</td>
<td>TBU - Time Base Facility Upper (for Reading)</td>
<td></td>
<td>31/31</td>
</tr>
</tbody>
</table>

---

**Figure 4-2**

The UIA/VEA registers are a subset of the entire PowerPC programming model.
Although each processor implementation supports both single- and double-precision floating-point operations, the processor represents all floating-point numbers internally in double-precision format. In fact, single-precision data loaded from memory into an FPR will be converted into double-precision format by the floating-point unit. For more information on PowerPC floating-point operation, refer to Appendix C, “Floating-Point on the PowerPC.”

The following floating-point code fragment shows that FPRs are designated in source code by appending the FPR number to the letters $fr$. FPR designation is the same for both single- and double-precision floating-point operations.

```
fadd fr3, fr4, fr5 : fr3=fr4+fr5 where fr3,fr4, and fr5 are double-precision floating-point numbers
```

**Condition Register**

The function of the PowerPC condition register (CR) is similar to the x86 EFLAGS register; it reflects the results of logical and comparative operations. CR is 32 bits wide, independent of implementation, and has eight independent 4-bit result fields that are addressed by the aliases CR0–CR7. For example, bits CR[4-7] are known as CR1[0-3]. The condition register is shown in Figure 4-3.

All x86 programmers — repeat to yourself three times: “CR does *not* stand for control register.” I *still* make that mistake when discussing the condition register.

As we'll see in Chapter 6, “The PowerPC Instruction Set,” both integer and floating-point instructions can be coded so that their results are reflected in the condition register. When the Re bit is set in an integer or floating-point instruction's opcode, CR0 and CR1 have specific purposes as defined by the PowerPC UISA. CR0 reflects the results of integer operations at the completion of an integer instruction. Likewise, CR1 is updated with the final results of floating-point instructions.

The various bit fields within CR0–CR7 can be manipulated by:

- Explicitly moving a value from a GPR into the CR using an `mtcrf` instruction
- Explicitly moving one CR 4-bit field to another with the `mcrf` instruction
- Copying XER[0-3] to the CR using the `mcrxr` instruction
The condition register (CR) is divided into eight 4-bit status fields.

- Copying a specified field from the FPSCR to a CR using the mcrfs instruction
- Performing condition register logical instructions on bits within the CR
- Using the specific form of an integer instruction that updates CR0 with the results from that instruction
- Using the specific form of a floating-point instruction that updates CR1 with the results from that instruction
- Explicitly specifying a CR field to reflect the result of either an integer or floating-point compare instruction

The following example shows two different instructions using CR fields as both source and destination operands. Further examples can be found in both Chapter 11, “PowerPC Assembly Language Examples,” or Appendix A, “PowerPC Instruction Set Reference.”

```
mcrxr CR5 ; moves the contents of XER[0-3] into CR[20-23]
```
CR0 Field

CR0 is automatically updated with the results of integer instructions if the instruction’s Rc bit is set. As Figure 4-3 illustrates, bits CR[0-2] report the results of comparing the current integer operation to zero. CR[3] (CR0[EQ]) is set by copying the status of XER[SO] (defined in a subsequent section) upon completion of the instruction.

CR1 Field

CR1 is automatically updated to reflect the results of floating-point operations when the instruction’s Rc bit is set. As shown in Figure 4-3, bits CR1[0-3] are copied from FPSCR[0-3] at the completion of the floating-point operation.

CR2–CR7 Fields

The six remaining CR fields reflect the results of compare instructions. The encoding of the particular compare instruction specifies the CR field that is the destination for the results of the compare. Table 4-1 summarizes the bit field meanings for the 4 bits within each field CR2 through CR7.

Table 4-1
CR2–CR7 Bit Field Meanings

<table>
<thead>
<tr>
<th>CR[bit]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Less than or floating-point less than (LT, FL)</td>
</tr>
<tr>
<td></td>
<td>Integer compare:</td>
</tr>
<tr>
<td></td>
<td>rA &lt; (SIMM or rB) ; signed comparison</td>
</tr>
<tr>
<td></td>
<td>rA &lt; (UIMM or rB) ; unsigned comparison</td>
</tr>
<tr>
<td>1</td>
<td>Greater than or floating-point greater than (GT, FG)</td>
</tr>
<tr>
<td></td>
<td>Integer compare:</td>
</tr>
<tr>
<td></td>
<td>rA &gt; (SIMM or rB) ; signed comparison</td>
</tr>
<tr>
<td></td>
<td>rA &lt; (UIMM or rB) ; unsigned comparison</td>
</tr>
<tr>
<td></td>
<td>Floating-point compare: frA &lt; frB</td>
</tr>
<tr>
<td>2</td>
<td>Equal or floating-point equal (SO, FE)</td>
</tr>
<tr>
<td></td>
<td>Integer compare:</td>
</tr>
<tr>
<td></td>
<td>rA = (SIMM, UIMM, or rB)</td>
</tr>
<tr>
<td></td>
<td>Floating-point compare: frA = frB</td>
</tr>
<tr>
<td>3</td>
<td>Summary overflow or floating-point unordered (SO, FU)</td>
</tr>
<tr>
<td></td>
<td>Integer compare: copy of XER[SO] after instruction completes</td>
</tr>
<tr>
<td></td>
<td>Floating-point compare: frA and/or frB is Not a Number (NaN)</td>
</tr>
</tbody>
</table>
Floating-Point Status and Control Register

The floating-point status and control register (FPSCR) is 32 bits wide on all implementations, and is shown in Figure 4-4. The FPSCR performs the following tasks:

- Record, enable, and disable exceptions generated by floating-point operations
- Record the type of result produced by a floating-point operation
- Control the rounding mode used by floating-point operations

Once set, bits listed in Figure 4-4 as “sticky” remain set until explicitly cleared by an mcrfs, mfsfi, mfsf, or mfsubO instruction.

---

**Figure 4-4**
The floating-point status and control register (FPSCR) controls the operation of the floating-point unit.
**Integer Exception Register**

The PowerPC *integer exception register* (XER), shown in Figure 4-5, is a 32-bit register across all implementations. It reports the results of *completed* integer operations. We’ll say that an instruction has completed when the architectural register set of the processor has been updated with the results of the instruction’s operation. Depending on instruction encoding, the contents of XER[SO] can be copied into CR0[3] upon completion of execution. Table 4-2 defines the conditions reported by XER.

---

### Table 4-2

<table>
<thead>
<tr>
<th>Condition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>0</td>
</tr>
<tr>
<td>601-Specific Byte Compare Value</td>
<td>60</td>
</tr>
</tbody>
</table>

---

**Figure 4-5**

The integer exception register (XER), link register (LR), count register (CTR), and time base register (TBR) are available for application software use.
### Table 4-2
The XER Bit Definitions

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0          | SO       | Summary Overflow  
The summary overflow (SO) bit is set whenever an instruction (except mtlspr) sets the OV bit. SO is not cleared until software explicitly clears it using mtlspr or mtxr. SO is not altered by instructions that cannot overflow, such as compare operations. Refer to the integer arithmetic section of Chapter 11, “PowerPC Assembly Language Examples,” for further details. |
| 1          | OV       | Overflow  
The overflow (OV) bit is set when an overflow occurs during the execution of an integer instruction. For example, an integer add or subtract instruction (with OE=1 in the opcode) will set OV if the carry out of bit 0 is not equal to the carry out of bit 1, and OV will be cleared otherwise. OV is not altered by instructions that cannot overflow, such as compare operations. Refer to the integer arithmetic section of Chapter 11, “PowerPC Assembly Language Examples,” for further details. |
| 2          | CA       | Carry  
The carry (CA) bit is set to indicate that a carry out of bit 0 occurred during the execution of an integer instruction. In particular, CA is set in the following situations:  
- addc, subfc, adde, and subfe will set CA if there is a carry out of bit 0, and clear CA otherwise.  
- srav, srawi, and srw will set CA if any 1-bits have been shifted out of a negative operand, and clear CA otherwise.  
CA is not altered by instructions that cannot overflow, such as compare operations. Refer to the integer arithmetic section of Chapter 11, “PowerPC Assembly Language Examples,” for further details. |
| 3-15       | —        | Reserved |
| 16-23      | BCV      | 601 — Byte Compare Value  
603, 604, 620 — Reserved  
This field is valid only on the PowerPC 601 microprocessor; on other implementations, this field is reserved. On the 601, this field contains the byte to be compared by the POWER lscbx (load string and compare byte indexed) instruction. |
Table 4-2
The XER Bit Definitions (Continued)

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td>25-31</td>
<td>BC</td>
<td>Byte Count</td>
</tr>
</tbody>
</table>
This field specifies the number of bytes to be transferred by a lswx (load string word indexed), stswx (store string word indexed), or lscbx (load string and compare byte indexed) instruction.

The XER can also hold a byte count for string operations. The load string word indexed (lswx) and store string word indexed (stswx) instructions use XER[25-31] as their byte count value. Both string instructions are described in Appendix A, “PowerPC Instruction Set Reference.”

On the 601, the XER has a field that does not exist on other processor implementations. Bits 16-23 of the XER are used as a byte compare value for the lscbx (load string and compare byte indexed) instruction defined by the POWER architecture. On all other processor implementations, this field is reserved and should be loaded with zeros when manipulating the XER register. For specific information on the lscbx instruction, refer to the 601 instruction reference on the CD-ROM.

Example:

```
    mf spr r5,1  ; put the contents of the XER register in GPR 5
    mf xer r5   ; the simplified mnemonic equivalent of the
                 ; above instruction
```

**Link Register**

The link register (LR) supplies the branch target address for use with the bclr instruction. As shown in Figure 4-5, the LR is a 64-bit register on 64-bit implementations and is 32 bits wide otherwise.

As with all PowerPC registers that hold instruction addresses (not data addresses), the lower two bits of the LR are ignored by the processor due to the 32-bit alignment of instructions. There are two ways to load a value into the link register:

- Explicitly loading a value into (or from) the LR using the mtspr or mf spr instructions
- Using the form of conditional or unconditional branch instructions that loads the LR with the address of the instruction following the branch instruction (the return address)
Example:

\begin{verbatim}
  mfspr r4,8 ; put the contents of LR into GPR 4
  mflr r4 ; the simplified mnemonic equivalent of the
  ; above instruction
\end{verbatim}

**Count Register**

The *count register* (CTR) is generally used as a loop counter register. As shown previously in Figure 4-5, the CTR is a 64-bit register on 64-bit implementations and is 32 bits wide otherwise.

There are two main uses for the CTR:

- The CTR can be used as a loop count-down register for properly coded branch instructions. Note that decrementing the CTR when it contains zero results in the CTR being set to 0xffffffff (-1).
- The CTR can be used to hold the target address for the `bcctrx` (branch conditional to count register) instruction.

Example:

\begin{verbatim}
  mfspr r5,9 ; put the contents of the CTR into GPR 9
  mfctr r5 ; the simplified mnemonic equivalent of the
  ; previous instruction.
\end{verbatim}

**Time Base Register**

The *time base register* (TBR) is the only register defined by the PowerPC virtual environment architecture (VEA). Like the UISA registers, the VEA time base register can be accessed by both user- and supervisor-mode software. However, user-level software can only read from the TBR. Supervisor-level software can read from and write to the TBR.

The PowerPC 603, 604, and 620 have a TBR. The 601 does not, but employs its *real-time clock* (RTC) register in a similar fashion to the TBR; the 601’s RTC register is discussed in the 601 section that follows.

The TBR is a 64-bit structure that comprises two 32-bit registers: the *time base upper* (TBU) and *time base lower* (TBL) registers, as shown in Figure 4-5. The TBR value increments at a system implementation-dependent rate until it reaches the 64-bit value 0xffffffffffffff, at which point it
rolls over to zero and resumes incrementing. No exception is generated by a TBR roll-over; explicit checking is required.

Note that the following code fragment is 32-bit implementation specific. On 64-bit implementations, such as the 620, it would be possible to read the TBR with a single instruction. Remember, reading from the TBR is a user-mode operation; writing to the TBR requires software executing at supervisor level. More extensive examples employing the time base register are found in Chapter 11, “PowerPC Assembly Language Examples.”

\[
\text{mftb r3,268} : \text{put contents of TBL into GPR 3} \\
\text{mftb r4,269} : \text{put contents of TBU into GPR 4}
\]

: another way to perform the same operation using simplified mnemonics:

\[
\text{mftb r3} \quad : \text{put contents of TBL into GPR 3} \\
\text{mftbu r4} \quad : \text{and TBU into GPR 4}
\]

**OEA and Implementation-Specific Register Set**

The final group of PowerPC registers consists of those defined by the PowerPC operating environment architecture (OEA) and processor implementation-specific registers. If user-level software attempts to access any of the registers discussed in this section, a privilege-level exception will result. Chapter 10, “Exceptions and Interrupts,” deals exclusively with PowerPC exceptions. If supervisor-level software attempts to access an invalid (undefined) special-purpose register (SPR), the result depends on the type of access:

- Attempting to load a value into an invalid SPR (using the mtspr instruction) will be executed as a no-op.
- Attempting to load a value from an invalid SPR (using the mfspr instruction) will result in loading the target register with an undefined value.

The following list enumerates the remaining categories of PowerPC registers as well as several examples in each category. Each category may include both OEA-defined and implementation-specific registers. Note that this is just an overview, and processor-specific implementations are addressed later in this chapter.
- **Configuration Registers**
The machine state register (MSR), processor version register (PVR), processor identification register (PIR), and hardware implementation dependent (HID) registers are examples of configuration registers.

- **Memory Management Registers**
The instruction and data block address translation (BAT) registers, table search description register (SDR1), address space register (ASR), and segment registers are examples of memory management registers. Additionally, the 603-specific IMISS/DMISS and HASH1/HASH2 registers fall into this category.

- **Exception Handling Registers**
The data address register (DAR), general special-purpose registers (SPRG0–SPRG3), data storage (memory) interrupt service register (DSISR), and machine status save/restore registers (SRR0 and SRR1) represent the set of exception handling registers.

- **Performance Monitoring Registers**
The performance monitor counter registers (PMC1 and PMC2), the monitor mode control register 0 (MMCR0), and the sampled data address and sampled data instruction registers (SDA and SIA) are performance monitoring registers. The performance monitoring facility is available on the PowerPC 604 and 620 processors.

- **Miscellaneous Registers**
The writable time base register (TBR) (603, 604, and 620 only), decre- menter register (DEC), external access register (EAR), and instruction address breakpoint and data address breakpoint registers (IABR and DABR) are examples of miscellaneous registers.

Just as the width of some of the user-level registers depends on the implementation width, some OEA and implementation-specific registers do the same. The machine state register, the data address register, table search description register, machine status save and restore registers, and special-purpose registers are 64 bits wide on 64-bit implementations and 32 bits wide on 32-bit implementations.
Common OEA Registers

Each PowerPC implementation can have a unique set of supervisor-level registers. Therefore, we’ll look at each implementation separately. Because all user-level registers are accessible by supervisor-level software, we’ll show the entire register set for each implementation. That way, we’ll have one reference for the complete register set on each PowerPC processor.

Some PowerPC OEA registers are common across all implementations. In the sections that follow, we’ll examine the registers common to all PowerPC processors and then cover implementation-specific registers, processor by processor.

Machine State Register

The machine state register (MSR), shown in Figure 4-6, has a variety of functions and is the main control register for the processor. The MSR is 32 bits wide on 32-bit implementations. On 64-bit implementations, it occupies the rightmost 32 bits of the 64-bit register. The value in the MSR can be changed in the following ways:

- The PowerPC exception mechanism can update the bits of the MSR as defined in Table 4-3.
- A value can be explicitly moved into the MSR using the mfsr instruction.
- System call (sc) and return from interrupt (rfi) instructions can modify the value of the MSR.

Table 4-3
The Machine State Register Bit Meanings

<table>
<thead>
<tr>
<th>Bit Number (32-bit)</th>
<th>Bit Number (64-bit)</th>
<th>Name</th>
<th>Power-On Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not available</td>
<td>0</td>
<td>SF</td>
<td>1</td>
<td>64-bit Mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 = The 64-bit processor is running in 32-bit mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 = Processor is running in 64-bit mode. (default)</td>
</tr>
</tbody>
</table>
## Table 4-3
The Machine State Register Bit Meanings (Continued)

<table>
<thead>
<tr>
<th>Bit Number (32-bit)</th>
<th>Bit Number (64-bit)</th>
<th>Name</th>
<th>Power-On Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1-32</td>
<td>Unknown</td>
<td>Reserved, but saved to SRR1 when an exception occurs.</td>
<td></td>
</tr>
<tr>
<td>1-4</td>
<td>33-36</td>
<td>Unknown</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>5-9</td>
<td>37-41</td>
<td>Unknown</td>
<td>Reserved, but saved to SRR1 when an exception occurs.</td>
<td></td>
</tr>
<tr>
<td>10-12</td>
<td>42-44</td>
<td>Unknown</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>45</td>
<td>POW</td>
<td>0</td>
<td>Power Management Enable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 = Disables programmable power modes (normal mode)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 = Enables programmable (reduced) power modes</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This bit has no effect on dynamic power management (DPM).</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>46</td>
<td>TGPR</td>
<td>0</td>
<td>Temporary GPR Remapping (603 only)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 = Normal operation</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 = TGPR mode. GPRO–GPR3 are remapped to TGPRO–TGPR3 for use by TLB miss routines.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Note: The contents of GPRO–GPR3 will remain unchanged while MSR[TGPR]=1. Attempts to use GPR4–GPR31 with MSR[TGPR]=1 will have undefined results.</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>47</td>
<td>ILE</td>
<td>0</td>
<td>Exception Little-Endian Mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>When an exception occurs, this bit is copied into MSR[ILE] to select the endian mode for the context established by the exception.</td>
<td></td>
</tr>
<tr>
<td>Bit Number (32-bit)</td>
<td>Bit Number (64-bit)</td>
<td>Name</td>
<td>Power-On Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------</td>
<td>------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>16</td>
<td>48</td>
<td>EE</td>
<td>0</td>
<td>External Interrupt Enable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 = While the bit is cleared, the processor delays recognition of external interrupts and decrementer exception conditions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 = The processor is enabled to take an external interrupt or the decrementer exception.</td>
</tr>
<tr>
<td>17</td>
<td>49</td>
<td>PR</td>
<td>0</td>
<td>Privilege Level</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 = The processor can execute both user- and supervisor-level instructions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 = The processor can execute only user-level instructions.</td>
</tr>
<tr>
<td>18</td>
<td>50</td>
<td>FP</td>
<td>0</td>
<td>Floating Point Available</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 = The processor prevents dispatch of floating-point instructions, including FP loads, stores, and moves.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 = The processor can execute floating-point instructions.</td>
</tr>
<tr>
<td>19</td>
<td>51</td>
<td>ME</td>
<td>0</td>
<td>Machine Check Enable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 = Machine check exceptions are disabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 = Machine check exceptions are enabled.</td>
</tr>
<tr>
<td>20</td>
<td>52</td>
<td>FEO</td>
<td>0</td>
<td>Floating-Point Exception Mode 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>See Figure 4-6.</td>
</tr>
<tr>
<td>21</td>
<td>53</td>
<td>SE</td>
<td>0</td>
<td>Single-Step Trace Enable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 = The processor executes instructions normally.</td>
</tr>
<tr>
<td>Bit Number (32-bit)</td>
<td>Bit Number (64-bit)</td>
<td>Name</td>
<td>Power-On Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------</td>
<td>------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>22</td>
<td>54</td>
<td>BE</td>
<td>0</td>
<td>Branch Trace Enable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 = The processor executes branch instructions normally.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 = The processor generates a trace exception (0xOd00) upon the completion of a branch instruction, regardless of whether the branch was taken.</td>
</tr>
<tr>
<td>23</td>
<td>55</td>
<td>FE1</td>
<td>0</td>
<td>Floating-Point Exception Mode 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>See Figure 4-6.</td>
</tr>
<tr>
<td>24</td>
<td>56</td>
<td>—</td>
<td>Unknown</td>
<td>Reserved, but saved in SRR1 when an exception occurs.</td>
</tr>
<tr>
<td>25</td>
<td>57</td>
<td>IP</td>
<td>1 (typical)</td>
<td>Exception Prefix</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The setting of this bit specifies whether an exception vector offset is prepended with 0xes or 0s. In the following description, nnnnn is the offset of the exception.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 = Exceptions are vectored to the physical address 0x000nnnnn.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 = Exceptions are vectored to the physical address 0xffffnnnnn.</td>
</tr>
</tbody>
</table>
### Table 4-3
The Machine State Register Bit Meanings (Continued)

<table>
<thead>
<tr>
<th>Bit Number (32-bit)</th>
<th>Bit Number (64-bit)</th>
<th>Name</th>
<th>Power-On Value</th>
<th>Description</th>
</tr>
</thead>
</table>
| 26                  | 58                  | IR   | 0              | Instruction Address Translation  
0 = Instruction address translation is disabled.  
1 = Instruction address translation is enabled. |
| 27                  | 59                  | DR   | 0              | Data Address Translation  
0 = Data address translation is disabled.  
1 = Data address translation is enabled. |
| 28 -                | 60                  | Unknown | Reserved, but saved in SRR1 when an exception occurs. |
| 29                  | 61                  | PM   | 0              | Performance monitor |
| 30                  | 62                  | RI   | 0              | Recoverable Exception (for system reset and machine check exceptions)  
0 = Exception is not recoverable.  
1 = Exception is recoverable. |
| 31                  | 63                  | LE   | 0              | Little Endian Mode Enabled  
0 = The processor runs in big endian mode.  
1 = The processor runs in little endian mode. |
The 32-bit register maps to the rightmost 32 bits of the 64-bit register.

The machine state register (MSR) is the main control register for PowerPC processors.

**Processor Version Register**

The 32-bit *processor version register* (PVR), shown in Figure 4-7, is a read-only register. Its two 16-bit fields contain the processor's version number and revision number.

The *version number* is a 16-bit value that uniquely identifies each processor implementation. If new models of the same processor are created, such as the case of the 603 and 603e, the processor version number will change.

The *revision number* is a 16-bit value that uniquely identifies each release within a particular implementation. For example, if we needed to
distinguish between revision 2.0 and revision 2.1 of the 604, this is the 16-bit value we'd need to check.

<table>
<thead>
<tr>
<th>Version</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001</td>
<td>PPC 601</td>
</tr>
<tr>
<td>0x0003</td>
<td>PPC 603</td>
</tr>
<tr>
<td>0x0006</td>
<td>PPC 603e</td>
</tr>
<tr>
<td>0x0004</td>
<td>PPC 604</td>
</tr>
<tr>
<td>0x0014</td>
<td>PPC 620</td>
</tr>
</tbody>
</table>

**Figure 4-7**
The processor version register (PVR) is used by software to distinguish between PowerPC processor implementations.

The following code example reads the PVR and tests to see whether the processor is a 604 with a revision greater or equal to 2.0.

```assembly
mfspr r10, PVR ; PVR has been defined as SPR 287
or  r11, r10, r10 ; r11 = r10 - save a copy of PVR
andi r10, r10, 0xffff0000 ; looking for processor version
cmpli 0, 0, r10, 0x40000 ; compare r10 with immediate
    ; 0x40000 = 604
bne Not604 ; simplified branch if not 604
andi r11, r11, 0xffff ; looking for processor revision
cmpli 0, 0, r10, 0x30 ; 2.0?
blt Not604 ; simplified branch
    ; revision is pre-3.0, get out

LetsGo:
    ; it's the processor and version
... ; we expect - do what we want...

Not604:
    ; not the right processor...continue
```

**Segment Registers**
The PowerPC's *segment registers* implement virtual memory. Segment registers are present only on 32-bit implementations such as the 601, 603, and 604. (There are no segment registers on the 620 because equivalent
information is contained in the 620's address space register.) Therefore, the remainder of this discussion of PowerPC segment registers is applicable only to 32-bit processor implementations.

Each of the 16 segment registers are 32 bits wide. As shown in Figure 4-8, there are two possible formats for a segment register. The value of the T bit (bit 0) determines whether the segment is memory-mapped or direct-store and directs the subsequent interpretation of the remaining bit fields.

<table>
<thead>
<tr>
<th>T = 0 format</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T = 1 format</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

If T = 0, the effective address is a reference to a memory-mapped segment. In this case, the bit fields are interpreted as shown in Table 4-4. If T = 1, the effective address is a reference to a direct-store (I/O) segment. No reference is made to the page tables. In this case, the bit fields are interpreted as shown in Table 4-5.

Segment registers can be accessed using the mtsr and mtsrin instructions. After accessing segment registers, software must perform a context synchronizing sequence to ensure that memory accesses respond according to the new state of the segment register. Context synchronization is discussed in Chapter 6, "The PowerPC Instruction Set."
### Table 4-4
Bit Field Definitions for $T = 0$ Segment Registers

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>The T bit selects the format of the segment register as shown in Figure 4-8.</td>
</tr>
<tr>
<td>1</td>
<td>$K_s$</td>
<td>Supervisor-State Protection Key Defines the access protection for pages contained in this segment.</td>
</tr>
<tr>
<td>2</td>
<td>$K_p$</td>
<td>User-State Protection Key Defines the access protection for pages contained in this segment.</td>
</tr>
<tr>
<td>3</td>
<td>N</td>
<td>No-Execute Protection Bit Segments with this bit defined are data-only segments that cannot be used to execute code.</td>
</tr>
<tr>
<td>4-7</td>
<td>(none)</td>
<td>Reserved</td>
</tr>
<tr>
<td>8-31</td>
<td>VSID</td>
<td>Virtual Segment ID</td>
</tr>
</tbody>
</table>

### Table 4-5
Bit Field Definitions for $T = 1$ Segment Registers

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>The T bit selects the format of the segment register as shown in Figure 4-8.</td>
</tr>
<tr>
<td>1</td>
<td>$K_s$</td>
<td>Supervisor-State Protection Key Used to help define the access protection for pages contained in this segment.</td>
</tr>
<tr>
<td>2</td>
<td>$K_p$</td>
<td>User-State Protection Key Used to help define the access protection for pages contained in this segment.</td>
</tr>
<tr>
<td>3-11</td>
<td>BUlD</td>
<td>Bus Unit ID</td>
</tr>
<tr>
<td>12-31</td>
<td>(none)</td>
<td>Device-specific data for I/O controller</td>
</tr>
</tbody>
</table>
**Table Search Description Register**

The *table search description register* (SDR1) is used when generating addresses associated with a page table. As shown in Figure 4-9, SDR1 is 32 bits on 32-bit implementations and 64 bits on 64-bit implementations. Table 4-6 summarizes the bit field descriptions. Page tables on the PowerPC are analogous to those found on the x86. Page tables and the use of SDR1 are discussed in Chapter 8, "Memory Management."

| The Table Search Description Register for 32-bit PowerPC Implementations |
|---|---|---|---|---|---|
| | HTABORG | Base Address | Maskable Bits | | HTABSIZE |
| 0 | 6 | 7 | 15 | 16 | 22 | 23 | 31 |

| The Table Search Description Register for 64-bit PowerPC Implementations |
|---|---|---|---|---|---|
| | HTABORG | | Maskable Bits | | HTABSIZE |
| 0 | 45 | 46 | 58 | 59 | 63 |

| Reserved |

<table>
<thead>
<tr>
<th>Figure 4-9</th>
</tr>
</thead>
<tbody>
<tr>
<td>The table search description register (SDR1) is used by the PowerPC processor's memory paging mechanism.</td>
</tr>
</tbody>
</table>

Processor implementation width impacts the function of the SDR1. As you would expect, the number of bits required to generate page table addresses is different for 32-bit processors than for 64-bit processors. Table 4-7 compares the function of the SDR1 on both implementation widths.

**Table 4-6**

The SDR1 Register Bit Definitions

<table>
<thead>
<tr>
<th>Bit Number (32-bit)</th>
<th>Bit Number (64-bit)</th>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–15</td>
<td>0–45</td>
<td>HTABORG</td>
<td>Physical base address of page table</td>
</tr>
<tr>
<td>16–22</td>
<td>46–58</td>
<td>(none)</td>
<td>Reserved</td>
</tr>
<tr>
<td>23–31</td>
<td>59–63</td>
<td>HTABSIZE</td>
<td>Encoded size of page table (used to generate mask)</td>
</tr>
</tbody>
</table>
### Table 4-7
SDR1 Function on 32-Bit and 64-Bit Implementations

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>32-Bit Implementation</th>
<th>64-Bit Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required page table boundary</td>
<td>$2^{16} = 64$ Kbytes</td>
<td>$2^{18} = 256$ Kbytes</td>
</tr>
<tr>
<td>Minimum number of bits used from hash table for indexing</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Minimum page table size</td>
<td>$2^{(\text{number of hash table bits})} \times \frac{\text{sizeof PTE}_{64bit}}{2^{18}} = \frac{64 \times 128}{256} = 64$ Kbytes</td>
<td>$2^{(\text{number of hash table bits})} \times \frac{\text{sizeof PTE}_{64bit}}{2^{11}} = \frac{64 \times 128}{256} = 128 = 256$ Kbytes</td>
</tr>
<tr>
<td>Page table size range</td>
<td>$2^{16} \text{ bytes} - 2^{25} \text{ bytes}$</td>
<td>$2^{18} \text{ bytes} - 2^{46} \text{ bytes}$</td>
</tr>
</tbody>
</table>

### General Special-Purpose Registers
As their name implies, the general special-purpose registers (SPGRs) are both general- and special-purpose. The SPRGs are 32 bits on 32-bit implementations and 64 bits on 64-bit implementations as shown in Figure 4-10. These registers can be used by an operating system (or other privileged software) to perform miscellaneous tasks. Although specific uses for these registers are not dictated by the PowerPC architecture specification, it does specify a conventional use for each register, as described in Table 4-8.

### Figure 4-10
The general special-purpose registers (SPRG0–SPRG3) are used by system software to perform miscellaneous operations.
Table 4-8
Conventional SPRG Usage

<table>
<thead>
<tr>
<th>Register</th>
<th>PowerPC Architecture Description of Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPRG0</td>
<td>Software may load a unique physical address in this register to identify an area of memory reserved for use by the first-level exception handler. This area must be unique for each processor in a multiprocessor system.</td>
</tr>
<tr>
<td>SPRG1</td>
<td>This register may be used as a scratch register by the first-level exception handler to save the contents of a GPR. The saved GPR can then be loaded from SPRG0 and used as a base register to save other GPRs to memory.</td>
</tr>
<tr>
<td>SPRG2</td>
<td>This register may be used by the operating system as needed.</td>
</tr>
<tr>
<td>SPRG3</td>
<td>This register may be used by the operating system as needed.</td>
</tr>
</tbody>
</table>

**Machine Status Save/Restore Registers**

The *machine status save/restore registers* (SRR0 and SRR1) save processor status information when an exception occurs and restore processor status at the end of exception handling (signaled by an rfi instruction). Both SRR0 and SRR1 are 64 bits wide on 64-bit implementations and 32 bits wide on 32-bit implementations as shown in Figure 4-11.

When an exception occurs, the processor loads SRR0 with the address of either the instruction that caused the exception or an instruction that follows. All instructions prior to the one pointed to by SRR0 are guaranteed to have completed execution; the state of the instruction pointed to by SRR0 depends on the type of the exception. When exception handling is complete and an rfi instruction is executed, the instruction at the address contained in SRR0 is restarted. Exception-specific manipulation of SRR0 is discussed in Chapter 10, “Exceptions and Interrupts.”

SRR1 saves and restores processor status during exception conditions. However, the data that is loaded into SRR1 either comes from the MSR or is exception-specific information. Figure 4-11 shows SRR1. For example, when an exception occurs on a 32-bit processor, bits SRR1[0,5-9,16-31] are loaded with MSR[0,5-9,16-31]. The corresponding event on a 64-bit processor causes bits SRR1[0-32,37-41,48-63] to be loaded with the corresponding bits in the 64-bit MSR.
These bits represent sufficient information to restore the processor to the state that existed prior to the exception. The state of the processor as defined by system register settings and address translation is known as the context of the processor and is discussed in detail in subsequent chapters. After exception handling is complete, rfi restores the context of the processor from SRR1 before returning to the exception-causing code. Exceptions and their effect on SRR0 and SRR1 are described in Chapter 10, "Exceptions and Interrupts."

Note that the PowerPC architecture specification warns that some PowerPC implementations may modify SRR0 and SRR1 in the following situations:

- During every instruction fetch that requires address translation when MSR[IR]=1
- During every instruction execution that requires address translation when MSR[DR]=1
Decrementer Register

The 32-bit decrementer (DEC) register, shown in Figure 4-12, is a counter register that can generate periodic interrupt. The frequency at which the DEC register counts down is system implementation-dependent and is identical to that of the time base register.

Figure 4-12
The decrementer (DEC) register is a single field that reports a continuously decrementing count that can be used for timing operations.

If DEC = 0, the next decrement will generate an exception unless masked by MSR[EE]. The operation of the decrementer adheres to the following rules:

- The frequency of the time base and the DEC countdown are identical.
- Loading a GPR from DEC has no effect on the value in DEC.
- Storing a value from a GPR to DEC replaces the value in DEC.
- Whenever bit 0 (MSb) of DEC changes from 0 to 1, a decrementer exception request is generated. Multiple DEC exception requests may be received before the first exception occurs; however, any additional requests are canceled when the exception occurs for the first request.
- If DEC is altered by software and the content of bit 0 is changed from 0 to 1, an exception request is generated.

For real-time applications, it is important to understand the impact of speculative execution as it applies to DEC. While the processor is speculatively executing code, DEC can be read at a point that is not within the linear instruction stream. If it is important that DEC be read only at a time that corresponds to the position of the read within the linear instruction stream, a context synchronization operation (such as isync) should be placed immediately before reading the DEC register.
Data Address Breakpoint Register

The data address breakpoint register (DABR) is an optional register that is defined by the PowerPC architecture to provide a breakpoint facility for data loads and stores. The DABR is 64 bits wide on 64-bit implementations and 32 bits wide on 32-bit implementations as shown in Figure 4-13. The 601, 604, and 620 implement the DABR; the 603 does not have a DABR.

![Figure 4-13](image)

The data address breakpoint register (DABR) sets debugger-type breakpoints for load and store operations.

The DABR allows detection of load and store accesses to a specific dword of data; instruction fetches are not trapped. Table 4-9 describes the bit fields of the DABR for both 32-bit and 64-bit implementations.

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-28</td>
<td>Data address breakpoint</td>
</tr>
<tr>
<td>29</td>
<td>Breakpoint translation enable</td>
</tr>
<tr>
<td>30</td>
<td>Data write enable</td>
</tr>
<tr>
<td>31</td>
<td>Data read enable</td>
</tr>
</tbody>
</table>

The following conditions define a data address breakpoint match for load/store operations and generate a data access exception (DSI). All of the following must be true:

- The effective address of the load/store operation’s memory operand and DAB match: EA[0-28] = DABR[DAB]. In other words, there is an
address match if any part of the load/store touches any part of the
dword specified in the DABR.

- The state of the data address translation enable bit in the MSR matches
  the state of the breakpoint translation enable bit in the DABR:
  MSR[DR] = DABR[BT].
- The instruction is a store and DABR[DW] = 1 or the instruction is a
  load and DABR[DR] = 1.

Unlike the DABR, the instruction address breakpoint register (IABR) is
not architected in the PowerPC specification. Although each implementa-
tion discussed in this book implements the IABR register, it is not an ele-
ment of the OEA register set and is discussed in the implementation-specific
sections that follow.

**External Access Register**

The 32-bit *external access register* (EAR) is accessed with the `eciwx`
(external control in word indexed) and `ecowx` (external control out word
indexed) instructions. The EAR can also be accessed explicitly by using
`mtsp` and `mfspr` instructions. Figure 4-14 shows the format of the EAR
and Table 4-10 describes the bit definitions for the EAR. The EAR is
defined as *optional* by the PowerPC architecture.

![Figure 4-14](image)

The external access register (EAR) is used during certain I/O operations initiated by the
`eciwx` and `ecowx` instructions.

The `eciwx` and `ecowx` instructions perform memory-mapped I/O
operations. When applications or drivers need to perform I/O for a particu-
lar device, they use the EAR[RID] value to select the device in a system
implementation-dependent manner. If EAR[E] is set, the `eciwx` and `ecowx`
instructions are able to perform input/output operations to the selected
device. For further information on the usage of the EAR, refer to `eciwx` and
`ecowx` in Appendix A, “PowerPC Instruction Set Reference.”
Table 4-10

The EAR Bit Definitions

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>E</td>
<td>Enable Bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 Disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 Enabled</td>
</tr>
<tr>
<td>1-25</td>
<td></td>
<td>If enabled, the <code>eciwx</code> and <code>ecowx</code> instructions can perform the specified external operation. If disabled, an <code>eciwx</code> or <code>ecowx</code> instruction causes a data access exception (DSI).</td>
</tr>
<tr>
<td>26-31</td>
<td>RID</td>
<td>Resource ID. On the 601, this field comprises only bits 28-31; bits 26 and 27 are reserved and have the value 0.</td>
</tr>
</tbody>
</table>

Data Address Register

The data address register (DAR) holds the effective address generated by a memory access instruction that causes an exception. For example, if a load instruction calculates and loads a value from an effective address that causes an alignment exception, the EA for that load is placed in the DAR for use during exception handling. As shown in Figure 4-15, the DAR is 64 bits wide on 64-bit implementations and is 32 bits wide on 32-bit implementations.

Figure 4-15

If a load/store operation generates an exception, the data address register (DAR) reports the address of the access.
Data Access and Alignment Exception Source Register

The 32-bit data storage (memory) interrupt exception source register (DSISR) identifies the cause of a data access exception (DSI) or data alignment exception (DAE). The DSISR is shown in Figure 4-16. Bit definitions for the DSISR vary depending on the type of exception: data access or alignment. Each exception and the DSISR bit definitions are discussed in Chapter 10, “Exceptions and Interrupts.”

<table>
<thead>
<tr>
<th>Bit</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DSISR</td>
</tr>
<tr>
<td>31</td>
<td>DSISR</td>
</tr>
</tbody>
</table>

Figure 4-16
The data access/alignment exception source register (DSISR) helps determine the exception-causing instruction during exception handling.

PowerPC 601 Register Set

Starting with this section, we’ll examine PowerPC registers that are specific to particular processor implementations. And following the evolutionary path of PowerPC processors, we’ll start off by examining the register set of the PowerPC 601, shown in Figure 4-17.

The 601 has two user-level, implementation-specific registers: the real-time clock (RTC) register and the multiply quotient (MQ) register. These registers are present on the 601 to ease the transition to the PowerPC architecture for programmers who have to port code written for POWER-based computers.

In addition to the RTC and MQ registers, the 601 provides the following implementation-specific registers:

- The checkstop sources and enables register (HID0).
- The 601 debug modes register (HID1).
- The instruction address breakpoint register (IABR) data address breakpoint register (DABR). The IABR and DABR registers are alternately known as HID2 and HID5.
- The processor identification register (PIR), alternately known as HID15.
The PowerPC 601’s programming model, like the i486, contains both supervisor-level and user-level registers.

**Real-Time Clock Registers**

The two 32-bit user-level RTC registers track the time and date on 601-based systems. On the 601, the RTC serves the same purpose as the time base facility on other PowerPC processors. As shown in Figure 4-18, the
RTC uses an upper (RTCU) and lower (RTCL) register to count seconds and nanoseconds respectively. There is enough range in the RTC registers to provide a 135-year rollover calendar.

**Figure 4-18**
The 601 real-time clock (RTC) registers count seconds and nanoseconds.

Like the time base registers, the RTC registers are read-only for user-level software and read/write for supervisor-level software. To emphasize this point, there are different SPR numbers for the RTC registers depending on the type of instruction used to access the register. To write to the RTC registers using mtspr, supervisor-level software uses SPR20 and SPR21. To read from the RTC using mfspr, all software uses SPR4 and SPR5. The use of different SPR numbers is shown in Figure 4-17.

Because the 601 implements RTC registers instead of the PowerPC-architected time base registers, the mftb (move from time base) instruction is not implemented. However, it is recommended that the mftb instruction be used for upward PowerPC compatibility. When executed, an illegal instruction exception will result, allowing the functionality of mftb to be emulated using the mfspr instruction. Because all other PowerPC implementations discussed in this book implement the time base register, the same code will work transparently on other processors without causing an exception.

The real-time clock lower (RTCL) register is a 32-bit nanosecond counter that has the following characteristics:

- The time between two consecutive ticks of the RTCL is guaranteed to be no longer than the time required to execute 10 `addi` (add immediate) instructions.
- Bits 0, 1, and 25–31 are reserved.
- The least-significant implemented bit of the RTCL (bit 24) is incremented every 128 ns.
The total period of the RTCL is $10^9$ ns (one second).

Unless altered by software, the RTCL reaches its maximal count value of $999,999,872$ ($10^9 - 128$) after $999,999,999$ ns. The next time RTCL is incremented, it cycles to all zeros and RTCU is incremented.

Reading from the RTCL register (using the \texttt{mf spr} instruction) does not affect its contents. Unimplemented bits are read as zeros.

When writing to the RTCL (using the \texttt{mt spr} instruction) using a GPR value, the bits of the GPR that correspond to unimplemented RTCL bits are ignored.

The real-time clock upper (RTCU) register is a full 32-bit second counter. When each bit is set in the RTCU (the maximal count), a subsequent RTCU increment will result in rollover to zero. Chapter 11, “PowerPC Assembly Language Examples,” contains code examples of how to read the RTC (and time base) registers.

\section*{Multiply Quotient Register}

The user-level MQ register is not found on any other PowerPC implementation. The MQ register supports those unique features used by the POWER architecture instructions that perform multiply and divide operations.

The 32-bit MQ register, shown in Figure 4-19, is not defined by the PowerPC architecture. It functions as a register extension to accommodate the product for the multiply ($\texttt{mulx}$) and divide ($\texttt{divx}$) instructions. The MQ register is also used as an operand during long rotate and shift operations.

Although not part of the PowerPC architecture, the MQ register may be modified as a side effect during the execution of the PowerPC instructions listed in Table 4-11. In this case, the undefined state results from the fact that the MQ register is used as a scratch register during multiply and divide operations on the 601.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{mq.png}
\caption{The 601's multiply quotient register (MQ) holds intermediate values during multiply and divide operations.}
\end{figure}
Table 4-11

601 Instructions That Affect the MQ Register

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Defining Architecture</th>
<th>Instruction Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>div</td>
<td>POWER</td>
<td>divide</td>
</tr>
<tr>
<td>divs</td>
<td>POWER</td>
<td>divide short</td>
</tr>
<tr>
<td>divw</td>
<td>PowerPC</td>
<td>divide word</td>
</tr>
<tr>
<td>divwu</td>
<td>PowerPC</td>
<td>divide word unsigned</td>
</tr>
<tr>
<td>mul</td>
<td>POWER</td>
<td>multiply</td>
</tr>
<tr>
<td>mulli</td>
<td>PowerPC</td>
<td>multiply low immediate</td>
</tr>
<tr>
<td>mullhw</td>
<td>PowerPC</td>
<td>multiply high word</td>
</tr>
<tr>
<td>mullhwu</td>
<td>PowerPC</td>
<td>multiply high word unsigned</td>
</tr>
<tr>
<td>mullw</td>
<td>PowerPC</td>
<td>multiply low</td>
</tr>
<tr>
<td>sle</td>
<td>POWER</td>
<td>shift left extended</td>
</tr>
<tr>
<td>sleq</td>
<td>POWER</td>
<td>shift left extended with MQ</td>
</tr>
<tr>
<td>sliq</td>
<td>POWER</td>
<td>shift left immediate with MQ</td>
</tr>
<tr>
<td>slliq</td>
<td>POWER</td>
<td>shift left long immediate with MQ</td>
</tr>
<tr>
<td>sllq</td>
<td>POWER</td>
<td>shift left long with MQ</td>
</tr>
<tr>
<td>slq</td>
<td>POWER</td>
<td>shift left with MQ</td>
</tr>
<tr>
<td>sraiq</td>
<td>POWER</td>
<td>shift right algebraic immediate with MQ</td>
</tr>
<tr>
<td>sraq</td>
<td>POWER</td>
<td>shift right algebraic with MQ</td>
</tr>
<tr>
<td>sre</td>
<td>POWER</td>
<td>shift right extended</td>
</tr>
<tr>
<td>srea</td>
<td>POWER</td>
<td>shift right extended algebraic</td>
</tr>
<tr>
<td>sreq</td>
<td>POWER</td>
<td>shift right extended with MQ</td>
</tr>
<tr>
<td>sriq</td>
<td>POWER</td>
<td>shift right immediate with MQ</td>
</tr>
<tr>
<td>srlq</td>
<td>POWER</td>
<td>shift right long immediate with MQ</td>
</tr>
<tr>
<td>srq</td>
<td>POWER</td>
<td>shift right long with MQ</td>
</tr>
</tbody>
</table>

Block Address Translation Registers

The supervisor-level block address translation (BAT) registers are part of the address translation mechanism for blocks of physical memory on PowerPC microprocessors. Address translation, often called *mapping*, is typically performed by operating systems or firmware. The two primary means of address translation and memory region protection on PowerPC processors are BAT *register-based* and segment *register-based*. In general, if a particular address can be translated using either mechanism, BAT address translation overrides segment-based translation. Both mechanisms are discussed in detail in Chapter 8, “Memory Management.”

The 601’s BAT registers are unique among the four processor implementations discussed in this book. While the SPR number for each of the 601’s BATs corresponds to IBATs in the PowerPC architecture specification, the
601’s BAT registers are used as unified BATs — they map both instruction and data memory. The 601 has eight unified BAT registers. The 603, 604, and 620 have eight instruction BAT registers and eight data BAT registers.

On all processor implementations, BAT registers are used in pairs: an upper and lower BAT register. The 601 has eight BAT registers in four upper/lower pairs. Thus, the 601 is capable of mapping four separate blocks of memory using the BAT mechanism.

As shown in Figure 4-20, the bit definitions of the 601’s BAT are unique. The 601’s BATs can map, at most, 8MB of memory with a single BAT pair. Other implementations can map as much as 256MB with a single pair of upper/lower BAT registers. Table 4-12 lists bit definitions for the 601’s BAT register.

<table>
<thead>
<tr>
<th>The Upper Block Address Translation (BAT) Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 14 15</td>
</tr>
<tr>
<td>Block Logical Page Index (BLPI)</td>
</tr>
</tbody>
</table>

Note: Key is either Ks or Ku depending on value of MSR[PR] (privilege level).

<table>
<thead>
<tr>
<th>The Lower Block Address Translation (BAT) Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 14 15 18 19</td>
</tr>
<tr>
<td>Physical Block Number (PBN)</td>
</tr>
<tr>
<td>Block Length (BSM)</td>
</tr>
<tr>
<td>128K</td>
</tr>
<tr>
<td>256K</td>
</tr>
<tr>
<td>512K</td>
</tr>
<tr>
<td>1M</td>
</tr>
<tr>
<td>2M</td>
</tr>
<tr>
<td>4M</td>
</tr>
<tr>
<td>8M</td>
</tr>
</tbody>
</table>

Figure 4-20
The 601’s upper and lower block address translation (BAT) registers implement virtual memory and memory protection.
Table 4-12
The 601 BAT Register’s Bit Definitions

<table>
<thead>
<tr>
<th>Upper/Lower Register</th>
<th>Bit Position</th>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| Upper                 | 0–14         | BLPI     | Block Logical Page Index  
This field determines if an address hits in the BAT array. To determine a hit, BLPI is compared with bits 0–14 of the logical address. |
| Upper                 | 15–24        | —        | Reserved |
| Upper                 | 25–27        | WIM      | Memory/Cache Access Mode Bits  
W = Write-through  
I = Caching inhibited  
M = Memory coherence |
| Upper                 | 28           | Ks       | Supervisor Mode Key  
As shown in Figure 4-20, this bit (along with the PP bits) determines the protection of the block. |
| Upper                 | 29           | Ku       | User Mode Key  
As shown in Figure 4-20, this bit (along with the PP bits) determines the protection of the block. |
| Upper                 | 30–31        | PP       | Protection Bits for Block  
This field determines the access protection for the block. As shown in Figure 4-20, PP and the supervisor/user key are used together. |
| Lower                 | 0–14         | PBN      | Physical Block Number  
This field generate bits 0–14 of the physical address of the block. |
| Lower                 | 15–24        | —        | Reserved |
| Lower                 | 25           | V        | Valid Bit  
When V = 1, the BAT register pair is valid. |
| Lower                 | 26–31        | BSM      | Block Size Mask  
BSM is a mask that encodes the size of the block. Bit values and their meanings are shown in Figure 4-20. |
Checkstop Sources and Enables Register

The supervisor-level, 32-bit checkstop sources and enables (HIDO) register enables and monitors the cause of a checkstop condition. (See the sidebar “What is a Checkstop?”) As shown in Figure 4-21, HIDO defines bits that are used to enable and decode the source of a checkstop event. Note that this 601 register is a processor hardware implementation-dependent (HID) register; this register enables checkstop sources specific to the PowerPC 601 processor.

HIDO[0] is the master checkstop enable bit; if it is cleared, all checkstops are disabled. If HIDO[0] is set, bits HIDO[15-31] are used to enable and disable individual checkstop sources for debugging purposes. Table 4-13 defines all the bits of the HIDO register and their initial power-on values. Note that moving a value into the HIDO register using the mtspr instruction will not cause a checkstop.

![Figure 4-21](image)

The 601's checkstop sources and enable register (HIDO) enables and monitors machine check conditions.
What is a Checkstop?

On PowerPC microprocessors, there is a built-in facility to halt processor operation in the event of a serious failure. This facility is the checkstop state. In the event of a failure, the processor sets bits in the HID0 register corresponding to the cause of the failure. The processor then halts operation by stopping all internal clocks. Only the PowerPC 601 has such control of the checkstop state; the 603, 604, and 620 do not have the functional equivalent of the 601's HID0 register.

Table 4-13
Checkstop Sources and Enable (HIDO) Register Bit Definitions

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Name</th>
<th>Power-On State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CE</td>
<td>1</td>
<td>Master checkstop enabled if set (=1)</td>
</tr>
<tr>
<td>1</td>
<td>S</td>
<td>0</td>
<td>Microcode checkstop detected if set</td>
</tr>
<tr>
<td>2</td>
<td>M</td>
<td>0</td>
<td>Double machine checkstop detected if set</td>
</tr>
<tr>
<td>3</td>
<td>TD</td>
<td>0</td>
<td>Multiple TLB hit checkstop detected if set</td>
</tr>
<tr>
<td>4</td>
<td>CD</td>
<td>0</td>
<td>Multiple cache hit checkstop detected if set</td>
</tr>
<tr>
<td>5</td>
<td>SH</td>
<td>0</td>
<td>Sequencer time out checkstop detected if set</td>
</tr>
<tr>
<td>6</td>
<td>DT</td>
<td>0</td>
<td>Dispatch time out checkstop detected if set</td>
</tr>
<tr>
<td>7</td>
<td>BA</td>
<td>0</td>
<td>Bus address parity error checkstop detected if set</td>
</tr>
<tr>
<td>8</td>
<td>BD</td>
<td>0</td>
<td>Bus data parity error if set</td>
</tr>
<tr>
<td>9</td>
<td>CP</td>
<td>0</td>
<td>Cache parity error if set</td>
</tr>
<tr>
<td>10</td>
<td>IU</td>
<td>0</td>
<td>Invalid microcode instruction if set</td>
</tr>
<tr>
<td>11</td>
<td>PP</td>
<td>0</td>
<td>I/O controller protocol error if set</td>
</tr>
<tr>
<td>12-14</td>
<td></td>
<td>0</td>
<td>Reserved</td>
</tr>
<tr>
<td>15</td>
<td>ES</td>
<td>1</td>
<td>Enable Microcode Checkstop</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If set and the processor detects that a bad microcode has been fetched from the processor's internal ROM, a checkstop condition occurs and bit HID0[S] is set.</td>
</tr>
<tr>
<td>16</td>
<td>EM</td>
<td>0</td>
<td>Enable Machine Check Checkstop</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If set and the processor's TEA# (Transfer Error Acknowledge) signal is asserted, a checkstop occurs and bit HID[M] is set.</td>
</tr>
</tbody>
</table>
Table 4-13.
Checkstop Sources and Enable (HIDO) Register Bit Definitions (Continued)

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Name</th>
<th>Power-On State</th>
<th>Description</th>
</tr>
</thead>
</table>
| 17           | ETD  | 0              | Enable TLB Checkstop  
If set and there is a double hit in the translation lookaside buffer during a page table lookup, a checkstop occurs and bit HIDO[TD] is set. |
| 18           | ECD  | 0              | Enable Cache Checkstop  
If set and there is a double hit in the internal cache, a checkstop occurs and bit HIDO[CD] is set. |
| 19           | ESH  | 0              | Enable Sequencer Time-out Checkstop  
If set and processor’s sequencer times out, a checkstop occurs and bit HIDO[SH] is set. |
| 20           | EDT  | 0              | Enable Dispatch Time-out Checkstop  
If set and processor’s dispatcher times out, a checkstop occurs and bit HIDO[DT] is set. |
| 21           | EBA  | 0              | Enable Address Bus Parity Checkstop  
If set and there is a parity error while the processor is snooping an address, a checkstop occurs and bit HIDO[BA] is set. |
| 22           | EBD  | 0              | Enable Data Bus Parity Checkstop  
If set and there is a parity error when the processor is reading data from an external device, a checkstop occurs and bit HIDO[BD] is set. |
| 23           | ECP  | 0              | Enable Cache Parity Checkstop  
If set and there is a parity error when the internal cache reads from the cache directory (or data storage area), a checkstop occurs and bit HIDO[CP] is set. |
| 24           | EIU  | 1              | Enable Invalid Microcode Instruction Checkstop  
If set and bad microcode is read from the processor’s internal ROM, a checkstop occurs and bit HIDO[IU] is set. |
Table 4-13.
Checkstop Sources and Enable (HIDO) Register Bit Definitions Continued

<table>
<thead>
<tr>
<th>Bit position</th>
<th>Name</th>
<th>Power-On State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>EPP</td>
<td>0</td>
<td>Enable I/O Controller Checkstop. If set and the processor detects an I/O error during an I/O bus transaction, a checkstop occurs and bit HIDO[PP] is set.</td>
</tr>
<tr>
<td>26</td>
<td>DRF</td>
<td>0</td>
<td>1 = Optional reload of alternate sector on instruction fetch miss is disabled.</td>
</tr>
<tr>
<td>27</td>
<td>DRL</td>
<td>0</td>
<td>Optional reload of alternate sector on load/store miss is enabled (1) or disabled (0).</td>
</tr>
<tr>
<td>28</td>
<td>IMM</td>
<td>0</td>
<td>Big endian mode (0) or little endian mode (1) is enabled.</td>
</tr>
<tr>
<td>29</td>
<td>PAR</td>
<td>0</td>
<td>Precharge of the ARTRY# and SHD# signals is enabled (0) or disabled (1).</td>
</tr>
<tr>
<td>30</td>
<td>EMC</td>
<td>0</td>
<td>Error (1) or no error (0) was detected in main cache during array initialization.</td>
</tr>
<tr>
<td>31</td>
<td>EHP</td>
<td>0</td>
<td>The HP_SNP_REQ# signal is enabled (1) or disabled (0).</td>
</tr>
</tbody>
</table>

**Debug Modes Register**

The supervisor-level debug modes (HID1) register enables the various debug modes on the 601. Figure 4-22 shows the bit definitions of the HID1 register. The 601 is the only PowerPC implementation that has a debug modes register.

The 601 uses the run mode exception (HID1[RM]=0b10) bit field much like the IABR and DABR. The 601’s run mode exception condition is described fully in Chapter 10, “Exceptions and Interrupts.” Setting the run mode exception bit (HID1[RM]=0b10) may produce unpredictable results if used when single-step mode is enabled (HID1[M]=0b100). It is possible to hang the 601 processor in an infinite loop by setting the single instruction step option in the M field (0b100) and the run mode exception option in the RM field (0b10).
When the TL bit is set, the tblie instruction does not broadcast on the system bus.

<table>
<thead>
<tr>
<th>601 Run Modes</th>
<th>Response to address compare or single step</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 Normal run mode</td>
<td>00 Hard stop</td>
</tr>
<tr>
<td>001 Undefined - DO NOT USE</td>
<td>01 Soft stop (wait for system activity to quiesce)</td>
</tr>
<tr>
<td>010 Limited instruction address compare</td>
<td>10 Trap to run mode exception</td>
</tr>
<tr>
<td>011 Undefined - DO NOT USE</td>
<td>11 Reserved - DO NOT USE</td>
</tr>
<tr>
<td>100 Single instruction step</td>
<td></td>
</tr>
<tr>
<td>101 Undefined - DO NOT USE</td>
<td></td>
</tr>
<tr>
<td>110 Full instruction address compare</td>
<td></td>
</tr>
<tr>
<td>111 Full branch target address compare</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4-22
The 601’s debug modes register (HID1) configures the processor’s run modes.

**Instruction Address Breakpoint Register**

The supervisor-level 32-bit *instruction address breakpoint register* (IABR) (also known as HID2) implements breakpoints for instruction and branch target addresses. The IABR is loaded with an effective address that is compared against the address of each instruction executed; if a match is found, the results depend on the setting of the 601’s debug modes register (HID1[M]). The 601’s IABR is shown in Figure 4-23.

<table>
<thead>
<tr>
<th>IABR (or HID2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 31</td>
</tr>
</tbody>
</table>

Figure 4-23
The 601’s instruction address breakpoint register (IABR or HID2) sets instruction breakpoints.
To generate a breakpoint exception, either the address of the current instruction or the address of a branch target must match the effective address contained in the IABR. The difference between an instruction address match and a branch address match depends on the mode specified by the value of HID1[M].

**Processor Identification Register**

The supervisor-level, 32-bit *processor identification register* (PIR) (also known as HID15) identifies a particular processor in multiprocessor systems. A 4-bit processor ID (PIR[PID]) can be loaded and read using the *mtspr* and *mfspr* instructions. The layout of the PIR is shown in Figure 4-24.

```
+--------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+---
### 603 Supervisor-Level Model

**Supervisor-Level Exception Handling Registers**

- SPR18: DSISR-DAE/ Source Instruction Service Register
- SPR19: DAR - Data Address Register
- SPR28: SRR0 - Save and Restore Register 0
- SPR27: SRR1 - Save and Restore Register 1
- SPR272: SPRG0 - SPR General Register 0
- SPR273: SPRG1 - SPR General Register 1
- SPR274: SPRG2 - SPR General Register 2
- SPR275: SPRG3 - SPR General Register 3

**Supervisor-Level Memory Management Registers**

**Instruction BAT Registers**

- SPR528: IBAT0U - BAT 0 Upper Register
- SPR529: IBAT0L - BAT 0 Lower Register
- SPR530: IBAT1U - BAT 1 Upper Register
- SPR531: IBAT1L - BAT 1 Lower Register
- SPR532: IBAT2U - BAT 2 Upper Register
- SPR533: IBAT2L - BAT 2 Lower Register
- SPR534: IBAT3U - BAT 3 Upper Register
- SPR535: IBAT3L - BAT 3 Lower Register

**Data BAT Registers**

- SPR536: DBAT0U - BAT 0 Upper Register
- SPR537: DBAT0L - BAT 0 Lower Register
- SPR538: DBAT1U - BAT 1 Upper Register
- SPR539: DBAT1L - BAT 1 Lower Register
- SPR540: DBAT2U - BAT 2 Upper Register
- SPR541: DBAT2L - BAT 2 Lower Register
- SPR542: DBAT3U - BAT 3 Upper Register
- SPR543: DBAT3L - BAT 3 Lower Register

**Segment Registers**

- SPR25: SDR1 - Table Search Description Register 1

**Software Table Search Registers**

- SPR976: DMISS - Data TLB Miss Address Register
- SPR977: DCMP - Data TLB Compare Register
- SPR978: HASH1 - Primary Hash Address Register
- SPR979: HASH2 - Secondary Hash Address Register
- SPR980: IMMSS - Instruction TLB Miss Address Register
- SPR981: ICMP - Instruction TLB Compare Register
- SPR982: RPA - Required Physical Address

**Configuration Registers**

- SPR1008: HID0 - Checkstop Sources and Enable Register
- SPR287: PVR - Processor Version Register

**Miscellaneous Registers**

- SPR284: TBL - Time Base Facility Lower (Writing)
- SPR285: TBU - Time Base Facility Upper (Writing)
- SPR1010: IABR - Instruction Address Breakpoint Register
- SPR22: DEC - Decrementer
- SPR282: EAR - External Address Register (Optional)

### 603 User-Level Model

**User Instruction Set Architecture**

**General-Purpose Registers**

- SPR0: GPR0
- SPR31: GPR31

**Floating-Point Registers**

- SPR0: FPR0
- SPR31: FPR31

**Condition Register**

- SPR0: CR

**Floating-Point Status and Control Register**

- SPR0: FPSCR

**User Virtual Environment Architecture**

- SPR1: XER - Integer Exception Register
- SPR8: LR - Link Register
- SPR9: CTR - Count Register

**Configuration Registers**

- SPR268: TBL - Time Base Facility Lower (Reading)
- SPR269: TBU - Time Base Facility Upper (Reading)

---

**Figure 4-25**

The 603's programming model is influenced by the processor's power-conscious design.
Unlike the other PowerPC processors, the 603 does not implement a processor identification register (PIR) for multiprocessor identification purposes. This means that the 603 is not as inherently equipped for multiprocessor support as the other PowerPC implementations.

**Block Address Translation Registers**

The 603, as all current PowerPC processors, uses BAT registers as one means of implementing virtual memory. Unlike the 601, the 603, 604, and 620 have both instruction BAT (IBAT) registers and data BAT (DBAT) registers as specified by the PowerPC architecture. The upper BAT (BATU) and lower BAT (BATL) registers are shown in Figure 4-26. The four IBAT and

<table>
<thead>
<tr>
<th>The Upper Block Address Translation (BAT) Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31</td>
</tr>
<tr>
<td>BEPI</td>
</tr>
<tr>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

Block Length Bits

- 128Kb: 0000 0000 0000
- 256Kb: 0000 0000 0001
- 512Kb: 0000 0000 0011
- 1Mb: 0000 0000 0111
- 2Mb: 0000 0000 1111
- 4Mb: 0000 0001 1111
- 8Mb: 0000 0011 1111
- 16Mb: 0000 0111 1111
- 32Mb: 0000 1111 1111
- 64Mb: 0001 1111 1111
- 128Mb: 0111 1111 1111
- 256Mb: 1111 1111 1111

<table>
<thead>
<tr>
<th>The Lower Block Address Translation (BAT) Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31</td>
</tr>
<tr>
<td>BRPN</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

PP Access Allowed

- 00: No access
- 01: Read only
- 10: Read/write

![Figure 4-26](image)
The 603, 604, and 620 processors use the same upper and lower block address translation registers.
four DBAT registers are used in pairs. Table 4-14 shows the bit field definitions for these registers.

**Table 4-14**
The 603, 604, and 620 BAT Registers Bit Definitions

<table>
<thead>
<tr>
<th>Upper/Lower Register</th>
<th>Bit Position</th>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upper</td>
<td>0-14</td>
<td>BEPI</td>
<td>Block Logical Page Index. This field specifies part of an address. It determines if a logical address hits in the BAT array. To determine a hit, BEPI is compared with bits 0-14 of the logical address.</td>
</tr>
<tr>
<td>Upper</td>
<td>15-18</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>Upper</td>
<td>19-29</td>
<td>BL</td>
<td>Block Size Mask. Encodes the size of the block mapped by the BAT. Bit values and their meanings are shown in Figure 4-26.</td>
</tr>
<tr>
<td>Upper</td>
<td>30</td>
<td>Vs</td>
<td>Supervisor Mode Valid Bit. In conjunction with MSR[PR], determines a logical address match condition.</td>
</tr>
<tr>
<td>Upper</td>
<td>31</td>
<td>Vp</td>
<td>User Mode Valid Bit. In conjunction with MSR[PR], determines a logical address match condition.</td>
</tr>
<tr>
<td>Lower</td>
<td>0-14</td>
<td>BRPN</td>
<td>Physical Block Number. In conjunction with the BATU[BL] field, generates the high order bits (0-14) of the block’s physical address.</td>
</tr>
<tr>
<td>Lower</td>
<td>15-24</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>Lower</td>
<td>25-28</td>
<td>WIMG</td>
<td>Memory/Cache Access Mode Bits. W = Write-through, I = Caching inhibited, M = Memory coherence, G = Guarded (DBATs only — not defined for IBATs)</td>
</tr>
<tr>
<td>Lower</td>
<td>29</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>Lower</td>
<td>30-31</td>
<td>PP</td>
<td>Protection Bits for Block. Determines the access protection for the block.</td>
</tr>
</tbody>
</table>
Let's work through how the 603 (as well as the 604 and 620) would determine a logical address match (a BAT hit) in a BAT-mapped region of memory. The determination of a BAT hit would take place every time the processor performs a load/store operation in an area of memory that is using address translation.

For our example, let's assume that we're using a DBAT (data memory BAT) that maps 16MB of memory from 0xF8000000 through 0xF8FFFFFF. First, we load BATU[BEPI] with the high-order 14 bits of the value 0xF800 — the two lowest-order bits are ignored — to specify the start of the memory region. Next, we load BATU[BL] with 0b00001111111 (= 0x07f) to specify the 16MB size of the region. Figure 4-26 lists the bit settings for various block lengths.

BATU[BEPI] contains the high-order bits that are compared with the logical address to determine a match. BATU[BL] specifies the size of the memory region mapped by the BAT register as well as the number of bits involved in the comparison between BEPI and the logical address. Every comparison involves the most-significant nibble (bits 0–3). The number of additional bits in the address [4–14] that are involved is dependent on the number of zero bits contained in BL.

In this example, BL contains 4 zero bits (BL = 0b00001111111). So any comparison between BEPI and a logical address involves bits 0–7 — always the most-significant nibble plus the 4 corresponding to the zero bits in BL. In this particular case, the following compare would take place (‘n’ values are not involved in the compare operation):

\[
\begin{align*}
0xf8nnnnnn & \quad \text{- 8 bits from address in BATU[BEPI]} \\
\text{compare} & \quad 0xf8000000 \quad \text{- our example logical address}
\end{align*}
\]

Clearly, we've got a match on our hands. The offset into the BAT region is specified by the bits in the logical address that were not part of the compare: 0x020000. The value that is loaded into BEPI (and BRPN) must have at least as many low-order 0 bits as there are 1 bits in BL. Otherwise, the number of bits used in the BEPI-logical address comparison would be insufficient to determine a match. Further explanation of BAT register operation can be found in Chapter 8, “Memory Management.”

**Hardware Implementation Register**

The supervisor-level, 32-bit hardware implementation 0 (HID0) register enables the implementation-specific features of the 603. Specifically, the
603’s power management bits are contained in the HID0 register. HID0[DOZE,NAP,SLEEP] put the 603 into various levels of power conservation. Figure 4-27 shows the 603 HID0 and Table 4-15 defines the HID0 bits. Note that some of the bits in the 603’s HID0 register correspond to signals on processor pins. These hardware references are discussed only when specifically useful to programmers.

**Figure 4-27**
The 603’s hardware implementation register 0 (HID0) is used primarily to enable checkpoint conditions.

**Memory Paging and Data Structures**
The PowerPC 603 manages memory paging and data structures associated with memory paging in software using the registers shown in Figure 4-28 and the PowerPC exception mechanism — something other PowerPC implementations do in hardware. Consequently, the 603 has a unique set of supervisor-level registers that are not found on other PowerPC implementations. In the following sections, we’ll briefly look at these registers. For additional information on memory paging and the 603’s paging implementation, refer to Chapter 8, “Memory Management.”
Table 4-15
The 603’s HIDO Bit Definitions

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>EMCP</td>
<td>Enable Machine Check Pin</td>
</tr>
<tr>
<td>1</td>
<td>—</td>
<td>Not used</td>
</tr>
<tr>
<td>2</td>
<td>EBA</td>
<td>Enable Bus Address Parity Checking</td>
</tr>
<tr>
<td>3</td>
<td>EBD</td>
<td>Enable Bus Data Parity Checking</td>
</tr>
<tr>
<td>4</td>
<td>SBCLK</td>
<td>Select Bus Clock for Test Clock Pin</td>
</tr>
<tr>
<td>5</td>
<td>EICE</td>
<td>Enable ICE (in-circuit emulation) Outputs</td>
</tr>
<tr>
<td>6</td>
<td>ECLK</td>
<td>Enable External Test Clock Pin</td>
</tr>
<tr>
<td>7</td>
<td>PAR</td>
<td>Disable Precharge #ATRTY and Shared Signals</td>
</tr>
<tr>
<td>8</td>
<td>DOZE</td>
<td>Doze Mode. PLL (phase locked loop), time base, and snooping remain active</td>
</tr>
<tr>
<td>9</td>
<td>NAP</td>
<td>Nap Mode. PLL and time base remain active</td>
</tr>
<tr>
<td>10</td>
<td>SLEEP</td>
<td>Sleep Mode. No external clock required</td>
</tr>
<tr>
<td>11</td>
<td>DPM</td>
<td>Enable Dynamic Power Management</td>
</tr>
<tr>
<td>12</td>
<td>RISEG</td>
<td>Reserved for test</td>
</tr>
<tr>
<td>13-14</td>
<td>—</td>
<td>Not used</td>
</tr>
<tr>
<td>15</td>
<td>NHR</td>
<td>Reserved</td>
</tr>
<tr>
<td>16</td>
<td>ICE</td>
<td>Instruction Cache Enable</td>
</tr>
<tr>
<td>17</td>
<td>DCE</td>
<td>Data Cache Enable</td>
</tr>
<tr>
<td>18</td>
<td>ILOCK</td>
<td>Instruction Cache LOCK</td>
</tr>
<tr>
<td>19</td>
<td>DLOCK</td>
<td>Data Cache LOCK</td>
</tr>
<tr>
<td>20</td>
<td>ICFI</td>
<td>Instruction Cache Flash Invalidate</td>
</tr>
<tr>
<td>21</td>
<td>DCI</td>
<td>Data Cache Flash Invalidate</td>
</tr>
<tr>
<td>22-26</td>
<td>—</td>
<td>Not used</td>
</tr>
<tr>
<td>27</td>
<td>FBIOB</td>
<td>Force Branch Indirect on Bus</td>
</tr>
<tr>
<td>28-30</td>
<td>—</td>
<td>Not used</td>
</tr>
<tr>
<td>31</td>
<td>NOOPTI</td>
<td>No-op touch instructions</td>
</tr>
</tbody>
</table>

Data and Instruction Miss Address Registers

The supervisor-level, 32-bit data and instruction miss address registers (DMISS and IMISS) are part of the 603’s software page table search mechanism. Both registers are read-only and have the same format, as shown in Figure 4-28. The DMISS and IMISS registers are loaded automatically upon a data or instruction translation lookaside buffer (TLB) miss exception. Values contained in the DMISS or IMISS registers are used to calculate the HASH1 and HASH2 values as well as by the tlbld and tlbli instructions.
The 603's Data and Instruction TLB Miss Address Registers (DMISS, IMISS)
Note: Both registers have the same format.

<table>
<thead>
<tr>
<th>DMISS/IMISS Page Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>31</td>
</tr>
</tbody>
</table>

The 603's Data and Instruction TLB Compare Registers (DCMP, ICMP)
Note: Both registers have the same format.

<table>
<thead>
<tr>
<th>V</th>
<th>VSID</th>
<th>H</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td></td>
<td>26</td>
<td>31</td>
</tr>
</tbody>
</table>

The 603's Primary and Secondary Hash Address Registers (HASH1, HASH2)
Note: Both registers have the same format.

<table>
<thead>
<tr>
<th>HTABORG[0-6]</th>
<th>Hashed Page Address</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>25</td>
<td>26</td>
<td>31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The 603's Required Physical Address Register (RPA)

<table>
<thead>
<tr>
<th>RPN</th>
<th>0 0 R C</th>
<th>WIMG</th>
<th>0 PP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>19 20 22 23 24 25</td>
<td>28 29 30 31</td>
<td></td>
</tr>
</tbody>
</table>

| Reserved |

**Figure 4-28**
The 603 implements software-based page table management using the DMISS, IMISS, DCMP, ICMP, HASH1, HASH2, and RPA registers.

Note that the DMISS register is always loaded with a big endian address, even when the processor is operating in little endian mode (MSR[LE]=1). This agrees with our understanding (in Chapter 3, “Of Eggs and Endians”) that PowerPC processors operating in little endian mode translate (or munge) the address of the memory operand — not the data contained in the memory operand.
Data and Instruction TLB Compare Registers

The supervisor-level, 32-bit data and instruction TLB compare registers (DCMP and ICMP) are part of the 603's software page table search mechanism. These two registers contain the first word in the required PTE (page table entry). When a TLB miss exception occurs, the DCMP and ICMP entries are constructed from the contents of the segment registers and the contents of the DMISS or IMISS registers.

Each PTE read from the page tables during the table search process should be compared with this value to determine a PTE match. Upon execution of a tlbld or tlvli instruction, the DCMP and ICMP register is loaded into the first word of the selected TLB entry. Figure 4-28 shows the DCMP and ICMP registers; Table 4-16 shows their bit definitions.

Table 4-16
The DCMP and ICMP Register Bit Definitions

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>V</td>
<td>Valid Bit. Set by the processor when a TLB miss exception occurs.</td>
</tr>
<tr>
<td>1-24</td>
<td>VSID</td>
<td>Virtual Segment ID. Copied from the VSID field of the corresponding segment register.</td>
</tr>
<tr>
<td>25</td>
<td>H</td>
<td>Hash Function Identifier. Cleared by the processor when a TLB miss exception occurs.</td>
</tr>
<tr>
<td>26-31</td>
<td>API</td>
<td>Abbreviated Page Index. Copied from the API of the effective address.</td>
</tr>
</tbody>
</table>

Primary and Secondary Hash Address Registers (HASH1, HASH2)

The supervisor-level, 32-bit primary and secondary hash address registers (HASH1 and HASH2) are part of the 603's software page table search mechanism. These registers contain the physical address of the primary and secondary PTEGs for the access that caused the TLB miss exception. Only bits 7-25 differ between them. For convenience, the 603 automatically constructs the full physical address by routing bits 0-6 of SDR1 into HASH1 and HASH2 and clearing bits 26-31. These registers are read-only and constructed from the contents of the DMISS and IMISS registers. The HASH1 and HASH2 registers are shown in Figure 4-28; Table 4-17 shows the bit definitions.
The PowerPC Programming Model

Table 4-17

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-6</td>
<td>HTABORG[0-6]</td>
<td>This field is a copy of the upper 7 bits of the HTABORG field from the SDR1 register.</td>
</tr>
<tr>
<td>7-25</td>
<td>Hashed page address</td>
<td>This field contains the address bits (7-25) of the PTEG (page table entry group) to be searched.</td>
</tr>
<tr>
<td>26-31</td>
<td>—</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Required Physical Address Register (RPA)

The supervisor-level, 32-bit required physical address register (RPA) is part of the 603’s software page table search mechanism. When performing a page table search operation, software must load the RPA with the second word of the correct page table entry (PTE). When the tlbld or tlbli instruction is executed, the contents of the RPA register is merged with the DMISS and IMISS registers and loaded into the selected TLB entry. The RPA register is shown in Figure 4-28 and the bit definitions are described in Table 4-18.

Table 4-18

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-19</td>
<td>RPN</td>
<td>Physical page number from PTE</td>
</tr>
<tr>
<td>20-22</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td>23</td>
<td>R</td>
<td>Referenced bit from PTE</td>
</tr>
<tr>
<td>24</td>
<td>C</td>
<td>Changed bit from PTE</td>
</tr>
<tr>
<td>25-28</td>
<td>WIMG</td>
<td>Memory/cache access attribute bits</td>
</tr>
<tr>
<td>29</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td>30-31</td>
<td>PP</td>
<td>Page protection bits from PTE</td>
</tr>
</tbody>
</table>

Instruction Address Breakpoint Register (IABR)

The supervisor-level instruction address breakpoint register (IABR) is shown in Figure 4-29. This 32-bit register controls the instruction address breakpoint exception on the 603. IABR[CEA] contains an effective address to
which each instruction address is compared. Setting IABR[IE] enables the exception; when an exception is taken, the instruction that causes the exception will not be completed before the exception handler is entered.

![Figure 4-29](image.png)
The 603's instruction address breakpoint register (IABR) differs from the IABR on the 604 and 620 in bit 31.

**PowerPC 604 Register Set**

Because the 603, 604, and 620 all implement the PowerPC architected programming model, they have very similar register sets. In fact, there are few differences between the 603 and 604 register sets; even fewer between the 604 and 620. When register use or function differs between the 603 and 604, we will re-examine the register in detail.

The 604 programming model is shown in Figure 4-30. The majority of 604 differences are in the implementation-specific registers. Of the OEA register set, only the MSR and DEC have 604-specific functionality.

**Machine State Register**

As shown previously in Figure 4-6, bit 29 of the MSR is the 604 and 620 performance monitor (PM) bit. Other PowerPC implementations treat MSR[PM] as a reserved field. The MSR[PM] bit is used in conjunction with the other 604 performance monitoring registers, discussed in the following sections. When MSR[PM] is set, the process that is currently running is considered marked. This bit is used to distinguish among multiple processes that may be running on a multitasking system.
The PowerPC 604 programming model represents the most advanced 32-bit PowerPC implementation to date.
Decrementer Register

The 604's supervisor-mode DEC register, shown previously in Figure 4-12, is functionally equivalent to the DEC on the 603. However, on the 604, the DEC register is always decremented at one-fourth the speed of the bus clock.

Figure 4-31
The 604's hardware implementation-dependent 0 (HIDO) register enables checkstop conditions and advanced processor features.

Hardware Implementation-Dependent Register

The 604's hardware implementation-dependent 0 (HIDO) register is similar to that found on the 603 and is shown in Figure 4-31. However, the 604's HIDO contains fields, such as PAR and BHTE, that are not present on other implementations. Table 4-19 summarizes the bits that are defined for the 604's HIDO.

Table 4-19
The 604's HIDO Bit Definitions

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>EMCP</td>
<td>Enable Machine Check Input Pin</td>
</tr>
<tr>
<td>1</td>
<td>ECP</td>
<td>Enable Cache Parity Checking</td>
</tr>
<tr>
<td>2</td>
<td>EBA</td>
<td>Enable Machine Check on Address Bus Parity Error</td>
</tr>
<tr>
<td>3</td>
<td>EBD</td>
<td>Enable Machine Check on Data Bus Parity Error</td>
</tr>
<tr>
<td>4-6</td>
<td></td>
<td>Reserved</td>
</tr>
</tbody>
</table>
Table 4-19
The 604’s HIDO Bit Definitions (Continued)

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>PAR</td>
<td>Disable Snoop Response High State Restore</td>
</tr>
<tr>
<td>8–14</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td>15</td>
<td>NHR</td>
<td>Not Hard Reset</td>
</tr>
<tr>
<td>16</td>
<td>ICE</td>
<td>Instruction Cache Enable</td>
</tr>
<tr>
<td>17</td>
<td>DCE</td>
<td>Data Cache Enable</td>
</tr>
<tr>
<td>18</td>
<td>ILOCK</td>
<td>Instruction Cache Lock</td>
</tr>
<tr>
<td>19</td>
<td>DLOCK</td>
<td>Data Cache Lock</td>
</tr>
<tr>
<td>20</td>
<td>ICFI</td>
<td>Instruction Cache Invalidate All</td>
</tr>
<tr>
<td>21</td>
<td>DCI</td>
<td>Data Cache Invalidate All</td>
</tr>
<tr>
<td>22–23</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td>24</td>
<td>SIED</td>
<td>Serial Instruction Execution Disable</td>
</tr>
<tr>
<td>25–28</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td>29</td>
<td>BHT</td>
<td>Branch History Table Enable</td>
</tr>
<tr>
<td>30–31</td>
<td>—</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

0 = The 604 executes one instruction at a time. The 604 does not post a trace exception after each instruction completes as it would if MSR[SE] or MSR[BE] were set.

1 = Instruction execution is not serialized.

Instruction Address Breakpoint Register

With the exception of the IABR[TE] bit, the 604’s supervisor-level IABR is functionally similar to that found on the 603. The IABR for the 604 and 620 processors is shown in Figure 4-32.
On the 604, there is an additional prerequisite for an IABR match condition. In particular, IABR[TE] must match the state of MSR[IR]; this means that instruction breakpoints are sensitive to the state of address translation (enabled or disabled). Table 4-20 defines the bits within the 604’s IABR.

### Table 4-20
The 604’s IABR Bit Definitions

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-29</td>
<td>CEA</td>
<td>Word Address to be Compared. (32-bit aligned)</td>
</tr>
<tr>
<td>30</td>
<td>BE</td>
<td>Breakpoint Enabled. When this bit is set, IABR breakpoint checking is active.</td>
</tr>
<tr>
<td>31</td>
<td>TE</td>
<td>Translation Enabled. An IABR match is signaled only when IABR[TE] = MSR[IR].</td>
</tr>
</tbody>
</table>

### Performance Monitoring Registers

The 604 and 620 PowerPC processors have built-in performance monitoring capabilities. The five registers that comprise this powerful feature are shown in Figure 4-33. The ability to debug and optimize software performance using the monitoring-capable PowerPC processors is discussed in Chapter 12, “Techniques and Tricks.”

The five supervisor-level registers that constitute the performance monitoring facility are as follows:

- The monitor *mode control register 0* (MMCR0) specifies the conditions that cause a performance monitoring exception.
- The *performance monitor counter 1 and 2* (PMC1 and PMC2) registers count various iterative events that relate to performance monitoring. When PMC1 or PMC2 reaches its maximal count, a performance monitoring exception is generated if enabled in MMCR0.
The sampled instruction address (SIA) and sampled data address (SDA) registers hold the address of the data or instruction that caused a performance monitor exception.

The PowerPC Programming Model

The Monitor Mode Control Register 0 (MMCR0)

The Performance Monitor Counter 1 and 2 (PMC1, PMC2)

Note: Both registers have the same format.

The Sampled Instruction and Data Address Registers (SIA, SDA)

Note: Both registers have the same format.

Figure 4-33

The performance monitoring registers are present only on the 604 and 620.

Performance Monitor Control Register

The performance monitor control register (MMCR0) is a 32-bit supervisor-level register that specifies the events to be used during performance statistic gathering. Shown in Figure 4-33, the MMCR0 enables and disables various counting and monitoring functionality. The bit field definitions for the MMCR0 register are given in Table 4-21.

Using the MMCR0[PCM1-SELECT] and MMCR0[PCM2-SELECT] fields, software is able to monitor an impressive array of processor events. The events that are able to be monitored are described in Tables 4-22 and 4-23. The ability to select two separate events allows complex statistics to be gathered about processor and system performance. Chapter 12, "Techniques and Tricks," discusses performance monitoring in detail.
## Table 4-21
The 604's MMCRO Bit Definitions

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0            | DIS   | Disable counting unconditionally.  
              |       | 1 = The values of the PMCnt counters cannot be changed by the processor's performance monitoring hardware. |
| 1            | DP    | Disable counting while in supervisor mode.  
              |       | 1 = If the processor is in supervisor mode, the counters are not changed by hardware. |
| 2            | DU    | Disable counting while in user mode.  
              |       | 1 = If the processor is in user mode, the counters are not changed by hardware. |
| 3            | DMS   | Disable counting while MSR[PM] is set.  
              |       | 1 = If MSR[PM] is set, the PMCnt counters are not changed by the processor's PM hardware. |
| 4            | DMR   | Disable counting while MSR[PM] is cleared.  
              |       | 1 = If MSR[PM] is cleared, the PMCnt counters are not changed by the processor's PM hardware. |
| 5            | ENINT | Enable performance monitor interrupt signaling.  
              |       | 1 = Interrupt signaling is enabled.  
              |       | Note that this bit is cleared by hardware when a performance monitor interrupt is signaled. To re-enable these interrupts, software must set this bit after handling the PM interrupt. Typically, the Initial Program Load (IPL) ROM code clears this bit before passing control to the OS. |
| 6            | DISCOUNT | Disable counting of PMC1 and PMC2 when a performance monitor interrupt is signaled.  
              |       | 1 = The signaling of a performance monitoring interrupt prevents the changing of the PMC1 counter.  
              |       | The PMC2 counter will not change if PMC2COUNTCTL = 0.  
              |       | Because a Time Base signal could have occurred along with an enabled PM counter interrupt, software should always reset INTONBITTRANS to 0 if the value in INTONBITTRANS was 1. |
### Table 4-21
The 604’s MMCRO Bit Definitions (Continued)

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7–8</td>
<td>RTCSELECT</td>
<td>64-Bit Time Base</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00 = Pick bit 63 to count</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01 = Pick bit 55 to count</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 = Pick bit 51 to count</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11 = Pick bit 47 to count</td>
</tr>
<tr>
<td>9</td>
<td>INTONBITTRANS</td>
<td>Cause interrupt signaling on bit transition.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Signal interrupt if chosen bit transitions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note: Software is responsible for setting and clearing INTONBITTRANS.</td>
</tr>
<tr>
<td>10–15</td>
<td>THRESHOLD</td>
<td>Threshold value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All 6 bits are supported by the 604 processor allowing threshold values from 0 to 63. The intent of the THRESHOLD support is to be able to characterize L1 data cache misses.</td>
</tr>
<tr>
<td>16</td>
<td>PMC1INTCTRL</td>
<td>Enable interrupt signaling due to PMC1 counter negative.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Enable PMC1 interrupt signaling when the highest-order bit is set (0x80000000) in PMC1.</td>
</tr>
<tr>
<td>17</td>
<td>PCM2INTCTRL</td>
<td>Enable interrupt signaling due to PMC2 counter negative.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Enable PMC2 interrupt signaling when the highest-order bit is set (0x80000000) in PMC2.</td>
</tr>
<tr>
<td>18</td>
<td>PCM2COUNTCTL</td>
<td>May be used to trigger counting of PMC2 after PMC1 has become negative or after a performance monitoring interrupt is signaled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Disable PMC2 counting until PMC1 bit 0 is set or until a PM interrupt is signaled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This provides a triggering mechanism for counting after a certain condition becomes true or after a preset time has elapsed. It can be used to support getting the count associated with a specific event.</td>
</tr>
<tr>
<td>19–25</td>
<td>PMC1SELECT</td>
<td>PMC1 input selector, 128 events selectable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Table 4-22 lists the 25 defined events.</td>
</tr>
<tr>
<td>26–31</td>
<td>PMC2SELECT</td>
<td>PMC2 input selector, 64 events selectable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Table 4-23 lists the 21 defined events.</td>
</tr>
</tbody>
</table>
### Table 4-22
The 25 Defined PMC1 Selectable Events

<table>
<thead>
<tr>
<th>MMCRO[19-25] Bit Encoding</th>
<th>Description of Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 0000</td>
<td>Nothing</td>
</tr>
<tr>
<td>000 0001</td>
<td>Processor cycles</td>
</tr>
<tr>
<td>000 0010</td>
<td>Number of instructions completed</td>
</tr>
<tr>
<td>000 0011</td>
<td>RTCSELECT bit transition</td>
</tr>
<tr>
<td>000 0100</td>
<td>Number of instructions dispatched</td>
</tr>
<tr>
<td>000 0101</td>
<td>icache miss</td>
</tr>
<tr>
<td>000 0110</td>
<td>dtlb misses</td>
</tr>
<tr>
<td>000 0111</td>
<td>Branch predicted incorrectly</td>
</tr>
<tr>
<td>000 1000</td>
<td>Number of reservations requested (larx is ready for execution)</td>
</tr>
<tr>
<td>000 1001</td>
<td>Number of load dcache misses that exceeded the threshold value with lateral L2 intervention</td>
</tr>
<tr>
<td>000 1010</td>
<td>Number of store dcache misses that exceeded the threshold value with lateral L2 intervention</td>
</tr>
<tr>
<td>000 1011</td>
<td>Number of mtsprr instructions dispatched</td>
</tr>
<tr>
<td>000 1100</td>
<td>Number of sync instructions</td>
</tr>
<tr>
<td>000 1101</td>
<td>Number of eieio instructions</td>
</tr>
<tr>
<td>000 1110</td>
<td>Number of integer instructions being completed every cycle (no loads or stores)</td>
</tr>
<tr>
<td>000 1111</td>
<td>Number of floating-point instructions being completed every cycle (no loads or stores)</td>
</tr>
<tr>
<td>001 0000</td>
<td>LSU-produced result</td>
</tr>
<tr>
<td>001 0001</td>
<td>SCIU1-produced result</td>
</tr>
<tr>
<td>001 0010</td>
<td>FPU-produced result</td>
</tr>
<tr>
<td>001 0011</td>
<td>Instructions dispatched to the LSU</td>
</tr>
<tr>
<td>001 0100</td>
<td>Instructions dispatched to the SCIU1</td>
</tr>
<tr>
<td>001 0101</td>
<td>Instructions dispatched to the FPU</td>
</tr>
<tr>
<td>001 0110</td>
<td>Snoop requests received</td>
</tr>
<tr>
<td>001 0111</td>
<td>Number of load dcache misses that exceeded the threshold value without lateral L2 intervention</td>
</tr>
<tr>
<td>001 1000</td>
<td>Number of store dcache misses that exceeded the threshold value without lateral L2 intervention</td>
</tr>
</tbody>
</table>
Table 4-23
The 21 Defined PMC2 Selectable Events

<table>
<thead>
<tr>
<th>MMCRO[19-25] Bit Encoding</th>
<th>Description of Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0000</td>
<td>Nothing</td>
</tr>
<tr>
<td>00 0001</td>
<td>Processor cycles</td>
</tr>
<tr>
<td>00 0010</td>
<td>Number of instructions completed</td>
</tr>
<tr>
<td>00 0011</td>
<td>RTCSELECT bit transition</td>
</tr>
<tr>
<td>00 0100</td>
<td>Number of instructions dispatched</td>
</tr>
<tr>
<td>00 0101</td>
<td>Number of cycles a load miss takes</td>
</tr>
<tr>
<td>00 0110</td>
<td>dcache misses</td>
</tr>
<tr>
<td>00 0111</td>
<td>itlb misses</td>
</tr>
<tr>
<td>00 1000</td>
<td>Branches completed</td>
</tr>
<tr>
<td>00 1001</td>
<td>Number of reservations successfully obtained (stcx succeeded)</td>
</tr>
<tr>
<td>00 1010</td>
<td>Number of mfspr instructions dispatched</td>
</tr>
<tr>
<td>00 1011</td>
<td>Number of icbi instructions</td>
</tr>
<tr>
<td>00 1100</td>
<td>Number of isync instructions</td>
</tr>
<tr>
<td>00 1101</td>
<td>Branch unit produced result</td>
</tr>
<tr>
<td>00 1110</td>
<td>SCIU0-produced result</td>
</tr>
<tr>
<td>00 1111</td>
<td>MCIU-produced result</td>
</tr>
<tr>
<td>01 0000</td>
<td>Instructions dispatched to the branch unit</td>
</tr>
<tr>
<td>01 0001</td>
<td>Instructions dispatched to the SCIU0</td>
</tr>
<tr>
<td>01 0010</td>
<td>Number of loads completed</td>
</tr>
<tr>
<td>01 0011</td>
<td>Instructions dispatched to the MCIU</td>
</tr>
<tr>
<td>01 0100</td>
<td>Number of snoop hits that have occurred</td>
</tr>
</tbody>
</table>

Performance Monitor Counter Registers

The two 32-bit performance monitor counter registers (PCM1 and PMC2) can be programmed to generate interrupt signals when they become negative (the high-order bit, 0x80000000, is set). The PMC1 and PMC2 registers are shown in Figure 4-33.

PMC1 and PCM2 can be read or written to by the mfspr and mtspr instructions. Software is expected to set the PMC registers to non-negative values; if software sets a negative value, an erroneous interrupt may be generated.
Sampled Instruction and Data Address Registers

After the performance monitoring capability is set up and enabled, performance monitor exceptions will be generated in accordance with the settings of MMCR0. The 32-bit sampled instruction address (SIA) and sampled data address (SDA) registers are simply used to hold addresses associated with the configured condition that caused the PM exception. The SIA and SDA are shown in Figure 4-33. There is no special formatting or description for these registers.

PowerPC 620 Register Set

The PowerPC 620 is the first full 64-bit implementation of the PowerPC architecture. There are few differences between the 620 register set, shown in Figure 4-34, and that of the 604 just discussed. The differences that exist are described below.

One of the first ways that the 64-bit 620 will be deployed is as a fast 32-bit processor. When porting 32-bit PowerPC code (from the 601, 603, or 604) to the 620, the following areas should be given special consideration:

- Like other PowerPC implementations, the 620 defines hardware implementation-dependent registers that do not exist on other processors. The following implementation-dependent registers require special-case handling on the 620: HID0, L2CR, L2SR, and BUSCSR.

- Because the general-purpose registers are 64 bits on 64-bit implementations, it is important for software to ensure that the upper 32 bits are cleared when addressing registers that do not define the upper 32 bits. This is particularly important when modifying registers that have a width that corresponds to the width of the implementation, such as the machine state register (MSR) and block address translation (BAT) registers.

- The page table entry (PTE) format on the 620 uses the 64-bit definition, which differs from the 32-bit definition. Any code that is responsible for managing PTE data structures would need to change to accommodate the new format. The differences between 64- and 32-bit memory management data structures is discussed in detail in Chapter 8.

- The SDR1 on the 620 is a 64-bit register and has a different format than the 32-bit version. Any code that references SDR1 would need to change to accommodate the 64-bit format.
The PowerPC Programming Model

620 Supervisor-Level Model
Supervisor-Level Exception Handling Registers
- SPR18 DSISR-DAE/ Source Instruction Service Register
- SPR19 DAR - Data Address Register
- SPR26 SRR0 - Save and Restore Register 0
- SPR27 SRR1 - Save and Restore Register 1
- SPR28 SRRG0 - SPR General Register 0
- SPR29 SRRG1 - SPR General Register 1
- SPR30 SRRG2 - SPR General Register 2
- SPR31 SRRG3 - SPR General Register 3

Supervisor-Level Memory Management Registers
- SPR500 SPRG0 - SPR General Register 0
- SPR501 SPRG1 - SPR General Register 1
- SPR502 SPRG2 - SPR General Register 2
- SPR503 SPRG3 - SPR General Register 3

Miscellaneous Registers
- SPR284 TBL - Time Base Facility Lower (Writing)
- SPR285 TBU - Time Base Facility Upper
- SPR1010 IABR - Instruction Address Breakpoint Register
- SPR22 DEC - Decrementer
- SPR282 EAR - External Address Register (Optional)
- SPR283 DABR - Data Address Breakpoint Register

Performance Monitoring Registers
- SPR52 MMCRR - Monitor Mode Control Register 0
- SPR53 PMC1 - Performance Monitor Counter 1
- SPR54 PMC2 - Performance Monitor Counter 2
- SPR55 SAD - Sampled Data Address
- SPR56 SIA - Sampled Instruction Address

620 User-Level Model
User Instruction Set Architecture
General-Purpose Registers
- SPR0 GPR0
- SPR1 GPR1
- SPR2 GPR2
- SPR3 GPR3
- SPR4 GPR4
- SPR5 GPR5
- SPR6 GPR6
- SPR7 GPR7
- SPR8 GPR8
- SPR9 GPR9
- SPR10 GPR10
- SPR11 GPR11
- SPR12 GPR12
- SPR13 GPR13
- SPR14 GPR14
- SPR15 GPR15
- SPR16 GPR16
- SPR17 GPR17
- SPR18 GPR18
- SPR19 GPR19
- SPR20 GPR20
- SPR21 GPR21
- SPR22 GPR22
- SPR23 GPR23
- SPR24 GPR24
- SPR25 GPR25
- SPR26 GPR26
- SPR27 GPR27
- SPR28 GPR28
- SPR29 GPR29
- SPR30 GPR30
- SPR31 GPR31

Floating-Point Registers
- SPR32 FPR0
- SPR33 FPR1
- SPR34 FPR2
- SPR35 FPR3
- SPR36 FPR4
- SPR37 FPR5
- SPR38 FPR6
- SPR39 FPR7
- SPR40 FPR8
- SPR41 FPR9
- SPR42 FPR10
- SPR43 FPR11
- SPR44 FPR12
- SPR45 FPR13
- SPR46 FPR14
- SPR47 FPR15
- SPR48 FPR16
- SPR49 FPR17
- SPR50 FPR18
- SPR51 FPR19
- SPR52 FPR20
- SPR53 FPR21
- SPR54 FPR22
- SPR55 FPR23
- SPR56 FPR24
- SPR57 FPR25
- SPR58 FPR26
- SPR59 FPR27
- SPR60 FPR28
- SPR61 FPR29
- SPR62 FPR30
- SPR63 FPR31

Condition Register
- SPR64 CR

Floating-Point Status and Control Register
- SPR65 FPSCR

User Virtual Environment Architecture
- SPR66 TBR268 TBL - Time Base Facility Lower (Reading)
- SPR67 TBR269 TBU - Time Base Facility Upper (Reading)

Configuration Registers
- SPR1000 MSR - Machine State Register
- SPR1001 HID0 - Checkstop Sources and Enable Register
- SPR1002 PVR - Processor Version Register
- SPR1003 PIR - Processor Identification Register

Level 2 Cache Control Registers
- L2CR - Level 2 Cache Control Register
- L2SR - Level 2 Cache Status Register
- BUSCSR - Bus Status and Control Register

Address Space Register
- ASR

Figure 4-34
The 620 PowerPC implements a fully 64-bit programming model.
Hardware
Implementation-Dependent Register

The 620's hardware implementation-dependent 0 (HIDO) register, shown in Figure 4-35, is similar to that found on the 604. However, there are fields that exist only on the 620: branch prediction modes (HIDO[25,26]) and instruction fetch modes (HIDO[27,28]). Table 4-24 summarizes the bits that are defined for the 620's HIDO.

<table>
<thead>
<tr>
<th></th>
<th>EMCP</th>
<th>ECPC</th>
<th>EBA</th>
<th>EBD</th>
<th>DCE</th>
<th>ICE</th>
<th>NHR</th>
<th>DPWF</th>
<th>ICFI</th>
<th>DCI</th>
<th>DFWT</th>
<th>SSME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Branch Prediction Modes
- 00 Static branch prediction without branch history table update
- 01 Dynamic branch prediction
- 10 Static branch prediction with BHT updates
- 11 Reserved

Instruction Fetch Modes
- 00 No speculative fetch off the chip from main memory
- 01 No speculative fetch off the chip with more than one pending branch
- 10 No speculative fetch off the chip with more than two pending branches
- 11 Allow speculative instruction fetch off the chip.

Figure 4-35
The 620's hardware implementation-dependent 0 (HIDO) register enables checkstops and advanced processor features.
### Table 4-24
The 620's HID0 Register Bit Definitions

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>EMCP</td>
<td>Enable Machine Check Input Pin</td>
</tr>
<tr>
<td>1</td>
<td>ECPC</td>
<td>Enable Cache Parity Checking</td>
</tr>
<tr>
<td>2</td>
<td>EBA</td>
<td>Enable Machine Check on Address Bus Parity Error</td>
</tr>
<tr>
<td>3</td>
<td>EBD</td>
<td>Enable Machine Check on Data Bus Parity Error</td>
</tr>
<tr>
<td>4-13</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>14</td>
<td>DPWF</td>
<td>Disable Processor Internal Watchdog Function</td>
</tr>
<tr>
<td>15</td>
<td>NHR</td>
<td>Not Hard Reset</td>
</tr>
<tr>
<td>16</td>
<td>ICE</td>
<td>Instruction Cache Enable</td>
</tr>
<tr>
<td>17</td>
<td>DCE</td>
<td>Data Cache Enable</td>
</tr>
<tr>
<td>18-19</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>20</td>
<td>ICFI</td>
<td>Instruction Cache Invalidate All</td>
</tr>
<tr>
<td>21</td>
<td>DCI</td>
<td>Data Cache Invalidate All</td>
</tr>
<tr>
<td>22</td>
<td>DFWT</td>
<td>Data Force Write-Through</td>
</tr>
<tr>
<td>24</td>
<td>SSME</td>
<td>Serial Instruction Execution Disable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = The 620 executes one instruction at a time. The 620 does not post a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>trace exception after each instruction completes as it would if MSR(SE) or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MSR(BE) were set.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Instruction execution is not serialized.</td>
</tr>
<tr>
<td>25-26</td>
<td>BPM</td>
<td>Branch Prediction Modes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00 = Static branch prediction without update of branch history table</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01 = Dynamic branch prediction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 = Static branch prediction with branch history table</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11 = Reserved (same as 00 state)</td>
</tr>
<tr>
<td>27-28</td>
<td>IFM</td>
<td>Instruction Fetch Modes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00 = No speculative fetch off the chip from main memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01 = No speculative fetch off the chip with more than one pending match</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 = No speculative fetch off the chip with more than two pending branches</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11 = Allow speculative instruction fetching</td>
</tr>
<tr>
<td>29-31</td>
<td></td>
<td>Reserved</td>
</tr>
</tbody>
</table>
**Address Space Register**

Found only on 64-bit implementations, such as the 620, the *address space register* (ASR) holds bits 0–51 of the segment table’s physical address. Recall that 64-bit processors don’t have segment registers as do 32-bit implementations. The 620 uses the ASR, shown in Figure 4-36, and the segment table to define the set of segments that can be addressed — just as 32-bit processors use segment registers. Chapter 8, “Memory Management,” deals with memory management on the PowerPC family and describes the ASR as well as segment-based address translation in detail.

<table>
<thead>
<tr>
<th>Physical Address of Segment Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

**Reserved segment table addresses**

- 0x0000 0000 0000 0000
- 0x0000 0000 0000 1000
- 0x0000 0000 0000 2000

These addresses correspond to locations in the exception vector table and cannot be used as segment table addresses.

**Figure 4-36**

The 620’s address space register (ASR) performs the same function as segment registers.

**Bus Status and Control Register**

The 64-bit *bus status and control register* (BUSCSR) contains all the information required to define the processor’s interface. Errors that occur on the bus and during bus transactions can be reflected in the BUSCSR. The error conditions that are reflected in the BUSCSR can occur during a read or write to this register; in such a situation, the error condition will either occur completely before or after the point at which data is written to or read from the register, but not any combination thereof. As a result, software does not have to deal with incomplete information resulting from reading the BUSCSR at exactly the wrong time — while it is being updated.
**Performance Monitoring Registers**

The 620's implementation of performance monitoring registers is the same as that found on the 604 with the exception of SIA and SDA register width. The SIA and SDA on the 620 are both 64 bits wide. The performance monitoring capabilities of the 604 and 620 are discussed in Chapter 12, "Techniques and Tricks."

**Summary**

In the chapters that follow, we'll start to put some of the pieces together, beginning with effective address calculation in Chapter 5, "Addressing Modes and Operand Conventions." Before long, you'll be hooked on the lavish set of registers that we covered in this chapter — you may even wonder how you ever managed with the limited register set of the x86.
Understanding addressing modes and operand conventions is a key step toward the goal of becoming a proficient programmer on the PowerPC. Addressing modes define the various mechanisms used to manipulate data. Operand conventions include naming, evaluation order, and position within an instruction. Knowing and following the rules increases software performance on PowerPC systems. In this chapter, we’ll discuss addressing modes and the calculation of effective addresses.

Fortunately, there are many similarities between effective address calculation on the Intel x86 and on the PowerPC. However, the differences that do exist result from the fundamental differences between RISC and CISC architectures. A solid understanding of this topic will be an important building block for the material covered in subsequent chapters.

In this chapter, we’ll briefly review the basics of Intel x86 effective address calculation and operand conventions, then we’ll cover the PowerPC versions in detail. If you’re comfortable with Intel operand conventions and effective address calculations, feel free to skip ahead to the PowerPC discussion.
**x86 Conventions**

Differences between x86 processors and their available addressing modes make it important to note which x86 family member we’re using. Unless otherwise noted, we’ll be referring to the Intel i486 for this and future discussions. On 386 and 486 processors operating in protected mode, for example, any register can be used as an index register (a RISC-like feature); on earlier processors, this was not possible.

The x86 architecture employs variable-length instructions and a two-operand instruction format to perform most operations. This differs from the fixed-length, three-operand PowerPC instruction format. Two-operand instructions that perform a mathematical or logical operation typically require that one of the two operands be overwritten by the result. This implies that one of the values will be unavailable for subsequent operations.

The CISC nature of the x86 family allows a very flexible instruction set architecture. However, the variable-length instructions and two-operand instruction format can hinder performance. Having to determine instruction length when code is poorly aligned can produce latency problems.

**Operand Types**

A *register operand* is simply the value contained in a particular register. Of course, some restrictions exist for certain register and operation combinations, and others depend on the processor’s operating mode. But, in general, all x86 registers can be used as instruction operands. The following code uses two register operands in a register/register move operation:

```c
mov eax, ecx ; move the contents of register ecx into eax
```

An *immediate operand* is a literal value that is embedded in the x86 instruction at the time of assembly. In the following example, both the 0x23 and the 0x03 immediate operands are stored as part of the opcodes for the `mov` and `shl` instructions, respectively. Immediate operands can be bytes, half-words (16 bits), or words (32 bits).

```c
mov EAX, 0x23 ; move immediate 0x23 into the EAX register
shl EAX, 0x03 ; shift EAX left 3 bits
```
All x86 instructions that perform a memory access (load, store, compare, and so on) do so by calculating the effective address (EA) of the memory operand. Calculation of the EA depends on the memory operand itself and the addressing mode. The mechanism and components used to generate the effective address are discussed in the next section.

The size of a memory operand is always explicitly encoded as part of the opcode. Because the same instruction mnemonic (mov, for example) is used to access memory regardless of operand size, x86 source code requires that operand size be either implied by the width of the register in the operation (EAX, AX, AL) or explicitly designated with a ptr size designation. For example, the mov instruction can be used to access an 8-, 16-, or 32-bit memory unit using implied width as shown here:

\[
\begin{align*}
\text{mov al, [esi]} & : \text{8-bit register al gets byte value at esi} \\
\text{mov ax, [esi]} & : \text{16-bit register ax gets half-word at esi} \\
\text{mov eax, [esi]} & : \text{32-bit register eax gets word at esi}
\end{align*}
\]

Or explicitly designated width can be used when the size of the operand can’t be inferred from the operands.

\[
\begin{align*}
\text{mov byte ptr [esi], 0} & : \text{8 bits of 0 written at esi} \\
\text{mov word ptr [esi], 0} & : \text{16 bits of 0 written at esi} \\
\text{mov dword ptr [esi], 0} & : \text{32 bits of 0 written at esi}
\end{align*}
\]

On x86 processors, the effective address is only part of the story. Memory operands use either an implicit or explicit segment reference. A memory operand is located by specifying an offset (the effective address) from the start of a segment. For the purposes of this discussion, we’ll assume that segment selection is implicit and that effective address calculation is relative to the appropriate segment register.

**Operand Movement**

Intel i486 instructions that move data may be grouped roughly into the following categories:

- Register to register
- Immediate to register
- Register to memory
- Memory to register
- Immediate to memory
- Memory to memory

Neither immediate/register nor register/register operations require the calculation of an effective address. As a result, register/register and immediate/register moves are simple and generally execute quickly. Both x86 and PowerPC architectures support register/memory and memory/register operations. However, the x86 implements far more addressing modes for accessing memory than does the PowerPC.

The ability to perform immediate/memory and memory/memory operand movement is notably absent on the register/register architecture of PowerPC processors. The RISC PowerPC architecture can load and store to memory only using indirect addressing. Registers must hold the effective address of any memory operand. The ability of the x86 to access memory and calculate effective addresses in a variety of different ways is one of the most powerful features of the x86 programming model.

**x86 Effective Address Calculation**

To resolve an effective address, the i486 must sum up to four components: base register, index register, scale factor, and immediate displacement. The valid combinations of these components are known as addressing modes. The x86 addressing modes that require effective address generation, and thus are relevant to our understanding of PowerPC effective address generation, are as follows.

- **Relative Addressing**
  Relative addressing requires a base register and optionally includes an index register and/or an immediate displacement. Any of the 32-bit general-purpose registers can be used as the base register in relative addressing. The following example shows EBP as the base register, ESI as the index register, and an immediate displacement of 4.

  ```
  mov eax,[ebp + esi + 4] ; relative addressing
  ```

- **Indirect (Index) Addressing**
  Indirect addressing uses a single 32-bit general-purpose register with an optional immediate displacement to determine the EA (effective address). Unlike relative addressing, no base register is used.
- **Direct Addressing**
  Direct addressing embeds an immediate value in the instruction opcode at assembly. This mode is commonly used to access well-known areas of memory whose offset does not change at run-time.

```assembly
mov eax,[esi + 0x10] ; eax gets the WORD
; at offset esi+0x10
```

**POWERPC CONVENTIONS**

One aspect of the PowerPC programming model that clearly identifies the PowerPC as a RISC architecture is its operand conventions and memory addressing modes. The register/register architecture of the PowerPC enforces register-based data movement and limits the number of available addressing modes.

Each PowerPC instruction is always 32 bits in length and is guaranteed to be aligned on a 32-bit word boundary in memory. This alignment means that the low-order 2 bits of an address are ignored by the processor when fetching instructions. When accessing data in memory, the processor uses the full 32 bits of the effective address.

In the following sections, we’ll examine the operand conventions that apply to all implementations of the PowerPC architecture. Following that, we’ll look at the addressing modes available for load/store operations and the calculation of effective addresses.

**Three-Operand Format**

The PowerPC implementations use a register/register architecture as described in Chapter 1, “The PowerPC Transition.” One common feature of a register/register architecture, such as the PowerPC architecture, is a three-operand instruction format. Using three operands for common operations (such as loads and stores) has the benefit of leaving the two original source operands intact after the operation is complete.
While reading this section, keep in mind that it's generally tough to confound a good x86 programmer. As a result, simple examples designed to show how you can quickly run out of registers always seem a bit contrived. But in the general case, the problem manifests itself more frequently. For example, it is common to encounter x86 register shortages when programming inside nested loops or when more than one operation is ongoing (such as simultaneous calculations of vertical and horizontal offsets for line drawing).

To see why the three-operand format has a distinct advantage, consider the following example. Assume that we have an array of 100 32-bit values and that for each element of the array we must fetch two sequential values. These two values are multiplied together and their 64-bit product is stored in the destination array. The two original values are then added and the 32-bit result is again stored in the new array.

Our x86 example is the MulAdd function, which takes two arguments: a source buffer pointer and a destination buffer pointer. Furthermore, we'll assume each argument is passed on the stack using the Pascal calling conventions (first argument pushed first; last argument pushed last) and that default segment registers are used. The x86 assembly language version of MulAdd is shown in Listing 5-1.

The first operand, loaded into EAX, was destroyed by the mul instruction and had to be reloaded. Clearly, having one more register in which to park the first operand could have saved us one redundant memory access per loop iteration. In Listing 5-2, note that the three-operand instruction format allows preservation of both source operands.

Certainly, saving one or two move instructions doesn't justify the existence of this operand format — but saving one move (or load/store operation) repeatedly or inside loops is the essence of good programming and that's exactly what using the three-operand format can buy you.

As shown in the PowerPC example, the destination register for load operations is the leftmost operand — the same format as found in the x86 move instruction. However, when performing store operations on the PowerPC, the register that contains the destination (effective) address is the rightmost operand. In other words, load to the left and store to the right.
Listing 5-1

The contrived MulAdd routine demonstrates the inefficiency associated with running out of registers.

```c
; int MulAdd(int *srcBuf, int *destBuf);
; (near call)
;
; At start, we know that the following is true:
; EBP = being used for stack frame
; ESI = used to fetch EAX
; EDI = pointer to a buffer used to store 64-bit
; result EAX*EBX and 32-bit result EAX+EBX
; On exit:
; EDI buffer updated
;
; start:
; push ebp
; mov ebp, esp
; mov edi, [ebp+0x08]
; mov esi, [ebp+0x0c]
; mov ecx, 0x100

arrayloop:
; mov eax, [esi]  ; get first operand
; mov ebx, [esi+0x04]  ; get second operand
; mul ebx  ; edx:eax = first op * second op
; mov [edi], eax  ; save low dword
; mov [edi+0x04], edx  ; and high dword
;
; at this point, we need the original eax again (to add to
; ebx)...it'd sure be nice to have another register around...
;
; mov eax, [esi]  ; reload first operand
; add eax, ebx  ; eax = first + second operand
; mov [edi+0x08], eax  ; store result of add operation
; add edi, 0x0c  ; bump dest buffer pointer
; add esi, 8  ; bump source to next operands
;
; loop arrayloop  ; loop until done

pop ebp
ret
```
Listing 5-2

The PowerPC implementation of MulAdd demonstrates the efficiency associated with the three-operand instruction format.

```
; PowerPC assembly language version of MulAdd
; assume: r4 -> *srcBuf
; r5 -> *destBuf
;
; on exit:
; destBuf is filled with appropriate data
;
start:
    li     r8,0x100     ; initialize counter
PPCLoop:
    lwz    r9,0(r4)    ; get first operand
    lwz    r12,4(r4)   ; get second operand
    mullhw r11,r12,r9  ; r11 = r9 * r12
    stw    r11,4(r5)   ; store high-word of mult
    mullw  r11,r12,r9  ; r11 = r9 * r12
    stw    r11,0(r5)   ; store low-word of mult
;
; note that the values contained in r12 and r9 are still intact
;
    addc   r11,r12,r9  ; r11 = r12 + r9
    stw    r11,8(r5)   ; store result of addition
    addic  r5,r5,12    ; increment destBuf ptr
    addic  r8,r8,-1    ; decrement counter
    cmpwi  r8,0        ; done yet?
    bne    PPCLoop     ; no, then loop
;
; return to calling code here...
```

**Naming Conventions**

When we discuss PowerPC instructions and operand placement, it is convenient to use generic names in place of actual registers and values. This is, in part, a result of differing register usage conventions in the PowerPC arena.
Appendix A, "PowerPC Instruction Set Reference," contains a complete list of register and value aliases; however, for the following discussion, it's helpful to know a subset of those names.

Table 5-1 lists some of the most common operand aliases. We'll use these names when discussing operand placement and instruction usage in the future. For the sake of clarity, we'll use the same operand naming conventions as the PowerPC microprocessor user manuals published by Motorola and IBM.

**Table 5-1**

Operand Field Conventions for Load/Store Operations

<table>
<thead>
<tr>
<th>Operand Field</th>
<th>Description/Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>rA, rB</td>
<td>Specifies a source or destination general-purpose register (GPR) used to calculate effective addresses.</td>
</tr>
<tr>
<td>rD</td>
<td>Specifies a destination GPR whose contents are used to calculate effective addresses.</td>
</tr>
<tr>
<td>rS</td>
<td>Specifies a source GPR whose contents are used to calculate effective addresses.</td>
</tr>
<tr>
<td>d (displacement)</td>
<td>Specifies the immediate displacement for load and store instructions in a 16-bit signed format. This value is sign extended to the width of the implementation (32 or 64 bits) and added to a base address.</td>
</tr>
<tr>
<td>NB (number of bytes)</td>
<td>Specifies the number of bytes that will be moved during an immediate string load/store.</td>
</tr>
</tbody>
</table>

Additionally, the CIA (current instruction address) is commonly used when discussing operand fields and naming conventions. The CIA is considered an internal register and is used exclusively by the processor as a pointer to the next instruction to execute. It is neither readable nor writeable by software. However, it is often a convenient convention to use in discussions. On x86 systems, EIP is equivalent to the CIA in purpose only.

Table 5-2 shows the common formats for load and store operations using the register naming conventions described in Table 5-1. Note that the "load" and "store" instructions used in the left column of Table 5-2 are not actual PowerPC mnemonics; they are used symbolically to represent load and store operations of all memory unit lengths (byte, half-word, word, doubleword).
Table 5-2
Common Forms of Load/Store Operations

<table>
<thead>
<tr>
<th>Load/Store Instruction</th>
<th>Description/Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>load rD, d(rA)</td>
<td>Load the destination register rD with the contents of the address formed by adding source register rA and immediate displacement d.</td>
</tr>
<tr>
<td>load rD, rA, rB</td>
<td>Load the destination register rD with the contents of the address formed by adding source registers rA and rB.</td>
</tr>
<tr>
<td>store rS, d(rA)</td>
<td>Store the contents of source register rS to the address formed by adding destination register rA and immediate offset d.</td>
</tr>
<tr>
<td>store rS, rA, rB</td>
<td>Store the contents of source register rS to the address formed by adding destination register rA and rB.</td>
</tr>
</tbody>
</table>

**Operand Types**

PowerPC instructions use a familiar set of operand types: register, immediate, and memory. Each operand type is analogous to the x86 operand category of the same name. The following sections define the PowerPC operand types. Differences between the x86 and PowerPC versions of an operand type are noted where they occur.

**Register Operands**

When registers are used as operands, the register number is encoded as part of the 32-bit instruction. Depending on the instruction type, a register operand can be a general-purpose register (GPR), floating-point register (FPR), or any of the other register types defined in Chapter 4, "The PowerPC Programming Model."

For load and store operations, GPRs are always used to hold base addresses. In other words, when loading a value from memory, a GPR must be a part of the effective address calculation — even if the destination is a floating-point register. Such register restrictions are noted in the individual instruction definitions in Appendix A, "PowerPC Instruction Set Reference."
Immediate Operands

The PowerPC implementations use immediate operands, but require that they be used in conjunction with a register operand. For example, the `andi` (AND immediate) instruction uses both a source register operand and the immediate operand as the two values to AND together.

The size of an immediate operand depends on the specific instruction. Typically, immediate operands are 16-bit values that are embedded in bits 16–31 of the 32-bit instruction. The largest immediate operand allowed for PowerPC processors is 16 bits. If you need to load a register with a 32-bit value, it requires two instructions. Think about it: if all instructions are 32 bits, an instruction that used a 32-bit immediate operand would have no room for the opcode. You’ve seen this situation crop up in the code sequences presented in earlier chapters, such as this example from Chapter 1:

```
addis r3, r0, 0x1234 ; load high 16 bits into register r3
ori r3, r3, 0xabcd ; load low 16 bits into register r3
lwz r4, VariableInMemory ; get the address of variable in r4
stw r3, 0(r4) ; store value in r3 to address in r4
```

To load the 32-bit immediate value (0x1234abcd) into GPR r3, two 16-bit immediate loads must be used. In the preceding example, the `addi` and `addis` instructions function as load immediate and load immediate shifted instructions, respectively.

Memory Operands

PowerPC RISC microprocessors can access memory only by using indireced register-based load and store operations. One benefit of this restriction is that memory operands on PowerPC processors are considerably simpler than on x86 processors.

Moving the contents of a memory variable into a register requires that the address of the variable must be loaded into a register, then a second register can be loaded with the value of the variable by using the first register (containing the address) as a base address. The following comparison illustrates this point:
PowerPC Programming for Intel Programmers

; x86 example of loading a 32-bit memory variable into a register
;
    mov    eax, VariableInMemory        ; simple enough...

; PowerPC example of loading a 32-bit memory variable into a register
; Note: rBase is used to denote a previously initialized GPR that
; is used as a base-pointer for variable references. rBase is
; occasionally referred to as a Table Of Contents.
;
    lwz    r4, VariableInMemory(rBase) ; get address of variable from base
    lwz    r3, 0(r4)                 ; load r3 with value of variable

Memory access instructions encode the size of the memory unit explicitly. The alignment of data in memory can affect performance and even cause exceptions. In particular, the consequences of data alignment depend on the current state of the processor: the endianness, the operation being performed, and even the processor implementation.

## Alignment and Misalignment

PowerPC implementations use 8-bit, 16-bit, and 32-bit memory operands. On 64-bit PowerPC processors, such as the 620, 64-bit memory operands may be used. Table 5-3 summarizes the PowerPC memory operands that are used in this book.

### Table 5-3

<table>
<thead>
<tr>
<th>Operand Name</th>
<th>Length</th>
<th>Lowest Order Bits If Aligned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>8 bits</td>
<td>xxxx</td>
</tr>
<tr>
<td>Half-word</td>
<td>2 bytes</td>
<td>xxx0</td>
</tr>
<tr>
<td>Word</td>
<td>4 bytes</td>
<td>xx00</td>
</tr>
<tr>
<td>Doubleword (dword)</td>
<td>8 bytes</td>
<td>x000</td>
</tr>
<tr>
<td>Quadword (qword)</td>
<td>16 bytes</td>
<td>0000</td>
</tr>
</tbody>
</table>

Note: Quad words are not used as memory operands and are included in this table only for completeness.
With the exception of `l accomp (load multiple word), `st accomp (store multiple word), and the various load and store string instructions, the operand sizes listed previously in Table 5-1 are used exclusively for all operations on PowerPC processors.

To demonstrate the problems that can occur with alignment, let's consider a 16-byte data structure with a starting address of 0xF2001004. The four low-order bits of the data structure's effective address are 0b0100. We see, from Table 5-1, that our data structure is word aligned.

If we wanted to dword align our data, we'd have to move the structure's starting location to a dword boundary such as 0xF2001008. If we specified the starting location as 0xF2001007 and tried to access the data as anything other than a byte-size value, we'd get an alignment exception. A byte-size access is acceptable because there are no alignment restrictions for 8-bit (byte-size) accesses.

### Programming Point: Alignment

For single-register memory accesses, the best performance is obtained when memory operands are aligned on boundaries equal to the size of the operand. For example, when accessing dwords, optimal performance is obtained by having each dword value aligned on a 64-bit boundary. In other words, the low-order three bits are set to zero. The following figure shows the relationship between the low-order bits and memory unit alignment. Note that the figure shows a 32-bit address; on 64-bit implementations such as the 620, the low-order three bits would be labeled 63, 62, and 61.

**Half-word aligned (16 bits)**

**Word aligned (32 bits)**

**Dword aligned (64 bits)**

Note: To achieve the alignment shown, the bit corresponding to the desired alignment must be zero. Additionally, all bits to its right (of lower order) must also be cleared.

The low-order address bits are cleared according to the unit of alignment.
Effective Address Calculations

When PowerPC processors access memory they must resolve the effective address (EA) for the memory location of interest. The effective address will be either 32 or 64 bits, depending on the processor's implementation width. In addition, the mechanism used to resolve the effective address depends on two elements: operand width and addressing mode.

PowerPC memory operands come in all widths shown in Table 5-3: bytes, half-words, words, and doublewords. PowerPC instruction mnemonic names explicitly reference the width of the memory operand. For example, consider the following instructions: \texttt{lbz} (load byte and zero), \texttt{lhz} (load half-word and zero), \texttt{lzw} (load word and zero), and \texttt{ld} (load double). There is no room for ambiguity with respect to memory operand size for each instruction.

There are several addressing modes available for load/store operations. Each addressing mode is accompanied by a load instruction that uses each particular addressing mode. PowerPC processors always use one of the three addressing modes listed here.

- Register indirect with immediate index mode

\[
\texttt{lbz} \ r2, 4(r4) \quad ; \ EA = r4+4
\]

- Register indirect with index mode

\[
\texttt{lbzx} \ r2, r4, r5 \quad ; \ EA = r4+r5
\]

- Register indirect mode

\[
\texttt{lswi} \ r2, r4, 10 \quad ; \ EA = r4, \text{number bytes=10}
\]

Register Indirect with Immediate Index Mode

The indexing register indirect modes of the PowerPC architecture are analogous to the x86's indexing modes. When an immediate displacement (d) is used, as shown in Figure 5-1, the displacement is stored in the instruction as a 16-bit signed number. In 32-bit PowerPC implementations, the 16-bit displacement is sign extended to form a 32-bit value; in 64-bit implementations, the 16-bit displacement is sign extended to form a 64-bit value.
In the example shown in Figure 5-1, the `lbz` (load byte and zero) instruction generates the effective address by adding the sign-extended displacement (4) to the contents of GPR rD, then loads a byte value from that address. The effective address is generated in the same fashion for store operations.

The following comparison between PowerPC and x86 immediate indexing shows the similarities between register-based indexing modes:

```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbz r2, 4(r3)</td>
<td>PPC: load r2 with byte value at (r3+4)</td>
</tr>
<tr>
<td>movzx eax, byte ptr [esi+4]</td>
<td>x86: load eax with byte value at (esi+4)</td>
</tr>
</tbody>
</table>
```

**Programming Point: Usage of GPR r0 in the rA Position**

The general-purpose register r0 has a special definition when used in the rA operand position. When GPR r0 is used in the rA operand position, a value of zero is used — not the contents of GPR r0. This provides an easy way to specify a value of zero, without having to allocate a register for that purpose. At first glance, this can be confusing. So let's take a look at a couple of examples.

```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbz r2, 100(r0)</td>
<td>PPC: load r2 with value at address 100</td>
</tr>
</tbody>
</table>
```

The effective address would be calculated as EA = 0 + 100, regardless of the value contained in r0. Using r0 in the rA position works for other instructions as
well. Consider using r0 in an addi instruction — it is functionally equivalent to loading a register with an immediate value.

; Use r0 in the rA position with; other instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>rD or rS</th>
<th>rA Offset</th>
<th>rB Base Address</th>
<th>Sub-opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5 6</td>
<td>10 11</td>
<td>15 16</td>
<td>20 21</td>
</tr>
</tbody>
</table>

Example: lbz rD, rA, rB

Reserved

Figure 5-2
Instructions that use register indirect with index addressing mode have a 32-bit base and 32-bit offset.

In the example in Figure 5-2, the lbzx (load byte and zero indexed) instruction loads a byte value from the effective address generated by adding the contents of GPR rA and GPR rB. The effective address is generated in the same fashion for store operations.

Again, a brief comparison of byte-loading operations on both the x86 and a PowerPC implementation show the similarities:

lbzx r2, r3, r4 ; PPC: load r2 with byte value at (r3+r4)
movzx eax, byte ptr [esi+ebx] ; x86: load eax with byte value at (esi+ebx)
Register Indirect Mode

The register indirect mode is used exclusively with the load/store string instructions. The effective address is simply the contents of the GPR that occupies the rA position. The number of bytes (NB) operand specifies the number of bytes to be loaded into the GPRs; this instruction does not perform memory-to-memory load/stores.

In the example in Figure 5-3, the lswi (load string word immediate) instruction would load 20 byte values from the effective address in GPR rA into GPRs starting with rD and going to rD+(20/4). A full explanation of the load/store string and lswi instructions may be found in Appendix A, “PowerPC Instruction Set Reference.” There is no equivalent instruction on x86 family members.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>rD or rS</th>
<th>rA Offset</th>
<th>Number of Bytes</th>
<th>Sub-opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>20</td>
</tr>
</tbody>
</table>

Example: lswi rD, rA, 20

Reserved

Figure 5-3
Only load and store string instructions use register indirect with index addressing modes.

Addressing Modes As Part of the Instruction

One of the fundamental differences between the RISC instruction set of the PowerPC processor and that of the CISC x86 family is the relationship between instruction and addressing modes. On x86 processors the mov instruction mnemonic is generic. The opcode generated depends on the context of the instruction — specifically, the addressing mode that is used. This situation is not unique to the mov mnemonic.

On PowerPC processors, the addressing mode and instruction mnemonic are closely coupled. There are instruction mnemonics specifically for immediate addressing as well as for the other addressing modes described previously. Recognizing this is helpful when confronting the conventions of the PowerPC architecture.
SUMMARY

There are many helpful similarities between the x86 and PowerPC operand conventions and addressing modes. In fact, the closer we look at the two architectures, the more similarities we'll find. In the next chapter, we'll take a much closer look at the different forms of PowerPC instructions and at the instruction set itself. By the end of the chapter, we'll be ready to start writing some code.
This chapter presents the PowerPC instruction set and the associated knowledge that will enable you to code, optimize, and understand how to write effective programs. But we’re operating under the fundamental assumption that no optimization comes without a little pain. An understanding of instruction synchronization, simplified mnemonics, and the various forms of each instruction is fundamental to writing efficient and effective PowerPC assembly language.

In many cases, this chapter introduces concepts that are relevant to the instruction set but does not attempt to explain them comprehensively. Rest assured that these concepts will be discussed in turn. For example, instruction timing is examined in Chapter 7, “The Sublime Art of Instruction Timing.” There are numerous examples of coding techniques in Chapter 11, “PowerPC Assembly Language Examples.” And Appendix A, “PowerPC Instruction Set Reference,” contains a detailed definition for each PowerPC instruction.


INSTRUCTION GUIDELINES

As I’ve stressed in previous chapters, a PowerPC instruction is always 32 bits in length and is aligned on a 32-bit boundary in program memory. Instructions are fetched into an instruction queue and dispatched into the processor’s pipeline. But keeping the pipeline full and free of stalls means that you must write efficient, quality code. This chapter is the starting place for the concepts required to create optimal code on PowerPC processors.

Sometimes it is necessary to refer to the contents of the instruction pointer and how it is modified in specific situations, such as exception conditions. On x86 processors, the instruction pointer is contained in the IP (or EIP) register, and is accessible to software of any privilege level. On PowerPC processors, the current instruction address (CIA) is analogous to the x86’s EIP register. However, the CIA is an internal, processor-only register, not accessible by software of any privilege level.

There are many concepts that must be presented along with a new instruction set. Some are important to understand before continuing, some simplify writing code, and some help you write efficient code. The concepts surrounding the use of the PowerPC instruction set presented in this chapter are in the order in which they most benefit learning. With the formalities out of the way, let’s dive in.

Synchronization

Synchronization on the PowerPC family of microprocessors is an important topic for two reasons: The rules that govern when you must synchronize are processor-implementation dependent and there is no equivalent concept on the x86 processor family.

Recall that PowerPC implementations are able to execute instructions out of order. As a consequence, the processor context at the time a specific instruction is executed is not necessarily what it would have been had the instruction stream been executed linearly. Outwardly, the results of out-of-order execution are transparent to both the programmer and user; internally, however, the context shifts that occur require attention. A processor’s context encompasses the current privilege level, address translation, and memory protection configuration. As each instruction is executed, it is subject to the rules established by the context of the processor.

The PowerPC architecture provides special instructions to ensure that all instructions appearing earlier in the instruction stream have executed
before proceeding. This process is known as \textit{context synchronization} and ensures that instructions execute in the context in which they were issued. In other words, each instruction sees the privilege level, address translation, and system register settings it would see if the code had been executed in a linear fashion.

The example shown in Listings 6-1a and 6-1b shows why context synchronization is necessary. Here, we want to load GPR \( r6 \) with a word from memory at virtual address \( 0xf8000010 \). Our example assumes that data address translation is enabled and that \( r4 \) and \( r5 \) contain the information necessary to set up both DBAT registers for the access to \( 0xf8000010 \). Executing the top block of instructions, instruction 1 loads GPR \( r3 \) with the source address. Instructions 2 through 4 set up the BAT registers so that we can access the virtual address contained in \( r3 \). Finally, instruction 5 loads \( r6 \) with the value contained at the virtual address contained in \( r3 \).

\textbf{Listing 6-1}

Linear instruction execution requires no thought as to appropriate context. But an out-of-order sequence could result from a lack of proper context synchronization.

\begin{verbatim}
Listing 6-1a: Linear Instruction Stream

; Assumes:
; r4,r5 contain valid BAT register information
;
addis r3,r0,0xf800 ; instruction #1
addi r3,r3,0x0010 ; instruction #2
mtdbat1 0,r5 ; instruction #3
mtdbatu 0,r4 ; instruction #4
lwz r6,0(r3) ; instruction #5
; ... code continues...

Listing 6-1b: Out-of-Order Version of Listing 6-1a.

addis r3,r0,0xf800 ; instruction #1
mtdbat1 0,r5 ; instruction #3
addi r3,r3,0x0010 ; instruction #2
lwz r6,0(r3) ; instruction #5 *ACCESS EXCEPTION*
mtdbatu 0,r4 ; instruction #4
; ... code continues...
\end{verbatim}
The block of instructions shows the result of out-of-order execution without context synchronization. In effect, we try to access virtual address 0xf8000010 (instruction #5) before the DBAT register is completely set up.

To avoid this problem, we must use a context synchronizing instruction such as shown in Listing 6-2. When the processor encounters the isync instruction, it waits until all partially executed instructions in the linear instruction stream complete before it continues execution. In this case, we want to be sure that the DBAT register is properly set up before trying to access an address that will be translated by that DBAT.

**Listing 6-2**

Context synchronization orders memory access.

```markdown
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addis r3,r0,0xf800</td>
<td>instruction #1</td>
</tr>
<tr>
<td>addi r3,r3,0x0010</td>
<td>instruction #2</td>
</tr>
<tr>
<td>mtdbatl 0,r5</td>
<td>instruction #3</td>
</tr>
<tr>
<td>mtdbatu 0,r4</td>
<td>instruction #4</td>
</tr>
<tr>
<td>isync</td>
<td>instruction #5 <em>context synchronizing</em></td>
</tr>
<tr>
<td>lwz r6,0(r3)</td>
<td>instruction #6</td>
</tr>
</tbody>
</table>
```

Some PowerPC instructions, such as mtmsr and rfi, perform context synchronization as a side effect of their execution and no additional synchronization is necessary. Additionally, these instructions can be used to the same end as the isync instruction in Listing 6-2. Any exception that is recoverable is also context synchronizing.

So when is synchronization necessary? The example shown in Listing 6-2 demonstrates that synchronization is required after any operation that alters the context in which code is executing. Operations that modify system registers or perform cache management operations can alter the current system context. Table 6-1 shows the PowerPC instructions that require context synchronization after they are used. A context synchronizing event (CSE) is one of the following: execution of an sc, rfi, sync, or isync instruction, or a context synchronizing exception.
Table 6-1
PowerPC Operations that Require Synchronization and Synchronization Suggestions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation Category</th>
<th>Synchronization Required Before Instruction</th>
<th>Synchronization Required After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtmsr[SF,PR,FP,ME,DR,SE,BE,IR]</td>
<td>Register access</td>
<td>None</td>
<td>CSE</td>
</tr>
<tr>
<td>mtmsr[LE, ILE]</td>
<td>Register access</td>
<td>None (see example code in Chapter 3)</td>
<td>None (see example code in Chapter 3)</td>
</tr>
<tr>
<td>mtmsr[POW]</td>
<td>Register access</td>
<td>Processor implementation dependent</td>
<td>Processor implementation dependent</td>
</tr>
<tr>
<td>mtsr</td>
<td>Register access</td>
<td>CSE</td>
<td>CSE</td>
</tr>
<tr>
<td>mtspr [ASR,IBAT, DBAT,DABR,EAR]</td>
<td>Register access</td>
<td>CSE</td>
<td>CSE</td>
</tr>
<tr>
<td>mtspr [SDR1]</td>
<td>Register access</td>
<td>CSE, sync</td>
<td>CSE</td>
</tr>
<tr>
<td>slbie, slbia,</td>
<td>Instruction</td>
<td>CSE</td>
<td>CSE, sync</td>
</tr>
<tr>
<td>ttbie, ttbia</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Because PowerPC processors use different resources according to the type of operation being performed, there are different instructions used to perform context synchronization. Unique PowerPC instructions exist to perform instruction execution synchronization, data access synchronization, and I/O synchronization.

The isync instruction flushes any prefetched instructions from the processor's instruction queue and waits for any issued instructions to complete. In other words, executing isync flushes both the processor's pipeline and its instruction queue.

The sync instruction behaves similarly to the isync instruction, but waits additionally until all pending memory accesses are completed. Because sync waits for both instruction execution and memory accesses, it can take a significant amount of time to complete and should be used prudently. The delay depends on the number and type of outstanding operations.

The eieio (Enforce In-order Execution of IO) instruction ensures that I/O operations occur in program order. This is crucial for I/O devices that require a specific sequence of accesses, such as the initialization of an external peripheral device. On the 603, the eieio instruction is treated as a no-op because the 603 does not reorder non-cacheable memory accesses such as I/O.
Do You Have Reservations?

The final topic related to synchronization is that of reservations. On PowerPC processors, a reservation can be thought of as a semaphore that ensures an atomic memory access. An atomic memory access is a read-modify-write sequence to a particular address that occurs with the guarantee that no other device (processor or peripheral) has modified that address. There are two pairs of PowerPC instructions that use reservations: lwarx/stwcx. and ldarx/stdcx. These four instructions must be used in pairs to achieve atomic memory accesses. For example, lwarx (load word and reserve indexed) creates the reservation and stwcx. (store word conditional indexed) performs a conditional store operation to achieve an atomic memory access. This process is described in detail below.

At most, one reservation can exist at a time on a processor. When an instruction that creates a reservation is executed, the previous reservation is replaced. The execution of a stwcx. instruction or modification of a reserved address will clear the current reservation.

A sequence of events that uses a reservation might proceed as follows:

- The routine that is running needs to perform an atomic word access to address \( n \). To guarantee that the access is atomic, the routine loads the word from \( n \) using the lwarx instruction.
- The access to address \( n \) using the lwarx instruction creates a semaphore based on that address.
- The routine modifies the word loaded from address \( n \) and is ready to store it back to the same address.
- Using the stwcx. instruction and address \( n \) as the destination, the store operation checks the semaphore to determine if the address has been modified since being read. If no modification has occurred, the store completes and the CR0[EQ] bit is set. If address \( n \) has been modified, the store does not complete and CR0[EQ] is cleared.

Instruction Suffixes

Instruction suffixes indicate which registers and bits are updated with the results of the instruction’s execution. Table 6-2 shows the suffixes that may be used with PowerPC instructions and the bits that are affected by each form.
Instructions may variously accept none, some, or all of the suffixes listed in Table 6-2. For notational convenience, these instruction mnemonics have an ‘x’ appended to them. For example, the addx instruction can be formed with no suffix or any of the three available suffixes. The specific suffix forms available for a particular instruction are given in the entry for that instruction in Appendix A, “PowerPC Instruction Set Reference.”

<table>
<thead>
<tr>
<th>Instruction Suffix</th>
<th>Suffix Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>(none)</td>
<td>(none)</td>
<td>No register updates</td>
</tr>
<tr>
<td>.</td>
<td>&quot;dot&quot;</td>
<td>Enables the update of the control register (CR).</td>
</tr>
<tr>
<td>o</td>
<td>&quot;oh&quot;</td>
<td>Enables the update of the XER[OV] bit.</td>
</tr>
<tr>
<td>o.</td>
<td>&quot;oh-dot&quot;</td>
<td>Enables the update of both the CR and XER[OV].</td>
</tr>
</tbody>
</table>

**Illegal Forms**

Just as some PowerPC instructions have a preferred form, other forms are prohibited and generate an exception if used. There are five types of illegal instructions for PowerPC processors:

- An instruction that sets instruction-defined or reserved bits differently from the definition in Appendix A, “PowerPC Instruction Set Reference.”
- An instruction form that uses an undefined primary opcode. Currently, the undefined primary opcodes are: 0x01, 0x04, 0x05, 0x06, 0x38, 0x39, 0x3c, 0x3d.
- An instruction that is not defined for a particular PowerPC processor implementation. For example, executing 64-bit-only 620 instructions on any of the 32-bit implementations is undefined.
- An instruction form that uses an undefined extended opcode. Currently, all extended opcodes not described in Appendix A are undefined.
- An instruction of the form 0x00000000. This restriction proves useful when misbehaved code attempts to execute uninitialized data, such as a NULL pointer.
Executing an illegal instruction generates an illegal instruction exception. This behavior provides a mechanism for the processor to emulate the illegal instruction in the exception handler. For further information on illegal instruction exceptions, refer to Chapter 10, “Exceptions and Interrupts.”

**Programming Point: Preferred Instruction Forms**

Some instructions, such as the load/store multiple instructions, load/store string instructions, and the OR immediate instruction can be encoded in more than one form. Often, one specific form, known as the preferred form, executes more efficiently than the other forms of the same instruction. Where instructions have a preferred form, it is noted in the instruction definition found in Appendix A, “PowerPC Instruction Set Reference.” Additionally, forms of integer instructions that update the carry bit (XER[CA]) or enable the overflow option (XER[OV]) may delay subsequent instructions.

**INSTRUCTION CATEGORIES**

The integer unit, the branch unit, load/store unit, and the floating-point unit each execute a subset of the total PowerPC instruction set. However, each PowerPC implementation can have a unique set of execution units and the unit that executes the instruction may vary from implementation to implementation. For example, the 604 and 620 have three integer units.

**Integer Instructions**

The set of PowerPC integer instructions can be divided into five subcategories based on function: arithmetic, compare, logical, rotate/shift, and load/store. Most integer instructions are executed by the integer unit. On the PowerPC 603, 604, and 620 processors, however, the load/store execution unit executes the integer load/store instructions. Integer instructions use general-purpose registers (GPRs) as both source and destination operands. And, unless otherwise specified, integer instructions treat source operands as signed integer values.
The instruction description tables that follow use a simple format: The first column shows the instruction name; the second shows the instruction form and operands; the final column gives a brief description of the instruction’s operation. Note that the description is not intended to fully specify the function of the instruction. Instruction details may be found in Appendix A, “PowerPC Instruction Set Reference.”

**Integer Arithmetic Instructions**

Integer arithmetic instructions perform non-floating-point arithmetic. Recall that only load and store instructions can access memory; values that are operated on by arithmetic instructions must first be loaded into a GPR. Table 6-3 lists the integer arithmetic instructions by function, their mnemonic and operands, and a description of their operation.

**Table 6-3**

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>add carrying</td>
<td>addcx rD,rA,rB</td>
<td>rD = (rA + rB)</td>
</tr>
<tr>
<td>add extended</td>
<td>addex rD,rA,rB</td>
<td>rD = (rA + rB + XE[RCA])</td>
</tr>
<tr>
<td>add immediate</td>
<td>addi rD,rA,SI MM</td>
<td>rD = (rA + SI MM)</td>
</tr>
<tr>
<td>add immediate</td>
<td>addicx rD,rA,SI MM</td>
<td>rD = (rA + SI MM)</td>
</tr>
<tr>
<td>add immediate</td>
<td>addis rD,rA,SI MM</td>
<td>rD = (rA + [SI MM &lt;&lt; 16])</td>
</tr>
<tr>
<td>add to minus one</td>
<td>addmex rD,rA</td>
<td>rD = (rA + XE[RCA] - 1)</td>
</tr>
<tr>
<td>add</td>
<td>addx rD,rA,rB</td>
<td>rD = (rA + rB)</td>
</tr>
<tr>
<td>add to zero extended</td>
<td>addzex rD,rA</td>
<td>rD = (rA + XE[RCA])</td>
</tr>
<tr>
<td>divide double word</td>
<td>divdux rD,rA,rB</td>
<td>64-bit only</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The 64-bit quotient rA/rB is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>placed into rD. The contents</td>
</tr>
<tr>
<td></td>
<td></td>
<td>of rA and rB are interpreted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>as 64-bit unsigned integers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The remainder is not supplied</td>
</tr>
<tr>
<td></td>
<td></td>
<td>as a result.</td>
</tr>
</tbody>
</table>
### Table 6-3
**Integer Arithmetic Instructions (Continued)**

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>divide double word</td>
<td>divdx rD,rA,rB</td>
<td>64-bit only</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The quotient rA/rB is placed into rD. The contents of rA and rB are interpreted as 64-bit signed integers. The remainder is not supplied as a result.</td>
</tr>
<tr>
<td>divide word</td>
<td>divwx rD,rA,rB</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The quotient rA/rB is placed into rD. The contents of rA and rB are interpreted as 32-bit signed integers. The remainder is not supplied as a result.</td>
</tr>
<tr>
<td>divide word unsigned</td>
<td>divwux rD,rA,rB</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The contents of rA and rB are interpreted as 32-bit unsigned integers. The 32-bit quotient rA/rB is placed into rD. The remainder is not supplied as a result.</td>
</tr>
<tr>
<td>multiply low immediate</td>
<td>mulli rD,rA,SIMM</td>
<td>32-bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The low-order 32 bits of the 64-bit product (rA * SIMM) are placed into rD.</td>
</tr>
<tr>
<td>multiply low</td>
<td>mullwx rD,rA,rB</td>
<td>64-bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The low-order 64 bits of the 128-bit product (rA * sign extend (SIMM)) are placed into rD.</td>
</tr>
<tr>
<td>multiply low double word</td>
<td>mulldx rD,rA,rB</td>
<td>64-bit only</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The low-order 64 bits of the 128-bit product (rA * rB) are placed into rD.</td>
</tr>
<tr>
<td>Instruction Name</td>
<td>Mnemonic and Operands</td>
<td>Operation</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>multiply high word</td>
<td>mulhwx rd,rA,rB</td>
<td>The high-order 32 bits of the signed product of the 32-bit signed integers rA and rB are placed into rD.</td>
</tr>
<tr>
<td>multiply high double word</td>
<td>mulhdx rd,rA,rB</td>
<td>64-bit only The high-order 64 bits of the signed product of the 64-bit signed integers rA and rB are placed into rD.</td>
</tr>
<tr>
<td>multiply high word unsigned</td>
<td>mulhwux rd,rA,rB</td>
<td>The high-order 32 bits of the unsigned product of the 32-bit unsigned integers rA and rB are placed into rD.</td>
</tr>
<tr>
<td>multiply high double word unsigned</td>
<td>mulhdux rd,rA,rB</td>
<td>64-bit only The high-order 64 bits of the unsigned product of the 64-bit unsigned integers rA and rB are placed into rD.</td>
</tr>
<tr>
<td>negate</td>
<td>negx rd,rA</td>
<td>rD = ((NOT rA) + 1) (two’s complement)</td>
</tr>
<tr>
<td>subtract from carrying</td>
<td>subfcx rd,rA,rB</td>
<td>rD = (rB - rA)</td>
</tr>
<tr>
<td>subtract from extended</td>
<td>subfex rd,rA,rB</td>
<td>rD = ((NOT rA) + rB + XER[CA])</td>
</tr>
<tr>
<td>subtract from immediate carrying</td>
<td>subfic rd,rA,IMM</td>
<td>rD = (SIMM - rA)</td>
</tr>
<tr>
<td>subtract from minus one extended</td>
<td>subfmex rd,rA</td>
<td>rD = ((NOT rA) + XER[CA] - 1)</td>
</tr>
<tr>
<td>subtract from zero extended</td>
<td>subfx rd,rA,rB</td>
<td>rD = (rB - rA)</td>
</tr>
<tr>
<td></td>
<td>subfzex rd,rA</td>
<td>rD = ((NOT rA) + XER[CA])</td>
</tr>
</tbody>
</table>
Integer Compare Instructions

Integer compare instructions perform either signed or unsigned comparisons based on the form of the instructions. The logical forms of the compare instructions, such as `cmp` and `cmpi`, perform unsigned comparisons. The algebraic forms, such as `cmp` and `cmpi`, perform signed comparisons. The results from a comparison are stored in the CR field designated by the `crFD` operand. However, if the `crFD` operand is omitted, the results will be stored in CR0. Table 6-4 lists the integer compare instructions, their mnemonic and operands, and a description of their operation.

<table>
<thead>
<tr>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cmpi crFD,l,rA,SIWM</code></td>
<td>A signed comparison is made between rA and the sign-extended value of SIWM.</td>
</tr>
<tr>
<td><code>cmp crFD,l,rA,rB</code></td>
<td>A signed comparison is made between rA and rB.</td>
</tr>
<tr>
<td><code>cmpi crFD,l,rA,UIWM</code></td>
<td>An unsigned comparison is made between rA and the zero-extended value of UIWM.</td>
</tr>
<tr>
<td><code>cmpi crFD,l,rA,rB</code></td>
<td>An unsigned comparison is made between rA and rB.</td>
</tr>
</tbody>
</table>

Integer Logical Instructions

Integer logical instructions perform bit manipulation on values contained within general-purpose registers. The “dot” form of integer logical instructions update the CR0 field of the condition register and are explicitly noted. Table 6-5 lists the integer comparison instructions and briefly describes their functionality. Table 6-5 lists the integer logical instructions by function, their mnemonic and operands, and a description of their operation.

Integer Rotate and Shift Instructions

Integer rotate and shift instructions provide a variety of ways to manipulate the bits of values contained in general-purpose registers. Table 6-6 lists the integer rotate instructions by function, their mnemonic and operands, and a description of their operation. Table 6-7 does the same for the integer shift instructions.
<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>andx rA,rS,rB</td>
<td>rA = (rS AND rB)</td>
</tr>
<tr>
<td>AND immediate</td>
<td>andi rA,rS,UIMM</td>
<td>rA = (rS AND UIMM)</td>
</tr>
<tr>
<td>AND immediate</td>
<td>andis rA,rS,UIMM</td>
<td>rA = (rS AND (UIMM &lt;&lt; 16))</td>
</tr>
<tr>
<td>NAND</td>
<td>nandx rA,rS,rB</td>
<td>rA = (NOT (rS AND rB))</td>
</tr>
<tr>
<td>AND with complement</td>
<td>andcx rA,rS,rB</td>
<td>rA = (rS AND (NOT rB))</td>
</tr>
<tr>
<td>OR</td>
<td>orx rA,rS,rB</td>
<td>rA = (rS OR rB)</td>
</tr>
<tr>
<td>OR immediate</td>
<td>ori rA,rS,UIMM</td>
<td>rA = (rS OR UIMM)</td>
</tr>
<tr>
<td>OR immediate shifted</td>
<td>oris rA,rS,UIMM</td>
<td>rA = (rS OR (UIMM &lt;&lt; 16))</td>
</tr>
<tr>
<td>NOR</td>
<td>norx rA,rS,rB</td>
<td>rA = (NOT (rS OR rB))</td>
</tr>
<tr>
<td>OR with complement</td>
<td>orcx rA,rS,rB</td>
<td>rA = (rS OR (NOT rB))</td>
</tr>
<tr>
<td>XOR</td>
<td>xorx rA,rS,rB</td>
<td>rA = (rS XOR rB)</td>
</tr>
<tr>
<td>XOR immediate</td>
<td>xor i rA,rS,UIMM</td>
<td>rA = (rS XOR UIMM)</td>
</tr>
<tr>
<td>XOR immediate shifted</td>
<td>xoris rA,rS,UIMM</td>
<td>rA = (rS XOR (UIMM &lt;&lt; 16))</td>
</tr>
<tr>
<td>equivalent</td>
<td>eqvx rA,rS,UIMM</td>
<td>rA = (NOT (rS XOR rB))</td>
</tr>
<tr>
<td>extend sign byte</td>
<td>exts bx rA,rS</td>
<td>The contents of the low-order eight bits of rS are placed into the low-order eight bits of rA, treated as a signed value, and sign extended to the high-order bits of rA.</td>
</tr>
<tr>
<td>extend sign half-word</td>
<td>extsh x rA,rS</td>
<td>The contents of the low-order 16 bits of rS are placed into the low-order 16 bits of rA, treated as a signed value, and sign extended to the high-order bits of rA.</td>
</tr>
</tbody>
</table>
### Table 6-5
Integer Logical Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>extend sign word</td>
<td>extswx rA,rS</td>
<td>The contents of the low-order 32 bits of rS are placed into the low-order 32 bits of rA, treated as a signed value, and sign extended to the high-order bits of rA.</td>
</tr>
<tr>
<td>count leading zeros word</td>
<td>cntlzwx rA,rS</td>
<td>32-bit A count of the number of consecutive zero bits starting at bit 0 of rS is placed into rA.</td>
</tr>
<tr>
<td>count leading zeros double word</td>
<td>cntlzdxx rA,rS</td>
<td>64-bit only A count of the number of consecutive zero bits starting at bit 0 of rS is placed into rA. This number ranges from 0 to 64, inclusive.</td>
</tr>
</tbody>
</table>

### Table 6-6
Integer Rotate Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>rotate left double word immediate then clear left</td>
<td>rldicxx rA,rS,SH,MB</td>
<td>64-bit only rS is rotated left SH bits. The result is ANDed with the generated mask and placed into rA.</td>
</tr>
<tr>
<td>rotate left double word immediate then clear right</td>
<td>rldicrx rA,rS,SH,ME</td>
<td>64-bit only rS is rotated left SH bits. The result is ANDed with the generated mask and placed into rA.</td>
</tr>
<tr>
<td>rotate left double word immediate then clear</td>
<td>rldicx rA,rS,SH,MB</td>
<td>64-bit only rS is rotated left SH bits. The result is ANDed with the generated mask and placed into rA.</td>
</tr>
</tbody>
</table>
### Table 6-6
Integer Rotate Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>rotate left double word</td>
<td>rldclx rA,rS,rB,MB</td>
<td>64-bit only</td>
</tr>
<tr>
<td>then clear left</td>
<td></td>
<td>rS is rotated left the number of bits specified in the low-order six bits of rB. The result is ANDed with the generated mask and placed into rA.</td>
</tr>
<tr>
<td>rotate left double word</td>
<td>rldcrex rA,rS,rB,ME</td>
<td>64-bit only</td>
</tr>
<tr>
<td>then clear right</td>
<td></td>
<td>rS is rotated left the number of bits specified in the low-order six bits of rB. The result is ANDed with the generated mask and placed into rA.</td>
</tr>
<tr>
<td>rotate left word</td>
<td>rlwimx rA,rS,SH,MB,ME</td>
<td>rS is rotated left SH bits. The result is ANDed with the generated mask and placed into rA.</td>
</tr>
<tr>
<td>immediate then AND with</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mask</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rotate left word</td>
<td>rlwimx rA,rS,rB,MB,ME</td>
<td>rS is rotated left the number of bits specified in the low-order five bits of rB. The rotated word is ANDed with the generated mask and the result is placed into rA.</td>
</tr>
<tr>
<td>immediate then mask insert</td>
<td>rlwimx rA,rS,SH,MB,ME</td>
<td>rS is rotated left by SH bits. The result is inserted into rA under control of the generated mask.</td>
</tr>
<tr>
<td>rotate left double word</td>
<td>rldmix rA,rS,SH,MB</td>
<td>64-bit only</td>
</tr>
<tr>
<td>word immediate then</td>
<td></td>
<td>rS is rotated left SH bits. The result is inserted into rA under control of the generated mask.</td>
</tr>
<tr>
<td>mask insert</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Some rotate and shift instructions generate a bit mask. The size of the mask that is generated depends on the *mask begin* (MB) and the *mask end* (ME) operands. As shown in Figure 6-1, the creation of the mask depends on the relationship between MB and ME. If MB ≤ ME, the mask consists of 1’s from MB to ME, inclusive. If MB > ME, the mask is generated by setting all bits from 0 to ME and MB to 31.
Table 6-7
Integer Shift Instructions

<table>
<thead>
<tr>
<th>Instruction Category</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>shift left</td>
<td>sldx rA,rS,iB</td>
<td>64-bit only</td>
</tr>
<tr>
<td>double word</td>
<td></td>
<td>rS is shifted left the number of bits specified in the low-order seven bits of rB. Bits shifted out of position 0 are lost. Zeros are shifted in on the right. The result is placed into rA.</td>
</tr>
<tr>
<td>shift left word</td>
<td>slwx rA,rS,iB</td>
<td>32-bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rS is shifted left the number of bits specified in the low-order six bits of rB. Bits shifted out of position 0 are lost. Zeros are shifted in on the right. The 32-bit result is placed into rA.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>64-bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The low-order 32 bits of rS are shifted left the number of bits specified in the low-order six bits of rB. Bits shifted out of position 32 are lost. Zeros are shifted in on the right. The 32-bit result is placed into rA.</td>
</tr>
<tr>
<td>shift right</td>
<td>srdx rA,rS,iB</td>
<td>64-bit only</td>
</tr>
<tr>
<td>double word</td>
<td></td>
<td>rS is shifted right the number of bits specified in the low-order seven bits of rB. Bits shifted out of position 63 are lost. Zeros are shifted in on the left. The result is placed into rA.</td>
</tr>
<tr>
<td>shift right word</td>
<td>srwx rA,rS,iB</td>
<td>32-bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rS is shifted right the number of bits specified in the low-order six bits of rB. Bits shifted out of position 31 are lost. Zeros are shifted in on the left. The 32-bit result is placed into rA.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>64-bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The low-order 32 bits of rS are shifted right the number of bits specified in the low-order six bits of rB. Bits shifted out of position 63 are lost. Zeros are shifted in on the left. The 32-bit result is placed into rA.</td>
</tr>
</tbody>
</table>
### Table 6-7
Integer Shift Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Category</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>shift right</td>
<td>sradx rA,rS,SH</td>
<td>64-bit only</td>
</tr>
<tr>
<td>algebraic double</td>
<td></td>
<td></td>
</tr>
<tr>
<td>word immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sradix rA,rS,SH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rS is shifted SH bits. Bits shifted out of position 63 are lost. Copies of bit 0 of rS are shifted in on the left. The result is placed into rA.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>shift right</td>
<td>srawix rA,rS,SH</td>
<td>32-bit</td>
</tr>
<tr>
<td>algebraic word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>srawix rA,rS,SH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The low-order 32 bits of rS are shifted right SH bits. Bits shifted out of position 31 are lost. The 32-bit result is sign extended and placed into rA.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>shift right</td>
<td>sradx rA,rS,rB</td>
<td>64-bit only</td>
</tr>
<tr>
<td>algebraic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>double word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sradx rA,rS,rB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The low-order 32 bits of rS are shifted right the number of bits specified in the low-order seven bits of rB. Bits shifted out of position 63 are lost. Copies of bit 0 of rS are shifted in on the left. The result is placed into rA.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>shift right</td>
<td>srawx rA,rS,rB</td>
<td>32-bit</td>
</tr>
<tr>
<td>algebraic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>srawx rA,rS,rB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The low-order 32 bits of rS are shifted right the number of bits specified in the low-order 5 bits of rB. Copies of bit 0 of rS are shifted in on the left. The result is placed into rA.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The 64-bit 620 generates masks using the same algorithm except that the ending bit position is 63. Instructions that specify only an ME operand generate a mask by setting all bits from 0 to ME and clearing the remaining
bits. Instructions that specify only an MB operand generate a mask by setting all bits from MB to 31 (63 on 64-bit implementations) and clearing the remaining bits.

**EXAMPLE #1: MB <= ME**
```
rlwlml r3,r4,4,28,31
```

![Mask Diagram](image)

In Example #1, the MASK is formed by setting all bits \( \text{MASK}[\text{MB-ME}] \) equal to 1 and clearing all other bits.

\[ \text{MASK}[\text{MB-ME}] = 1 \]

**EXAMPLE #2: MB > ME**
```
rlwlml r3,r4,4,28,21
```

![Mask Diagram](image)

In Example #2, the MASK is formed by setting all bits from \( \text{MASK}[0-\text{ME}] \) equal to 1, \( \text{MASK}[\text{MB-31}] \) equal to 1, and clearing all other bits. On 64-bit implementations, the ending bit number is 63.

\[ \text{MASK}[0-\text{ME}] = 1 \quad \text{MASK}[\text{MB-31}] = 1 \]

**Figure 6-1**
The generation of bit masks for rotate and shift instructions depends on the ME, MB, and SH fields.
Integer Load and Store Instructions

Several times now, I’ve mentioned that the only way to access and modify memory values on PowerPC processors is to load the value from memory into a register, operate on that value, and store it back out. The integer load and store instructions described here are the key to memory access. The concepts associated with the generation of effective addresses, covered in Chapter 5, “Addressing Modes and Operand Conventions,” are used extensively in the discussion that follows. Refer to Chapter 5 for a complete discussion of PowerPC addressing modes and effective address generation.

Integer load and store instructions are related to floating-point load and store operations because floating-point data contained in floating-point registers (FPRs) is loaded/stored to addresses contained in general-purpose registers (GPRs) — the domain of the integer unit.

Some integer load and store operations have an explicit update form. In particular, after a value is loaded from or stored to an effective address in memory, these instructions update the rA register operand with the effective address used for the operation. This can be a useful feature when accessing arrays or structures with constant offsets in memory.

There are two types of PowerPC instructions that are useful when moving blocks of memory. The load/store multiple word instructions (lmw, stmw) and the load/store string instructions (lswi, lswx, stswi, stswx) perform similar functions — with one important exception. The load/store multiple word instructions require an aligned effective address; using an EA that is not 32-bit aligned will generate an alignment exception. The load/store string instructions do not have alignment requirements and are useful in cases where misalignment is possible. The multiple word and string instructions may not represent the most efficient way to perform memory-block operations; when using these instructions, empirical tests may be used to determine the most efficient memory-block operation implementations.

Table 6-8 lists the integer load instructions by function, their mnemonic and operands, and a description of their operation. Table 6-9 does the same for the integer store instructions. Table 6-10 lists the load/store instructions, including those that facilitate data manipulation in bi-endian environments, and Table 6-11 displays multiple word instructions.
Table 6-8
Integer Load and Store Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>load byte and zero</td>
<td>lbz rD,d(rA)</td>
<td>The byte in memory addressed by (rA + d) is loaded into the low-order eight bits of rD. The remaining bits in rD are cleared.</td>
</tr>
<tr>
<td>load byte and zero indexed</td>
<td>lbzx rD,rA,rB</td>
<td>The byte in memory addressed by (rA + rB) is loaded into the low-order eight bits of rD. The remaining bits in rD are cleared.</td>
</tr>
<tr>
<td>load byte and zero with update</td>
<td>lbzu rD,d(rA)</td>
<td>The byte in memory addressed by (rA + d) is loaded into the low-order eight bits of rD. The remaining bits in rD are cleared. rA is updated with (rA + d)</td>
</tr>
<tr>
<td>load byte and zero with update indexed</td>
<td>lbzux rD,rA,rB</td>
<td>The byte in memory addressed by (rA + rB) is loaded into the low-order eight bits of rD. The remaining bits in rD are cleared. rA is updated with (rA + rB).</td>
</tr>
<tr>
<td>load half-word and zero</td>
<td>lhz rD,d(rA)</td>
<td>The half-word in memory addressed by (rA + d) is loaded into the low-order 16 bits of rD. The remaining bits in rD are cleared.</td>
</tr>
<tr>
<td>load half-word and zero indexed</td>
<td>lhzx rD,rA,rB</td>
<td>The half-word in memory addressed by (rA + rB) is loaded into the low-order 16 bits of rD. The remaining bits in rD are cleared.</td>
</tr>
<tr>
<td>load half-word and zero with update</td>
<td>lhzu rD,d(rA)</td>
<td>The half-word in memory addressed by (rA + d) is loaded into the low-order 16 bits of rD and the remaining bits in rD are cleared. rA is updated with (rA + d).</td>
</tr>
<tr>
<td>load half-word and zero with update indexed</td>
<td>lhzux rD,rA,rB</td>
<td>The half-word in memory addressed by (rA + rB) is loaded into the low-order 16 bits of rD. The remaining bits in rD are cleared. rA is updated with (rA + rB).</td>
</tr>
<tr>
<td>Instruction Name</td>
<td>Mnemonic and Operands</td>
<td>Description of Operation</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------------</td>
<td>------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>load half-word algebraic</td>
<td>lha rD,d(rA)</td>
<td>The half-word in memory addressed by (rA + d) is loaded into the low-order 16 bits of rD. The remaining bits in rD are filled with a copy of the most significant bit of the loaded half-word.</td>
</tr>
<tr>
<td>load half-word algebraic indexed</td>
<td>lhax rD,rA,rB</td>
<td>The half-word in memory addressed by (rA + rB) is loaded into the low-order 16 bits of rD. The remaining bits in rD are filled with a copy of the most significant bit of the loaded half-word.</td>
</tr>
<tr>
<td>load half-word algebraic with update</td>
<td>lhau rD,d(rA)</td>
<td>The half-word in memory addressed by (rA + d) is loaded into the low-order 16 bits of rD. The remaining bits in rD are filled with a copy of the most significant bit of the loaded half-word. rA is updated with (rA + d).</td>
</tr>
<tr>
<td>load half-word algebraic with update indexed</td>
<td>lhaux rD,rA,rB</td>
<td>The half-word in memory addressed by (rA + rB) is loaded into the low-order 16 bits of rD. The remaining bits in rD are filled with a copy of the most significant bit of the loaded half-word. rA is updated with (rA + rB).</td>
</tr>
<tr>
<td>load word and zero</td>
<td>lwz rD,d(rA)</td>
<td>32-bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The word in memory addressed by (rA + d) is loaded into the low-order 32 bits of rD.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>64-bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The word in memory addressed by (rA + d) is loaded into the low-order 32 bits of rD. The remaining bits in the high-order 32 bits of rD are cleared.</td>
</tr>
</tbody>
</table>
### Table 6-8
Integer Load and Store Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
</table>
| load word and zero indexed| lwz x rD,rA,rB        | 32-bit  
The word in memory addressed by (rA + rB) is loaded into the low-order 32 bits of rD. |
|                           |                       | 64-bit  
The word in memory addressed by (rA + rB) is loaded into the low-order 32 bits of rD. The remaining bits in high-order 32 bits of rD are cleared. |
| load word and zero with update| lwzu rD,d(rA)       | 32-bit  
The word in memory addressed by (rA + d) is loaded into the low-order 32 bits of rD. rA is updated with (rA + d). |
|                           |                       | 64-bit  
The word in memory addressed by (rA + d) is loaded into the low-order 32 bits of rD. rA is updated with (rA + d). The remaining bits in the high-order 32 bits of rD are cleared. |
| load word zero with update indexed| lzu x rD,rA,rB | 32-bit  
The word in memory addressed by (rA + rB) is loaded into the low-order 32 bits of rD. rA is updated with (rA + rB). |
|                           |                       | 64-bit  
The word in memory addressed by (rA + rB) is loaded into the low-order 32 bits of rD. rA is updated with (rA + rB). The remaining bits in the high-order 32 bits of rD are cleared. |
| load word algebraic       | lwa rD,ds(rA)        | 64-bit only  
The word in memory addressed by (rA + ds) is loaded into the low-order 32 bits of rD. The remaining bits in
Table 6-8
Integer Load and Store Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
</table>
| load word algebraic indexed | lwax rD,rA,rB          | 64-bit only  
The word in memory addressed by \((rA + rB)\) is loaded into the low-order 32 bits of \(rD\). The remaining bits in the high-order 32 bits of \(rD\) are filled with a copy of the most significant bit of the loaded word. |
| load double word indexed  | ld rD,ds(rA)            | 64-bit only  
The double word in memory addressed by \((rA + rB)\) is loaded into \(rD\). |
| load double word with update | ldu rD,ds(rA)          | 64-bit only  
The double word in memory addressed by \((rA + ds)\) is loaded into \(rD\). \(rA\) is updated with \((rA + ds)\). |
| load double word with update indexed | ldux rD,rA,rB | 64-bit only  
The double word in memory addressed by \((rA + rB)\) is loaded into \(rD\). \(rA\) is updated with \((rA + rB)\). |
### Table 6-9
Integer Store Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>store byte</td>
<td>stb rS,d(rA)</td>
<td>The contents of the low-order eight bits of rS are stored into the byte in memory addressed by [rA + d].</td>
</tr>
<tr>
<td>store byte indexed</td>
<td>stbx rS,rA,rB</td>
<td>The contents of the low-order eight bits of rS are stored into the byte in memory addressed by [rA + rB].</td>
</tr>
<tr>
<td>store byte with update</td>
<td>stbu rS,d(rA)</td>
<td>The contents of the low-order eight bits of rS are stored into the byte in memory addressed by [rA + d]. rA is updated with [rA + d].</td>
</tr>
<tr>
<td>store byte with update indexed</td>
<td>stbux rS,rA,rB</td>
<td>The contents of the low-order eight bits of rS are stored into the byte in memory addressed by [rA + rB]. rA is updated with [rA + rB].</td>
</tr>
<tr>
<td>store half-word</td>
<td>sth rS,d(rA)</td>
<td>The contents of the low-order 16 bits of rS are stored into the half-word in memory addressed by [rA + d].</td>
</tr>
<tr>
<td>store half-word indexed</td>
<td>sthx rS,rA,rB</td>
<td>The contents of the low-order 16 bits of rS are stored into the half-word in memory addressed by [rA + rB].</td>
</tr>
<tr>
<td>store half-word with update</td>
<td>sthu rS,d(rA)</td>
<td>The contents of the low-order 16 bits of rS are stored into the half-word in memory addressed by [rA + d]. rA is updated with [rA + d].</td>
</tr>
<tr>
<td>store half-word with update indexed</td>
<td>sthu x rS,rA,rB</td>
<td>The contents of the low-order 16 bits of rS are stored into the half-word in memory addressed by [rA + rB]. rA is updated with [rA + rB].</td>
</tr>
<tr>
<td>store word</td>
<td>stw rS,d(rA)</td>
<td>The contents of rS are stored into the word in memory addressed by [rA + d].</td>
</tr>
</tbody>
</table>
Table 6-9
Integer Store Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>store word indexed</td>
<td>stwu rS,rA,rB</td>
<td>The contents of rS are stored into the word in memory addressed by (rA + rB).</td>
</tr>
<tr>
<td>store word with update</td>
<td>stwu rS,d(rA)</td>
<td>The contents of rS are stored into the word in memory addressed by (rA + d). rA is updated with (rA + d).</td>
</tr>
<tr>
<td>store word with update indexed</td>
<td>stwux rS,rA,rB</td>
<td>The contents of rS are stored into the word in memory addressed by (rA + rB). rA is updated with (rA + rB).</td>
</tr>
<tr>
<td>store double word</td>
<td>std rS,ds(rA)</td>
<td>64-bit only The contents of rS are stored into the double word in memory addressed by (rA + ds).</td>
</tr>
<tr>
<td>stored double word indexed</td>
<td>stdx rS,rA,rB</td>
<td>64-bit only The contents of rS are stored into the double word in memory addressed by (rA + rB).</td>
</tr>
<tr>
<td>store double word with update</td>
<td>stdu rS,ds(rA)</td>
<td>64-bit only The contents of rS are stored into the double word in memory addressed by (rA + rB). rA is updated with (rA + rB).</td>
</tr>
<tr>
<td>store double word with update indexed</td>
<td>stdux rS,rA,rB</td>
<td>64-bit only The contents of rS are stored into the double word in memory addressed by (rA + rB). rA is updated with (rA + rB).</td>
</tr>
</tbody>
</table>
### Table 6-10
**Integer Load and Store with Byte Reverse Instructions**

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>load half-word byte reverse indexed</td>
<td>lhbrx rD,rA,rB</td>
<td>The two bytes at effective address (rA + rB) are swapped. The reversed half-word is loaded into the low-order 16-bits of rD.</td>
</tr>
<tr>
<td>store half-word byte reverse indexed</td>
<td>sthbrx rS,rA,rB</td>
<td>The position of two low-order bytes in rS are swapped and stored at effective address (rA + rB).</td>
</tr>
<tr>
<td>load word byte reverse indexed</td>
<td>lwbrx rD,rA,rB</td>
<td>Each byte in the word at effective address (rA + rB) is reversed with respect to the high- and low-order positions. The reversed word is loaded into rD.</td>
</tr>
<tr>
<td>store word byte reverse indexed</td>
<td>stwbrx rS,rA,rB</td>
<td>Each byte in rS is reversed with respect to the high- and low-order positions. The reversed word is stored at effective address (rA + rB).</td>
</tr>
</tbody>
</table>

### Table 6-11
**Integer Multiple Word Instructions**

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>load multiple word</td>
<td>lmw rD,d(rA)</td>
<td>(32−rD) words are loaded from effective address (rA + d) into general-purpose registers starting with rD.</td>
</tr>
<tr>
<td>store multiple word</td>
<td>stmw rS,d(rA)</td>
<td>(32−rS) words are stored from general-purpose registers starting with S to effective address (rA + d).</td>
</tr>
<tr>
<td>load string word immediate</td>
<td>lswi rD,rA,NB</td>
<td>NB bytes are loaded from effective address (rA) into general-purpose registers starting with rS.</td>
</tr>
</tbody>
</table>
Table 6-11  
Integer Multiple Word Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>store string word immediate</td>
<td>stswi rS,rA,NB</td>
<td>NB bytes are stored from general-purpose registers starting with rS to effective address (rA).</td>
</tr>
<tr>
<td>load string word indexed</td>
<td>lswx rD,rA,rB</td>
<td>The number of bytes specified by XER[25-31] are loaded from effective address (rA + rB) into general-purpose registers starting with rS.</td>
</tr>
<tr>
<td>store string word indexed</td>
<td>stswx rS,rA,rB</td>
<td>The number of bytes specified by XER[25-31] are stored from general-purpose registers starting with rS to effective address (rA + rB).</td>
</tr>
</tbody>
</table>

Floating-Point Instructions

The PowerPC instructions that are executed by the floating-point unit use the floating-point registers (FPRs). The floating-point load/store instructions use the general-purpose registers (GPRs) as well. FPRs are used exclusively to hold double-precision floating-point data. GPRs are used to hold the effective address for floating-point loads and stores. In the following sections, floating-point registers have the form frA and general-purpose registers (as before) have the form rA. Appendix C discusses floating-point operation on PowerPC processors in detail.

Floating-point operations on PowerPC processors conform to the IEEE 754 standard unless the processor is placed in a non-standard mode by software. Because FPRs are used exclusively with double-precision floating-point values, a single-precision value must be converted (automatically, by the processor) into double-precision format during the load operation. This and other PowerPC floating-point issues are covered in the glossary at the end of this book.

The set of PowerPC floating-point instructions can be divided into seven subcategories based on functionality: arithmetic, compare, multiply/add, rounding/conversion, FPSCR manipulation, move, and load/store.
### Programming Point: Single- vs. Double-Precision Instructions

On most PowerPC processors, single-precision floating-point instructions will execute faster than their double-precision counterparts. This is especially true for the 601 and 603 processors. Instruction timing and processors-specific instruction execution information is covered in Chapter 7, "The Sublime Art of Instruction Timing."

### Floating-Point Arithmetic Instructions

The floating-point arithmetic instructions, listed in Table 6-12, are analogous to integer arithmetic instructions in the type of operations they perform. These instructions operate exclusively on FPRs. In all cases, the results produced by these instructions are subject to the rounding and normalization rules in effect at the time of their execution.

#### Table 6-12

Floating-Point Arithmetic Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>floating add</td>
<td>faddx frD,frA,frB</td>
<td>frD = (frA + frB)</td>
</tr>
<tr>
<td>(double-precision)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>floating add single</td>
<td>faddsx frD,frA,frB</td>
<td>frD = (frA + frB)</td>
</tr>
<tr>
<td>floating subtract</td>
<td>fsubx frD,frA,frB</td>
<td>frD = (frA – frB)</td>
</tr>
<tr>
<td>(double-precision)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>single</td>
<td></td>
<td></td>
</tr>
<tr>
<td>floating multiply</td>
<td>fmulx frD,frA,frC</td>
<td>frD = (frA * frC)</td>
</tr>
<tr>
<td>(double-precision)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>floating divide</td>
<td>fdivx frD,frA,frB</td>
<td>frD = (frA / frB)</td>
</tr>
<tr>
<td>(double-precision)</td>
<td></td>
<td>No remainder is preserved.</td>
</tr>
<tr>
<td>floating divide single</td>
<td>fdivsx frD,frA,frB</td>
<td>frD = (frA / frB)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No remainder is preserved.</td>
</tr>
</tbody>
</table>
### Table 6-12
Floating-Point Arithmetic Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>floating square root (double-precision)</td>
<td>fsqrtx frD,frB</td>
<td>The square root of frB is placed into frD. (optional instruction)</td>
</tr>
<tr>
<td>floating square root single</td>
<td>fsqrt sx frD,frB</td>
<td>The square root of frB is placed into frD. (optional instruction)</td>
</tr>
<tr>
<td>floating reciprocal estimate single</td>
<td>fresx frD,frB</td>
<td>frD = (1.0 / frB) The estimate placed into frD is correct to a precision of one part in 256 of the reciprocal of frB. (optional instruction)</td>
</tr>
<tr>
<td>floating reciprocal square root estimate</td>
<td>frsqrtex frD,frB</td>
<td>A double-precision estimate of the reciprocal of the square root of frB is placed into frD. The estimate placed into frD is correct to a precision of one part in 32 of the reciprocal of the square root of frB. (optional instruction)</td>
</tr>
<tr>
<td>floating select</td>
<td>fse lx frD,frA,frC,frB</td>
<td>frA is compared to the value zero. If greater than or equal to zero, frD is set to the contents of frC. If the operand is less than zero or is a NaN, frD is set to the contents of frB, ignoring the sign of zero (+O, −O). (optional instruction)</td>
</tr>
</tbody>
</table>

### Floating-Point Compare Instructions

Table 6-13 lists the floating-point compare instructions that are used to compare the contents of two FPRs. Each compare sets one of the four bits in the specified CR field as follows:
CR[FL] set if (frA < frB) (less than)
CR[FG] set if (frA > frB) (greater than)
CR[FE] set if (frA = frB) (equal)
CR[FU] set if (frA ? frB) (unordered)

The remaining three bits that are not explicitly set by the compare operation are cleared.

As described in the glossary, floating-point values can have a sign associated with a value of zero. However, a comparison between +0 and -0 will produce an equal result and set CR[FE].

Table 6-13
Floating-Point Compare Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>floating compare</td>
<td>fcmpu cfrD,frA,frB</td>
<td>frA is compared to frB. The result of the compare is placed into cfrD and FPSCR[16-19].</td>
</tr>
<tr>
<td>unordered</td>
<td></td>
<td></td>
</tr>
<tr>
<td>floating compare</td>
<td>fcmpo cfrD,frA,frB</td>
<td>frA is compared to frB. The result of the compare is placed into cfrD and FPSCR[16-19].</td>
</tr>
<tr>
<td>ordered</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Floating-Point Multiply-Add Instructions

Floating-point multiply-add instructions combine both operations without the intermediate rounding that would be obtained by performing each operation independently. In multiply-add instructions, the entire 106-bit intermediate product takes part in the add portion of the operation. Table 6-14 lists the PowerPC floating-point multiply-add instructions.

Floating-Point Rounding Instructions

Floating-point rounding instructions convert floating-point values from double-precision to single-precision and from floating-point to integer. They also simply round floating-point values. Table 6-15 summarizes the floating-point rounding instructions.
Table 6-14
Floating-Point Multiply-Add Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(double-precision)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subtract (double-precision)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subtract single</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(double-precision)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subtract (double-precision)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>single</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6-15
Floating-Point Rounding Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>floating round to</td>
<td>fxrpx frD,frB</td>
<td>If frB is already in single-precision range, it is placed into frD. Otherwise, frB is rounded to single-precision using the rounding mode specified by FPR[RN] and placed into frD.</td>
</tr>
<tr>
<td>single-precision</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


### Table 6-15
Floating-Point Rounding Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>floating convert from integer double word</td>
<td>fcfdx frD,frB</td>
<td>64-bit only The 64-bit signed integer operand in frB is converted to an infinitely precise FP integer. If the result of the conversion is already in double-precision range, it is placed into frD. Otherwise, the result of the conversion is rounded to double-precision using the rounding mode specified by FPSCR[RN] and placed into frD.</td>
</tr>
<tr>
<td>floating convert to integer double word (64-bit)</td>
<td>fctidx frD,frB</td>
<td>frB is converted to a 64-bit signed integer using the rounding mode specified by FPSCR[RN] and placed into frD.</td>
</tr>
<tr>
<td>floating convert to integer double word with round toward zero</td>
<td>fctidzx frD,frB</td>
<td>frB is converted to a 64-bit signed integer using the rounding mode round toward zero and placed in frD.</td>
</tr>
<tr>
<td>floating convert to integer word</td>
<td>fcwi x frD,frB</td>
<td>frB is converted to a 32-bit signed integer using the rounding mode specified by FPSCR[RN] and placed in the low-order 32 bits of frD. Bits 0–31 of frD are undefined.</td>
</tr>
<tr>
<td>floating convert to integer word with round toward zero</td>
<td>fcwizx frD,frB</td>
<td>frB is converted to a 32-bit signed integer using the rounding mode round toward zero and placed in the low-order 32 bits of frD. Bits 0–31 of frD are undefined.</td>
</tr>
</tbody>
</table>

### Floating-Point FPSCR Instructions

The FPSCR instructions are used to manipulate the floating-point status and control register (FPSCR). Each of the FPSCR instructions listed in Table 6-16 will synchronize execution on the floating-point unit. The synchronization
caused by FPSCR instructions is conceptually similar to the discussion early in this chapter: All floating-point instructions will complete and all updates to the FPSCR by exception events will complete before the FPSCR instruction is executed. However, floating-point load and store instructions are not affected by the execution of any of the FPSCR instructions.

Table 6-16
Floating-Point FPSCR Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>move from FPSCR</td>
<td>mfssx frD</td>
<td>The contents of the FPSCR are placed into bits 32–63 of frD. Bits 0–31 of frD are undefined.</td>
</tr>
<tr>
<td>move to condition register from FPSCR</td>
<td>mcrfs crfD, crfS</td>
<td>The FPSCR field crfS is copied to CR field crfD. All exception bits copied (except FEX and VX) are cleared in FPSCR.</td>
</tr>
<tr>
<td>move to FPSCR field immediate</td>
<td>mfssfix crfD, IMM</td>
<td>The contents of the IMM field are placed into FPSCR field crfD. The contents of FPSCR[FX] are altered only if crfD = 0.</td>
</tr>
<tr>
<td>move to FPSCR fields</td>
<td>mfssfx FM, frB</td>
<td>Bits 32–63 of frB are placed into the FPSCR under control of the field mask specified by FM. The field mask identifies the 4-bit fields affected.</td>
</tr>
<tr>
<td>move to FPSCR bit 0</td>
<td>mfssb0x crbD</td>
<td>Bit crbD of the FPSCR is cleared.</td>
</tr>
<tr>
<td>move to FPSCR bit 1</td>
<td>mfssb1x crbD</td>
<td>Bit crbD of the FPSCR is set.</td>
</tr>
</tbody>
</table>

**Floating-Point Move Instructions**

Floating-point move instructions transfer data between FPRs. With the exception of the fmrr (floating-point move) instruction, all instructions in Table 6-17 alter the floating-point sign bit as specified. All FP move instructions have a “dot” suffix form that will update the CR1 field of the CR register.
Table 6-17
Floating-Point Move Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>floating move register</td>
<td>fmrx frD,frB</td>
<td>frD = frB</td>
</tr>
<tr>
<td>floating negate</td>
<td>fnegx frD,frB</td>
<td>frD = ( - frB)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The contents of frB with bit 0 inverted are placed into frD.</td>
</tr>
<tr>
<td>floating absolute value</td>
<td>fabsx frD,frB</td>
<td>frD =</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The contents of frB with bit 0 cleared are placed into frD.</td>
</tr>
<tr>
<td>floating negative absolute value</td>
<td>fnabsx frD,frB</td>
<td>frD = ( - frB)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The contents of frB with bit 0 set are placed into frD.</td>
</tr>
</tbody>
</table>

Floating-Point Load and Store Instructions

The floating-point load and store instructions share characteristics of both floating-point and integer instructions because the effective address calculation is performed by the integer unit using GPRs. However, it's appropriate to introduce them with the other floating-point instructions. Tables 6-18 and 6-19 list the floating-point load and store instructions, respectively.

Table 6-18
Floating-Point Load Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>load floating-point single</td>
<td>llfs frD,d(rA)</td>
<td>The word in memory addressed by (rA + d) is interpreted as a single-precision operand, converted to FP double-precision format, and placed into frD.</td>
</tr>
</tbody>
</table>
### Table 6-18
Floating-Point Load Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>load floating-point</td>
<td>lfsx frD,rA,rB</td>
<td>The word in memory addressed by (rA + rB) is interpreted as a single-precision operand, converted to FP double-precision format, and placed into frD.</td>
</tr>
<tr>
<td>single indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>load floating-point</td>
<td>lfsu frD,d(rA)</td>
<td>The word in memory addressed by (rA + d) is interpreted as a single-precision operand, converted to FP double-precision format, and placed into frD. rA is updated with (rA + d).</td>
</tr>
<tr>
<td>single with update</td>
<td></td>
<td></td>
</tr>
<tr>
<td>load floating-point</td>
<td>lfsux frD,rA,rB</td>
<td>The word in memory addressed by (rA + rB) is interpreted as a single-precision operand, converted to FP double-precision format, and placed into frD. rA is updated with (rA + rB).</td>
</tr>
<tr>
<td>single with update indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>load floating-point</td>
<td>lfd frD,d(rA)</td>
<td>The double word in memory addressed by (rA + d) is placed into register frD.</td>
</tr>
<tr>
<td>double</td>
<td></td>
<td></td>
</tr>
<tr>
<td>load floating-point</td>
<td>lfdx frD,rA,rB</td>
<td>The double word in memory addressed by (rA + rB) is placed into register frD. rA is updated with (rA + rB).</td>
</tr>
<tr>
<td>double indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>load floating-point</td>
<td>lfdu frD,d(rA)</td>
<td>The double word in memory addressed by (rA + d) is placed into register frD. rA is updated with (rA + d).</td>
</tr>
<tr>
<td>double with update</td>
<td></td>
<td></td>
</tr>
<tr>
<td>load floating-point</td>
<td>lfdux frD,rA,rB</td>
<td>The double word in memory addressed by (rA + rB) is placed into register frD. rA is updated with (rA + rB).</td>
</tr>
<tr>
<td>double with update indexed</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 6-19
Floating-Point Store Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>store floating-point singles</td>
<td>sfs frS,d(rA)</td>
<td>The contents of frS are converted to single-precision and stored into the word in memory addressed by [rA + d].</td>
</tr>
<tr>
<td>store floating-point single indexed</td>
<td>sfsx frS,rA,rB</td>
<td>The contents of frS are converted to single-precision and stored into the word in memory addressed by [rA + rB].</td>
</tr>
<tr>
<td>store floating-point single with update</td>
<td>sfsu frS,d(rA)</td>
<td>The contents of frS are converted to single-precision and stored into the word in memory addressed by [rA + d]. rA is updated with [rA + d].</td>
</tr>
<tr>
<td>store floating-point single with update indexed</td>
<td>sfsux frS,rA,rB</td>
<td>The contents of frS are converted to single-precision and stored into the word in memory addressed by [rA + rB]. rA is updated with [rA + rB].</td>
</tr>
<tr>
<td>store floating-point double</td>
<td>sfid frS,d(rA)</td>
<td>The contents of frS are stored into the double word in memory addressed by [rA + d].</td>
</tr>
<tr>
<td>store floating-point double indexed</td>
<td>sfidx frS,rA,rB</td>
<td>The contents of frS are stored into the double word in memory addressed by [rA + rB].</td>
</tr>
<tr>
<td>store floating-point double with update</td>
<td>sfdu frS,d(rA)</td>
<td>The contents of frS are stored into the double word in memory addressed by [rA + d]. rA is updated with [rA + d].</td>
</tr>
<tr>
<td>store floating-point double with update indexed</td>
<td>sfdux frS,rA,rB</td>
<td>The contents of frS are stored into the double word in memory addressed by [rA + rB]. rA is updated with [rA + rB].</td>
</tr>
<tr>
<td>store floating-point as integer word indexed</td>
<td>stiwx frS,rA,rB</td>
<td>The contents of the low-order 32 bits of frS are stored, without conversion, into the word in memory addressed by [rA + rB]. This instruction is optional in each processor implementation.</td>
</tr>
</tbody>
</table>
Branch Instructions

PowerPC instructions that are executed by the branch processing unit (BPU) change the flow of execution. Depending on the form of the instruction, a branch is taken either unconditionally or based on the state of bits in the condition register. The PowerPC branch instructions can be categorized as relative, absolute, link register, and count register. Each category contains both conditional and unconditional forms of the branch instructions.

PowerPC branch instructions provide the same functionality as x86 call and jump instructions. Using various branch instruction forms, conditional and unconditional transfer of execution as well as calls to subroutines is possible.

The operation of the conditional forms is governed by the BO and BI bit fields in the instruction encoding. Table 6-20 shows the definitions of the BO and BI fields in conditional branch instructions.

The unconditional branch instruction forms have a relatively simple format and only one operand — the target address of the branch. However, the format of conditional branch instructions is considerably more complex. In fact, there are more simplified mnemonics for conditional branching than for any other instruction category. When writing PowerPC assembly language programs (see Chapter 11, "PowerPC Assembly Language Examples"), it is common to use only the simplified form of a conditional branch. Simplified mnemonics for branch instructions are discussed at the end of this chapter.

The BPU uses branch prediction (described in Chapter 7, "The Sublime Art of Instruction Timing") in an attempt to achieve zero-cycle branches by resolving the branch outcome before the instruction is actually executed. Because the implementation of branch prediction varies among processors, we will only refer to branch prediction in generic terms in this section.

As described in Table 6-20, the BI field identifies a bit in the condition register to use as a condition for branch operations. The CR and each of the CR fields is fully defined in Chapter 4, "The PowerPC Programming Model"; however, it is useful to have a CR field reference near the BO and BI encodings for conditional branching. Table 6-21 lists CR bit definitions.

During execution, the processor continually scans the instruction stream for branch instructions in an attempt to resolve branches before they are taken. Upon locating a branch instruction, the processor determines if there is enough information to resolve the branch at that point in time. An unconditional branch can always be resolved (it has no dependencies); a conditional branch depends on values in the condition register. If sufficient
information is available, the branch can be immediately resolved and is removed from the instruction stream. If there is not sufficient information, the processor may attempt to resolve the branch using the hint bit, as described in the next programming point.

Table 6-20

Encoding for the BO and BI Operands

<table>
<thead>
<tr>
<th>BO[0-4] Encoding (5 bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000y</td>
<td>Decrement the CTR. Branch if the decremented CTR does not equal zero and the condition specified by the BI operand is FALSE.</td>
</tr>
<tr>
<td>0001y</td>
<td>Decrement the CTR. Branch if the decremented CTR equals zero and the condition specified by the BI operand is FALSE.</td>
</tr>
<tr>
<td>001xy</td>
<td>Branch if the condition specified by the BI operand is FALSE.</td>
</tr>
<tr>
<td>0100y</td>
<td>Decrement the CTR. Branch if the decremented CTR does not equal zero and the condition specified by the BI operand is TRUE.</td>
</tr>
<tr>
<td>0101y</td>
<td>Decrement the CTR. Branch if the decremented CTR equals zero and the condition specified by the BI operand is TRUE.</td>
</tr>
<tr>
<td>011xy</td>
<td>Branch if the condition specified by the BI operand is TRUE.</td>
</tr>
<tr>
<td>1x00y</td>
<td>Decrement the CTR. Branch if the decremented CTR does not equal zero.</td>
</tr>
<tr>
<td>1x01y</td>
<td>Decrement the CTR. Branch if the decremented CTR equals zero.</td>
</tr>
<tr>
<td>1x1xx</td>
<td>Branch always (unconditional branch).</td>
</tr>
<tr>
<td>nnnnn</td>
<td>The 5-bit BI field specifies the bit in the condition register that is used in the branch conditional test. The 5-bit number is interpreted as a single-bit position ranging from 0 to 31.</td>
</tr>
</tbody>
</table>

Legend

x = Bit is ignored; these positions should be cleared (zero) since future PowerPC processors may define them.
y = Hint bit — used by PowerPC processors as a way to determine if a branch is likely to be taken. Note that the branch always encoding for BO does not have a hint bit.
Table 6-21
Condition Register Bit Definitions

<table>
<thead>
<tr>
<th>CR Bit Position</th>
<th>CR Field</th>
<th>Description (TRUE if set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 1, 2, 3</td>
<td>CR0</td>
<td>Negative (LT), Positive (GT), Zero (EQ), Overflow (SO)</td>
</tr>
<tr>
<td>4, 5, 6, 7</td>
<td>CR1</td>
<td>FP exception (FX)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FP enabled exception (FEX)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FP invalid exception (VX)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FP overflow exception (OX)</td>
</tr>
<tr>
<td>8, 9, 10, 11</td>
<td>CR2</td>
<td>CR2–CR7 have equivalent bit definitions. The four</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bits in each field, from high-order (bit 0) to low</td>
</tr>
<tr>
<td></td>
<td></td>
<td>order (bit 3), are defined as follows:</td>
</tr>
<tr>
<td>12, 13, 14, 15</td>
<td>CR3</td>
<td>Bit 0: Less than (LT) or FP less than (FL)</td>
</tr>
<tr>
<td>16, 17, 18, 19</td>
<td>CR4</td>
<td>Bit 1: Greater than (GT) or FP greater than (FG)</td>
</tr>
<tr>
<td>20, 21, 22, 23</td>
<td>CR5</td>
<td>Bit 2: Equal (EQ) or FP equal (FE)</td>
</tr>
<tr>
<td>24, 25, 26, 27</td>
<td>CR6</td>
<td>Bit 3: Summary overflow (SO) or FP unordered (FU)</td>
</tr>
<tr>
<td>28, 29, 30, 31</td>
<td>CR7</td>
<td></td>
</tr>
</tbody>
</table>

When looking over the various forms of the branch instructions in Table 6-22, notice that some instructions must be coded using the BO and BI fields (described in Table 6-20). Here is an interesting feature to remember: Some values of the BO field allow a single branch instruction to perform multiple operations. For example, if the BO field of a branch instruction is set to 0x0a, the branch operation consists of three independent operations: decrementing the CTR, testing the CTR and CR for a condition, and the actual conditional branch. This means that for some situations (looping is a good example), a single properly coded branch instruction can take the place of three separate instructions. Listing 11-4c (in Chapter 11, “PowerPC Assembly Language Examples”) demonstrates this little optimization.

The following tables are formatted just as the previous sections: The first column contains the instruction name; the second shows the instruction and operands. The final column gives a brief description of the instruction’s operation. Note that the description provided is not intended to fully specify the functionality of the instruction. Each instruction is fully specified in Appendix A, “PowerPC Instruction Set Reference.”
Programming Point: Giving the Processor a Hint

A conditional branch instruction can supply the processor with a "hint" about whether or not a branch is likely to be taken. This is the type of optimization that a compiler is likely to perform for you — and it's one that can really increase performance.

Knowing your code and how it behaves is the most important aspect of the optimizing process. When implementing a loop, for example, we know that the branch back to the start of the loop will be taken every time except once, when the loop terminates. A hint in this case is both appropriate and can save execution time. In Chapter 7, "The Sublime Art of Instruction Timing," we'll discuss other types of branch prediction used on PowerPC processors and how to take advantage of them.

Table 6-22
Branch Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>unconditional branch</td>
<td>b target_address</td>
<td>Branch to the address computed as the sum of the immediate address and the address of the current instruction.</td>
</tr>
<tr>
<td>unconditional branch absolute</td>
<td>ba target_address</td>
<td>Branch to the absolute address specified.</td>
</tr>
<tr>
<td>unconditional branch then link</td>
<td>bl target_address</td>
<td>Branch to the address computed as the sum of the immediate address and the address of the current instruction. The instruction address following this instruction is placed into LR.</td>
</tr>
</tbody>
</table>

A basic concept in the art of writing efficient code is optimize the common case first. Here, we’re given a tremendous opportunity to do just that. When the hint bit is cleared (0) in the BO encoding and the displacement is negative, the branch is predicted as taken. When the hint bit is cleared and the displacement is positive or the branch uses the LR or CTR, the branch is predicted as not taken. If the hint bit is set (1), the previous predictions are reversed. The PowerPC architecture specifies the default setting of the hint bit as cleared (0).

The branch is taken if the following algorithm evaluates as true:

\[
\{(BO[0] \& BO[2]) \ll \text{Sign of displacement}\} = \text{hint bit setting}
\]
### Table 6-22
Branch Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>unconditional branch</td>
<td>bla target_address</td>
<td>Branch to the absolute address specified. The instruction address following this instruction is placed into LR.</td>
</tr>
<tr>
<td>absolute then link</td>
<td></td>
<td></td>
</tr>
<tr>
<td>branch conditional</td>
<td>bc BO,Bl,target_address</td>
<td>Branch conditionally to the address computed as the sum of the immediate address and the address of the current instruction.</td>
</tr>
<tr>
<td>conditional absolute</td>
<td>bca BO,Bl,target_address</td>
<td>Branch conditionally to the absolute address specified.</td>
</tr>
<tr>
<td>branch conditional</td>
<td>bcl BO,Bl,target_address</td>
<td>Branch conditionally to the address computed as the sum of the immediate address and the address of the current instruction. The instruction address following this instruction is placed into LR.</td>
</tr>
<tr>
<td>then link</td>
<td></td>
<td></td>
</tr>
<tr>
<td>branch conditional</td>
<td>bcla BO,Bl,target_address</td>
<td>Branch conditionally to the absolute address specified.</td>
</tr>
<tr>
<td>absolute then link</td>
<td></td>
<td></td>
</tr>
<tr>
<td>branch conditional</td>
<td>bclrl BO,Bl</td>
<td>Branch conditionally to the address specified in the LR.</td>
</tr>
<tr>
<td>to link register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>branch conditional</td>
<td>bcctr BO,Bl</td>
<td>Branch conditionally to the address specified in the count register.</td>
</tr>
<tr>
<td>to count register</td>
<td></td>
<td>Note: If the &quot;decrement and test CTR&quot; option (BO[2] = 0) is specified, the instruction form is invalid.</td>
</tr>
</tbody>
</table>
Table 6-22
Branch Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch conditional to count register then link</td>
<td>bcctrl BO,Bl</td>
<td>Branch conditionally to the address specified in the count register. The instruction address following this instruction is placed into LR. Note: If the “decrement and test CTR” option ([BO[2] = 0]) is specified, the instruction form is invalid.</td>
</tr>
</tbody>
</table>

**Miscellaneous Instructions**

The remaining PowerPC instructions fall into a variety of categories. However, they are all executed by the integer unit. The instruction name is listed in the leftmost column of Tables 6-23 through 6-28. These miscellaneous instructions perform the various operations necessary to maintain a working system: system register manipulation, memory management, synchronization, and cache control.

The usefulness and functionality of the instructions listed in Table 6-18 will be resolved as we work through the next few chapters that deal with memory management and cache operation.

System registers, like general-purpose registers and floating-point registers, are used during the course of normal operation. However, unlike the GPRs and FPRs, not all system registers are accessible by user-level software. In general, system registers contain information concerning the state of currently executing software and the processor configuration. Bit definitions and privilege-level information for each of the registers in Table 6-22 can be found in Chapter 4, “The PowerPC Programming Model.”

PowerPC trap instructions, shown in Table 6-24, perform a similar function to the software interrupt mechanism of the x86 architecture. If the conditions specified in the instruction encoding are met, a trap exception is generated and the system trap handler is invoked. The TO field specifies the relationship used to test the remaining two operands: equivalent, less-than, greater-than, and so on.
### Table 6-23
System Register Manipulation Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition register OR</td>
<td>cror crbD,crbA,crbB</td>
<td>CR[crbD] = (CR[crbA] OR CR[crbB])</td>
</tr>
<tr>
<td>condition register NOR</td>
<td>crnor crbD,crbA,crbB</td>
<td>CR[crbD] = (NOT (CR[crbA] OR CR[crbB]))</td>
</tr>
<tr>
<td>condition register OR</td>
<td>crorc crbD,crbA,crbB</td>
<td>CR[crbD] = (CR[crbA] OR (NOT CR[crbB]))</td>
</tr>
<tr>
<td>condition register XOR</td>
<td>crxor crbD,crbA,crbB</td>
<td>CR[crbD] = (CR[crbA] XOR CR[crbB])</td>
</tr>
<tr>
<td>move condition register field</td>
<td>mcrf crfD,crfS</td>
<td>crfD = crfS</td>
</tr>
<tr>
<td>move to condition register from XER</td>
<td>mcrxr crfD</td>
<td>crfD = XER[0:3] XER[0:3] = 0000</td>
</tr>
<tr>
<td>move from condition register</td>
<td>mfcr rD</td>
<td>The contents of the condition register are placed into the low-order 32 bits of rD. The contents of the high-order 32 bits of rD are cleared in 64-bit implementations.</td>
</tr>
<tr>
<td>move from time base</td>
<td>mftb rD,TBR</td>
<td>rD = TBU or rD = TBL The TBR field denotes either the time base lower (TBL) or time base upper (TBU). The contents of the designated register are copied to rD.</td>
</tr>
</tbody>
</table>
### Table 6-23
System Register Manipulation Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>move to condition register fields</td>
<td>mtcrf CRM,rS</td>
<td>The contents of the low-order 32 bits are placed into the CR under control of the field mask specified by operand CRM. The field mask identifies the 4-bit field affected.</td>
</tr>
<tr>
<td>move to machine state register</td>
<td>mtmsr rS</td>
<td>MSR = rS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This instruction is a supervisor-level instruction and is context synchronizing.</td>
</tr>
<tr>
<td>move from machine state register</td>
<td>mfmsr rD</td>
<td>rD = MSR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This is a supervisor-level instruction.</td>
</tr>
<tr>
<td>move from special-purpose register</td>
<td>mf spr rD,SPR</td>
<td>rD = SPR</td>
</tr>
<tr>
<td>move to special-purpose register</td>
<td>mtspr SPR,rS</td>
<td>SPR = rS</td>
</tr>
<tr>
<td>move to segment register</td>
<td>mtsr SR,rS</td>
<td>32-bit only</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SR = rS</td>
</tr>
<tr>
<td>move to segment register indirect</td>
<td>mtsrin rS,rB</td>
<td>32-bit only</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The contents of rS are copied to the segment register selected by bits 0-3 of rB</td>
</tr>
<tr>
<td>move from segment register</td>
<td>mfsr rD,SR</td>
<td>rD = SR</td>
</tr>
<tr>
<td>move from segment register indirect</td>
<td>mfsrin rD,rB</td>
<td>32-bit only</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The contents of the segment register selected by bits 0-3 of rB are copied into rD.</td>
</tr>
</tbody>
</table>
Table 6-24
Trap Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>trap double word</td>
<td>td TO,rA,rB</td>
<td>64-bit only</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rA is compared with rB. If any bit in the TO operand is set and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.</td>
</tr>
<tr>
<td>trap double word</td>
<td>tdi TO,rA,SIMM</td>
<td>64-bit only</td>
</tr>
<tr>
<td>immediate</td>
<td></td>
<td>rA is compared with the sign extended SIMM operand. If any bit in the TO operand is set and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.</td>
</tr>
<tr>
<td>trap word</td>
<td>tw TO,rA,rB</td>
<td>The low-order 32 bits of rA are compared with the low-order 32 bits of rB. If any bit in the TO operand is set and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.</td>
</tr>
<tr>
<td>trap word immediate</td>
<td>twi TO,rA,SIMM</td>
<td>The low-order 32 bits of rA are compared with the sign extended SIMM operand. If any bit in the TO operand is set and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.</td>
</tr>
</tbody>
</table>

To facilitate use of trap instructions, every possible value of the TO field has a symbolic, simplified encoding. All TO field encodings are described in Table 6-36 in the simplified mnemonics section at the end of this chapter.

Memory synchronization is discussed earlier in this chapter. In general, the PowerPC memory synchronization instructions shown in Table 6-25 control the order in which memory operations (such as load and store accesses) are issued and completed. Synchronization is often required after a change of processor context; the processor's context represents various configuration settings and is discussed in conjunction with synchronization earlier in this chapter.
<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>load double word and reserve indexed</td>
<td>ldarx rD,rA,rB</td>
<td>64-bit only The doubleword in memory addressed by (rA + rB) is loaded into rD.</td>
</tr>
<tr>
<td>load word and reserve indexed</td>
<td>lwarx rD,rA,rB</td>
<td>The word in memory addressed by (rA + rB) is loaded into the low-order 32 bits of rD.</td>
</tr>
<tr>
<td>store double word conditional indexed</td>
<td>stdcx. rS,rA,rB</td>
<td>64-bit only If a reservation exists and the effective address specified by the stdcx. instruction is the same as that specified by the load and reserve instruction that established the reservation, rS is stored into the double word in memory addressed by (rA + rB) and the reservation is cleared.</td>
</tr>
<tr>
<td>store word conditional indexed</td>
<td>stwcx. rS,rA,rB</td>
<td>If a reservation exists and the effective address specified by the stwcx. instruction is the same as that specified by the load and reserve instruction that established the reservation, rS is stored into the word in memory addressed by (rA + rB), and the reservation is cleared.</td>
</tr>
<tr>
<td>synchronize</td>
<td>sync</td>
<td>Ensures that all instructions and memory accesses previously initiated by the processor appear to have completed before any subsequent instructions are initiated.</td>
</tr>
<tr>
<td>enforce in-order execution of I/O</td>
<td>eieio</td>
<td>Orders load and store instructions executed by the processor.</td>
</tr>
<tr>
<td>instruction synchronize</td>
<td>isync</td>
<td>Waits for all previous instructions to complete and then discards any instructions that have been fetched, forcing all subsequent instructions to be fetched from memory and executed in the context established by the previous instructions.</td>
</tr>
</tbody>
</table>
Cache management is a unique system maintenance feature in the sense that it can be performed by both user- and supervisor-level software. The implementation (and existence) of on-chip caches is processor-implementation dependent; however, all PowerPC processors discussed in this book have on-chip caches. The cache management instructions listed in Table 6-26 provide user- and supervisor-level software a means to touch, zero, and invalidate regions of the on-chip cache(s).

Note: When an exception condition occurs on PowerPC processors, the on-chip cache facility will be enabled upon entry to the exception handler. This situation and its ramifications are discussed in Chapter 10, “Exceptions and Interrupts.”

**Table 6-26**

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>data cache block touch</td>
<td>dcbt rA,rB</td>
<td>User-level Fetches the block containing the byte addressed by ((rA + rB)) into the data cache. This act may increase performance if the data in the block is frequently accessed.</td>
</tr>
<tr>
<td>data cache block touch for store</td>
<td>dcbtst rA,rB</td>
<td>User-level Fetches the block containing the byte addressed by ((rA + rB)) into the data cache. This act can increase performance since it is likely that the program will store into the addressed byte.</td>
</tr>
<tr>
<td>data cache block set to zero</td>
<td>dcblz rA,rB</td>
<td>User-level The cache block containing the byte addressed by ((rA + rB)) is cleared.</td>
</tr>
<tr>
<td>data cache block store</td>
<td>dcbst rA,rB</td>
<td>User-level Hints that performance will be improved if the block containing the byte addressed by ((rA + rB)) is fetched into the data cache.</td>
</tr>
<tr>
<td>data cache block flush</td>
<td>dcbf rA,rB</td>
<td>User-level Invalidates the block in the data cache addressed by ((rA + rB)), copying the block to memory first, if dirty.</td>
</tr>
<tr>
<td>instruction cache block invalidate</td>
<td>icbi rA,rB</td>
<td>User-level The block containing the byte addressed by ((rA + rB)) is invalidated.</td>
</tr>
<tr>
<td>data cache block invalidate</td>
<td>dcbi rA,rB</td>
<td>Supervisor-level The block containing the byte addressed by ((rA + rB)) is invalidated.</td>
</tr>
</tbody>
</table>
The external I/O instructions shown in Table 6-27 provide a means for user-level software to communicate with special devices. In particular, when devices use addresses as pointers (for a frame buffer or large floating-point lookup table), this input/output mechanism may prove more efficient than traditional memory-mapped I/O.

### Table 6-27
External I/O Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>external control in</td>
<td>eclwx rD,rA,rB</td>
<td>Allows a system designer to implement memory mapping and I/O for special devices.</td>
</tr>
<tr>
<td>word indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>external control out</td>
<td>ecowx rS,rA,rB</td>
<td>Allows a system designer to implement memory mapping and I/O for special devices.</td>
</tr>
<tr>
<td>word indexed</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The system linkage instructions are shown in Table 6-28. The `sc` instruction allows user-level software to invoke operating system services, if they exist. When an `sc` instruction is executed, a system call exception is generated. After exception handling is complete, the `rfi` instruction is used to return to the user-level software that issued the system call.

### Table 6-28
System Linkage Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>system call</td>
<td><code>sc</code></td>
<td>Calls the operating system to perform a service. This instruction is context synchronizing.</td>
</tr>
<tr>
<td>return from interrupt</td>
<td><code>rfi</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>32-bit</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MSR[16-31] = SRR1[16-31]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The next instruction is fetched from address SRR0[0-29]:00</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>64-bit</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MSR[0-32,37-41,48-63] = SRR1[0-32,37-41,48-63]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The next instruction is fetched from the word-aligned address in SRR0.</td>
</tr>
</tbody>
</table>
Translation lookaside buffer (TLB) management is the domain of supervisor-level system software. The instructions listed in Table 6-29 are used to invalidate (remove) entries from the TLB. Note that the PowerPC architecture defines TLBs (and therefore TLB instructions) as optional.

On 64-bit PowerPC implementations such as the 620, segment registers are replaced by segment tables, which are cached in segment lookaside buffers (SLBs) in an analogous manner to TLBs.

Table 6-29
Translation Lookaside Buffer (TLB) Management Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic and Operands</th>
<th>Description of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLB invalidate entry</td>
<td>slbie rB</td>
<td>64-bit only</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If the SLB contains any entry corresponding to the effective address specified in rB, that entry is removed from the SLB.</td>
</tr>
<tr>
<td>SLB invalidate all</td>
<td>slbia</td>
<td>64-bit only</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All SLB entries are invalidated.</td>
</tr>
<tr>
<td>TLB invalidate entry</td>
<td>tlbie rB</td>
<td>64-bit only</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If the TLB contains an entry corresponding to the effective address specified in rB, that entry is removed from the TLB.</td>
</tr>
<tr>
<td>TLB invalidate all</td>
<td>tlbia</td>
<td>All TLB entries are made invalid.</td>
</tr>
<tr>
<td>TLB synchronize</td>
<td>tlbsync</td>
<td>Ensures that all tlbie instructions previously executed by the processor executing the tlbsync instruction have completed on all processors.</td>
</tr>
</tbody>
</table>

Simplified Mnemonics

Certain PowerPC instructions appear often in simple, commonly needed operations. For example, the addi (add immediate) and addis (add immediate shifted) instructions are often used to load a general-purpose register with a value. For example, the following code fragment from Chapter 1, "The PowerPC Transition," loads GPR r3 with the value 0x1234abcd:
addis  r3, r0, 0x1234 ; load high 16 bits with 0x1234
ori    r3, r3, 0xabcd ; load low 16 bits with 0xabcd

Used as shown, these two add instructions emulate a load instruction. In fact, these two instructions are used so frequently to perform loads that two new mnemonics were defined to simplify this operation and make it more obvious. If we rewrite the above fragment using their simplified aliases, it looks like this:

lis    r3, 0x1234 ; load high 16 bits with 0x1234
ori    r3, 0xabcd ; load low 16 bits with 0xabcd

Here, the *lis* (load immediate shifted) instruction, an example of *simplified mnemonics*, performs the same function as the original code. These simplified mnemonics are assembler mnemonics that represent a more complex form of a common operation.

There is a strong resemblance between simplified mnemonics and software-defined macros. In truth, the main difference is that simplified mnemonics are defined within the compiler or assembler and not within the source code file.

The opcodes generated by an assembler for both fragments are identical — the simplified mnemonics exist only for the convenience of the programmer. Of course, the time and effort that simplified mnemonics save depends on the complexity of the instruction. But the ability to use some instructions (such as *rlwinm*) is greatly enhanced due to their simplified form.

Simplified mnemonics exist for many types of operations. In the following section, we'll list the various simplified mnemonics and their applications. Simplified mnemonics are also defined along with the instruction they simplify in Appendix A, “PowerPC Instruction Set Reference.” Tables 6-30 through 6-32 list the most commonly used simplified mnemonics. As demonstrated in Chapter 11, “PowerPC Assembly Language Examples,” using the simplified instruction form greatly enhances a programmer’s ability to work with assembly language code.

Because simplified mnemonics are aliases for PowerPC processor instructions, their implementation is dependent on the assembler, much like a macro. However, the PowerPC architecture specification defines them and recommends their implementation, so support will be universal.
### Table 6-30
The Most Frequently Used Simplified Mnemonics

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Simplified Mnemonic</th>
<th>Equivalent Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>no-op or nop</td>
<td>nop</td>
<td>ori r0,r0,r0</td>
</tr>
<tr>
<td>load immediate</td>
<td>li rD,value</td>
<td>addi rA,r0,value</td>
</tr>
<tr>
<td>load immediate shifted</td>
<td>lis rD,value</td>
<td>addis rA,r0,value</td>
</tr>
<tr>
<td>load address</td>
<td>la rD,d(rA)</td>
<td>addi rD, rA,d</td>
</tr>
<tr>
<td>load address</td>
<td>la rD,variable</td>
<td>addi rD, rA,variable</td>
</tr>
<tr>
<td>move register</td>
<td>mr rA,rS</td>
<td>or rA,rS,rS</td>
</tr>
<tr>
<td>not [complement register]</td>
<td>not rA,rS</td>
<td>nor rA,rS,rS</td>
</tr>
<tr>
<td>move to condition register (CR)</td>
<td>mtc rS</td>
<td>mtcf 0xff,rS</td>
</tr>
</tbody>
</table>

### Table 6-31
Simplified Subtract Mnemonics and Equivalents

<table>
<thead>
<tr>
<th>Operation</th>
<th>Simplified Mnemonic</th>
<th>Equivalent Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtract immediate</td>
<td>subi rD,rA,value</td>
<td>addi rD,rA,-value</td>
</tr>
<tr>
<td>subtract immediate shifted</td>
<td>subs rD,rA,value</td>
<td>addis rD,rA,-value</td>
</tr>
<tr>
<td>subtract immediate with carry</td>
<td>subic rD,rA,value</td>
<td>addic rD,rA,-value</td>
</tr>
<tr>
<td>subtract immediate with carry</td>
<td>subic rD,rA,value</td>
<td>addic rD,rA,-value</td>
</tr>
<tr>
<td>subtract</td>
<td>sub rD,rA,rB</td>
<td>subf rD,rB,rA</td>
</tr>
<tr>
<td>subtract with carry</td>
<td>subc rD,rA,rB</td>
<td>subfc rD,rB,rA</td>
</tr>
</tbody>
</table>
Table 6-32
Simplified Compare Mnemonics and Equivalents

<table>
<thead>
<tr>
<th>Operation</th>
<th>Simplified Mnemonic</th>
<th>Equivalent Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>compare double word</td>
<td>cmpdi crfD, rA, SIMM</td>
<td>cmpi crfD, 1, rA, SIMM</td>
</tr>
<tr>
<td>immediate</td>
<td>crfD, rA, rB</td>
<td></td>
</tr>
<tr>
<td>compare double word</td>
<td>cmpd</td>
<td>cmp crfD, 1, rA, rB</td>
</tr>
<tr>
<td>compare logical double word</td>
<td>cmpldi</td>
<td>cmpli crfS, 1, rA, UIMM</td>
</tr>
<tr>
<td>word immediate</td>
<td>crfD, rA, UIMM</td>
<td></td>
</tr>
<tr>
<td>compare logical double word</td>
<td>cmpld</td>
<td>cmpl crfD, 1, rA, rB</td>
</tr>
<tr>
<td>double word</td>
<td>crfD, rA, rB</td>
<td></td>
</tr>
<tr>
<td>compare word immediate</td>
<td>cmplw</td>
<td>cmpl crfD, 0, rA, UIMM</td>
</tr>
<tr>
<td>compare word</td>
<td>crfD, rA, rB</td>
<td></td>
</tr>
<tr>
<td>compare logical word</td>
<td>cmplwi</td>
<td>cmpli crfD, 0, rA, UIMM</td>
</tr>
<tr>
<td>immediate</td>
<td>crfD, rA, UIMM</td>
<td></td>
</tr>
<tr>
<td>compare logical word</td>
<td>cmplw</td>
<td>cmpl crfD, 0, rA, rB</td>
</tr>
<tr>
<td>immediate</td>
<td>crfD, rA, rB</td>
<td></td>
</tr>
</tbody>
</table>

Rotate and Shift Instructions

The rotate and shift instructions, shown in Table 6-33, are among the most complex in the PowerPC instruction set. Because the normal rotate and shift instructions must handle all possible rotation and shifting operations, they can be both complex and difficult to use. The simplified mnemonics make performing common bit-wise operations on the contents of the GPRs much easier. There are several operations to which the simplified mnemonics are well suited:

- **Extract**
  The selection of an $n$-bit bit field starting at position $b$ in the source register, justification of the bit field in the target register, and clearing of all other bits in the target register.

- **Insert**
  The selection of a justified $n$-bit bit field starting at position $b$ in the source register and insertion of that bit field into the target register at position $b$. Other bits in the target register are unaltered.
- **Rotate**
  The rotation of the contents of a GPR by \( n \) bits.

- **Shift**
  The shifting of the contents of a GPR by \( n \) bits, clearing the vacated bits.

- **Clear**
  The clearing of the leftmost or rightmost \( n \) bits.

- **Clear left and shift left**
  The clearing of the leftmost \( b \) bits and shifting the result left by \( n \) bits. Such an operation can be used to scale an array index by the width of an element.

### Table 6-33
Simplified Rotate and Shift Mnemonics and Equivalents

<table>
<thead>
<tr>
<th>Category</th>
<th>Operation</th>
<th>Simplified Mnemonic</th>
<th>Equivalent Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>rotate/shift</td>
<td>extract and left justify immediate</td>
<td>extrdi rA,rS,n,b</td>
<td>rldicr rA,rS,b,n-1</td>
</tr>
<tr>
<td>double</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rotate/shift</td>
<td>extract and right justify immediate</td>
<td>extrdi rA,rS,n,b</td>
<td>rldicl rA,rS,b+n,64-n</td>
</tr>
<tr>
<td>shift</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rotate/shift</td>
<td>insert from right immediate</td>
<td>insrdi rA,rS,n,b</td>
<td>rldimi rA,rS,64-[b+n],b</td>
</tr>
<tr>
<td>shift</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rotate/shift</td>
<td>rotate left immediate</td>
<td>rotdi rA,rS,n</td>
<td>rldicr rA,rS,n,0</td>
</tr>
<tr>
<td>shift dword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rotate/shift</td>
<td>rotate right immediate</td>
<td>rotdri rA,rS,n</td>
<td>rldicr rA,rS,64-n,0</td>
</tr>
<tr>
<td>shift dword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rotate/shift</td>
<td>rotate left immediate</td>
<td>rotdi rA,rS,rB</td>
<td>rldcl rA,rS,rB,0</td>
</tr>
<tr>
<td>shift dword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rotate/shift</td>
<td>shift left immediate</td>
<td>sldi rA,rS,n</td>
<td>rldicr rA,rS,n,63-n</td>
</tr>
<tr>
<td>shift dword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rotate/shift</td>
<td>shift right immediate</td>
<td>srdi rA,rS,n</td>
<td>rldicl rA,rS,64-n,n</td>
</tr>
<tr>
<td>shift dword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>Operation</td>
<td>Simplified Mnemonic</td>
<td>Equivalent Instruction</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------</td>
<td>---------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>rotate/shift</td>
<td>clear left</td>
<td>clrldi rA,rS,n</td>
<td>rldcl rA,rS,0,n</td>
</tr>
<tr>
<td>dword</td>
<td>immediate</td>
<td>[n &lt; 64]</td>
<td></td>
</tr>
<tr>
<td>rotate/shift</td>
<td>clear right</td>
<td>clrrdi rA,rS,n</td>
<td>rldicr rA,rS,0,63-n</td>
</tr>
<tr>
<td>dword</td>
<td>immediate</td>
<td>[n &lt; 64]</td>
<td></td>
</tr>
<tr>
<td>rotate/shift</td>
<td>clear left and shift left immediate</td>
<td>clrslldi rA,rS,b,n</td>
<td>rldic rA,rS,n,b-n</td>
</tr>
<tr>
<td>dword</td>
<td>[n ≤ b ≤ 63]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rotate/shift</td>
<td>extract and left justify immediate</td>
<td>extlwi rA,rS,n,b</td>
<td>rlwinm rA,rS,b,0,n-1</td>
</tr>
<tr>
<td>word</td>
<td>[n &gt; 0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rotate/shift</td>
<td>extract and right justify immediate</td>
<td>extrwi rA,rS,n,b</td>
<td>rlwinm rA,rS,b+n,32-n,31</td>
</tr>
<tr>
<td>word</td>
<td>[n &gt; 0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rotate/shift</td>
<td>insert from left justify immediate</td>
<td>inslwi rA,rS,n,b</td>
<td>rlwinm rA,rS,32-b,32-n,31</td>
</tr>
<tr>
<td>word</td>
<td>[n &gt; 0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rotate/shift</td>
<td>rotate left</td>
<td>rotlwi rA,rS,n</td>
<td>rlwinm rA,rS,n,0,31</td>
</tr>
<tr>
<td>word</td>
<td>immediate</td>
<td>[n &gt; 0]</td>
<td></td>
</tr>
<tr>
<td>rotate/shift</td>
<td>rotate right</td>
<td>rotlwi rA,rS,n</td>
<td>rlwinm rA,rS,32-n,31,31</td>
</tr>
<tr>
<td>word</td>
<td>immediate</td>
<td>[n &gt; 0]</td>
<td></td>
</tr>
<tr>
<td>rotate/shift</td>
<td>rotate left</td>
<td>rotlw rA,rS,tB</td>
<td>rlwinm rA,rS,tB,0,31</td>
</tr>
<tr>
<td>word</td>
<td></td>
<td>[n &gt; 0]</td>
<td></td>
</tr>
<tr>
<td>rotate/shift</td>
<td>shift left immediate</td>
<td>slwi rA,rS,n</td>
<td>rlwinm rA,rS,n,0,31-n</td>
</tr>
<tr>
<td>word</td>
<td>[n &lt; 32]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rotate/shift</td>
<td>shift right</td>
<td>srwi rA,rS,n</td>
<td>rlwinm rA,rS,32-n,31,31</td>
</tr>
<tr>
<td>word</td>
<td>immediate</td>
<td>[n &lt; 32]</td>
<td></td>
</tr>
<tr>
<td>rotate/shift</td>
<td>clear left</td>
<td>clrlwi rA,rS,n</td>
<td>rlwinm rA,rS,0,n,31</td>
</tr>
<tr>
<td>word</td>
<td>immediate</td>
<td>[n &lt; 32]</td>
<td></td>
</tr>
<tr>
<td>rotate/shift</td>
<td>clear right</td>
<td>clrrwi rA,rS,n</td>
<td>rlwinm rA,rS,0,31-n</td>
</tr>
<tr>
<td>word</td>
<td>immediate</td>
<td>[n &lt; 32]</td>
<td></td>
</tr>
<tr>
<td>rotate/shift</td>
<td>clear left and shift left immediate</td>
<td>clrlslwi rA,rS,b,n</td>
<td>rlwinm rA,rS,b,31-n</td>
</tr>
<tr>
<td>word</td>
<td>[n ≤ b ≤ 31]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Branch Instructions

The goal of the simplified branch instructions is to provide a unique mnemonic for each combination of valid values for the BO and BI operand fields. Although this greatly simplifies the common branching cases, it creates a large number of branch instructions. Table 6-34 summarizes the simplified branch instructions that incorporate the BO field. These instructions are most useful when terminating loops or testing explicit conditions.

Table 6-34
Simplified Branch Mnemonics and Equivalents

<table>
<thead>
<tr>
<th>Root Instruction</th>
<th>(bc) Relative</th>
<th>(bca) Absolute</th>
<th>(bclr) To (LR)</th>
<th>(bcctr) To (CTR)</th>
<th>(bc) Relative</th>
<th>(bcla) Absolute</th>
<th>(bclrl) To (LR)</th>
<th>(bcctrl) To (CTR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Operation</td>
<td>Form that Updates the Link Register</td>
<td>Form that Doesn’t Update the Link Register</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>branch unconditionally</td>
<td>(n/a)</td>
<td>(n/a)</td>
<td>(bclr)</td>
<td>(bc)</td>
<td>(n/a)</td>
<td>(n/a)</td>
<td>(bclrl)</td>
<td>(bcctrl)</td>
</tr>
<tr>
<td>branch if condition TRUE</td>
<td>(bt)</td>
<td>(bta)</td>
<td>(bflr)</td>
<td>(btctr)</td>
<td>(bfl)</td>
<td>(bfla)</td>
<td>(bflrl)</td>
<td>(bflctrl)</td>
</tr>
<tr>
<td>branch if condition FALSE</td>
<td>(bf)</td>
<td>(bfa)</td>
<td>(bflr)</td>
<td>(bfctr)</td>
<td>(bfl)</td>
<td>(bfla)</td>
<td>(bflrl)</td>
<td>(bflctrl)</td>
</tr>
<tr>
<td>decrement CTR, branch if CTR non-zero</td>
<td>(bdnz)</td>
<td>(bdnza)</td>
<td>(bdnzlr)</td>
<td>(n/a)</td>
<td>(bdnz)</td>
<td>(bdnzla)</td>
<td>(bdnz)</td>
<td>(bdnz)</td>
</tr>
<tr>
<td>decrement CTR, branch if CTR non-zero AND condition TRUE</td>
<td>(bdnzt)</td>
<td>(bdnzt)</td>
<td>(bdnzt)</td>
<td>(n/a)</td>
<td>(bdnzt)</td>
<td>(bdnzt)</td>
<td>(bdnzt)</td>
<td>(bdnzt)</td>
</tr>
<tr>
<td>decrement CTR, branch if CTR non-zero AND condition FALSE</td>
<td>(bdnzf)</td>
<td>(bdnzf)</td>
<td>(bdnzflr)</td>
<td>(n/a)</td>
<td>(bdnzf)</td>
<td>(bdnzfl)</td>
<td>(bdnzf)</td>
<td>(bdnzf)</td>
</tr>
<tr>
<td>decrement CTR, branch if CTR zero</td>
<td>(bdz)</td>
<td>(bdza)</td>
<td>(bdzlr)</td>
<td>(n/a)</td>
<td>(bdz)</td>
<td>(bdzla)</td>
<td>(bdz)</td>
<td>(bdz)</td>
</tr>
<tr>
<td>decrement CTR, branch if CTR zero AND condition TRUE</td>
<td>(bdzt)</td>
<td>(bdzt)</td>
<td>(bdzflr)</td>
<td>(n/a)</td>
<td>(bdzt)</td>
<td>(bdzfl)</td>
<td>(bdzt)</td>
<td>(bdzt)</td>
</tr>
<tr>
<td>decrement CTR, branch if CTR zero AND condition FALSE</td>
<td>(bdzf)</td>
<td>(bdzf)</td>
<td>(bdzflr)</td>
<td>(n/a)</td>
<td>(bdzf)</td>
<td>(bdzfl)</td>
<td>(bdzf)</td>
<td>(bdzf)</td>
</tr>
</tbody>
</table>

\(n/a\) - No simplified form for this branch operation can be derived from the indicated root mnemonic.
Some combinations in the above table do not have a simplified form (represented by a "n/a" entry). In such cases, the branch operation shown in the leftmost column and the root instruction at the top of the column don't make sense together. For example, all branch forms that rely on the CTR for the target address (in the bcctr and bcctr columns) do not have a form that allows decrementing the CTR.

Using the simplified branch instructions in Table 6-34, the following example shows the difference between using the non-simplified version and the simplified version. The branch example below would be appropriate at the bottom of a loop; it assumes that a compare operation has occurred that sets the bits of CR0.

```
; Simplified vs. Non-simplified branching using instructions in Table 6-34.

bc 0x8,0x2,NotDoneYet       ; First, decrement CTR.
; Then branch to NotDoneYet if CTR does not equal zero and CR0[EQ] is set.

; The equivalent simplified mnemonic from Table 6-34:

bdnzt NotDoneYet            ; Decrement CTR, and then branch to NotDoneYet if CTR does not equal zero and CR0[EQ] is set.
```

Table 6-35 summarizes the branch instructions that incorporate the BI field. These are useful when doing typical compare operations and are similar to the compare instructions found on x86 processors. Listing 6-3 shows an example of using the simplified branch mnemonics in Table 6-35.
### Table 6-35

**Simplified Branch Instructions Using Common Comparison Conditions**

<table>
<thead>
<tr>
<th>Standard Branch Mnemonic Root Instruction</th>
<th>Does’t Update Link Register</th>
<th>Updates Link Register</th>
<th>Branch Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root Instruction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bc Relative</td>
<td>bca Absolute To LR</td>
<td>bcctr Relative</td>
<td>less than (lt)</td>
</tr>
<tr>
<td>bclr To CTR</td>
<td>bltl btlr</td>
<td>blctbr</td>
<td></td>
</tr>
<tr>
<td>bcl Relative</td>
<td>bcla Absolute To LR</td>
<td>bcltl bcltr</td>
<td></td>
</tr>
<tr>
<td>bcctrl To CTR</td>
<td>bcltl bcltr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch Operation</td>
<td>Updates Link Register</td>
<td>Does’t Update Link</td>
<td></td>
</tr>
<tr>
<td>less than (lt)</td>
<td>blt bltra</td>
<td>blctrl</td>
<td></td>
</tr>
<tr>
<td>less than or equal (le)</td>
<td>ble blea blear</td>
<td>blete r</td>
<td></td>
</tr>
<tr>
<td>equal (eq)</td>
<td>beq beqar</td>
<td>beqcr</td>
<td></td>
</tr>
<tr>
<td>greater than or equal (ge)</td>
<td>bge bgea bgear</td>
<td>bgecr</td>
<td></td>
</tr>
<tr>
<td>greater than (gt)</td>
<td>bgg bgga bglar</td>
<td>bgltcr</td>
<td></td>
</tr>
<tr>
<td>not less than (nl)</td>
<td>bnl bnlra bnlrar</td>
<td>bnlctr</td>
<td></td>
</tr>
<tr>
<td>not equal (ne)</td>
<td>bne bnea bnear</td>
<td>bnecr</td>
<td></td>
</tr>
<tr>
<td>not greater than (ng)</td>
<td>bng bnga bnglar</td>
<td>bngcr</td>
<td></td>
</tr>
<tr>
<td>summary overflow (so)</td>
<td>bso bsoa bsoar</td>
<td>bsocr</td>
<td></td>
</tr>
<tr>
<td>not summary overflow (ns)</td>
<td>bns bnsa bnsar</td>
<td>bnsctr</td>
<td></td>
</tr>
<tr>
<td>un-ordered (un)</td>
<td>bun bunra bunrar</td>
<td>bunctr</td>
<td></td>
</tr>
<tr>
<td>not un-ordered (nu)</td>
<td>bnu bnuar bnuar</td>
<td>bunctr</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Listing 6-3

An example of how the same branch would be written using standard and simplified mnemonics.

```plaintext
: Listing 6-3
:
; Simplified vs. Non-simplified branching using
; instructions in Table 6-35.
;
bc 0x4,0x2,NotEqualPlace : Branch if the condition specified by
; the BI field (0x2 = CR0[EQ]) is FALSE.
; This means: "branch if not equal"
```
The equivalent simplified mnemonic from Table 6-35:

```
 bne NotEqualPlace
```

; Branch if not equal - both the BO and 
; BI fields are included in mnemonic.

**Trap Instructions**

The `tw` (trap word) instruction invokes the system trap handler (described in detail in Chapter 10, “Exceptions and Interrupts”). The operand format for the trap word instruction is as follows:

```
tw TO,rA,rB
```

; trap word based on a and b comparison 
; according to TO bit setting.

; The following example demonstrates the usage of the simplified 
; trap mnemonic:

```
tw eq,r3,r4
```

; The system trap handler would be invoked 
; if r3 and r4 contained equivalent values.

The operands rA and rB are compared according to the bit set in the 5-bit operand TO. Table 6-36 shows the compare operation that results based on TO[0-4]. On 64-bit PowerPC implementations, such as the 620, only the lower-order 32 bits of the 64-bit GPR are used in the comparison.

**Table 6-36**

Common TO Operand Settings for the Trap Instruction

<table>
<thead>
<tr>
<th>Hexadecimal Values of the TO Operand</th>
<th>Name of Compare Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>lgt (logically greater than)</td>
</tr>
<tr>
<td>0x02</td>
<td>llt (logically less than)</td>
</tr>
<tr>
<td>0x04</td>
<td>eq (equal to)</td>
</tr>
<tr>
<td>0x05</td>
<td>lge (logically greater than or equal to)</td>
</tr>
<tr>
<td>0x06</td>
<td>lle (logically less than or equal to) or lge (logically not greater than)</td>
</tr>
<tr>
<td>0x08</td>
<td>gt (greater than)</td>
</tr>
<tr>
<td>0x12</td>
<td>ge (greater than or equal to)</td>
</tr>
<tr>
<td>0x16</td>
<td>lt (less than)</td>
</tr>
<tr>
<td>0x20</td>
<td>le (less than or equal to) or ng (not greater than)</td>
</tr>
<tr>
<td>0x24</td>
<td>ne (not equal to)</td>
</tr>
<tr>
<td>0x31</td>
<td>unconditional (condition always satisfied)</td>
</tr>
</tbody>
</table>
THE SUBLIME ART OF INSTRUCTION TIMING

"When you have eliminated the impossible, whatever remains, however improbable, must be the truth."
—Sherlock Holmes in Sign of Four
by Sir Arthur Conan Doyle

The x86 architecture provides a favorable environment in which to learn about optimization. Instructions always execute serially, for example. And finding the number of cycles required by each instruction to execute requires only that you read the manual. After learning the techniques and tricks required to program each processor, you're armed with the knowledge required to write some pretty darn fast code. And after writing that code, estimating its execution time is a simple matter of addition.

The PowerPC processors benefit equally from well-written code. But in terms of instruction timing, it's a whole new ball game. PowerPC processors implement advanced architectural features such as instruction pipelining, out-of-order execution, and branch prediction. As a result, how an instruction is executed depends to a great extent on when it is executed and, by implication, what is going on elsewhere in the processor at that time.

The instruction set architecture of RISC processors (such as the PowerPC family) lends itself to efficient programming. When compared with the instruction set of CISC processors
(such as the x86), there are generally fewer choices concerning how to perform a specific task. If you’ve skimmed through Appendix A, “PowerPC Instruction Set Reference,” you may have a hard time believing that, but let me assure you — you’ll find it to be true. With the x86, there are so many ways to perform the same operation, there is an added level of complexity to writing the fastest possible code sequence.

This chapter concentrates on the mechanisms that influence instruction execution on PowerPC processors. After looking at instruction flow through the various processor implementations, we’ll investigate techniques for creating efficient software. In Chapter 11, “PowerPC Assembly Language Examples,” we’ll use these ideas and expand upon them in a few real-world examples.

**The Pipeline**

Instruction execution and timing on PowerPC microprocessors are centered around the concept of an instruction pipeline, which distributes instructions to the various execution units. Additionally, each execution unit may have its own pipeline which is fed from the dispatch stage of the master pipeline; execution unit pipelines typically have fewer stages.

It is important that we adopt a generalized PowerPC pipeline model during the following discussions. The pipeline model described in this chapter represents a close approximation to the pipelines found on the 603, 604, and 620. Because pipeline characteristics are not defined by the PowerPC architecture, pipeline implementation varies between processors. For this reason, the generalized PowerPC pipeline model used in this chapter represents the best means of conducting a meaningful instruction timing discussion that addresses more than one PowerPC processor.

As described in Chapter 2, “Foundations and Architecture,” the pipelining of instructions allows the apparent simultaneous execution of multiple instructions. Our general model of a PowerPC pipeline has six stages: fetch, decode, dispatch, execute, complete, and write-back. Each stage is responsible for a discrete portion of the process of executing an instruction. Figure 7-1 illustrates the processor pipeline model.

It’s common — and useful — to ask whether the master pipeline exists physically within the processor or is simply a useful but fictitious mechanism for discussing the flow of instructions through the stages of execution. In fact, both descriptions are accurate to some extent. The fact that each
processor implementation contains a unique pipeline model makes generalization difficult. However, the important (and functional) concept to remember is this: The pipeline physically exists to the extent that delays in one stage of the pipeline can impact the operation of other stages.

![Figure 7-1](image_url)

*PowerPC processors use the same master pipeline configuration.*

In Figure 7-1, instructions are shown starting at the left and moving through the pipeline toward the right. As an instruction moves from one pipeline stage into the next, it frees the previous stage to accept subsequent instructions. In this manner, the processing of more than one instruction can be overlapped.

The execute stage is unique in that the details of an instruction’s execution are dependent on the particular execution unit to which it is dispatched. After an instruction exits the execute stage, it enters the complete stage where the results of its execution are reflected in the processor’s registers.

The ultimate goal of pipeline operation is to complete as many instructions as possible in a given time. There is a cause-and-effect relationship involved here: The number of completed instructions is limited to the number of instructions dispatched. So keeping the pipeline moving is a high priority. To keep instructions flowing through the pipeline, the pipelines in the individual execution units must also keep moving. If any pipeline stalls, instructions may back up far enough to inhibit instruction dispatch.
The following sections describe each stage in the master pipeline. Note that the implementation of each stage may vary (even disappear) across PowerPC processor implementations. The functionality of each stage is discussed on a per-processor basis.

**Resource Management**

It's helpful to think of the pipeline as a limited set of processor resources that are consumed by instructions as they enter the pipeline and freed as they exit the pipeline. Using this analogy, efficient programming becomes an exercise in resource management.

Figure 7-2 depicts several sequential views of the pipeline as the instruction sequence shown here passes through the first few stages.

; Example code sequence
; Load a byte and test it
Start:
```
        lbz r12,0(r3)  ; load a byte pointed to by r3 into r12
        addic r3,r3,1  ; increment our pointer: r3
        cmpi 0,r12,0x00 ; is it a NULL character?
```

![Figure 7-2](image-url)

*Figure 7-2*

Four consecutive snapshots of the pipeline show how multiple instructions can be processed simultaneously — one per pipeline stage.

Figure 7-2 represents optimal flow through the pipeline. Each instruction advances to the subsequent stage every clock cycle. In reality, flow through the pipeline can stall, causing a delay in the execution of some
(or all) of the instructions. An explanation and examples of pipeline stalls are found in the next section.

Starting at the first row in Figure 7-2 (clock cycle 1), the `lbz` (load byte and zero) instruction enters the fetch stage of the pipeline. Subsequent views of the pipeline show instructions moving from left to right through the stages of the pipeline.

By clock cycle 3, all three of the instructions from our example sequence are being processed simultaneously. The `cmpi` (compare immediate) instruction has just entered the fetch stage and the `lbz` instruction is ready to be dispatched. Using this model, it is easy to imagine the pipeline continuously executing several instructions simultaneously.

The total number of clock cycles that it takes to execute an instruction and write back the results is known as the instruction’s latency. And the measure of the number of instructions that are processed per clock cycle is known as throughput. A fundamental goal of PowerPC programming is to minimize the latency of each instruction. Using pipelining, branch prediction, and multiple execution units, PowerPC processors are able to achieve a throughput that averages more than one instruction per cycle.

**Pipeline Stalls**

The previous example represents an ideal pipeline; instructions always advanced a stage each clock cycle. This is helpful for a basic understanding of instruction flow through a pipeline; however, it is not sufficient to understand the details of pipeline implementations on each processor.

Each execution unit may have an instruction buffer known as a reservation station. This decreases the chance that the processor pipeline will not be able to dispatch an instruction to a busy execution unit. This situation, called a pipeline stall, means that the normal progress of instructions to the next stage in the pipeline grinds to a halt. Pipeline stalls occur for well-known reasons, and writing efficient code for the PowerPC means minimizing the number of stalls.

**Timing Basics**

Instruction execution times are generally fixed on x86 processors that predate the Pentium. The number of cycles required to execute a move
instruction differs between a 386 and 486. And as the 486 has matured, the
timing for some instructions has changed between revisions, but these
changes are generally reflected in the documentation. Overall, the changes
are minimal enough not to strain our use of the term constant — something
you’ll soon come to appreciate.

<table>
<thead>
<tr>
<th>Reservation Stations</th>
</tr>
</thead>
<tbody>
<tr>
<td>What if an execution unit is busy or required data is not available when the dispatch stage tries to dispatch an instruction? These distinct and real possibilities occur regularly during normal execution on PowerPC processors. To minimize the effects of busy execution units and data dependencies holding up dispatch (and therefore delaying all instructions), some execution units have one or more reservation stations.</td>
</tr>
</tbody>
</table>

A reservation station is an instruction buffer that can hold a dispatched instruction until the conditions for execution of the current instruction are met. Reservation stations exist on the PowerPC 603, 604, and 620 processors only. Using the reservation station, the pipeline can dispatch the instruction and does not stall. Of course, if the reservation station fills up, we’re back to where we started.

Determining the execution time of instructions on PowerPC processors is a complex matter. PowerPC instructions have a documented time in clock cycles associated with their execution, but that figure represents only the minimum number of cycles that it takes to execute the instructions — it says nothing about the maximum number of cycles. An instruction’s execution time is influenced by variables such as data dependencies, busy execution units, and other obstacles that affect flow through the pipeline.

As mentioned previously, the number of clock cycles that it takes to completely execute an instruction, such that its results are known to subsequent instructions, is known as the instruction’s latency. Like instruction cycle times for the x86 family of processors, the latency for individual PowerPC instructions varies from implementation to implementation.

**Timing Issues**

There are many factors that influence timing on PowerPC processors. A general understanding of each aspect is an important precursor to the rest of this chapter. As we’ll see, there is not one, single, core concept associated with instruction timing; in fact there are several equally important ideas that
must all be given attention. Some of the most significant architectural features that influence instruction timing include:

- **Out-of-order execution**
  When an instruction with a long latency is issued before an instruction with a short latency, out-of-order execution frees the processor from waiting on the slower instruction.

- **Branch folding**
  When the direction of a branch can be determined by the processor before the branch instruction is executed, it may be replaced with instructions from the target address. This mechanism allows the processor to simulate execution from a single linear code stream.

- **Branch prediction**
  The direction of branch instructions may also be predicted. Depending on the processor implementation, either static or dynamic branch prediction (or both) is used by the processor to minimize the performance degrading effects of handling branch instructions.

Each of these features are designed to increase the processor’s efficiency during instruction execution. And each will be discussed in the sections that follow.

**Out-of-Order Execution**

Simply put, out-of-order execution means that instructions execute in other than program order. This situation occurs as a result of instructions taking different lengths of time to complete. PowerPC processors can support out-of-order execution because their architecture provides multiple execution units. For example, while the integer unit is executing an integer instruction, the floating-point unit can work on a floating-point instruction.

But there’s a catch to out-of-order execution: All software (and the PowerPC architecture’s exception model) expects and depends on precise (in-order) results. Let’s illustrate the point with an overly simple example. In the code shown following, you certainly would not want line 3 executed before line 2:

1:  \[ x = \text{MiscFunction}(); \]
2:  \[ y = x+10; \]
3:  \[ x = x*y; \]
There are a number of mechanisms on PowerPC processors that exist solely to ensure precise results from out-of-order execution. Rename buffers and completion buffers are two resources that are designed to enforce inorder results. The PowerPC 601 does not implement completion buffers or rename buffers. As with any resource, situations can occur when a rename buffer or completion buffer is unavailable when an instruction is ready to be dispatched. In these cases, dispatch stalls and cannot continue until there are sufficient resources to execute the instruction. The ability to execute instructions out of order can thus restrict instruction flow through the pipeline.

**Branching**

In a linear instruction stream — one that contains no branches — the only dependencies that can stall execution are busy execution units and data dependencies. And with some proper instruction ordering (scheduling), both these problems can be nearly eliminated. Adding branch instructions to the mix, however, complicates the situation significantly.

Branch instructions mean that there are suddenly occasions (a taken branch) when subsequent entries in the master pipeline are no longer valid. In such situations, these invalid entries must be flushed (emptied) and new instructions must be fetched from the new instruction stream (corresponding to the taken branch). Fortunately, the processor may be able to (and often can) remove the branch from the instruction stream using branch prediction (discussed in the following section). However, if branches cannot be predicted, even the most clever instruction scheduling has little influence over the detrimental effects of branch instructions on pipelining.

**Branch Prediction and Speculative Execution**

Branch instructions are, by their nature, the most common cause of pipeline inefficiencies. Branch prediction attempts to eliminate the penalty associated with flushing the pipeline after determining the direction (taken or not taken) of a branch. If the processor can successfully predict the direction of a branch, then instructions can be fetched ahead of time from the target address. This reduces the detrimental effects of a pipeline flush and subsequent refilling.

When the direction of a branch is predicted and execution of instructions continues along the predicted path, the processor is said to be
speculatively executing. Speculative execution is an important performance-enhancing technique that is used to increasing advantage in the more advanced PowerPC processors.

**Branch Folding**

*Branch folding* is the process of replacing a branch instruction with its target instruction stream before that branch reaches the dispatch stage. The target instruction stream corresponds to either branch direction: taken or not taken. That is, if the branch is not taken, the target instruction stream corresponds to the next sequential instruction. The branch folding mechanism is used on the PowerPC 601 and 603 processors.

Branch instructions can be folded when the direction (taken or not taken) can be precisely determined (as opposed to predicted) from available information. Like branch prediction, the process of branch folding involves determination of the direction of the branch. But because the folded outcome is known to be valid, the action following the folding of a branch instruction is resolute. If the branch was not taken, it is simply removed from the instruction stream and the next instruction is free to take the place of the folded branch instruction. If the branch was taken, the pipeline is flushed and fetching begins at the new address.

PowerPC processors use the condition register, link register, and other resources to determine if branches can be folded out of the instruction stream. If no dependencies exist between the current instruction in the instruction queue and the branch instruction that will be folded, the processor is able to remove the branch entirely. In general, the branch instructions that are folded are replaced with their target instruction streams. In this manner, the processor (specifically, the dispatch stage) sees only a linear stream of instructions.

**Static Branch Prediction**

Unlike dynamic branch prediction, static branch prediction is defined by the PowerPC architecture. The PowerPC 601 and 603 both implement a form of *static branch prediction*. Static branch prediction is a software-based mechanism that provides hints to the processor concerning the likely direction of a branch. In particular, compilers and assemblers can encode information into conditional branch instructions that tell the processor the most likely outcome of the branch.
The 603’s branch processing unit (BPU) will occasionally encounter a branch instruction that inhibits further execution. The BPU must wait until the data dependency is resolved before executing the branch. However, instead of stalling the entire execution process, the branch unit may predict the direction of the branch and begin an instruction fetch to the predicted path. Any instructions from the predicted path that are executed before the branch is resolved are said to be speculatively executed.

**Dynamic Branch Prediction**

All PowerPC processors perform some type of branch prediction. Dynamic branch prediction is performed only by the PowerPC 604 and 620. The dynamic branch prediction mechanism in the 604 and 620 uses two internal structures to assist in the prediction process: the branch target address cache (BTAC) and the branch history table (BHT). In general, dynamic branch prediction uses the previous behavior of the specified branch to guess the direction of the outcome; the BTAC and BHT store the previous behavior.

On the 604, the BTAC is a 64-entry cache that holds the target addresses taken for recently executed branch instructions. In other words, it records a short history of successful branches. Each entry in the BTAC has two keys: the address of the branch and the address of its target last time the branch was taken. The processor uses the BTAC to evaluate branch instructions during the decode stage. If the branch’s target address is found in the BTAC (a BTAC hit), the processor assumes that the branch will be taken and folds the branch out, replacing it with the target instruction stream. If the current branch target address is absent from the BTAC or there is insufficient information to resolve the branch, the processor uses the BHT during the dispatch stage.

On the 604, the BHT is a 512-entry cache that contains target addresses for branch instructions that depend entirely on a condition reported by the control register (CR). The least significant 9 bits of the address of the branch instruction are used to index into the BHT. Each entry in the BHT contains 2 bits that are set to hint toward one of four prediction states: strongly taken, taken, not-taken, and strongly not-taken. This processor uses these hints to predict the direction of branch instructions that depend only on the CR. Dynamic branch prediction on the PowerPC 620 uses a similar BTAC and BHT configuration.

The processor updates the prediction hints in the BHT each time a branch is taken. For example, if a BHT entry predicts “not-taken” for the current branch and the branch is actually taken, the BHT entry would be
updated to "taken." Likewise, if that same branch was not taken (as predicted) the same BHT entry would be updated to "strongly not-taken." In this manner, the prediction is refined based on past behavior. Figure 7-3 shows how predicted state information is updated based on the outcome of the current branch instruction.

Dynamic branch prediction takes place within three pipeline stages: fetch, decode, and dispatch. During the instruction fetch stage, the BTAC is checked for the current branch target address. If there is a hit in the BTAC, then instruction fetch continues at the address associated with the predicted path. In the decode and dispatch stages, the first branch instruction is identified and predicted. The method of prediction is based on the type of branch instruction: Unconditional branches automatically redirect execution; branches depending on the CTR are predicted based on the current CTR value; and branches depending on the CR are predicted using the BHT.
Rename Registers and Completion Buffers

Rename registers and completion buffers are two separate mechanisms that facilitate the implementation of the PowerPC precise exception model and speculative execution. The PowerPC precise exception model is covered in Chapter 10, “Exceptions and Interrupts.”

PowerPC processors have the ability to predict branches and to execute multiple instructions per clock cycle, increasing instruction throughput. But branches are occasionally predicted incorrectly. To minimize the effect of incorrectly predicted branches and instructions that should not have been executed, the PowerPC 603, 604, and 620 processors use a feature known as completion buffers. The results of speculatively executed instructions are stored in the completion buffers; when the instructions are determined to be valid (correctly predicted branch), the processor uses the completion buffer to update the architectural register set. In this manner, the architectural registers are affected only by instructions from the proper code stream.

Rename registers — which practice register aliasing — are used to minimize contention for general-purpose or floating-point registers due to out-of-order execution. The following code fragment demonstrates how rename registers can minimize register dependencies during normal execution. Note that register aliasing is completely transparent to both the programmer and user.

In Listing 7-1, let’s assume that the first load instruction misses in the cache. Typically, execution would have to stall while the value at 0x00 is fetched from main memory. The second instruction (store to 0x10) will move into a store reservation station while it waits for the data to come in from main memory.

Listing 7-1

Rename registers minimize delays in execution.

```assembly
lwc r5, 0x00 ; <CACHE MISS> load from 0 into r5
stw r5, 0x10 ; load from 0x10 into r5
lwc r5, 0x100 ; <CACHE HIT> load from 0x100 into r5
stw r5, 0x110 ; load from 0x110 into r5
```
If the third instruction (load from 0x100) hits in the cache, we could actually execute it. But wait a minute — instruction #3 depends on r5, for which we’re already waiting. This is precisely where register aliasing comes into play. Instruction #3 will execute and update a register alias, which will be standing in for the real GPR r5, allowing the instruction to execute and avoiding further stalls. When the previous instructions finish executing, the processor uses the register alias to complete each instruction in program order. The same logic within the processor that tracks the register usage allows speculative execution.

**Instruction Fetch**

On all the PowerPC implementations discussed in this book, the instruction fetch stage is responsible for retrieving instructions from either the cache or from main memory and placing them into the instruction queue (buffer). All instructions must pass through the fetch stage.

The time required for instruction fetch directly depends on whether or not the instruction was found in the processor’s *level one* (on-chip) cache. If the required instructions are not in the instruction cache, a memory access is initiated by this stage. The cache operation for the PowerPC processors is covered in Chapter 9, “The PowerPC Cache.”

**Instruction Decode**

The decode stage processes instructions received from the fetch stage. The decode stage is responsible for feeding the dispatch stage with instruction information that can be used to dispatch instructions to the appropriate execution unit.

On the 601 and 603 processors, the functions of the decode stage are provided by the fetch and dispatch stages. Branch instructions are identified in the fetch stage and dispatched directly to the branch processing unit. All other instructions are both decoded and dispatched by the dispatch stage. On the 604 and 620, the decode stage is responsible for high-priority decoding only. Other non-time-critical instruction decoding is handled by the dispatch stage.

**Instruction Dispatch**

The dispatch stage issues instructions to the various execution units. Additionally, the dispatch stage tracks resource availability for each
execution unit to determine if a unit is capable of handling the instruction and which instructions can be dispatched during the current cycle. The 601’s dispatch stage can process up to three instructions per clock cycle. This rate requires that one instruction be dispatched to each of the three execution units: integer, floating-point, and branch processing.

The 601 does not implement reservation stations on any of the processor’s execution units. As a result, if the target execution unit for instruction dispatch is busy, the pipeline stalls until the target execution unit is free. However, other instructions may be issued to free units from any of the lower four instruction queue positions not occupied by the stalled instruction. A branch instruction will never occupy the 601’s dispatch stage; they are removed by the decode stage. Note that the PowerPC 601 implements a one-level queue for floating-point instructions.

The 603 issues up to two instructions per clock cycle from the lower two instruction queue positions. As shown in Figure 7-4, the 603 and subsequent implementations employ reservation stations to reduce the chance of stalls. Each execution unit on the 603 has one reservation station.

The 603, 604, and 620 PowerPC processors implement one or more reservation stations at each execution unit.

When the 603 dispatches an instruction, it allocates and associates one of its five completion buffers to that instruction until the instruction completes. If the 603 is unable to allocate a completion buffer for the instruction occupying the dispatch stage, dispatch will stall until a buffer is
available. During the course of execution, all information concerning the execution of the instruction is kept in the completion buffer.

To track any register-updating results during execution, the 603 also allocates a rename buffer for each instruction that is dispatched. In this manner, instructions may execute out of program order, but can be completed in order using the results stored in the completion buffer and rename buffer.

The 604 and 620 can dispatch up to four instructions per clock cycle. Additionally, the dispatch stage on the 604 and 620 is responsible for reading the instruction's operands from the appropriate register file (GPR or FPR). When the dispatch stage finishes, it has latched all dispatched instructions and operands into the appropriate execution units. The 604 and 620 implement completion buffers and rename buffers in the same manner as the 603. However, the 604 and 620 have a 16-entry completion buffer — more than twice the size of the completion buffer on the 603.

**Instruction Execute**

All instructions entering the execute stage of the pipeline have passed through the same common stages. During the execute stage, instructions are sent only to the execution unit responsible for their execution.

In addition, each execute stage may have an independent number of pipeline stages. For example, the floating-point unit of the PowerPC 604 has a three-stage execution pipeline; therefore, the single execute stage for floating-point instructions on the 604 actually represents three stages within the FPU.

**Instruction Complete**

As an instruction exits the execute stage, it enters the complete stage where the results of its execution are posted to the processor's architectural registers. The complete stage ensures that instructions complete in program order by processing the completion buffers and rename buffers in strict program order.

The effects of any tentatively executed instructions (those found to be incorrect due to mispredicted branches, for example) manifest themselves only in the rename buffers. When such instructions are verified as valid, the complete stage uses the rename buffers to update its architectural (real)
register set. The architectural registers are affected only by instructions from the correct code stream.

**Instruction Write-Back**

The write-back stage updates the architectural register set (writing back) with the results of the current instruction. As a final pass at completing the instruction, any information that was not written back during the completion stage is updated in the architectural registers during this stage.

The information that is used to update the architectural registers depends on the instruction; such information may include values contained in general-purpose registers, floating-point registers, the condition register, or link register.

**Real-World Instruction Timing**

A complete discussion of the factors affecting code flow through the PowerPC pipelines is a suitable topic for an entire book. However, this chapter would not be complete without making an effort to demonstrate how real code flows through a real pipeline.

Listing 7-2 shows the main loop of a floating-point matrix multiply (taken from an example in Chapter 11, “PowerPC Assembly Language Examples”). This code is a good example for two reasons: First, it uses the floating-point unit to the extent that resource contention must be considered. Second, the matrix multiply operation is typically used in code that must be fast (such as graphics software). Understanding how instructions flow through the pipeline is a good way to understand what aspects need optimization.

Before we begin our investigation of instruction timing using Listing 7-2, we must make the following assumptions to decrease the scope of the example and simplify our discussion:

- The code is executing on a PowerPC 604 microprocessor. This assumption dictates the implementation of the pipeline. In particular, we’ll be using a six-stage pipeline that requires in-order instruction dispatch to execution units. The pipeline stages are fetch, decode, dispatch, execute, complete, and write-back.
Each load and store operation and instruction fetch is available from the on-chip cache. In other words, no main memory accesses are required.

There is no contention for processor resources such as register buffers.

During each fetch stage, the maximum number of instructions (four) are fetched from the cache. Depending on instruction location within the cache, this may not reflect real-world operation.

**Listing 7-2**

Main Loop of 4 x 4 Matrix Multiply

```
; MatrixMultLoop:
00  lfs  f0,0(r4)       ; Get x value
01  lfs  f5,4(r4)       ; Get y value
02  fmuls f1,f0,f31     ; Get xres = x * m[0][0]
03  fmuls f2,f0,f30     ; Get yres = x * m[0][1]
04  fmuls f3,f0,f29     ; Get zres = x * m[0][2]
05  fmuls f0,f0,f28     ; Get wres = x * m[0][3]
06  fmadds f1,f5,f27,f1 ; Get xres = xres + y*m[1][0]
07  lfs  f4,8(r4)       ; Get z value
08  fmadds f2,f5,f26,f2 ; Get yres = yres + y*m[1][1]
09  fmadds f3,f5,f25,f3 ; Get zres = zres + y*m[1][2]
10  fmadds f0,f5,f24,f0 ; Get wres = wres + y*m[1][3]
11  lfs  f9,12(r.4)     ; Get w value
```

As we’ll find out while examining the instruction flow through the pipeline in Table 7-1, even with simplifying assumptions, instruction timing is a complex topic.

The leftmost column of Table 7-1 contains cycle 0 — our starting point. As instructions flow through the pipeline, clock cycles increase from left to right. Each instruction name appears with its line number in the fetch stage; after the fetch stage, only the line number is used to track each instruction’s progress through the pipeline.
# Table 7-1
Instruction Timing for 10 Cycles of Listing 7-2.

<table>
<thead>
<tr>
<th>Cycle 0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 F</td>
<td>0 DE</td>
<td>0 DS</td>
<td>0 EX</td>
<td>0 C</td>
<td>0 W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ifs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 F</td>
<td>1 DE</td>
<td>1 DS</td>
<td>1 DS</td>
<td>1 EX</td>
<td>1 C</td>
<td>1 W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ifs</td>
<td></td>
<td></td>
<td>delay</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 F</td>
<td>2 DE</td>
<td>2 DS</td>
<td>2 EX1</td>
<td>2 EX2</td>
<td>2 EX3</td>
<td>2 C</td>
<td>2 W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fmul</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 F</td>
<td>3 DE</td>
<td>3 DS</td>
<td>3 DS</td>
<td>3 EX1</td>
<td>3 EX2</td>
<td>3 C</td>
<td>3 W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fmul</td>
<td></td>
<td></td>
<td>delay</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 F</td>
<td>4 DS</td>
<td>4 DS</td>
<td>4 DS</td>
<td>4 EX1</td>
<td>4 EX2</td>
<td>4 EX3</td>
<td>4 C</td>
<td>4 W</td>
<td></td>
</tr>
<tr>
<td>fmul</td>
<td>delay</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 F</td>
<td>5 DS</td>
<td>5 DS</td>
<td>5 DS</td>
<td>5 EX1</td>
<td>5 EX2</td>
<td>5 EX3</td>
<td>5 C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fmul</td>
<td>delay</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 F</td>
<td>6 DS</td>
<td>6 DS</td>
<td>6 DS</td>
<td>6 EX1</td>
<td>6 EX2</td>
<td>6 EX3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fmad</td>
<td>delay</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 F</td>
<td>7 DS</td>
<td>7 DS</td>
<td>7 DS</td>
<td>7 EX</td>
<td>7 C</td>
<td>7 W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ifs</td>
<td>delay</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 F</td>
<td>8 DS</td>
<td>8 DS</td>
<td>8 DS</td>
<td>8 EX1</td>
<td>8 EX2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fmad</td>
<td>delay</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 F</td>
<td>9 DS</td>
<td>9 DS</td>
<td>9 DS</td>
<td>9 EX1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fmad</td>
<td>delay</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 F</td>
<td>10 DS</td>
<td>10 DS</td>
<td>10 DS</td>
<td>10 EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fmad</td>
<td>delay</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 F</td>
<td>11 DS</td>
<td>11 DS</td>
<td>11 DS</td>
<td>11 C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ifs</td>
<td>delay</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

F = Fetch Stage  
DE = Decode Stage  
DS = Dispatch Stage  
EX = Execute; EX1 – EX3 = Three stages of floating-point execution unit  
C = Completion  
W = Write-back  
***** = Instruction has completed execution.  
delay = Instruction is delayed for some reason.
In cycle 0, the first four instructions are fetched from the instruction cache. Each instruction’s name is shown below the instruction number in the fetch stage; for cycle 0, two `lfs` (load floating-point single) instructions and two `fmuls` (floating-point multiply single) instructions are beginning execution. In cycle 1, the first four instructions enter the decode stage and the next four instructions are fetched from the instruction cache. So far, everything is well behaved.

In cycle 2, instructions 8–11 are fetched from the instruction queue; all other instructions advance to the next stage without delay.

In cycle 3, instruction 1 begins execution in the load/store unit. As a result, instruction 2 must be delayed in the dispatch stage until the load/store unit’s execute stage is free on the next cycle. Similarly, instruction 2 begins execution in the first of three floating-point execute stages; therefore, instruction 3 is delayed. All other instructions advance to the next stage.

In cycle 4, instruction 0 has vacated the load/store unit’s execute stage, allowing instruction 1 to execute. Instruction 2 enters the second stage of the FP unit’s execute stage, allowing instruction 3 to execute. Additionally, instructions 4–7 are all delayed in the dispatch stage due to occupied execute stages in the FP and load/store units.

In cycle 5, instruction 0 has completed execution and writes its results back to the architectural register set. Instruction 4 is allowed to enter the execute stage; subsequent instructions remain delayed.

The remaining instructions continue to flow through the pipeline in the same fashion as previously described. To put this example into the proper context, note that our example did not consider processor resource contention, cache misses for instructions and data, or branch prediction. Plus, this example only applies to the PowerPC 604 processor!

**Summary**

Discussions of PowerPC instruction timing seem to dwell more on what cannot be said than actual material that can be used in day-to-day programming. At the very least, any meaningful discussion cannot be generalized to all processors — timing information is strictly processor dependent.

The tools and techniques that will turn instruction timing into a science (rather than art form) are just becoming available. While this may seem a
little late for such fundamental tools, consider how long it took the PC industry to produce truly optimized software. The advanced features that PowerPC processors use to execute code (superscalar operation, branch prediction, and so on) require advanced tools to measure and refine their operation. Chapter 12, “Techniques and Tricks,” discusses the performance monitoring capabilities of the 604 and 620 — which just might turn out to be the Rosetta stone of PowerPC instruction timing.

Fortunately, instruction timing is not the only means of gauging optimized code. In Chapter 11, “PowerPC Assembly Language Examples,” we’ll examine how to implement several common programming constructs, including how to optimize each example.
Chapter 8

MEMORY MANAGEMENT

"Everything which had been disconnected before began at once to assume its true place, and I had a shadowy presentiment of the whole sequence of events."
— Sherlock Holmes in Crooked Man
by Arthur Conan Doyle

The details of memory management are frequently hidden from both users and programmers alike — and for good reason. The first time I picked up an i386 data book and tried to grasp the details of memory management, I was completely distracted by the terminology and the distinction between the various types of memory, including virtual, linear, physical, effective, real, logical, and so on.

In this chapter, we'll be looking at all aspects of the PowerPC's memory management capabilities as they apply to the programmer. But to avoid any distraction from the topic at hand, we'll take a few paragraphs to define the terminology.

MEMORY NOMENCLATURE

Physical memory means the chips that you hold in your hand. When plugged into your computer, they are arranged as a linear series of 8-bit bytes. Each byte is given a physical address
that ranges from 0 to a maximum of 4,294,967,295 \((2^{32}-1)\). Physical memory is grist for the mill of memory management.

*Virtual memory* is a catch-all term. It implies that some form of memory management such as address translation or memory protection is enabled. For the most part, both the PowerPC family and x86 use the terms physical memory and virtual memory consistently. Other terms are architecture-specific.

**x86 Terminology**

When software refers to memory, regardless of operating mode, memory model, or other architectural concerns, the address it uses is called a *logical address*. When the processor is operating in real mode, the logical address is known as a *segmented address*. When memory management is enabled, the logical address is called a *virtual address*. (The offset portion of these addresses is called the *effective address*.)

In either case, the logical address is translated by the segmentation mechanism into a non-segmented *linear address*. In turn, the paging mechanism translates the linear address into a physical address. If paging is disabled, logical addresses are equivalent to linear addresses. If memory management is disabled altogether, then logical, linear, and physical addresses have a direct correspondence.

**PowerPC Terminology**

On PowerPC processors, any memory access generates an *effective address*. When memory management is enabled, the effective address is subsequently translated by one of the PowerPC family's three virtual memory management mechanisms into a physical address. When memory management is disabled, the PowerPC processor is said to be using *real mode* addressing and all address translation is disabled; in this case, the effective address corresponds directly to a physical address. Table 8-1 summarizes the terminology associated with memory management on both PowerPC and x86 processors as used in this book.
### Table 8-1
The Differences Between x86 and PowerPC Memory Management Terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>PowerPC Definition</th>
<th>x86 Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Memory</td>
<td>All the memory that physically resides on the memory bus of the processor.</td>
<td>Same as PowerPC definitions.</td>
</tr>
<tr>
<td>Virtual Memory</td>
<td>The address space created using the memory management facilities of the processor. Program access to virtual memory is possible only when it coincides with physical memory.</td>
<td>Same as PowerPC definitions.</td>
</tr>
<tr>
<td>Real Mode Address</td>
<td>An effective address that is equivalent to the physical address due to address translation being disabled.</td>
<td>A logical address that has a direct and fixed correspondence to a physical address. Available only when operating in real mode.</td>
</tr>
<tr>
<td>Effective Address</td>
<td>An address generated during a memory access. An effective address does not imply anything with regard to address translation or memory protection.</td>
<td>The offset portion of a segmented (real mode) or virtual (protected mode) address.</td>
</tr>
<tr>
<td>Linear Address</td>
<td>Not commonly used.</td>
<td>The address that results when segmentation is resolved; input to the paging mechanism to generate physical addresses. If x86 segmentation is disabled, linear addresses are equivalent to logical addresses.</td>
</tr>
<tr>
<td>Logical Address</td>
<td>Not commonly used.</td>
<td>A two-part (segment and offset) address used by software to indirectly reference memory; translated by the segmentation mechanism.</td>
</tr>
<tr>
<td>Virtual Address</td>
<td>An intermediate address used in the translation of an effective address to a physical address using the segmentation/paging mechanism.</td>
<td>Not commonly used.</td>
</tr>
</tbody>
</table>
I/O Address Space

Although computer systems based on x86 processors support memory-mapped I/O, they typically implement a dedicated I/O address space that is accessed using in and out instructions. PowerPC processors implement only memory-mapped I/O. Memory-mapped devices can appear anywhere in physical memory and are accessed using load and store instructions — the same as every other memory access.

Direct store segments can also be used to define I/O regions on PowerPC processors. However, the use of direct store segments is increasingly uncommon and memory-mapped I/O is the preferred I/O mechanism. Direct store segments are discussed later in this chapter.

x86 Memory Management

For early PCs, based on the 8088 and 8086 processors, memory management wasn’t an issue. Operating only in real mode, these processors performed no translation other than mathematically summing the segment and offset of a memory address. The 80286 introduced protected mode, segment-based memory management, and virtual memory. Although advanced for its time, the 80286 was still primitive by today’s standards. Serious memory management began with the i386’s support for both segment-based and paged-based memory management. And because we’re interested in only serious memory management, the x86 discussion that follows will focus on the features introduced with the i386.

Segmentation and Paging

Segmentation and paging are two independent memory management facilities on x86 processors. Employed together, they provide a rich set of address translation and virtual memory protection features. On x86 processors, paging can be enabled without segmentation, allowing the creation of virtual memory without segmentation’s extensive protection features enabled.

Figure 8-1 shows the translation required for a paged linear address and does not consider the effects of segmentation. In general, segmentation adds only an extra level of indirection to the virtual to physical address translation process.
At the top of Figure 8-1, a 32-bit linear address is broken into three parts. Bits 22–31 form a 10-bit index into the page directory table (PDT) that selects a page directory entry (PDE). Bits 12–21 form a 10-bit index into the selected page table. The page table entry supplies 20 bits that are concatenated with the low-order 12 bits of the original linear address to form the final 32-bit physical address.

When paging is enabled, the third control register (CR3) always points to the PDT. The PDT is the top-level page table and the starting point of the translation process. Each page table is 4K in size and contains 1,024 4-byte entries. The pointer to the selected page table is found at the physical
address equal to the summation of the contents of CR3 and the index into the PDT.

Similarly, bits 12–21 form the page table index in the same manner. The page table entry that is located by the newly generated index contains the high-order 20 bits of the physical address associated with the original linear address. No translation is required for the low-order 12 bits of the linear address because they map directly to the low-order 12 bits of the physical address.

The x86 protected mode segmentation mechanism adds an additional level of address indirection. The segmentation mechanism supports variable-sized segments, demand-based virtual memory, and other advanced memory management features, but its primary purpose is protection of memory space. A thorough knowledge of x86 segmentation is crucial to x86 protected mode programming — but less so with the PowerPC. Important similarities or differences between the two architectures will be noted where appropriate. (Several references that provided detailed treatment of x86 memory management are listed in the bibliography.)

**PowerPC Memory Management**

Memory management is typically the domain of operating systems and device drivers. Even if you don’t plan to write an operating system, there are still memory management issues that concern you as a programmer. The memory management configuration of a system — from the firmware level to the operating system — affects the performance and functionality of your program. Understanding the details of PowerPC memory management is another key to writing solid, efficient code.

The memory management unit (MMU) in each PowerPC implementation is responsible for the translation of effective addresses to physical addresses. Additionally, the MMU is responsible for memory protection. As described in Chapter 5, “Addressing Modes and Operand Conventions,” effective addresses on PowerPC processors are 32 bits wide and are used during memory accesses for both data and instruction fetching.

All PowerPC processors except the 601 distinguish between instruction and data memory regions. On the 603, 604, and 620, there is an MMU for instructions (IMMU) and an MMU for data (DMMU). Depending on what is being accessed, the appropriate MMU is responsible for the translation process.
Acronyms Abound: Memory Management Reference

This chapter contains more than its share of acronyms. And even though they are defined when first used, you may find it helpful to have a central reference.

Memory Management Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Meaning</th>
<th>Acronym</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Abbreviated page index</td>
<td>IMMU</td>
<td>Instruction memory management unit</td>
</tr>
<tr>
<td>ASR</td>
<td>Address space register (64-bit implementation-specific term)</td>
<td>ISI</td>
<td>Instruction storage exception</td>
</tr>
<tr>
<td>BAT</td>
<td>Block address translation</td>
<td>MMU</td>
<td>Memory management unit</td>
</tr>
<tr>
<td>BEPI</td>
<td>Block effective page index</td>
<td>PBN</td>
<td>Physical block number (601-specific term)</td>
</tr>
<tr>
<td>BLPI</td>
<td>Block logical page index</td>
<td>PTE</td>
<td>Page table entry</td>
</tr>
<tr>
<td>BRPN</td>
<td>Block real page number (601-specific term)</td>
<td>PTEG</td>
<td>Page table entry group</td>
</tr>
<tr>
<td>DBAT</td>
<td>Data block address translation</td>
<td>RPN</td>
<td>Real page number</td>
</tr>
<tr>
<td>DMMU</td>
<td>Data memory management unit</td>
<td>STE</td>
<td>Segment table entry (64-bit implementation-specific term)</td>
</tr>
<tr>
<td>DSI</td>
<td>Data storage exception</td>
<td>TLB</td>
<td>Translation lookaside buffer</td>
</tr>
<tr>
<td>HTAB</td>
<td>Hashed page table</td>
<td>VPN</td>
<td>Virtual page number</td>
</tr>
<tr>
<td>IBAT</td>
<td>Instruction block address translation</td>
<td>VSID</td>
<td>Virtual segment ID</td>
</tr>
</tbody>
</table>

Memory Management Unit Features

The PowerPC memory management unit (MMU) implements memory protection and address translation on PowerPC processors. Each processor’s MMU may have implementation-specific characteristics, but the basic feature set remains constant within 32- or 64-bit implementations. The following list highlights the feature set of the PowerPC MMU.
- 2\(^{64}\) bytes of effective address space
- 2\(^{32}\) bytes of physical address space on 32-bit implementations and 2\(^{64}\) bytes of physical address space on 64-bit implementations.
- 256MB segment size. Each segment is able to specify basic memory protection restrictions that are used in conjunction with the page table protection features.
- 4KB page size. Memory protection and access rights may be assigned on a page-by-page basis.
- 128KB to 256MB block address translation (BAT) region sizes. The BAT mechanism implements memory protection and address translation as an alternative facility to the segment/page mechanism.
- PowerPC memory protection features include no-execute segments, user/supervisor page-level protection, and user/supervisor BAT protection.
- Translation lookaside buffer (TLB) support for efficient page table lookups.

**Address Translation and Memory Protection**

PowerPC processors implement four types of address translation: real mode addressing, block address translation, segment/page translation, and direct-store I/O. The address translation mechanism used for a particular access depends on the type of access (memory or I/O) and the effective address of the access. The flowchart in Figure 8-2 diagrams the process of determining which translation mechanism will handle a particular access.

As shown in the flowchart, an effective address is generated by a read (instruction fetch or data read) or write access. The processor then checks its machine state register (MSR) to determine if address translation is enabled. If not (MSR[IR]=0 and MSR[DR]=0), then real mode addressing is employed. If address translation is enabled, the processor next checks to see if the access is to a region of memory mapped by the BAT facility. If so, it is termed a **BAT hit** and address translation proceeds using the PowerPC BAT registers. If the access is not handled by the BAT facility, the PowerPC segmentation/paging facility is responsible for translating the address.

Each type of address translation has its own protection characteristics. By the end of this chapter, you should have a sense of the translation and
Memory Management

protection facilities available using the PowerPC MMU. Chapter 11, "PowerPC Assembly Language Examples," contains source code that demonstrates how to set BAT registers.

Figure 8-2
An effective address can be translated by one of four mechanisms on PowerPC microprocessors.
Real Mode Addressing

Both the x86 and PowerPC architectures use the term real mode. Unfortunately, but not unexpectedly, they define it quite differently. On x86 processors, real mode refers to an addressing mode that allows programs that were originally written for the 8086 and 8088 to be run on later processors such as the i386 and i486. The memory management capabilities of the i386 and i486 processors are not used.

On PowerPC processors, the term real mode addressing (or real addressing mode) implies the processor state where all address translation is disabled (MSR[IR]=0 and MSR[DR]=0). In real mode addressing, virtual effective addresses are treated as physical effective addresses; all BAT and page memory protection is disabled.

When using real mode addressing, any attempt to access memory beyond the range of physical memory will cause a machine check exception and could result in a checkstop condition. Additionally, because of their reliance on address translation, the eciwx and exowx instructions will have undefined results.

Block Address Translation

Block address translation (BAT) is the second of four memory translation mechanisms that exist on all PowerPC processors. BAT mapping can be used for large contiguous regions such as floating-point lookup tables and video display buffers. The BAT facility is enabled only when address translation is enabled in the machine state register (MSR[IR]=1 and/or MSR[DR]=1).

If a particular region of memory is mapped using both the BAT facility and the segmentation/paging facility (an allowable configuration), block address translation takes priority. The BAT facility is distinguished by its ability to map an area of memory that is larger than a single page as a single unit. When using BAT translation, each block of memory is defined by a pair of BAT registers (BATU, BATL), as described in Chapter 4, “The PowerPC Programming Model.”

The details of block address translation differ between processor implementations. In particular, the 601’s BAT register definitions differ from that of the 603, 604, and 620. In addition, the PowerPC 620 BAT register format differs from that of other 32-bit implementations. Figure 8-3 shows the format of the BAT registers found on the PowerPC processors. Each of the
BAT registers shown in Figure 8-3 are described in detail in Chapter 4, “The PowerPC Programming Model.”

The 601's Upper Block Address Translation (BATU) Register

<table>
<thead>
<tr>
<th>Block Effective Page Index (BEPI)</th>
<th>WIM</th>
<th>Ks</th>
<th>Ku</th>
<th>PP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 14 15</td>
<td>24 25 27 28 29 30 31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: Key is either Ks or Ku depending on value of MSR[PR] (privilege level).

The 601's Lower Block Address Translation (BATL) Register

<table>
<thead>
<tr>
<th>Block Real Page Number (BRPN)</th>
<th>V</th>
<th>Block Length (BSM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 14 15 18 19</td>
<td>24 25 26 31</td>
<td></td>
</tr>
</tbody>
</table>

The 603, 604, and 620's Upper Block Address Translation (IBATU, DBATU) Registers

<table>
<thead>
<tr>
<th>Block Effective Page Index (BEPI)</th>
<th>Block Length (BL)</th>
<th>Vp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 14 15 18 19</td>
<td>29 30 31</td>
<td></td>
</tr>
</tbody>
</table>

The 603, 604, and 620's Lower Block Address Translation (IBATL, DBATL) Registers

<table>
<thead>
<tr>
<th>Block Real Page Number (BRPN)</th>
<th>WIMG</th>
<th>PP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 14 15</td>
<td>24 25 28 29 30 31</td>
<td></td>
</tr>
</tbody>
</table>

| Reserved | Not implemented for IBAT registers on 603, 604, 620 |

Figure 8-3
The PowerPC BAT registers map memory regions larger than a single 4K page.

The 601 type of BAT register pair maps both instruction memory and data memory. The 603, 604, and 620 processors have separate instruction (IBAT) and data (DBAT) mapping register pairs. Each BAT pair is configured by supervisor-level software using the mfspr (move from special-purpose register) and mtspr (move to special-purpose register) instructions.
A BAT register pair defines a block of memory by describing the starting address in virtual memory, the corresponding physical address, the size of the block, and the desired protection features. The biggest difference between the 601 and the 603, 604, and 620 is the maximum size of the block that a BAT register pair can define, as shown in Table 8-2. The position of the block size bit field also differs between the 601 (BATL[BSM]) and the other implementations (BATU[BL]).

### Table 8-2
BAT Register Block Size Bit Definitions for All Processor Implementations

<table>
<thead>
<tr>
<th>601’s BATL[BSM]</th>
<th>Block Size</th>
<th>603,604,620’s BATU[BL]</th>
<th>Block Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0000</td>
<td>128K</td>
<td>000 0000 0000</td>
<td>128K</td>
</tr>
<tr>
<td>00 0001</td>
<td>256K</td>
<td>000 0000 0011</td>
<td>256K</td>
</tr>
<tr>
<td>00 0011</td>
<td>512K</td>
<td>000 0000 0111</td>
<td>512K</td>
</tr>
<tr>
<td>00 0111</td>
<td>1M</td>
<td>000 0000 1111</td>
<td>1M</td>
</tr>
<tr>
<td>01 1111</td>
<td>2M</td>
<td>000 0000 1111</td>
<td>8M</td>
</tr>
<tr>
<td>11 1111</td>
<td>4M</td>
<td>000 0011 1111</td>
<td>16M</td>
</tr>
<tr>
<td></td>
<td>8M</td>
<td>000 1111 1111</td>
<td>32M</td>
</tr>
<tr>
<td></td>
<td>16M</td>
<td>011 1111 1111</td>
<td>64M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>128M</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>256M</td>
<td></td>
</tr>
</tbody>
</table>

As shown in Table 8-2, the smallest memory region that can be mapped using the BAT mechanism is 128K \((2^{17}\) bytes). As a result, the low-order 17 bits of the effective address are always treated as an offset within the block and require no translation. The remaining 15 bits must be specified in the BRPN field to completely identify the base physical address of the BAT-mapped block.

**Address Recognition and Translation**

A BAT hit occurs when the processor recognizes an effective address as being mapped by a BAT register pair. Chapter 4, “The PowerPC Programming Model,” describes the address comparison process that occurs during a BAT hit. When an effective address hits in one of the BAT register pairs, that BAT pair is used in the effective address to physical address translation.
Figure 8-4(a) shows how the specific fields of the address and BAT registers fit into the effective address to physical address translation process.

(a) The PowerPC BAT Translation Process

(b) Plugging in Sample Values to Verify the Translation Logic

The PowerPC block address translation process vaguely resembles the x86 paging translation mechanism.
The low-order 17 bits of the effective address are used directly as an offset within the block and require no translation. The upper 4 bits of the effective address play no part in the translation process. Instead, they’re used in the determination of a BAT hit for the memory access we’re translating.

The 11 bits that remain (bits 4–14 of the effective address) are ANDed with the block length field of the BAT registers (either BL or BSM) to give the upper 11 bits of the offset within the block. The ANDing process removes any high-order bits that will be supplied by the BRPN (the base physical address) field and keeps all bits not supplied by the BRPN field.

Note that the full acronym corresponding to the BRPN field is *block real page number*. This is a case of Motorola/IBM terminology contention; recall that the PowerPC architecture specification uses the term *real memory* in place of *physical memory* (used by Motorola and in this book).

In the next step, the high-order 4 bits of the BRPN field are copied to the high-order 4 bits of the physical address. The low-order 11 bits of the BRPN field are ORed with the high-order 11 bits of the offset within the block calculated in the previous step. This ensures that any bits saved by the previous AND operation appear in the physical address. The low-order 17 bits (simply along for the ride) are then copied into the final physical address.

Figure 8-4(b) shows the same process with sample values in the fields. An effective address (0xf8001000) that hits in a 128K BAT region is translated into its physical address equivalent (0x01e01000).

This BAT translation example uses a 32-bit effective address and, therefore, the 32-bit translation mechanism. On a 64-bit processor, such as the 620, the 64-bit effective address is translated in an analogous fashion. However, the size of the BRPN field in the BAT register is 36 bits.

**BAT Memory Protection**

A memory access that hits in a BAT register-mapped block of memory is subject to the memory protection specified by the BAT register pair. Table 8-3 lists the various combinations of protection settings that the BAT mechanism provides. Note that the Vs and Vp fields (valid bits) of the 603, 604, and 620 are equivalent to the 601’s Ks and Ku fields. The valid bits are used in the determination of a BAT hit and correspond to the *Ignored* entries in Table 8-3; they do not play an explicit role in the BAT protection mechanism. The access protection bits (the PP field), which are located within the
BAT register, define memory protection details for the block. In particular, the PP bits define read, write, and no-access properties for BAT regions.

**Table 8-3**

BAT Register Bit Settings and Protection Summary for All PowerPC Implementations

<table>
<thead>
<tr>
<th>Bit Settings</th>
<th>Allowed Access Information</th>
<th>Action Taken as a Result of a Particular Privilege-Level BAT Region Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vs Vp PP or or Ks Ku</td>
<td>Privilege Level</td>
<td>Access Rights</td>
</tr>
<tr>
<td>0 0 xx</td>
<td>Both levels</td>
<td>No access</td>
</tr>
<tr>
<td>0 1 00</td>
<td>User</td>
<td>No access*</td>
</tr>
<tr>
<td>0 1 x1</td>
<td>User</td>
<td>Read only</td>
</tr>
<tr>
<td>0 1 10</td>
<td>User</td>
<td>Read/write</td>
</tr>
<tr>
<td>1 0 00</td>
<td>Supervisor</td>
<td>No access*</td>
</tr>
<tr>
<td>1 0 x1</td>
<td>Supervisor</td>
<td>Read only</td>
</tr>
<tr>
<td>1 0 10</td>
<td>Supervisor</td>
<td>Read/write</td>
</tr>
<tr>
<td>1 1 00</td>
<td>Both levels</td>
<td>No access*</td>
</tr>
<tr>
<td>1 1 x1</td>
<td>Both levels</td>
<td>Read only</td>
</tr>
<tr>
<td>1 1 10</td>
<td>Both levels</td>
<td>Read/write</td>
</tr>
</tbody>
</table>

x = Don’t care bit

* = Exception is generated; otherwise access is ignored

In Table 8-3, the **allowed** entries refer to legal accesses to BAT regions of memory. The **ignored** entries refer to the fact that the memory access is not translated by the BAT mechanism. Instead, the access is handed off to the segmentation/paging mechanism described in the next section. There is no further BAT involvement for such an access.
The *generates exception* entries correspond to a BAT protection violation. In cases where the BAT-established protection is violated, the following exception conditions result:

- A data access that violates BAT-established memory protection results in a data storage exception (DSI), described in Chapter 10, "Exceptions and Interrupts." On entry to the exception handler, DSISR[4] will be set to indicate that the exception was caused by a data memory access.

- Any access to instruction memory (such as an instruction fetch operation) that violates BAT-established memory protection will result in an instruction storage exception (ISI), described in Chapter 10, "Exceptions and Interrupts." On entry to the exception handler, SRR1[4] will be set to indicate that the exception was caused by an instruction memory access.

The protection configuration established by a BAT register pair is consistent throughout the entire region mapped by that pair. This differs from the PowerPC segmentation/paging mechanism, which can implement protection on a page-by-page basis.

**PowerPC Segmentation and Paging**

Segmentation and paging implement virtual memory on PowerPC processors — as an alternative to the BAT virtual memory mechanism. A potentially confusing situation arises because both terms (segmentation and paging) exist within the context of x86 memory management; however, the implementation of segmentation and paging differs dramatically between the two architectures.

An effective address will be translated using the segmentation and paging mechanism only if the translation is not performed by the BAT mechanism. That is, the BAT mechanism takes precedence over the segmentation and paging mechanism. Unlike the x86 segmentation and paging mechanisms, PowerPC segmentation and paging are inseparable architectural features. This is an important distinction. Therefore, during subsequent discussions, the process of address translation using the PowerPC segmentation/paging mechanism shall be referred to as *page translation*.

**Segments and Segment Descriptors**

PowerPC segments are 256MB regions of virtual memory. Each segment is made up of 4K pages that contain information used during the translation of effective addresses to physical addresses. A PowerPC segment is defined by a segment descriptor. As shown in Figure 8-5, segment descriptors exist
in two forms, depending on the width of the processor implementation. On 32-bit implementations, segment descriptors are stored in one of the 16 32-bit segment registers. On 64-bit implementations, segment descriptors are stored as 128-bit entries in a segment table. The segment table exists in physical memory and is pointed to using the address space register (ASR). Both the ASR and segment registers are defined in Chapter 4, "The PowerPC Programming Model."

<table>
<thead>
<tr>
<th>64-bit Implementation: Segment Table Entry Format is 128 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effective Segment ID (ESID)</td>
</tr>
<tr>
<td>00000000000000000000000000000000 V T KsKp N 0 0 0 0</td>
</tr>
<tr>
<td>0 35 36 55 56 57 58 59 60 61 63</td>
</tr>
</tbody>
</table>

| Virtual Segment ID (VSID)                                   |
| 000000000000000000000000000000000000000000000000000000000000000 |
| 0 51 52 63                                                   |

<table>
<thead>
<tr>
<th>32-bit Implementation: Segment Descriptor is 32-bit Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>T KsKp N 0 0 0 0</td>
</tr>
<tr>
<td>Virtual Segment ID (VSID)</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31</td>
</tr>
</tbody>
</table>

Reserved

**Figure 8-5**

Segment descriptors differ between 32-bit and 64-bit PowerPC implementations.

Each segment descriptor, independent of implementation, contains enough information to determine the segment type (T bit), the virtual segment ID (VSID), and privilege-level attributes. If the T bit of the segment descriptor is clear, the segment descriptor defines a memory segment; if the T bit is set, the descriptor defines a direct store segment. For now, we'll limit our attention to memory segments. Direct store segments are covered later in this chapter.

Additionally, each segment descriptor contains a no-execute bit. The no-execute (N) bit represents the only protection attribute that segment descriptors may specify — all other protection details are defined in the page table entry data structure (discussed in the following section). When
the N bit is set in a segment descriptor, any attempt to execute instructions from memory mapped by that segment will result in an ISI exception.

Not only are all PowerPC segments 256MB (2^{28} bytes) in size, they must also be aligned on 256MB boundaries. And when using 32-bit effective addresses (EA), that leaves 4 bits in the EA to specify the segment. The highest-order 4 bits of an effective address (EA[0-3]) index into one of 16 segment registers.

Upon determining that an access will be translated by the page translation mechanism, the processor checks the segment descriptor to determine the type of segment: direct store or memory. The flow of this process is shown in Figure 8-6. Page translation specifically implies effective address translation using memory segments and is the focus of this discussion. Direct store segments are discussed later in this chapter.

---

**Figure 8-6**
The segmentation/paging translation mechanism handles both direct store segment access and memory segment accesses.
The page translation and BAT mechanisms interpret effective addresses differently. To understand the details of page translation, it is useful to examine the format of an effective address. Figure 8-7 shows how 32-bit effective addresses are interpreted during the page translation process.

### PowerPC 32-bit Effective Addresses

Seen by the Segmentation/Paging Mechanism

<table>
<thead>
<tr>
<th>Segment Register Number</th>
<th>6-bit API</th>
<th>10 bits</th>
<th>Byte Offset 12 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4</td>
<td>19 20</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

16-bit Page Index

**Figure 8-7**

A 32-bit effective address is interpreted as a series of fields that depend on the method of translation.

### Page Table Entries

Page table entries (PTEs) are data structures containing information used to translate virtual memory addresses (effective addresses) to physical memory addresses. Each PTE defines a 4K page; each page maps 4K of effective address to 4K of physical addresses. The PTEs are created and maintained by system software, such as an operating system. PTEs are stored in page tables using a hashing algorithm (discussed in a following section).

On 32-bit implementations, PTE entries occupy two 32-bit words. On 64-bit implementations, a PTE entry occupies two 64-bit doublewords. Although the same bit fields exist in both PTE formats, exact bit field position and size depends on the PTE format. This is an important consideration when designing software that must run on both 32-bit and 64-bit processor implementations.

A PTE entry for both a 32-bit and 64-bit implementation is shown in Figure 8-8. Each PTE defines the various characteristics of page-based address translation and memory protection. In particular, the fields that constitute a PTE are as follows:

- **The virtual segment ID (VSID)** corresponds to the high-order bits of the virtual page number. This field is used in conjunction with other fields (H, V, API) to find the PTE in a translation lookaside buffer or page table.
The abbreviated page index (API) field works in conjunction with the VSID to determine if the PTE is a match for the virtual address by direct comparison.

The valid (V) bit and the hash function identifier (H) ensure that only valid PTE entries are part of virtual address comparisons.

The referenced (R) bit and the changed (C) bit are used by the operating system to track page usage history. This information is then used to determine which pages should be swapped in from or out to disk.

The WIMG (write, cache inhibited, memory coherency, and guarded) bits describe the allowed cache behavior for the memory described by the PTE.

The PP (page protection) bits define the protection characteristics of the page. These bits control the read and write permissions for the page.

---

### 32-bit Page Table Entry Format

<table>
<thead>
<tr>
<th>V</th>
<th>Virtual Segment ID (VSID)</th>
<th>H</th>
<th>Abbreviated Page Index (API)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>API</td>
<td>0 0 0 R C WIMG 0 PP</td>
</tr>
</tbody>
</table>

### 64-bit Page Table Entry Format

<table>
<thead>
<tr>
<th>Virtual Segment ID (VSID)</th>
<th>API</th>
<th>0 0 0 0 0 H V</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>API</td>
<td>0 0 0 R C WIMG 0 PP</td>
</tr>
</tbody>
</table>

---

**Figure 8-8**  
The field positions within page table entries differ depending on implementation width.

Now that we’ve defined the various components of page translation, we’re ready to examine the translation process itself. The following section will step through each stage of translating an effective address to a physical address using the PowerPC page translation mechanism. Remember — page translation implies both the segmentation and paging facilities of PowerPC processors.
Page Translation

The aim of the PowerPC implementation of the page translation mechanism is to minimize the time required to translate an effective address to a physical address.

As mentioned in the previous section, there are two key data structures associated with page translation: segment descriptors and page table entries. Segment descriptors partition memory into 256MB regions, and page table entries (PTEs) supply the information necessary to translate effective addresses to physical addresses.

Figure 8-9 shows the steps involved in the translation of a 32-bit effective address to a 32-bit physical address using the PowerPC page translation facility. The differences between 32-bit and 64-bit page translation are covered in a following section; during this discussion, we'll use a 32-bit effective address to examine the page translation process.

---

**Figure 8-9**

A 32-bit effective address is temporarily converted to a 52-bit virtual address before translation to a 32-bit physical address.
The effective address to physical address translation begins by using the high-order 4 bits of the effective address to select a segment register. The 24-bit virtual segment ID (VSID) entry from the segment register is then prepended to the low-order 28 bits of the effective address. (The 16-bit page index and 12-bit byte offset are not modified from their value in the effective address.) The result is a 52-bit virtual address. This virtual address is an internal processor representation used only as an intermediate step during translation.

This is an appropriate time to review the acronyms defined earlier in the chapter. The high-order 40 bits of the virtual address are known as the virtual page number (VPN); the lower 12-bit offset is unchanged from the original effective address. Conversion of the VPN to a 20-bit real page number (RPN) requires information contained in a PTE. To locate the PTE that corresponds to the VPN, the page table is searched. However, to increase the efficiency of PTE lookup (and consequently overall performance), some PowerPC processors implement a translation lookaside buffer (TLB).

The 601, 603, 604, and 620 each implement a TLB, discussed later in this chapter. If the PTE cannot be located in either the TLB or page table, a page fault exception is generated so that system software can update the page tables. Note that if a PTE search fails for a memory access to data memory, a DSI exception results; an ISI exception is generated for accesses to instruction memory.

To facilitate the mapping of a VPN to RPN, the PowerPC architecture specifies a hashing algorithm to place PTEs into page tables. The hashing algorithm is implemented by system software as a portion of its memory management responsibilities. Using this hashing algorithm, a page table entry (containing the RPN) that corresponds to a particular VPN can be located efficiently within the hashed page table (HTAB).

After locating the RPN from the PTE, the 32-bit physical address is formed by concatenating the 20-bit RPN with the 12-bit offset field. Translation is complete.

Both the TLB and the HTAB are of finite size. As such, valid translations might not be found in the HTAB for all memory accesses during execution. Specifically, the HTAB and TLB might be too small to contain all of the currently valid translations. In such cases, DSI/ISI exceptions are generated to signal the system software that it must handle the situation.

**Page Protection**

The majority of memory protection mechanisms used by segment/page address translation reside at the page level and are defined by entries in the
PTE. However, segment descriptors support a no-execute option which prevents instructions from being fetched from a segment so marked. An attempt to fetch instructions from a no-execute segment results in an access exception.

Page-level memory protection supports limiting access to read-only, read-write, and specific privilege-level restrictions. Like the BAT protection mechanism, memory protection based on the segmentation/paging mechanism defines supervisor-level and user-level key bits as well as page-based protection bits. Two key bits (Ks and Kp, shown previously in Figure 8-5) are located in the segment descriptors and correspond to the user and supervisor privilege levels. The state of each key bit is used in conjunction with the page protection bits located in the page table entries (PTE[PP]). For example, the memory protection for a supervisor-level memory access would be governed by the settings of the supervisor-key bit (Ks) and the PTE[PP] field. Each field that defines the memory protection associated with a particular page is discussed in the following paragraphs.

The processor uses three important bit fields to implement the page protection mechanism: the privilege-level bit of the MSR (MSR[PR]), the segment descriptor's Ks and Kp bits, and the page protection bits of a PTE (PTE[PP]).

The privilege-level bit in the machine state register (MSR[PR]) indicates the privilege level under which the access is occurring. When MSR[PR]=1, the memory access is a user-level access. When MSR[PR]=0 (clear), the memory access is a supervisor-level access.

The Ks and Kp bits are located in the segment descriptor. Depending on the privilege level of the memory access, either the Ks or Kp bit will be used to determine the access restrictions for the page. Furthermore, the Ks and Kp bits will be used in conjunction with PTE[PP] bits to determine read/write restrictions. Ks is the supervisor-level key bit and Kp is the user-level key bit. Note that the Ks and Kp bits are defined for segment descriptors only and are not related to either the Ks or Ku bits of the 601's BAT register.

The PP bits located in the PTE define the memory protection details for the page. The PP bits for page-based memory protection are always used in conjunction with the key bit (Ku or Kp) of the segment descriptor.

When a program attempts to access memory under control of the segment/page protection mechanism, MSR[PR] is examined and its value selects the key bit from the segment descriptor. If MSR[PR]=0, the Ks key is selected; if MSR[PR]=1, Kp is selected.

The selected key, in conjunction with the PTE[PP] bits, determines if the current access is of the correct type and privilege level. Table 8-4
summarizes the PowerPC page level protection features with respect to the key bit extracted from the segment descriptor.

**Table 8-4**
Page Translation Memory Protection Summary for All PowerPC Implementations

<table>
<thead>
<tr>
<th>PP Bits</th>
<th>Privilege Level and Type of Access Attempted</th>
<th>User Read Access (key = 1)</th>
<th>User Write Access (key = 1)</th>
<th>Supervisor Read Access (key = 0)</th>
<th>Supervisor Write Access (key = 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Supervisor read/write</td>
<td>Generates exception</td>
<td>Generates exception</td>
<td>Allowed</td>
<td>Allowed</td>
</tr>
<tr>
<td>01</td>
<td>Supervisor write-only</td>
<td>Allowed</td>
<td>Generates exception</td>
<td>Allowed</td>
<td>Allowed</td>
</tr>
<tr>
<td>10</td>
<td>Both user and supervisor read/write</td>
<td>Allowed</td>
<td>Allowed</td>
<td>Allowed</td>
<td>Allowed</td>
</tr>
<tr>
<td>11</td>
<td>Both user and supervisor read-only</td>
<td>Allowed</td>
<td>Generates exception</td>
<td>Allowed</td>
<td>Generates exception</td>
</tr>
</tbody>
</table>

A generates exception entry in Table 8-4 means that an access under the stated condition causes a segment/page memory protection violation. The processor responds by generating either an ISI or DSI exception condition, depending on whether the attempted memory access was for an instruction or data, respectively. The allowed field indicates that the access is legal and address translation will proceed normally.

Table 8-4 summarizes the conditions that will result in memory protection violations during page translation. However, there are some situations worthy of special mention. For example, any user-level access (key bit = Kp = 1) to memory when privilege bits are cleared (PP = 0b00) causes a protection violation exception. Similarly, any write access when PP = 0b11 violates memory protection and generates an exception.

**Hashing and Hashed Page Tables**

Page table lookup is an important topic for PowerPC programmers for two reasons: First, the implementation of TLBs on PowerPC processors differs dramatically from those of the x86 architecture. Second, all modern
operating systems depend on virtual memory and overall system performance is largely dependent on the overhead associated with page table management.

Performing the lookups required to resolve an address for every memory address retards processor performance. To alleviate this condition, PowerPC processors employ a translation lookaside buffer (TLB) to cache the most recently used effective address translations.

The management of TLBs can take place in hardware, both hardware and software, or entirely in software. PowerPC processors use a combination of hardware and software. However, some PowerPC implementations (such as the 603) use strictly software to manage the TLBs. This is observable in Chapter 10, "Exceptions and Interrupts," where additional 603 exception conditions are defined, specifically for TLB management.

Earlier in the chapter, we saw how an effective address is translated to a physical address when segmentation and paging are in effect. This process was diagrammed in Figure 8-9. In one of the pictured steps, the virtual page number (VPN) passes through a block marked "TLB/page table entry lookup." On exit from the block, the page table entry has been located. The process of locating the appropriate PTE is the subject of this section. The remainder of this discussion refers to Figure 8-10.

As stated previously, the TLB is simply a cache that holds recently used page table entries. When seeking a PTE, the processor first checks the TLB. If present (a TLB hit), the PTE is retrieved from the TLB and the RPN is extracted and the address translation proceeds with almost no delay.

If the PTE isn't found in the TLB (a TLB miss), the processor must resort to searching the page tables in main memory to find the required PTE. The page table search process uses a hashing algorithm specified by the PowerPC architecture and implemented by the operating system to distribute PTEs evenly within the page tables.

Hashing involves performing arithmetic transformations on the keys to be used for a search — in this case the RPN of the PTE. There have been a number of well-written articles describing the benefits of a hashed page table for large virtual address spaces. Since the PowerPC architecture supports 64-bit effective addresses, a hashed page table is a good performance/space solution. That is, hashed page tables maximize performance (by decreasing PTE lookup overhead) while minimizing the physical memory required to store the tables.
**Figure 8-10**

A virtual page number locates a PTE in either the TLB or page table.

### Page Translation on 64-Bit Implementations

Segmentation on 64-bit PowerPC processors such as the 620 differs considerably from that used on 32-bit implementations. The *address space register* (ASR), found exclusively on 64-bit implementations, points to a table of segment descriptors in physical memory. In concert with the ASR, these descriptors, known as *segment table entries* (STEs) eliminate the need for segment registers on 64-bit processors.

The segment table is 4K \((2^{12} \text{ bytes})\) in length and must begin at a 4K boundary in physical memory. Consequently, the lower 12 bits of the table’s base address will always be zeroes. The base physical address of the segment table is stored in the ASR.
When address translation is enabled (MSR[IR]=1 or MSR[DR]=1), it's good practice to ensure that the ASR is loaded with the address of a valid segment table. Unless every memory access can be translated by the BAT mechanism, the ASR will eventually be used to find the segment table and segment table entries. If the ASR points to an invalid memory region or an uninitialized table, a machine check exception may be generated. The ASR should never be initialized to any of the first three 4K regions of physical memory (0x0000, 0x1000, or 0x2000). These addresses are reserved for use by the PowerPC exception vector table.

The segment table entries are 128 bits wide and are placed in the segment table using a hashing algorithm. The STE controls the segment table search process and defines memory protection features for the segment. Figure 8-5 compares the format of 32- and 64-bit segment descriptors. Using the STEs, 64-bit processors are able to translate effective addresses used to access memory segments in a similar manner to that found on 32-bit processors. Figure 8-11 shows the 64-bit effective address to physical address translation process.

![Figure 8-11](image)

Page translation on 64-bit PowerPC processors use information found in segment table entries.
PowerPC Direct-Store Segments

The direct-store interface is available on PowerPC processors only to support legacy direct-store devices from the POWER architecture. The direct-store mechanism is not optimized for performance and should only be used when absolutely required by a device. The recommended (and more efficient) method of implementing I/O devices with PowerPC processors is to memory map all I/O regions. When the T bit of a segment descriptor is set, the descriptor defines the region of memory that is to be used as a direct-store segment.

Direct-store segments translate the effective addresses of memory access only when the access is not handled by the BAT mechanism. All direct-store segments are cache inhibited and all accesses completely bypass the cache.

There is no page-level address translation or page-level protection for direct-store segments. However, the Ks and Kp key bits from the segment descriptor are sent to the memory controller for protection validation. Typically, the memory controller will not implement any protection based on the descriptor's key bits.

The likelihood of direct-store devices being used with PowerPC systems intended for the home and desktop is quite low. Therefore, we can eschew an in-depth discussion of direct-store segments.

Summary

Memory management on PowerPC processors is a complex and far-reaching topic. We've covered enough information to understand the basic address translation and memory protection mechanisms. When programming PowerPC systems, the information contained in this chapter is often sufficient in providing enough information to create efficient, solid code. However, if you are an operating system programmer or have an interest in more resolved detail, I suggest you read the user's manual for the PowerPC processor that you are currently working with, as the finer points tend to be hardware implementation dependent.
The PowerPC Cache

"Caches are not really all that difficult to understand."
— The Cache Memory Book by Jim Handy

The details of cache design can fill an entire book — and do. Several references are given in the bibliography. And although a complete discussion of cache design is beyond the scope of this book, it makes sense to review the aspects of caching that apply directly to the PowerPC family.

At a basic level, the job of a microprocessor is to perform operations on data. And the faster that the processor can access data, the greater the chance that the processor can achieve its maximum performance. The fastest form of memory on processors is register memory. Fortunately, PowerPC processors have plenty of registers. Of course, you can’t store an entire program and its associated data in the processor’s registers alone. That’s where main memory comes in.

Programs and data are stored in the comparative vastness of a computer system’s main memory. Unfortunately, accessing main memory is much slower than accessing data in registers. To speed things up, an image of a portion of main memory can be stored in a processor’s on-chip cache. The on-chip cache is memory that’s considerably faster than main memory and much larger than the processor’s register file. The concept of a
cache arises out of a memory hierarchy as shown in Figure 9-1. The i486 and all PowerPC processors implement an on-chip cache. The size, configuration, and style may vary between processors, but they all share the same basic operation.

Microprocessor (registers)

Level 1 Cache (internal or on-chip)

Level 2 Cache (usually external to processor)

Physical Memory Storage (disk drive)

**Figure 9-1**
The further away from the microprocessor — the slower the memory system.

**PowerPC Caches**

All current PowerPC implementations have an on-chip level 1 cache. The size and feature set vary from implementation to implementation. The PowerPC architecture does not specify the type, organization, implementation, or even the existence of a cache for each processor. As this book is being written, all PowerPC processors implement on-chip caches.

The 601 implements a single, unified (instruction and data) cache. All other PowerPC implementations use separate caches for code and data (Harvard architecture cache model). If a future PowerPC implementation does not implement an on-chip cache, the architecture specification guaran-
tees that all cache-oriented instructions will not halt processor operation; however, they may cause exceptions.

**Table 9-1**

<table>
<thead>
<tr>
<th>Processor Implementation</th>
<th>Cache Size (data/instruction)</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPC 601, 601+</td>
<td>32K, unified</td>
<td>Eight-way</td>
</tr>
<tr>
<td>PPC 602</td>
<td>4K/4K</td>
<td>Two-way</td>
</tr>
<tr>
<td>PPC 603</td>
<td>8K/8K</td>
<td>Two-way</td>
</tr>
<tr>
<td>PPC 603e</td>
<td>16K/16K</td>
<td>Two-way</td>
</tr>
<tr>
<td>PPC 604</td>
<td>16K/16K</td>
<td>Four-way</td>
</tr>
<tr>
<td>PPC 620</td>
<td>32K/32K</td>
<td>Four-way</td>
</tr>
</tbody>
</table>

**CACHE ARCHITECTURE**

The description of the 604's cache configuration is a good place to start defining our terms. Although we'll refer specifically to the PowerPC 604 in this section, the cache terminology that is discussed generally applies to any cache on any processor. For the purposes of our cache terminology discussion, we'll use the following description of the 604's cache implementation: on-chip, 16K, four-way set-associative, physically indexed data and instruction caches.

The fact that the 604's cache is contained within the processor is important. An on-chip (level 1) cache must be distinguished from an off-chip (level 2) cache. The 604 also has separate caches for instructions and data, and thus conforms to the Harvard architecture. In contrast, a single (unified) cache for both instructions and data is said to conform to the Von Neumann cache architecture. The size of each cache on the 604 is 16K.

A cache's design restricts where data from a main memory address can appear within the cache. A fully associative cache doesn't impose any restrictions on address placement within the cache. This method produces a high hit-to-miss ratio at the expense of speed. A direct-mapped cache, where each main memory address can appear in only one location, operates more quickly when the memory request is a cache hit.
A set-associative cache strikes a balance between the two approaches. Any particular memory address can be mapped to only a subset of the cache entries. Thus the speed of cache searches is not unreasonably degraded and the hit-to-miss ratio is increased. The 604's cache is four-way set-associative, meaning that there are four possible cache locations in which data from a particular address in main memory can be stored.

Each 16K cache is divided into four 4K arrays, called ways. Each way is then divided into 128 (4K/32 bytes) sets of 32-byte cache lines. Each cache line holds 32 bytes of contiguous main memory. The first byte in each cache line corresponds to a main memory address that is evenly divisible by 32 (0 × 20) bytes — a 256-bit boundary. Residing on an 8-word boundary implies that the low-order 5 bits of the first byte's effective address are zeroes. The 8-word alignment of cache blocks corresponds directly to the alignment of page boundaries.

How a 32-Bit Effective Address is Cached

<table>
<thead>
<tr>
<th>Cache Tag</th>
<th>Set Address (using set #1)</th>
<th>Byte Within Block 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19 20</td>
<td>26 27 31</td>
</tr>
</tbody>
</table>

![Diagram showing how a 32-bit effective address is cached](image)

Figure 9-2

A 32-bit effective address is decomposed into three components that are used to index into the cache.

In Figure 9-2, we see that the lower 5 bits of the effective address are used to select a byte within a 32-byte cache block. Bits 20 - 26 of the effective address are used as an index to select one of the 128 sets. The high-order bits of the effective address EA[0-19] correspond to the cache tag.

The features of the virtual environment architecture (VEA) are accessible to both user- and supervisor-level software. Therefore, both applications
and system software can access the cache management features of the processor.

**The Virtual Environment**

The PowerPC architecture specifies a VEA that defines caches, virtual memory, and the multiprocessing features of PowerPC implementations. Because of the emphasis placed on multiprocessing ability, programmers should be aware that they are working in an environment where physical memory may be shared between two or more processes — or processors! The virtual environment architecture defines and ensures the following virtual memory features on PowerPC processors: atomic memory accesses, memory access ordering, and memory coherency.

An atomic memory access is a memory access that completes without fragmentation. In this context, fragmentation means a load or store memory access that is interrupted (fragmented) by some external event, such as an exception or memory access by another processor in a multiprocessor system. Atomic memory accesses are a concern on multiprocessing systems due to the potential for a different processor to need to access the same region of memory. Atomic accesses guarantee that one processor won’t read (or write) to the same location while it is being used by another.

On PowerPC processors, specific instructions exist to guarantee atomic memory accesses. The `ldarx/stdcx` and `lwarx/stwcx` instruction pairs generate a reservation that can be used to ensure an atomic memory access. Reservations are discussed in Chapter 6, “The PowerPC Instruction Set.”

**Memory Access Ordering**

Memory access ordering refers to the specific order in which the processor performs load and store memory accesses and the order in which those accesses complete. As described in Chapter 6, “The PowerPC Instruction Set,” synchronization must be considered when in-order accesses are required. The `eieio` and `sync` instructions order load and store operations.

The `eieio` instruction provides software control of the order in which loads and stores are performed for certain types of memory, such as memory-mapped I/O. For example, the `eieio` instruction ensures that a
sequence of accesses to an I/O device's control registers are performed in the desired order.

The `sync` instruction ensures that all memory accesses are coherent. That is, there is agreement (between processors and processes) as to the contents and access rights of a particular memory location at the time that access occurs. A program can use the `sync` instruction to guarantee that all other updates to a shared memory location have completed before performing a subsequent access.

**Coherency**

Coherency may be generally defined as an orderly and structured relationship between independent components of a system. And memory coherency refers to the state of agreement concerning the contents of a shared memory location between two or more elements of a computer system. One of the goals of a coherent memory system is to provide the same image of memory to all devices using the system.

Coherency enables cooperative use of shared resources such as main memory. If multiple devices use shared memory locations without ensuring coherency, each device may write (and subsequently read) a different value from the same location. In addition to the `eieio` and `sync` instructions, the `lwarx` and `stwcx` instructions may be used to ensure coherent load and store operations.

The VEA specifies a "weakly consistent" memory model for systems that implement shared memory. This configuration places the responsibility of ordering memory accesses on the programmer. Because the memory ordering responsibility is not placed on the processor, there is significant opportunity for improved memory system performance.

**Cache Access Attributes**

In general, all instruction and data accesses on PowerPC processors are performed under the control of four memory/cache access attributes. These attributes — termed WIMG attributes — are programmed into the memory management data structures of the processor by operating system software during initialization. In particular, both page table entries (PTEs) and BAT registers contain WIMG attribute bits, as described in Chapter 8, "Memory Management." Consequently, system software may define the WIMG
attributes on a per-page (for page-translated memory) and per-block (for BAT-translated memory) basis.

The WIMG attributes are defined as individual bits. The W and I attributes control how the processor itself uses the cache. The M attribute controls coherency for the addressed memory location. The G attribute can prevent out-of-order loading and prefetching from the addressed memory location. Each of these attributes are described in the following list.

- **Write-through (W)**
  When W = 1, the memory access is designated as write-through. Data from the CPU is simultaneously written to both the cache and main memory to guarantee coherency.

- **Caching-inhibited (I)**
  When I = 1, the memory access is caching-inhibited. The cache is bypassed and the load or store is performed using main memory. During a caching-inhibited access, the location in main memory is prevented from loading into the cache.

- **Memory coherency (M)**
  When M = 1, the processor enforces memory coherency by ensuring that store operations by all processors to the same memory location are serialized. The memory coherency attribute provides a software-based alternative to hardware-enforced memory coherency.

- **Guarded (G)**
  When G = 1, the processor will inhibit out-of-order memory fetches and accesses at the cost of performance.

There are six combinations of the WIM bits that are supported by PowerPC processors. In each case, the guarded bit (G) may be either set or cleared without impact on the remaining WIM bits. Table 9-2 summarizes the WIM combinations.

### Cache Management Instructions

There are both user- and supervisor-level PowerPC cache management instructions. Furthermore, cache management may be undertaken for purposes ranging from application software performance tuning to operating system memory management functions. For example, several user-level cache instructions (dcbt and dcbtst) discussed in this section exist solely for performance optimization. An understanding of their operation will be useful for both system-level and application-level programmers.
<table>
<thead>
<tr>
<th>W-I-M Bit Setting</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Data and instructions are cached. A load or store operation whose target hits in the cache may use that entry in the cache. The processor does not need to enforce memory coherency for accesses it initiates.</td>
</tr>
<tr>
<td>001</td>
<td>Data is cached. A load or store operation whose target hits in the cache may use that entry in the cache. The processor enforces memory coherency for accesses it initiates.</td>
</tr>
<tr>
<td>010</td>
<td>Caching is inhibited. The access is performed to memory and completely bypasses the cache. The processor does not need to enforce memory coherency for accesses it initiates.</td>
</tr>
<tr>
<td>011</td>
<td>Caching is inhibited. The access is performed to memory and completely bypasses the cache. The processor enforces memory coherency for accesses it initiates.</td>
</tr>
<tr>
<td>100</td>
<td>Data is cached. A load operation whose target hits in the cache may use that entry in the cache. Store operations are written to memory. The target location of the store may be cached and is updated as a hit in the cache. The processor does not need to enforce memory coherency for accesses it initiates.</td>
</tr>
<tr>
<td>101</td>
<td>Data is cached. A load operation whose target hits in the cache may use that entry in the cache. Store operations are written to memory. The target location of the store may be cached and is updated as a hit in the cache. The processor enforces memory coherency for accesses it initiates.</td>
</tr>
<tr>
<td>11x</td>
<td>Not supported.</td>
</tr>
</tbody>
</table>
This section focuses on the cache management instructions that are available to software of both privilege levels. In general, there are separate instructions for controlling the instruction and data caches. However, on implementations that have a unified cache (such as the PowerPC 601), instruction-cache control instructions should be used. Note that any cache management instruction that generates an effective address corresponding to a direct-store segment (discussed in Chapter 8, “Memory Management”) is treated as a no-op by the processor.

**dcbt and dcbtst**

The user-level dcbt (data cache block touch) instruction provides a means for improving performance through the use of software-initiated prefetch hints. Note, however, that use of this instruction does not guarantee that a cache block will be fetched. In general, software uses the dcbt instruction to request a cache block fetch in anticipation of its need by the program. Subsequently, the program can use the data from the cache as opposed to more costly accesses to main memory.

The user-level dcbtst (data cache block touch for store) instruction is similar to the dcbt instruction. Where dcbt is used for load operations, dcbtst is used for store operations. In particular, software may use dcbtst to request a cache block fetch to guarantee that a subsequent store will be to a cached location.

The dcbt (data cache block touch) and dcbtst (data cache block touch for store) instructions are provided strictly for performance optimization. These instructions do not affect the correct execution of a program independent of the success (or failure) of the cache block fetch as described in the following section.

**dcbz**

The dcbz (data cache block set to zero) instruction clears a single cache block. During execution, the dcbz instruction is treated as a store to the target effective address using both memory protection and address translation. The following list summarizes the operation of the dcbz instruction.
- If the target address resides in the data cache, all bytes of the cache block are cleared.
- If the target address is not in the data cache and caching is allowed for the corresponding page, the cache block is established in the cache and all bytes of the cache block are cleared. Note that establishing the block does not require fetching the cache block from main memory.
- If the target address is designated as either caching-inhibited or write-through, there are two possible results. Either all bytes in main memory that correspond to the addressed cache block are cleared, or an alignment exception is generated.
- If the target address is designated as coherency-required, and the cache block exists in the data cache (of any processor), it is kept coherent in those caches.

**dcbst**

The dcbst (data cache block store) instruction allows software to verify that the latest version of the target effective address resides in main memory. During execution, the dcbst instruction is treated as a load operation. The operation of the dcbst instruction depends on the coherency requirements of the target address.

- If coherency is required and the target address exists in the data cache of any processor and has been modified, the data is written to main memory.
- If coherency is not required and the target address exists in the data cache of the executing processor and has been modified, the data is written to main memory.

**dcbf**

The operation of the dcbf (data cache block flush) instruction depends on the coherency requirements of the target address.
If coherency is required and the target address exists in the data cache of any processor and has been modified, the data is written to main memory.

If coherency is not required and the target address exists in the data cache of the executing processor and has been modified, the data is written to main memory.

**icbi**

The icbi (instruction cache block invalidate) instruction is one of two instruction cache management instructions. In general, instruction caches are not required to be consistent with data caches, memory, or I/O operations.

Like other cache management instructions, the icbi instruction depends on the coherency requirements of the target address.

- If coherency is required and the target address exists in the instruction cache of any processor, the cache block is made invalid for all processors. Subsequent references to the target address will cause the cache block to be refetched from main memory.

- If coherency is not required and the target address exists in the instruction cache of the executing processor, the cache block is made invalid in the executing processor so that subsequent references cause the cache block to be refetched from main memory.

The icbi instruction is provided for use on Harvard architecture processors, which have separate instruction and data caches. The target effective address is translated and checked for memory protection violations as if the access were a load operation.

**isync**

The isync (instruction cache synchronize) instruction waits for all previous instructions to complete and then discards any prefetched instructions, causing subsequent instructions to be fetched from memory. These instructions will execute in the context established by the previous instructions. Context is defined as a processor’s privilege-level, address translation, and memory
protection characteristics. The isync instruction has no effect on other processors (if they exist) or on their caches.

**dcbi**

The function of the supervisor-level dcbi (data cache block invalidate) instruction depends on the memory mode associated with the cache block containing the byte addressed by the effective address. In general, dcbi will cause the cache block containing the byte addressed by the effective address to be invalidated and marked as unusable by the processor. Subsequently, the cache block must be reloaded from main memory.

**Summary**

While cache management is predominantly the domain of operating system software, there are times when application software can benefit from its deployment. This fact is exemplified by the existence of both user- and supervisor-level cache management instructions. In particular, software that is able to characterize its own memory usage requirements may be able to reduce the frequency of on-chip cache misses and, therefore, to reduce the overhead of accessing main memory.
The comprehensive approach would have been to title this chapter *Exceptions and/or Interrupts*. Nowhere else is the terminology contention between the two architectures more stark. Exceptions and interrupts on the PowerPC family of microprocessors are roughly equivalent to the x86 concept of exceptions and interrupts. The similarities, differences, and associated terminology will be made clear in the following pages.

Let’s examine how each camp defines an exception and an interrupt by summarizing the PowerPC user documentation and the *Intel i486 Programmer’s Reference Manual*.

- **Exception**
  
  *PowerPC*: An error or unusual condition arising in the execution of instructions. PowerPC exceptions may also be generated as a result of external signals. In general, an exception is any event that causes the normal instruction sequence to be abruptly changed. Handling a PowerPC exception causes a transition to a supervisor-level privilege state. PowerPC exceptions can be either *synchronous* or *asynchronous*. 

---

Electra: Begone; there is no power to help in thee.  
Chrysothemis: Not so; but in thee, no mind to learn.  
— Electra by Sophocles
**Intel x86**: An unusual condition that is detected by the processor as a consequence of executing instructions. These include programmed exceptions (software interrupts) that are generated by executing the INT \( n \) instruction and processor-detected error conditions. Intel x86 exceptions are strictly *synchronous* events.

**Interrupt**

**PowerPC**: On PowerPC processors, *interrupts are subsumed by exceptions*. That is, interrupts are just a special case of exceptions.

**Intel x86**: Interrupts occur at random times during the execution of a program, in response to signals from hardware. Interrupts are used to handle events external to the processor. Intel x86 interrupts are *asynchronous* events. This distinction will be useful during subsequent discussions.

Clearly, the two sets of definitions aren’t interchangeable. To avoid confusion, you’ll need to set aside Intel conventions and embrace those of the PowerPC. Other terms associated with interrupts and exceptions will reinforce the difference. The following section defines the terms and conventions required to discuss exceptions and interrupts on PowerPC processors.

**Definitions**

An exception or interrupt is simply a vehicle by which an event can force a transfer of control to a specific software routine known as a *handler*. This transfer of control may be provoked by either an internal or external event that requires the processor’s immediate attention. Unlike the *call* mechanism, processed within the linear instruction stream, a handler can be invoked omnipotently by the processor during the otherwise normal execution of software.

The term *exception* refers to an error, unusual condition, or external signal. The exception may or may not cause a transfer of execution or other identifiable result. Whether or not an exception handler is called or status bits are set depends on the configuration of the processor at the time of the exception. In particular, some exceptions can be enabled and disabled by supervisor-level software.

This definition applies to both internal and external events. To distinguish between internal and external sources of exceptions, the PowerPC architecture defines two terms: *synchronous* and *asynchronous*. Synchronous exceptions are caused by internal events such as the execution of an
Asynchronous exceptions are caused by external events such as pushing the reset button on your PowerPC computer system. Additionally, all exceptions are classified as precise or imprecise. Each of these exception categories is described in detail in this chapter.

The term interrupt is a synonym for an asynchronous exception, mentioned previously. It refers specifically to an external event such as pushing the reset button on your PowerPC computer system. You'll see this term used infrequently in this chapter.

Of course, no rule is without exceptions. For example, after an exception handler handles an exception, it returns to the original code by executing the rfi (return from interrupt) instruction! This contention arises out of the terminology differences between the user documentation and the PowerPC architecture. Such sources of potential confusion will be noted where they occur.

**Asynchronous and Synchronous Exceptions**

A synchronous exception is generated by the execution of a particular instruction or instruction sequence. For example, the sc (system call) instruction generates a system call exception when it is executed. All synchronous exception events are deterministic. That is, upon entry to an exception handler, the cause of the exception can be determined using the information provided by the processor.

An asynchronous exception event has no relationship to the instructions that are being executed by the processor. For example, pushing the system reset button on your computer system forces the processor to stop whatever code sequence it was executing to service the exception. There is no correlation between the exception and the code that was being executed. There are four asynchronous exceptions: system reset, machine check, decremeter, and external interrupt. All asynchronous exceptions are nondeterministic. That is, the exact cause of the exception cannot be determined by software — we may understand someone hit the reset switch, but software has no means of determining the cause of the exception within the exception handler.
Precise and Imprecise Exceptions

Exceptions are classified as *precise* or *imprecise* depending on the processor’s ability to determine both the instruction that caused the exception and the proper place to resume execution after handling the exception. There is only one imprecise exception (the imprecise mode floating-point enabled exception) — everything else is precise.

Precise exceptions provide all the information needed to determine the instruction that caused the exception and where to resume execution. When precise exceptions occur, the address at which execution is to resume is placed in the SRR0 register. The instruction pointed to by SRR0 is unique for two reasons: All instructions prior to the one identified by SRR0 are guaranteed to have completed. And all instructions after the instruction pointed to by SRR0 are guaranteed not to have begun execution. The exact status of the instruction pointed to by the SRR0 depends on the type of exception.

The imprecise mode floating-point enabled exception is implemented as a program exception (vector 0x00700). There are two FP enabled exception modes: nonrecoverable and recoverable. The mode of the floating-point enabled program exception is determined by the setting of the MSR[FE0,FE1] bits, shown in Figure 10-1.

A nonrecoverable imprecise exception causes the processor to execute the associated exception handler, but the processor cannot reliably determine the event that triggered the exception or when the event occurred. It is this arbitrary aspect that makes the exception event nonrecoverable and imprecise.

A recoverable imprecise exception makes enough information available to the processor exception handling that the processor can identify the instruction that caused the exception. Furthermore, no incorrect values resulting from this exception will have been used in instructions that followed the offending instruction.

Exceptions and Privilege Level

When the i486 processor operates in protected mode, the privilege level associated with an exception handler is arbitrary, depending on the setting of a descriptor within the interrupt descriptor table (IDT) and associated code segment selectors. PowerPC processors, in contrast, switch to supervisor mode at the beginning of exception processing. This is an important
point: There is an inherent privilege level associated with exception handling on PowerPC processors.

Figure 10-1
Nonrecoverable and recoverable modes on the machine state register.

Context Synchronizing Exceptions

Most recoverable exceptions, where execution is resumed from the original linear instruction stream, are context synchronizing. Some exceptions that are recoverable, such as the trace exception, are not context synchronizing. The concepts of context and context synchronization were first introduced in Chapter 6, "The PowerPC Instruction Set."
When a context synchronizing exception occurs, the processor performs the following steps to synchronize its state before the instruction handler is invoked by the exception mechanism:

- All currently issued instructions complete to the extent that they cannot generate any subsequent exceptions.
- All currently issued instructions complete in the context in which they were issued.
- All instructions that are issued after exception handling are executed in the context established by the exception handler. In order to properly return to the original code stream, the exception handler must restore the original context.

**Exception Categories and Priorities**

There are a number of categories of exceptions on PowerPC processors and each has an associated priority. The PowerPC exception priorities are necessary to define the exception-handling procedure when multiple exceptions occur simultaneously. For example, if an operating system is handling a page fault and the user hits the reset switch, we expect (and desire) the processor to jump immediately to the reset vector and perform any last-minute (or last-microsecond) shutdown procedures.

The following list summarizes the general categories of PowerPC exceptions. Following the list, Table 10-1 details the priority of each exception category.

- **Asynchronous and Nonmaskable Exceptions**
  Generated by a machine check or system reset. Asynchronous and nonmaskable exceptions represent the highest-priority exception category. Both sources of these exception events are external to the processor and instruction execution.

- **Asynchronous and Maskable Exceptions**
  Generated by an external interrupt or a decremener interrupt. As shown in Table 10-1, these exceptions represent the bottom of the priority scale.

- **Synchronous and Precise**
  Generated by instructions. Excludes floating-point imprecise exceptions. SRR0 points either to the instruction that caused the exception or to the instruction following, depending on the exception type. Instruction execution is synchronized up to the instruction pointed to by SRR0, and no instructions past the SRR0-designated instruction have begun execution.
### Table 10-1

**PowerPC Exception Priorities**

<table>
<thead>
<tr>
<th>Exception Category</th>
<th>Priority</th>
<th>Exception Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous / nonmaskable</td>
<td>1 (highest)</td>
<td>System Reset&lt;br&gt;System reset is the highest-priority exception; all other exceptions are ignored when this exception condition exists. The system reset exception corresponds to vector offset 0x00100.</td>
</tr>
<tr>
<td>Asynchronous / nonmaskable</td>
<td>2</td>
<td>Machine Check&lt;br&gt;If a machine check exception occurs, all exceptions of lower priority are ignored. The machine check exception corresponds to vector offset 0x00200.</td>
</tr>
<tr>
<td>Synchronous / precise</td>
<td>3</td>
<td>Exceptions Caused by Instruction Execution&lt;br&gt;All instructions in the linear code stream that come before the instruction that caused the exception are completed. If any of these instructions cause an exception, that exception is handled first.</td>
</tr>
<tr>
<td>Imprecise</td>
<td>4</td>
<td>Program Imprecise Floating-Point Mode Enabled Exception&lt;br&gt;This exception is maskable using the MSR[FE0,FE1] bits and is handled as a program exception (vector offset 0x00700).</td>
</tr>
<tr>
<td>Asynchronous / maskable</td>
<td>5</td>
<td>External Interrupt&lt;br&gt;This exception is caused by an external interrupt condition and can be masked using the MSR[EE] bit (MSR[EE]=1 is enabled). If an external interrupt is pending at the point that external interrupts are enabled (by setting MSR[EE]=1), the exception will be generated immediately.</td>
</tr>
<tr>
<td>Asynchronous / maskable</td>
<td>6 (lowest)</td>
<td>Decrementer&lt;br&gt;The decrementer exception is the lowest-priority exception and is generated by the processor only when no other exception conditions exist. This exception condition can be masked using the MSR[EE] bit (MSR[EE]=1 is enabled). If a decrementer exception is pending at the point that external interrupts are enabled (by setting MSR[EE]=1), the exception will be generated immediately.</td>
</tr>
</tbody>
</table>
Synchronous and Imprecise

Generated by instructions. The floating-point imprecise mode exception is the only synchronous/imprecise exception defined by the PowerPC architecture. This exception is treated as a program exception (vector offset 0x00700).

**Programming Point: One Instruction, Multiple Exceptions**

Synchronous/precise exceptions are an important class of PowerPC exceptions. They result from the execution of instructions — something we as programmers control directly. But a single instruction, in the worst case, can generate multiple exceptions. When this happens, it's crucial to understand the order in which the exceptions are generated and, therefore, the order in which they're handled. The following list shows the order in which exceptions are generated for several types of operations. Note that the *trace exception* (debugger single-step mechanism) is always the last exception to be generated.

Exceptions caused by the execution of instructions (synchronous/precise) are always reported serially in the order shown in the table. In fact, all synchronous exceptions on PowerPC processors are reported in a serial manner. The inability to generate more than one exception simultaneously means that the exception must be prioritized. Asynchronous exceptions, which can occur at any time, are the highest-priority exceptions and can interrupt the handling of other exception conditions.

<table>
<thead>
<tr>
<th>Operation Type</th>
<th>Exceptions Generated and Their Priority (highest to lowest)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer loads and stores</td>
<td>Alignment exception</td>
</tr>
<tr>
<td></td>
<td>DSI exception</td>
</tr>
<tr>
<td></td>
<td>Trace exception</td>
</tr>
<tr>
<td>Floating-point loads and stores</td>
<td>FP unavailable exception</td>
</tr>
<tr>
<td></td>
<td>Alignment exception</td>
</tr>
<tr>
<td></td>
<td>SI exception</td>
</tr>
<tr>
<td></td>
<td>Trace exception</td>
</tr>
<tr>
<td>Return from interrupt and move to MSR instructions</td>
<td>Privilege-level exception</td>
</tr>
<tr>
<td></td>
<td>Precise-mode FP exception</td>
</tr>
<tr>
<td></td>
<td>Trace exception</td>
</tr>
</tbody>
</table>
Exceptions and Vectors

The i386 and later members of the x86 family have a supervisor-level register called the interrupt descriptor table register (IDTR). The IDTR points to a table of interrupt vectors in memory called the interrupt descriptor table (IDT). The IDTR is needed because the IDT can be stored at an arbitrary location in physical memory.

In contrast, PowerPC vector tables can exist at one of only two fixed locations in memory. The PowerPC vector table starts at either 0x00000000 or 0xffff0000 (for 32-bit implementations) based on the setting of the MSR[IP] bit. All PowerPC vector offsets are relative to this base value.

Exception vectors on x86 processors are packed together in 8-byte interrupt descriptor table entries. The offsets of adjacent PowerPC vectors are typically separated by 0x100 bytes. The difference is a matter of convenience and efficiency. The IDT is a memory-based table; therefore, the processor is already having to fetch (read) an address from memory and jump to it to handle an exception. Adding room for exception code wouldn’t save you the initial memory access to the IDT. However, the PowerPC vector locations are statically defined — the processor simply knows where to jump and no memory access is required.

In effect, the PowerPC system provides 0x100 bytes in which to write an exception handler that is absolutely free of jumps or calls. When writing a handler, efficiency is very important — jumps and calls waste precious clock cycles. If the exception handler fits within the 0x100 bytes of vector space, no additional memory is required. However, if an exception handler does exceed 0x100 bytes, other routines and memory may be used as appropriate.

Table 10-2 summarizes all PowerPC exceptions and associated vector offsets that are defined by the PowerPC architecture. In general, exceptions that are defined by the PowerPC architecture are equivalent across all processor implementations. However, to allow for a variety of PowerPC processor implementations, there are exceptions that are hardware implementation dependent.

The PowerPC architecture defines the exceptions listed in Table 10-2. Unless specifically noted above, all PowerPC processors implement the architected exceptions. PowerPC processors are used in a variety of applications, from desktop computers to embedded controllers. In order to accommodate the range of uses, each processor may implement exception conditions not defined by the PowerPC architecture specification, as shown in Table 10-3.
### Table 10-2
Summary of PowerPC Exceptions and Vectors

<table>
<thead>
<tr>
<th>Exception Name</th>
<th>Vector Offset</th>
<th>Cause/Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>0x00000</td>
<td>This vector is reserved.</td>
</tr>
<tr>
<td>System Reset</td>
<td>0x00100</td>
<td>Generated by the assertion of the system reset signal. The physical means of asserting either signal is system implementation dependent. (All PowerPC implementations)</td>
</tr>
<tr>
<td>Machine Check</td>
<td>0x00200</td>
<td>Generated when signals such as #TEA or #MCP are asserted. If MSR[ME] is cleared when a machine check exception is generated, the processor enters the check stop state. Note that MSR[ME] is cleared automatically when any exception is taken. The signals that generate machine check exceptions are implementation dependent. (All PowerPC implementations)</td>
</tr>
<tr>
<td>DSI (Data Access Exception)</td>
<td>0x00300</td>
<td>Generated when a data memory access cannot be performed due to conditions described in the ISI exception section in this chapter. Instructions that generate a DSI exception include load/store, memory control, and cache control instructions. (All PowerPC implementations)</td>
</tr>
<tr>
<td>ISI (Instruction Access Exception)</td>
<td>0x00400</td>
<td>Generated when an instruction fetch cannot be performed due to conditions described in the ISI exception section in this chapter. (All PowerPC implementations)</td>
</tr>
<tr>
<td>External Interrupt</td>
<td>0x00500</td>
<td>Generated when the #INT signal is asserted. As soon as this signal is detected by the processor, no further instructions are dispatched. The exception handler is invoked when all currently dispatched instructions have completed. If an exception is generated by one of the dispatched instructions, it is handled first. (All PowerPC implementations)</td>
</tr>
<tr>
<td>Alignment</td>
<td>0x00600</td>
<td>Generated when the processor cannot perform a memory access due to conditions described in the alignment exception section in this chapter. (All PowerPC implementations)</td>
</tr>
</tbody>
</table>
### Table 10-2
Summary of PowerPC Exceptions and Vectors (Continued)

<table>
<thead>
<tr>
<th>Exception Name</th>
<th>Vector Offset</th>
<th>Cause/Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>0x00700</td>
<td>Generated for one of the following reasons:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Floating-point enabled exception</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Attempted execution of an illegal instruction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Execution privilege violation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Execution of a trap instruction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(All PowerPC implementations)</td>
</tr>
<tr>
<td>Floating-Point Unavailable</td>
<td>0x00800</td>
<td>601/603 — A floating-point unavailable exception is generated when MSR[FP]=0 (floating-point available bit is clear) and an attempt is made to execute a floating-point instruction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>604/620 — Reserved (Not implemented)</td>
</tr>
<tr>
<td>Decrementer</td>
<td>0x00900</td>
<td>Generated when the most significant bit of the decrementer register changes from 0 to 1 and the MSR[EE] bit is set. If MSR[EE]=0 (exception disabled) and a decrementer exception is pending, it will be taken as soon as MSR[EE] is set. (All PowerPC implementations)</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x00b00</td>
<td>Reserved on all implementations.</td>
</tr>
<tr>
<td>System Call</td>
<td>0x00c00</td>
<td>Generated when the sc (system call) instruction is executed. (All PowerPC implementations)</td>
</tr>
<tr>
<td>Trace</td>
<td>0x00d00</td>
<td>601 — Reserved. The 601 does not generate a trace exception.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>603/604/620 — Generates a trace exception in two cases: When MSR[SE]=1 and any instruction other than rfi (return from interrupt) is completed, and when MSR[BE]=1 and the currently completing instruction is a branch instruction.</td>
</tr>
<tr>
<td>Floating-Point Assist</td>
<td>0x00e00</td>
<td>Reserved on all implementations.</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x00e10 - 0x00eff</td>
<td>This area of the exception vector table is reserved by the PowerPC architecture specification.</td>
</tr>
</tbody>
</table>

The exceptions listed in Table 10-3 are considered implementation dependent (not defined by the PowerPC architecture specification). As a result, each exception definition will necessarily vary between processor implementations as noted.
<table>
<thead>
<tr>
<th>Exception Name</th>
<th>Vector Offset</th>
<th>Cause/Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct-Store Exception (I/O controller interface error)</td>
<td>0x00a00</td>
<td>Generated when an access to a direct-store (I/O controller interface) segment fails. (601 only)</td>
</tr>
<tr>
<td>Performance Monitoring Exception</td>
<td>0x00f00</td>
<td>601 — Reserved. 603 — Reserved. 604/620 — The performance monitoring exception is unique to the PowerPC 604 and 620. If MSR[EE]=1 and the performance monitor exception is enabled in MMICRO, a performance monitoring exception will be generated for performance counter-negative conditions as described in Chapter 12, &quot;Techniques and Tricks.&quot;</td>
</tr>
<tr>
<td>Instruction TLB Miss</td>
<td>0x01000</td>
<td>601/604/620 — Reserved 603 — Generated when an effective address for an instruction fetch cannot be translated by the instruction translation lookaside buffer (ITLB).</td>
</tr>
<tr>
<td>Data TLB Miss on Load</td>
<td>0x01100</td>
<td>601/604/620 — Reserved 603 — Generated when an effective address for a data load operation cannot be translated by the data translation lookaside buffer (DTLB).</td>
</tr>
<tr>
<td>Data TLB Miss on Store</td>
<td>0x01200</td>
<td>601/604/620 — Reserved 603 — Generated when an effective address for a data store operation cannot be translated by the DTLB. Additionally, this exception can be generated when a DTLB hit occurs and the change bit in the PTE is set due to a data store operation.</td>
</tr>
<tr>
<td>Instruction Address Breakpoint</td>
<td>0x01300</td>
<td>601 — Reserved 603/604/620 — Generated when the address in the instruction address breakpoint register (IABR) matches the address of the next instruction to complete in the completion unit. The IABR enable bit (IABR[30]) must be set.</td>
</tr>
<tr>
<td>System Management Interrupt</td>
<td>0x01400</td>
<td>601 — Reserved 603/604/620 — Generated when MSR[EE] is set and the #SMI input signal is asserted.</td>
</tr>
</tbody>
</table>
### Table 10-3
Implementation-Specific Exception Vectors (Continued)

<table>
<thead>
<tr>
<th>Exception Name</th>
<th>Vector Offset</th>
<th>Cause/Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>0x01500 - 0x01fff</td>
<td>This area is reserved on all implementations.</td>
</tr>
<tr>
<td>Run Mode/Trace Exception</td>
<td>0x02000</td>
<td>601 — Generated depending on the settings of the H1D1 and machine state (MSR[SE]) registers. 603/604/620 — Reserved</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x02100 - 0x02fff</td>
<td>This area is reserved on all implementations.</td>
</tr>
<tr>
<td>End of Vector Locations</td>
<td>0x02fff</td>
<td>This is the last offset that can be used for exception vectors, as specified by the PowerPC architecture.</td>
</tr>
</tbody>
</table>

### Exception Descriptions

Having summarized each exception, it's time to get down to the details. Before we begin, there are a few aspects of exception processing that every programmer should know. First, on entry to an exception handler, address translation is disabled. This means that before an exception handler can access memory using the virtual memory set up by an operating system, the handler is responsible for explicitly enabling address translation. As a result, coding exceptions can be tricky when using memory management features. By contrast, during exception handling on x86 processors, interrupts are disabled and address translation remains enabled.

Secondly, on entry to an exception handler, the processor's on-chip caches are enabled. The initial state of the caches (before the exception condition) is arbitrary. This is true (at least) for the PowerPC 601, 603, and 604 processors.

### Tracing an Exception

Now, before we take a closer look at each of the PowerPC exceptions, it's helpful to understand the basic exception mechanism. The easiest way to do so is to trace through a generic exception occurring on a generic processor.

First, the exception is generated by some arbitrary operation. To cover the most common case, we'll assume that our exception is of the synchronous/precise variety. The exception mechanism saves the state of the processor to the save/restore registers (SRR0/SRR1). In general, SRR0 is set to point to the instruction where execution will resume after exception...
processing (the return address). The SRR1 register typically holds exception-specific information and saves the state of the MSR; this operation preserves the context of the excepting code for later restoration. Other registers, such as the DSISR, are occasionally used for problem determination.

The 601, 604, and 620 copy sufficient bits from the MSR to restore the context on exit from the handler. The 603 uses the SRR1[0-15] bits to describe TLB miss conditions. In particular, the following bits are set:

- SRR1[0-3] are set to the value of CR0.
- SRR1[5-9] are set to MSR[5-9].
- SRR1[13] is set for an ITLB (instruction translation lookaside buffer) miss and cleared for a DTLL (data translation lookaside buffer) miss.
- SRR1[14] is set to indicate TLB set 1 and cleared to indicate TLB set 0.
- SRR1[15] is set to indicate a store miss and cleared to indicate a load miss.
- SRR1[16-31] are copied from the corresponding bits of the MSR.

Upon entering the exception handler, the registers that describe the exception condition and machine state should be saved immediately. This reduces the risk of losing the register contents if another exception is taken. For example, assume that a higher-priority exception is taken and subsequently overwrites the MSR, SRR0, and SRR1 contents associated with the first exception. After handling the new exception, the first exception handler no longer has sufficient information to resume execution after exception processing.

The MSR is loaded with new values dependent upon the exception type. Note that for all exception types, both instruction and data address translation are disabled at the start of exception processing (MSR[IR, DR]=0). Additionally, on-chip caches are enabled at the start of exception processing (even if they were off before the exception occurred). In general, MSR[13-31] are set as shown in Figure 10-2. With the context saved and the new MSR context set up, the exception handler is free to handle the exception as appropriate.

<table>
<thead>
<tr>
<th>POW</th>
<th>ILE</th>
<th>EE</th>
<th>PR</th>
<th>FP</th>
<th>ME</th>
<th>SE</th>
<th>BE</th>
<th>IP</th>
<th>IR</th>
<th>DR</th>
<th>PM</th>
<th>RI</th>
<th>LE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>n/a</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 10-2

The machine state register (MSR) at the start of exception processing.

Note that the value of MSR[ILE] is set to MSR[ILE] at the start of exception processing.
Finally, the exception handler executes the rfi (return from interrupt/exception) instruction. The rfi instruction is context synchronizing. Furthermore, an exception caused by any instruction in the handler must be handled before the rfi is executed. If appropriate (depending on exception type), the rfi instruction restores the value of the MSR from the SRR1 register. Having reestablished the context of the preexception code, rfi ensures that all subsequent instructions execute in that reestablished context.

Some exceptions exist across all processors, some are hardware implementation specific. The following sections describe each of the exceptions for the various PowerPC implementations that are defined by the PowerPC architecture.

**System Reset Exception (0x00100)**

The system reset exception is an asynchronous/nonmaskable exception generated by the assertion of the #SRESET (soft reset) or #HRESET (hard reset) signal. However, the physical means of asserting either signal is system implementation dependent. System reset is the highest-priority exception. This exception is valid on the PowerPC 601, 603, 604 and 620.

The system reset exception causes execution to be immediately transferred to the system reset vector. The following conditions exist at this time:

- SRR0 points to the next instruction to be executed in the instruction stream that was executing before the exception.
- SRR1 is loaded with bits from the MSR; the bits occupy corresponding positions.
- If execution cannot be resumed due to the current processor state, MSR[RI] and SRR1[62] are cleared to 0.

**Machine Check (0x00200)**

The asynchronous/nonmaskable machine check exception is the second-highest-priority exception. The causes of a machine check exception are both system and processor implementation dependent. However, the following conditions will always trigger a machine check: a parity error, a bus error (#TEA signal), or the assertion of the machine check signal (#MCP). This exception is valid on the PowerPC 601, 603, 604, and 620.

A machine check exception is taken only if enabled (MSR[ME]=1). The following table summarizes the machine check enabling bits for the 603,
604, and 620 processors. The PowerPC 601’s checkstop enable bits are
unique compared to the other PowerPC processors. The 601’s HID0 register
and each checkstop bit definition are described in Chapter 4, “The
PowerPC Programming Model.”

Table 10-4
Machine Check Enables for the PowerPC 603 and 604 Processors

<table>
<thead>
<tr>
<th>Register[Bit]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HID0[0]</td>
<td>Setting this bit enables the machine check input pin.</td>
</tr>
<tr>
<td>HID0[1]</td>
<td>Enable cache parity checking.</td>
</tr>
<tr>
<td>HID0[2]</td>
<td>Enable machine check on address bus parity error.</td>
</tr>
<tr>
<td>HID0[3]</td>
<td>Enable machine check on data bus parity error.</td>
</tr>
</tbody>
</table>

When both the MSR[ME] and one of the bits listed in Table 10-4 are set, machine check exceptions are recognized and handled. If a particular machine check exception occurs that is not enabled, the processor enters checkstop state, all instruction execution is suspended, and the processor must be restarted.

Data Access Exception (0x00300)
The data access exception (DSI) is a synchronous/precise exception generated when an access to data memory cannot be performed. This exception is valid on the PowerPC 601, 603, 604, and 620. When a DSI exception occurs, the following conditions will be true:

- The SRR0 register points to the instruction that caused the DSI exception.
- The SRR1 register is loaded with bits from the MSR; the bits occupy corresponding positions.
- The DSISR is set to allow software to determine the cause of the exception. The DSISR register is described in Chapter 3, “Of Eggs and Endians.”
- The DAR (data address register) contains the effective address used in the data memory access that caused the exception. Where the SRR0 register pointed to the exception-causing instruction, the DAR points to the effective address that was used in that operation.
The DSI exception can be generated for any reason corresponding to one of the bits of the DSISR. These bits are described in Table 10-5. In order to fully define the condition that caused the exception, more than one bit in the DSISR may be set. For example, if a store operation violated DBAT memory protection, DSISR[4] and DSISR[6] would both be set to one.

### Table 10-5

**DSISR Bit Definitions**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Fault Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A load/store instruction caused the fault.</td>
</tr>
<tr>
<td>1</td>
<td>Address translation failed for the access. Address translation failure for the DSI exception may mean not finding the address in one of the hash tables or the address was not mapped by a DBAT register.</td>
</tr>
<tr>
<td>2</td>
<td>Always cleared.</td>
</tr>
<tr>
<td>3</td>
<td>Always cleared.</td>
</tr>
<tr>
<td>4</td>
<td>A memory access was not permitted by page protection or the DBAT mapping. This differs from the source reported by bit 1 in that address translation exists, but the access failed due to memory protection mechanisms.</td>
</tr>
<tr>
<td>5</td>
<td>An eciwx, ecowx, lwarx/ldarx, or stwcx./stdcx. instruction was attempted to a direct-store segment. Bit 5 may also be set if the lwarx/ldarx or stwcx./stdcx. instruction is used with memory marked as write-through.</td>
</tr>
<tr>
<td>6</td>
<td>Caused by a store operation; bit 6 is cleared if the exception was caused by a load operation.</td>
</tr>
<tr>
<td>7</td>
<td>Always cleared.</td>
</tr>
<tr>
<td>8</td>
<td>Always cleared.</td>
</tr>
<tr>
<td>9</td>
<td>A data address breakpoint register (DABR) match occurred. This bit is the data address equivalent of the instruction address breakpoint exception [offset Oxf1300].</td>
</tr>
<tr>
<td>10</td>
<td>Set if the segment table search fails to find a translation for the address. Bit 10 is always cleared on the 601, 603, and 604 and set only on 64-bit implementations.</td>
</tr>
<tr>
<td>11</td>
<td>Set if the instruction that caused the exception was an eciwx or ecowx and the external access register’s enable bit is set [EAR[E]=1].</td>
</tr>
<tr>
<td>12-31</td>
<td>Always cleared.</td>
</tr>
</tbody>
</table>
Instruction Access Exception (0x00400)

The instruction access exception (ISI) is a synchronous/precise exception that is generated when an instruction fetch cannot be performed. This exception is valid on the PowerPC 601, 603, 604, and 620. Similar to the DSI exception, an ISI exception is generated for any of the following reasons:

- The effective address for the instruction fetch cannot be translated by the paging or IBAT mechanisms. If the exception is generated due to a failure of the paging mechanism, this exception is equivalent to a page fault.
- The effective address for the instruction fetch corresponds to a direct-store segment.
- The instruction fetch operation violates memory protection set up by either the paging or IBAT mechanisms.

When an ISI exception occurs, the following conditions will be true:

- SRR0 points to the next instruction to execute in the instruction stream that was executing before the exception.
- The SRR1 register is set to allow software to determine the cause of the exception. The SRR1 bit values are summarized in Table 10-6.

### Table 10-6

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-32</td>
<td>Always cleared.</td>
</tr>
<tr>
<td>33</td>
<td>Set if the effective address could not be translated.</td>
</tr>
<tr>
<td>34</td>
<td>Always cleared.</td>
</tr>
<tr>
<td>35</td>
<td>Set if the fetch was to a direct-store segment.</td>
</tr>
<tr>
<td>36</td>
<td>Set if the fetch violated memory protection.</td>
</tr>
<tr>
<td>37-41</td>
<td>Always cleared.</td>
</tr>
<tr>
<td>42</td>
<td>Set if the page table search failed to find a translation for the effective address of the fetch.</td>
</tr>
<tr>
<td>43-47</td>
<td>Always cleared.</td>
</tr>
</tbody>
</table>
External Interrupt Exception (0x00500)

The external interrupt exception is generated by the assertion of the external interrupt (#INT) signal. This exception is valid on the PowerPC 601, 603, 604, and 620. The external interrupt exception is enabled by setting the MSR[EE] bit. If an external interrupt is pending when external interrupts are enabled (MSR[EE]=1), the exception occurs immediately. The external interrupt exception handler will be entered before the next instruction in the program stream that set MSR[EE] is executed.

External interrupt exceptions typically are generated by external peripheral devices. It is up to the external interrupt exception handler to determine the source of the external interrupt and service the peripheral device.

When an external interrupt exception occurs, the following conditions will be true:

- SRR0 points to the next instruction to execute in the original instruction stream that was executing before the exception.
- The SRR1 register is loaded with bits from the MSR; the bits occupy corresponding positions.

Alignment Exception (0x00600)

The alignment exception is a synchronous/precise exception that may be generated under a number of circumstances. Fundamentally, all alignment exceptions are caused by the inability to perform a memory access. But the alignment exception is perhaps the most complex PowerPC exception with respect to number of possible causal conditions. This exception is valid on the PowerPC 601, 603, 604, and 620.

The PowerPC architecture defines the misalignment situations that may cause an alignment exception. That is, the architecture allows for a future PowerPC processor to implement the necessary logic to handle every misalignment condition in hardware, without the need to generate an exception.

The alignment exception stores the processor state into SRR0 and SRR1, and uses the DSISR to determine the cause of the exception in the same fashion as the DSI exception. Recall that we define an aligned operand as having an address in memory on a boundary that is a multiple of the operand’s size. For example, an aligned word (4-byte) operand would have an effective address ending in 0, 4, 8, or 0xc.
The following general conditions can generate an alignment exception:

- The operand of a floating-point load/store is not word-aligned.
- The operand of an integer dword load/store operation is not word-aligned.
- The operand of a load/store to an address is not aligned on a word boundary while the processor is in little endian mode.
- The operand of any load/store crosses a protection boundary. A protection boundary is defined as the boundary between two areas of memory (protection domains) that are mapped by a BAT register pair, mapped by the PowerPC paging mechanism, or is designated as an I/O segment.

Alignment exceptions are generated for load/store operations that cross a protection boundary only when data address translation is enabled (MSR[DR]=1). That is, the concept of protection domains applies only when the PowerPC processor’s address translation mechanism is enabled. If data address translation is disabled (MSR[DR]=0), all memory protection facilities (BAT and paging protection) for data memory accesses are also disabled.

When an alignment exception occurs, the following conditions will be true:

- SRR0 points to the instruction that caused the exception. In the case of an alignment exception, SRR0 points precisely to the exception-causing instruction. Upon entry to other exception handlers, SRR0 may be set with less precision.
- The SRR1 register is loaded with bits from the MSR; the bits occupy corresponding positions.
- DSISR is set to enable software to determine the cause of the exception. The DSISR bit values are summarized in Table 10-7. Determining the cause of the exception is important to handlers that emulate the exception-causing instruction. Other bits in the DSISR are set to indicate the addressing mode used by the exception-causing instruction: register indirect with either index or immediate index mode.

Some pairs of instructions result in the same bit value being loaded into the DSISR. In particular, load or store instructions that use register indirect with index addressing can set the DSISR to the same value that would have resulted if the corresponding instruction used register indirect with immediate index addressing. Similarly, for load or store instructions that use
register indirect with immediate index addressing, DSISR can hold a value
that would have resulted from an instruction that uses register indirect
addressing.

### Table 10-7
The DSISR Reports the Source of an Alignment Exception

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits 0–14</td>
<td>Always cleared.</td>
</tr>
<tr>
<td>Bits 15–16</td>
<td>Cleared for instructions that use register indirect with immediate addressing. DSISR[15-16] are set to bits 29–30 of the exception-causing instruction when the instruction uses register indirect with index addressing.</td>
</tr>
<tr>
<td>Bit 17</td>
<td>Set to bit 25 of the exception-causing instruction for instructions that use register indirect with index addressing. Set to bit 5 of the exception-causing instruction for instructions that use register indirect with immediate addressing.</td>
</tr>
<tr>
<td>Bits 18–21</td>
<td>Set to bits 21–24 of the exception-causing instruction for instructions that use register indirect with index addressing. Set to bits 1–4 of the exception-causing instruction for instructions that use register indirect with immediate addressing.</td>
</tr>
<tr>
<td>Bits 22–26</td>
<td>Always set to bits 6–10 of the exception-causing instruction.</td>
</tr>
<tr>
<td>Bits 27–31</td>
<td>Always set to bits 11–15 of the instruction [rA] for update-form instructions or to any register number not in the range of registers loaded by a valid form instruction, for lmw, lswi, and lswx instructions. Otherwise undefined.</td>
</tr>
</tbody>
</table>

The instruction pairs that can generate the same DSISR values are shown in Table 10-8. The settings for DSISR bits 15–21 and the instructions that they indicate are shown in Table 10-9.

### Table 10-8
Instructions Reported Identically in the DSISR

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instruction</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbz/lbzx</td>
<td>lwzu/lwzux</td>
<td>stw/stwx</td>
</tr>
<tr>
<td>lwz/lwzx</td>
<td>sth/stx</td>
<td>lfd/lfdx</td>
</tr>
<tr>
<td>sib/sibux</td>
<td>lfs/lfsx</td>
<td>lha/lhax</td>
</tr>
<tr>
<td>std/stdx</td>
<td>stfd/stfdx</td>
<td>ldu/ldux</td>
</tr>
<tr>
<td>stdx</td>
<td>lhzul/hzux</td>
<td>stwu/stwux</td>
</tr>
<tr>
<td>lbdul/lbdux</td>
<td>ld/ldx</td>
<td>lfdul/lfdux</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 10-9
DSISR Settings Used in Exception Cause Determination

<table>
<thead>
<tr>
<th>DSISR[15-21]</th>
<th>Instruction</th>
<th>DSISR[15-21]</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0 0000</td>
<td>lwarx, lwz</td>
<td>01 1 0000</td>
<td>ldux</td>
</tr>
<tr>
<td>00 0 0001</td>
<td>ldarx</td>
<td>01 1 0010</td>
<td>stdux</td>
</tr>
<tr>
<td>00 0 0010</td>
<td>stw</td>
<td>01 1 0101</td>
<td>lwaux</td>
</tr>
<tr>
<td>00 0 0100</td>
<td>lhz</td>
<td>10 0 0010</td>
<td>stdwcx</td>
</tr>
<tr>
<td>00 0 0101</td>
<td>lha</td>
<td>10 0 0111</td>
<td>stdcx</td>
</tr>
<tr>
<td>00 0 0111</td>
<td>lmw</td>
<td>10 0 1010</td>
<td>stdwbrx</td>
</tr>
<tr>
<td>00 1 0000</td>
<td>lfs</td>
<td>10 0 1000</td>
<td>lhbrix</td>
</tr>
<tr>
<td>00 1 0010</td>
<td>ltd</td>
<td>10 1 0100</td>
<td>stdwx</td>
</tr>
<tr>
<td>00 1 0101</td>
<td>stfs</td>
<td>10 1 0110</td>
<td>ecowx</td>
</tr>
<tr>
<td>00 1 0111</td>
<td>stdf</td>
<td>10 1 1111</td>
<td>dcbz</td>
</tr>
<tr>
<td>00 1 1101</td>
<td>ldl, ldu, lwa</td>
<td>11 0 0000</td>
<td>lwzx</td>
</tr>
<tr>
<td>00 1 1111</td>
<td>std, stdu</td>
<td>11 0 0010</td>
<td>stdx</td>
</tr>
<tr>
<td>00 1 0000</td>
<td>lwzu</td>
<td>11 0 0100</td>
<td>stdux</td>
</tr>
<tr>
<td>00 1 0010</td>
<td>stwu</td>
<td>11 0 1001</td>
<td>stdx</td>
</tr>
<tr>
<td>00 1 0100</td>
<td>lhzu</td>
<td>11 0 0110</td>
<td>stdx</td>
</tr>
<tr>
<td>00 1 0101</td>
<td>lhou</td>
<td>11 0 1000</td>
<td>stdx</td>
</tr>
<tr>
<td>00 1 0110</td>
<td>sthu</td>
<td>11 0 1111</td>
<td>stdx</td>
</tr>
<tr>
<td>00 1 0111</td>
<td>stmw</td>
<td>11 0 1010</td>
<td>stdx</td>
</tr>
<tr>
<td>00 1 1000</td>
<td>lfsu</td>
<td>11 0 1011</td>
<td>stdx</td>
</tr>
<tr>
<td>00 1 1001</td>
<td>lfdx</td>
<td>11 0 1111</td>
<td>stdx</td>
</tr>
<tr>
<td>00 1 1010</td>
<td>stfsu</td>
<td>11 1 0000</td>
<td>stdx</td>
</tr>
<tr>
<td>00 1 1011</td>
<td>stdfu</td>
<td>11 1 0010</td>
<td>stdx</td>
</tr>
<tr>
<td>01 0 0000</td>
<td>ldx</td>
<td>11 1 0100</td>
<td>stdx</td>
</tr>
<tr>
<td>01 0 0010</td>
<td>stdx</td>
<td>11 1 0101</td>
<td>stdx</td>
</tr>
<tr>
<td>01 0 0101</td>
<td>lwaux</td>
<td>11 1 0110</td>
<td>stdx</td>
</tr>
<tr>
<td>01 0 1000</td>
<td>lswx</td>
<td>11 1 1000</td>
<td>stdx</td>
</tr>
<tr>
<td>01 0 1001</td>
<td>stswx</td>
<td>11 1 1001</td>
<td>stdx</td>
</tr>
<tr>
<td>01 0 1010</td>
<td>stswi</td>
<td>11 1 1010</td>
<td>stdx</td>
</tr>
<tr>
<td>01 0 1011</td>
<td>stswi</td>
<td>11 1 1011</td>
<td>stdx</td>
</tr>
</tbody>
</table>

Program Exception (0x00700)
The program exception is a synchronous/precise exception and can be generated for any reason corresponding to one of the bits of the SRR1 register. This exception is valid on the PowerPC 601, 603, 604, and 620.
When a program exception occurs, the following conditions will be true:

- The SRR0 register is set according to the following conditions:
  For all program exceptions other than floating-point imprecise mode exceptions (see MSR[FE0,FE1] in Figure 10-1), SRR0 contains the address of the instruction that caused the program exception.
  For floating-point imprecise mode program exceptions, SRR0 contains the address of an instruction that may be the exception-causing instruction. SRR0 may point to a subsequent instruction located an arbitrary number of instructions after the exception-causing instruction.
- If this exception is pending (FPSCR[FEX]=1) but disabled in the MSR (MSR[FE0,FE1]=0), the exception will occur when an instruction alters the setting of MSR[FE0,FE1].
- The SRR1 register is set so software can determine the cause of the exception. The SRR1 bit values are summarized in Table 10-10.

### Table 10-10

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits 0-10</td>
<td>Always cleared.</td>
</tr>
<tr>
<td>Bit 11</td>
<td>Set for an IEEE floating-point enabled program exception.</td>
</tr>
<tr>
<td>Bit 12</td>
<td>Set for an illegal instruction form program exception. Illegal instruction forms are discussed in Chapter 6, &quot;The PowerPC Instruction Set.&quot;</td>
</tr>
<tr>
<td>Bit 13</td>
<td>Set if the execution of an instruction violated privilege-level protection. A privilege-level violation occurs when software executing in user mode attempts to execute a supervisor-level instruction.</td>
</tr>
<tr>
<td>Bit 14</td>
<td>Set if a trap instruction was executed.</td>
</tr>
<tr>
<td>Bit 15</td>
<td>Set if SRRO does not contain the address of the exception-causing instruction (but rather a subsequent instruction). If bit 15 is clear, SRRO points to the exception-causing instruction.</td>
</tr>
<tr>
<td>Bits 16-31</td>
<td>Loaded with the corresponding bits from the MSR.</td>
</tr>
</tbody>
</table>
Floating-Point Unavailable Exception (0x00800)

The floating-point unavailable exception is generated when software attempts to execute any floating-point instruction and floating-point available is disabled in the MSR (MSR[FP]=0).

When a floating-point unavailable exception occurs, the following conditions will be true:

- SRR0 contains the address of the instruction that caused the exception.
- The SRR1 register is loaded with bits from the MSR; the bits occupy corresponding positions.

Decrementer Exception (0x00900)

A decrementer exception is generated when the count contained in the decrementer register passes through zero (when bit 0 changes from 0 to 1). This exception is valid on the PowerPC 601, 603, 604, and 620.

As described in Chapter 4, “The PowerPC Programming Model,” the decrementer register is a countdown register that can generate periodic interrupts for low-resolution timing operations. The rate at which the decrementer counts down is directly related to the processor clock speed. As such, the count rate is system implementation dependent.

The decrementer exception is enabled and disabled by setting and clearing the MSR[EE] bit. If a decrementer exception is pending when MSR[EE] is set, the exception occurs immediately. The decrementer exception handler will be entered before the next instruction in the program that set MSR[EE] is executed.

When a decrementer exception occurs, the following conditions will be true:

- SRR0 points to the next instruction to execute in the original instruction stream that was executing before the exception.
- The SRR1 register is loaded with bits from the MSR; the bits occupy corresponding positions.

Direct-Store Exception (0x00a00)

The direct-store (I/O controller interface error) exception is defined only on the PowerPC 601. This exception is generated when a load or store
operation to a direct-store (I/O controller interface) segment cannot be performed. Other PowerPC processors provide the functionality of the direct-store exception via the DSI exception.

When a direct-store exception occurs, the following conditions will be true:

- SRR0 points to the next instruction to execute in the original instruction stream that was executing before the exception.
- The SRR1 register is loaded with bits from the MSR; the bits occupy corresponding positions.
- The DAR (data address register) points to the first byte of the operand that caused the exception.

Direct-store exceptions on the 601 differ from typical memory access exceptions. The update form of both loads and stores cause the target register to be updated before the exception handler is invoked. The lwarx, stwx, and lscbx instructions cause DSI exceptions, as opposed to direct-store exceptions. Finally, floating-point loads and stores to direct-store segments are not supported on the 601; attempting such an operation generates an alignment exception.

**System Call Exception (0x00c00)**

A system call exception is generated by the execution of an sc (system call) instruction. This synchronous/precise exception is context synchronizing (discussed earlier in this chapter). This exception is valid on the PowerPC 601, 603, 604, and 620.

When a system call exception occurs, the following conditions will be true:

- SRR0 points to the instruction following the system call instruction.
- The SRR1 register is loaded with bits from the MSR; the bits occupy corresponding positions.

**Trace Exception (0x00d00)**

The trace exception is an optional exception on PowerPC processors and its implementation is processor dependent. The PowerPC 601 implements a
trace/run mode exception at vector offset 0x02000; this is not discussed in this section. The 603, 604, and 620 each implement the trace exception described here at vector offset 0x00d00.

To enable the trace exception, MSR[SE] must be set (MSR[SE]=1). Setting MSR[SE] has the effect of serializing instruction execution — only one instruction is executed at a time. Of course, single-stepping through code is useful only when you can observe results as they would actually occur when running the code normally. Because of this need, trace exceptions have two fundamental requirements.

First, the trace exception must not change the context of the code that is being single-stepped. This implies that the trace exception cannot be context synchronizing; a trace exception is not generated for instructions causing other exceptions or context synchronization such as the sc, rfi, or trap instructions.

Second, the processor must be able to guarantee that the results of each instruction are reflected in the architectural registers when the trace exception handler gets control. Serializing instruction execution, as noted above, ensures that the trace exception handler sees only the effects of atomic instruction execution.

The MSR[SE] bit is cleared (disabling trace exceptions) when a trace exception occurs. Otherwise, the processor would attempt to single-step the trace exception handler, resulting in an infinite loop. The rfi instruction restores the setting of the MSR from the contents of the SRR1 register, typically reenabling the trace exception.

Note that the PowerPC 603 will not generate a trace exception for execution of an isync instruction. Other PowerPC processors will generate a trace exception when executing an isync instruction and MSR[SE]=1.

When a trace exception occurs, the following conditions will be true:

- SRR0 points to the next instruction to be executed after the instruction currently being single-stepped.
- The SRR1 register is loaded with bits 16–31 from the MSR; the bits occupy corresponding positions in SRR1. On the 604, there are additional bits set in the SRR1 register. These bits are set as shown in Table 10-11.
Table 10-11
Additional Bits Defined in the 604's SRR1 Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-2</td>
<td>Set to 0b010.</td>
</tr>
<tr>
<td>3</td>
<td>Set for a load instruction, cleared otherwise.</td>
</tr>
<tr>
<td>4</td>
<td>Set for a store instruction, cleared otherwise.</td>
</tr>
<tr>
<td>5-9</td>
<td>Cleared.</td>
</tr>
<tr>
<td>10</td>
<td>Set for lswx or stswx, cleared otherwise.</td>
</tr>
<tr>
<td>11</td>
<td>Set for mtspr instruction.</td>
</tr>
<tr>
<td>12</td>
<td>Set for taken branch, cleared otherwise.</td>
</tr>
<tr>
<td>13-15</td>
<td>Cleared.</td>
</tr>
</tbody>
</table>

Floating-Point Assist Exception (0x00e00)

The floating-point assist exception is optional and is not implemented on any current PowerPC processors. It may be implemented on future PowerPC implementations to facilitate the software emulation of floating-point exceptions.

If the processor attempts to perform a floating-point operation that is not supported in hardware, the floating-point assist exception is invoked. Using the exception handler, the floating-point operation can be emulated in software.

When a floating-point assist exception occurs, the following conditions will be true:

- SRR0 points to the next instruction to execute in the original instruction stream that was executing before the exception.
- The SRR1 register is loaded with bits from the MSR; the bits occupy corresponding positions.

Implementation-Specific Exception Vectors

The remaining vectors are processor implementation specific. That is, they are not defined by the PowerPC architecture and are optional on each
processor implementation. Therefore, each exception description lists the PowerPC processors that implement the vector.

**Performance Monitoring Exception (0x00f00)**

Performance monitoring is available only on the 604 and 620 processors. It allows retrieval of statistical information concerning instruction dispatch, execution, and completion; memory access; and much more. Performance monitoring capabilities of the 604 and 620 are discussed in Chapter 12, "Techniques and Tricks."

The priority of this exception falls between that of the external interrupt exception and the decremener exception. A performance monitoring exception is treated as a normal PowerPC exception and is generated in response to two conditions:

- A counter condition that has been configured in one of the performance monitor counter registers (PMC1 and PMC2).
- A time-base flipped bit counter that has been configured in the MMCR0 register.

**Software TLB Search Exceptions (0x01000-0x01200)**

Exception vectors 0x01000 through 0x01200 are unique to the PowerPC 603 microprocessor because the 603 implements its page table search mechanism in software. These exceptions are part of the 603's software-based page address translation mechanism. Other PowerPC implementations use hardware to perform the same operation and, therefore, do not generate these exceptions. The 603's software-based page table searching mechanism is described in Chapter 8, "Memory Management."

The ITLB miss, DTLB miss, and DTLB miss on store exceptions described here update the SRR0 and SRR1 registers in an identical manner when an exception occurs. When any of the 603 TLB exceptions occur, the following conditions are true:

- SRR0 points to the next instruction to execute in the original instruction stream that generated the exception.
- The SRR1 register is loaded with bits from the MSR; the bits occupy corresponding positions.
Instruction TLB Miss Exception (0x01000)

The ITLB (instruction translation lookaside buffer) miss exception is specific to the 603. An ITLB miss exception is generated when the effective address for an instruction load/store or cache operation cannot be translated by the 603’s ITLBS.

If the ITLB miss exception handler cannot find the page table entry (PTE) that corresponds to the effective address that generated the exception, this exception becomes equivalent to a page fault. In that case, the handler must attempt to restore the state of the processor before invoking the instruction access exception (ISi exception, offset 0x00400).

Data TLB Miss On Load Exception (0x01100)

The DTLB (data translation lookaside buffer) miss on load exception is specific to the 603. A DTLB miss on load exception is generated when the effective address for a data load or cache operation cannot be translated by the 603’s DTLBs.

Like the ITLB miss, if the exception handler cannot find the PTE that corresponds to the effective address that generated the exception, this exception becomes equivalent to a page fault. In this case, the handler must attempt to restore the state of the processor before invoking the data access exception (DSI exception, offset 0x00300).

Data TLB Miss On Store Exception (0x01200)

The DTLB (data translation lookaside buffer) miss on store exception is specific to the 603. A DTLB miss on store exception is generated in one of two cases:

- When the effective address for a data store or cache operation cannot be translated by the 603’s DTLBs, this exception is generated. As with the DTLB miss on load exception, if the exception handler cannot find the PTE that corresponds to the effective address that generated the exception, this exception becomes equivalent to a page fault. In that case, the handler must attempt to restore the state of the processor before invoking the DSI exception.

- If the changed bit of a DTLB page table entry needs to be updated for a store operation, this exception will be generated.
Instruction Address Breakpoint (0x01300)
The instruction address breakpoint exception is generated when a match between the contents of the IABR (instruction address breakpoint register) and the CIA occurs. The exception will occur before the instruction causing the exception has completed execution. This exception is enabled by setting the IABR[30] bit.
This exception is valid on the PowerPC 603, 604, and 620 processors. The 601 uses the run mode/trace exception in place of this exception.

System Management Interrupt Exception (0x01400)
The system management interrupt (SMI) exception is generated by the assertion of the system management interrupt (#SMI) signal. This exception is enabled by setting the external interrupt bit in the MSR (MSR[EE]=1). In the case where the #INT and #SMI signals are asserted simultaneously, the #SMI signal is processed first. This exception is valid on the 603, 604, and 620 processors; the 601 does not have an equivalent exception.
When an SMI exception occurs, the following conditions will be true:

- SRR0 points to the next instruction to execute in the original instruction stream that was executing when the exception occurred.
- The SRR1 register is loaded with bits from the MSR; the bits occupy corresponding positions.

The system management interrupt provides a mechanism that will allow operating systems to perform system management functions (power, security, and so on) in response to an external signal other than the external interrupt.

Run Mode/Trace Exception (0x02000)
A run mode/trace exception is generated when an instruction address breakpoint register (IABR) or data address breakpoint register (DABR) finds a match or when trace mode is enabled (MSR[SE]=1). This exception is enabled by setting HID1[RM] = HID1[8,9] = 0b10. This exception is defined only for the PowerPC 601. The run mode/trace exception is similar to the 603, 604, and 620 implementation of the trace exception (0x00d00).
When a 601 run mode/trace exception occurs, the following conditions will be true:

- SRR0 points to the instruction that caused the exception.
- The SRR1 register is loaded with bits from the MSR; the bits occupy corresponding positions.

The run modes for the 601 processor are specified by the setting of the HID1[M]=HID1[1-3] bits; the action taken by the processor when an address match occurs depends on the setting of HID1[RM]=HID1[8,9]. Table 10-12 summarizes the 601's run modes and address match actions. Table 10-13 summarizes the actions taken upon detection of an address match.

**Table 10-12**
The PowerPC 601's Run Modes

<table>
<thead>
<tr>
<th>HID1[M]=HID1[1-3] Setting</th>
<th>601 Run Mode Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Normal run mode. No breakpoints set.</td>
</tr>
<tr>
<td>001</td>
<td>Undefined — do not use.</td>
</tr>
<tr>
<td>010</td>
<td>Limited instruction address compare. When the address specified in the 601's HID2 matches the current instruction address (CIA), the address match action specified by HID[RM] is taken. The compare performed by this may not detect floating-point and branch addresses.</td>
</tr>
<tr>
<td>011</td>
<td>Undefined — do not use.</td>
</tr>
<tr>
<td>100</td>
<td>Single-step instruction. Use of the trace exception setting [MSR[SE]] is preferred over this run mode.</td>
</tr>
<tr>
<td>101</td>
<td>Undefined — do not use.</td>
</tr>
<tr>
<td>110</td>
<td>Full instruction address compare. When the address specified in the 601's HID2 matches the CIA, the address match action specified by HID[RM] is taken. The target address for b, bc, bcr, and bcc instructions is tested.</td>
</tr>
<tr>
<td>111</td>
<td>Full branch target address compare. When the branch target address specified in the 601's HID2 matches the current branch target address, the address match action specified by HID[RM] is taken. All instructions are tested.</td>
</tr>
</tbody>
</table>
After configuring the 601’s run mode using HID[M], the next step is to configure the action taken when an address match occurs. The HID[RM] bits, shown in Table 10-13, determine the PowerPC 601’s response to an address match condition or a trace condition.

**Table 10-13**

PowerPC 601’s HID[RM] Bits

<table>
<thead>
<tr>
<th>Setting</th>
<th>601 Response to Address Match/Trace Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Hard stop; halt the processor clock.</td>
</tr>
<tr>
<td>01</td>
<td>Soft stop; wait for system activity to settle.</td>
</tr>
<tr>
<td>10</td>
<td>Generate a run mode/trace exception.</td>
</tr>
<tr>
<td>11</td>
<td>Reserved — do not use.</td>
</tr>
</tbody>
</table>

Note that if the 601 is set to single-step each instruction (HID1[M]=0b100) and is set to generate a run mode/trace exception (HID1[RM]=0b10), the processor will be caught in an infinite loop.

**Summary**

This chapter marks the conclusion of the overview of the PowerPC architecture and each PowerPC implementation. We’ve covered nearly every aspect of several PowerPC processors and should be ready to focus a bit on programming. The remaining two chapters deal with hands-on PowerPC programming.

Chapter 11, “PowerPC Assembly Language Examples,” is dedicated to a discussion of PowerPC assembly language instructions and programming. Chapter 12, “Techniques and Tricks,” deals with several miscellaneous topics, including optimization hints and the performance monitoring features of the 604 and 620.
"The chief virtue that language can have is clearness, and nothing detracts from it so much as the use of unfamiliar words."

— Hippocrates

As a programmer, you are already familiar with many of the operations that will be discussed in this chapter. The aspect that is unfamiliar (and must be learned) is the use of PowerPC assembly language. You have, no doubt, used some form of the if-then-else operation, but it's unlikely that you've coded such an operation on a PowerPC microprocessor using PowerPC assembly language. And one of the easiest ways to learn programming on a new computer architecture is by example.

In a sense, Chapters 1 through 10 were a prelude to this chapter. We're finally ready to start playing with PowerPC assembly language. Stepping through basic programming constructs and progressing towards increasingly complex and platform-specific examples is a good way to become familiar with PowerPC assembly. By the end of this chapter, you'll have experienced enough PowerPC assembly language to begin writing your own.

Register use is one area in which we must be careful to state our assumptions. In the x86 world there exist a number of different conventions for passing arguments to subroutines via the stack; Pascal and cdecl are two examples. And, because
the PowerPC industry is relatively young, there is more than one set of register usage conventions for calling subroutines.

To make the code listed in this section useful (and portable), we’ll follow the register usage conventions shown in Table 11-1. They conform to the PowerOpen ABI, Motorola’s register usage conventions, and to IBM’s WorkPlace OS register usage conventions.

Table 11-1
PowerPC Register Usage Conventions

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Software Handling</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPR r0</td>
<td>Volatile</td>
<td>Miscellaneous system usage. This register should not be modified by user-level software. r0 also represents a value of zero in some instructions (see Chapters 5 and 6).</td>
</tr>
<tr>
<td>GPR r1</td>
<td>Preserved</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>GPR r2</td>
<td>Preserved</td>
<td>Reserved for system use. This register is usually used as a pointer to a system data area.</td>
</tr>
<tr>
<td>GPR r3</td>
<td>Volatile</td>
<td>First argument of a function’s argument (parameter) list. r3 is also used to return values to the caller when required.</td>
</tr>
<tr>
<td>GPR r4–r10</td>
<td>Volatile</td>
<td>Second through eighth argument passed to a subprogram.</td>
</tr>
<tr>
<td>GPR r11–r31</td>
<td>Varies</td>
<td>The use of these registers is strictly operating system dependent. They are defined as preserved for the examples in this book.</td>
</tr>
</tbody>
</table>

A volatile register is one whose value may change through the course of subroutine calls; system software does not expect a volatile register to be preserved. All registers, except for r1 and r2, are volatile. Because system software depends on the values contained in r1 and r2, they must be preserved through any calls to subroutines in the code that we write. The operating system, on the other hand, is free to alter their values.

In the examples that follow, we’ll define r11–r31 as preserved. In practice, their use is dictated by the conventions of the operating system being run.

Several other register format conventions are commonly used. For example, a no-op (no operation) instruction on the PowerPC is coded as an OR immediate instruction using GPR0 as each of the first two operands and
the immediate value zero for the third operand. There are three possible ways to express this, shown in the code fragment following. In all cases, we'll use the first operand form with no extra characters or separators.

```
; Three encodings of the no-op instruction using different
; register format conventions. Note that the first method
; represents the format used in this book.
;
ori  r0, r0, 0 ; 1st: no extra characters (preferred)
ori  r.0, r.0, 0 ; 2nd: using a period as a separator character
ori  %r0, %r0, 0 ; 3rd: prepending a percent sign
```

The majority of examples in this chapter will execute at both user and supervisor level. However, the examples that deal with PowerPC-specific operations (such as BAT register manipulation) assume that the processor is running and executing at supervisor level. When supervisor-level instructions are discussed, we'll make a specific note in the comments for that instruction.

In the examples that follow, any high-level description of the example operation will be shown using the C programming language. Whenever possible, each example will follow the same basic format. First, the operation of interest is introduced. Then, our specific implementation of the operation is discussed, followed by the PowerPC assembly source code listing. A brief commentary follows in which the interesting features of the preceding source code will be discussed and explained.

**FUNDAMENTAL OPERATIONS**

Before we dive into the implementation of common programming constructs using PowerPC assembly language, let's take a careful look at a few individual instructions and their operation. By now, you've absorbed quite a bit of background information and are probably eager to begin programming in PowerPC assembly, but just a bit more preface is necessary.

There are many ways to solve a single problem. In fact, as you learn more about a particular platform, the number of ways to implement a solution seems to grow. At this point, we're just now ready to begin programming PowerPC platforms — and the number of ways to do anything is
limited. As a means of introduction in this section, I’ll emphasize clarity over cleverness when demonstrating common operations.

**Instruction Format**

The detailed format for each PowerPC instruction is defined in Appendix A, “PowerPC Instruction Set Reference.” However, there are several generalizations that we can make. (Note that the instructions used in the following examples are discussed in detail following this section.)

When a value is loaded into a register, the first register operand is the destination register. Consider the following example:

```
or  r0, r2, r3 ; r0 is the destination
; and r2 & r3 are sources
```

Here, using the `or` instruction, the leftmost register (`r0`) is the destination. The two source registers (`r2` and `r3`) will be ORed together and the result stored in `r0`. To remember this ordering sequence, you can chant: “load to the left.”

When a value from a register is stored to memory, the order is reversed from that described above.

```
stw  r3, 0x00(r4) ; the contents of r3 are stored
; to address in r4
```

In this example, the `stw` (store word) instruction stores the contents of `r3` to the address contained in `r4`. The source register is now `r3` and the destination register is `r4` — opposite from the positional conventions of the previous example.

**Fundamental Operations**

It has been said that an adult can function in society with a working vocabulary of only 2,000 words. Out of tens of thousands of possible words, we are able to adequately express ourselves using a small subset of the total dictionary. Analogously, out of the total PowerPC instruction set, certain instructions appear more frequently than others. Consider, for example, how common the `mov` instruction is throughout x86 assembly source code.
In this section we’ll concentrate on establishing a core working vocabulary of PowerPC assembly language instructions.

The examples in this section all use general-purpose registers (GPRs) as operands. Therefore, the term register will always refer to a GPR unless specifically noted. Additionally, most examples start with GPRs containing a value such as 0xffffffff. The choice of an initial value is completely arbitrary; a value such as 0xffffffff is useful only in the sense that it shows the operand width explicitly (as opposed to 0x1) and any change to the contents of the register is obvious.

Now let’s examine the operation of some fundamental PowerPC instructions. Using only these instructions, you’ll be able to implement many basic programming constructs:

- Placing a value into a register
- Reading a value from memory
- Writing a value to memory
- Comparing two values
- Branching
- Performing arithmetic and logical operations

**Loading Values Into a Register**

On PowerPC processors, there are three things that you can load into a general-purpose register: an immediate value, the contents of another register, and the contents of a location in memory. To accomplish the first two, we’ll use the `mr`, `li`, and `lis` instructions.

The `mr` (move register) instruction transfers the contents of one GPR into another, as shown here:

```
; BEFORE: r3 = 0xffffffff
; r4 = 0x12345678
mr r3, r4
; AFTER:  r3 = 0x12345678
; r4 = 0x12345678
```

The contents of r3 before the `mr` instruction (0xffffffff) are replaced by the value contained in r4 (0x12345678). There is no modification of r4. As described in Chapter 6, “The PowerPC Instruction Set,” the `mr` instruction is a simplified form of the `or` instruction.
The \texttt{li} (load immediate) and \texttt{lis} (load immediate shifted) instructions load a general-purpose register with an immediate value. The size of the immediate value is limited (by the PowerPC instruction format) to a maximum of 16 bits. To load a register with a 32-bit immediate value requires two separate 16-bit loads: one for the lower 16 bits and one for the upper 16 bits. The following two instructions load a single GPR with a 32-bit immediate value.

\begin{verbatim}
STEP 1. Load the upper 16 bits and clear lower 16 bits
Note: \texttt{lis} $\text{r9, 0x1234}$ (equivalent to \texttt{addi r9,r0,0x1234})

BEFORE: $r9 = 0xffffffff$
\texttt{lis} $r9, 0x1234$
AFTER: $r9 = 0x12340000$

STEP 2. Load the lower 16 bits
Note: This operation is equivalent to $r9 \text{=} 0x5678$

BEFORE: $r9 = 0x12340000$
\texttt{ori} $r9, r9, 0x5678$
AFTER: $r9 = 0x12345678$
\end{verbatim}

In step 1, the \texttt{lis} instruction loads the 16-bit signed immediate value (0x1234) into the upper (high-order) 16 bits of $r9$ and clears the lower 16 bits to zeroes. In step 2, the lower 16 bits of $r9$ are loaded with the signed immediate value 0x5678. The \texttt{ori} performs a bit-wise OR of the immediate value with $r9$, leaving the upper 16 bits of $r9$ unmodified.

The \texttt{li} instruction loads the lower 16 bits of a GPR with an immediate value and can be thought of as the counterpart to the \texttt{lis} instruction. The following code fragment shows \texttt{li} in action.

\begin{verbatim}
li r9, 0x1234 (equivalent to addi r9,r0,0x01)

BEFORE: $r9 = 0xffffffff$
\texttt{li} $r9, 0x1234$
AFTER: $r9 = 0x00001234$
\end{verbatim}

The \texttt{li} instruction loads the lower 16 bits of the target register with the signed quantity 0x1234. However, the 16-bit value is sign extended to 32 bits before being placed in the register. For this reason, \texttt{li} can’t be used after an \texttt{lis} instruction to complete a 32-bit load operation. Loading a GPR with a non-negative immediate value using the \texttt{li} instruction will have the effect of
zeroing out the upper 16 bits. Using $\text{li}$ to load a negative number sets the upper 16 bits of the GPR to 0xFFFF.

**Loading a Value from Memory**

The previous section focused on loading GPRs with the contents of other GPRs and with immediate values. This section demonstrates loading GPRs with values read from memory.

The x86 instruction set defines a single-instruction mnemonic (mov) for all operand sizes. The PowerPC instruction set provides a separate instruction for each operand size: byte, half-word, and word. The following example shows how to load a GPR with a 32-bit word from memory using the $\text{lwz}$ (load word and zero) instruction.

```assembly
lwz r3, 0x00(r4) reads a word from the address (offset + r4).
Assumes that the value 0x12345678 is stored at address 0xf81a0200

; BEFORE: r3 = 0xffffffff
; r4 = 0xf81a0200
lwz r3, 0x00(r4)
; AFTER: r3 = 0x12345678
; r4 = 0xf81a0200
```

In this example, r3 is loaded with the 32-bit value stored at the address contained in r4. This example assumes that the value 0x12345678 is stored at address 0xf81a0200. The 0x00 quantity that precedes (r4) is a 16-bit signed immediate offset that is added to the base address contained in r4 to produce the effective address. As shown, the $\text{lwz}$ operation is loading a 32-bit quantity from the effective address generated by adding 0x00 to the contents of r4. An immediate offset is a useful way to access the elements of a data structure with known offsets from a fixed address, such as elements in an array.

When considering the $\text{lwz}$ instruction, it's natural to ask, “what gets zeroed?” In this chapter, we focus on 32-bit PowerPC implementations — each GPR is 32 bits wide. When loading a 32-bit quantity from memory into a 32-bit GPR, there is no room for any additional “zeroing” of unused bits. However, on 64-bit PowerPC implementations such as the 620, the upper 32 bits of a GPR would be zeroed after using the $\text{lwz}$ instruction. On 32-bit implementations, upper-bit zeroing takes place when using the $\text{lbz}$ (load byte and zero) and $\text{lhz}$ (load half-word and zero) instructions.
Writing a Value to Memory

The complementary step to loading a value read from memory into a register is to write the register’s contents back to memory. As with the load instructions, the PowerPC instruction set contains a separate instruction for each size of operand. This following example demonstrates writing a 32-bit word into system memory using the `stw` (store word) instruction.

```
; stw r5, 0x04(r6) stores a word to the address (offset + r6)
; BEFORE: r5 = 0x1234abcd
; r6 = 0xf81a0200
stw r5, 0x04(r6)
; AFTER: r5 = 0x1234abcd
; r6 = 0xf81a0200
```

In this example, the contents of `r5` (0x1234abcd) are stored to the effective address generated by adding `0x04` to the contents of `r6` (0xf81a0204). As with the `lwz` instruction above, the 16-bit signed immediate offset that precedes `(r6)` is an offset from the base address contained in `r6`. None of the registers used in the operation are modified by the `stw` instruction.

When storing bytes and half-words to memory, the `stb` (store byte) and `sth` (store half-word) instructions are used in an analogous manner.

Comparing Two Values

Modern programming couldn’t exist without a method of changing the flow of execution based on conditional evaluation. And the first step of conditional execution is, of course, evaluating a conditional expression. To that end, the first example uses the `cmpwi` (compare word immediate) instruction to compare the contents of a GPR to a 16-bit signed immediate value. The `cmpwi` instruction treats both the register and the immediate value as signed.

```
; Compare with immediate value
; cmpwi r3, -1 compares the contents of r3 to a signed 16-bit value
;
; BEFORE: r3 = 0xffffffff
; CR0 = CR[0-3] = 0bxxxx (don’t care)
cmpwi r3, -1
; AFTER: r3 = 0xffffffff
; CR0 = 0b0010 = EQ bit set
```
Before the execution of the `cmpwi` instruction, `r3` contains the signed value -1 (0xffffffff). The value of CR0 is irrelevant — it is updated with the results of the `cmpwi` instruction. After executing this instruction, the contents of `r3` remain unchanged. However, the contents of CR0 have been updated. In this case, CR0[EQ] is set to indicate that the contents of `r3` and the 16-bit signed immediate value are equal. This is true because the 16-bit immediate value is sign-extended to 32 bits before comparing it to the contents of `r3`. (Instructions that perform unsigned comparisons are also available.)

The next example uses `cmpw` to compare the contents of `r3` to the contents of `r4`, treating each value as a signed 32-bit integer.

```assembly
; Comparing the contents of two registers.
; cmpw r3,r4 compares the signed contents of r3 to r4

 cmpw r3,r4

 ; BEFORE: r3 = 0xffffffff
 ; r4 = 0x00000001
 ; CR0 = CR[0-3] = 0bxxxx (don’t care)

 ; AFTER: r3 = 0xffffffff
 ; r4 = 0x00000001
 ; CR0 = 0b1000 = LT bit set
```

The value of the CR0 field before the comparison is, again, unimportant; the comparison operation resets all bits in the CR field. In this case, the less-than (CR0[LT]) bit is set, indicating that the contents of `r3` are less than the contents of `r4`.

### Branching

The second aspect of conditional execution is branching. There are two general categories of branch instructions: conditional and unconditional. Unconditional branching is equivalent to the x86 `jmp` (jump) and `call` instructions. The implementation of conditional branching is similar on both architectures in that the name (and encoding) of the conditional branch instruction implies the conditional value on which it depends. The two examples we'll see use the `bl` (unconditional branch with link register update) and `blt` (branch if less than) instructions to demonstrate each category.

Our first example shows an unconditional branch that stores a return address in the link register.
Unconditional branching

Unconditionally branches to the target address and stores the following address in the link register.

- BEFORE: link register = don't care
- AFTER: link register contains address of instruction following bl

Here, the unconditional branch to BranchTarget has the side effect of storing the address of the instruction following bl into the link register (LR). This mechanism is commonly used to call subroutines, which in turn use the value stored in the LR as a return address to return to the calling code.

The second example demonstrates conditional branching. In this case, we assume that a previous compare operation has updated the contents of CR0.

Conditional branching to the link register

ble branches to the address contained in the link register based on values set in the condition register.

- BEFORE: CR0 set by compare
- AFTER: branch taken or not
- CR0 = unchanged

The blt instruction takes the branch to the target address only if CR0[LT] is set. The branch instruction does modify the contents of any registers. Just as found in the x86 instruction set, there are many forms of branch instructions that correspond to the various conditions resulting from compare operations.

Arithmetic and Logical Operations

Arithmetic and logical operations comprise a large number of programming operations. The addi (add immediate) instruction adds two quantities: the contents of a GPR and a 16-bit signed immediate value. The sum is stored into a separate GPR.
In this example, the contents of r4 and 0x0101 are added together and the sum is placed into r3. The immediate value is treated as a signed quantity and sign-extended to 32 bits prior to the addition.

The `and` instruction performs a bit-wise AND of the contents of two GPRs. The resulting value is stored into a separate GPR. The choice of GPRs in the following example is arbitrary — the same GPR could be used in any (or all three) operand positions.

In this example, the contents of r4 and r5 are ANDed together and the result is placed into r3. The ANDed value is treated as a signed quantity and sign extended to 32 bits prior to the addition.

The `mullw` instruction multiplies the two 32-bit values to obtain a 64-bit result. The lower 32 bits of the 64-bit product are stored into the destination register.
In this example, the contents of r4 and r5 are multiplied to form an intermediate 64-bit product. The 64-bit product is truncated to 32 bits and stored into r3. There are forms of this instruction (mulw.) that can be used to determine if the 64-bit product produced an overflow condition when truncated to 32 bits.

The ori (OR immediate) instruction performs a bit-wise OR of two quantities: the contents of a GPR and an immediate 16-bit unsigned value. The result of the OR operation is placed into a separate GPR.

; The or immediate arithmetic instruction.
; The ori instruction ORs the contents of a GPR with
; a 16-bit immediate value and stores the result into a
; separate GPR.
    ori r3,r4,0x0edc

; BEFORE: r3=0x00000000
;         r4=0x00000123
; AFTER:  r3=0x00000fff
;         r4=0x00000123

Here, the contents of r4 are ORed with the unsigned value 0x0edc. The result is placed into r3. Note that no other registers are modified as a result of either operation.

**Basic Programming Constructs**

After you're familiar with basic operations, the next step is to move to a higher level of programming components: branch types, conditional execution, and looping. As we work through the code sequences that follow, you'll become familiar with how PowerPC assembly language fits together, the most commonly used operations, and begin building a set of routines that may be useful in your own programs.

**Branches, Calls, and Returns**

The x86 architecture defines separate jump and call instructions. PowerPC processors use branch instructions to perform all branches, jumps, and subroutine call operations. PowerPC branch instructions come in both conditional and unconditional forms. Listing 11-1a shows how a C program called `main()` sets up and calls a function called `test()`, which requires two-integer (32-bit word) arguments.
Listing 11-1a
The main() function, shown in its C version.

```c
// Listing 11-1a. Branching and return example
// C-version of main()
main(void)
{
    int testVal1=0x12345678,
        testVal2=0xabcdef12; // two 32-bit ints

    testVal2 = test(testVal1, testVal2); // make a call to test()
}
```

Listing 11-1b shows the same routine, this time in PowerPC assembly language. We assume that when main() is entered, the return address (to whatever code invoked the main() routine) is in the link register (LR). This is a common PowerPC calling convention.

Listing 11-1b
The assembly version of the main() function demonstrates how subroutines are called.

```assembly
Listing 11-1b. Branching and return example.
: PowerPC assembly language version of main()
:
main:
    mflr r0 ; save our return address in r0
    stw r0, 0(r1) ; put it on non-volatile stack
    lis r3,0x1234(r1) ; load high-order 16 bits of r3
    ori r3,0x5678 ; and the low-order 16 bits
    lis r4,0xabcd(r1) ; load high-order 16 bits of r4
    ori r4,0xef12 ; and load low-order 16 bits

    We are now ready to branch to test(). Note that r3 contains the first argument and r4 contains the second argument.

    bl test ; branch to test() & update link register

    Subroutine executes and returns.

    mr r4,r3 ; store returned value in r4 (testVal2)
    lwz r0,0(r1) ; get our return addr from stack
    mtlr r0 ; move into link register
    blr ; return to whoever invoked main() using the link register as the return value
```
The `main()` function is treated as a subroutine by other software (such as the operating system). This realization is useful in understanding the register maintenance responsibilities placed on both the caller and the callee. We’ll examine such responsibilities in the example code that follows.

The LR is used (and its contents overwritten) when we call `test()`. It is `main()`’s responsibility, therefore, to save its return address. And `main()` does so by loading the contents of the LR into r0 using the `mflr` (move from link register) instruction. But Table 11-1 states that r0 is volatile; accordingly, we must save the contents of r0 somewhere else; in this case, on the stack using the stack pointer contained in r1. This entry sequence is common to many of the code fragments and functions in this chapter. We will subsequently pay little attention to the entry and exit sequences when there is no new information to be gained.

To prepare for the call to `test()`, `main()` loads r3 and r4 with the two integer arguments. Using the `lis/ori` instruction sequence discussed previously, the 32-bit quantities are loaded into r3 and r4.

By convention, `test()` is free to modify r3 and r4 during execution. As a result, we can’t count on the two values being intact upon return to `main()`. If the two integer values are required after the call to `test()`, they should be loaded into two non-volatile GPRs (r11 and r12, for example) and then transferred into r3 and r4 for the subroutine.

Next, `main()` calls `test()` using the `bl` (unconditional branch with link register update) instruction. This is an important point — the return address is passed to the subroutine in the link register (LR) and not on the stack as in the x86 world. Why? Because one of the fundamental aspects of a RISC architecture is that registers are used for practically everything. And that includes passing arguments and return addresses.

When `test()` returns, its return value is located in r3. As shown in Listing 11-1a, we want to store this return value into testVal2, which corresponds to r4 in Listing 11-1b. Although we don’t subsequently use the return value in this example, tracking its location helps to understand the call and return mechanisms. We use the `mr` (move register) instruction to transfer the return value into r4.

After saving the return value, `main()` is basically done. The `lwz` (load word and zero) and `mtlr` (move to link register) instructions are used to retrieve `main()`’s return address from the stack and place it into the LR.
Finally, the `blr` (unconditional branch to link register) instruction returns to the code that invoked the `main()` routine. If we rewrote the `main()` routine to return a value, we would use r3 in the same manner as the `test()` subroutine.

**If-Else Operation**

The *if-else* statement is perhaps the most fundamental conditional control flow operation; it provides the most basic means of changing the flow of execution based on the outcome of an expression. Listing 11-2a shows the C version of the *if-else* statement.

**Listing 11-2a**

Different values are returned based on the C language implementation of an *if-else* statement.

```c
// Listing 11-2a
//
// if (word1 == 0x10) // if (expression)
// return(word1 * 2); // statement1
// else // else
// return(word2 * 4); // statement2
```

Listing 11-2b shows a direct translation of the C implementation of the *if-else* construct into PowerPC assembly language. Since this is a code fragment, we’re only concerned with our stated assumptions: r3 contains word1 and r4 contains word2.

**Listing 11-2b**

There is a direct correlation between the assembly and C versions of the *if-else* statement.

```assembly
; Listing 11-2b
;
; On Entry:
;   r3 = 32-bit word1
;   r4 = 32-bit word2
; On Exit:
;   r3 = return value
;
; IfElse:
    cmpwi r3,0x10 ; (IF) - compare immediate: r3 == 0x10?
    bne Around1 ; branch to Around1 if not equal
    slwfi r3,r3,1 ; (STMT1) shift left immediate 1 bit
```
The `cmpwi` (compare word immediate) instruction is the key conditional statement of this code fragment. The `cmpwi` instruction corresponds to the `if` statement in Listing 11-2a and compares the contents of r3 with the immediate value 0x10.

The subsequent `bne` (branch if not equal) instruction transfers execution to statement2 (the else case) if the contents of r3 do not equal 0x10. If the branch is not taken, execution will fall through to the first `slwi` (shift left word immediate) instruction, which corresponds to statement1. The `slwi` instruction shifts the contents of r3 left by one — the equivalent of multiplying by two. After shifting, flow of execution branches unconditionally around the `slwi` that corresponds to statement2 in the next to last line of code.

If the `bne` instruction is taken, the path of execution corresponds to the else case of Listing 11-2a. In this case, we shift the contents of GPR r4 left by two — the equivalent of multiplying by four. The shifted value is stored into r3 in the same instruction.

The `blr` (unconditional branch to link register) instruction is used as the return statement. Note that upon exit, the return value is contained in r3, just as we would expect.

### Else-If Operation

A simple extension of the if-else operation is the nested else-if operation, shown in Listing 11-3a. Most C compilers allow numerous levels of else-if statement nesting. To add a new programming element to this example, the else-if conditional statement contains a compound test using word2.

The else-if operation represents a nice forward progression from the previous example because this operation contains a minimum of two conditional statements. Depending on the number of nested else-if statements, the assembly implementation will have to test and branch around any unused code.
Listing 11-3a

The C implementation of an else-if closely resembles the if-else operation.

```c
// Listing 11-3a. else-if example
// C version
// Assumes: word1, word2 are 32-bit ints that have been initialized
// if (word1 == 0x10)
//    return(word1 * 2);
else if ((word2 > 0x20) && (word2 < 0x30))
    return(word2 * 4);
else
    return(word2 + word1);
```

As shown in the assembly language version in Listing 11-3b, the else-if operates similarly to the if-else but adds an additional level of condition checking. Judging from the code below, many else-if (and if-else) operations can be implemented using only the compare-branch pairing of PowerPC instructions. The behavior of these two instructions is closely analogous to the x86 family's compare-jump instruction pairing.

Listing 11-3b

The assembly language implementation of the else-if has at least two conditional operations.

```assembly
; Listing 11-3b.
; PowerPC assembly language version of else-if

ElseIf:
    cmpwi r3,0x10 ; word1 == 0x10?; CR0 is updated by default
    bne Around1 ; no, check next conditional
    sli r3,r3,1 ; yes, multiply by two and return
    b AllDone ; jump to return

Around1:
    cmpwi r4,0x20 ; word2 > 0x20?; CR0 is updated by default
    ble Around2 ; no, perform else operation
    cmpwi r4,0x30 ; word2 < 0x30?; CR0 is updated by default
    bge Around2 ; no, perform else operation
    sli r3,r4,2 ; yes, multiply by four and return
    b AllDone ; jump to return

Around2:
    addc r3,r4,r3 ; add the two words

AllDone:
    blr ; return to caller
```
The beginning of this routine closely resembles that of the previous if-else example. In the first if-else operation (ElseIf label), the compare-branch instruction pairing is used to compare the value of word1 with 0x10 and fall through if the compare result is equal. If the result is not equal, the branch (bne) is used to transfer execution to the next conditional statement. This represents the else-if operation.

At the end of each condition, there is an unconditional branch to a single exit point — the AllDone label. Each code sequence is responsible for setting up the return value in GPR r3 before branching to the AllDone label and returning to the calling code.

**For Loop Operation**

Another key programming construct is the for loop. In Listing 11-4a, we see a simple for loop in which a single addition operation is performed during each iteration of the loop.

**Listing 11-4a**

This C language for loop sums the contents of an integer array.

```c
// Listing 11-4a.
//
// Assume the following definitions
// int word1, int word2[];
//
// for (x=0; x < 0x100; x++)
//   word1 += word2[x];
```

Listing 11-4b shows an assembly language implementation of the for loop. On entry to this code fragment, r3 contains the value of word1 and r4 contains the effective address of the word2 array. The first step is to clear r6 using the li instruction. GPR r6 functions as the for loop counter. Next, GPR r5 is cleared by loading it with a copy of the contents of r6 (zero) using the mr instruction. GPR r5 is used as the offset into the word2 array. The first two instructions thus initialize the registers used during looping.

**Listing 11-4b**

This assembly implementation of a for loop uses a GPR as a counter.

```assembly
: Listing 11-4b. for loop using GPR as counter
: On entry:
: r3 contains the value of word1
: r4 contains the base effective address of word2[]
```
The for loop begins with the `lwzx` (load word and zero indexed) instruction loading an element of the array into r7. This indexed form of the `lwz` instruction uses the contents of a separate GPR as an offset instead of a 16-bit immediate value. Therefore, the effective address used in the load word instruction is the sum of the contents of r4 and r5. The indexed form is useful because of the 4GB addressing range possible when using a 32-bit GPR as an offset.

During each loop, r5 is incremented by 4 to point to the next 32-bit array element; r4 always contains the array’s base effective address. The loop counter, r6, is incremented by one each trip through the loop. The values retrieved from the array (in r7) are accumulated in r3 using the `add` instruction. Finally, the compare and branch pair determines when the counter has reached its limit.

Although we have plenty of general-purpose registers available, there is a built-in counter register (CTR) that is useful for looping operations. The for loop shown in Listing 11-4b uses r6 as the loop counter. In Listing 11-4c, the CTR is the loop counter. Both for loop examples are functionally equivalent.

**Listing 11-4c**

This for loop uses the CTR as a loop-counter.

```assembly
ForLoop2:
li r6,0x100  ; load 0x100 into r6
li r5,0     ; zero r5 by loading zero immediate
mtctr r6    ; store r6 count value to CTR
```

```
Both the general-purpose registers and the CTR can be used as loop counters. And although the second example is more difficult to read, it has two fewer instructions within the loop because the branch instruction is able to decrement CTR, test for zero, and branch in just one instruction.

The bdnz instruction decrements the CTR (by one) automatically; any interval greater than one cannot be handled implicitly by the bdnz instruction. If the for loop advanced by more than one during each iteration (r+=2, for example), then Listing 11-4c could not be used as shown. However, the looping method shown in Listing 11-4b could easily handle decrementing (or incrementing) a counter by more than one each iteration — a GPR-based counter can be modified without restriction.

While and Do While Operations
While loops iterate based on conditional evaluation. A while loop, however, makes no use of an index variable. Listing 11-5a shows the C version of a simple while loop that adds a variable number of elements in an array of integers. The PowerPC assembly language version is shown in Listing 11-5b.

Listing 11-5a
While loops represent another useful looping technique.

```c
Listing 11-5a.
// C version of while loop example
// sums intArray[num..1]

// Assumes: int num;        // number of elements to add
// int r;                   // counter variable
// int intArray[];          // array of integers[num]

while (num > 0)
    r += intArray[num-];
```
Listing 11-5b

While loops can be implemented using the same instructions as for loops.

```assembly
; Listing 11-5b. PowerPC assembly version of addArray()
; On entry:
; r3 contains num
; r4 contains the address of intArray
; On exit:
; r3 contains the sum of intArray[num..1]
; Internal:
; r10 used as a copy of r3 (value of num)
; r12 used as an index into intArray

addArray:
  cmpwi r3,0    ; (WHILE) greater than zero?
  bng addDone  ; no...branch to return
  mr r10,r3    ; move num argument into r10

addLoop:     ; (LOOP)
  sli r12,r10,2 ; r12 = (num * 4) - array index
  lwzx r12,r12,r4 ; load r12 with intArray[r10]
  addc r3,r12,r3 ; r3 += r12
  subi r10,1    ; decrement num
  cmpwi r10,0   ; are we at zero?
  bgt addLoop   ; no, branch back to addLoop

addDone:
; while loop is complete: continue with normal execution
```

As shown in Listing 11-5b, the `mr`, `cmpwi`, and `bng` instructions evaluate the while condition (num > 0). If while is not greater than zero, the loop terminates immediately. Otherwise, the loop starts adding the elements of the array. Because the counter in this example (num, contained in r10) is decremented by one in each loop iteration, we could have used CTR as the loop counter and saved a couple of instructions as discussed in the previous example.

Note also that the while-do style loop in Listing 11-5b can be converted into a do-while loop by removing the initial test for zero before starting the addLoop loop.
The Switch/Case Operation

For our final look into basic programming constructs, we’ll use the switch/case operation. Although the switch/case construct is simply another form of conditional execution, it adds a few interesting aspects. Multiple conditions may evaluate to the same branch, for example, as shown in Listing 11-6a.

**Listing 11-6a**
The switch/case operation can be thought of as a set of if-else statements.

```c
// Listing 11-6a.
//
// Assumes: int testInt; // the switch value
//         int r; // the return value
//
r=1; // initialize the return value
switch(testInt) {
    case 1: // if == 1
        r = r*4;
        break;
    case 2: // if == 2
    case 3: // if == 3
        r = r*8;
        break;
    default: // all other cases
        r = 0;
        break;
}
```

Listing 11-6b shows the assembly translation for the switch/case statement.

**Listing 11-6b**
The assembly implementation of a switch/case operation can be optimized for special cases.

```assembly
; Listing 11-6b. PowerPC assembly version of case statement
; On entry:
; r3 = testInt
; On exit:
; r3 = return value based on case statement
; Internal:
; r4 = scratch register
mulNum:
```
mr r4,r3 ; put testInt word in r4
li r3,1 ; load immediate: r3 = 1
cmpwi r4,1 ; (CASE 1): CR0 is updated by default
bgt GTone ; if CR0[GT] is set

cmpwi r4,0 ; zero?
ble default ; if less than or equal to zero
sli r3,r3,0x2 ; r3 *= 4
b mulNumDone ; unconditional b to code exit

GTone:
cmpwi r4,0x3 ; (CASE 2 and CASE 3)
bgt default ; if greater than 3
sli r3,r3,0x3 ; r3 *= 8
b mulNumDone ; unconditional b to code exit
default:
li r3,0 ; set r3 value to zero
mulNumDone:
; case statement is done: continue with normal execution

The first case statement to be checked is case 1. We could explicitly test for \( r4 == 1 \), but we would have less information if the test failed than we would if we tested for \( r4 > 1 \). In general, assembly programmers have an advantage over compilers in situations such as this implementation of a switch/case statement. As programmers, we can look at the big picture (bigger than a compiler’s perspective) and code such that our implementations take advantage of the situation at hand.

At the top of the routine, we copy the contents of GPR r3 into r4. This accomplishes two things: First, we save a copy of the switch value to test on a case-by-case basis. Second, we are subsequently free to modify r3, which is commonly used to pass values to subroutines and return values from subroutines.

Next, r3 (variable r) is initialized using the li (load immediate) instruction. At this point, we begin the switch/case statement, which consists of a sequence of compare-branch pairings that we’ve seen in previous examples. The switch/case operation is done when execution reaches mulNumDone; at this point, execution continues as normal.

Note that if we reach GTone in Listing 11-6b, we know that the switch value is greater than one. Therefore, we only had to test the switch value to determine if it was greater than 3; if so, then both cases failed. This is a small example of the advantages human code generators have over compilers when writing efficient code.
INTERMEDIATE PROGRAMMING EXAMPLES

Now that we’ve got our feet wet with some simple introductory routines, let’s tackle some day-to-day programming tasks. In this section, we’ll look at two more complex examples: copying a string and a linked-list insert function.

String Manipulation

If computers never had to express the results of executing software to humans, programmers wouldn’t have to use string manipulation routines as much as we do. However, it’s hard to escape the fact that strings and string handling are a significant part of most programs. To that end, let’s examine the implementation of `strcpy()`, C’s common string copying routine. The C language version, shown in Listing 11-7a, is similar to the standard C library implementation of `strcpy()`.

Listing 11-7a

The `strcpy()` function can be implemented using efficient C code.

```c
// Listing 11-7a.
// strcpy(char *s1, char *s2) - Copy string s2 to s1
// Note: s1 must be large enough to hold contents of s2.
// Returns: *s1
char *strcpy(char *s1, const char *s2)
{
    char *tmp = s1;  // declare and auto-initialize
    while(*s1++ = *s2++);  // while *s2 not NULL, copy!
    return(tmp);  // return pointer to copy
}
```

Listing 11-7b shows the PowerPC assembly language implementation. Like its C language counterpart, the assembly implementation of `strcpy()` is quite efficient.

Listing 11-7b

The implementation of `strcpy()` is accomplished in only eight instructions.

```assembly
; Listing 11-7b. First PowerPC assembly version of strcpy()
```
; On entry:
; r3 contains the address of s1 (destination)
; r4 contains the address of s2 (source)
; On exit:
; r3 contains the address of s1 (destination)
; Internal:
; r10 is used as a scratch destination pointer
; r9 is used to copy bytes from string to string

strcpy:
	mr r10,r3 ; copy the destination ptr to r10

strloop:
	lbz r9,0(r4) ; get the first byte from source
	stb r9,0(r10) ; store byte to destination
	addic r10,r10,1 ; increment destination index
	addic r4,r4,1 ; increment source index
	cmpwi r9,0 ; was this byte the NULL terminator?
	bne strloop ; no, branch back up and repeat
	bclr ; return; Strings are copied.

Our implementation of strcpy() is the first example that uses the register indirect with immediate index addressing mode. When loading and storing bytes (or larger units) to a base address with a constant offset, this mode is quite useful.

Accessing Data Structures

Let's build on the concepts that we established in the strcpy() example by writing a routine that manipulates a linked-list structure. The example structure definition, shown in Listing 11-8a, contains a link element, a character pointer, and an integer data element. Each element in the structure will be accessed from a base register using constant offsets, just as was each byte in the strcpy() routine above.

Listing 11-8a
The 12-byte (3-word) TestDataStr is used in the link list insert routine.

    // Listing 11-8a.
    //
    // Text Data Structure
    // note: sizeof(TSTDATA) = 3 32-bit words = 12 bytes
Listing 11-8b shows the C routine that will insert an entry (link list node) at the end of a linked list. To generate a realistic example, the `insert()` function calls three library routines: `strlen()`, `malloc()`, and `strcpy()`. All standard C libraries implement these three routines. Furthermore, we'll be able to judge the efficiency of register-based calling conventions when calling library support functions.

**Listing 11-8b**

The insert routine represents the first example to call standard C library functions.

```c
// Listing 11-8b. Linked list insert function

/*
     * Link list insert function.
     *  @param headPtr  Pointer to the first element of the list.
     *  @param text     Text to insert into the list.
     *  @param Data     Data to include in the new list node.
     *  @return A pointer to the newly added list node.
     */

TSTDATA *insert(TSTDATA headPtr, char *text, int Data)
{
    TSTDATA tempPtr, curPtr;
    char *textData;

    if (headPtr == NULL)
        return(TSTDATA)0;  // fail

    textData = (char *)malloc(strlen(text));
    tempPtr = (TSTDATA)malloc(sizeof(TSTDATA));

    strcpy(textData, text);  // get a copy of the text
    tempPtr->txt = textData;  // fill in our new structure
    tempPtr->data = Data;

    curPtr = headPtr;  // save the first node
    while(curPtr->link != NULL)
    {
        curPtr = curPtr->link;  // get to end of list
    }

    curPtr->link = tempPtr;  // we are now at the end of the list
    tempPtr->link = NULL;

    return(tempPtr);
}
```
Listing 11-8c shows the PowerPC assembly language version of our list insertion code. And while the code may appear intimidating at first glance, it is straightforward once we start stepping through it line by line. So hold onto your hats and let’s take a look.

**Listing 11-8c**

Linked list manipulation is commonly implemented using high-level languages, but there is plenty to learn from an assembly version.

: Listing 11-8c. Inserting a node into a linked list

; On entry:
; r3 contains headPtr (pointer to TSTDATA structure)
; r4 contains a pointer to the text message
; r5 contains our misc data
; On exit:
; r3 = 0 for a failure, or
; r8 = pointer to newly inserted node in link list
; Internal:
; r31 = scratch copy of r3 (data structure ptr)
; r29 = scratch copy of r5 (integer data)
; r28 = scratch copy of r4 (text message pointer)

; insert:
    cmpwi r3.0 ; did someone pass in a NULL pointer?
    beq insExit ; NULL - exit this routine

    mflr r0 ; put return pointer into r0
    stw r0,4(r1) ; save return pointer on stack
    stwu r1,-32(r1) ; create 32-byte local storage

; use mr (move register) simplified mnemonic to copy register contents
;    mr r31,r3 ; copy r3 to r31 (head structure pointer)
    mr r28,r4 ; copy r4 to r28 (text char pointer)

; First allocate room for a new structure. Note that sizeof is a macro that determines the size of the TSTDATA structure at the time of assembly.
;    mr r3,sizeof(TSTDATA) ; r3 = 3*4 = 12 bytes = sizeof(TSTDATA)
    bl malloc ; call library function malloc()
    mr r8,r3 ; ptr to structure memory is returned in r3
              ; copy r3 to r8, now r8 = tempPtr

; Let's immediately store the data we have available in the new structure.
    stw r5,8(r8) ; (tempPtr->data = Data)
346 PowerPC Programming for Intel Programmers

```assembly
stw r12,0(r8); tempPtr->link = NULL;

; Now we're ready to call the library function strlen().
; Recall that our register usage conventions define
; r11-r31 as preserved across subroutine calls. If this
; were not the case, we would need to save each register
; that could get overwritten on the stack.

mr r3,r28; place string pointer address into r3
bl strlen; call library function strlen()

; The string length is returned in r3, which is used as an argument
; for the malloc function below. Note: Calls to subroutines use the
; bl (unconditional branch with link register update) mechanism.

bl malloc; call library function malloc() with size in r3
stw r3,4(r8); (tempPtr->txt = textData)
mr r4,r28; load r4 with original string pointer
bl strcpy; copy the string using example in Listing 11-7a

; Now we're ready to walk the list, looking for the
; last node. We load the link pointer into r12, compare
; it with zero (NULL), and loop if we need to continue
; walking the list.

b midLoop; jump into the middle of the walk loop

walkList:
    lwz r31,0(r31); (curPtr = curPtr->link); zero offset in struct

midLoop:
    lwz r12,0(r31); get curPtr->link for testing purposes
    cmpi r12,0; is the next link a NULL?
    bne walkList

doneWalking:
    stw r8,0(r31); curPtr->link = tempPtr
    mr r3,r8; move new structure pointer into r3 for return

insDone:
    lwz r0,36(r1); restore return pointer from stack
    mtlr r0; move it into r0
    addic r1,r1,32; remove stack frame

insExit:
    bclr 0x14,0; this label is used for NULL data
    return
```
Listing 11-8c contains a couple of new ideas. First, the `insert()` routine stores the return address on the stack (pointed to by r1) and reserves 32 bytes of local storage. Saving the return address contained in the link register (LR) on the stack is necessary because we call library functions from `insert()`, and each call saves a return address in LR to return to our code.

The reason for the creation of a local storage area on the stack, however, isn’t immediately obvious. In practice, the stack frame is needed to save the contents of the registers that are used during the routine (r29, r30, r31). But because we’re assuming that r11 through r31 are preserved, we don’t need to save r29, r30, or r31 in our example.

Even though the stack frame is not a necessary component of this example, it represents another valuable technique. Programmers who are familiar with the x86’s `enter` and `leave` instructions will notice the similarity. Any time a subroutine must modify registers that are required to be preserved across the call, using a stack frame as shown in Listing 11-8c is an appropriate solution. That is, if calling a subroutine could overwrite volatile registers that the calling code depends on, the calling code should save the register values before the subroutine call. After creating the stack frame, the registers can be saved to the newly allocated area using the `stw` instruction and r1 as the stack base pointer.

Reading the Time Base Register

The time base register (TBR), defined and described in Chapter 4, “The PowerPC Programming Model,” holds a long-period counter that can be used for system timing (such as time-of-day calendars). The user-level code in Listing 11-9 shows how to read the 64-bit TBR on all PowerPC implementations.

Two separate read instructions are required due to the size of the TBR (64 bits). On 64-bit PowerPC implementations, it is possible to optimize Listing 11-9 to use only one 64-bit read. Only supervisor-level software is allowed write-access to the time base register.

Listing 11-9

Reading the time base register can be performed by software running at any privilege level.

```assembly
; Listing 11-9. Reading the TBR (upper and lower)
;
; NOTE: This is a user-level function. Any writes to
; the TBR require supervisor-level privilege.
;```
On entry:
nothing assumed

On exit:
r3 contains upper 32 bits of time base register
r4 contains lower 32 bits of time base register
r5 contains upper 32 bits if TBR; same as r3

ReadTBR:
  mftbu r3 ; get upper portion of TBR
  mftb r4 ; get lower portion of TBR
  mftbu r5 ; get upper TBR to check for carry
  cmpw r3,r5 ; check for TBR carry; true if r3 ≠ r5
  bne ReadTBR ; must get values again
  bclr 0x14,0x0 ; return to caller with r3:r4 = TBR

Recall that the exact rate at which the TBR counts is dependent on processor and system implementation. At power-on, the TBR is initialized by system software such as firmware or an operating system initialization routine.

Note that mftbu is a simplified mnemonic for the mftb instruction that specified the upper 32 bits of the TBR; however, mftb is used as a simplified mnemonic in listing 11-9 since it implies the lower portion of the TBR. The mftb instruction is defined in Appendix A, “PowerPC Instruction Set Reference.”

**Advanced Topics**

So far, we’ve examined a few useful (but generic) programming operations using PowerPC assembly language. But we have yet to really dig into the features that make the PowerPC processor family unique. In this section, we’ll look at some increasingly detailed examples.

**Processor Version Determination**

One of the first operations software has to perform is to determine on which PowerPC processor it’s executing. There may be special software support that a particular implementation requires or perhaps specific
optimizations that cannot be used on all processors. The supervisor-level `getVersion()` function, shown in Listing 11-10, will return a value to the caller based on the defined PPC6xx values below.

**Listing 11-10**

Determining on which PowerPC it's running may be your program's first job.

```assembly
: Listing 11-10. Return the processor version number
:
: The following definitions can be used to test the value
: returned from getVersion()
:
PPC601 equ 1
PPC603 equ 3
PPC603e equ 6
PPC604 equ 4
PPC620 equ 20
:
int getVersion(void) - return the processor version to caller
:
getVersion:
    mfpr r3               ; load r3 with processor version register
    rlwinm r3,r3,16,16,31 ; extract version number from PVR value
    blr                   ; return w/ version in r3
```

Okay, maybe this first *advanced* example isn't too advanced, but it is useful! Listing 11-10 could be called as the argument of a switch statement to perform processor implementation-dependent initialization before continuing on to other generic routines. In this manner, a single PowerPC program could handle unique processor-by-processor setup.

**Multiple Word Shifts**

It's often necessary to shift a single quantity that is larger than the size of a general-purpose register. In particular, large bit masks that represent patterns of pixels may be efficiently shifted using the following routines. Listing 11-11 implements both a left and right shift through three registers (96 bits, total) for any 32-bit PowerPC processor. The number of bits to shift, \( n \), must be less than 32 and must be hard-coded into the routine.

**Listing 11-11**

Multiple word shift operations allow the manipulation of bit fields that are wider than a single 32-bit general-purpose register.

```assembly
: Listing 11-11. Three 32-bit word shift by n-bit places
```
NOTE: Each occurrence of the shift value n (<32) should be replaced with a numeric value in the listing below.

On entry:
- r3 contains high-order 32 bits
- r4 contains middle 32 bits
- r5 contains low-order 32 bits
- r6 left or right flag; left = zero; right = non-zero

On exit:
- r3, r4, r5 contains the shifted value

MultiShift:
- `cmpwi r6, 0` ; check shift flag for left
- `bne DoRightShift` ; nope. do right shift

DoLeftShift:
- `rlwinm r3, r3, n, 0, 31-n` ; start with high-order 32-bits
- `rlwimi r3, r4, n, 32-n, 31`  
- `rlwinm r4, r4, n, 0, 31-n`  
- `rlwimi r4, r5, n, 32-n, 31`  
- `rlwinm r5, r5, n, 0, 31-n`  
- `b MultiShiftDone`  

DoRightShift:
- `rlwinm r5, r5, 32-n, n, 31` ; start with low-order 32-bits
- `rlwimi r5, r4, 32-n, 0, n-1`  
- `rlwinm r4, r4, 32-n, 0, 31`  
- `rlwimi r4, r3, 32-n, 0, n-1`  
- `rlwinm r3, r3, 32-n, 0, 31`  

MultiShiftDone:
- `bclr 0x14, 0x0` ; return to caller

The rotate instructions used in Listing 11-11 represent the most complex encoding schemes in the entire PowerPC instruction set. The five separate arguments specify the destination register, source register, shift quantity, starting mask bit position, and ending mask bit position (respectively). The `rlwinm` (rotate left word immediate then AND with mask) and `rlwimi` (rotate left word immediate then mask insert) instructions are defined in Appendix A, “PowerPC Instruction Set Reference.”

The left shift operation starts by rotating the high-order bits within the r3 register, followed by the rotate/insert operation into r4. This process continues through the low-order bits in r5. The right shift operation starts with
the low-order bits in r5 and progresses through increasingly higher-order bits.

Note that both the right and left shift operations use only left shift/rotate instructions. In fact, right shift and right rotate operations are performed using left shift/rotate instructions and a negative value for the number of bit positions to shift.

**Floating-Point 4 x 4 Matrix Multiply**

Matrix math is a popular programming topic and matrix multiplication is an algorithm staple in the programming industry — especially in the graphics arena. Matrix multiplication and code optimization are complementary subjects; software that uses matrix math will commonly require tight code and high performance (for example, CAD, rendering software, and, of course, games).

In general, N x N matrices may be multiplied to generate a resulting N x N matrix (r[N][N]) using the following algorithm.

```c
for (row=0; row < N; row++)
    for (col=0; col < N; col++)
        for (k=0; r[row][col]=0; k < N; k++)
            r[row][col] += p[row][col] * q[row][col];
```

Listing 11-12 implements a version of the algorithm above optimized for 4 x 4 matrices and is an excellent example of PowerPC floating-point instructions. Using single-precision floating-point numbers, the two source matrices (pointed to by r4 and r5) are multiplied together and stored in the destination matrix (pointed to by r3).

Each 4 x 4 matrix is defined using the following declaration:

```c
typedef 4x4matrix {
    single matrix[4][4]; // single precision
} Matrix;
```

**Listing 11-12**

Matrix multiplication is a fundamental operation used with computer graphics.

```c
; Listing 11-12.
;
; MatrixMul( Matrix *dest, Matrix *s1, Matrix *s2 )
; This function will multiply two 4x4 matrices.
```
Note that s2 can be used as the destination matrix as well.

Inputs: *s1: pointer to the first source matrix.
       *s2: pointer to the second source matrix.
Output: *dest: pointer to the destination matrix
        which will be computed.

Assumes:
        r3 *dest
        r4 *s1
        r5 *s2

MatrixMul:
Save non-volatile registers as appropriate for platform

Load these for a test later...
lwz r6, 48(r4)
lwz r7, 52(r4)
lwz r8, 56(r4)
lfs f0, 60(r4)

Load source matrix2 since we need it for every loop.
lfs f31, 0(r5) ; load first element of source matrix
lfs f30, 4(r5) ; load second, and so on...
lfs f29, 8(r5)
lfs f28, 12(r5)
lfs f27, 16(r5)
lfs f26, 20(r5)
lfs f25, 24(r5)
lfs f24, 28(r5)
lfs f23, 32(r5)
lfs f22, 36(r5)
lfs f21, 40(r5)
lfs f11, 44(r5)
lfs f10, 48(r5)
lfs f8, 52(r5)
lfs f7, 56(r5)
lfs f6, 60(r5)

Test and see if r4 matrix has its last row [0 0 0 X], which
is very common. If so, just assign the result matrix to be
r4's last entry times X and skip an iteration.

or  r9, r6,r7 ; or everything together
or  r9, r9,r8
cmplwi r9, 0x0 ; if r9 ==0, then all of the
bne  doAll ; registers contained zero
If we get here, all we have to do is 4 multiplies and we have the last row of our result.

```
li    r6, 0x3         ; put 3 into r6
fmuls f1, f10,f0      ; perform first multiply
mtctr r6              ; load ctr with loop count: 3
fmuls f2, f8,f0       ; start last row of result
stfs f1, 48(r4)
fmuls f3, f7,f0
stfs f2, 52(r4)
fmuls f4, f6,f0
stfs f3, 56(r4)
stfs f4, 60(r4)
b    matrixMultLoop
```

```
doAll:
li    r6, 0x4          ; r6 <- 4 (loop counter)
mtctr r6               ; count register <- r6
```

```
matrixMultLoop:
lfs f0, 0(r4)           ; Get x value
lfs f5, 4(r4)           ; Get y value
fmuls f1, f0,f31        ; Get xres = x * m[0][0]
fmuls f2, f0,f30        ; Get yres = x * m[0][1]
fmuls f3, f0,f29        ; Get zres = x * m[0][2]
fmuls f0, f0,f28        ; Get wres = x * m[0][3]
fmadds f1, f5,f27,f1    ; Get xres = xres + y*m[1][0]
lfs f4, 8(r4)           ; Get z value
fmadds f2, f5,f26,f2    ; Get yres = yres + y*m[1][1]
fmadds f3, f5,f25,f3    ; Get zres = zres + y*m[1][2]
fmadds f0, f5,f24,f0    ; Get wres = wres + y*m[1][3]
lfs f9, 12(r4)          ; Get w value
fmadds f1, f4,f23,f1    ; Get xres = xres + z*m[2][0]
addi r4, r4,16          ; Increment pointer to source_2
fmadds f2, f4,f22,f2    ; Get yres = yres + z*m[2][1]
fmadds f3, f4,f12,f3    ; Get zres = zres + z*m[2][2]
fmadds f0, f4,f11,f0    ; Get wres = wres + z*m[2][3]
fmadds f1, f9,f10,f1    ; Get xres = xres + w*m[3][0]
stfs f1, 0(r3)          ; Store xres
fmadds f2, f9,f08,f2    ; Get yres = yres + w*m[3][1]
stfs f2, 4(r3)          ; Store yres
fmadds f3, f9,f7,f3     ; Get zres = zres + w*m[3][2]
stfs f3, 8(r3)          ; Store zres
fmadds f0, f9,f6,f0     ; Get wres = wres + w*m[3][3]
stfs f0, 12(r3)         ; Store wres
addi r3, r3,16         ; Increment pointer to source_1
bdnz matrixMultLoop
```

; Restore non-volatile registers as appropriate for platform
blr                      ; return
This is a fast, real-world $4 \times 4$ matrix multiply. The key feature of this matrix multiply routine is preloading the values from source matrix 2 into the floating-point registers. Because these source values are used repeatedly, costly reloading is avoided during the main multiplication loop.

**BAT Register Manipulation**

Memory management and the PowerPC’s block address translation (BAT) registers were introduced in Chapter 8, “Memory Management.” Configuring the BAT registers is the responsibility of supervisor-level system software (operating systems and firmware). Whether you plan on writing such software or not, the following routines are good examples of supervisor-level register manipulation using a C-callable function.

Listing 11-13 shows the supervisor-level `modDBATpair()` routine. This C-callable routine accepts arguments that specify the BAT register number (0–3), two 32-bit pointers for the original BAT register settings, and two 32-bit values that specify the new BAT settings. Because this routine returns the original BAT pair values to the calling code, the original memory mapping configuration may be restored at a later time.

**Listing 11-13**

A C-callable routine that modifies any DBAT register pair is a common tool in PowerPC system software.

```c
: Listing 11-13. Data BAT register update routine
 :
: void modDBATpair (unsigned long BATnum,
:                   unsigned long *upperOrig,
:                   unsigned long *lowerOrig,
:                   unsigned long upperNew,
:                   unsigned long lowerNew)
 :
: // Inputs: BATnum  = BAT to replace (0 - 3)
:           upperNew = New BATn Upper Register.
:           lowerNew = New BATn Lower Register.
: :
: // Returns: *upperOrig = Previous BATn Upper Register.
:           *lowerOrig = Previous BATn Lower Register.
: :
: // On entry:
:   r3 = BATnum
:   r4 = ptr to upperOrig
:   r5 = ptr to lowerOrig
:   r6 = upperNew
```
r7 = lowerNew

modDBATpair:
  cmpwi r3, 1 ; DBAT1?
  beq modDBAT1
  cmpwi r3, 2 ; DBAT2?
  beq modDBAT2
  cmpwi r3, 3 ; DBAT3?
  beq modDBAT3
  cmpwi r3, 0 ; DBAT0?
  bne modDone ; invalid BAT number passed in...return

modDBAT0:
  mfdbatu r8, 0 ; handle DBAT0
  stw r8, 0(r4) ; r8 gets current BAT0 Upper
  mfdbat1 r9, 0 ; r9 gets current BAT0 Lower
  stw r9, 0(r5) ; save r9 to lowerOrig
  mtdbatu 0, r6 ; store upperNew into BAT0 Upper
  mtdbat1 0, r7 ; store lowerNew into BAT0 Lower
  b modDone ; unconditional to return

modDBAT1:
  mfdbatu r8, 1 ; handle DBAT1
  stw r8, 0(r4) ; r8 gets current BAT1 Upper
  mfdbat1 r9, 1 ; r9 gets current BAT1 Lower
  stw r9, 0(r5) ; save r9 to lowerOrig
  mtdbatu 1, r6 ; store upperNew into BAT1 Upper
  mtdbat1 1, r7 ; store lowerNew into BAT1 Lower
  b modDone ; unconditional to return

modDBAT2:
  mfdbatu r8, 2 ; handle DBAT2
  stw r8, 0(r4) ; r8 gets current BAT2 Upper
  mfdbat1 r9, 2 ; r9 gets current BAT2 Lower
  stw r9, 0(r5) ; save r9 to lowerOrig
  mtdbatu 2, r6 ; store upperNew into BAT2 Upper
  mtdbat1 2, r7 ; store lowerNew into BAT2 Lower
  b modDone ; unconditional to return

modDBAT3:
  mfdbatu r8, 3 ; handle DBAT3
  stw r8, 0(r4) ; r8 gets current BAT3 Upper
  mfdbat1 r9, 3 ; r9 gets current BAT3 Lower
  stw r9, 0(r5) ; save r9 to lowerOrig
Listing 11-13 contains four blocks of very similar code; this is because the immediate value that designates the BAT register must be explicit in the instruction encoding at the time of assembly. That is, the immediate value that refers to a particular BAT register must be hard-coded in the mfdbatu, mtdbatu, mtdbatl, and mfdbatl instructions. Hypothetically, if a register or an x86-style memory variable could be used to specify the BAT register, Listing 11-13 would be quite simple.

Note that the mtdbatu and mfdbatu instructions are simplified mnemonics for the mtspr and mfspr instructions using the appropriate BAT register SPR value.

**Atomic Memory Accesses**

The PowerPC architecture pays particular attention to the multiprocessing issues that processors must be capable of handling. As a result, all PowerPC processors have built-in mechanisms that allow atomic memory accesses. In other words, a *read-modify-write* operation is guaranteed to complete without another mechanism (such as another processor) disturbing the contents of the effective address of the memory operation. Note that *read-modify-write* refers to the sequence of events that are performed without interruption: reading a value from memory, modifying that value in some manner, and writing it back out to the same address in memory.

The PowerPC is able to guarantee atomic memory accesses through the use of reservations, discussed in Chapter 6, “The PowerPC Instruction Set.” The instructions that employ the reservation mechanism are lwarx (load word and reserve indexed), ldarx (load double word and reserve indexed), stwarx (store word conditional indexed), and stdcx (store double word conditional indexed). These load/store operations are used in pairs to create reservations for particular effective addresses.

In general, operating systems will provide a standard set of atomic access primitives for use by applications and OS utility software; common primitives include fetch-and-store, compare-and-swap, and test-and-set.
Each primitive follows the same operating procedure in order to ensure atomic memory accesses; this general procedure is shown in Figure 11-1.

![Diagram of memory access operations]

**Figure 11-1**
The typical reservation sequence loops while the store conditional operation is not successful.

Note that between the load-and-reserve instruction and the store-conditional instruction, any number of programming operations may execute. There are two restrictions: any subsequent load-and-reserve instruction will supersede the original reservation and the next store-conditional instruction must be for the effective address of the reservation. If either of these two restrictions are violated, the conditional load/store sequence that guarantees an atomic access will fail. That is, the store operation will not be performed because the processor could not guarantee that the store access would be atomic.

Listing 11-14 shows an example of a test-and-set operation using the atomic load/store sequence. The idea is to read a word from main memory, test the word for a predetermined value, and based on that result, set a new value in the word and store it back.
Listing 11-14

Test-and-set is a common operation that uses atomic memory accesses.

; Listing 11-14. Test-and-set code fragment

; On entry:
; r3 contains the value to test for
; r4 contains the effective address of interest
; r5 the new value to write to r4 — if test succeeds

; On exit:
; r5 contains the original value from EA

TestAndSet:
lwarx r6,0,r4 ; load and reserve using EA in r4
cmpw r6,r3 ; compare contents of EA with value in r3
bne TestContinue ; our test failed, do not proceed

; at this point, the value at (r4) corresponds to r3, so continue

stwcx. r5,0,r4 ; store new value to EA in r4
bne TestAndSet ; reservation was lost and store failed
       ; branch back to TestAndSet and retry

TestContinue:

; execution continues...

Listing 11-14 demonstrates several important aspects of using lwarx/stwcx. instruction combinations. The lwarx/stwcx. pair should always be used with the same effective address. In the example above, we used the EA contained in r4 for both the load-and-reserve and store-conditional instruction. However, if a pending reservation must be cleared, using a stwcx. instruction to a valid (but unimportant) effective address will clear the current reservation. If the compare instruction in our test-and-set example fails, we have executed a lwarx instruction with no corresponding stwcx. instruction. This is an acceptable situation; the next lwarx instruction to execute replaces the current reservation.

There is no reason that the original value to test must be loaded from r4 using the load-and-reserve instruction; it could have been loaded with an lwz instruction in a similar manner. However, if the test succeeds, the lwarx instruction would then have to be issued to the same address, repeating the load to create a reservation. The most efficient technique is strictly processor and system dependent.
Simplified vs. Un simplified Mnemonics

Let's examine how Listing 11-14 might look if we were not using simplified mnemonics. All the examples in this chapter use simplified mnemonics — just as any PowerPC programmer would in day-to-day programming. However, if you were to look at the assembly output from a compiler or disassemble a PowerPC binary file, you might see instructions that are very different from what you'd expect.

In general, the output of compilers and disassemblers don't use simplified mnemonics. As a result, Listing 11-15 looks much more confusing than its simplified counterpart. In the following listing, each instruction's simplified counterpart is noted in the comments to the right.

Listing 11-15
Implementing the same routine from 11-14 without simplified mnemonics is much less intuitive.

; Listing 11-15.
;
Else:
  cmpi CR0,0x0,r3,0x10 ; (IF) - same as cmpwi r3,0x10
  bc 0x4,0x2,Around1 ; branch if the condition specified by
                   ; the BI field (0x2 = CR0[EQ]) is FALSE.
                   ; same as bne Around1
  rlwinm r5,r3,0x1,0x0,0x1e ; wow! - same as slwi r5,r3,1
  b Around2 ; unconditional
Around1:
  rlwinm r5,r4,0x2,0x0,0x1d ; - same as slwi r5,r4,2
Around2:
  or r3,r5,r5 ; load r3 w/ ret value - same as mr r3,r5
  bclr 0x14,0x0 ; return - same as b1r

As shown, the compare instruction takes four arguments instead of two. Similarly, the first branch instruction requires three arguments instead of one. This results from the explicit specification of the condition register field and which bits to test within that field. Note that when the condition register is not explicitly specified, CR0 is assumed.

Both slwi instructions are replaced with rlwinm instructions. The comparatively complex rlwinm instruction requires five arguments! Clearly, we would be better off using the simplified form. Finally, the return value is transferred into GPR r3 using an or instruction before branching unconditionally to the link register using bclr.
Clearly, a programmer’s ability to write PowerPC assembly language is greatly enhanced through the use of simplified mnemonics. And it always pays to understand what’s going on behind the scenes — which justifies this brief examination of non-simplified instruction encoding. Having done so, we are free to ignore the cumbersome forms of commonly used instructions.

**Summary**

Many of the examples presented in this chapter can be further optimized. In fact, most of the routines are written to maximize readability and not efficiency.

For example, one potential optimization would be to return directly from each exit and avoid the intermediate jumps. The performance of Listing 11-3b might be improved slightly by replacing the unconditional branch to AllDone with the `bclr` (return) instruction.

The processor’s branch prediction mechanism will minimize the impact of this type of intermediate branching, however. But it’s never a good idea to rely on the processor’s ability to run your code in an optimal manner when you could have coded it more efficiently. In the real world, contention for processor resources at the time of execution could make it difficult to run your code in the most efficient manner.

In Chapter 12, “Techniques and Tricks,” we’ll take a look at some suggestions for creating maximally efficient code — which instructions to use and which instructions to keep out of your code.
"I know a trick worth two of that."
- King Henry IV by William Shakespeare

I've never met the programmer who didn't look back at a first programming project and laugh (or weep) at the unsophisticated nature of that first code. As time goes on, we tend to gather little tricks, undocumented knowledge, and favorite algorithms that help us make our programs a little better or a little faster than everyone else's.

The PowerPC family of processors is still quite new. Nonetheless, there is plenty of great information available that can make your programming easier — and better. This chapter acts as a catch-all for a few interesting topics that would not fill a chapter of their own. Specifically, general programming tips and using the performance monitoring facility of the 604 and 620.

**Programming Tips**

This section contains some general programming guidelines that will help you avoid the cycle eaters! I mean those aspects
of the processor that can steal your program's efficiency. And those poorly written parts of your code that do the same thing just as effectively.

Some of the tips you'll see here will seem more like common sense. Others are useful insights. And some are esoteric cases that you may never run across. Of course, the effectiveness and behavior of the guidelines may vary among PowerPC implementations. In all cases, however, I'm sure you'll garner something useful.

**Interleave Memory Accesses**

PowerPC processors that have a dedicated load/store unit (such as the 603, 604 and 620) benefit if you schedule your code so that individual execution units don't get overloaded and stall execution.

For example, the PowerPC processors that have a dedicated load/store unit do not require the use of the integer or floating-point units to process memory operations. As a result, scheduling your code for these implementations by interleaving memory access operations with integer or floating-point instructions can improve performance.

**Interleave Integer Operations**

The 604 and 620 each have three integer units (IUs). Two of the three integer units will execute the simple integer operations shown in Table 12-1. The third integer unit supports complex integer operations such as bit field manipulation operations.

The nature of these processors means that you can interleave multiple independent integer operations. In this case, we characterize as independent an operation that does not contain a data dependency on a previous instruction. Single-cycle instructions should also be interleaved with multiple-cycle instructions. All integer instructions not listed in Table 12-1 are multiple-cycle integer instructions.

The PowerPC 601 and 603 processors don't have multiple integer execution units. As such, interleaving neither improves nor hurts their performance.

**Table 12-1**

Single-Cycle Integer Instructions on the 604 and 620

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>add, addc, adde</td>
</tr>
<tr>
<td>addi, addic, addic, addis</td>
</tr>
<tr>
<td>addme, addze</td>
</tr>
<tr>
<td>and, andc, andi.</td>
</tr>
</tbody>
</table>
Avoid Load/Store Multiple/ String Instructions

The dynamic register renaming facility of the 604 and 620 permits instructions to execute out of order with respect to their original program sequence, which increases overall throughput. However, in other PowerPC implementations such as the 601 and 603, certain instructions (including all load/store multiple/string operations) monopolize these processor resources, which can adversely affect performance.

The most common use of such instructions is in subroutine prologues or epilogues. Examples of multi-word instructions include \texttt{lmw} (load multiple word), \texttt{lswi} (load string word immediate), \texttt{stmw} (store multiple word), and \texttt{stswi} (store string word immediate). Avoid multiple-word and string instructions on the PowerPC 601 and 603 processors. Instead, try the following mechanisms:

- Expand the register save/restore code in-line
- Branch to special save/restore functions that use in-line sequences of save and restore instructions

On all PowerPC implementations, use the \textit{load with update} instructions judiciously. They’re subject to this multiple register effect if too many of these instructions appear consecutively. How many is too many? If possible, no more than three consecutive load with updates should be used.
Exploit Rename Registers

The PowerPC 604 and 620 microprocessors provide register renaming to increase execution speed. Because there are a limited number of rename buffers implemented in the processor, you should minimize use of this resource. One relatively simple means of doing this is using immediate addressing whenever possible.

For example, an integer register copy can be performed in a single cycle using a number of different instructions. However, using an ori instruction (with an immediate operand of zero) uses only one source register operand. When using the register indirect form of the ori instruction, two source registers are required.

Don't Serialize Execution

Some PowerPC instructions that are used for memory synchronization and serialization have side effects that affect performance adversely. For example, operations that manipulate condition register fields, particularly when multiple condition fields are being accessed by a single instruction, degrade code performance.

Avoid using the mtcrf instruction on the PowerPC 604 processor to update multiple condition register fields. The performance of the mtcrf instruction varies depending on whether access is to one field, no fields, or multiple fields. This is summarized as follows:

- Those mtcrf instructions that update only one field are executed in either of the single-cycle integer units (SCIUs) and the condition register (CR) field is renamed as with any other SCIU instruction.
- Those mtcrf instructions that update either multiple fields or no fields are dispatched to the MCIU and an internal flag is set. While that flag is set, the following instructions will not be dispatched to the MCIU: mtcrf instructions of the same type; mtspr instructions that update the count or link registers; branch instructions that depend on the condition register; and CR logical instructions. The flag is cleared when the mtctr, mtcrf, or mtlr instruction that set the flag completes execution.

Because mtcrf instructions that update a single field do not require such synchronization, they are not subject to the same coding precautions.
The Performance Monitoring Facility

The PowerPC 604 and 620 microprocessors contain a powerful optimization tool — the ability to monitor their own performance. This facility is not defined by the PowerPC architecture and availability is processor-implementation dependent. This section describes the PowerPC 604's performance monitor facility; the PowerPC 620 implements a similar performance monitoring facility.

Monitoring the performance of software, or profiling, is a common operation during software development. Software-based profiler programs make it possible to determine the areas in which code is spending a majority of its time — and may merit optimization. However, the fundamental problem with software-based monitoring and profiling is the overhead associated with running the control software itself. Sometimes the influence can be factored out; sometimes there is no way to distinguish between the software being profiled and the software doing the profiling.

The PowerPC monitoring facility is part of the processor, so there is comparatively little overhead associated with its use. Using the performance monitoring facility, valuable information — such as the number of instructions executed per clock cycle — can be gathered while the software executes.

In general, performance monitor driver software must be written for operating systems before performance monitoring will be supported. That is, there must exist underlying support for the performance monitor facility, separate from the software being monitored. The driver would configure the performance monitoring registers and enable the monitoring facility. Using the driver and some form of output (such as video, a serial terminal, or disk file), many different statistics can be gathered and recorded which help to characterize the operation of both the processor and executing software.

Certain counter conditions in the performance monitoring counter registers can be configured to generate a performance monitor exception (offset 0x00f00). This provides the driver software with an opportunity to record counter values and perform other monitoring duties. Note, however, that counter monitoring is not restricted to performance monitor exception handlers — they can be read by supervisor-level software at any time.
The Monitor Mode Control Register 0 (MMCRO)

The Performance Monitor Counter 1 and 2 (PMC1, PMC2)
Note: Both registers have the same format.

The Sampled Instruction and Data Address Registers (SIA, SDA)
Note: Both registers have the same format.

---

**Performance Monitor Control Register**

The *performance monitor control register* (MMCR0), a 32-bit supervisor-level register, is shown in Figure 12-1. Using MMCR0, you can monitor events, enable exceptions, and specify counting conditions. If you select a combination of events, the monitoring of each will occur concurrently. The two fields within MMCR0 that are relevant are the performance monitor counter register (PMC1 and PMC2) selection fields of MMCR0.

To select the events that are monitored, supervisor-level software must configure MMCR0[19-25] (for PMC1) and MMCR0[26-31] (for PMC2) according to Table 12-2. Note that only the 5 low-order bits are used to configure both bit fields and the remaining high-order bits are zero.
<table>
<thead>
<tr>
<th>MMCR0</th>
<th>Description of PMC1 Event</th>
<th>MMCR0</th>
<th>Description of PMC2 Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 0000</td>
<td>Nothing</td>
<td>000 0000</td>
<td>Nothing</td>
</tr>
<tr>
<td>000 0001</td>
<td>Processor cycles</td>
<td>000 0001</td>
<td>Processor cycles</td>
</tr>
<tr>
<td>000 0010</td>
<td>Number of instructions completed</td>
<td>000 0010</td>
<td>Number of instructions completed</td>
</tr>
<tr>
<td>000 0011</td>
<td>Real-time clock SELECT bit transition</td>
<td>000 0011</td>
<td>Real-time clock SELECT bit transition</td>
</tr>
<tr>
<td>000 0100</td>
<td>Number of instructions dispatched</td>
<td>000 0100</td>
<td>Number of instructions dispatched</td>
</tr>
<tr>
<td>000 0101</td>
<td>Instruction cache miss</td>
<td>000 0101</td>
<td>Number of cycles a load-miss operation takes to complete</td>
</tr>
<tr>
<td>000 0110</td>
<td>Data translation lookaside buffer misses</td>
<td>000 0110</td>
<td>Data cache misses</td>
</tr>
<tr>
<td>000 0111</td>
<td>Branch predicted incorrectly</td>
<td>000 0111</td>
<td>Instruction translation lookaside buffer misses</td>
</tr>
<tr>
<td>000 1000</td>
<td>Number of reservations requested (latx is ready for execution)</td>
<td>000 1000</td>
<td>Branches completed</td>
</tr>
<tr>
<td>000 1001</td>
<td>Number of load data cache misses that exceeded the threshold value with lateral L2 intervention</td>
<td>000 1001</td>
<td>Number of reservations successfully obtained (stcx succeeded)</td>
</tr>
<tr>
<td>000 1010</td>
<td>Number of store data cache misses that exceeded the threshold value with lateral L2 intervention</td>
<td>000 1010</td>
<td>Number of mfspr instructions dispatched</td>
</tr>
<tr>
<td>000 1011</td>
<td>Number of mtspr instructions dispatched</td>
<td>000 1011</td>
<td>Number of icbi instructions</td>
</tr>
<tr>
<td>000 1100</td>
<td>Number of sync instructions</td>
<td>000 1100</td>
<td>Number of isync instructions</td>
</tr>
<tr>
<td>000 1101</td>
<td>Number of eieio instructions</td>
<td>000 1101</td>
<td>Branch unit produced result</td>
</tr>
</tbody>
</table>
Table 12-2
PMC1 and PMC2 Selectable Events (Continued)

<table>
<thead>
<tr>
<th>MMCRO [19-25]</th>
<th>Description of PMC1 Event</th>
<th>MMCRO [26-31]</th>
<th>Description of PMC2 Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 1110</td>
<td>Number of integer instructions being completed every cycle (no loads or stores)</td>
<td>00 1110</td>
<td>Single cycle integer unit 0 produced result</td>
</tr>
<tr>
<td>000 1111</td>
<td>Number of floating-point instructions being completed every cycle (no loads or stores)</td>
<td>00 1111</td>
<td>Multi-cycle integer unit produced result</td>
</tr>
<tr>
<td>001 0000</td>
<td>Load store unit produced result</td>
<td>01 0000</td>
<td>Instructions dispatched to the branch unit</td>
</tr>
<tr>
<td>001 0001</td>
<td>Single cycle integer unit 1 produced result</td>
<td>01 0001</td>
<td>Instructions dispatched to the single cycle integer unit 0</td>
</tr>
<tr>
<td>001 0010</td>
<td>Floating-point unit produced result</td>
<td>01 0010</td>
<td>Number of loads completed</td>
</tr>
<tr>
<td>001 0011</td>
<td>Instructions dispatched to the load store unit</td>
<td>01 0011</td>
<td>Instructions dispatched to the multi-cycle integer unit</td>
</tr>
<tr>
<td>001 0100</td>
<td>Instructions dispatched to the single cycle integer unit 1</td>
<td>01 0100</td>
<td>Number of snoop hits that have occurred</td>
</tr>
<tr>
<td>001 0101</td>
<td>Instructions dispatched to the floating-point unit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001 0110</td>
<td>Snoop requests received</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001 0111</td>
<td>Number of load dcache misses that exceeded the threshold value without lateral L2 intervention</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001 1000</td>
<td>Number of store dcache misses that exceeded the threshold value without lateral L2 intervention</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first five events listed for both counters are known as the reference events. One reference event may be monitored in combination with any other event to provide common, useful measurements of processor performance. For example, if PMC1 counts the number of processor cycles and PMC2 counts the number of completed instructions, one could derive the number of instructions per cycle.
The number of instructions completed per cycle is a significant quantity on superscalar RISC processors because it directly reflects the performance of the processor with respect to code being executed. In particular, every superscalar processor has a theoretical maximum number of instructions that can be completed per cycle (see Chapter 2, "Foundations and Architecture," for PowerPC processor limits). In theory, if a linear instruction stream contained perfectly scheduled code, the superscalar processor should be able to complete instructions at its theoretical limit. Anything less than the theoretical limit is the result of branching and inefficient code scheduling. Therefore, the number of instructions completed per cycle is an important statistic that indicates how efficiently the processor is running and executing code.

When using the performance monitor with a typical operating system, many different processes may be running simultaneously. In this case, pre-emptive task switching poses problems for the monitoring of a particular piece of code. To distinguish between a monitored process and all other processes, the monitoring software uses the MSR[PM] bit. Each process, in practice, would track its own, unique MSR configuration. When a particular process resumes execution, it loads the MSR with its own unique value to re-establish the context for execution. Setting the MSR[PM] bit in a process’s unique copy of the MSR allows the performance monitor to distinguish between processes. Any process having MSR[PM] = 1 will be tracked and monitored.

**Performance Monitor Counter Registers**

The two 32-bit performance monitor counter registers (PMC1 and PMC2), shown in Figure 12-1, can be programmed to generate a single interrupt signal when their sign bits (the high-order bit, 0x80000000) transitions from 0 to 1. PMC1 and PMC2 can be read from or written to by the mfspr and mtspr instructions. Software is expected to initialize the PMC registers to non-negative values. If software places a negative value in the register, an erroneous interrupt will be generated if performance monitoring exceptions are enabled.

**The Sampled Address Registers**

The sampled instruction address (SIA) and sampled data address (SDA) registers, shown in Figure 12-1, are 32 bits wide on 32-bit implementations.
and 64 bits wide on 64-bit implementations. Both sampled address registers can be configured by using mtSpr and mfspr instructions. Controlled by setting MMCR0, these two registers point to the instruction (or data) that caused the threshold event-related performance monitor interrupt (PMI).

Threshold events generate a PMI when the value contained in either PMC1 or PMC2 reaches the limit set in MMCR0[10-15]. The ability to generate a PMI based on a threshold condition makes it possible to characterize very specific events. For example, if PMC1 is counting the number of load cache misses (event 0b01001), and the threshold is set to 2, then only load cache misses taking more than two cycles are counted in PMC1. Threshold events are clock cycle-based (as opposed to iteration-based) and intended to be used to characterize on-chip cache performance.

The SIA contains the effective address of an instruction that started executing at or before the time that the processor generates the PMI. If the PMI is triggered by a threshold-event, the SIA points to the exact instruction that caused the event count to exceed the threshold. The instruction whose effective address is placed in the SIA as the result of a PMI is known as the sampled instruction. If the PMI is generated as the result of a counter-negative condition, then the address placed in the SIA corresponds to the address of the last instruction to complete execution during the clock cycle in which the exception is generated.

The SDA contains the effective address of an operand being referenced by an instruction that was executing at or before the time that the processor generated the PMI. In this case, the contents of the SDA register do not correspond to the value contained in the SIA register. However, if the PMI was triggered by a threshold-event, the SDA contains the effective address of the operand of the sampled instruction.

Performance Monitoring Interrupt

When the most significant (high-order) bit of a counter register (PMC1 or PMC2) transitions from 0 to 1 — indicating a negative value — a PMI is generated and the processor vectors to offset 0x00f00. A PMI is considered an external interrupt (exception) and is therefore enabled only when MSR[EE] = 1. A PMI may be generated upon transition of the time base register from 0 to 1. Using the time base facility to generate a PMI allows profiling software with respect to real time.
This appendix defines the entire PowerPC Instruction Set architecture for both user and supervisor mode instructions. Chapter 6, “The PowerPC Instruction Set,” introduced the PowerPC instruction set and grouped the individual instructions into general categories according to function. This reference will list all instructions alphabetically, so you can turn to any instruction without considering the category in which it belongs.

All PowerPC instructions for the 601, 603, 604, and 620 are 32 bits long and always word-aligned in memory. Aligning each instruction on 32-bit boundaries gives the processor the freedom to ignore the two low-order bits when developing or retrieving addresses.

An awareness of the following conventions will help you interpret each instruction definition.

- Hexadecimal numbers are preceded by “0x.”
- Binary numbers are preceded by “0b.”
- When examples are present in instruction definitions, the first lines are C code and the lines following are the PowerPC assembly language equivalent. The C/assembly format both illustrates the instruction and demonstrates a typical context in which the instruction would be used.
- Bits 0–5 in an instruction encoding always specify the primary opcode. Some instructions also have an extended opcode, generally located in bits 21–30.
- Reserved fields are shaded. If a reserved field is set other than as specified by the instruction encoding, the instruction form is invalid and will result in an exception condition.

- Because this book covers the 601 as well as the 603, 604, and 620 processors, the POWER architecture instructions that are not part of the PowerPC Instruction Set Architecture are described on the CD-ROM that accompanies this book.

### Table A-1
Instruction Bit Definitions and Formats

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA 30</td>
<td></td>
<td>Absolute Address Bit</td>
</tr>
<tr>
<td>BD 16-29</td>
<td>Immediate field specifying a 14-bit signed two's complement branch displacement that is concatenated on the right with OOb and sign-extended to 32 bits.</td>
<td></td>
</tr>
<tr>
<td>BI 11-15</td>
<td>Specifies a bit in the Control Register (CR) to be used as the condition of a branch conditional instruction.</td>
<td></td>
</tr>
<tr>
<td>BO 6-10</td>
<td>Specifies options for the branch conditional instructions. The encoding is described in Chapter 6, which discusses conditional branch control.</td>
<td></td>
</tr>
<tr>
<td>crbA 11-15</td>
<td>Specifies a bit in the CR to be used as a source.</td>
<td></td>
</tr>
<tr>
<td>crbB 16-20</td>
<td>Specifies a bit in the CR to be used as a source.</td>
<td></td>
</tr>
<tr>
<td>crbD 6-10</td>
<td>Specifies a bit in the CR or in the FPSCR as the destination of the result of an instruction.</td>
<td></td>
</tr>
<tr>
<td>crfD 6-8</td>
<td>Specifies one of the CR or FPSCR fields as a destination.</td>
<td></td>
</tr>
<tr>
<td>crfS 11-13</td>
<td>Specifies one of the CR or FPSCR fields as a source.</td>
<td></td>
</tr>
<tr>
<td>CRM 12-19</td>
<td>This field represents a mask that is used to identify the CR fields that are to be updated by the mlcrf instruction.</td>
<td></td>
</tr>
</tbody>
</table>
## Table A-1
Instruction Bit Definitions and Formats (Continued)

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>16-31</td>
<td>Immediate field specifying a 16-bit signed two's complement integer that is sign-extended to 32-bits.</td>
</tr>
<tr>
<td>FM</td>
<td>7-14</td>
<td>Identifies the FPSCR fields that are to be updated by the mtfsf instruction.</td>
</tr>
<tr>
<td>frA</td>
<td>11-15</td>
<td>Specifies a source FPR.</td>
</tr>
<tr>
<td>frB</td>
<td>16-20</td>
<td>Specifies a source FPR.</td>
</tr>
<tr>
<td>frC</td>
<td>21-25</td>
<td>Specifies a source FPR.</td>
</tr>
<tr>
<td>frD</td>
<td>6-10</td>
<td>Specifies a destination FPR.</td>
</tr>
<tr>
<td>frS</td>
<td>6-10</td>
<td>Specifies a source FPR.</td>
</tr>
<tr>
<td>IMM</td>
<td>16-19</td>
<td>An immediate field used as data and placed into a field in the FPSCR (FPSCR field depends on crfD).</td>
</tr>
<tr>
<td>LI</td>
<td>6-29</td>
<td>Immediate field specifying a 24-bit, signed two's complement integer that is concatenated on the right with 00b and sign-extended to 32 bits.</td>
</tr>
<tr>
<td>LK</td>
<td>31</td>
<td>Link Bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = Instruction does not update the link register.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Updates the link register. If the instruction is a branch instruction, the address of the instruction following the branch instruction is placed into the link register.</td>
</tr>
<tr>
<td>MB</td>
<td>21-25</td>
<td>These fields are used in rotate instructions to specify a 32-bit mask consisting of 1 bits from bit MB+32 to ME+32 inclusive, and 0 bits elsewhere.</td>
</tr>
<tr>
<td>ME</td>
<td>26-30</td>
<td></td>
</tr>
<tr>
<td>NB</td>
<td>16-20</td>
<td>Specifies the number of bytes to move in an immediate string load or store operation.</td>
</tr>
<tr>
<td>opcode</td>
<td>0-5</td>
<td>Primary Opcode</td>
</tr>
<tr>
<td>OE</td>
<td>21</td>
<td>Used for extended arithmetic to enable setting XER[OV, SO].</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = XER will not be updated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = XER will be updated to reflect the result of the operation.</td>
</tr>
<tr>
<td>rA</td>
<td>11-15</td>
<td>Specifies a GPR to be used as a source or as a destination.</td>
</tr>
<tr>
<td>rB</td>
<td>16-20</td>
<td>Specifies a GPR to be used as a source.</td>
</tr>
</tbody>
</table>
Table A-1
Instruction Bit Definitions and Formats (Continued)

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rc</td>
<td>31</td>
<td>Record Bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = CR is not updated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = CR is updated to reflect the result of the operation. For integer instructions, CR[0-3] are set to reflect the result as a signed quantity. The result as an unsigned quantity or a bit string can be deduced from the CR[EQ] bit. For floating-point instructions, CR[4-7] are set to reflect floating-point exception, FP enabled exception, FP invalid operation exception, and FP overflow exception.</td>
</tr>
<tr>
<td>rD</td>
<td>6-10</td>
<td>Specifies a destination general-purpose register (GPR).</td>
</tr>
<tr>
<td>rS</td>
<td>6-10</td>
<td>Specifies a source GPR.</td>
</tr>
<tr>
<td>SH</td>
<td>16-20</td>
<td>Specifies a shift amount.</td>
</tr>
<tr>
<td>SIMM</td>
<td>16-31</td>
<td>Immediate field specifies a 16-bit unsigned quantity.</td>
</tr>
<tr>
<td>SPR</td>
<td>11-20</td>
<td>Specifies a special-purpose register for the mtspr and mfspr instructions.</td>
</tr>
<tr>
<td>TO</td>
<td>6-10</td>
<td>Specifies the conditions on which to trap.</td>
</tr>
<tr>
<td>UIMM</td>
<td>16-31</td>
<td>Immediate field specifies a 16-bit unsigned integer.</td>
</tr>
<tr>
<td>XO</td>
<td>21-30</td>
<td>All fields are secondary opcode fields.</td>
</tr>
<tr>
<td></td>
<td>22-30</td>
<td></td>
</tr>
<tr>
<td></td>
<td>26-30</td>
<td></td>
</tr>
<tr>
<td></td>
<td>or 30</td>
<td></td>
</tr>
</tbody>
</table>

Reading an Instruction Definition

Figure A-1 shows our example instruction and details the typical fields that could appear in an instruction definition.
Instruction Name: **addic.**

---

**ADD SIGNED IMMEDIATE TO REGISTER, SET CARRY BIT, AND RECORD**

**FORMS**
- addic. rD, rA, SIMM

**Simplified Mnemonics**
- subic. rD, rA, value = addic. rD, rA, value

**Bit Definition**

<table>
<thead>
<tr>
<th>Ord</th>
<th>D</th>
<th>A</th>
<th>SIMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

**How the instruction works**

**Pseudo Code**

```
r0 ← rA + EXTS(SIMM)
if carry
   set XER[CA]
```

**Description**

The **addic.** (add immediate carrying and record) instruction calculates the sum of a 16-bit signed value and rA and places the result into destination register rD. The 16-bit immediate value, SIMM, is sign extended to 32 bits before the addition operation. If there is a carry out of the most significant bit (MSb) of the result, XER[CA] is set; otherwise, XER[CA] is cleared. The results of this operation are recorded in the CR0 field of the condition register.

**Registers Affected**
- CR0[L,T,GT,EQ,SO]
- XER[CA]

**Example**

```
; XER[CA] set if r8+r3 generates a carry out of the most significant bit
; CR0 reflects the results of this addition operation

addic. r10, r3, 16 ; value in r3 plus 16 is stored in r10
```

---

**Figure A-1**

How to read instructions.
In Figure A-1, the first field to notice is which execution unit is responsible for executing the instruction. This is useful when analyzing instruction scheduling. In general, the execution unit is constant across the PowerPC implementations discussed in this book. However, the load/store unit (if present) is responsible for the execution of load/store instructions; if absent, the integer unit handles load/store instruction execution.

Next, the processor implementations that support this instruction are listed. If a particular PowerPC implementation is absent from this list, then the corresponding instruction represents an invalid form on that implementation. For example, if the field shows 603/604/620, then the instruction is valid for all implementations except the PowerPC 601 processor.

Generally, the most useful part of the definition will be the Forms field, which will show all possible formats for that particular instruction. Following the Forms field, there is an optional Simplified Mnemonics field. Simplified mnemonics are discussed in Chapter 6, “The PowerPC Instruction Set.” Any mnemonics that use this particular instruction as a base will be listed here, along with the equivalent base form. Note that each simplified mnemonic has a unique definition entry in Appendix A.

Next, the Bit Definition field describes the instruction’s bit-wise encoding. Each 32-bit instruction bit definition includes the primary opcode, secondary opcode (if present), and each operand encoding within the instruction.

In the pseudo-code field, a simple pseudo code is used to describe the operation of the instruction. The pseudo-code style found in this book is generally equivalent to the style found in the PowerPC data books. The only areas that differ are those that C-style notation seems more appropriate; a complete listing of all notation and conventions is presented in Table A-2.

Subsequently, the Description field contains a textual description of the operation of the instruction. This section describes typical usage, implementation details, and other interesting and useful features of the instruction.

The Registers Affected field describes the system status registers (condition register, link register, etc.) that are updated as a result of executing this instruction. This field does not list any of the general-purpose or floating-point registers that are modified during execution.

Optionally, the instruction definition concludes with an Example field. Most popular instructions contain this section. This field contains a brief assembly language code fragment that demonstrates how the instruction is used and how it looks in real code.
### Table A-2

**Pseudo Code Notation and Conventions**

<table>
<thead>
<tr>
<th>Notation/Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>←</td>
<td>Assignment. The element on the right of the arrow is assigned to the element on the left of the arrow.</td>
</tr>
<tr>
<td>←iea</td>
<td>Assignment of an instruction effective address.</td>
</tr>
<tr>
<td>¬, NOT</td>
<td>NOT logical operator.</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication.</td>
</tr>
<tr>
<td>/</td>
<td>Division.</td>
</tr>
<tr>
<td>+, /</td>
<td>Two’s-complement addition.</td>
</tr>
<tr>
<td>−</td>
<td>Two’s-complement subtraction, unary minus.</td>
</tr>
<tr>
<td>=, ≠</td>
<td>Equals and Not Equals relations.</td>
</tr>
<tr>
<td>⟨,⟩, ⟨,⟩, ≥</td>
<td>Signed comparison relations.</td>
</tr>
<tr>
<td>.(period)</td>
<td>Update. When used as a character of an instruction mnemonic, a period (.) means that the instruction updates the condition register field.</td>
</tr>
<tr>
<td>c</td>
<td>Carry. When used as a character of an instruction mnemonic, a ‘c’ indicates a carry out in XER[CA].</td>
</tr>
<tr>
<td>e</td>
<td>Extended precision. When used as the last character of an instruction mnemonic, an ‘e’ indicates the use of XER[CA] as an operand in the instruction and records a carry out in XER[CA].</td>
</tr>
<tr>
<td>o</td>
<td>Overflow. When used as a character of an instruction mnemonic, an ‘o’ indicates the record of an overflow in XER[OV] and CR0[SO] for integer instructions of CR1[SO] for floating-point instructions.</td>
</tr>
<tr>
<td>⟨U,⟩U</td>
<td>Unsigned comparison relations.</td>
</tr>
<tr>
<td>?</td>
<td>Unordered comparison relation.</td>
</tr>
<tr>
<td>&amp;, l</td>
<td>AND, OR logical operators.</td>
</tr>
<tr>
<td>II</td>
<td>Used to describe the concatenation of two values (that is, 010 II 111 is the same as 010111).</td>
</tr>
<tr>
<td>⊕, ≜</td>
<td>Exclusive-OR, Equivalence logical operators (for example, a⊕=b = (a⊕¬b)).</td>
</tr>
<tr>
<td>Notation/Convention</td>
<td>Meaning</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------</td>
</tr>
<tr>
<td>Obnnnn</td>
<td>A number expressed in binary format.</td>
</tr>
<tr>
<td>Oxnnnn</td>
<td>A number expressed in hexadecimal format.</td>
</tr>
<tr>
<td>(n)x</td>
<td>The replication of x, n times (that is, x concatenated to itself n-1 times). (n)0 and (n)1 are special cases. A description of the special cases follows:</td>
</tr>
<tr>
<td></td>
<td>* (n)0 means a field of n bits with each bit equal to 0. Thus, (5)0 is equivalent to 0b000000.</td>
</tr>
<tr>
<td></td>
<td>* (n)1 means a field of n bits with each bit equal to 1. Thus (5)1 is equivalent to 0b11111.</td>
</tr>
<tr>
<td>(rA)0</td>
<td>The contents of rA if the rA field has the value 1–31, or the value 0 if the rA field is 0.</td>
</tr>
<tr>
<td>(rX)</td>
<td>The contents of rX.</td>
</tr>
<tr>
<td>X[n]</td>
<td>n is a bit of field within x, where x is a register.</td>
</tr>
<tr>
<td>x^0</td>
<td>x is raised to the nth power.</td>
</tr>
<tr>
<td>ABS(x)</td>
<td>Absolute value of x.</td>
</tr>
<tr>
<td>CEIL[x]</td>
<td>Least integer ≥ x.</td>
</tr>
<tr>
<td>Characterization</td>
<td>Reference to the setting of status bits in a standard way that is explained in the text.</td>
</tr>
<tr>
<td>CIA</td>
<td>Current Instruction Address</td>
</tr>
<tr>
<td></td>
<td>The 32-bit address of the instruction being described by a sequence of pseudo code. Used by relative branches to set the next instruction address (NIA) and by branch instructions with (LK = 1) to set the link register. Note, that unlike the x86 architecture, the CIA does not correspond to any architected register such as EIP.</td>
</tr>
<tr>
<td>Clear</td>
<td>Clear the leftmost and rightmost n bits of a register to 0. This operation is used for rotate and shift instructions.</td>
</tr>
</tbody>
</table>
### Table A-2

Pseudo Code Notation and Conventions (Continued)

<table>
<thead>
<tr>
<th>Notation/Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear left and shift</td>
<td>Clear the leftmost ( b ) bits of a register, then shift the register left by ( n ) bits. This operation can be used to scale a known non-negative array index by the width of an element. These operations are used for rotate and shift instructions.</td>
</tr>
<tr>
<td>Cleared</td>
<td>Bits are set to 0.</td>
</tr>
<tr>
<td>Do</td>
<td>Do Loop</td>
</tr>
<tr>
<td></td>
<td>- Indenting shows range.</td>
</tr>
<tr>
<td></td>
<td>- &quot;To&quot; and/or &quot;by&quot; clauses specify incrementing an iteration variable.</td>
</tr>
<tr>
<td></td>
<td>- &quot;While&quot; clauses give termination conditions.</td>
</tr>
<tr>
<td>DOUBLE(x)</td>
<td>Result of converting ( x ) from floating-point single-precision format to floating-point double-precision format.</td>
</tr>
<tr>
<td>Extract</td>
<td>Select a field of ( n ) bits starting at bit position ( b ) in the source register, right or left justify this field in the target register, and clear all other bits of the target register to zero. This operation is used for rotate and shift instructions.</td>
</tr>
<tr>
<td>EXTS(x)</td>
<td>Result of extending ( x ) on the left with sign bits.</td>
</tr>
<tr>
<td>GPR[x]</td>
<td>General-purpose registers.</td>
</tr>
<tr>
<td>if...then...else</td>
<td>Conditional Execution</td>
</tr>
<tr>
<td></td>
<td>- Indenting shows range</td>
</tr>
<tr>
<td></td>
<td>- Else is optional</td>
</tr>
<tr>
<td>Insert</td>
<td>Select a field of ( n ) bits in the source register, insert this field starting at position ( b ) of the target register, and leave all other bits of the target register unchanged. (No simplified mnemonic is provided for insertion of a field when operating on doublewords; such an insertion requires more than one instruction.) This operation is used for rotate and shift instructions. (Note that simplified mnemonics are referred to as extended mnemonics in the architecture specification.)</td>
</tr>
</tbody>
</table>
Table A-2
Pseudo Code Notation and Conventions (Continued)

<table>
<thead>
<tr>
<th>Notation / Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leave</td>
<td>Leave innermost do loop, or the do loop described in leave statement.</td>
</tr>
<tr>
<td>MASK(x,y)</td>
<td>Mask having ones in positions x through y (wrapping if x &gt; y) and zeroes elsewhere.</td>
</tr>
<tr>
<td>MEM(x,y)</td>
<td>Contents of y bytes of memory starting at address x. In 32-bit mode of a 64-bit implementation, the high-order 32 bits of the 64-bit value x are ignored.</td>
</tr>
<tr>
<td>NIA</td>
<td>Next Instruction Address</td>
</tr>
<tr>
<td></td>
<td>The 32-bit address of the next instruction to be executed (the branch destination) after a successful branch. In pseudo code, a successful branch is indicated by assigning a value to NIA (see below). For instructions that do not branch, the NIA = CIA + 4. Note, that unlike the x86 architecture, the NIA does not correspond to any architected register such as EIP.</td>
</tr>
<tr>
<td>OEA</td>
<td>PowerPC operating environment architecture.</td>
</tr>
<tr>
<td>Rotate</td>
<td>Rotate the contents of a register right or left n bits without masking. This operation is used for rotate and shift instructions.</td>
</tr>
<tr>
<td>ROTL<a href="x,y">64</a></td>
<td>Result of rotating the 64-bit value x left y positions.</td>
</tr>
<tr>
<td>ROTL<a href="x,y">32</a></td>
<td>Result of rotating the 64-bit value x left x y positions, where x is 32 bits long.</td>
</tr>
<tr>
<td>ROTL<a href="x,y">64</a></td>
<td>Result of rotating the 64-bit value x left 64-y positions.</td>
</tr>
<tr>
<td>ROTL<a href="x,y">32</a></td>
<td>Result of rotating the 64-bit value x left 32-y positions, where x is 32 bits long.</td>
</tr>
<tr>
<td>Set</td>
<td>Bits are set to 1.</td>
</tr>
<tr>
<td>Shift</td>
<td>Shift the contents of a register right or left n bits, clearing vacated bits (logical shift). This operation is used for rotate and shift instructions.</td>
</tr>
<tr>
<td>SINGLE (x)</td>
<td>Result of converting x from floating-point double-precision format to floating-point single-precision format.</td>
</tr>
<tr>
<td>SPR(x)</td>
<td>Special-purpose register x.</td>
</tr>
<tr>
<td>TRAP</td>
<td>Invoke the system trap handler.</td>
</tr>
<tr>
<td>Undefined</td>
<td>An undefined value. The value may vary from one implementation to another, and from one execution to another on the same implementation.</td>
</tr>
</tbody>
</table>
ASSOCIATIVITY AND PRECEDENCE

When reading pseudo code for an instruction’s operation, there may be ambiguous sequences of operations and operators. Table A-3 gives precedence rules to allow you to sort out the more involved pseudo-code listings. The operators that are listed first (higher) in the table are applied before those listed lower in the table. Operators at the same level in the table associate from left to right, from right to left, or not at all as shown in the Associativity column.

Table A-3
Pseudo Code Precedence Rules

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[n], function evaluation</td>
<td>Left to right</td>
</tr>
<tr>
<td>[n]x or replication, x[n] or explanation</td>
<td>Right to left</td>
</tr>
<tr>
<td>unary -, NOT</td>
<td>Right to left</td>
</tr>
<tr>
<td>* , /</td>
<td>Left to right</td>
</tr>
<tr>
<td>+ , -</td>
<td>Left to right</td>
</tr>
<tr>
<td>II</td>
<td>Left to right</td>
</tr>
<tr>
<td>=, !=, &lt;, &lt;=, &gt;, &gt;=, ?</td>
<td>Left to right</td>
</tr>
<tr>
<td>I</td>
<td>Left to right</td>
</tr>
<tr>
<td>- [range]</td>
<td>None</td>
</tr>
<tr>
<td>← , ←lea</td>
<td>None</td>
</tr>
</tbody>
</table>

DECODING AN INSTRUCTION OPCODE

You may never have a need or desire to decode a single PowerPC instruction. However, if the opportunity presents itself, you should be prepared. Figure A-2 shows how the opcode for a typical addi instruction relates to the instruction’s definition.

Figure A-2
How to decode an instruction.
It may help to flip to the addi definition and back to see how the instruction is defined before continuing. Understanding where the primary opcode, source and destination registers, and SIMM field reside will be important in just a few sentences. Ready?

The opcode 0x38841234 corresponds to the addi r4,r4,0x1234 instruction. Each digit in the opcode has a gray "umbrella" showing where the nibble originates in the encoding below. It's pretty obvious that we'll have to pick apart the opcode to figure out how it relates to its defined encoding.

From the addi instruction definition, we know the primary opcode is 0x0e and it resides in bits 0–5. The only thing to remember here is that the rightmost bit (bit 5) corresponds to the least significant bit of the primary opcode, as demonstrated by the numbering below the primary opcode bits: 1,2,4,...,32.

The same bit ordering is used for the destination and source register as well. Using the numbering below each of the sets of register bits, it's easy to see how register 4 (r4) was encoded. Since there are 32 general-purpose registers and 32 floating-point registers, this 5-bit scheme works nicely.

Finally, we come to the immediate portion of the opcode: the 0x1234 value. Looking at the bits that correspond to SIMM (bits 16–31), it's clear that you simply encode each of the nibbles, in Big Endian fashion, into the remaining bits.

Of course, the good news is that with a fixed-length instruction set, it's never any more difficult than the above example. It may take a bit of practice, but it'll be well worth the trouble next time you're at a party and searching for a conversation starter.
**INTEGER UNIT**

**601/603/604/620**  
**User Mode**

**FORMS**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>rD, rA, rB</th>
<th>OE</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>addi</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>addo</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>addo</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x1f</th>
<th>D</th>
<th>SPR</th>
<th>0x153</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

```
rD ← rA + rB
```

**DESCRIPTION**

The `add` instruction calculates the sum of rA and rB and places the result into destination register rD. The `add` instruction is preferred over other forms of addition instructions because it sets comparatively few status bits.

For 32-bit implementations, the setting of the XER reflects overflow of the lower-order 32-bit result. For 64-bit implementations, the setting of the XER reflects overflow of the 64-bit result. An overflow condition exists if the carry out of the MSb of the result is not equal to the carry out of the MSb+1.

**REGISTERS AFFECTED**

- CR0[LT,GT,EQ,SO] if Rc = 1
- XER[S0,OV] if OE = 1

**EXAMPLE**

```c
long1 += 0xface;  // globally declared long
```

Assumes:
- r4 = contains address of long1
```
lwz r3, 0(r4)       ; get value from address of long1
addis r5, r0, 0x1   ; put 0x1 in upper part of r5
addi r5, r5, -1330  ; subtract to get 0xface
add r3, r3, r5     ; perform add
stw r3, 0(r4)      ; store back results
```
**addc**

**ADD REGISTERS AND SET CARRY**

**FORMS**

<table>
<thead>
<tr>
<th>Form</th>
<th>OE</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addc</code></td>
<td>rD,rA,rB</td>
<td>0 0</td>
</tr>
<tr>
<td><code>addc</code></td>
<td>rD,rA,rB</td>
<td>0 1</td>
</tr>
<tr>
<td><code>addco</code></td>
<td>rD,rA,rB</td>
<td>1 0</td>
</tr>
<tr>
<td><code>addoc</code></td>
<td>rD,rA,rB</td>
<td>1 1</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>D</th>
<th>A</th>
<th>B</th>
<th>OE</th>
<th>0x0a</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0x1f</strong></td>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

```plaintext
rD ← rA + rB
if carry
    set XER[CA]
```

**DESCRIPTION**

The `addc` (add with carry) instruction calculates the sum of rA and rB and places the result into destination register rD, setting the carry bit as appropriate. If there is a carry out of the most significant bit (MSb) of the result, XER[CA] is set; otherwise, XER[CA] is cleared.

For 32-bit implementations, the setting of the XER reflects overflow of the lower-order 32-bit result. For 64-bit implementations, the setting of the XER reflects overflow of the 64-bit result. An overflow condition exists if the carry out of the MSb of the result is *not* equal to the carry out of the MSb+1.

**REGISTERS AFFECTED**

- CR0[LT,GT,EQ,SO] if Rc = 1
- XER[CA]
- XER[SO,OV] if OE = 1

**EXAMPLE**

```plaintext
addc r10,r9,r12 ; the value in r9 is added to r12 and the
                  ; result is stored in r10 with carry bit
                  ; set appropriately
```
**INTEGER UNIT**

**601/603/604/620**

**User Mode**

**FORMS**

<table>
<thead>
<tr>
<th>Form</th>
<th>OE</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>adde</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>adde.</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>addeo</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>addeo.</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>OxFF</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>OE</th>
<th>OxFFa</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[ rD \leftarrow rA + rB + XER[CA] \]

**DESCRIPTION**

The adde (add extended) instruction calculates the sum of \( rA, rB, \) and the carry bit and places the result into destination register \( rD \). For extended addition operations, the carry bit is usually set by a previous addc (add with carry) instruction.

For 32-bit implementations, the setting of the XER reflects overflow of the lower-order 32-bit result. For 64-bit implementations, the setting of the XER reflects overflow of the 64-bit result. An overflow condition exists if the carry out of the MSb of the result is not equal to the carry out of the MSb+1.

**REGISTERS Affected**

- CR0[LT,GT,EQ,SO] if \( Rc = 1 \)
- XER[CA]
- XER[SO,OV] if \( OE = 1 \)

**Example**

adde r10,r9,r12 ; the value in r9 is added to r12 and to the carry bit and the result is stored in r10
addi
ADD SIGNED IMMEDIATE TO
REGISTER

FORMS
addi  rD,rA,SIMM

SIMPLIFIED MNEMONICS
Load Immediate:    li  rD,value     =  addi  rD,0,value
Load Address:      la  rD,disp(rA)  =  addi  rD,rA,disp
Subtract Immediate: subi rA,rB,value  =  addi  rD,rA,-value

BIT DEFINITION

<table>
<thead>
<tr>
<th>0x0e</th>
<th>D</th>
<th>A</th>
<th>SIMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

PSEUDO CODE
if rA = r0 then rD ← EXTS(SIMM)
else rD ← rA + EXTS(SIMM)

DESCRIPTION
The addi (add immediate) instruction calculates the sum of a 16-bit signed value and
rA and places the result into destination register rD. The 16-bit immediate value,
SIMM, is sign extended to 32 bits before the addition operation. This instruction is
preferred for addition because it sets few status bits. In the case of rA = r0, addi uses
the value 0, not the contents of GPR r0.

The simplified mnemonic li (load immediate) loads an immediate value into a regist­
er. The la mnemonic allows computation of a base-displacement operand; this is
useful to obtain the address of a variable specified by name, allowing the compiler/
assembler to supply the base register number and compute the displacement. The
subi mnemonic is equivalent to adding with a negative value.

REGISTERS AFFECTED
None

EXAMPLE
longl = 0xlbadface;    // longl is a globally declared long

; Assumes:
; r4 = contains address of longl
;
addis r3, r0, 0xlbae    ; load upper value
addi r3, r3, -1330      ; subtract to get desired value
stw r3, 0(r4)          ; store 0xlbadface into longl
INTEGER UNIT

601/603/604/620
User Mode

FORMS
adic rD,rA,SIMM

BIT DEFINITION

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>A</th>
<th>SIMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6 7 8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12 13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>17 18</td>
<td>19</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>22 23</td>
<td>24</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>27 28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

PSEUDO CODE

rD ← rA + EXTS(SIMM)
if carry
    set XER[CA]

DESCRIPTION

The addic (add immediate carrying) instruction calculates the sum of a 16-bit signed value and rA and places the result into destination register rD. The 16-bit immediate value, SIMM, is sign extended to 32 bits before the addition operation. If there is a carry out of the most significant bit (MSb) of the result, XER[CA] is set; otherwise, XER[CA] is cleared.

REGISTERS AFFECTED

XER[CA]

EXAMPLE

; XER[CA] will be set if 16+r3 generates a carry
:
adic r10,r3,16 ; value in r3 plus 16 is stored in r10
The addic (add immediate carrying and record) instruction calculates the sum of a 16-bit signed value and RA and places the result into destination register RD. The 16-bit immediate value, SIMM, is sign extended to 32 bits before the addition operation. If there is a carry out of the most significant bit (MSb) of the result, XER[CA] is set; otherwise, XER[CA] is cleared. The results of this operation are recorded in the CR0 field of the condition register.

REGISTERS AFFECTED
- CR0[LT,GT,EQ,SO]
- XER[CA]

EXAMPLE
- XER[CA] set if 16+R3 generates a carry out of the most significant bit
- CR0 reflects the results of this addition operation
- addic. R10,R3,16 ; value in R3 plus 16 is stored in R10
**INTEGER UNIT**

**601/603/604/620**

**User Mode**

**Forms**

```
addis  rD, rA, SIMM
```

**Simplified Mnemonics**

- **Load Immediate Shifted**: `lis  rD, value = addis rD, 0, value`
- **Subtract Immediate Shifted**: `subis rD, rA, value = addis rD, rA, -value`

**Bit Definition**

```
<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>A</th>
<th>SIMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>27</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>26</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Pseudo Code**

```
if rA = r0 then rD ← EXTS(SIMM || 0x0000)
else rD ← rA + EXTS(SIMM || 0x0000)
```

**Description**

The *addis* (add immediate shifted) instruction calculates the sum of the 16-bit signed value (SIMM \(\ll 16\)) and \(rA\) and places the result into destination register \(rD\). This instruction is preferred for addition operations because it sets few status bits. Note that *addis* uses the value 0, not the contents of GPR \(r0\), if \(rA = r0\).

The *lis* simplified mnemonic can be used to load an immediate value into a register. In the example below, the *lis* instruction could have been substituted for the *addis* instruction. The *subis* instruction is equivalent to adding shifted with a negative value; it is available for programming convenience.

**Registers Affected**

None

**Example**

```
Assumes:
: r4 = address of longl

: addis  r3, r0, 0x1badface  ; load upper value
: addi  r3, r3, -1330       ; subtract to get desired value
: stw  r3, 0(r4)            ; store 0x1badface into longl
```

// longl is a globally declared long
**addmex**  
*ADD REGISTER, MINUS ONE, AND CARRY BIT*

**FORMS**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>OE</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>addme</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>addme</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>addmeco</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>addmeco</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>0x1f</th>
<th>D</th>
<th>A</th>
<th>00000</th>
<th>OE</th>
<th>0xea</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

**Reserved**

**PSEUDO CODE**

```
rD ← rA + XER[CA] - 1
```

**DESCRIPTION**

The *addmex* (add to minus one extended) instruction calculates the sum of rA, XER[CA], and minus one (0xffffffff) and places the result into destination register rD.

For 32-bit implementations, the setting of the XER reflects overflow of the lower-order 32-bit result. For 64-bit implementations, the setting of the XER reflects overflow of the 64-bit result. An overflow condition exists if the carry out of the MSb of the result is *not* equal to the carry out of the MSb+1.

**REGISTERS AFFECTED**

- CR0[LT,GT,EQ,SO] (if Rc = 1)
- XER[CA]
- XER[SO,OV] (if OE = 1)
** INTEGER UNIT **

** 601/603/604/620 **

** USER MODE **

---

### addzex

**ADD CARRY BIT TO REGISTER**

---

**FORMS**

<table>
<thead>
<tr>
<th>Form</th>
<th>Destination (rD)</th>
<th>OE</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>addze</td>
<td>rD, rA</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>addze</td>
<td>rD, rA</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>addzeo</td>
<td>rD, rA</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>addzeo</td>
<td>rD, rA</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

---

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>0x1f</th>
<th>D</th>
<th>A</th>
<th>00000</th>
<th>OE</th>
<th>0xca</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

---

**PSEUDO CODE**

```
rD ← rA + XER[CA]
```  

**DESCRIPTION**

The addze (add to zero extended) instruction calculates the sum of rA and the carry bit and stores the result into destination register rD.

For 32-bit implementations, the setting of the XER reflects overflow of the lower-order 32-bit result. For 64-bit implementations, the setting of the XER reflects overflow of the 64-bit result. An overflow condition exists if the carry out of the MSb of the result is *not* equal to the carry out of the MSb+1.

**REGISTERS AFFECTED**

- CR0[LT,GT,EQ,SO] (if Rc = 1)
- XER[CA]
- XER[SO,OV] (if OE = 1)
PERFORM A BITWISE AND OF TWO REGISTERS

**FORMS**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>rA</td>
<td>rS</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>and.</td>
<td>rA</td>
<td>rS</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>A</th>
<th>B</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

```plaintext
rA ← rS & rB
```

**DESCRIPTION**

The `and` (and) instruction performs a bitwise AND of `rS` with `rB` and stores the result into destination register `rA`.

**REGISTERS AFFECTED**

CR0[LT,GT,EQ,SO] (if Rc = 1)

**EXAMPLE**

```plaintext
longA &= longB; // globally declared longs

; Assumes:
; r4 = contains the value of longA
; r5 = contains the value of longB
;
; and r4, r4, r5 ; AND r4 with r5, put into r4
```
**INTEGER UNIT**

**601/603/604/620**

**User Mode**

**FORMS**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Destination</th>
<th>Source 1</th>
<th>Source 2</th>
<th>Complement</th>
<th>State</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>andc</strong></td>
<td>rA</td>
<td>rS</td>
<td>rB</td>
<td>0</td>
<td></td>
<td>Rc</td>
</tr>
<tr>
<td><strong>andc.</strong></td>
<td>rA</td>
<td>rS</td>
<td>rB</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Bit</th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>0x1f</th>
<th>0x3c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

```plaintext
rA ← rS & (NOT rB)
```

**DESCRIPTION**

The **andc** (AND with complement) instruction performs a bitwise AND of rS with the complement of rB and places the result into destination register rA.

**REGISTERS AFFECTED**

CR0[LT,GT,EQ,SO] (if Rc = 1)

**EXAMPLE**

```plaintext
longA &= ~longB; // both are globally declared longs
```

; Assumes:
; r3 = contains the address of longA
; r4 = contains the value of longA
; r5 = contains the value of longB
;
; andc r4, r4, r5 ; complement and AND together
; stw r4, 0(r3) ; store back results
**AND REGISTER WITH UNSIGNED IMMEDIATE**

**Forms**

\[ \text{andi.} \quad rA, rS, \text{UIMM} \]

**Bit Definition**

<table>
<thead>
<tr>
<th>0x1c</th>
<th>S</th>
<th>A</th>
<th>UIMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

**Pseudo Code**

\[ rA \leftarrow rS \& (0x0000 || \text{UIMM}) \]

**Description**

The `andi` (AND immediate) instruction ANDs \( rS \) with the 16-bit unsigned value \( \text{UIMM} \) and places the result into \( rA \). The 16-bit immediate value, \( \text{UIMM} \), is zero extended to 32 bits before performing the AND operation.

**Registers Affected**

\[ \text{CR0[LT,GTEQ,SO]} \]

**Example**

\[ \text{longA} \&= 0xface; \quad // \text{globally declared long} \]

```assembly
; Assumes:
; \( r4 \) = address of \text{longA}
:
\text{lwz} \quad r3, 0(r4) ; \text{get value from address}
\text{andi} \quad r3, r3, 0xface ; \text{and immediate w/ 0xface}
\text{stw} \quad r3, 0(r4) ; \text{store back results}
```
INTEGER UNIT
601/603/604/620
USER MODE

FORMS
andis. rA,rS,UIMM

BIT DEFINITION

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>A</th>
<th>UIMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1d</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

PSEUDO CODE
rA ← rS & (UIMM || 0x0000)

DESCRIPTION
The andis. (AND immediate shifted) instruction ANDs the contents of rS with (UIMM << 16) and places the result into rA.

REGISTERS AFFECTED
CR0[LT,GT,EQ,SO]
bx

**BRANCH UNCONDITIONALLY TO TARGET ADDRESS**

**FORMS**

<table>
<thead>
<tr>
<th>Form</th>
<th>Description</th>
<th>AA</th>
<th>LK</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>Branch unconditionally to</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ba</td>
<td>target-address</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>bl</td>
<td>target-address</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>bla</td>
<td>target-address</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Simplified Mnemonics**


**Bit Definition**

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-31</td>
<td>LI</td>
<td>Address</td>
</tr>
<tr>
<td>32-39</td>
<td>AA</td>
<td>Address</td>
</tr>
<tr>
<td>40-47</td>
<td>LK</td>
<td>Address</td>
</tr>
</tbody>
</table>

**Pseudo Code**

if AA, then NIA ← iea EXTS(LI || 0b00)
else NIA ← iea CIA+EXTS(LI || 0b00)

if LK, then
LR ← iea CIA+4

**Description**

The bx (branch) instruction branches unconditionally to the target address, where target-address specifies the branch target address.

If AA = 0, then the branch target address is the sum of LI || 0b00 (sign-extended) and the address of this instruction. If AA = 1, then the branch target address is the value LI || 0b00 (sign-extended).

If LK = 1, then the effective address of the instruction following the branch instruction is placed into the link register.

**Registers Affected**

LR (if LK = 1)
Branch Processing Unit
601/603/604/620
User Mode

Forms

<table>
<thead>
<tr>
<th>bc</th>
<th>BO, BI, target-addr</th>
<th>0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>bca</td>
<td>BO, BI, target-addr</td>
<td>1 0</td>
</tr>
<tr>
<td>bcl</td>
<td>BO, BI, target-addr</td>
<td>0 1</td>
</tr>
<tr>
<td>bcla</td>
<td>BO, BI, target-addr</td>
<td>1 1</td>
</tr>
</tbody>
</table>

Simplified Mnemonics


Pseudo Code

\[
\text{if } !\text{BO}[2], \text{then } CTR \leftarrow CTR-1 \\
\text{ctr}_{-}\text{ok} \leftarrow \text{BO}[2] | ((\text{CTR} \& 0) \oplus \text{BO}[3]) \\
\text{cond}_{-}\text{ok} \leftarrow \text{BO}[0] | (\text{CR}[\text{BI}] = \text{BO}[1]) \\
\text{if } \text{ctr}_{-}\text{ok} \& \text{cond}_{-}\text{ok} \text{ then} \\
\quad \text{if } AA \text{ then } NIA \leftarrow \text{ieaEXTS(BD} || \text{0b00}) \\
\quad \text{else } NIA \leftarrow \text{ieaCIA} + \text{EXTS(BD} || \text{0b00}) \\
\text{if } \text{LK}, \text{then } LR \leftarrow \text{ieaCIA}+4
\]

Description

The bc (branch conditional) instruction branches conditionally to the branch target address specified by target_address.

The BI field specifies the bit in the condition register (CR) to be tested as the condition of the branch. The BO field is used as described in Table 6-20 of Chapter 6. If AA = 0, then the branch target address is the sum of BD || 0b00 (sign-extended) and the address of this instruction. If AA = 1, then the branch target address is the value BD || 0b00 (sign-extended). If LK = 1, then the effective address of the instruction following the branch instruction is placed into the link register.

Registers Affected

- CTR (if BO[2] = 0)
- LR (if LK = 1)
**bcctr**

**Branch conditionally to count register**

**Forms**

<table>
<thead>
<tr>
<th></th>
<th>LK</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcctr</td>
<td>BO, BI 0</td>
</tr>
<tr>
<td>bcctr</td>
<td>BO, BI 1</td>
</tr>
</tbody>
</table>

**Simplified Mnemonics**

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>bcctr 12,0</th>
</tr>
</thead>
<tbody>
<tr>
<td>blctr</td>
<td>bcctr 12,0</td>
</tr>
<tr>
<td>bnectr cr2</td>
<td>bcctr 4,10</td>
</tr>
</tbody>
</table>

**Bit Definition**

<table>
<thead>
<tr>
<th></th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x13</td>
<td>BO</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31</td>
<td></td>
</tr>
</tbody>
</table>

**Pseudo Code**

```
cond_ok ← BO[0]|(CR[BI] = BO[1])
if cond_ok then
    NIA ← ieCTR || 0b00
if LK then
    LR ← ieCIA+4
```

**Description**

The `bcctr` (branch conditional to count register) instruction conditionally branches to the target address contained in the count register. The BI field specifies the bit in the condition register to be used as the condition of the branch. The BO field is used as described in Table 6-20 of Chapter 6, and the branch target address is CTR[0-29] || 0b00. If LK = 1, then the effective address of the instruction following the branch instruction is placed into the link register. If the “decrement and test CTR” option is specified (BO[2] = 0), the instruction form is invalid.

On the 620, the branch target address is CTR[0-61] || 0b00. When 64-bit implementations are operating in 32-bit mode, the high-order 32 bits of the target address are cleared to zero.

**Registers Affected**

LR (if LK = 1)
**Branch Processing Unit**

**601/603/604/620 User Mode**

**Forms**

- `bclr` BO,BI 0
- `bclrc` BO,BI 1

**Simplified Mnemonics**

- `bltr`  
- `bnlcr` cr2  
- `bdnzlr`  

**Bit Definition**

<table>
<thead>
<tr>
<th>Reserved</th>
<th>BO</th>
<th>BI</th>
<th>00000</th>
<th>0x10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x13</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

**Pseudo Code**

```plaintext
if !BO[2], then CTR ← CTR-1
ctr_ok ← BO[2] | ((CTR≠0) ⊕ BO[3])
cond_ok ← BO[0] | (CR[BI] = BO[1])
if ctr_ok & cond_ok then
  if AA then NIA ← ieaEXTS(BD || 0b00)
  else NIA ← ieaCIA + EXTS(BD || 0b00)
if LK, then LR ← ieaCIA+4
```

**Description**

The `bclr` (branch conditional to link register) instruction conditionally branches to the target address contained in the link register. The BI field specifies the bit in the condition register to be used as the condition of the branch. The BO field is used as described in Table 6-20 of Chapter 6, and the branch target address is LR[0-29] || 0b00. If LK = 1, then the effective address of the instruction following the branch instruction is placed into the link register.

On the 620, the branch target address is CTR[0-61] || 0b00. When 64-bit implementations are operating in 32-bit mode, the high-order 32 bits of the target address are cleared to zero.

**Registers Affected**

- CTR (if BO[2] = 0)
- LR (if LK = 1)
clrldi
CLEAR LEFT DOUBLEWORD IMMEDIATE

FORMS
clrldi rA,rS,n (n<64)  =  rldicl rA,rS,0,n

BIT DEFINITION

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>A</th>
<th>00000</th>
<th>n</th>
<th>000</th>
<th>0</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

PSEUDO CODE

m ← MASK(n,63)
rA ← (rA & m)

DESCRIPTION

The clrldi (clear left doubleword immediate) instruction clears the high-order (left-most) n bits of rS and places the result in destination register rA. This instruction is a simplified form of the rldicl instruction.

REGISTERS AFFECTED
CR0[LT,GT,EQ,SO] (if Rc = 1)
INTEGER UNIT

620

USER MODE

FORMS

\texttt{clrlsldi} \ rA,rS,b,n \ (n \leq b \leq 63) \quad \equiv \quad \texttt{rldic} \ rA,rS,n,b-n

\textbf{BIT DEFINITION}

<table>
<thead>
<tr>
<th></th>
<th>0x1e</th>
<th>S</th>
<th>A</th>
<th>n*</th>
<th>b-n</th>
<th>0x02</th>
<th>n*</th>
<th>R0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Note: This is a split field.

\textbf{PSEUDO CODE}

\begin{align*}
r & \leftarrow \text{ROTL}(rS,n) \\
 m & \leftarrow \text{MASK}(b-n, 63-n) \\
rA & \leftarrow (r \& m)
\end{align*}

\textbf{DESCRIPTION}

The \texttt{clrlsldi} (clear left and shift left doubleword immediate) instruction clears the high-order (left-most) \(b\) bits of \(rS\) and shifts the result left by \(n\) bits. The result is placed in destination register \(rA\). This instruction is a simplified form of the \texttt{rldic} instruction.

\textbf{REGISTERS AFFECTED}

\texttt{CR0[LT,GT,EQ,SO]} (if \(Rc = 1\))

\textbf{EXAMPLE}

; Assumes: we want to clear the high order 8 bits of \(r3\)
; and shift the result left by 4 bits. We also
; want to store the result into \(r4\).
;
; starting value of \(r3\) = \texttt{0xcdef1234abcd5678}

\texttt{clrlsldi} \ r4, \ r3, 8, 4 \ ; \text{here, } b=8 \text{ and } n=4

; The instruction operation proceeds as follows:
; step 1: rotate \(r3\) left by \(n\) bits = 4 bits
; result: \(r3 = 0x\text{def1234abcd5678c}\)
; step 2: generate mask from \((b-n)\) through \((63-n) = \text{1 bits from 4-59}
; result: mask = \texttt{0xffffffffffff0}
; step 3: AND rotated data with mask
; result: \(r3 = 0x\text{ef1234abcd56780}\)
; step 4: store result into destination register \(r4\)
; result: \(r4 = 0x\text{ef1234abcd56780}\)
CLRLSLWI
CLEAR LEFT AND SHIFT LEFT BY IMMEDIATE

FORMS
CLRLSLWI rA,rS,b,n (n ≤ b ≤ 31) = RLWINM rA,rS, n, b-n, 31-n

BIT DEFINITION

<table>
<thead>
<tr>
<th>0x15</th>
<th>S</th>
<th>A</th>
<th>n</th>
<th>b-n</th>
<th>31-n</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
</tr>
</tbody>
</table>

PSEUDO CODE

r ← ROTL(rS, n)
m ← MASK(b-n, 31-n)
rA ← (r & m)

DESCRIPTION

The CLRLSLWI (clear left and shift left word immediate) instruction clears the high-order (left-most) b bits of rS and shifts the result left by n bits. The result is placed in destination register rA. This instruction is a simplified form of the RLWINM instruction.

REGISTERS AFFECTED
CR0[LT,GT,EQ,SO] (if Rc = 1)

EXAMPLE

Assumes: we want to clear the high order 4 bits of r3 and shift the result left by 4 bits. We also want to store the result into r4.

starting value of r3 = 0xcdef1234

CLRLSLWI r4, r3, 4, 4 ; here, b=4 and n=4

The instruction operation proceeds as follows:

step 1: rotate r3 left by n bits = 4 bits
result: r3 = 0xdef1234c

step 2: generate mask from (b-n) through (31-n) =1 bits from 0-27
result: mask = 0xffffffff0

step 3: AND rotated data with mask
result: r3 = 0xdef12340

step 4: store result into destination register r4
result: r4 = 0xdef12340
**INTEGER UNIT**

**601/603/604/620**

**User Mode**

**clrlwi**

**Clear left immediate**

**Forms**

\[ \text{clrlwi } rA,rS,n \ (n<32) = \text{rlwinm } rA,rS,0,n,31 \]

**Bit Definition**

<table>
<thead>
<tr>
<th>0x15</th>
<th>S</th>
<th>A</th>
<th>0</th>
<th>n</th>
<th>31</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
</tr>
</tbody>
</table>

**Pseudo Code**

\[ m \leftarrow \text{MASK}(n,31) \]

\[ rA \leftarrow (rS \& m) \]

**Description**

The `clrlwi` (clear left word immediate) instruction clears the high-order (leftmost) \( n \) bits of \( rS \) and places the result in destination register \( rA \). This instruction is a simplified form of the `rlwinm` instruction.

**Registers Affected**

\( \text{CR0[LT,GT,EQ,SO]} \) (if \( \text{Rc} = 1 \))
**clrrdi**

**CLEAR RIGHT IMMEDIATE**

**FORMS**

\[ \text{clrrdi } rA, rS, n \ (n < 64) = \text{rldicr } rA, rS, 0, 63-n \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>A</th>
<th>00000</th>
<th>63 – n</th>
<th>0x01</th>
<th>0</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[ m \leftarrow \text{MASK}(0, 63-n) \]

\[ rA \leftarrow rS \& m \]

**DESCRIPTION**

The `clrrdi` (clear right immediate) instruction clears the low-order (rightmost) \(n\) bits of \(rS\) and places the result in destination register \(rA\). This instruction is a simplified form of the `rldicr` instruction.

**REGISTERS AFFECTED**

\(\text{CR0[LT, GT, EQ, SO]}\) (if \(Rc = 1\))
**INTEGER UNIT**

**601/603/604/620**

**User Mode**

**clrrwi**

**CLEAR RIGHT IMMEDIATE**

**FORMS**

\[
\text{clrrwi } rA, rS, n \text{ (n<32)} = \text{rlwinm } rA, rS, 0, 0, 31-n
\]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>0x15</th>
<th>S</th>
<th>A</th>
<th>0</th>
<th>0</th>
<th>31 – n</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[
m \leftarrow \text{MASK}(0, 31-n)
\]

\[
rA \leftarrow (r \& m)
\]

**DESCRIPTION**

The clrrwi (clear right immediate) instruction clears the low-order (rightmost) \( n \) bits of \( rS \) and places the result in destination register \( rA \).

**REGISTERS AFFECTED**

\( \text{CR0[LT,GT,EQ,SO]} \) (if \( \text{Rc} = 1 \))
**INTEGER UNIT**

**601/603/604/620**

**User Mode**

---

### cmp

**COMPARE**

**REGISTERS**

#### FORMS

\[
\text{cmp} \quad \text{crfD,L,rA,rB}
\]

#### SIMPLIFIED MNEMONICS

- \[
\text{cmpd} \quad \text{rA,rB} \quad \equiv \quad \text{cmp} \quad 0,1,\text{rA,rB}
\]
- \[
\text{cmpw} \quad \text{cr3,rA,rB} \quad \equiv \quad \text{cmp} \quad 3,0,\text{rA,rB}
\]

#### BIT DEFINITION

<table>
<thead>
<tr>
<th>Res.</th>
<th>crfD</th>
<th>L</th>
<th>A</th>
<th>B</th>
<th>0000000000</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x1f</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

#### PSEUDO CODE

\[
\begin{align*}
\text{if } \text{rA} < \text{rB} \text{ then } c & \leftarrow 0b100 \\
\text{else if } \text{rA} > \text{rB} \text{ then } c & \leftarrow 0b010 \\
\text{else } c & \leftarrow 0b001
\end{align*}
\]

\[
\text{CR}[4*\text{crfD}-4*\text{crfD}+3] \leftarrow c \mid \mid \text{XER}[SO]
\]

#### DESCRIPTION

The \text{cmp} (compare) instruction compares rA with rB, treating the operands as signed integers. The result of the comparison is placed into condition register field crfD; if crfD is not specified, CR0 is used.

On the 620, L controls whether the instruction operands are treated as 64- or 32-bit operands, with L = 0 indicating 32-bit operands and L = 1 indicating 64-bit operands. L is ignored on the 601, 603, and 604.

#### REGISTERS AFFECTED

CR Field specifically controlled by operand crfD[LT,GT,EQ,SO]

#### EXAMPLE

\[
\begin{align*}
\text{if } (\text{longA} < \text{longB}) \quad & \quad \text{// both are globally declared longs} \\
& \quad \text{longA} = \text{longB};
\end{align*}
\]

: Assumes:

: r3 = longA, 32-bit value
: r4 = longB, 32-bit value

: \[
\begin{align*}
\text{cmp} \quad 0x0, 0x0, \text{r3}, \text{r4} & \quad ; \text{is r3 greater than r4?} \\
\text{bgt} \quad \text{Around} & \quad ; \text{yes, jump around assignment} \\
\text{mr} \quad \text{r3, r4} & \quad ; \text{place contents of r4 into r3}
\end{align*}
\]

Around:
**INTEGER UNIT**

**620**  
**USER MODE**

**FORMS**  
cmpd crfD,rA,rB = cmp crfD,1,rA,rB

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Res.</th>
<th>crfD</th>
<th>0</th>
<th>1</th>
<th>A</th>
<th>B</th>
<th>0000000000</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
</tr>
<tr>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
</tr>
<tr>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
</tr>
<tr>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

if rA < rB then c ← 0b100  
else if rA > rB then c ← 0b010  
else c ← 0b001  
CR[4*crfD-4*crfD+3] ← c || XER[SO]

**DESCRIPTION**

The cmpd (compare doubleword) instruction compares rA with rB, treating the operands as signed integers. The result of the comparison is placed into CR Field crfD; if crfD is not specified, CR0 is used. This instruction is a 64-bit simplified form of the cmp instruction.

**REGISTERS AFFECTED**

CR Field specifically controlled by operand crfD[LT,GT,EQ,SO]
**cmpdi**

**COMPARE DOUBLEWORD REGISTER WITH SIGNED IMMEDIATE**

**FORMS**

\[ \text{cmpdi crfD,rA,SIMM} = \text{cmpi crfD,1,rA,SIMM} \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>0x0b</th>
<th>crfD</th>
<th>0</th>
<th>1</th>
<th>A</th>
<th>SIMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Res.</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[
\begin{align*}
\text{if } rA & < \text{EXTS(SIMM) then } c \leftarrow 0b100 \\
\text{else if } a & > \text{EXTS(SIMM) then } c \leftarrow 0b010 \\
\text{else } & c \leftarrow 0b001 \\
\text{CR}[4*\text{crfD-4*crfD+3}] & \leftarrow c || \text{XER}[SO]
\end{align*}
\]

**DESCRIPTION**

The **cmpdi** (compare doubleword immediate) instruction compares rA with the sign-extended value of the SIMM field, treating both operands as signed integers. The result of the comparison is placed into CR field crfD. If the crfD field is not specified in the instruction, CR0 is used. This instruction is a simplified form of the cmpi instruction.

**REGISTERS AFFECTED**

CR Field specifically controlled by operand crfD[LT,GT,EQ,SO]

**EXAMPLE**

\[
\begin{align*}
\text{if (dwordA < 0x10)} & \quad \text{\slash / globally declared 64-bit value} \\
\text{dwordA = 0x10;} & \\
\text{; Assumes:} & \\
\text{; r3 = dwordA, 64-bit value} & \\
\text{; cmpdi } & \\
\text{bgt & Around} & \quad \text{\slash / compare to 0x10} \\
\text{li & r3, 0x10} & \quad \text{\slash / greater than? jump around} \\
\text{; less than. do assignment}
\end{align*}
\]
INTEGER UNIT
601/603/604/620
User Mode

FORMS
cmpi  crfD,L,rA,SIMM

SIMPLIFIED MNEMONIC
cmpdi  rA,value  =  cmpi  0,1,rA,value
cmpwi  cr3,rA,value  =  cmpi  3,0,rA,value

BIT DEFINITION

<table>
<thead>
<tr>
<th></th>
<th>crfD</th>
<th>0</th>
<th>L</th>
<th>A</th>
<th>SIMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0b</td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

PSEUDO CODE
if rA < EXTS(SIMM) then c 0b100
else if rA > EXTS(SIMM) then c 0b010
else c 0b001
CR[4*crfD-4*crfD+3] c || XER[S0]

DESCRIPTION
The cmpi (compare immediate) instruction compares rA with a 16-bit signed value, treating both operands as signed integers. The 16-bit immediate value, SIMM, is sign-extended to 32 bits before the operation. The result of the comparison is placed into CR field crfD. If the crfD field is not specified in the instruction, CR0 is used.

On the 620, L controls whether the instruction operands are treated as 64- or 32-bit operands, with L = 0 indicating 32-bit operands and L = 1 indicating 64-bit operands. L is ignored on the 601, 603, and 604.

REGISTERS AFFECTED
CR Field specifically controlled by operand crfD[LT,GT,EQ,S0]

EXAMPLE
if (longA < 0x10)  // globally declared long
    longA = 0x10;

; Assumes:
; r3 = dwordA, 32-bit value
;
cmpi  0x0, 0x0, r3, 0x10  ; 32-bit compare: r3 to 0x10
bgt   Around             ; greater than? jump around
li    r3, 0x10           ; less than, do assignment

Around:
**cmpl**

**COMPARE REGISTERS UNSIGNED**

### FORMS

```
cmpl crfD,L,rA,rB
```

### BIT DEFINITION

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>crfD</th>
<th>L</th>
<th>A</th>
<th>B</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

### PSEUDO CODE

```plaintext
if rA < rB then c ← 0b100
else if rA > rB then c ← 0b010
else c ← 0b001
CR[4*crfD-4*crfD+3] ← c || XER[SO]
```

### DESCRIPTION

The `cmpl` (compare logical) instruction compares rA with rB, treating their contents as unsigned values. The result of the comparison is placed into CR Field crfD. If the crfD Field is not specified in the instruction, CR0 is used.

On the 620, L controls whether the instruction operands are treated as 64- or 32-bit operands, with L = 0 indicating 32-bit operands and L = 1 indicating 64-bit operands. L is ignored on the 601, 603, and 604.

**REGISTERS AFFECTED**

CR field specifically controlled by operand crfD[LT,GT,EQ,SO]
**Integer Unit**

**620**

**User Mode**

**cmpld**

**Compare Doubleword**

**Registers Unsigned**

**Forms**

\[
\text{cmpld crfD}, \text{rA}, \text{rB} \equiv \text{cmpl crfD}, 1, \text{rA}, \text{rB}
\]

**Bit Definition**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>A</th>
<th>B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td>cf</td>
<td>0</td>
<td>1</td>
<td></td>
<td>Reserved</td>
</tr>
</tbody>
</table>

**Pseudo Code**

\[
\begin{align*}
\text{if } \text{rA} &< \text{rB} \text{ then } c \leftarrow 0b100 \\
\text{else if } \text{rA} &> \text{rB} \text{ then } c \leftarrow 0b010 \\
\text{else } c &\leftarrow 0b001 \\
\text{CR}[4*\text{crfD}-4*\text{crfD}+3] &\leftarrow c \| \text{XER}[SO]
\end{align*}
\]

**Description**

The **cmpld** (compare logical doubleword) instruction compares rA with rB, treating their contents as unsigned 64-bit values. The result of the comparison is placed into CR Field crfD. If the crfD field is not specified in the instruction, CR0 is used. This instruction is a simplified form of the **cmp** instruction.

**Registers Affected**

CR Field specifically controlled by operand crfD[LT,GT,EQ,SO]
**cmpldi**

**COMPARE DOUBLEWORD REGISTER WITH UNSIGNED IMMEDIATE**

**FORMS**

\[ \text{cmpldi } crfD,rA,UIMM = \text{ cmpli } crfD,1,rA,UIMM \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Res.</th>
<th>crfD</th>
<th>0</th>
<th>1</th>
<th>A</th>
<th>UIMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
</tr>
<tr>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

if \( rA < U \) \((48) \) UIMM \) then \( c \leftarrow 0b100 \)
else if \( rA > U \) \((48) \) UIMM \) then \( c \leftarrow 0b010 \)
else \( c \leftarrow 0b001 \)
CR\([4*crfD-4*crfD+3]\] \( \leftarrow c \| \text{ XER[SO]} \)

**DESCRIPTION**

The cmpldi (compare logical doubleword immediate) instruction compares \( rA \) with a 16-bit unsigned value. The 16-bit immediate value, UIMM, is zero-extended to 64 bits before the operation. The result of the comparison is placed into CR Field crfD. If the crfD field is not specified in the instruction, CR0 is used. This instruction is a simplified form of the cmpli instruction.

**REGISTERS AFFECTED**

CR Field specifically controlled by operand crfD\([LT,GT,EQ,SO]\]
### INTEGER UNIT

**601/603/604/620**

**User Mode**

#### cmpli

**COMPARE REGISTER WITH UNIGNED IMMEDIATE**

**Forms**

cmpli crfD,L,rA,UIMM

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Res.</th>
<th>0x0a</th>
<th>crfD</th>
<th>O</th>
<th>L</th>
<th>A</th>
<th>UIMM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

**Pseudo Code**

if rA < U (0x0000 || UIMM) then c ← 0b100
else if rA > U (0x0000 || UIMM) then c ← 0b010
else c ← 0b001
CR[4*crfD-4*crfD+3] ← c || XER[SO]

**Description**

The cmpli (compare logical immediate) instruction compares rA with a 16-bit unsigned value. The 16-bit immediate value, UIMM, is zero-extended before the operation. The result of the comparison is placed into CR Field crfD. If the crfD field is not specified in the instruction, CR0 is used. This instruction is a simplified form of the cmpli instruction.

On the 620, L controls whether the instruction operands are treated as 64- or 32-bit operands, with L = 0 indicating 32-bit operands and L = 1 indicating 64-bit operands. L is ignored on the 601, 603, and 604.

**Registers Affected**

CR Field specifically controlled by operand crfD[LT,GT,EQ,SO]
**cmplw**

**COMPARE REGISTERS**

**UNSIGNED**

**FORMS**

\[
\text{cmplw} \ crfD,rA,rB = \ \text{cmpl} \ crfD,0,rA,rB
\]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td>crfD</td>
</tr>
<tr>
<td>0</td>
<td>L</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>0x20</td>
<td>0</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

if \( rA < U rB \) then \( c \leftarrow 0b100 \)
else if \( rA > U rB \) then \( c \leftarrow 0b010 \)
else \( c \leftarrow 0b001 \)

\( \text{CR}[4*\text{crfD}-4*\text{crfD}+3] \leftarrow c \mid | \ \text{XER}[SO] \)

**DESCRIPTION**

The `cmplw` (compare logical word) instruction compares \( rA \) with \( rB \), treating their contents as unsigned values. The result of the comparison is placed into CR Field crfD. If the crfD field is not specified in the instruction, CR0 is used. This instruction is a simplified form of the `cmpl` instruction.

**REGISTERS AFFECTED**

CR Field specifically controlled by operand crfD[LT,GT,EQ,SO]
**INTEGER Unit**

**601/603/604/620 User Mode**

**cmplw**

**COMPARE REGISTER WITH UNSIGNED IMMEDIATE**

**FORMS**

\[
\text{cmplw}i \text{ crfD,rA,UIMM} \equiv \text{cmpli crfD,0,rA,UIMM}
\]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Res.</th>
<th>crfD</th>
<th>0</th>
<th>0</th>
<th>A</th>
<th>UIMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oxa</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[
\begin{align*}
\text{if} \ rA < U (0x0000 || UIMM) & \text{ then } c \leftarrow 0b100 \\
\text{else if} \ rA > U (0x0000 || UIMM) & \text{ then } c \leftarrow 0b010 \\
\text{else} & \quad c \leftarrow 0b001 \\
\text{CR}[4*\text{crfD}-4*\text{crfD}+3] & \leftarrow c || \text{XER}[SO]
\end{align*}
\]

**DESCRIPTION**

The cmplw (compare logical word immediate) instruction compares rA with a 16-bit unsigned value. The 16-bit immediate value, UIMM, is zero-extended before the operation. The result of the comparison is placed into CR Field crfD. If the crfD field is not specified in the instruction, CR0 is used. This instruction is a simplified form of the cmpli instruction.

**REGISTERS Affected**

CR Field specifically controlled by operand crfD[L,T,G,E,Q,S]
**INTERRUPT UNIT**

**601/603/604/620**

**USER MODE**

---

**cmpw**

**COMPARE**

**REGISTERS SIGNED**

---

**FORMS**

\[
\text{cmpw crfD}, rA, rB = \text{cmp crfD}, 0, rA, rB
\]

---

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Res.</th>
<th>crfD</th>
<th>0</th>
<th>0</th>
<th>A</th>
<th>B</th>
<th>0000000000</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td></td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>B</td>
<td>00000000000</td>
<td>0</td>
</tr>
</tbody>
</table>

---

**PSEUDO CODE**

\[
\text{if } rA < rB \text{ then } c \leftarrow 0b100 \\
\text{else if } rA > rB \text{ then } c \leftarrow 0b010 \\
\text{else } c \leftarrow 0b001 \\
\text{CR[4*crfD-4*crfD+3]} \leftarrow c \mid | \ XER[SO]
\]

---

**DESCRIPTION**

The `cmpw` (compare word) instruction compares `rA` with `rB`, treating the operands as signed integers. The result of the comparison is placed into condition register field `crfD`; if `crfD` is not specified, `CR0` is used. This instruction is a simplified form of the `cmp` instruction.

---

**REGISTERS AFFECTED**

CR Field specifically controlled by operand `crfD[LT,GT,EQ,SO]`

---

**EXAMPLE**

\[
\text{if } (\text{longA} < \text{longB}) \quad // \text{both are globally declared longs} \\
\text{longA} = \text{longB};
\]

; Assumes:
; r3 = longA, 32-bit value
; r4 = longB

; 
\text{cmpw r3, r4} \quad ; \text{longA greater than longB?}
\text{bgt Around} \quad ; \text{yes, jump around assignment}
\text{mr r3, r4} \quad ; \text{place contents of r4 into r3}

Around:
**INTEGER UNIT**

**601/603/604/620 USER MODE**

**FORMS**

\[
\text{cmpwi crfD, rA, SIMM} \quad \equiv \quad \text{cmpi crfD, 0, rA, SIMM}
\]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Res.</th>
<th>crfD</th>
<th>0</th>
<th>0</th>
<th>A</th>
<th>SIMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0b</td>
<td>0x0b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0x0b</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

```plaintext
a \leftarrow rA
if rA < \text{EXTS(SIMM)} then c \leftarrow 0b100
else if rA > \text{EXTS(SIMM)} then c \leftarrow 0b010
else \quad c \leftarrow 0b001
\text{CR[4*crfD-4*crfD+3]} \leftarrow \text{c \mid \mid XER[SO]}
```

**DESCRIPTION**

The **cmpwi** (compare word immediate) instruction compares rA with a 16-bit signed value, treating both operands as signed integers. The 16-bit immediate value, SIMM, is sign-extended to 32 bits before the operation. The result of the comparison is placed into CR field crfD. If the crfD field is not specified in the instruction, CR0 is used. This instruction is a simplified form of the **cmpl** instruction.

**REGISTERS AFFECTED**

CR Field specifically controlled by operand crfD[LT,GT,EQ,SO]

**EXAMPLE**

```plaintext
if \text{(word1 == 0x10)}
\quad \text{word1 = (word1 \ll 1);} ;\text{Assumes:}
\quad r3 = 32\text{-bit word1}

\quad \text{cmpwi r3,0x10} ;\text{(IF) - compare immediate: r3 == 0x10?}
\quad \text{bne Aroundl} ;\text{branch if not equal to Around1}
\quad \text{slwi r3,r3,1} ;\text{(STMT1) shift left immediate 1 bit}
\quad \text{Around1:} ;\text{execution continues as normal}
```
**COUNT LEADING ZERO BITS**

**OF DOUBLEWORD REGISTER**

**FORMS**

<table>
<thead>
<tr>
<th></th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>cntlzd</td>
<td>rA,rS</td>
</tr>
<tr>
<td>cntlzd.</td>
<td>rA,rS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>A</th>
<th>00000</th>
<th>0x3a</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

**PSEUDO CODE**

\[
\begin{align*}
  \text{n} & \leftarrow 0 \\
  \text{do while } n < 64 \\
  & \text{if } \text{rS}[n] = 1 \text{ then leave} \\
  & \text{n} \leftarrow n + 1 \\
  & \text{rA} \leftarrow n
\end{align*}
\]

**DESCRIPTION**

The cntlzd (count leading zeroes) instruction counts the number of consecutive bits starting at bit 0 of register rS that contain zeroes. The result is placed into rA and ranges from 0 to 64.

**REGISTERS AFFECTED**

CR0[L,T,G,E,Q,S] (Rc = 1)

Note: If Rc = 1, then LT is cleared in the CR0 Field.
### INTEGER UNIT

**601/603/604/620**

**User Mode**

#### FORMS

<table>
<thead>
<tr>
<th>cntlzwx</th>
<th>RA, RS</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>cntlzwx</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cntlzwx</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

#### BIT DEFINITION

<table>
<thead>
<tr>
<th>0x1f</th>
<th>S</th>
<th>A</th>
<th>00000</th>
<th>0x1a</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### PSEUDO CODE

```plaintext
n ← 0
do while n < 32
    if rS[n] = 1 then leave
    n ← n + 1
rA ← n
```

#### DESCRIPTION

The `cntlzwx` (count leading zeroes word) instruction counts the number of consecutive bits starting at bit 0 of register rS that contain zeroes. The result is placed into rA and ranges from 0 to 32.

#### REGISTERS AFFECTED

CR0[LT, GT, EQ, SO] (if Rc = 1)

Note: If Rc = 1, then LT is cleared in the CR0 Field.
**crand**

**AND CONDITION REGISTER FIELDS**

**FORMS**

\[ \text{crand \ crbD,crbA,crbB} \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x13</th>
<th>crbD</th>
<th>crbA</th>
<th>crbB</th>
<th>0x101</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[ \text{CR[crbD]} \leftarrow \text{CR[crbA] \& CR[crbB]} \]

**DESCRIPTION**

The **crand** (condition register AND) instruction ANDs the bit in the condition register (CR) specified by \( \text{crbA} \) with the bit in the condition register specified by \( \text{crbB} \). The result is placed into the condition register bit specified by \( \text{crbD} \).

**REGISTERS AFFECTED**

CR: bit specified by operand \( \text{crbD} \)
**INTEGER UNIT**

**601/603/604/620**

**User Mode**

**crandc**

**AND CONDITION REGISTER FIELD WITH COMPLEMENT**

**FORMS**

\[ \text{crandc} \quad \text{crbD,crbA,crbB} \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>crbD</th>
<th>crbA</th>
<th>crbB</th>
<th>0x81</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[ \text{CR}[,\text{crbD}] \leftarrow \text{CR}[,\text{crbA}] \land \text{NOT} (\text{CR}[\text{crbB}]) \]

**DESCRIPTION**

The `crandc` (condition register AND with complement) instruction ANDs the bit in the condition register (CR) specified by `crbA` with the complement of the bit in the condition register specified by `crbB`. The result is placed into the condition register bit specified by `crbD`.

**REGISTERS Affected**

CR: bit specified by operand `crbD`
**creqv**

**XOR CONDITION REGISTER**

**FIELDS THEN COMPLEMENT**

**FOMRS**

creqv crbD,crbA,crbB

**SIMPLIFIED MNEMONICS**

crset crbD = creqv crbD,crbD,crbD

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x13</th>
<th>crbD</th>
<th>crbA</th>
<th>crbB</th>
<th>0x121</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**


**DESCRIPTION**

The creqv (condition register equivalent) instruction XORs the bit in the condition register specified by crbA with the bit in the condition register specified by crbB. The result is complemented and placed into the condition register bit specified by crbD.

**REGISTERS AFFECTED**

CR: bit specified by operand crbD
**INTEGER UNIT**

**601/603/604/620**

**User Mode**

**FORMS**

\[ \text{crnand crbD,crbA,crbB} \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x13</th>
<th>crbD</th>
<th>crbA</th>
<th>crbB</th>
<th>0xe1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[ \text{CR[crbD]} \leftarrow \text{NOT (CR[crbA] \& CR[crbB])} \]

**DESCRIPTION**

The **crnand** (condition register NAND) instruction ANDs the bit in the condition register (CR) specified by crbA with the bit in the condition register specified by crbB. The result is complemented and placed into the condition register bit specified by crbD.

**REGISTERS AFFECTED**

CR: bit specified by operand crbD
**crnor**

**OR CONDITION REGISTER**

**FIELDS THEN COMPLEMENT**

**FORMS**

\[
\text{crnor} \quad \text{crbD,crbA,crbB}
\]

**Simplified Mnemonics**

\[
\text{crnot} \quad \text{crbD,crbA} = \text{crnor} \quad \text{crbD,crbA,crbA}
\]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>crbD</th>
<th>crbA</th>
<th>crbB</th>
<th>0x21</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

**Pseudo Code**

\[
\text{CR[crbD]} \leftarrow \text{NOT (CR[crbA] | CR[crbB])}
\]

**Description**

The `crnor` (condition register NOR) instruction ORs the bit in the condition register (CR) specified by `crbA` with the bit in the condition register specified by `crbB`. The result is complemented and placed into the condition register bit specified by `crbD`.

**Registers Affected**

CR: bit specified by operand `crbD`
**INTEGER UNIT**

**601/603/604/620**

**User Mode**

** Forms**

\[ \text{cror~crbD}, \text{crbA}, \text{crbB} \]

**Simplified Mnemonics**

\[ \text{crmove~crbD}, \text{crbA} \equiv \text{cror~crbD}, \text{crbA}, \text{crbA} \]

**Bit Definition**

<table>
<thead>
<tr>
<th>0x13</th>
<th>crbD</th>
<th>crbA</th>
<th>crbB</th>
<th>0x1c1</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pseudo Code**

\[ \text{CR}[\text{crbD}] \leftarrow (\text{CR}[\text{crbA}] \mid \text{CR}[\text{crbB}]) \]

**Description**

The \( \text{cror} \) (condition register OR) instruction ORs the bit in the condition register (CR) specified by \( \text{crbA} \) with the bit in the condition register specified by \( \text{crbB} \). The result is placed into the condition register bit specified by \( \text{crbD} \).

**Registers Affected**

CR: bit specified by operand \( \text{crbD} \)
**crorc**

**OR CONDITION REGISTER FIELD WITH COMPLEMENT**

**FORMS**

```
crorc      crbD, crbA, crbB
```

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>crbD</th>
<th>crbA</th>
<th>crbB</th>
<th>0x1a1</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-31</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

```
CR[crbD] ← CR[crbA] | NOT(CR[crbB])
```

**DESCRIPTION**

The `crorc` (condition register OR with complement) instruction ORs the bit in the condition register (CR) specified by `crbA` with the complement of the bit in the condition register specified by `crbB`. The result is placed into the condition register bit specified by `crbD`.

**REGISTERS AFFECTED**

CR: bit specified by operand `crbD`
**INTEGER UNIT**

**601/603/604/620**

**User Mode**

**crxor**

**XOR Condition**

**Register Fields**

**Forms**

`crxor crbD,crbA,crbB`

**Simplified Mnemonics**

`crclr crbD` == `crxor crbD,crbD,crbD`

**Bit Definition**

<table>
<thead>
<tr>
<th>0x13</th>
<th>crbD</th>
<th>crbA</th>
<th>crbB</th>
<th>0xc1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pseudo Code**

`CR[crbD] ← (CR[crbA] | CR[crbB])`

**Description**

The `crxor` (condition register XOR) instruction XORs the bit in the condition register specified by `crbA` with the bit in the condition register specified by `crbB`. The result is placed into the condition register bit specified by `crbD`.

**Registers Affected**

CR: bit specified by operand `crbD`
dcbf
FLUSH DATA CACHE BLOCK

FORMS
dcbf rA,rB

BIT DEFINITION

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>A</th>
<th>B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td>0000</td>
<td></td>
<td></td>
<td>0x56</td>
</tr>
</tbody>
</table>

PSEUDO CODE
EA is the sum (rA|0) + rB.

DESCRIPTION
The dcbf (data cache block flush) instruction is a user-level cache management instruction. The action taken depends on the memory mode associated with the target address, and on the state of the block. The list below describes the action taken for the various cases. The actions described will be executed regardless of whether the page or block containing the addressed byte is designated as write-through or if it is in caching-inhibited or caching-allowed mode.

COHERENCY REQUIRED (WIM = xx1)
- Unmodified Block — Invalidates copies of the block in the caches of all processors.
- Modified Block — Copies the block to memory. Invalidates copies of the block in the caches of all processors.
- Absent Block — If modified copies of the block are in the caches of other processors, causes them to be copied to memory and invalidated. If unmodified copies are in the caches of other processors, causes those copies to be invalidated.

COHERENCY NOT REQUIRED (WIM = xx0)
- Unmodified Block — Invalidates the block in the processor’s cache.
- Modified Block — Copies the block to memory. Invalidates the block in the processor’s cache.

This instruction operates as a load from the addressed byte with respect to address translation and protection. If EA specifies a memory address for which SR[T] = 1, the instruction is treated as a no-op.

REGISTERS AFFECTED
None
The `dcbi` (data cache block invalidate) instruction is a supervisor-level cache management instruction. The action taken depends on the memory mode associated with the target address, and on the state of the block. The list below describes the action to take if the block containing the byte addressed by the EA is or is not in the cache. The actions described will be executed regardless of whether the page containing the addressed byte is in caching-inhibited or caching-allowed mode. This is a supervisor-level instruction.

**Coherency Required (WIM = XX1)**
- Unmodified Block — Invalidates copies of the block in the caches of all processors.
- Modified Block — Invalidates copies of the block in the caches of all processors. (Discards the modified contents.)
- Absent Block — If copies are in the caches of any other processors, causes those copies to be invalidated.

**Coherency Not Required (WIM = XX0)**
- Unmodified Block — Invalidates the block in the local cache.
- Modified Block — Invalidates the block in the local cache. (Discards the modified contents.)
- Absent Block — Does nothing.

This instruction operates as a store to the addressed byte with respect to address translation and protection. The reference and change bits are modified appropriately. If EA specifies a memory address for which `SR[T] = 1`, the instruction is treated as a no-op.

**Registers Affected**
None
**dcbst**

**STORE DATA CACHE BLOCK**

**FORMS**

dcbst rA, rB

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>_reserved</th>
<th>A</th>
<th>B</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td>00000</td>
<td></td>
<td></td>
<td></td>
<td>Res.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

EA is the sum (rA|0) + rB.

**DESCRIPTION**

The dcbst (data cache block store) instruction is a user-level cache management instruction.

If the block containing the byte addressed by the EA is in coherence required mode, and a block containing the byte addresses by the EA is in the data cache of any processor and has been modified, the writing of it to main memory is initiated. If the block containing the byte addressed by the EA is in coherence-not-required mode, and a block containing the byte addressed by the EA is in the data cache of this processor and has been modified, the writing of it to main memory is initiated.

The function of this instruction is independent of the write-through and caching-inhibited/allowed modes of the page or block containing the byte addressed by EA. This instruction operates as a load from the addressed byte with respect to address translation and protection. If EA specifies a memory address for which SR[T] = 1, the instruction is treated as a no-op.

**REGISTERS AFFECTED**

None
dcbt  TOUCH DATA CACHE BLOCK

**Forms**
dcbt    rA,rB

**Bit Definition**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>A</th>
<th>B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td>00000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pseudo Code**

EA is the sum (rA|0) + rB.

**Description**

The dcbt (data cache block touch) instruction is a user-level cache management instruction.

This instruction is a hint that performance will probably be improved if the block containing the byte addressed by the EA is fetched into the data cache, because the program will probably soon load from the addressed byte. Executing dcbt does not cause any exceptions to be invoked.

This instruction operates as a load from the addressed byte with respect to address translation and protection except that no exception occurs in the case of a translation fault or protection violations. If EA specifies a memory address for which SR[T] = 1, the instruction is treated as a no-op.

The purpose of this instruction is to allow the program to request a cache block fetch before it is actually needed by the program. The program can later perform loads to put data into registers. However, the processor is not obliged to load the addressed block into the data cache. If the sector is loaded, it will be either in shared state or exclusive unmodified state.

**Registers Affected**

None
**dcbtst**

**TOUCH FOR STORE ON DATA CACHE BLOCK**

**FORMS**

\[ \text{dcbtst} \quad rA,rB \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x1f</th>
<th>00000</th>
<th>A</th>
<th>B</th>
<th>0xf6</th>
<th>0</th>
</tr>
</thead>
</table>

**PSEUDO CODE**

EA is the sum \((rA|0) + rB\).

**DESCRIPTION**

The dcbtst (data cache block touch for store) instruction is a user-level cache management instruction.

This instruction is a hint that performance will probably be improved if the block containing the byte addressed by the EA is fetched into the data cache, because the program will probably soon store into the addressed byte. Executing dcbtst does not cause any exceptions to be invoked.

This instruction operates as a load from the addressed byte with respect to address translation and protection except that no exception occurs in the case of a translation fault or protection violations. Since dcbtst does not modify memory, it not recorded as a store (the change (C) bit is not modified in the page tables). If EA specifies a memory address for which SR[T] = 1, the instruction is treated as a no-op.

The purpose of this instruction is to allow the program to schedule a cache block fetch before it is actually needed by the program. The program can later perform stores to put data into memory. However, the processor is not obliged to load the addressed block into the data cache.

**REGISTERS AFFECTED**

None
**Integer Unit**

**601/603/604/620 User Mode**

**Forms**

dcbz rA,rB

**Bit Definition**

<table>
<thead>
<tr>
<th></th>
<th>0x1f</th>
<th>00000</th>
<th>A</th>
<th>B</th>
<th>0x3f6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
</tr>
</tbody>
</table>

**Pseudo Code**

EA is the sum (rA|0) + rB.

**Description**

The dcbz (data cache block set to zero) instruction is a user-level cache management instruction. The dcbz instruction executes as follows.

- If the block containing the byte addressed by EA is in the data cache, all bytes of the block are cleared to zero.
- If the block containing the byte addressed by EA is not in the data cache and the corresponding page is caching-allowed, the block is allocated in the data cache without fetching the block from main memory, and all bytes of the block are set to zero.
- If the page containing the byte addressed by EA is caching-inhibited or write-through mode, either all bytes of main memory that correspond to the addressed cache block are cleared or the alignment exception handler is invoked and the handler should clear all the bytes of memory that correspond to the addressed block.
- If the block containing the byte addressed by EA is in coherency-required mode, and the block exists in the data cache(s) of any other processor(s), it is kept coherent in those caches.

This instruction is treated as a store to the addressed byte with respect to address translation and protection. If EA specifies a memory address for which SR[T] = 1, the instruction is treated as a no-op.
**divdx**

**DIVIDE DOUBLEWORD REGISTERS**

**FORMS**

<table>
<thead>
<tr>
<th>Form</th>
<th>Description</th>
<th>OE</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>divd</td>
<td>rD, rA, rB</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>divd.</td>
<td>rD, rA, rB</td>
<td>0 1</td>
<td></td>
</tr>
<tr>
<td>divdo</td>
<td>rD, rA, rB</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>divdo.</td>
<td>rD, rA, rB</td>
<td>1 1</td>
<td></td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Bit Definition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td>D A B OE 0x1e9</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31</td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

```plaintext
rD ← rA / rB
```

**DESCRIPTION**

The **divd** (divide doubleword) instruction calculates rB divided by rA and places the 64-bit quotient into destination register rD. The remainder is not supplied as a result.

Both the operands and the quotient are interpreted as 64-bit signed integers. The quotient is the unique signed integer that satisfies the equation dividend = (quotient + divisor) + r, where 0 ≤ r < |divisor| if the dividend is non-negative, and -|divisor| < r ≤ 0 if the dividend is negative.

If an attempt is made to perform the following divisions:

- (0x8000 0000 0000 0000 -1)
- (<anything> / 0)

the contents of rD are undefined, as are the contents of the LT, GT and EQ bits of the CR0 Field (if Rc = 1). In this case, if OE = 1 then OV is set.

The 64-bit signed remainder of dividing rA by rB can be computed as follows, except in the case that rA = -2^63 and rB = -1:

```plaintext
divd rD, rA, rB ;rD = quotient
mulld rD, rD, rB ;rD = quotient * divisor
subf rD, rD, rA ; rD = remainder
```

**REGISTERS AFFECTED**

- CR0[LT, GT, EQ, SO] (if Rc = 1)
- XER[SO, OV] (if OE = 1)

The setting of the affected bits in the XER is mode-independent, and reflects overflow of the 64-bit result.
**INTEGER UNIT**

620

**USER MODE**

**FORMS**

<table>
<thead>
<tr>
<th></th>
<th>Registers</th>
<th>OE</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>divdu</td>
<td>rD,rA,rB</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>divdu.</td>
<td>rD,rA,rB</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>divduo</td>
<td>rD,rA,rB</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>divduo.</td>
<td>rD,rA,rB</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x1f</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>OE</th>
<th>0x1c9</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

rD ← rA / rB

**DESCRIPTION**

The **divdu** (divide doubleword unsigned) instruction calculates rB divided by rA and places the 64-bit quotient into destination register rD. The remainder is not supplied as a result.

Both the operands and the quotient are interpreted as unsigned integers. The first 3 bits of CR0 Field are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies the equation dividend = (quotient + divisor) + r, where 0 ≤ r < divisor.

If an attempt is made to divide by zero, the contents of rD are undefined, as are the contents of the LT, GT and EQ bits of the CR0 Field (if Rc = 1). In this case, if OE = 1 then OV is set. The 64-bit unsigned remainder of dividing rA by rB can be computed as follows:

```
divdu  rD,rA,rB : rD = quotient
mullid rD,rD,rB : rD = quotient * divisor
subf  rD,rD,rA : rD = remainder
```

**REGISTERS AFFECTED**

- CR0[LT,GT,EQ,SO] (if Rc = 1)
- XER[SO,OV] (if OE = 1)

The setting of the affected bits in the XER is mode-independent, and reflects overflow of the 64-bit result.
**PowerPC Programming for Intel Programmers**

**divwx**

**DIVIDE REGISTERS**

**FORMS**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>OE</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>divw</td>
<td>rD,rA,rB</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>divw.</td>
<td>rD,rA,rB</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>divwo</td>
<td>rD,rA,rB</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>divwo.</td>
<td>rD,rA,rB</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

- **0x1f**

<table>
<thead>
<tr>
<th>Bit</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>OE</th>
<th>0x1eb</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[
{\text{rD}} \leftarrow \frac{{\text{rA}}}{{\text{rB}}}
\]

**DESCRIPTION**

The **divw** (divide word) instruction calculates the quotient \( \frac{rA}{rB} \) and places the result into \( rD \). The remainder is not supplied as a result. Both operands are interpreted as signed integers. The quotient is the unique signed integer that satisfies the following:

\[
\text{dividend} = (\text{quotient} \times \text{divisor}) + r
\]

where \( 0 \leq r < |\text{divisor}| \) if the dividend is non-negative, and

\[
-|\text{divisor}| < r \leq 0
\]

if the dividend is negative.

If an attempt is made to perform the following divisions:

- \((0x8000 0000 0000 0000 / -1)\) or
- \((<\text{anything}> / 0)\)

the contents of \( rD \) are undefined, as are the contents of the LT, GT, and EQ bits of the CR0 field. In these cases, if \( OE = 1 \) then \( OV \) is set to 1.

Note: The 32-bit signed remainder of dividing \( rA \) by \( rB \) can be computed as follows, except in the case that \( rA = -2^{31} \) and \( rB = -1 \).

\[
\text{divww r}D,\text{r}A,\text{r}B ; \text{r}D=\text{quotient}
\]

\[
\text{mul} \quad \text{r}D,\text{r}D,\text{r}B ; \text{r}D=\text{quotient} \times \text{divisor}
\]

\[
\text{subf} \quad \text{r}D,\text{r}D,\text{r}A ; \text{r}D=\text{remainder}
\]

**REGISTERS AFFECTED**

- **CR0[LT,GT,EQ,SO]** (if \( Rc = 1 \))
- **XER[SO,OV]** (if \( OE = 1 \)) (mode independent)
EXAMPLE

longA = (longB / 0x29); // globally declared longs

; Assumes:
; r5 = address of longA
; r6 = address of longB

lwz r3, 0(r6) ; longB value at address in r4
li r4, 0x29 ; put 0x29 in r4
divw r3, r3, r4 ; perform division
stw r3, 0(r5) ; store results at address of longA
**divwux**

**DIVIDE REGISTERS UNSIGNED**

**FORMS**

<table>
<thead>
<tr>
<th></th>
<th>OE</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>divwu</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>divwu.</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>divwuo</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>divwuo.</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>OE</th>
<th>0x1cb</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

```
rD ← rA / rB
```

**DESCRIPTION**

The `divwu` (divide word unsigned) instruction calculates the quotient `rA/rB` and places the result into `rD`. The remainder is not supplied as a result. Both operands are interpreted as unsigned integers. The quotient is the unique signed integer that satisfies the following:

\[
\text{dividend} = (\text{quotient} \times \text{divisor}) + r
\]

where \(0 \leq r < \text{divisor}\) if the dividend is non-negative, and \(-0 \leq r < \text{divisor}\).

If an attempt is made to divide by zero, then the contents of `rD` are undefined as are (if `Rc = 1`) the contents of the LT, GT, and EQ bits of the CR0 field. In these cases, if `OE = 1` then `OV` is set to 1.

The 32-bit signed remainder of dividing `rA` by `rB` can be computed as follows, except in the case that `rA = -2^{31}` and `rB = -1`.

```
divwu rD,rA,rB ; rD=quotient
mull rD,rD,rB ; rD=quotient \times \text{divisor}
subf rD,rD,rA ; rD=remainder
```

**REGISTERS AFFECTED**

- CR0[LT,GT,EQ,SO] (if `Rc = 1`)
- XER[SO,OV] (if `OE = 1`)

---

**INTEGER UNIT**

601/603/604/620

**USER MODE**
**INTEGER Unit**

*601/603/604/620*  
**User Mode**

**eciwx**  
**Input Word Using External Control and Indexed Addressing**

**Forms**  
eciwx  
rd, ra, rb

**Bit Definition**

<table>
<thead>
<tr>
<th></th>
<th>0x1f</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>0x136</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pseudo Code**

if ra = r0 then  
    b ← 0  
else  
    b ← ra

if EAR[E] = 1 then  
    paddr ← address translation of EA  
    send load request for paddr to device identified by EAR[RID]  
    rd ← word from device

else  
    DSISR[11] ← 1  
generate data access exception

**Description**

The eciwx (external control input word indexed) instruction allows the system designer to map special devices in an alternative way. The MMU translation of the EA is not used to select the special device, as it is used in load/store instructions. Rather, it is used as an address operand that is passed to the device over the address bus. Four pins (the burst and size pins on the 60x bus) are used to select the device; these four pins output the 4-bit resource ID (RID) field that is located in the EAR register. The eciwx instruction also loads a word from the data bus that is output by the special device.

The eciwx instruction and the EAR register can be very efficient when mapping special devices such as graphics devices that use addresses as pointers. The effective address (EA) is the sum (ra[10] + rb).

A load word request for the physical address corresponding to the EA is sent to the device identified by EAR[RID], bypassing the cache. The word returned by the device is placed in rd. EAR[E] must equal one; if it does not, a DSI exception is generated. The EA must be a multiple of 4 (32-bit aligned) or one of the following will occur:

- An alignment exception is generated.
- A data access exception is generated (possible only if EAR[E] = 0).
- The results are boundedly undefined.
The `eciwx` instruction is supported for effective addresses that reference ordinary segments (that is, SR[T] = 0), and for EAs mapped by the DBAT registers. If the EA references a direct-store segment (SR[T] = 1), either a DSI exception occurs or the results are boundedly undefined. If this instruction is executed when MSR[DR] = 0 (physical addressing mode), the results are boundedly undefined.

This instruction is defined as an optional instruction by the PowerPC architecture, and may not be available in all PowerPC implementations. Additionally, this instruction is treated as a load from the addressed byte with respect to address translation, memory protection and referenced and changed recording, and the ordering done by `eielo`.

**REGISTERS AFFECTED**

None
INTEGER UNIT
601/603/604/620
USER MODE

FORMS
ecowx rS,rA,rB

BIT DEFINITION

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11| 12| 13| 14| 15| 16| 17| 18| 19| 20| 21| 22| 23| 24| 25| 26| 27| 28| 29| 30| 31|
| ox1f | S | A | B | 0x1b6 | Res. |

PSEUDO CODE
if rA = r0 then b ← 0 else b ← rA
EA ← b + rB
if EAR[E] = 1 then
  paddr ← address translation of EA
  send store request for paddr to device identified by EAR[RID]
  send rS to device
else
  DSISR[11] ← 1
  generate data access exception

DESCRIPTION
The ecowx (external control out word indexed) instruction and the EAR register can be very efficient when mapping special devices such as graphics devices that use addresses as pointers. The EA is the sum (rA|00) + rB. A store word request for the physical address corresponding to EA and the contents of the low-order 32 bits of rS are sent to the device identified by EAR[RID], bypassing the cache. EAR[E] must be 1; if it is not, a DSI exception is generated.

EA must be a multiple of 4 (32-bit aligned) or one of the following will occur:

■ An alignment exception is generated.
■ A DSI exception is generated (possible only if EAR[E] = 0).
■ The results are boundedly undefined.

The ecowx instruction is supported for effective addresses that reference ordinary segments (that is, SR[T] = 0), and for EAs mapped by the DBAT registers. If EA refers to a direct-store segment (SR[T] = 1), either a DSI exception occurs or the results are boundedly undefined. If this instruction is executed when MSR[DR]-0 (physical addressing mode), the results are boundedly undefined.

This instruction is defined as an optional instruction by the PowerPC architecture, and may not be available in all PowerPC implementations. Additionally, this instruction is treated as a store from the addressed byte with respect to address translation, memory protection and reference and changed recording, and the ordering done by elieio.

REGISTERS AFFECTED
None
**eieio**

**ENFORCE IN-ORDER EXECUTION OF I/O OPERATIONS**

**FORMS**
eieio

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>Reserved</th>
<th>Reserved</th>
<th>Reserved</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>0x356</td>
</tr>
</tbody>
</table>

**DESCRIPTION**

The eieio (enforce in order execution of I/O) instruction provides an ordering function for the effects of load and store instruction executed by a processor. These loads and stores are divided into two sets, which are ordered separately. The memory access caused by a dcbz instruction is ordered like a store. The two sets follow:

1. Loads and stores to memory that is both caching-inhibited and guarded, and stores to memory that is write-through required.

   The eieio instruction controls the order in which the accesses are performed in main memory. It ensures that all applicable memory accesses previously initiated by the processor have completed with respect to main memory before any applicable memory accesses subsequently initiated by the processor access main memory. It acts like a barrier that flows through the memory queues and to main memory, preventing the reordering of memory accesses across the barrier. No ordering is done for dcbz if the instruction causes the system alignment error handler to be invoked.

   All accesses in this set are ordered as a single set — that is, there is not one order for load and stores to caching-inhibited and guarded memory and another order for stores to write-through memory.

2. Stores to memory that have all of the following attributes — caching-allowed, write-through not required, and memory-coherency required.

   The eieio instruction controls the order in which the accesses are performed with respect to coherent memory. It ensures that all applicable stores previously initiated by the processor have completed with respect to coherent memory before any applicable stores subsequently initiated by the processor complete with respect to coherent memory.
The \texttt{eieio} instruction does not affect the order of other data accesses. With the exception of \texttt{dcbz}, \texttt{eieio} does not affect the order of cache operations (whether caused explicitly by execution of a cache management instruction, or implicitly by the cache coherency mechanism). For more information refer to Chapter 6, "The PowerPC Instruction Set." The \texttt{eieio} instruction does not affect the order of accesses in one set with respect to accesses in the other set.

The \texttt{eieio} instruction may complete before previously initiated memory accesses have been performed with respect to main memory or coherent memory as appropriate.

The \texttt{eieio} instruction is intended for use in managing shared data structures, in doing memory-mapped I/O, and in preventing load/store combining operations in main memory. For the first use, the shared data structure and the lock that protects it must be altered only by stores that are in the same set (1 or 2, see previous discussion). For the second use, \texttt{eieio} can be thought of as placing a barrier into the stream of memory accesses issued by a processor, such that any given memory access appears to be on the same side of the barrier to both the processor and the I/O device.

Note that the \texttt{eieio} instruction does not connect hardware considerations to it such as multiprocessor implementations that send an \texttt{eieio} address-only broadcast (useful in some designs). For example, if a design has an external buffer that re-orders loads and stores for better bus-efficiency, the \texttt{eieio} broadcast signals to that buffer that previous loads/stores (marked caching-inhibited, guarded, or write-through required) must complete before any following loads/stores (marked caching-inhibited, guarded, or write-through required).

\textbf{601 Processor}

The synchronize (\texttt{sync}) and the enforce in-order execution of I/O (\texttt{eieio}) instructions are handled in the same manner internally to the 601. These instructions delay execution of subsequent instruction until all previous instructions have completed to the point that they can no longer cause an exception, all previous memory accesses are performed globally, and the \texttt{sync} or \texttt{eieio} operation is broadcast onto the 601 bus interface. \texttt{eieio} orders loads/stores to caching inhibited memory and stores to write-through required memory.

\textbf{Registers Affected}

None
**eqvX**

PERFORM LOGICAL EQUIVALENCE OF REGISTERS

**FORMS**

<table>
<thead>
<tr>
<th></th>
<th>rA, rS, rB</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>eqv</td>
<td>rA, rS, rB</td>
<td>0</td>
</tr>
<tr>
<td>eqv.</td>
<td>rA, rS, rB</td>
<td>1</td>
</tr>
</tbody>
</table>

**Bit Definition**

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>Ox11c</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pseudo Code**

rA ← NOT (rS XOR rB)

**Description**

The eqv (equivalent) instruction performs a bitwise equivalence of rS with rB and places the result in destination register rA. Equivalence is implemented as the complement of the XOR function.

**Registers Affected**

CR0[LT,GT,EQ,SO] (if Rc = 1)
**INTEGER UNIT**

**620**

**User Mode**

**FORMS**

\[ \text{extldi} \ r_A, r_S, n, b \ (n > 0) \quad \Rightarrow \quad \text{rldicr} \ r_A, r_S, b, n-1 \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x1e</th>
<th>S</th>
<th>A</th>
<th>b*</th>
<th>n-1</th>
<th>0x01</th>
<th>b*</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

*Note: This is a split field.*

**PSEUDO CODE**

\[
\begin{align*}
\text{r} & \leftarrow \text{ROT}_{L}(r_S, b) \\
\text{m} & \leftarrow \text{MASK}(0, n-1) \\
\text{r}_A & \leftarrow (r \& m) \\
\end{align*}
\]

**DESCRIPTION**

The `extldi` (extract and left justify doubleword immediate) instruction extracts an \( n \)-bit field that starts at \( r_S[b] \), clearing the remaining 64-\( n \) bits. The result is left-justified (padded on the right with zeroes) in destination register \( r_A \). This instruction is a simplified form of the `rldicr` instruction.

**REGISTERS AFFECTED**

CR0[LT,GT,EQ,SO] (if \( \text{Rc} = 1 \))

**EXAMPLE**

; Assumes: we want to extract the second highest-order byte of \( r_3 \) and
; left justify the byte into register \( r_4 \).
; The high-order byte starts at bit position zero.
;
; starting value of \( r_3 = 0x01234567abcdef0 \)
; (note: second highest-order byte = 0x23)

\[ \text{extldi} \ r_4, r_3, 8, 4 \quad ; \text{here, n=8 (8-bits per byte)} \]
\[ \text{and b=4, the starting bit of the byte} \]

; The instruction operation proceeds as follows:
;
; step 1: rotate \( r_3 \) left by \( b \) bits = 4 bits
; result: \( r_3 = 0x2345678abcdef001 \)
; step 2: generate mask from 0 through \( n-1 = 1 \) bits from 0-7
; result: mask = 0xff00000000000000
; step 3: AND rotated data with mask
; result: \( r_3 = 0x2300000000000000 \)
; step 4: store result into destination register \( r_4 \)
; result: \( r_4 = 0x2300000000000000 \) (note: value is left justified)
**extlwi**

**Extract bitfield from word and left justify**

**Forms**

\[
\text{extlwi } rA, rS, n, b \quad (n > 0) = \text{ rlwinm } rA, rS, b, 0, n-1
\]

**Bit Definition**

**Pseudo Code**

\[
\begin{align*}
    r &\leftarrow \text{ROTL}(rS, b) \\
    m &\leftarrow \text{MASK}(0, n-1) \\
    rA &\leftarrow (r \& m)
\end{align*}
\]

**Description**

<table>
<thead>
<tr>
<th>(0x15)</th>
<th>(S)</th>
<th>(A)</th>
<th>(b)</th>
<th>(0)</th>
<th>(n-1)</th>
<th>(Rc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The **extlwi** (extract and left justify word immediate) instruction extracts an \(n\)-bit field that starts at \(rS[b]\), clearing the remaining 32-\(n\) bits. The result is left-justified in destination register \(rA\). This instruction is a simplified form of the **rlwinm** instruction.

**Registers Affected**

\(\text{CR0}[LT, GT, EQ, SO]\) (if \(Rc = 1\))

**Example**

Assumes: we want to extract the low-order byte of \(r3\) and left justify the byte into register \(r4\).

- The low-order byte starts at bit position 24.
- Starting value of \(r3 = 0x\text{abcd1234}\)
  - (Note: low-order byte = \(0x34\))

\[
\text{extlwi } r4, r3, 8, 24 : \text{ here, } n=8 \text{ (8-bits per byte) and } b=24, \text{ the starting bit of the byte}
\]

The instruction operation proceeds as follows:

- step 1: rotate \(r3\) left by \(b\) bits = 24 bits
- result: \(r3 = 0x\text{34abcd12}\)
- step 2: generate mask from 0 through \(n-1\) = 1 bits from 0-7
- result: mask = \(0xff000000\)
- step 3: AND rotated data with mask
- result: \(r3 = 0x\text{34000000}\)
- step 4: store result into destination register \(r4\)
- result: \(r4 = 0x\text{34000000}\) (note: value is left justified)
**INTEGER UNIT**

**620**

**User Mode**

**FORMS**

extrdi rA, rS, n, b (n > 0) = rldicl rA, rS, b + n, 64 - n

**BIT DEFINITION**

```
<table>
<thead>
<tr>
<th>0x1e</th>
<th>S</th>
<th>A</th>
<th>b + n*</th>
<th>64 - n</th>
<th>0x00</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
```

*Note: This is a split field.

**PSEUDO CODE**

r ← ROTL(rS, b + n)
m ← MASK(64 - n, 63)
rA ← (r & m)

**DESCRIPTION**

The extrdi (extract and right justify doubleword immediate) instruction extracts an n-bit field that starts at rS[b], clearing the remaining 64 - n bits. The result is right-justified in destination register rA. This instruction is a simplified form of the rldicl instruction.

**REGISTERS AFFECTED**

CR0[LT, GT, EQ, SO] (if Rc = 1)

**EXAMPLE**

```c
; Assumes: we want to extract the next to highest-order byte of r3 and right-justify the byte into register r4.
; The next to highest-order byte starts at bit position 8.
;
; starting value of r3 = 0x012345678abcdef0
; (note: next to highest-order byte = 0x23)
extrdi r4, r3, 8, 8 ; here, n=8 (8-bits per byte)
; and b=8, the starting bit of the byte
; The instruction operation proceeds as follows:
;
; step 1: rotate r3 left by b+n bits = 16 bits
; result: r3 = 0x45678abedef00123
;
; step 2: generate mask from 64-n through (63)= 1 bits from 56-63
; result: mask = 0x00000000000000ff
;
; step 3: AND rotated data with mask
; result: r3 = 0x0000000000000023
;
; step 4: store result into destination register r4
; result: r4 = 0x0000000000000023 (note: value is right justified)
```
extrwi
EXTRACT AND RIGHT JUSTIFY IMMEDIATE

**FORMS**
extrwi rA,rS,n,b (n > 0) = rlwinm rA,rS, b+n, 32-n, 31

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x15</th>
<th>S</th>
<th>A</th>
<th>b + n*</th>
<th>32 – n</th>
<th>31</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Note: This is a split field.

**PSEUDO CODE**
r ← ROTL(rS, b+n)
m ← MASK(32-n, 31)
rA ← (r & m)

**DESCRIPTION**
The extrwi (extract and right justify immediate) instruction extracts an n-bit field that starts at rS[b], clearing the remaining 32-n bits. The result is right-justified in destination register rA. This instruction is a simplified form of the rlwinm instruction.

**REGISTERS AFFECTED**
CR0[LT,GT,LE,SE] (if Rc = 1)

**EXAMPLE**
: Assumes: we want to extract the next to lowest-order byte of r3 and right-justify the byte into register r4.
: The next to lowest-order byte starts at bit position 16.
: starting value of r3 = 0xabcd1234
: (note: low-order byte = 0x12)
extrwi r4, r3, 8, 16 ; here, n=8 (8-bits per byte)
: and b=16, the starting bit of the byte
: The instruction operation proceeds as follows:
: step 1: rotate r3 left by b+n bits = 24 bits
: result: r3 = 0x34abcd12
: step 2: generate mask from 32-n through 31 = 1 bits from 24-31
: result: mask = 0x000000ff
: step 3: AND rotated data with mask
: result: r3 = 0x00000012
: step 4: store result into destination register r4
: result: r4 = 0x00000012 (note: value is right-justified)
**INTEGER UNIT**

**601/603/604/620**

**User Mode**

**FORMS**

<table>
<thead>
<tr>
<th>extsb</th>
<th>rA, rS</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>extsb.</td>
<td>rA, rS</td>
<td>1</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>A</th>
<th>00000</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td>S</td>
<td>A</td>
<td>00000</td>
<td>0x3b</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

rA[24-31] ← rS[24-31]
rA[0-23] ← rS[24]

**DESCRIPTION**

The `extsb` (extend sign byte) instruction places the contents of the low-order 8 bits of rS into the low-order 8 bits of rA. The sign bit of the byte (bit 24) is copied into bits 0 – rA. On 64-bit implementations, bit rS[56] is placed into rA[0-55].

**REGISTERS AFFECTED**

CR0[LT,GT,EQ,SO] (if Rc = 1)
**extshx**

**SIGN-EXTEND HALF-WORD**

**FORMS**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>extsh</td>
<td>rA,rS</td>
<td>0</td>
</tr>
<tr>
<td>extsh.</td>
<td>rA,rS</td>
<td>1</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ox1f</td>
<td>S</td>
<td>A</td>
<td>00000</td>
<td>Ox39a</td>
<td>Rc</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

rA[16-31] ← rS[16-31]

rA[0-15] ← rS[16]

**DESCRIPTION**

The `extsh` (extend sign half-word) instruction places the contents the low-order 16 bits of rS into the low-order 16 bits of rA. The sign bit of the half-word (bit 16) is copied into bits 0 – rA. On 64-bit implementations, bit rS[48] is placed into rA[0-47].

**REGISTERS AFFECTED**

CR0[LT,GT,EQ,SO] (if Rc = 1)
**INTEGER UNIT**

620
**User Mode**

**extsw**

**SIGN-EXTEND WORD**

**FORMS**

<table>
<thead>
<tr>
<th>extsw</th>
<th>rA,rS</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>extsw</td>
<td>rA,rS</td>
<td>1</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x1f</th>
<th>S</th>
<th>A</th>
<th>00000</th>
<th>0x3da</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

rA[32-63] ← rS[32-63]
rA[0-31] ← rS[32]

**DESCRIPTION**

The extsw (extend sign word) instruction places the contents of the low-order 32 bits of rS into the low-order 32 bits of rA. Bit 32 of rS is copied into bits 0 – rA.

**REGISTERS AFFECTED**

CR0[LT,GT,EQ,SO] (if Rc = 1)
**fabsx**

**FLOATING-POINT ABSOLUTE VALUE**

**FORMS**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>frD,frB</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>fabs</td>
<td>frD,frB</td>
<td>0</td>
</tr>
<tr>
<td>fabs.</td>
<td>frD,frB</td>
<td>1</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>00000</th>
<th>B</th>
<th>0x108</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

frD ← frB
frD[0] ← 0

**DESCRIPTION**

The `fabs` (floating-point absolute value) instruction copies frB into frD and clears bit frD[0].

**REGISTERS AFFECTED**

CR1[FX,FEX,VX,OX] (if Rc = 1)
**Floating-Point Unit**

**601/603/604/620 User Mode**

**Forms**

| Fradd  | frD, frA, frB | 0  |
| Fradd. | FrD, frA, frB | 1  |

**Bit Definition**

<table>
<thead>
<tr>
<th>0x3f</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>00000</th>
<th>0x15</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

**Pseudo Code**

```c
frTemp ← frA + frB
if frTemp[MSb] ≠ 1
   frD ← NORMALIZE(frTemp)
else
   frD ← TARGET_PRECISION(frTemp)
```

**Description**

The `fadd` (floating-point add) instruction calculates the sum of frA and frB. If the most significant bit of the resulting significant is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field FPSCR[RN] and placed into frD.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added algebraically to form an intermediate sum. All 53 bits in the significand as well as all three guard bits (G, R, and X) enter into the computation.

If a carry occurs, the sum’s significand is shifted right one bit position and the exponent is increased by one. FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

**Registers Affected**

- CR1[FX, FEX, VX, OX] (if Rc = 1)
- FPSCR[FPRF, FR, FI, FX, OX, UX, XX, VXSnan, VXISI]
Example

float2 := (float1 + 10.2);  // globally declared floats

; Assumes:
; r3 = contains address of float1
; r4 = contains address of float2
; r5 = contains address of constant data

lfs  f2, @r3 ; fp value in float1
lf d f1, @r5 ; load float double
fadd f2, f2, f1 ; do add: float1+10.2
lfs  f1, @r4 ; fp value in float2
fsubs f1, f1, f2 ; subtract previous result
stfs f1, @r4 ; float store back to float2
**FLOATING-POINT UNIT**

**601/603/604/620**

**USER MODE**

**FORMS**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>fadds</td>
<td>frD, frA, frB</td>
<td>0</td>
</tr>
<tr>
<td>fadd</td>
<td>frD, frA, frB</td>
<td>1</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>00000</th>
<th>0x15</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

```plaintext
frTemp ← frA + frB
if frTemp[MSb] != 1
   frD ← NORMALIZE(frTemp)
else
   frD ← TARGET_PRECISION(frTemp)
```

**DESCRIPTION**

The fadds (floating-point add single precision) instruction calculates the sum of frA and frB. If the most significant bit of the resultant significant is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into frD.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increases by one for each bit shifted, until the two exponents are equal. The two significands are then added algebraically to form an intermediate sum. All 53 bits in the significand as well as all three guard bits (G, R, and X) enter into the computation.

If a carry occurs, the sum's significand is shifted right one bit position and the exponent is increased by one. FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

**REGISTERS AFFECTED**

- CR1[FX,FEX,VX,OX] (if Rc = 1)
- FPSCR[FPRF,FR,FI,FX,OX,UX,XX,VXSNAN,VXISI]
**fcid**

CONVERT INTEGER DOUBLE-WORD TO FLOATING-POINT

**FORMS**

\[
\begin{align*}
\text{Rc} & \quad \text{fcfid frD,frB} \\
0 & \quad \text{fcfid frD,frB} \\
1 & \quad \text{fcfid frD,frB}
\end{align*}
\]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>00000</th>
<th>B</th>
<th>0x34e</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3f</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>29</td>
<td>30</td>
<td>31</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

**DESCRIPTION**

The `fcid` (floating convert from integer doubleword) instruction converts the 64-bit signed fixed-point operand in frB to an infinitely precise floating-point integer. If the result of the conversion is already in double-precision range, it is placed into register frD. Otherwise the result of the conversion is rounded to double precision using the rounding mode specified by FPSCR[RN] and placed into register frD.

FPSCR[FPRF] is set to the class and sign of the result. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

**REGISTERS AFFECTED**

- CR1[FX,VX,FEX,OX] (if Rc = 1)
- FPSCR[FPPR,FR,FI,FX,XX]
FLOATING-POINT UNIT
601/603/604/620
User Mode

**FORMS**
fcmpo  cfrD,frA,frB

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>Res.</th>
<th>0x3f</th>
<th>crfD</th>
<th>00</th>
<th>A</th>
<th>B</th>
<th>0x20</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

if frA is a NaN or frB is a NaN then  c ← 0b0001
else if frA < frB then c ← 0b1000
else if frA > frB then c ← 0b0100
else c ← 0b0010

FPCC ← c
CR[4*crfD - 4*crfD+3] ← c

if frA is a SNaN or frB is a SNaN then VXSNAN ← 1
   if VE=0 then VXVC ← 1
else if frA is a QNaN or frB is a QNaN then VXVC ← 1

**DESCRIPTION**
The fcmpo (floating-point compare ordered) instruction compares the floating-point operand in frA to the floating-point operand in frB. The result of the compare is placed into CR Field crfD and the FPCC.

If one of the operands is a NaN, either quiet or signaling, then CR Field crfD and the FPCC are set to reflect unordered. If one of the operands is a signaling NaN, then VSXNAN is set, and if invalid operation is disabled (VE = 0), then VXVC is set. Otherwise, if one of the operands is a QNaN, then VXVC is set.

**REGISTERS AFFECTED**
CR specified by crfD[FPCC,FX,VSXNAN,VXVC]
fcmpu
UNORDERED FLOATING-POINT
COMPARE

FORMS
fcmpu crfD,frA,frB

BIT DEFINITION

<table>
<thead>
<tr>
<th>0x3f</th>
<th>crfD</th>
<th>00</th>
<th>A</th>
<th>B</th>
<th>0x00</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

PSEUDO CODE
if frA is a NaN or

frB is a NaN then

else if frA < frB then

else if frA > frB then

else

FPCC ← c
CR[4*crfD-4 * crfD+3] ← c

if frA is a SNaN or

frB is a SNaN then VSXNAN ← 1

DESCRIPTION
The fcmpu (floating-point compare unordered) instruction compares the floating-point operand in frA to the floating-point operand in frB. The result of the compare is placed into CR Field crfD and the FPCC.

If one of the operands is a NaN, either quiet or signaling, then CR Field crfD and the FPCC are set to reflect unordered. If one of the operands is a signaling NaN, then VSXNAN is set.

REGISTERS AFFECTED
CR specified by crfD[FPCC,FX,VSXNAN,VXVC]
**Floating-Point Unit**

**620**  
**User Mode**

**Convert Floating-Point to Integer Doubleword**

**Fctidx**

**FORMS**

Rc  
fctid frD,frB 0  
fctid. frD,frB 1

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>00000</th>
<th>B</th>
<th></th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**DESCRIPTION**

The fctid (floating convert to integer doubleword) converts the floating-point operand in frB to a 64-bit signed fixed-point integer, using the rounding mode specified by FPSCR[RN], and placed into frD. If the operand in frB is greater than $2^{63} - 1$, then frD is set to 0x7fffffffffffffff. If the operand in frB is less than $-2^{63}$, then frD is set to 0x8000000000000000.

Except for enabled invalid operation exceptions, FPSCR[FPRF] is undefined. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

**REGISTERS AFFECTED**

- CR1[FX,VX,FEX,OX] (if Rc = 1)
- FPSCR[FPRF(undefined),FR,FI,FX,XX,VXSNAN,VXCVI]
fctidzx

**CONVERT FLoATING-POINT TO INTEGER DOUBLEWORD WITH ROUND TOWARDS ZERO**

**FORMS**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Rc</td>
<td>0</td>
</tr>
<tr>
<td>fctidz</td>
<td>frD,frB</td>
</tr>
<tr>
<td>fctidz</td>
<td>frD,frB</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x3f</th>
<th>D</th>
<th>00000</th>
<th>B</th>
<th>0x32f</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**DESCRIPTION**

The fctidz (floating convert to integer doubleword with round towards zero) instruction converts the floating-point operand in frB to a 64-bit signed fixed-point integer, using the round toward zero rounding mode, and placed into frD. If the operand in frB is greater than 2^63 - 1, then frD is set to 0x7fffffffffffffff. If the operand in frB is less than -2^63, then frD is set to 0x8000000000000000.

Except for enabled invalid operation exceptions, FPSCR[FPRF] is undefined. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

**REGISTERS AFFECTED**

- CR1[FX,VX,FEX,OX] (if Rc = 1)
- FPSCR[FPR(undefined),FR,FI,FX,XX,VXSNAN,VXCVI]
The \texttt{fctiw} (floating-point convert to integer word) instruction converts the floating-point operand in register \texttt{frB} to a 32-bit signed integer, using the rounding mode specified by \texttt{FPSCR[RN]}, and places it in bits \texttt{frD[32-63]}. Bits \texttt{frD[0-31]} are undefined. If the operand in \texttt{frB} is greater than $2^{31}-1$, bits 32–63 of \texttt{frD} are set to \texttt{0x80000000}.

Except for trap-enabled invalid operation exceptions, \texttt{FPSCR[FPRF]} is undefined. \texttt{FPSCR[FR]} is set if the result is incremented when rounded. \texttt{FPSCR[FI]} is set if the result.

**REGISTERS AFFECTED**

- \texttt{CR1[FX,FEX,VX,OX]} (if \texttt{Rc = 1})
- \texttt{FPSCR[FPRF(undefined),FR,FI,FX,XX,VXSNAN,VXCVI]}
The `fctiwz` (floating-point convert to integer word with round toward zero) instruction converts the floating-point operand in register `frB` to a 32-bit signed integer, using the rounding mode round toward zero, and places it in bits `frD[32-63]`. Bits `frD[0-31]` are undefined. If the operand in `frB` is greater than $2^{31}-1$, bits 32–63 of `frD` are set to `0x80000000`.

Except for trap-enabled invalid operation exceptions, FPSCR[FPRF] is undefined. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result

**REGISTERS AFFECTED**

- CR1[FX,FEX,VX,OX] (if Rc = 1)
- FPSCR[FPRF(undefined),FR,FI,FX,XX,VXSNAN,VXCVI]
**Floating-Point Unit**

**601/603/604/620 User Mode**

**Forms**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdiv</td>
<td>frD, frA, frB</td>
<td>0</td>
</tr>
<tr>
<td>fdiv.</td>
<td>frD, frA, frB</td>
<td>1</td>
</tr>
</tbody>
</table>

**Bit Definition**

<table>
<thead>
<tr>
<th>D</th>
<th>A</th>
<th>B</th>
<th>00000</th>
<th>0x12</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

**Pseudo Code**

\[ frD \leftarrow frA / frB \]

**Description**

The `fdiv` (floating-point divide) instruction divides the floating-point operand in register `frA` by the floating-point operand in register `frB`. No remainder is preserved. If an operand is a denormalized number, then it is prenormalized before the operation is started. If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field `RN` of the `FPSCR` and placed into `frD`. Floating-point division is based on exponent subtraction and division of the significands.

`FPSCR[FPRF]` is set to the class and sign of the result, except for invalid operation exceptions when `FPSCR[VE] = 1` and zero divide exceptions when `FPSCR[ZE] = 1`.

**Registers Affected**

- CR1[FX, FEX, VX, OX] (if Rc = 1)
- FPSCR[FPRF, FR, FI, FX, OX, UX, XX, ZX, VXSNAN, VXIDI, VXZDZ]
**fdDIVSx**

**DIVIDE FLOATING-POINT**

**REGISTERS SINGLE-PRECISION**

**FORMS**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>fdivs</td>
<td>0</td>
</tr>
<tr>
<td>fdivs.</td>
<td>1</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ox3b</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>00000</td>
<td>0x12</td>
<td>Rc</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

frD ← frA / frB

**DESCRIPTION**

The fdivs (floating-point divide single-precision) instruction divides the floating-point operand in register frA by the floating-point operand in register frB. No remainder is preserved. If an operand is a denormalized number, then it is prenormalized before the operation is started. If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into frD. Floating-point division is based on exponent subtraction and division of the significands.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1 and zero divide exceptions when FPSCR[ZE] = 1.

**REGISTERS AFFECTED**

- CR1[FX,FEX,VX,OX] (if Rc = 1)
- FPSCR[FPRF,FR,FI,FX,OX,UX,XX,ZX,VXSNAN,VXIDI,VXZDZ]

**EXAMPLE**

```plaintext
float1 = 3.1415;  // globally declared floats
float3 /= float1;
```

; Assumes:
; r3 = contains address of float1
; r4 = contains address of float3
; r5 = contains address of constant data

```plaintext
lfs f2, @(r5) ; get 3.1415 from constant data
stfs f2, @(r3) ; do assignment: float1 = 3.1415
lfs f1, @(r4) ; get value of float3 f1
fdDIVS f1, f1, f2 ; do division: float /= float1
stfs f1, @(r4) ; store fp result back to float3 address
```
**Floating-Point Unit**

**601/603/604/620**

**User Mode**

**FORMS**

<table>
<thead>
<tr>
<th>fmadd</th>
<th>frD,frA,frC,frB</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmadd</td>
<td>frD,frA,frC,frB</td>
<td>0</td>
</tr>
<tr>
<td>fmadd.</td>
<td>frD,frA,frC,frB</td>
<td>1</td>
</tr>
</tbody>
</table>

**Bit Definition**

<table>
<thead>
<tr>
<th>Ox3f</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>0x1d</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pseudo Code**

frD ← (frA * frC) + frB

**Description**

The fmadd (floating-point multiply-add) instruction multiplies the floating-point operand in register frA by the floating-point operand in register frC. The floating-point operand in register frB is added to this intermediate result. If an operand is a denormalized number, then it is prenormalized before the operation is started. If the most significant bit of the resulting significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into frD.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

**Registers Affected**

- CR1[FX,FEX,VX,OX] (if Rc = 1)
- FPSCR[FPRF,FR,FI,FX,OX,UX,XX,VXSNAN,VXISI,VXIMZ]
**fmaddsx**

**Floating-point multiply-add single precision**

**Forms**

```
<table>
<thead>
<tr>
<th></th>
<th>FrD, FrA, FrC, FrB</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmadds</td>
<td>Rc = 0</td>
</tr>
<tr>
<td>fmadds</td>
<td>Rc = 1</td>
</tr>
</tbody>
</table>
```

**Bit Definition**

<table>
<thead>
<tr>
<th></th>
<th>0x3b</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>0x1d</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

**Pseudo Code**

```
frD ← (frA * frC) + frB
```

**Description**

The fmadds (floating-point multiply-add single) instruction multiplies the floating-point operand in register frA by the floating-point operand in register frC. The floating-point operand in register frB is added to this intermediate result. If an operand is a denormalized number, then it is prenormalized before the operation is started. If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into frD. FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

**Registers Affected**

- CR1[FX, FEX, VX, OX] (if Rc = 1)
- FPSCR[FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ]
**FLOATING-POINT UNIT**

**601/603/604/620**

**User Mode**

**FORMS**

<table>
<thead>
<tr>
<th>fmr</th>
<th>frD,frB</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>fmr.</td>
<td>frD,frB</td>
<td>1</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x3f</th>
<th>D</th>
<th>00000</th>
<th>B</th>
<th>0x48</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

frD ← frB

**DESCRIPTION**

The fmr (floating-point move register) instruction places the contents of frB into frD.

**REGISTERS AFFECTED**

CR1[FX,FEX,VX,OX] (if Rc = 1)
**fmsub**

**FLOATING-POINT**

**MULTIPLY-SUBTRACT**

**FORMS**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmsub frD,frA,frC,frB 0</td>
<td>The fmsub (floating-point multiply-subtract double-precision) instruction multiplies the floating-point operand in register frA by the floating-point operand in register frC. The floating-point operand in register frB is subtracted from this intermediate value and the result is placed in destination register frD. If an operand is a denormalized number, then it is prenormalized before the operation is started. If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into frD. FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.</td>
</tr>
<tr>
<td>fmsub. frD,frA,frC,frB 1</td>
<td></td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3f</td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>0x1c</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

frD ← (frA * frC) - frB

**DESCRIPTION**

The fmsub (floating-point multiply-subtract double-precision) instruction multiplies the floating-point operand in register frA by the floating-point operand in register frC. The floating-point operand in register frB is subtracted from this intermediate value and the result is placed in destination register frD.

- If an operand is a denormalized number, then it is prenormalized before the operation is started.
- If the most significant bit of the resultant significand is not a one, the result is normalized.
- The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into frD.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

**REGISTERS AFFECTED**

- CR1[FX,FEX,VX,OX] (if Rc = 1)
- FPSCR[FPRF,FR,FI,FX,OX,UX,XX,VXSNAN,VXISI,VXIMZ]
FLOATING-POINT UNIT

601/603/604/620
USER MODE

FORMS

<table>
<thead>
<tr>
<th>Form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmsubs</td>
<td>multiply, subtract single precision</td>
</tr>
<tr>
<td>fmsubsx</td>
<td>multiply, subtract single precision, extended precision</td>
</tr>
</tbody>
</table>

BIT DEFINITION

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>0x3b</td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>Rc</td>
<td></td>
</tr>
</tbody>
</table>

PSEUDO CODE

frD ← (frA * frC) - frB

DESCRIPTION

The fmsubs (floating-point multiply-subtract single) instruction multiplies the floating-point operand in register frA by the floating-point operand in register frC. The floating-point operand in register frB is subtracted from this intermediate result. If an operand is a denormalized number, then it is prenormalized before the operation is started. If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into frD.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

REGISTERS AFFECTED

- CR1[FX,FEX,VX,OX] (if Rc = 1)
- FPSCR[FPRF,FR,FI,FX,OX,UX,XX,VXSNAN,VXISI,VXIMZ]
PowerPC Programming for Intel Programmers

**fmulx**

MULTIPLY FLOATING-POINT

REGISTERS

**FORMS**

<table>
<thead>
<tr>
<th>Form</th>
<th>Description</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmul</td>
<td>frD, frA, frC</td>
<td>0</td>
</tr>
<tr>
<td>fmul&lt;</td>
<td>frD, frA, frC</td>
<td>1</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>RC</td>
</tr>
<tr>
<td>29</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>9</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

frD ← frA * frB

**DESCRIPTION**

The fmul (floating-point multiply) instruction multiplies the floating-point operand in register frA by the floating-point operand in register frC. If an operand is a denormalized number, then it is prenormalized before the operation is started. If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into frD.

Floating-point multiplication is based on exponent addition and multiplication of the significands. FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

**REGISTERS AFFECTED**

- CR1[FX, FEX, VX, OX] (if Rc = 1)
- FPSCR[FPRF, FR, FX, OX, UX, XX, VXSNAN, VXIMZ]

**EXAMPLE**

double1 = 3.1415; // globally declared doubles
double2 *= double1;

: Assumes:
: r3 = contains address of double1
: r4 = contains address of double2
: r5 = contains address of constant data
:
lfs  f1, 0(r5);  ; get 3.1415 const. from r5
stfd f1, 0(r3);  ; do assignment: double1 = 3.1415
lfd  f2, 0(r4);  ; get value of double2 from r4
fmul f1, f1, f2;  ; multiply them: double2 *= double1
stfd f1, 0(r4);  ; store double2 results at address in r4
**Floating-Point Unit**

**601/603/604/620 User Mode**

---

### Forms

- fmuls: \( frD, frA, frC \) 0
- fmuls: \( frD, frA, frC \) 1

---

### Bit Definition

<table>
<thead>
<tr>
<th>0x3b</th>
<th>D</th>
<th>A</th>
<th>00000</th>
<th>C</th>
<th>0x19</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

---

### Pseudo Code

\[
frD \leftarrow frA \times frB
\]

---

### Description

The `fmuls` (floating-point multiply single-precision) instruction multiplies the floating-point operand in register \( frA \) by the floating-point operand in register \( frC \). If an operand is a denormalized number, then it is prenormalized before the operation is started. If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field \( RN \) of the FPSCR and placed into \( frD \).

Floating-point multiplication is based on exponent addition and multiplication of the significands. FPSCR[FPREF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

### Registers Affected

- CR1[FX,FEX,VX,OX] (if \( Rc = 1 \))
- FPSCR[FPREF,FR,FI,FX,OX,UX,XX,VXSNAN,VXIMZ]

### Example

```c
flt2 *= flt1;       // globally defined single precision floating point values
```

:Assumes:
:  r3 = contains address of flt1
:  r4 = contains address of flt2
 :

```c
lfs f1, 0(r3)       ; load fp value from address in r3
lfs f2, 0(r4)       ; load flt2 value from address in r4
fmuls f1, f1, f2    ; perform multiply: flt2 *= flt1
stfs f1, 0(r4)      ; fp store back to flt2 address in r4
```
fnabsx
FLOATING-POINT NEGATIVE
ABSOLUTE VALUE

FORMS

<table>
<thead>
<tr>
<th>fnabs</th>
<th>frD,frB</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>fnabs</td>
<td>frD,frB</td>
<td>0</td>
</tr>
<tr>
<td>fnabs.</td>
<td>frD,frB</td>
<td>1</td>
</tr>
</tbody>
</table>

BIT DEFINITION

<table>
<thead>
<tr>
<th>0x3f</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>D</td>
</tr>
</tbody>
</table>

PSEUDO CODE

frD ← frB
frD[0] = 1

DESCRIPTION

The fnabs (floating-point negative absolute value) instruction copies frB into frD and sets bit frD[0].

REGISTERS AFFECTED

CR1[FX,FEX,VX,OX] (if Rc = 1)
FLOATING-POINT UNIT

601/603/604/620

USER MODE

FORMS

\[
\begin{align*}
\text{fneg} & \quad \text{frD,frB} & & 0 \\
\text{fneg} & \quad \text{frD,frB} & & 1
\end{align*}
\]

BIT DEFINITION

<table>
<thead>
<tr>
<th></th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ox3f</td>
<td>D</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

PSEUDO CODE

\[ \text{frD} \leftarrow -(\text{frB}) \]

DESCRIPTION

The \text{fneg} (floating-point negate) instruction copies frB into frD and complements bit frD[0].

REGISTERS AFFECTED

CR1[FX,FEX,VX,OX] (if Rc = 1)
fnmaddx
FLOAtING-POINT
NEgATIVE mULtiply-ADD

FORMS

<table>
<thead>
<tr>
<th>fnmadd</th>
<th>frD,frA,frC,frB</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>fnmaddx</td>
<td>frD,frA,frC,frB</td>
<td>0</td>
</tr>
</tbody>
</table>

BIT DEFINITION

<table>
<thead>
<tr>
<th>Ox3f</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Ox1f</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>17</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

PSEUDO CODE

frD ← -(frA * frC) + frB

DESCRIPTION

The fnmadd (floating-point negative multiply-add) instruction multiplies the floating-point operand in register frA by the floating-point operand in register frC. The floating-point operand in register frB is added to this intermediate result. If an operand is a denormalized number, then it is prenormalized before the operation is started. If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into frD.

The instruction produces the same result as would be obtained by using the floating-point multiply-add instruction and then negating the result, with the following exceptions:

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception retain the sign bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

REGISTERS AFFECTED

- CR1[FX,FEX,VX,OX] (if Rc = 1)
- FPSCR[FPRF,FR,FI,FX,OX,UX,XX,VXSNAN,VXISI,VXIMZ]
**Floating-Point Unit**

**601/603/604/620 User Mode**

**fnmaddsx**

Floating-point negative multiply-add single-precision

---

**Forms**

<table>
<thead>
<tr>
<th>Form</th>
<th>Description</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>fnmadds frD,frA,frC,frB</td>
<td>Rd</td>
<td>0</td>
</tr>
<tr>
<td>fnmadds. frD,frA,frC,frB</td>
<td>Rd</td>
<td>1</td>
</tr>
</tbody>
</table>

**Bit Definition**

```
<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>0x1f</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Pseudo Code**

```
frD ← -((frA * frC) + frB)
```

**Description**

The fnmadd (floating-point negative multiply-add single-precision) instruction multiplies the floating-point operand in register frA by the floating-point operand in register frC. The floating-point operand in register frB is added to this intermediate result. If an operand is a denormalized number, then it is prenormalized before the operation is started. If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into frD.

The instruction produces the same result as would be obtained by using the floating-point multiply-add instruction and then negating the result, with the following exceptions:

- NaNs propagate with no effect on their sign bit.
- NaNs that are generated as the result of a disabled invalid operation exception retain the sign bit of zero.
- NaNs that are converted to NaNs as the result of a disabled invalid operation exception retain the sign bit of the NaN.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

**Registers Affected**

- CR1[FX,FEX,VX,OX] (if Rc = 1)
- FPSCR[FPRF,FR,FI,FX,OX,UX,XX,VXSNAN,VXISI,VXIMZ]
**fnmsubx**

**FLOATING-POINT NEGATIVE MULTIPLY-SUBTRACT**

**FORMS**

<table>
<thead>
<tr>
<th></th>
<th>frD, frA, frC, frB</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>fnmsub</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>fnmsub.</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>0x3f</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>0x1e</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

frD ← -(frA * frC) - frB

**DESCRIPTION**

The *fnmsub* (floating-point negative multiply-subtract) instruction multiplies the floating-point operand in register frA by the floating-point operand in register frC. The floating-point operand in register frB is subtracted to this intermediate result.

If an operand is a denormalized number, then it is prenormalized before the operation is started. If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into frD.

The instruction produces the same result as would be obtained by using the floating-point multiply-subtract instruction and then negating the result, with the following exceptions:

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception retain the sign bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

**REGISTERS AFFECTED**

- CR1[FX, FEX, VX, OX] (if Rc = 1)
- FPSCR[FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXSI, VXIMZ]
**Floating-Point Unit**

**601/603/604/620**

**User Mode**

**fnmsubs**

**Floating-point negative multiply-subtract single-precision**

**Forms**

<table>
<thead>
<tr>
<th>fnmsubs</th>
<th>frD,frA,frC,frB</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>fnmsubs.</td>
<td>frD,frA,frC,frB</td>
<td>1</td>
</tr>
</tbody>
</table>

**Bit Definition**

<table>
<thead>
<tr>
<th>0x3b</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>0x1e</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>31</td>
</tr>
</tbody>
</table>

**Pseudo Code**

frD ← -((frA * frC) - frB)

**Description**

The fnmsubs (floating-point negative multiply-subtract single-precision) instruction multiplies the floating-point operand in register frA by the floating-point operand in register frC. The floating-point operand in register frB is subtracted to this intermediate result. If an operand is a denormalized number, then it is prenormalized before the operation is started. If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into frD.

The instruction produces the same result as would be obtained by using the floating-point multiply-subtract instruction and then negating the result, with the following exceptions:

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception retain the sign bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

**Registers Affected**

- CR1[FX,FEX,VX,OX] (if Rc = 1)
- FPSCR[FPRF,FR,FI,FX,OX,UX,XX,VXSNAN,VXISI,VXIMZ]
**fresx**

**FLOATING reciprocal ESTIMATE SINGLE-PRECISION**

**FORMS**

<table>
<thead>
<tr>
<th>fres</th>
<th>frD,frB</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>fres</td>
<td>frD,frB</td>
<td>1</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x3b</th>
<th>D</th>
<th>00000</th>
<th>B</th>
<th>00000</th>
<th>0x18</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**DESCRIPTION**

The fres (floating reciprocal estimate single) instruction performs a single-precision estimate of the reciprocal of the floating-point operand in register frB and places the result into frD. The estimate placed into register frD is correct to a precision of one part in 256 — or 8 bits with the remaining bits equal to zero. The function may be expressed as:

\[
\text{ABS}\left(\text{estimate } - \left(\frac{1}{x}\right)\right) \leq \frac{1}{256}
\]

Note that the PowerPC architecture makes no provision for a double-precision version of the fresx instruction. This is because graphics applications (and other performance-critical applications) are expected to need only the single-precision version. This instruction is optional in the PowerPC architecture.

Operation with various special values of the operand is summarized below:

<table>
<thead>
<tr>
<th>Operand</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>-\infty</td>
<td>-0</td>
<td>None</td>
</tr>
<tr>
<td>-0</td>
<td>-\infty*</td>
<td>ZX</td>
</tr>
<tr>
<td>+0</td>
<td>+\infty*</td>
<td>ZX</td>
</tr>
<tr>
<td>+\infty</td>
<td>+0</td>
<td>None</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN**</td>
<td>VXSNAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>

* No result if FPSCR[ZE] = 1.
** No result if FPSCR[VE] = 1.

**REGISTERS AFFECTED**

- CR1[FX,FEX,VX,OX] (if Rc = 1)
- FPSCR[FPRF,FR,FI,FX,OX,UX,XX,VXSNAN]
FLOATING-POINT UNIT

601/603/604/620
User Mode

FORMS

```
frsp   frD,frB  0
frsp.  frD,frB  1
```

BIT DEFINITION

```
<table>
<thead>
<tr>
<th>0x3f</th>
<th>D</th>
<th>00000</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>
```

DESCRIPTION

The frsp (floating-point round to single-precision) instruction rounds frB to single-precision. If it is already in single-precision range, the floating-point operand in register frB is placed into frD. Otherwise the floating-point operand in register frB is rounded to single-precision using the rounding mode specified by FPSCR[RN] and placed into frD. Appendix C discusses floating-point operation on PowerPC processors.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

REGISTERS AFFECTED

- CR1[FX,FEX,VX,OX] (if Rc = 1)
- FPSCR[FPRF,FR,FI,FX,OX,UX,XX,VXSNAN]
The frsqrte (floating-point square root estimate) instruction performs a double-precision estimate of the reciprocal of the square root of the floating-point operand in register frB and places the result into register frD. The estimate placed into register frD is correct to a precision of one part in 32 — or 5 bits with the remaining bits equal to zero. The function may be expressed as:

\[
\text{ABS} \left( \frac{\text{estimate} - \left( \frac{1}{\sqrt{x}} \right)}{\frac{1}{\sqrt{x}}} \right) \leq \frac{1}{32}
\]

No single-precision version of the frsqrte instruction is provided; however, both frB and frD are representable in single-precision format.

Operation with various special values of the operand is summarized below:

<table>
<thead>
<tr>
<th>Operand</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>-(\infty)</td>
<td>QNaN**</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>&lt;0</td>
<td>QNaN**</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>-0</td>
<td>-0*</td>
<td>ZX</td>
</tr>
<tr>
<td>+0</td>
<td>+0*</td>
<td>ZX</td>
</tr>
<tr>
<td>+(\infty)*</td>
<td>+0</td>
<td>None</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN**</td>
<td>VXSNAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>

* No result if FPSCR[ZE] = 1.
** No result if FPSCR[VE] = 1.

REGISTERS AFFECTED

- CR1[FX,FEX,VX,OX] (if Rc = 1)
- FPSCR[FPRF,FR,FI,FX,OX,UX,XX,VXSNAN,VXSI]
**DESCRIPTION**

The `f.sel` (floating select) instruction compares the floating-point operand in register `frA` to the value zero. If the operand is greater than or equal to zero, register `frD` is set to the contents of `frC`. If the operand is less than zero or is a NaN, register `frD` is set to the contents of `frB`. The comparison ignores the sign of zero (that is, +0 and -0 are equivalent).

Care must be taken in using `f.sel` if IEEE compatibility is required, or if the values being tested can be NaNs or infinities. Appendix C discusses floating-point operation on PowerPC processors.

**REGISTERS AFFECTED**

CR1[FX,FEX,VX,OX] (if Rc = 1)
The `fsqrt` (floating-point square root double-precision) instruction calculates the double-precision square root of the floating-point operand in register `frB` and places the result into register `frD`. If the most significant bit of the result is not a one, the result is normalized. The result is rounded to the target precision under control of the setting of `FPSCR[RN]`.

Operation with various special values of the operand is summarized below:

<table>
<thead>
<tr>
<th>Operand</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞</td>
<td>QNaN*</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>&lt;0</td>
<td>QNaN*</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td>None</td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td>None</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN*</td>
<td>VXSNAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>

* No result if `FPSCR[VE] = 1`.

**REGISTERS AFFECTED**

- CR1[FX,FEX,VX,OX] (if Rc = 1)
- FPSCR[FPRF,FR,FI,FX,OX,UX,XX,VXSNAN,VXSI]
FLOATING-POINT
UNIT
603/604/620
USER MODE

FORMS

<table>
<thead>
<tr>
<th>fsqrts</th>
<th>frD,frB</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>fsqrts</td>
<td>frD,frB</td>
<td>1</td>
</tr>
</tbody>
</table>

BIT DEFINITION

<table>
<thead>
<tr>
<th></th>
<th>Reserved</th>
<th></th>
<th>Reserved</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3b</td>
<td>D</td>
<td>00000</td>
<td>B</td>
<td>00000</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

DESCRIPTION

The fsqrts (floating-point square root single-precision) instruction calculates the single-precision square root of the floating-point operand in register frB and places the result into register frD. If the most significant bit of the result is not a one, the result is normalized. The result is rounded to the target precision under control of the setting of FPSCR[RN].

Operation with various special values of the operand is summarized below:

<table>
<thead>
<tr>
<th>Operand</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞</td>
<td>QNaN*</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>&lt;0</td>
<td>QNaN*</td>
<td>VXSQRT</td>
</tr>
<tr>
<td>0</td>
<td>-0</td>
<td>None</td>
</tr>
<tr>
<td>+∞</td>
<td>±∞</td>
<td>None</td>
</tr>
<tr>
<td>NaN</td>
<td>QNaN*</td>
<td>VXSNAN</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>None</td>
</tr>
</tbody>
</table>

* No result if FPSCR[VE] = 1.

REGISTERS AFFECTED

- CR1[FX,FEX,VX,OX] (if Rc = 1)
- FPSCR[FPRF,FR,FI,FX,OX,UX,XX,VXSNAN,VXISI]
### fsub

**SUBTRACT FLOATING-POINT REGISTERS**

#### FORMS

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsub</td>
<td>fsub</td>
<td>0</td>
</tr>
<tr>
<td>fsub</td>
<td>fsub</td>
<td>1</td>
</tr>
</tbody>
</table>

#### BIT DEFINITION

<table>
<thead>
<tr>
<th>Bit</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ox3f</td>
</tr>
<tr>
<td>1</td>
<td>Ox14</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>00000</td>
</tr>
</tbody>
</table>

#### PSEUDO CODE

\[ \text{frD} \leftarrow \text{frA} - \text{frB} \]

#### DESCRIPTION

The `fsub` (floating-point subtract) instruction subtracts the floating-point operand in register `frB` from the floating-point operand in register `frA`. If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field `RN` of the FPSCR and placed into `frD`.

The execution of the floating-point subtract instruction is identical to that of floating-point add with the sign bit (bit 0) of `frB` inverted. FPSCR[FPFRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

#### REGISTERS AFFECTED

- CR1[FX,FEX,VX,OX] (if Rc = 1)
- FPSCR[FPFR,FPIFX,OX,UX,XX,VXSNAN,VXISI]

#### EXAMPLE

```plaintext
float2 -= (float1 + 10.2);    // globally declared floats
Assumes:
: r3 = contains address of float1
: r4 = contains address of float2
: r5 = contains address of constant data
:
lfs f2, 0(r3) ; load fp value in float1 into f2
lf d f1, 0(r5) ; load float double into f1
fadd f2, f2, f1 ; do add: float1+10.2
lfs f1, 0(r4) ; fp value in float2 into f1
fsub f1, f1, f2 ; subtract previous result
stfs f1, 0(r4) ; store back to float2 using address in r4
```
**FLOATING-POINT UNIT**

**User Mode**

**Fprints**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>fsubs</td>
<td>frD,frA,frB</td>
<td>0</td>
</tr>
<tr>
<td>fsubs.</td>
<td>frD,frA,frB</td>
<td>1</td>
</tr>
</tbody>
</table>

**Bit Definition**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ox3b</td>
<td>D, A, B, 0, 1</td>
</tr>
</tbody>
</table>

**Pseudo Code**

frD ← frA - frB

**Description**

The fsubs (floating-point subtract single-precision) instruction subtracts the floating-point operand in register frB from the floating-point operand in register frA. If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into frD.

The execution of the floating-point subtract instruction is identical to that of floating-point add with the sign bit (bit 0) of frB inverted. FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

** Registers Affected**

- CR1[FX,FEX,VX,OX] (if Rc = 1)
- FPSCR[FPRF,FR,F1,FX,OX,UX,XX,VSXSNAN,VXISI]

**Example**

float2 -= (float1 + 10.2);  // globally declared floats

; Assumes:
; r3 = contains address of float1
; r4 = contains address of constant data

1fs f2, 0(r3) ; load fp value in from address of float1 into f2
1fd f1, 0(r4) ; load float double constant data into f1 from r4
fadd f2, f2, f1 ; do add: float1+10.2
1fs f1, 0(r3) ; load fp value in float2 into f1
fsubs f1, f1, f2 ; subtract previous result
stfs f1, 0(r3) ; float store back to float2 to address in r3
icbi
INVALIDATE INSTRUCTION
CACHE BLOCK

FORMS
icbi rA,rB

BIT DEFINITION

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td>00000</td>
<td>A</td>
<td>B</td>
<td>0x3d6</td>
</tr>
</tbody>
</table>

PSEUDO CODE
EA is the sum (rA|0) + rB

DESCRIPTION
The icbi (instruction cache block invalidate) instruction is a user-level cache management instruction.

Since the 601 has a unified cache, it treats the icbi instruction as a no-op, even to the extent of not validating the EA. On the 603, 604, and 620 PowerPC processors, if the block containing the byte addressed by EA is in coherency-required mode, and a block containing the byte addressed by EA is in the instruction cache of any processor, the block is made invalid in all such processors, so that subsequent references cause the block to be refetched.

If the block containing the byte addressed by EA is in coherency-not-required mode, and a block containing the byte addressed by EA is in the instruction cache of this processor, the block is made invalid in this processor, so that subsequent references cause the block to be fetched from main memory or cache.

The function of this instruction is independent of the write-through and write-back modes and caching-inhibited/allowed modes of the block containing the byte addressed by EA.

REGISTERS AFFECTED
None
**INTEGER UNIT**

**601/603/604/620**

**User Mode**

**FORMS**

\[ \text{inslwi } rA, rS, n, b \ (n > 0) = \text{ rlwimi } rA, rS, 32 - b, b, b + n - 1 \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x14</th>
<th>S</th>
<th>A</th>
<th>32 - b</th>
<th>b</th>
<th>b + n - 1</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\( r \leftarrow \text{ROTL}(rS, 32 - b) \)
\( m \leftarrow \text{MASK}(b, b + n - 1) \)
\( rA \leftarrow (r \& m) | (rA \& \text{NOT } m) \)

**DESCRIPTION**

The inslwi (insert from left word immediate) instruction inserts a left-justified \( n \)-bit field from rS into rA, starting at bit position b. This instruction is a simplified form of the rlwimi instruction.

**REGISTERS AFFECTED**

CR0[LT,GT,EQ,SO] (if Rc = 1)

**EXAMPLE**

; Assumes: we want to insert a left-justified byte from r3 into r4 starting at bit position 8.
; starting value of r3 = 0xab000000
; starting value of r4 = 0x11223344

inslwi r4, r3, 8, 8 ; here, n=8 (8-bits per byte) ; and b=8, where we want to put it

; The instruction operation proceeds as follows:
; step 1: rotate r3 left by 32-b bits = 24 bits
; result: r3 = 0x00ab0000
; step 2: generate mask from b through b+n-1 = 1 bits from 8-15
; result: mask = 0x00ff0000
; step 3: AND rotated data with mask
; result: r3 = 0x00ab0000
; step 4: AND r4 with NOT(mask)
; result: r4 = 0x11003344
; step 5: OR results from steps 3 and 4 and place in r4
; result: r4 = 0x11ab3344
insrdi  
**INSERT BIT-FIELD INTO DOUBLEWORD FROM RIGHT**

**FORMS**  
\[ \text{insrdi } rA, rS, n, b = \text{ rldimi } rA, rS, 64-b+n, b \]

**BIT DEFINITION**

```
\begin{array}{cccccccc}
\hline
\text{S} & \text{A} & 64-b+n & b & 0x03 & \text{Rc} \\
0x1e & & & & & \\
\end{array}
```

**PSEUDO CODE**

```
r \leftarrow \text{ROTL}(rS, 64-b+n)
m \leftarrow \text{MASK}(b, (64-b+n))
rA \leftarrow (r \& m) \mid (rA \& \text{NOT} m)
```

**DESCRIPTION**

The `insrdi` (insert from right doubleword immediate) instruction inserts a right-justified \( n \)-bit field from \( rS \) into \( rA \), starting at bit position \( b \). This instruction is a simplified form of the `rldimi` instruction.

**REGISTERS AFFECTED**

\( \text{CR0}[\text{LT}, \text{GT}, \text{EQ}, \text{SO}] \) (if \( \text{Rc} = 1 \))

**EXAMPLE**

```
; Assumes: we want to insert a right-justified half-word from \( r3 \)  
; into \( r4 \) starting at bit position 32.  
;  
; starting value of \( r3 = 0x0000000000005678 \)  
; starting value of \( r4 = 0x1122aabbccddeeff \)  
insrdi \( r4, r3, 16, 32 \) ; \( \text{here, } n=16 \) (16-bits per half-word)  
; and \( b=32 \), where we want to put it  
;  
; The instruction operation proceeds as follows:  
;  
; step 1: rotate \( r3 \) left by \( 64-b+n \) bits = 16 bits  
; result: \( r3 = 0x000000000005678000 \)  
; step 2: generate mask from \( b \) through 63-16 = 32 through 48  
; result: \( \text{mask} = 0x0000000000ffff000 \)  
; step 3: AND rotated data with mask  
; result: \( r3 = 0x000000000005678000 \)  
; step 4: AND \( r4 \) with \( \text{NOT} \)(mask)  
; result: \( r4 = 0x1122aabbccddeeff \)  
; step 5: OR results from steps 3 and 4 and place in \( r4 \)  
; result: \( r4 = 0x1122aabb5678eeff \)
```
**INTEGER UNIT**

**601/603/604/620**  
**User Mode**

**FORMS**

\[
\text{insrwi } rA,rS,n,b \ (n>0) \equiv \ \text{rlwimi } rA,rS, 32-b+n, b, b+n-1
\]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>0x14</th>
<th>S</th>
<th>A</th>
<th>32-b+n</th>
<th>b</th>
<th>b+n-1</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[
\begin{align*}
    r &\leftarrow \text{ROTL}(rS, 32-b+n) \\
    m &\leftarrow \text{MASK}(b, b+n-1) \\
    rA &\leftarrow (r \& m) \mid (rA \& \lnot m)
\end{align*}
\]

**DESCRIPTION**

The insrwi (insert from right word immediate) instruction inserts a right-justified \(n\)-bit field from \(rS\) into \(rA\), starting at bit position \(b\). This instruction is a simplified form of the rlwimi instruction.

**REGISTERS AFFECTED**

\(\text{CR0}[\text{LT,GT,EQ,SO}]\) (if \(\text{Rc} = 1\))

**EXAMPLE**

\[
\begin{align*}
    ; \text{Assumes: we want to insert a right-justified byte from } r3 \\
    ; \hspace{1cm} \text{into } r4 \text{ starting at bit position } 16. \\
    ; \\
    ; \text{starting value of } r3 = 0x000000ff \\
    ; \text{starting value of } r4 = 0x11223344 \\
    \text{insrwi } r4, r3, 8, 16 \quad ; \text{here, } n=8 \ (8\text{-bits per byte}) \\
    ; \hspace{1cm} \text{and } b=16, \text{where we want to put it} \\
    ; \text{The instruction operation proceeds as follows:} \\
    ; \\
    ; \text{step 1: rotate } r3 \text{ left by } 32-b+n \text{ bits = 24 bits} \\
    ; \hspace{1cm} \text{result: } r3 = 0x00ff0000 \\
    ; \text{step 2: generate mask from } b \text{ through } b+n-1 = 1 \text{ bits from 16-23} \\
    ; \hspace{1cm} \text{result: mask } = 0x00ff0000 \\
    ; \text{step 3: AND rotated data with mask} \\
    ; \hspace{1cm} \text{result: } r3 = 0x00ff0000 \\
    ; \text{step 4: AND r4 with NOT(mask)} \\
    ; \hspace{1cm} \text{result: } r4 = 0x11003344 \\
    ; \text{step 5: OR results from steps 3 and 4 and place in } r4 \\
    ; \hspace{1cm} \text{result: } r4 = 0x11ff3344
\end{align*}
\]
The `isync` (instruction synchronize) instruction is context synchronizing. The `isync` instruction waits for all previously dispatched instructions to complete and then discards any fetched instructions, causing subsequent instructions to be fetched (or refetched) from memory and to execute in the context established by the previous instructions. This instruction has no effect on other processors (in a multiprocessor system) or on their caches.

The `isync` instruction causes a refetch serialization that waits for all prior instructions to complete and then executes the next sequential instruction. Execution of subsequent instructions is held until all previous instructions have completed to the point where they can no longer cause an exception and all store queues have completed translation. Any instruction after an `isync` see all effects of prior instructions.

**REGISTER AFFECTED**
None

**EXAMPLE**
See the 601 endian mode switching example in Chapter 3, “Of Eggs and Endians.”
**INTEGER UNIT**

**601/603/604/620**

**USER MODE**

**la**

**LOAD REGISTER WITH ADDRESS**

**FORMS:**

\[
\begin{align*}
\text{la } rD, & \text{d(rA)} = \text{addi } rD, rA,d \\
\text{la } rD, & \text{variable} = \text{addi } rD,rA,\text{variable}
\end{align*}
\]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x0e</th>
<th>D</th>
<th>A</th>
<th>SIMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[
\text{if } rA = r0 \text{ then } rD \leftarrow \text{EXTS(SIMM)} \\
\text{else } rD \leftarrow rA + \text{EXTS(SIMM)}
\]

**DESCRIPTION**

The `la` instruction allows computation of a base-displacement operand; this is useful to obtain the address of a variable specified by name, allowing the compiler/assembler to supply the base register number and compute the displacement. The `la` (load address) instruction is a simplified form of the `addi` instruction.

The second form of the `la` instruction is useful when accessing elements of a data structure using a base offset contained in GPR `rA`. This usage requires further explanation. The variable operand is an immediate value that corresponds to the offset of an element in a data structure. Assume that variable refers to a data element that is located at an offset that is variable bytes from the address in register `rA` and the compiler/assembler has been told to use register `rA` as a base for references to the data structure containing variable. In this case, the second form of the `la` instruction can be used to load the address of variable into destination register `rD`.

**REGISTERS AFFECTED**

None
**lbz**  
**LOAD REGISTER WITH BYTE AND ZERO EXTEND**

**FORMS**

\[ \text{lbz } rD, d(rA) \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x22</th>
<th>D</th>
<th>A</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[
\begin{align*}
\text{if } rA = r0 & \text{ then } b \leftarrow 0 \\
\text{else } & b \leftarrow rA \\
EA & \leftarrow b + \text{EXTS}(d) \\
rD & \leftarrow 0x000000 || \text{MEM}(EA, 1)
\end{align*}
\]

**DESCRIPTION**

The `lbz` (load byte and zero) instruction loads a byte from memory addressed by EA into the lower 8 bits of rD. The effective address (EA) is the sum \((rA0) + d\), where \(d\) is a 16-bit signed value that is sign-extended to 32 bits (64 bits on 64-bit implementations).

On 32-bit implementations, \(rD[24-31]\) is loaded with the byte addressed by the EA. On 64-bit implementations, \(rD[56-63]\) is loaded with the byte addressed by the EA. All higher-order bits are cleared to 0.

**REGISTERS AFFECTED**

None

**EXAMPLE**

```assembly
temp1 = 0;  // temp1 and temp2 are globally defined unsigned
temp2 = temp1;

; Assumes:
; r3 = contains address of temp1
; r5 = contains address of temp2
;
l1       r4, 0   ; load r4 with immediate zero
stb      r4, 0(r3) ; store 0 to temp1; address in r3
lbz      r3, 0(r3) ; get byte value from temp1
stb      r3, 0(r5) ; store byte from r3 into temp2; address in r5
```
**INTEGER UNIT**

601/603/604/620

**USER MODE**

**FORMS**

`lbzu`  `rD, d(rA)`

**BIT DEFINITION**

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**PSEUDO CODE**

```c
EA ← rA + EXTS(d)
rD ← (24)0 || MEM(EA,1)
rA ← EA
```

**DESCRIPTION**

The `lbzu` (load byte and zero with update) instruction loads a byte from memory addressed by EA into the lower 8 bits of rD. The effective address (EA) is the sum (rA0) + d, where d is a 16-bit signed value that is sign-extended to 32 bits (64 bits on 64-bit implementations). The EA is placed into rA.

On 32-bit implementations, rD[24-31] is loaded with the byte addressed by the EA. On 64-bit implementations, rD[56-63] is loaded with the byte addressed by the EA. All higher-order bits are cleared to 0. The PowerPC architecture defines load with update instructions with operand rA = r0 or rA = rD as invalid forms.

**REGISTERS AFFECTED**

None

**EXAMPLE**

```c
for (r=0; r<10; r++)
    byte1[r] = byte2[r];    // globally declared arrays of unsigned chars
```

; Assumes:
; r3 = contains address of byte2 array
; r4 = contains address of byte1 array

; li r5, 0 ; zero r5; used as 'r'
subi r4, r4, 1 ; adjust index for use w/ update
subi r3, r3, 1 ; adjust index for use w/ update

LOOP:
    lbzu r6, 1(r3) ; get byte from r3 and update r3
    addi r5, r5, 1 ; inc r5
    cmpi r5, 10 ; compare r5 to 10
    stbu r6, 1(r4) ; store byte to byte1 array
    blt LOOP ; branch to LOOP if (r5 < 10)
**lbzux**

Load register with byte using indexed addressing, zero extend, with EA update

**Forms**

lbzux rD,rA,rB

**Bit Definition**

<table>
<thead>
<tr>
<th>0x1f</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>0x77</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pseudo Code**

EA ← rA + rB
rD ← 0x000000 || MEM(EA,1)
rA ← EA

**Description**

The **lbzux** (load byte and zero with update indexed) instruction loads a byte from memory addressed by the EA into the lower 8 bits of rD. The effective address (EA) is the sum (rA[0] + rB).

On 32-bit implementations, rD[24-31] is loaded with the byte addressed by the EA. On 64-bit implementations, rD[56-63] is loaded with the byte addressed by the EA. All higher-order bits are cleared to 0. The PowerPC architecture defines load with update instructions with operand rA = r0 or rA = rD as invalid forms.

**Registers Affected**

None
**INTEGER UNIT**

**601/603/604/620**

**USER MODE**

**FORMS**

\[
\text{lbzx } \quad rD, rA, rB
\]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>$0x1f$</th>
<th>$D$</th>
<th>$A$</th>
<th>$B$</th>
<th>$0x57$</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

If $rA = 0$ then $b \leftarrow 0$
else $b \leftarrow rA$

$EA \leftarrow b + rB$

$rD \leftarrow 0x000000 \ || \ \text{MEM}(EA,1)$

**DESCRIPTION**

The `lbzx` (load byte and zero indexed) instruction loads a byte from memory addressed by the EA into the lower 8 bits of $rD$. The effective address (EA) is the sum $(rA0) + rB$.

On 32-bit implementations, $rD[24-31]$ is loaded with the byte addressed by the EA. On 64-bit implementations, $rD[56-63]$ is loaded with the byte addressed by the EA. All higher-order bits are cleared to 0. The PowerPC architecture defines load instructions with operand $rA = r0$ or $rA = rD$ as invalid forms.

**REGISTERS AFFECTED**

None
**Id**

LOAD REGISTER WITH DOUBLEWORD

**FORMS**

Id rD,ds(rA)

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x3a</th>
<th>D</th>
<th>A</th>
<th>ds</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

if rA = r0 then b ← 0
else b ← rA
EA ← b + EXTS(ds || 0b00)
rD ← MEM(EA,8)

**DESCRIPTION**

The Id (load doubleword) instruction loads the doubleword in memory addressed by EA into destination register rD. The effective address (EA) is the sum (rA0) + (ds || 0b00). Note that ds is a 14-bit signed value which is concatenated on the right with 0b00; this 16-bit value is sign-extended to 64 bits.

**REGISTERS AFFECTED**

None
INTEGER UNIT

620
USER MODE

FORMS
idarx rD,rA,rB

BIT DEFINITION

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| Ox1f | D | A | B | Ox54 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |

PSEUDO CODE

if rA = r0 then b ← 0
else b ← rA
EA ← b + rB
RESERVE ← 1
RESERVE_ADDR ← physical_addr(EA)
rD ← MEM(EA,8)

DESCRIPTION

The idarx (load doubleword and reserve indexed) instruction loads the doubleword in memory addressed by EA into destination register rD. The effective address (EA) is the sum (rA10) + rB.

This instruction creates a reservation for use by a store doubleword conditional instruction. An address computed from the EA is associated with the reservation, and replaces any address previously associated with the reservation. EA must be a multiple of eight. If it is not, either the system alignment exception handler is invoked or the results are boundedly undefined. Reservations are discussed in Chapter 6, “The PowerPC Instruction Set.”

REGISTERS AFFECTED

None
ldu
LOAD REGISTER WITH DOUBLE-WORD AND UPDATE EA

FORMS
ldu rD,ds(rA)

BIT DEFINITION

<table>
<thead>
<tr>
<th>0x3a</th>
<th>D</th>
<th>A</th>
<th>ds</th>
<th>0x01</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PSEUDO CODE
EA ← rA + EXTS(ds || 0b00)
rD ← MEM(EA, 8)
rA ← EA

DESCRIPTION
The ldu (load doubleword with update) instruction loads the doubleword in memory addressed by EA into destination register rD. The EA is placed into rA. The effective address (EA) is the sum rA + (ds || 0b00). Note that ds is a 14-bit signed value which is concatenated on the right with 0b00; this 16-bit value is sign-extended to 64-bits. The PowerPC architecture defines load with update instructions with operand rA = r0 or rA = rD as invalid forms.

REGISTERS AFFECTED
None
**INTEGER UNIT**

**620 USER MODE**

**FORMS**

ldux rD,rA,rB

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x1f</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>0x35</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

EA ← rA + rB
rD ← MEM(EA,8)
rA ← EA

**DESCRIPTION**

The ldux (load doubleword with update indexed) instruction loads the doubleword in memory addressed by EA into destination register rD. The effective address (EA) is the sum rA + rB. The EA is placed into rA. The PowerPC architecture defines load with update instructions with operand rA = r0 or rA = rD as invalid forms.

**REGISTERS AFFECTED**

None
**ldx**

**LOAD REGISTER WITH DOUBLE-WORD USING INDEXED ADDRESSING**

**FORMS**

\[ \text{ldx } rD, rA, rB \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x1f</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>0x15</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[
\begin{align*}
\text{if } rA = r0 \text{ then } b &\leftarrow 0 \\
\text{else } b &\leftarrow rA \\
\text{EA} &\leftarrow b + rB \\
rD &\leftarrow \text{MEM(EA,B)}
\end{align*}
\]

**DESCRIPTION**

The `ldx` (load doubleword indexed) instruction loads the doubleword in memory addressed by `EA` into destination register `rD`. The effective address (`EA`) is the sum `(rA|0) + rB`.

**REGISTERS AFFECTED**

None
INTEGER UNIT AND
FLOATING-POINT UNIT

601/603/604/620
User Mode

FORMS

| lfd | frD,d(rA) |

BIT DEFINITION

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>A</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

PSEUDO CODE

if rA = r0 then b ← 0
else b ← rA
EA ← b + EXTS(d)
frD ← MEM(EA,b)

DESCRIPTION

The lfd (load floating-point double-precision) instruction loads a double-precision FP value from memory addressed by the EA into frD. The effective address (EA) is the sum (rA0) + d.

REGISTERS AFFECTED

None

EXAMPLE:

double1 = 3.1415;  // globally declared doubles
double2 *= double1;

; Assumes:
; r3 = contains address of constant FP data
; r4 = contains address of double1
; r5 = contains address of double2
;
lfst f1, 0(r3)  ; get 3.1415 const. data from r3 address
stfd f1, 0(r4)  ; do assignment of double1 = 3.1415
lfd f2, 0(r5)   ; get value of double2 from r5 address
fmul f1, f1, f2 ; multiply them: double2 *= double1
stfd f1, 0(r5)  ; store double results to double2
**Ifdu**

**LOAD FLOATING-POINT REGISTER WITH EA UPDATE**

**FORMS**

Ifdu frD,d(rA)

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>0x33</th>
<th>D</th>
<th>A</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

EA ← rA + EXTS(d)
frD ← MEM(EA,8)

**DESCRIPTION**

The ifdu (load floating-point double-precision) instruction loads a double-precision FP value from memory addressed by EA into frD. The effective address (EA) is the sum (rA + d). The effective address (EA) is placed into rA. The PowerPC architecture defines load with update instructions with operand rA = r0 as invalid.

**REGISTERS AFFECTED**

None
# INTEGER UNIT AND FLOATING-POINT UNIT

## 601/603/604/620 User Mode

### FORMS

```
ifdux frD,rA,rB
```

### BIT DEFINITION

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11| 12| 13| 14| 15| 16| 17| 18| 19| 20| 21| 22| 23| 24| 25| 26| 27| 28| 29| 30| 31|
| Ox1f | D | A | B | 0x277 | Res. |

### PSEUDO CODE

```
EA ← rA + rB
frD ← MEM(EA, B)
rA ← EA
```

### DESCRIPTION

The `ifdux` (load floating-point double-precision with update indexed) instruction loads a double-precision FP value from memory addressed by the EA into frD. The effective address (EA) is the sum (rA0 + rB. The EA is placed into rA. The PowerPC architecture defines load with update instructions with operand rA = r0 as an invalid form.

### REGISTERS AFFECTED

None
**Ifdx**

**Load floating-point register using indexed addressing**

**Forms**
- `Ifdx   fRD,rA,rB`

**Bit Definition**

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>A</th>
<th>B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x1f</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pseudo Code**

if rA = r0 then b ← 0
else b ← rA
EA ← b + rB
frD ← MEM(EA,8)

**Description**

The `Ifdx` (load floating-point double-precision indexed) instruction loads a double-precision FP value from memory addressed by the EA into fRD. The effective address (EA) is the sum (rA|0) + rB.

**Registers Affected**

None
INTEGER UNIT AND FLOATING-POINT UNIT

601/603/604/620

User Mode

**FORMS**

\[ \text{lfs} \quad \text{frD,}d(rA) \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x30</th>
<th>D</th>
<th>A</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[
\begin{align*}
\text{if } rA &= r0 \text{ then } b \leftarrow 0 \\
\text{else } b &\leftarrow rA \\
\text{EA} &\leftarrow b + \text{EXTS}(d) \\
\text{frD} &\leftarrow \text{DOUBLE(MEM(EA,4))}
\end{align*}
\]

**DESCRIPTION**

The `lfs` (load floating-point single-precision) instruction loads a single-precision FP value from memory addressed by the EA, converts the value to double precision, and places the result into frD. The effective address (EA) is the sum \((rA0) + d\), where \(d\) is a 16-bit signed value that is sign-extended to 32-bits (64-bits on 64-bit implementations). Appendix C contains further information on floating-point conversion.

**REGISTERS AFFECTED**

None

**EXAMPLE**

```plaintext
tmpf1 = 3.14157; tmpf2 = 10.5; // globally declared floats
tmpf2 *= tmpf1;

; Assumes:
; r3 = contains address of constant FP data
; r4 = contains address of tmpf1
; r5 = contains address of tmpf2
;
lfs  f1, 0(r3) ; load fp value from offset 0
lfs  f2, 4(r3) ; load fp value from offset 4
stfs f1, 0(r4) ; initialize tmpf1
fmuls f1, f1, f2 ; do multiply
stfs f1, 0(r5) ; save result in tmpf2
```
Ifsu

LOAD FLOATING-POINT
REGISTER USING SINGLE-PRECISION WITH EA UPDATE

FORMS
Ifsu frD,d(rA)

BIT DEFINITION

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>A</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PSEUDO CODE

EA ← rA + EXTS(d)
frD ← DOUBLE(MEM(EA,4))
rA ← EA

DESCRIPTION

The Ifsu (load floating-point single-precision with update) instruction loads a single-precision FP value from memory addressed by the EA, converts the value to double precision, and places the result into frD. The effective address (EA) is the sum (rA|0) + d, where d is a 16-bit signed value that is sign-extended to 32 bits (64 bits on 64-bit implementations). The EA is placed into rA. Appendix C contains further information on floating-point conversion.

REGISTERS AFFECTED
None
**INTEGER UNIT AND FLOATING-POINT UNIT**

**601/603/604/620 User Mode**

**FORMS**
```
lfсуx fрD,rA,rB
```

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>0x237</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

```
EA ← rA + rB
frD ← DOUBLE(MEM(EA, 4))
rA ← EA
```

**DESCRIPTION**

The `lfсуx` (load floating-point single-precision with update indexed) instruction loads a single-precision FP value from memory addressed by the EA, converts the value to double-precision, and places the result into frD. The effective address (EA) is the sum (rA+i) + rB. EA is placed into rA. Appendix C contains further information on floating-point conversion.

**REGISTERS AFFECTED**

None
**lfsx**

**LOAD FLOATING-POINT REGISTER USING SINGLE-PRECISION AND INDEXED ADDRESSING**

**FORMS**

\[ \text{lfsx } frD,rA,rB \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>0x1f</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>0x217</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[
\begin{align*}
\text{if } rA &= r0 \text{ then } b \leftarrow 0 \\
\text{else } b &\leftarrow rA \\
\text{EA} &\leftarrow b + rB \\
frD &\leftarrow \text{DOUBLE(MEM(EA, 4))}
\end{align*}
\]

**DESCRIPTION**

The `lfsx` (load floating-point single-precision indexed) instruction loads a single-precision FP value from memory addressed by the EA, converts the value to double-precision, and places the result into frD. The effective address (EA) is the sum (rA[r0] + rB. Appendix C contains further information on floating-point conversion.

**REGISTERS AFFECTED**

None

**EXAMPLE**

\[
\begin{align*}
\text{for}(r=0; r<10; r++) \\
\text{sf2}[r] &= \text{sf1}[r]; & \quad // \text{sf1, sf2 are arrays of type float}
\end{align*}
\]

; Assumes:
; r3 = contains address of sf2 array
; r5 = contains address of sf1 array
;
li r6, 0 ; zero r6; use as counter
li r4, 0 ; zero r4

LOOP:
lfsx f1, r5, r4 ; load our float value into f1
addi r6, r6, 1 ; inc for-loop counter
cmpwi r6, 10 ; done yet?
stfsx f1, r3, r4 ; store float value into sf2[]
addi r4, r4, 4 ; increment index value
blt LOOP ; if less-than, loop
**INTEGER UNIT**

**601/603/604/620**

**User Mode**

**FORMS**

\[ \text{lha \quad r}_D, d(r_A) \]

**Bit Definition**

<table>
<thead>
<tr>
<th>0x2a</th>
<th>D</th>
<th>A</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

**Pseudo Code**

\[
\begin{align*}
\text{if } r_A = r_0 & \text{ then } b \leftarrow 0 \\
\text{else } & b \leftarrow r_A \\
\text{EA} & \leftarrow b + \text{EXTS}(d) \\
\text{r}_D & \leftarrow \text{EXTS}(\text{MEM(EA,2)})
\end{align*}
\]

**Description**

The \text{lha} (load half-word algebraic) instruction loads a half-word from memory addressed by EA into the lower 16 bits of \( r_D \). The effective address (EA) is the sum \( (rA0) + d \), where \( d \) is a 16-bit signed value that is sign-extended to 32 bits (64 bits on 64-bit implementations).

On 32-bit implementations, \( r_D[16-31] \) is loaded with the half-word addressed by the EA. On 64-bit implementations, \( r_D[48-63] \) is loaded with the half-word addressed by the EA. The remaining higher-order bits of \( r_D \) are filled with a copy of the most significant bit of the loaded half-word.

**Registers Affected**

None

**Example**

\[
\text{tmp\_short1} = 10; \\
\text{tmp\_short2} = \text{tmp\_short1}; // globally declared shorts
\]

; Assumes:
; \( r_3 \) = contains address of \text{tmp\_short1}
; \( r_5 \) = contains address of \text{tmp\_short2}

li \quad r_4, 10 ; load \( r_4 \) with immediate value 10
sth \quad r_4, 0(r_3) ; store 10 into \text{tmp\_short1} using address in \( r_3 \)
lha \quad r_3, 0(r_3) ; get that value back
sth \quad r_3, 0(r_5) ; store value in \text{tmp\_short2} using address in \( r_5 \)
**lhau**

**LOAD REGISTER WITH**

**HALF-WORD AND SIGN**

**EXTEND WITH EA UPDATE**

**FORMS**

`lhau rD,d(rA)`

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>0x2b</th>
<th>D</th>
<th>A</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

EA ← rA + EXTS(d)
rD ← EXTS(MEM(EA, 2))
rA ← EA

**DESCRIPTION**

The `lhau` (load half-word algebraic with update) instruction loads a half-word from memory addressed by EA into the lower 16 bits of rD. The effective address (EA) is the sum (rA|0) + d, where d is a 16-bit signed value that is sign-extended to 32 bits (64 bits on 64-bit implementations). The EA is placed into rA.

On 32-bit implementations, rD[16-31] is loaded with the half-word addressed by the EA. On 64-bit implementations, rD[48-63] is loaded with the half-word addressed by the EA. The remaining higher-order bits of rD are filled with a copy of the most significant bit of the loaded half-word.

**REGISTERS AFFECTED**

None
**INTEGER UNIT**

**601/603/604/620**

**User Mode**

**FORMS**

lhaux  rD, rA, rB

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td></td>
<td></td>
<td></td>
<td>0x177</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

EA ← rA + rB
rD ← EXTS(MEM(EA, 2))
rA ← EA

**DESCRIPTION**

The lhaux (load half-word algebraic with update indexed) instruction loads a half-word from memory addressed by EA into the lower 16 bits of rD. The effective address (EA) is the sum (rA[0]) + rB. The EA is placed into rA.

On 32-bit implementations, rD[16-31] is loaded with the half-word addressed by the EA. On 64-bit implementations, rD[48-63] is loaded with the half-word addressed by the EA. The remaining higher-order bits of rD are filled with a copy of the most significant bit of the loaded half-word. Note that the PowerPC architecture defines load with update instructions with operand rA = r0 or rA = rD as invalid forms.

**REGISTERS AFFECTED**

None
**lha(x)**

**LOAD REGISTER WITH**

**HALF-WORD USING INDEXED**

**ADDRESSING AND SIGN EXTEND**

**FORMS**

\[ \text{lha}(x) \quad rD, rA, rB \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x1f</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>0x157</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[
\begin{align*}
\text{if } rA = r0 & \text{ then } b \leftarrow 0 \\
\text{else } b & \leftarrow rA \\
EA & \leftarrow b + rB \\
rD & \leftarrow \text{EXTS(MEM(EA, 2))}
\end{align*}
\]

**DESCRIPTION**

The \text{lha}(x) (load half-word algebraic indexed) instruction loads a half-word from memory addressed by EA into the lower 16 bits of rD. The effective address (EA) is the sum (rA + 0) + rB.

On 32-bit implementations, rD[16-31] is loaded with the half-word addressed by the EA. On 64-bit implementations, rD[48-63] is loaded with the half-word addressed by the EA. The remaining higher-order bits of rD are filled with a copy of the most significant bit of the loaded half-word.

**REGISTERS AFFECTED**

None
**INTEGER UNIT**

601/603/604/620

**User Mode**

**FORMS**

lhbr x rD,rA,rB

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ox1f</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Ox316</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

if rA = r0 then b ← 0
else b ← rA
EA ← b + rB
rD ← 0x0000 || (MEM(EA+1)) || (MEM(EA))

**DESCRIPTION**

The lhbr x (load half-word byte-reverse indexed) instruction loads a half-word value from memory addressed by the EA, byte-reverses the value, and stores the result into destination register rD. The effective address (EA) is the sum (rA\|0) + rB.

On 32-bit implementations, bits 0–7 of the half-word in memory addressed by EA are loaded into rD[24-31]; bits 8–15 of the half-word in memory are loaded into rD[16-23]. On 64-bit implementations, bits 0–7 of the half-word in memory addressed by EA are loaded into rD[56-63]; bits 8–15 of the half-word in memory are loaded into rD[48-55]. In all cases, the remaining higher-order bits of rD are cleared to 0.

The PowerPC architecture cautions programmers that some implementations may run this instruction with greater latency (perhaps much greater) than a sequence of individual load/store instructions that produce the same results.

**REGISTERS AFFECTED**

None
**lhz**

**LOAD REGISTER WITH HALF-WORD AND ZERO EXTEND**

**FORMS**

\[ \text{lhz} \quad rD, d(rA) \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x28</th>
<th>D</th>
<th>A</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td>22</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td>26</td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td>28</td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[
\begin{align*}
\text{if} \ rA = r0 & \text{ then } b \leftarrow 0 \\
\text{else } b & \leftarrow rA \\
\text{EA} & \leftarrow b + \text{EXTS}(d) \\
rD & \leftarrow (0x0000 || \text{MEM(EA,2)})
\end{align*}
\]

**DESCRIPTION**

The **lhz** (load half-word and zero) instruction loads a half-word value from memory addressed by the EA into destination register rD. The effective address (EA) is the sum (rA|0|) + d, where d is a 16-bit signed value that is sign-extended to 32 bits (64 bits on 64-bit implementations).

On 32-bit implementations, rD[16-31] is loaded with the half-word addressed by the EA. On 64-bit implementations, rD[48-63] is loaded with the half-word addressed by the EA. The remaining higher-order bits of rD are cleared to 0.

**REGISTERS AFFECTED**

None
INTEGER UNIT

601/603/604/620

USER MODE

**lhzu**

LOAD REGISTER WITH
HALF-WORD AND ZERO
EXTEND WITH EA UPDATE

**FORMS**

lhzu \( \text{rD},d(\text{rA}) \)

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>A</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x29</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |

**PSEUDO CODE**

\[
\text{EA} \leftarrow \text{rA} + \text{EXTS}(d) \\
\text{rD} \leftarrow (0x0000 \mid \text{(MEM(EA, 2)}) \\
\text{rA} \leftarrow \text{EA}
\]

**DESCRIPTION**

The `lhzu` (load half-word and zero with update) instruction loads a half-word value from memory addressed by the EA into destination register \( \text{rD} \). The effective address (EA) is the sum of \( \text{rA} + d \), where \( d \) is a 16-bit signed value that is sign-extended to 32 bits (64 bits on 64-bit implementations).

On 32-bit implementations, \( \text{rD}[16-31] \) is loaded with the half-word addressed by the EA. On 64-bit implementations, \( \text{rD}[48-63] \) is loaded with the half-word addressed by the EA. The remaining higher-order bits of \( \text{rD} \) are cleared to 0. The EA is placed into \( \text{rA} \). The PowerPC architecture defines load with update instructions with operand \( \text{rA} = \text{r0} \) or \( \text{rA} = \text{rD} \) as invalid forms.

**REGISTERS AFFECTED**

None
**lhzux**

**LOAD REGISTER WITH HALF-WORD USING INDEXED ADDRESSING, ZERO EXTEND, AND UPDATE EA**

**FORMS**

lhzux rD,rA,rB

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Bit</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>0x1f</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

EA ← rA + rB
rD ← (0x0000 || (MEM(EA, 2)))
rA ← EA

**DESCRIPTION**

The lhzux (load half-word and zero with update indexed) instruction loads a half-word value from memory addressed by the EA into destination register rD. The effective address (EA) is the sum (rA+0) + rB.

On 32-bit implementations, rD[16-31] is loaded with the half-word addressed by the EA. On 64-bit implementations, rD[48-63] is loaded with the half-word addressed by the EA. The remaining higher-order bits of rD are cleared to 0. The EA is placed into rA. The PowerPC architecture defines load with update instructions with operand rA = r0 or rA = rD as invalid forms.

**REGISTERS AFFECTED**

None
INTEGER UNIT

601/603/604/620

USER MODE

FORMS

lhzx rD,rA,rB

BIT DEFINITION

<table>
<thead>
<tr>
<th></th>
<th>0x1f</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0x117</td>
</tr>
</tbody>
</table>

PSEUDO CODE

if rA = r0 then b ← 0
else b ← rA
EA ← b + rB
rD ← (16)0 || (MEM(EA,2))

DESCRIPTION

The lhzx (load half-word and zero indexed) instruction loads a half-word value from memory addressed by the EA into destination register rD. The effective address (EA) is the sum (rA[0]) + rB.

On 32-bit implementations, rD[16-31] is loaded with the half-word addressed by the EA. On 64-bit implementations, rD[48-63] is loaded with the half-word addressed by the EA. The remaining higher-order bits of rD are cleared to 0.

REGISTERS AFFECTED
None
Ii
LOAD REGISTER WITH IMMEDIATE

FORMS
li rD, value = addi rA, r0, value

BIT DEFINITION

<table>
<thead>
<tr>
<th>0x0e</th>
<th>D</th>
<th>0</th>
<th>SIMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

PSEUDO CODE
rD ← EXTS(SIMM)

DESCRIPTION
The Ii (load immediate) instruction loads a 16-bit signed value into destination register rD. The 16-bit immediate value, SIMM, is sign-extended to 32 bits before the operation. Note that no addition is taking place during such an operation. The Ii instruction is a simplified form of the addi instruction.

REGISTERS AFFECTED
None

EXAMPLE
: Assumes:
: we want to zero r6 and copy that value to r5
:
ForLoop1:
    li    r6, 0        ; clear r6 by sign extending 0
    mr    r5, r6       ; move zero from r6 into r5
**INTEGER Unit**

**601/603/604/620**

**User Mode**

**Forms**

\[
\text{l} \text{i} \text{s} \ rD, \text{value} = \text{addi} \ rA,rO,\text{value}
\]

**Bit Definition**

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 0x0e | D | 0 | SIMM |   |

**Pseudo Code**

\[ rD \leftarrow \text{EXTS}(\text{SIMM}) \ll 16 \]

**Description**

The `lis` (load immediate shifted) instruction loads a 16-bit signed value into destination register rD. The 16-bit immediate value, SIMM, is shifted left by 16-bits to form a 32-bit value before the operation. Note that no addition is taking place during such an operation. The `lis` instruction is a simplified form of the `addi` instruction.

**Registers Affected**

None

**Example**

The following code fragment loads 0x12345678 into GPR r3.

: Assumes:
: we want load r3 with 0x12345678 and copy that value to r4
:

\[
\begin{align*}
\text{l} \text{i} \text{s} & \quad r3,0x1234 \quad ; \text{load high-order 16 bits of } r3 \\
\text{o} \text{ri} & \quad r3,r3,0x5678 \quad ; \text{and the low order 16 bits} \\
\text{mr} & \quad r4,r3 \quad ; \text{copy } r3 \text{ value to } r4
\end{align*}
\]
**lmw**

**LOAD REGISTERS WITH MULTIPLE WORDS**

### FORMS

- `lmw rD,d(rA)`

### BIT DEFINITION

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>A</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2e</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### PSEUDO CODE

```plaintext
if rA = r0 then b ← 0
else b ← rA
EA ← b + EXTS(d)
r ← rD
do while r ≤ 31
    GPR(r) ← MEM(EA,4)
    r ← r + 1
    EA ← EA + 4
```

### DESCRIPTION

The `lmw` (load multiple word) instruction operates on blocks of data (larger than 32 bits). The effective address (EA) is the sum (`rA`+0) + d, where d is a 16-bit signed value that is sign-extended to 32 bits (64 bits on 64-bit implementations).

Exactly n, where n = (32-rD), consecutive words starting at EA are loaded into GPRs rD through r31. The EA must be a multiple of 4; otherwise, the system alignment exception handler is invoked if the load crosses a page boundary. If rA is in the range of registers specified to be loaded, it will be skipped in the load process. If operand rA = r0, the register is not considered as used for addressing, and will be loaded.

The PowerPC architecture cautions programmers that some implementations may run this instruction with greater latency (perhaps much greater) than a sequence of individual load/store instructions that produce the same results.

### REGISTERS AFFECTED

None
**INTEGER UNIT**

**601/603/604/620**

**User Mode**

**lswi**

**LOAD REGISTERS WITH MULTIPLE BYTES**

**FORMS**

\[ lswi \quad rD,rA,NB \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x1f</th>
<th>D</th>
<th>A</th>
<th>NB</th>
<th>0x255</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

```plaintext
if rA = r0 then EA \leftarrow 0
else EA \leftarrow rA
if NB = 0 then n \leftarrow 32
else n \leftarrow NB
r \leftarrow rD - 1
I \leftarrow 32
do while n \geq 0
  if i = 32 then
    r \leftarrow r + 1 (mode32)
    GPR(r) \leftarrow 0
    GPR(r)[i-1+i] \leftarrow MEM(EA,1)
  I \leftarrow i + 8
```

**DESCRIPTION**

The `lswi` (load string word immediate) instruction operates on blocks of data (larger than 32 bits). The effective address (EA) is `(rA | 0)`. Let \( n = NB \) if \( NB \neq 0 \), \( n = 32 \) if \( NB = 0 \); \( n \) is the number of bytes to load. Let \( nr = \text{CEIL}(n/4) \); \( nr \) is the number of registers to be loaded with data. \( n \) consecutive bytes starting at the EA are loaded into GPRs \( rD \) through \( rD+nr-1 \). Bytes are loaded left to right in each register. The sequence of registers wraps around to \( r0 \) if required. If the 4 bytes of register \( rD + nr - 1 \) are only partially filled, the unfilled low-order byte(s) of that register are cleared to 0.

If \( rA \) and \( rB \) are in the range of registers specified to be loaded, it will be skipped in the load process. If operand \( rA = 0 \), the register is not considered as used for addressing, and will be loaded. Under certain conditions (for example, segment boundary crossings) the data alignment error handler may be invoked. For additional information, see Chapter 10, “Exceptions and Interrupts.”

The PowerPC architecture cautions programmers that some implementations may run this instruction with greater latency (perhaps much greater) than a sequence of individual load/store instructions that produce the same results.

**REGISTERS AFFECTED**

None
**lswx**

Load registers with multiple bytes using indexed addressing

**Forms**

\[ \text{lswx } rD,rA,rB \]

**Bit Definition**

<table>
<thead>
<tr>
<th>0x1f</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>0x215</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pseudo Code**

\[
\text{if } rA = r0 \text{ then } b \leftarrow 0 \\
\text{else } b \leftarrow rA \\
\text{EA} \leftarrow b + rB \\
n \leftarrow \text{XER}[25-31] \\
r \leftarrow rD - 1 \\
I \leftarrow 32 \\
do \text{ while } n > 0 \\
\text{if } i = 32 \text{ then} \\
r \leftarrow r + 1 \text{ (mode32)} \\
\text{GPR}(r) \leftarrow 0 \\
\text{GPR}(r)[i-i+7] \leftarrow \text{MEM}(\text{EA},1) \\
I \leftarrow i + 8
\]

**Description**

The `lswx` (load string word indexed) instruction operates on blocks of data (larger than 32 bits). The effective address (EA) is the sum \((rA0) + rB\). Let \(n = \text{XER}[25-31]\); \(n\) is the number of bytes to load. Let \(nr = \text{CEIL}(n/4)\); \(nr\) is the number of registers to receive data. If \(n=0\), the content of \(rD\) is undefined.

Bytes are loaded left to right in each register. The sequence of registers wraps around to \(r0\) if required. If the four bytes of register \(rD+nr-1\) are only partially filled, the unfilled low-order byte(s) of that register are cleared to 0. If \(n = 0\), the content of \(rD\) is undefined.

If \(rA\) and \(rB\) are in the range of registers specified to be loaded, it will be skipped in the load process. If operand \(rA = 0\), the register is not considered as used for addressing, and will be loaded. Under certain conditions (for example, segment boundary crossings) the data alignment error handler may be invoked. For additional information, see Chapter 10, “Exceptions and Interrupts.”

The PowerPC architecture cautions programmers that some implementations may run this instruction with greater latency (perhaps much greater) than a sequence of individual load/store instructions that produce the same results.

**Registers Affected**

None
**INTEGER UNIT**

**620**  
**USER MODE**

**lwa**  
LOAD REGISTER WITH WORD  
AND SIGN EXTEND

**FORMS**  
lwa rD,ds(rA)

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x3a</th>
<th>D</th>
<th>A</th>
<th>ds</th>
<th>0x02</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

if rA = r0 then b ← 0  
else b ← rA  
EA ← b + EXTS(ds || 0b00)  
rD ← EXTS(MEM(EA,4))

**DESCRIPTION**

The lwa (load word algebraic) instruction loads the word in memory addressed by EA into the low-order bits of rD. The contents of the high-order 32 bits of rD are filled with a copy of bit 0 of the loaded word. The effective address (EA) is the sum (rA+0) + (ds || 0b00). Note that ds is a 14-bit signed value which is concatenated on the right with 0b00; this 16-bit value is sign-extended to 64 bits.

**REGISTERS AFFECTED**

None
**Iwarx**

**Load Register with Word Using Indexed Addressing and Create Reservation**

**Forms**

\[ \text{lwarx } r_D, r_A, r_B \]

**Bit Definition**

<table>
<thead>
<tr>
<th>0x1f</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>0x14</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pseudo Code**

\[
\begin{align*}
\text{if } r_A &= r_0 \text{ then } b \leftarrow 0 \\
\text{else} & \quad b \leftarrow r_A \\
\text{EA} & \leftarrow b + r_B \\
\text{RESERVE} & \leftarrow 1 \\
\text{RESERVE_ADDR} & \leftarrow \text{func(EA)} \\
r_D & \leftarrow \text{MEM(EA,4)}
\end{align*}
\]

**Description**

The `lwarx` (load word and reserve indexed) instruction loads the word in memory addressed by EA into destination register `r_D`. The effective address (EA) is the sum `(r_A0) + r_B`. The `lwarx` instruction can be used to perform atomic memory accesses. This instruction creates a reservation for use by a store word conditional instruction. The physical address computed from the EA is associated with the reservation, and replaces any address previously associated with the reservation. Reservations are discussed in Chapter 6, “The PowerPC Instruction Set.”

The EA must be a multiple of 4. If it is not, the alignment exception handler will be invoked if the load crosses a page boundary, or the results will be boundedly undefined.

**Registers Affected**

None
**INTEGER UNIT**

**620**

**USER MODE**

**FORMS**

lwaux rD,rA,rB

**BIT DEFINITION**

\[
\begin{array}{cccccc}
0x1f & D & A & B & 0x175 & 0 \\
\end{array}
\]

**PSEUDO CODE**

EA ← rA + rB
rD ← EXT(MEM(EA,4))
rA ← EA

**DESCRIPTION**

The lwaux (load word algebraic with update indexed) instruction is defined for 64-bit PowerPC implementations only. Using it on a 32-bit implementation will cause the system illegal instruction error handler to be invoked.

The word in memory addressed by EA is loaded into the low-order bits of rD. The contents of the high-order 32 bits of rD are filled with a copy of bit 0 of the loaded word. The effective address (EA) is the sum rA + rB. The EA is placed into rA. If rA = r0, or rA = rD, the instruction is invalid.

**REGISTERS AFFECTED**

None
**lwax**

**LOAD REGISTER WITH WORD**
**USING INDEXED ADDRESSING**
**AND SIGN EXTEND**

**FORMS**
lwax rD,rA,rB

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>0x155</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

if rA = r0 then b ← 0
else b ← rA
EA ← b + rB
rD ← EXTS(MEM(EA,4))

**DESCRIPTION**
The lwax (load word algebraic indexed) instruction loads the word in memory addressed by EA into the low-order bits of rD. The contents of the high-order 32 bits of rD are filled with a copy of bit 0 of the loaded word. The effective address (EA) is the sum (rA|0) + rB.

**REGISTERS AFFECTED**
None
**INTEGER UNIT**

**601/603/604/620**

**User Mode**

**FORMS**

`lwbrx rD, rA, rB`

**Bit Definition**

<table>
<thead>
<tr>
<th>Ox1f</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>Ox216</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
</tr>
<tr>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
</tr>
<tr>
<td>47</td>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
</tr>
</tbody>
</table>

**Pseudo Code**

if rA = r0 then b ← 0
else b ← rA
EA ← b + rB
rD ← MEM(EA+3, 1) || MEM(EA+2, 1) || MEM(EA+1, 1) || MEM(EA, 1)

**Description**

The `lwbrx` (load word byte-reverse indexed) instruction loads a word from memory and byte-reverses the value before storing the result into destination register rD. The effective address (EA) is the sum (rA10) + rB.

On 32-bit implementations, bits 0-7 of the word in memory addressed by EA are loaded into rD[24-31]. Bits 8-15 of the word in memory are loaded into rD[16-23]. Bits 16-23 of the word in memory addressed by EA are loaded into rD[8-15]. Bits 24-31 of the word in memory addressed by EA are loaded into rD[0-7].

On 64-bit implementations, bits 0-7 of the half-word in memory addressed by EA are loaded into rD[56-63]; bits 8-15 of the half-word in memory are loaded into rD[48-55]. Bits 16-23 of the word in memory addressed by EA are loaded into rD[40-47]. Bits 24-31 of the word in memory addressed by EA are loaded into rD[32-39]. The remaining higher-order 32 bits of rD are cleared to 0.

The PowerPC architecture cautions programmers that some implementations may run this instruction with greater latency (perhaps much greater) than a sequence of individual load/store instructions that produce the same results.

**Registers Affected**

None
**lwz**

**LOAD REGISTER WITH WORD AND ZERO EXTEND**

**FORMS**

\[ \text{lwz } rD, d(rA) \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x20</th>
<th>D</th>
<th>A</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

```plaintext
if rA = r0 then b ← 0
else b ← rA
EA ← b + EXTS(d)
rD ← MEM(EA, 4)
```

**DESCRIPTION**

The \textit{lwz} (load word and zero) instruction loads a word from memory addressed by the effective address (EA) into the destination register \( rD \). The effective address is the sum of \( rA + 0 \) + \( d \), where \( d \) is a 16-bit signed value that is sign-extended to 32 bits (64 bits on 64-bit implementations). On 64-bit implementations, the high-order 32 bits of \( rD \) are cleared to 0.

**REGISTERS AFFECTED**

None

**EXAMPLE**

```plaintext
templ = 0; // templ and temp2 are globally defined unsigned
temp2 = templ;
```

: Assumes:
: \( r3 \) = contains address of \( \text{templ} \)
: \( r5 \) = contains address of \( \text{temp2} \)
: 
li r4, 0 ; load \( r4 \) with immediate zero
stb r4, 0(r3) ; store 0 to \( \text{templ} \)
lbz r3, 0(r3) ; get byte value from \( \text{templ} \)
stb r3, 0(r5) ; store byte from \( r3 \) into \( \text{temp2} \)
**INTEGER UNIT**

601/603/604/620

**User Mode**

**Forms**

lwzu rD, d(rA)

**Bit Definition**

**Pseudo Code**

EA ← rA + EXTS(d)

rD ← MEM(EA, 4)

rA ← EA

<table>
<thead>
<tr>
<th>0x21</th>
<th>D</th>
<th>A</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

**Description**

The **lwzu** (load word and zero with update) instruction loads a word from memory addressed by the EA into destination register rD. The effective address (EA) is the sum (rA0) + d, where d is a 16-bit signed value that is sign-extended to 32 bits (64 bits on 64-bit implementations). The EA is placed into rA. On 64-bit implementations, the high-order 32 bits of rD are cleared to 0. The PowerPC architecture defines load with update instructions with operand rA = r0 or rA = rD as invalid forms.

**Registers Affected**

None
**lwzux**

**LOAD REGISTER WITH WORD**
**USING INDEXED ADDRESSING,**
**ZERO EXTEND, AND UPDATE EA**

**FOMRS**

\[ \text{lwzux \ rD, rA, rB} \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>0x1f</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>0x37</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[
\text{EA} \leftarrow \text{rA} + \text{rB} \\
\text{rD} \leftarrow \text{MEM(EA, 4)} \\
\text{rA} \leftarrow \text{EA}
\]

**DESCRIPTION**

The \text{lwzux} (load word and zero with update indexed) instruction loads a word from memory addressed by the EA into destination register rD. The effective address (EA) is the sum \text{rA} + \text{rB}. The EA is placed into rA. On 64-bit implementations, the high-order 32 bits of rD are cleared to 0. The PowerPC architecture defines load with update instructions with operand \text{rA} = \text{r0} or \text{rA} = \text{rD} as invalid forms.

**REGISTERS AFFECTED**

None
**INTEGER UNIT**

*601/603/604/620 USER MODE*

**FORMS**

\[ \text{lwzx} \quad \text{rD, rA, rB} \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Bit</th>
<th>0x1f</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>0x17</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[
\begin{align*}
&\text{if } rA = r0 \text{ then } b \leftarrow 0 \\
&\text{else } b \leftarrow rA \\
&EA \leftarrow b + rB \\
rD \leftarrow \text{MEM}(EA, 4)
\end{align*}
\]

**DESCRIPTION**

The \textit{lwzx} (load word and zero indexed) instruction load a word from memory addressed by the EA into destination register rD. The effective address (EA) is the sum \((rA[0]) + rB\). On 64-bit implementations, the high-order 32 bits of rD are cleared to 0.

**REGISTERS AFFECTED**

None
**mcrf**

**COPY CONDITION**

**REGISTER FIELD**

**FORMS**

mcrf crfD,crfS

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Res.</th>
<th>crfD</th>
<th>crfS</th>
<th>Reserved</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x13</td>
<td>0</td>
<td>0</td>
<td>000000</td>
<td>0</td>
</tr>
</tbody>
</table>

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

**PSEUDO CODE**

CR[4*crfD - 4*crfD+3] ← CR[4*crfS - 4*crfS+3]

**DESCRIPTION**

The mcrf (move condition register field) instruction copies the contents of condition register field crfS into condition register field crfD. All other condition register fields remain unchanged.

Note that if the link bit (bit 31) is set for this instruction, the PowerPC architecture considers the instruction to be of an invalid form. Use of invalid instruction forms is not recommended. This description is provided for informational purposes only.

**REGISTERS AFFECTED**

CR[LT,GT,EQ,SO] (CR field specified by operand crfD)
**Floating-Point Unit**

**601/603/604/620 User Mode**

**mcrfs**  
MOVE FROM FPSCR TO CONDITION REGISTER

**Forms**
mcrfs crfD,crfS

**Bit Definition**

<table>
<thead>
<tr>
<th>Res.</th>
<th>crfD</th>
<th>crfS</th>
<th>Reserved</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3f</td>
<td>00</td>
<td>00</td>
<td>00000</td>
<td>0x40</td>
</tr>
</tbody>
</table>

**Pseudo Code**
CR[4*crfD - 4*crfS+3] ← CR[4*crfS - 4*crfS+3]

**Description**
The mcrfs (move to condition register from FPSCR) instruction copies the contents of FPSCR field crfS into condition register field crfD. All exception bits copied are reset to zero in the FPSCR.

**Registers Affected**
- CR [FX,FEX,VX,OX] (CR field specified by operand crfD)
- FX, OX (if crfS = 0)
- UX,ZX,XX,VXSNAN (if crfS = 1)
- VXISI,VXIDI,VXZDZ,VXIMZ (if crfS = 2)
- VXVC (if crfS = 3)
- VXSOFT,VXSQRT,VXCVI (if crfS = 5)
mcrx\(r\)

**Move from XER to Condition Register**

**Forms**

\[ mcrf s \quad \text{crfD} \]

**Bit Definition**

<table>
<thead>
<tr>
<th>0x1f</th>
<th>crfD</th>
<th>00</th>
<th>00000</th>
<th>00000</th>
<th>0x200</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pseudo Code**

\[
\text{CR}[4*\text{crfD} - 4*\text{crfD}+3] \leftarrow \text{XER}[0-3] \\
\text{XER}[0-3] \leftarrow 0b0000
\]

**Description**

The \textit{mcrx}r (move to condition register from XER) instruction copies the contents of \text{XER}[0-3] into the condition register field designated by \text{crfD}. All other fields of the condition register remain unchanged. \text{XER}[0-3] is cleared to zero.

**Registers Affected**

- \text{CR}[\text{LT,GT,EQ,SO}] (CR field specified by operand \text{crfD})
- \text{XER}[0-3]
**INTEGER UNIT**

**601/603/604/620 USER MODE**

**FORMS**

mcrfs rD

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>Reserved</th>
<th>Reserved</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td>D</td>
<td>00000</td>
<td>0x13</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>4 5 6 7</td>
<td>8 9 10</td>
<td>11 12</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

rD ← CR

**DESCRIPTION**

The `mfc` (move from condition register) instruction places the contents of the condition register into destination register rD.

**REGISTERS AFFECTED**

None
**mffsx**

**Move from FPSCR to Floating-point Register**

### Forms

<table>
<thead>
<tr>
<th>mffs</th>
<th>frD</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>mffs</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>mffs</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### Bit Definition

<table>
<thead>
<tr>
<th></th>
<th>Reserved</th>
<th>Reserved</th>
<th>frD</th>
<th>00000</th>
<th>00000</th>
<th>Ox247</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ox3f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Description

The `mffs` (move from FPSCR) instruction places the contents of FPSCR into bits `frD[32-63]`. Bits `frD[0-31]` are undefined.

### Registers Affected

- CR(CR1 field)
- [LT, GT, EQ, SO] (if Rc = 1)
### INTEGER UNIT

**601/603/604/620**

**SUPERVISOR MODE**

### FORMS

| mfmsr | rD |

### BIT DEFINITION

<table>
<thead>
<tr>
<th></th>
<th>Reserved</th>
<th>Reserved</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td>D</td>
<td>00000</td>
<td>0x53</td>
</tr>
</tbody>
</table>

### PSEUDO CODE

rD ← MSR

### DESCRIPTION

The mfmsr (move from MSR) instruction places the contents of MSR into destination register rD.

### REGISTERS AFFECTED

None
**mfspr**

**Move from Special Purpose Register to General Purpose Register**

**Forms**

mfspr rD,SPR

**Simplified Mnemonics**

- mfxer rD = mfspr rD,1
- mflr rD = mfspr rD,8
- mfctr rD = mfspr rD,9

Chapter 6, “The PowerPC Instruction Set,” contains a complete listing of the simplified mnemonics for moving from/to each of the special-purpose registers.

**Bit Definition**

<table>
<thead>
<tr>
<th>Res.</th>
<th>0x1f</th>
<th>D</th>
<th>SPR</th>
<th>0x153</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pseudo Code**

n ← SPR[5-9] || SPR[0-4]

rD ← SPR(n)

**Description**

The mfspr (move from SPR) instruction places the contents of the designated special-purpose register into destination register rD. The SPR field denotes a special-purpose register, encoded as shown in Table A-3. If the SPR field contains any value other than one of the values shown in Table A-3 (for a particular implementation), then one of the following will occur:

- The system illegal instruction error handler is invoked.
- The system supervisor-level instruction error handler is invoked.
- The results are boundedly undefined within the target register.

SPR = 1 if and only if reading the register is supervisor-level. Execution of this instruction specifying a defined and supervisor-level register when MSR[PR] = 1 will result in a supervisor-level instruction exception.

If MSR[PR] = 1 then the only effect of executing an instruction with an SPR number that is not shown in the following table and has SPR[0] = 1 is to cause a supervisor-level instruction type program exception or an illegal instruction type program exception. For all other cases, MSR[PR] = 0 or SPR[0] = 0, if the SPR field contains any value that is not shown in Table A-3, then either an illegal instruction type program exception occurs or the results are boundedly undefined.
See Table A-3 for mfspr encoding details.

Note that for mtsp and mfspr instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16-20 of the instruction and the low-order bits appearing in bits 11-15. See the example below for further details.

**REGISTERS AFFECTED**

None

<table>
<thead>
<tr>
<th>Table A-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPR Encodings for mfspr Instruction.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SPR Register</th>
<th>ER Value</th>
<th>Implementation</th>
<th>Access Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQ</td>
<td>0 / 0x0</td>
<td>00000</td>
<td>601</td>
</tr>
<tr>
<td>XER</td>
<td>1 / 0x1</td>
<td>00000</td>
<td>00001</td>
</tr>
<tr>
<td>RTCU</td>
<td>4 / 0x4</td>
<td>00000</td>
<td>00100</td>
</tr>
<tr>
<td>RTCL</td>
<td>5 / 0x5</td>
<td>00000</td>
<td>00101</td>
</tr>
<tr>
<td>DEC</td>
<td>6 / 0x6</td>
<td>00000</td>
<td>00110</td>
</tr>
<tr>
<td>LR</td>
<td>8 / 0x8</td>
<td>00000</td>
<td>01000</td>
</tr>
<tr>
<td>CTR</td>
<td>9 / 0x9</td>
<td>00000</td>
<td>01001</td>
</tr>
<tr>
<td>DSISR</td>
<td>18 / 0x12</td>
<td>00000</td>
<td>10010</td>
</tr>
<tr>
<td>DAR</td>
<td>19 / 0x13</td>
<td>00000</td>
<td>10011</td>
</tr>
<tr>
<td>DEC</td>
<td>22 / 0x16</td>
<td>00000</td>
<td>11001</td>
</tr>
<tr>
<td>SDR1</td>
<td>25 / 0x19</td>
<td>00000</td>
<td>11010</td>
</tr>
<tr>
<td>SRR0</td>
<td>26 / 0x1a</td>
<td>00000</td>
<td>11011</td>
</tr>
<tr>
<td>SRR1</td>
<td>27 / 0x1b</td>
<td>00000</td>
<td>11100</td>
</tr>
<tr>
<td>SPRG0</td>
<td>272 / 0x110</td>
<td>01000</td>
<td>10000</td>
</tr>
<tr>
<td>SPRG1</td>
<td>273 / 0x111</td>
<td>01000</td>
<td>10001</td>
</tr>
<tr>
<td>SPRG2</td>
<td>274 / 0x112</td>
<td>01000</td>
<td>10010</td>
</tr>
<tr>
<td>SPRG3</td>
<td>275 / 0x113</td>
<td>01000</td>
<td>10011</td>
</tr>
<tr>
<td>EAR</td>
<td>282 / 0x11a</td>
<td>01000</td>
<td>11010</td>
</tr>
<tr>
<td>PVR</td>
<td>287 / 0x11f</td>
<td>01000</td>
<td>11111</td>
</tr>
<tr>
<td>IBAT0U</td>
<td>528 / 0x210</td>
<td>10000</td>
<td>10000</td>
</tr>
<tr>
<td>IBAT0L</td>
<td>529 / 0x211</td>
<td>10000</td>
<td>10001</td>
</tr>
<tr>
<td>IBAT1U</td>
<td>530 / 0x212</td>
<td>10000</td>
<td>10010</td>
</tr>
</tbody>
</table>
### Table A-3

SPR Encodings for mfspr Instruction (Continued)

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Decimal / Hex</th>
<th>spr[5-9]</th>
<th>spr[0-4]</th>
<th>Implementation</th>
<th>Access Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBAT1L</td>
<td>531 / 0x213</td>
<td>10000</td>
<td>10111</td>
<td>601,603,604</td>
<td>Supervisor</td>
</tr>
<tr>
<td>IBAT2U</td>
<td>532 / 0x214</td>
<td>10000</td>
<td>10100</td>
<td>601,603,604</td>
<td>Supervisor</td>
</tr>
<tr>
<td>IBAT2L</td>
<td>533 / 0x215</td>
<td>10000</td>
<td>10101</td>
<td>601,603,604</td>
<td>Supervisor</td>
</tr>
<tr>
<td>IBAT3U</td>
<td>534 / 0x216</td>
<td>10000</td>
<td>10110</td>
<td>601,603,604</td>
<td>Supervisor</td>
</tr>
<tr>
<td>IBAT3L</td>
<td>535 / 0x217</td>
<td>10000</td>
<td>10111</td>
<td>601,603,604</td>
<td>Supervisor</td>
</tr>
<tr>
<td>DBAT0U</td>
<td>536 / 0x218</td>
<td>10000</td>
<td>11000</td>
<td>603,604</td>
<td>Supervisor</td>
</tr>
<tr>
<td>DBAT0L</td>
<td>537 / 0x219</td>
<td>10000</td>
<td>11001</td>
<td>603,604</td>
<td>Supervisor</td>
</tr>
<tr>
<td>DBAT1U</td>
<td>538 / 0x21a</td>
<td>10000</td>
<td>11010</td>
<td>603,604</td>
<td>Supervisor</td>
</tr>
<tr>
<td>DBAT1L</td>
<td>539 / 0x21b</td>
<td>10000</td>
<td>11011</td>
<td>603,604</td>
<td>Supervisor</td>
</tr>
<tr>
<td>DBAT2U</td>
<td>540 / 0x21c</td>
<td>10000</td>
<td>11100</td>
<td>603,604</td>
<td>Supervisor</td>
</tr>
<tr>
<td>DBAT2L</td>
<td>541 / 0x21d</td>
<td>10000</td>
<td>11101</td>
<td>603,604</td>
<td>Supervisor</td>
</tr>
<tr>
<td>DBAT3U</td>
<td>542 / 0x21e</td>
<td>10000</td>
<td>11110</td>
<td>603,604</td>
<td>Supervisor</td>
</tr>
<tr>
<td>DBAT3L</td>
<td>543 / 0x21f</td>
<td>10000</td>
<td>11111</td>
<td>603,604</td>
<td>Supervisor</td>
</tr>
<tr>
<td>DMISS</td>
<td>976 / 0x3d0</td>
<td>11110</td>
<td>10000</td>
<td>603</td>
<td>Supervisor</td>
</tr>
<tr>
<td>DCMP</td>
<td>977 / 0x3d1</td>
<td>11110</td>
<td>10001</td>
<td>603</td>
<td>Supervisor</td>
</tr>
<tr>
<td>HASH1</td>
<td>978 / 0x3d2</td>
<td>11110</td>
<td>10010</td>
<td>603</td>
<td>Supervisor</td>
</tr>
<tr>
<td>HASH2</td>
<td>979 / 0x3d3</td>
<td>11110</td>
<td>10011</td>
<td>603</td>
<td>Supervisor</td>
</tr>
<tr>
<td>IMISS</td>
<td>980 / 0x3d4</td>
<td>11110</td>
<td>10100</td>
<td>603</td>
<td>Supervisor</td>
</tr>
<tr>
<td>ICMP</td>
<td>981 / 0x3d5</td>
<td>11110</td>
<td>10101</td>
<td>603</td>
<td>Supervisor</td>
</tr>
<tr>
<td>RPA</td>
<td>982 / 0x3d6</td>
<td>11110</td>
<td>10110</td>
<td>603</td>
<td>Supervisor</td>
</tr>
<tr>
<td>HID0</td>
<td>1008 / 0x3f0</td>
<td>11111</td>
<td>10000</td>
<td>Imp. specific</td>
<td>Supervisor</td>
</tr>
<tr>
<td>IABR (HID2)</td>
<td>1010 / 0x3f2</td>
<td>11111</td>
<td>10010</td>
<td>Imp. specific</td>
<td>Supervisor</td>
</tr>
<tr>
<td>DABR (HID5)</td>
<td>1013 / 0x3f5</td>
<td>11111</td>
<td>10101</td>
<td>Imp. specific</td>
<td>Supervisor</td>
</tr>
<tr>
<td>PIR (HID15)</td>
<td>1023 / 0x3ff</td>
<td>11111</td>
<td>11111</td>
<td>Imp. specific</td>
<td>Supervisor</td>
</tr>
</tbody>
</table>
**INTEGER UNIT**

**601/603/604 SUPERVISOR MODE**

**FORMS**

\[
mfsr \quad \text{rD,SR}
\]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Res.</th>
<th>D</th>
<th>SR</th>
<th>00000</th>
<th>0x253</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

**PSEUDO CODE**

\[
rD \leftarrow \text{SEGREG(SR)}
\]

**DESCRIPTION**

The \text{mfsr} (move from segment register) instruction places the contents of segment register \text{SR} into destination register \text{rD}. This instruction is defined only for 32-bit implementations; using it on a 64-bit implementation causes an illegal instruction type program exception. Note that 64-bit implementations use the ASR and a memory-based table of segment descriptors instead of segment registers.

**REGISTERS AFFECTED**

None
**mfsrin**

**MOVE FROM SEGMENT REGISTER TO GENERAL PURPOSE REGISTER USING INDIRECT ADDRESSING**

**FORMS**

\[ \text{mfsrin} \ rD,rB \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>Reserved</th>
<th>D</th>
<th>00000</th>
<th>B</th>
<th>0x293</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[ rD \leftarrow \text{SEGREGR}(rB[0-3]) \]

**DESCRIPTION**

The mfsrin (move from segment register indirect) instruction places the contents of segment register SR into destination register rD. The mfsrin instruction is defined only for 32-bit implementations; using it on a 64-bit implementation causes an illegal instruction type program exception. Note that 64-bit implementations use the ASR and a memory-based table of segment descriptors instead of segment registers.

**REGISTERS AFFECTED**

None
**INTEGER UNIT**

603/604/620

**User Mode**

**FORMS**

- `mftb rD, TBR`

**SIMPLIFIED MNEMONICS**

- `mftb rD` = `mftb rD, 268`
- `mftbu rD` = `mftb rD, 269`

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Bit</th>
<th>D</th>
<th>tbr</th>
<th>0x1f</th>
<th>0x173</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

```plaintext
if (TBR = 268)
    if (64-bit implementation)
        rD ← TB
    else rD ← TBR[lower]
else if (TBR = 269)
    if (64-bit implementation)
        rD ← 32(0) || TBR[upper]
    else rD ← TBR[upper]
else invoke system illegal instruction error handler
```

**DESCRIPTION**

The `mftb` (move from time base register) instruction places the contents of the time base register into destination register rD. TBL (time base lower) is defined as SPR 268. TBU (time base upper) is defined as SPR 269.

**REGISTERS AFFECTED**

None
**mrx**

**MOVE REGISTER**

**FORMS**

- \( mr \) \( rA, rS \) = or \( rA, rS, rS \)
- \( mr. \) \( rA, rS \) = or. \( rA, rS, rS \)

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x1f</th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>0x1bc</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[ rA \leftarrow rS \]

**DESCRIPTION**

The \( mr \) (move register) instruction places the contents of \( rS \) into \( rA \). The simplified \( mr \) instruction is provided to convey the idea that no computation is being performed, but merely data movement (from one general-purpose register to another). This instruction is a simplified form of the or instruction.

**REGISTERS AFFECTED**

\( CR0[LT,GT,EQ,SO] \) (if \( Rc = 1 \))

**EXAMPLE**

```c
if (longA < longB) { // both are globally declared longs
    longA = longB;
}
```

; Assumes:
; \( r3 = \) \( longA \), 32-bit value
; \( r4 = \) \( longB \)

```
cmpwi r3, r4 ; longA greater than longB?
bgt Around ; yes, jump around assignment
mr r3, r4 ; place contents of \( r4 \) into \( r3 \)
```

**Around:**

; execution continues as normal
**INTEGER UNIT**

**601/603/604/620**

**User Mode**

**mtcr**

Move from register to condition register

**Forms**

\[ \text{mtcr} \quad rS \equiv \text{mtcrf} \quad 0xff, rS \]

**Bit Definition**

<table>
<thead>
<tr>
<th>S</th>
<th>CRM</th>
<th>0x90</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x1f</td>
<td>0</td>
</tr>
<tr>
<td>0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

The **mtcr** (move to condition register) instruction moves the contents of rS into the condition register. This instruction is a simplified form of the **mtcrf** instruction.

**Registers Affected**

All CR fields.
**mtcrf**

**Move from Register to Condition Register Fields**

**Forms**

\[ \text{mtc"rf} \quad \text{CRM}, rS \]

**Bit Definition**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td>S</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>CRM</td>
<td>0</td>
</tr>
<tr>
<td>0x90</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

**Pseudo Code**

\[
\begin{align*}
\text{mask} & \leftarrow (4)(\text{CRM}[0]) || (4)(\text{CRM}[1]) || \ldots (4)(\text{CRM}[7]) \\
\text{CR} & \leftarrow (rS[32-63] \& \text{mask}) | (\text{CR} \& \neg \text{mask})
\end{align*}
\]

**Description**

The *mtc"rf* (move to condition register field) places the contents of rS into the condition register under control of the field mask specified by CRM. The field mask identifies the 4-bit fields affected. Let \(i\) be an integer in the range 0 – 7. If CRM\((i) = 1\), CR Field \(i\) (CR bits \(4^i \cdot i \) through \(4^i \cdot i + 3\)) is set to the contents of the corresponding field of the rS.

**Registers Affected**

CR fields selected by mask.
**INTEGER UNIT**

**601/603/604/620 USER MODE**

**FORMS**

- mtfsb0 crbD 0
- mtfsb0. crbD 1

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x3f</th>
<th>crbD</th>
<th>00000</th>
<th>00000</th>
<th>0x46</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

FPSCR[crbD] = 0

**DESCRIPTION**

The mtfsb0 (move to FPSCR bit 0) instruction clears bit crbD of the FPSCR to zero.

**REGISTERS AFFECTED**

- CR(CR1 Field)[LT,GT,EQ,SO] (if Rc = 1)
- FPSCR bit crbD

Note: Bits 1 and 2 (FEX and VX) cannot be explicitly cleared.
**mfsb1**

**Set bit of FPSCR**

**Form**

\[
\begin{align*}
\text{mfsb1} & \quad \text{crbD} 0 \\
\text{mfsb1} & \quad \text{crbD} 1
\end{align*}
\]

**Bit Definition**

<table>
<thead>
<tr>
<th>Reserved</th>
<th>Reserved</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3f</td>
<td>crbD</td>
<td>00000</td>
</tr>
</tbody>
</table>

**Pseudo Code**

\[ \text{FPSCR}[\text{crbD}] = 1 \]

**Description**

The `mfsb1` (move to FPSCR bit 1) instruction sets bit crbD of the FPSCR to one.

**Registers Affected**

- CR(CR1 Field)[FX,FEX,VX,OX] (if Rc = 1)
- FPSCR bit crbD and FX

Note: Bits 1 and 2 (FEX and VX) cannot be explicitly set.
**Integer Unit**

**601/603/604/620**

**User Mode**

**MTFSFX**

**Move from floating-point register to FPSCR fields**

**Forms**

<table>
<thead>
<tr>
<th>MTFSF</th>
<th>FM, frB</th>
<th>Rc</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTFSF</td>
<td>FM, frB</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Bit Definition**

<table>
<thead>
<tr>
<th>Res.</th>
<th>0x3f</th>
<th>FM</th>
<th>0</th>
<th>B</th>
<th>0x2c7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

The **mtfsf** (move to FPSCR fields) instruction places bits 32–63 of register frB into the FPSCR under control of the field mask specified by FM. The field mask identifies the 4-bit fields affected. Let $i$ be an integer in the range 0–7. If $FM(i) = 1$, FPSCR Field $i$ (FPSCR bits $4*i$ through $4*i+3$) is set to the contents of the corresponding field of the low-order 32 bits of register frB. Updating fewer than all eight fields of the FPSCR may cause substantially poorer performance on some implementations than updating all the fields.

When FPSCR[0-3] is specified, bits 0 (FX) and 3 (OX) are set to the values of frB[32] and frB[35] (that is even if this instruction causes OX to change from 0 to 1, FX is set from frB[32] and not by the usual rule that FX is set to 1 when an exception bit changes from 0 to 1). Bits 1 and 2 (FEX and VX) are set according to the usual rule and not from frB[33-34].

**Registers Affected**

- CR(CR1 Field)[FX,FEX,VX,OX] (if Rc=1)
- FPSCR: FPSCR fields selected by mask.
The `mtfsfi` (move to FPSCR field immediate) instruction places the value IMM into FPSCR field crfD. When FPSCR[0-3] is specified, bits 0 (FPSCR[FX]) and 3 (FPSCR[OX]) are set to the values of IMM[0] and IMM[3] (that is even if this instruction causes OX to change from 0 to 1, FX is set from IMM[0] and not by the usual rule that FPSCR[FX] is set to 1 when an exception bit changes from 0 to 1). Bits 1 and 2 (FPSCR[FEX and VX]) are set according to the rules defined in Chapter 3, “Of Eggs and Endians,” and not from IMM[1-2].

**REGISTERS AFFECTED**

- CR(CR1 Field)[FX,FEX,VX,OX] (if Rc=1)
- FPSCR field crfD
**mtmsr**

**Move from register to machine state register**

**FORMS**

mtmsr rS

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x1f</th>
<th>S</th>
<th>00000</th>
<th>00000</th>
<th>0x92</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

MSR ← rS

**DESCRIPTION**

The mtmsr (move from MSR) instruction places the contents of rS into the MSR.

**REGISTERS AFFECTED**

MSR[All Bits]
**mtspr**

**MOVE FROM REGISTER TO SPECIAL PURPOSE REGISTER**

**FORMS**

mtspr  SPR, rS

**SIMPLIFIED MNEMONICS**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtxer</td>
<td>rD = mtspr 1, rD</td>
</tr>
<tr>
<td>mtctr</td>
<td>rD = mtspr 8, rD</td>
</tr>
<tr>
<td>mtclr</td>
<td>rD = mtspr 9, rD</td>
</tr>
</tbody>
</table>

Chapter 6, “The PowerPC Instruction Set,” contains a complete listing of the simplified mnemonics for moving from/to each of the special-purpose registers.

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>0x1f</th>
<th>S</th>
<th>SPR</th>
<th>0x1d3</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[ n \leftarrow SPR[5:9] \llcorner SPR[0:4] \]

\[ SPR(n) \leftarrow rS[0:31] \]

**DESCRIPTION**

The mtspr (move to special purpose register) instruction places the contents of rS into the designated special-purpose register. The SPR field denotes a special-purpose register, encoded as shown in the table that accompanies the definition of the mtspr instruction.

The value of SPR[0] is 1 if and only if reading the register is at the supervisor level. Execution of this instruction specifying a supervisor-level register when MSR[PR] = 1 will result in a supervisor-level instruction exception. For an invalid instruction form in which SPR[0] = 1, if MSR[PR] = 1 a supervisor-level type program exception will occur instead of a no-op.

**REGISTERS AFFECTED**

None
**INTEGER UNIT**

**601/603/604**

**SUPERVISOR MODE**

---

### mtsr

**Move from register to segment register**

#### FORMS

\[ \text{mtsr} \quad \text{SR}, rS \]

#### BIT DEFINITION

<table>
<thead>
<tr>
<th></th>
<th>Res.</th>
<th>Reserved</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td>S</td>
<td>0</td>
<td>SR</td>
</tr>
</tbody>
</table>

#### PSEUDO CODE

\[ \text{SEGR} \left( \text{SR} \right) \leftarrow rS \]

#### DESCRIPTION

The `mtsr` (move to segment register) instruction places the contents of `rS` into segment register `SR`.

This instruction is defined only for 32-bit implementations; using it on a 64-bit implementation causes an illegal instruction type program exception. Note that 64-bit implementations use the ASR and a memory-based table of segment descriptors instead of segment registers.

#### REGISTERS AFFECTED

None
**mtsrin**

**MOVE FROM REGISTER TO SEGMENT REGISTER USING INDIRECT ADDRESSING**

**FORMS**

mtsrin rS,rB

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Reserved</th>
<th>0x1f</th>
<th>S</th>
<th>00000</th>
<th>B</th>
<th>0xf2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

SEGREG(rB[0-3]) ← rS

**DESCRIPTION**

The mtsrin (move to segment register indirect) instruction copies the contents of rS to the segment register selected by bits 0–3 of rB.

This instruction is defined only for 32-bit implementations; using it on a 64-bit implementation causes an illegal instruction type program exception. Note that 64-bit implementations use the ASR and a memory-based table of segment descriptors instead of segment registers.

**REGISTERS AFFECTED**

None
**INTEGER UNIT**

**620**

**USER MODE**

**FORMS**

<p>| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>mulhd</td>
<td>rD,rA,rB</td>
<td>Rc</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mulhd.</td>
<td>rD,rA,rB</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x1f</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>0</th>
<th>0x49</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

prod[0-127] ← rA * rB
rD ← prod[0-63]

**DESCRIPTION**

The `mulhd` (multiply high doubleword) instruction multiplies rA by rB and places the high-order 64 bits of the 128-bit product into destination register rD. Both the operands and the product are interpreted as signed integers. This instruction may execute faster on some implementations if rB contains the operand having the smaller absolute value.

**REGISTERS AFFECTED**

CR0[L_T,G_T,E_Q,S_O] (if Rc = 1)

Note: The setting of CR0 bits LT, GT, and EQ is mode-dependent, and reflects overflow of the 64-bit result.
**mulhdux**

**MULTIPLY REGISTERS AS**

**UNSIGNED KEEPING HIGH-ORDER PORTION OF RESULT**

**FORMS**

<table>
<thead>
<tr>
<th></th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>mulhdu</td>
<td>rD, rA, rB</td>
</tr>
<tr>
<td>mulhdu</td>
<td>rD, rA, rB</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>0x1f</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>0x09</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[
\text{prod}[-127] \leftarrow rA \times rB \\
rD \leftarrow \text{prod}[0-63]
\]

**DESCRIPTION**

The `mulhdux` (multiply high doubleword unsigned) instruction multiplies `rA` by `rB` and places the high-order 64-bits of the 128-bit product into destination register `rD`. Both the operands and the product are interpreted as unsigned integers, except that if `Rc = 1` the first 3 bits of `CR0 Field` are set by signed comparison of the result to zero. This instruction may execute faster on some implementations if `rB` contains the operand having the smaller absolute value.

**REGISTERS AFFECTED**

`CR0[LT,GT,EQ,SO]` (if `Rc = 1`)

Note: The setting of `CR0` bits `LT`, `GT`, and `EQ` is mode-dependent, and reflects overflow of the 64-bit result.
**INTEGER UNIT**

**601/603/604/620**  
**User Mode**

**FORMS**

<table>
<thead>
<tr>
<th>mulhw</th>
<th>rD,rA,rB</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>mulhw.</td>
<td>rD,rA,rB</td>
<td>1</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>0</th>
<th>0x4b</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

```plaintext
prod[0-31] ← prod[0-31]
prod[0-31] ← undefined
```

**DESCRIPTION**

The `mulhw` (multiply high word) instruction multiplies rA by rB and places the result into destination register rD. The contents of rA and rB are interpreted as 32-bit signed integers. They are multiplied to form a 64-bit signed integer product. The high-order 32 bits of the 64-bit product are placed into destination register rD.

If the smaller absolute value of the two multipliers is placed in rB, the instruction may complete execution quicker. See Chapter 7, “The Sublime Art of Instruction Timing,” for additional information about instruction performance.

**REGISTERS AFFECTED**

CR0[LT,GT,EQ,SO] (if Rc = 1)
mulhwux

MULTIPLY REGISTERS AS
UNSIGNED KEEPING HIGH-ORDER PORTION OF RESULT

FORMS

<table>
<thead>
<tr>
<th>mulhwux</th>
<th>rD,rA,rB</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>mulhwux</td>
<td>rD,rA,rB</td>
<td>1</td>
</tr>
</tbody>
</table>

BIT DEFINITION

<table>
<thead>
<tr>
<th></th>
<th>0x1f</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>0</th>
<th>0x0b</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>rD[32-63] ← prod[0-31]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rD[0-31] ← undefined</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DESCRIPTION

The mulhwux (multiply high word unsigned) instruction multiplies rA by rB and places the result into destination register rD. The contents of rA and of rB are extracted and interpreted as 32-bit unsigned integers. They are multiplied to form a 64-bit unsigned integer product. The high-order 32 bits of the 64-bit product are placed into destination register rD.

If the smaller absolute value of the two multipliers is placed in rB, the instruction may complete execution quicker. See Chapter 7, “The Sublime Art of Instruction Timing,” for additional information about instruction performance. The instruction causes the contents of the MQ to become undefined.

REGISTERS AFFECTED

CR0[LT,GT,EQ,SO] (if Rc = 1)
**INTEGER UNIT**

**620 USER MODE**

**mulld**

**MULTIPLY DOUBLEWORD**

**REGISTERS**

**FORMS**

<table>
<thead>
<tr>
<th>Form</th>
<th>Destination</th>
<th>OE</th>
<th>OE</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>mulld</td>
<td>rD,rA,rB</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>mulld.</td>
<td>rD,rA,rB</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>mulldo</td>
<td>rD,rA,rB</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>mulldo.</td>
<td>rD,rA,rB</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Bit</th>
<th>0x1f</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>OE</th>
<th>0xe9</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

prod[0-127] ← rA * rB
rD ← prod[64-127]

**DESCRIPTION**

The **mulld** (multiply doubleword) instruction multiplies rA by rB. The low-order 64-bits of the 128-bit product is placed into destination register rD.

Both the operands and the product are interpreted as signed integers. The low-order 64 bits of the product are independent of whether the operands are regarded as signed or unsigned 64-bit integers. If OE = 1, then CV is set if the product cannot be represented in 64 bits. This instruction may execute faster on some implementations if rB contains the operand having the smaller absolute value.

**REGISTERS Affected**

- **CR0[LT,GT,EQ,SO]** (if Rc = 1)
  
  Note: CR0 Field may not reflect the “true” (infinitely precise) result if overflow occurs (see XER below).

- **XER[SO,OV]** (if OE = 1)
  
  Note: The setting of the affected bits in the XER is mode-dependent, and reflects overflow of the 64-bit result.
The `mulli` (multiply low word immediate) instruction multiplies a `rA` by a 16-bit signed value, and places the low-order 32 bits of the 48-bit product into destination register `rD`. The low-order bits of the 32-bit product are independent of whether the operands are treated as signed or unsigned integers.

**Pseudo Code**

```
prod[0-48] ← rA * SIMM
rD ← prod[16-48]
```

**Description**

**REGISTERS AFFECTED**

None

**Example**

```
tmp_long *= 10;       // global unsigned long

: Assumes:
: r3 = contains address of tmp_long
:
  lwz r4, 0(r3)       ; get value at address
  mulli r4, r4, 10    ; perform multiply low immed.
  stw r4, 0(r3)       ; store back low-order 32-bits of result
```
** INTEGER UNIT **

** 601/603/604/620 **

** User Mode **

** FORMS **

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>mullw</td>
<td>rD, rA, rB</td>
</tr>
<tr>
<td>mullw.</td>
<td>rD, rA, rB</td>
</tr>
<tr>
<td>mullwo</td>
<td>rD, rA, rB</td>
</tr>
<tr>
<td>mullwo.</td>
<td>rD, rA, rB</td>
</tr>
</tbody>
</table>

** BIT DEFINITION **

<table>
<thead>
<tr>
<th>0x1f</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>OE</th>
<th>0xeb</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

** PSEUDO CODE **

r0 ← rA[32-63] * rB[32-63]

** DESCRIPTION **

The **mullw** (multiply low word) instruction multiplies rA with rB and places the low-order 32 bits of the 64-bit product into destination register rD. The low-order bits of the 32-bit product are independent of whether the operands are treated as signed or unsigned integers. However, OV is set based on the result interpreted as a signed integer.

If the smaller absolute value of the two multipliers is placed in rB, the instruction may complete execution quicker. See Chapter 7, “The Sublime Art of Instruction Timing,” for additional information about instruction performance.

** REGISTERS AFFECTED **

- CR0[LT,GT,EQ,SO] (if Rc = 1)
- XER[SO,OV] (if OE = 1)

** EXAMPLE **

```plaintext
long1 *= char1; // both are globally declared unsigned values
```

; Assumes:
; r3 = contains address of char1
; r4 = contains address of long1
:
1bz r3, 0(r3) ; get byte from address of char1
lwz r5, 0(r4) ; get word from address of long1
mullw r5, r5, r3 ; perform multiply
stw r5, 0(r4) ; store back results
nandx
AND TWO REGISTERS AND
COMPLEMENT RESULT

FORMS

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>nand</td>
<td>rA,rS,rB</td>
<td>0</td>
</tr>
<tr>
<td>nand.</td>
<td>rA,rS,rB</td>
<td>1</td>
</tr>
</tbody>
</table>

BIT DEFINITION

<table>
<thead>
<tr>
<th></th>
<th>0x1f</th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>0x1dc</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PSEUDO CODE

rA ← NOT (rS & rB)

DESCRIPTION

The nand (not and) instruction performs a bitwise AND of rS with rB, takes the one's complement of the value, and places the final result into rA.

REGISTERS AFFECTED

CR0[LT,GT,EQ,SO] (if Rc = 1)
**INTEGER UNIT**

**601/603/604/620**

**User Mode**

**FORMS**

<table>
<thead>
<tr>
<th>neg</th>
<th>rD, rA</th>
<th>OE</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>neg.</td>
<td>rD, rA</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>nego</td>
<td>rD, rA</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>nego.</td>
<td>rD, rA</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>negx</td>
<td>rD, rA</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Res.</th>
<th>0x1f</th>
<th>D</th>
<th>A</th>
<th>00000</th>
<th>OE</th>
<th>0x68</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

rD ← NOT(rA) + 1

**DESCRIPTION**

The `neg` instruction negates rA by forming the two’s complement, the sum of NOT(rA) and one, and places the result into destination register rD. If rA contains the most negative 32-bit number, the low-order 32 bits of the result contain the most negative 32-bit number and, if OE = 1, OV is set.

On 32-bit implementations, the most negative number is 0x80000000; on 64-bit implementations, the most negative number is 0x8000000000000000.

**REGISTERS AFFECTED**

- CR0[LT,GT,EQ,SO] (if Rc = 1)
- XER[SO, OV] (if OE = 1)
**nop**

**NO OPERATION**

### INTRODUCTION

**FORMS**

\[
\text{nop} = \text{ori} \; r0, r0, 0
\]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x18</th>
<th>00000</th>
<th>00000</th>
<th>0x0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[
\text{if } rA = r0 \text{ then } rD = rA + \text{EXTS(SIMM || (16)0)} \\
\text{else } rD = rA + (\text{SIMM || (16)0})
\]

**DESCRIPTION**

The **nop** (no operation) instruction does not perform any logical, arithmetic, or memory operation. The **nop** instruction is used to assist in instruction scheduling and does not alter the state of any registers. This instruction is a simplified form of the **ori** instruction.

**REGISTERS AFFECTED**

None
**INTEGER UNIT**

**601/603/604/620**

**User Mode**

**FORMS**

<table>
<thead>
<tr>
<th>nor</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>rA, rS, rB</td>
<td>0</td>
</tr>
<tr>
<td>nor. rA, rS, rB</td>
<td>1</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>0x1f</th>
<th>0x7c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

rA ← NOT(rS | rB)

**DESCRIPTION**

The `nor` instruction performs a bitwise OR of rS with rB and places the one's complement of the result into destination register rA. Note that when rS = rB, the resulting value is the one's complement of rS.

**REGISTERS AFFECTED**

CR0[LT, GT, EQ, SO] (if RC = 1)

**EXAMPLE**

```plaintext
long1 = -char1; // both are globally declared unsigned values

; Assumes:
; r3 = contains value of char1
; r4 = contains address of long1
;
lbz r3, 0(r3)    ; get byte from address
nor r3, r3, r3   ; perform one's complement
stw r3, 0(r4)    ; store result
```
The `not` (NOT) instruction calculates the one's complement of `rS` and places the result into destination register `rA`. The `not` instruction is a simplified form of the `nor` instruction.

**REGISTERS AFFECTED**
CR0[LT,GT,EQ,SO] (if Rc = 1)
INTEGER UNIT

601/603/604/620
USER MODE

FORMS

\[
\begin{align*}
&\text{or } rA, rS, rB & Rc \\
&\text{or } rA, rS, rB & 1
\end{align*}
\]

SIMPLIFIED MNEMONICS

\[
\text{mr } rA, rS \equiv \text{or } rA, rS, rD
\]

BIT DEFINITION

<table>
<thead>
<tr>
<th>Ox1f</th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>Ox1bc</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

PSEUDO CODE

\[
rA \leftarrow rS \mid rB
\]

DESCRIPTION

The or instruction performs a bitwise OR of \(rS\) with \(rB\) and places the result into destination register \(rA\).

REGISTERS AFFECTED

\(\text{CR0}[\text{LT,GT,EQ,SO}]\) (if \(Rc = 1\))

EXAMPLE

\[
\text{int1} \leftarrow 0x12345678; \quad // \text{globally declared int}
\]

; Assumes:
; \(r4\) = contains address of \(\text{int1}\)
;
\text{lwx} \quad r3, 0(r4) \quad ; \text{get value from address in } r4
\text{addis} \quad r5, r0, 0x1234 \quad ; \text{load upper part of } r5 \text{ with value to \char'134 or'}
\text{addi} \quad r5, r5, 0x5678 \quad ; \text{load lower part of } r5
\text{or} \quad r3, r3, r5 \quad ; \text{perform or operation: } r3 = r3 \mid r5
\text{stw} \quad r3, 0(r4) \quad ; \text{store back results to address in } r4
**orc**

**OR REGISTER WITH COMPLEMENTED REGISTER**

**FORMS**

\[
\begin{align*}
orc & \quad rA, rS, rB & \quad 0 \\
orc & \quad rA, rS, rB & \quad 1 \\
\end{align*}
\]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>(0x1f)</th>
<th>(S)</th>
<th>(A)</th>
<th>(B)</th>
<th>(0x19c)</th>
<th>(Rc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 2 3 4 5 6 7 8 9 10 11 12 13 14 15</td>
<td></td>
<td></td>
<td></td>
<td>20 21 22 23 24 25 26 27 28 29 30 31</td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[
rA \leftarrow rS \mid \text{NOT}(rB)
\]

**DESCRIPTION**

The **orc** (OR with complement) instruction performs a bitwise OR of \(rS\) with the complement of \(rB\) and places the result into destination register \(rA\).

**REGISTERS AFFECTED**

\(\text{CR0}[\text{LT}, \text{GT}, \text{EQ}, \text{SO}]\) (if \(Rc = 1\))

**EXAMPLE**

\[
\text{int1} \leftarrow -\text{int2}; \quad // \text{globally declared integers}
\]

; Assumes:
; \(r4\) = contains address of \(\text{int1}\)
; \(r5\) = contains address of \(\text{int2}\)

\[
\begin{align*}
lwz & \quad r3, 0(r4) \quad ; \text{get value from address in } r4 \\
lwz & \quad r5, 0(r5) \quad ; \text{get value from address in } r5 \\
orc & \quad r3, r3, r5 \quad ; \text{or and complement } r3 \text{ with } r5 \\
stw & \quad r3, 0(r4) \quad ; \text{store back results to address in } r4
\end{align*}
\]
**INTEGER UNIT**

601/603/604/620

**User Mode**

**FORMS**

\[ \text{ori \ rA,rS,UIMM} \]

**Simplified Mnemonics**

\[ \text{nop} = \text{ori \ r0,r0,0} \]

**Bit Definition**

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>A</th>
<th>UIMM</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x18</td>
<td>0</td>
<td>0</td>
<td>0x00</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pseudo Code**

\[ \text{rA} \leftarrow \text{rS} | (0x0000 || \text{UIMM}) \]

**Description**

The \text{ori} (OR immediate) instruction performs a bitwise OR of \text{rS} with a 32-bit unsigned value and places the result into destination register \text{rA}. The 16-bit immediate value, \text{UIMM}, is zero-extended to 32 bits before the operation. This instruction is used to perform the preferred no-op (an instruction that does nothing):

\[ \text{ori r0,r0,0} \]

**Registers Affected**

None

**Example**

\[ \text{int1} = \text{int2} | 0x12; \] // globally declared integers

; Assumes:
; \text{r4} = contains address of \text{int1}
; \text{r5} = contains address of \text{int2}

; \text{lwz} \quad \text{r3, 0(r5)} \quad \text{; get value from address in \text{r5}}
\text{ori} \quad \text{r3, r3, 0x12} \quad \text{; perform immediate or w/ 0x12}
\text{stw} \quad \text{r3, 0(r4)} \quad \text{; store results to \text{int1 address}}
**oris**

**OR SHIFTED IMMEDIATE WITH REGISTER**

**FORMS**
oris rA,rS,UIMM

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x19</th>
<th>S</th>
<th>A</th>
<th>UIMM</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**
rA ← rS | (UIMM || 0x0000)

**DESCRIPTION**
The oris (OR immediate shifted) instruction performs a bitwise OR of rS with a 32-bit unsigned value and places the result into destination register rA. The 16-bit immediate value, UIMM, is shifted left 16 bits before the operation.

**REGISTERS AFFECTED**
None
INTEGER UNIT

601/603/604/620

SUPERVISOR MODE

FORMS

rfi

BIT DEFINITION

<table>
<thead>
<tr>
<th>0x13</th>
<th>00000</th>
<th>00000</th>
<th>00000</th>
<th>0x32</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PSEUDO CODE

MSR[16-31] ← SRR[16-31]
NIA ←tea SRR0[0-29] || 0b00

DESCRIPTION

The rfi (return from interrupt) instruction is used to return from exception handling code; rfi restores the contents of the machine state register (MSR) saved upon entry to the exception handler. Bits 16–31 of SRR1 are placed into bits 16–31 of the MSR, then the next instruction is fetched, under control of the new MSR value, from the address SRR0[0-29] || 0b00. This instruction is context synchronizing.

REGISTERS AFFECTED

MSR[All bits]
Form

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |

*Note: This is a split field.*

**BIT DEFINITION**

**PSEUDO CODE**

\[
\begin{align*}
    n & \leftarrow rB[58-63] \\
    r & \leftarrow \text{ROTL}(rS,n) \\
    m & \leftarrow \text{MASK}(MB,63) \\
    rA & \leftarrow (r \& m)
\end{align*}
\]

**DESCRIPTION**

The rldcl (rotate left doubleword then clear left) instruction rotates the contents of rS left by the number of bits specified by operand in the low-order 6 bits of rB. A mask is generated having 1 bits from bit MB through bit 63 and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed in destination register rA.

Note that the rldcl instruction can be used to extract and rotate bit fields using the methods shown below:

To extract an n-bit field, that starts at bit position b in register rS, right-justified into rA (clearing the remaining 64-n bits of rA), set the low-order 6 bits of rB to b+n and MB = 64-n.

To rotate the contents of a register left by variable n bits, set the low-order 6 bits of rB to n and MB = 0. (This is equivalent to rotating right 64-n bits.)

**REGISTERS AFFECTED**

CR0[LT,GT,EQ,SO] (if Rc = 1)
**INTEGER UNIT**

620

**USER MODE**

**FORMS**

<table>
<thead>
<tr>
<th></th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>rldcr</td>
<td>rA, rS, rB, ME 0</td>
</tr>
<tr>
<td>rldcr</td>
<td>rA, rS, rB, ME 1</td>
</tr>
</tbody>
</table>

**Simplified Mnemonics**

See Chapter 6, "The PowerPC Instruction Set," for a detailed list of simplified mnemonics.

**Bit Definition**

<table>
<thead>
<tr>
<th>0x1e</th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>ME</th>
<th>0x09</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pseudo Code**

```plaintext
n ← rB[58-63]
r ← ROTL(rS, n)
m ← MASK(0, ME)
rA ← (r & m)
```

**Description**

The `rldcr` (rotate left doubleword then clear right) instruction rotates the contents of `rS` left by the number of bits specified by the low-order 6 bits of `rB`. A mask is generated having 1 bits from bit 0 through bit ME and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed in destination register `rA`.

Note that the `rldcr` instruction can be used to extract and rotate bit fields using the methods shown below:

- To extract an `n`-bit field, that starts at bit position `b` in register `rS`, left-justified into `rA` (clearing the remaining 64-`n` bits of `rA`, set the low-order 6 bits of `rB` to `b` and `ME = n-1`.

- To rotate the contents of a register left (right) by variable `n` bits by setting the low-order 6 bits of `rB` to `n(64-n)` and `ME = 63`.

**Registers Affected**

CR0[LT,GT,EQ,SO] (if Rc = 1)
**Rldicx**

**Rotate register left by immediate then clear with mask**

**Forms**

<table>
<thead>
<tr>
<th></th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rldic</td>
<td>0</td>
</tr>
<tr>
<td>Rldic</td>
<td>1</td>
</tr>
</tbody>
</table>

**Simplified Mnemonics**

Clear left and shift left immediate: clrlsldi rA,rS,b,n = rldic rA,rS,n,b-n


**Bit Definition**

<table>
<thead>
<tr>
<th>0x1e</th>
<th>S</th>
<th>A</th>
<th>SH*</th>
<th>MB</th>
<th>0x02</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Note: This is a split field.

**Pseudo Code**


r ← ROTL(rS,n)

m ← MASK(MB, 63-n)

rA ← (r & m)

**Description**

The rldic (rotate left doubleword then clear) instruction rotates the contents of rS left by the number of bits specified by the SH operand. A mask is generated having 1 bits from bit 0 through bit MB and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed in destination register rA.

Note that the rldic instruction can be used to clear and shift bit fields using the methods shown below:

- To clear the high-order \( b \) bits of the contents of a register and then shift the result by \( n \) bits, set \( SH = n \) and \( MB = b-n \).
- To clear the high-order \( n \) bits of a register, set \( SH = 0 \) and \( MB = n \).

**Registers Affected**

CR0[LT,GT,EQ,SO] (if Rc = 1)

**Example**

See the example that accompanies the clrlsldi instruction.
INTEGER UNIT

620
USER MODE

**FORMS**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Result (Rc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rldiclx rA,rS,SH,MB</td>
<td>0</td>
</tr>
<tr>
<td>rldiclx rA,rS,SH,MB</td>
<td>1</td>
</tr>
</tbody>
</table>

**SIMPLIFIED**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Equivalent Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>extrdi rA,rS,n,b (n&gt;0)</td>
<td>rldiclx rA,rS,b+n,64-n</td>
</tr>
<tr>
<td>rotldi rA,rS,n</td>
<td>rldiclx rA,rS,n,0</td>
</tr>
<tr>
<td>rotldi rA,rS,n</td>
<td>rldiclx rA,rS,64-n,0</td>
</tr>
<tr>
<td>srldi rA,rS,n (n&lt;64)</td>
<td>rldiclx rA,rS,64-n,n</td>
</tr>
<tr>
<td>clrldi rA,rS,n (n&lt;64)</td>
<td>rldiclx rA,rS,0,n</td>
</tr>
</tbody>
</table>


**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1e</td>
<td>Definition of fields</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[
\begin{align*}
  n & \leftarrow SH[5] \text{ OR } SH[0-4] \\
  r & \leftarrow ROTL(rS,n) \\
  m & \leftarrow MASK(MB,63) \\
  rA & \leftarrow (r \land m)
\end{align*}
\]

**DESCRIPTION**

The **rldiclx** (rotate register left by immediate then clear left with mask) instruction rotates the contents of register **rS** left the number of bits specified by operand **SH**. A mask is generated having 1 bits from bit **MB** through bit 63 and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed in destination register **rA**.

Note that the **rldiclx** instruction can be used to extract, rotate, shift, and clear bit fields using the methods shown below:

- To extract an **n**-bit field, that starts at bit position **b** in **rS**, right-justified into **rA** (clearing the remaining 64-**n** bits of **rA**), set **SH** = **b**+**n** and **MB** = 64-**n**.
- To rotate the register left (right) by **n** bits, set **SH** = **n**(64-**n**) and **MB** = 0.
- To shift the contents of a register right by **n** bits, set **SH** = 64-**n** and **MB** = **n**.
- To clear the high-order **n** bits of a register, set **SH** = 0 and **MB** = **n**.

**REGISTERS AFFECTED**

CR0[LT,GT,EQ,SO] (if Rc = 1)

**EXAMPLE**

See the examples that accompany the **extrdi** and **clrdi** instructions.
**rldicr**

**ROnate register left by immediate then clear right with mask**

**FORMS**

```
rldicr  rA,rS,SH,ME  Rc
rldicr  rA,rS,SH,ME  0
```

**SIMPLIFIED MNEMONICS**

```
extldi rA,rS,n,b  =  rldicr rA,rS,b,n-1
sldi  rA,rS,n  =  rldicr rA,rS,n,63-n
clrrdi  rA,rS,n  =  rldicr rA,rS,0,63-n
```


**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x1e</th>
<th>S</th>
<th>A</th>
<th>SH*</th>
<th>ME</th>
<th>0x01</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
</tr>
</tbody>
</table>

*Note: This is a split field.

**PSEUDO CODE**

```
r  =  R0TL(rS,n)
m  =  MASK(0, ME)
rA  =  (r & m)
```

**DESCRIPTION**

The rldicr (rotate left doubleword immediate then clear right) instruction rotates the contents of rS left by the number of bits specified by the SH operand. A mask is generated having 1 bits from bit 0 through bit ME and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed in destination register rA.

Note that the rldicr instruction can be used to extract, rotate, shift, and clear bit fields using the methods shown below:

- To extract an n-bit field, that starts at bit position b in rS, left-justified into rA (clearing the remaining 64-n bits of rA), set SH = b and ME = n-1.
- To rotate the register left (right) by n bits, set SH = n(64-n) and ME = 0.
- To shift the contents of a register right by n bits, set SH = n(64-n) and ME = 63.
- To clear the low-order n bits of a register, set SH = 0 and ME = 63-n.

**REGISTERS AFFECTED**

CR0[LT,GT,EQ,SO] (if Rc = 1)

**EXAMPLE**

See the examples that accompany the extldi and clrrdi instructions.
**INTEGER UNIT**

**620**

**USER MODE**

**FORMS**

\[
\text{rldimi} \quad rA, rS, SH, MB \quad 0 \\
\text{rldimis} \quad rA, rS, SH, MB \quad 1 \\
\]

**Simplified Mnemonics**

\[
\text{insrdi} \ rA, rS, n, b \quad \equiv \quad \text{rldimi} \ rA, rS, 64-(b+n), b \\
\]


**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x1e</th>
<th>S</th>
<th>A</th>
<th>sh*</th>
<th>mb*</th>
<th>0x03</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Note: This is a split field.*

**Pseudo Code**

\[
\begin{align*}
n &\leftarrow \text{sh}[5] || \text{sh}[0-4] \\
r &\leftarrow \text{ROTL} (rS, n) \\
b &\leftarrow \text{mb}[5] || \text{mb}[0-4] \\
m &\leftarrow \text{MASK}(b, -n) \\
rA &\leftarrow (r & m) \mid (rA \& \neg m)
\end{align*}
\]

**Description**

The **rldimi** (rotate left doubleword immediate then mask insert) instruction rotates the contents of rS left by the number of bits specified by the SH operand. A mask is generated having 1 bits from bit MB through bit 63-SH and 0 bits elsewhere. The rotated data is inserted into rA under control of the generated mask.

Note that the **rldimi** instruction can be used to insert an n-bit field, that is right-justified in rS, into rA starting at bit position b, by setting SH = 64-(b+n) and MB = b.

**Registers Affected**

CR0[LT,GT,EQ,SO] (if Rc = 1)

**Example**

See the example that accompanies the **insrdi** instruction.
**rlwimix**

**Rotate register left by immediate then insert**

**Forms**

- `rlwimi`  `rA, rS, SH, MB, ME  Rc`
- `rlwimi`  `rA, rS, SH, MB, ME  1`

**Simplified Mnemonics**

- Insert from left immediate: `inslwi rA, rS, n, b = rlwimi rA, rS, (32 - b), b, 0, (b + n - 1)`
- Insert from right immediate: `insrwi rA, rS, n, b = rlwimi rA, rS, (32 - b + n), b, (b + n - 1)`

**Bit Definition**

<table>
<thead>
<tr>
<th>0x14</th>
<th>S</th>
<th>A</th>
<th>SH</th>
<th>MB</th>
<th>ME</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pseudo Code**

- `r ← ROTL(rS, SH)`
- `m ← MASK(MB, ME)`
- `rA ← (r & m) | (rA & NOT m)`

**Description**

The `rlwimi` (rotate left word immediate then mask insert) instruction rotates the contents of `rS` left by `SH` bits. A mask is generated having 1 bits from bit `MB` through bit `ME` and 0 bits elsewhere. The rotated value is inserted into `rA` under control of the generated mask.

Note that `rlwimi` can be used to insert a bit field into the contents of `rA` using the methods shown below:

- To insert an `n`-bit field, that is left justified in `rS`, into `rA` starting at bit position `b`, set `SH = 32 - b`, `MB = b`, and `ME = (b + n) - 1`.
- To insert an `n`-bit field, that is right-justified in `rS`, into `rA` starting at bit position `b`, set `SH = 32 - (b + n)`, `MB = b`, and `ME = (b + n) - 1`.

**Registers Affected**

CR0[LT,GT,EQ,SO](if Rc = 1)

**Example**

See the examples that accompany the `inslwi` and `insrwi` instructions.
INTEGER UNIT

601/603/604/620

User Mode

FORMS

<table>
<thead>
<tr>
<th>Rlwinm</th>
<th>rA, rS, SH, MB, ME</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rlwinm</td>
<td>rA, rS, SH, MB, ME</td>
<td>0</td>
</tr>
<tr>
<td>Rlwinm</td>
<td>rA, rS, SH, MB, ME</td>
<td>1</td>
</tr>
</tbody>
</table>

SIMPLIFIED MNEMONICS

clear right immediate: clrwi rA, rS, n (n < 32) = rlwinm rA, rS, 0, 0, 31-n
clear left immediate: clrlwi rA, rS, n (n < 32) = rlwinm rA, rS, 0, n, 31
rotate right immediate: rotrwi rA, rS, n = rlwinm rA, rS, 32-n, 0, 31
shift right immediate: srwi rA, rS, n (n < 32) = rlwinm rA, rS, 32-n, n, 31
extract and right justify immediate: extrwi rA, rS, n, b (n > 0) = rlwinm rA, rS, b+n, 32-n, 31
extract and left justify immediate: extlwi rA, rS, n, b (n > 0) = rlwinm rA, rS, b, 0, n-1
rotate left immediate: rotlwi rA, rS, n = rlwinm rA, rS, n, 0, 31

BIT DEFINITION

<table>
<thead>
<tr>
<th>0x15</th>
<th>S</th>
<th>A</th>
<th>SH</th>
<th>MB</th>
<th>ME</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
</tr>
</tbody>
</table>

PSEUDO CODE

r ← ROTL(rS, SH)
m ← MASK(MB, ME)
rA ← (r & m)

DESCRIPTION

The rlwinm (rotate left word immediate then AND with mask) instruction rotates the contents of rS left by SH bits. A mask is generated having 1 bits from bit MB through bit ME and 0 bits elsewhere. The rotated value is inserted into rA under control of the generated mask.

Note that rlwinm can be used to insert a bit field into the contents of rA using the methods shown below:

- To extract an n-bit field that starts at bit position b of rS, right-justified into rA (clearing the remaining 32-n bits of rA), set SH = b+n, MB = 32-n, and ME = 31.
- To extract an n-bit field that starts at bit position b of rS, left-justified into rA (clearing the remaining 32-n bits of rA), set SH = b, MB = 0, and ME = n-1.
- To rotate the contents of a register left (or right) by n bits, set SH = n(32-n), MB = 0, and ME = 31.
- To shift the contents of a register right by n bits, set SH = 32-n, MB = n, and ME = 31.
- To clear the low-order n bits of a register, set SH = 0, MB = 0, and ME = 31-n.

REGISTERS AFFECTED

CR0[LT, GT, EQ, SO] (if Rc = 1)

EXAMPLE

See the examples that accompany the extlwi, extrwi, clrwi, and clrlwi instructions.
**rlwnm**

**ROTATE REGISTER LEFT THEN AND WITH MASK**

**FORMS**

- **rlwnm**  \( rA, rS, rB, MB, ME \)  \( Rc \) 0
- **rlwnm.**  \( rA, rS, rB, MB, ME \)  \( Rc \) 1

**SIMPLIFIED MNEMONICS**

- Shift left immediate: \( \text{slwi } rA, rS, n \) (\( n < 32 \))  \( = \) \( \text{rlwnm } rA, rS, n, 0, 31-n \)
- Clear left and shift left immediate: \( \text{clrlslwi } rA, rS, b, n \) (\( n \leq b \leq 31 \))  \( = \) \( \text{rlwnm } rA, rS, n, b-n, 31-n \)
- Rotate left: \( \text{rotlw } rA, rS, rB \)  \( = \) \( \text{rlwnm } rA, rS, rB, 0, 31 \)

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x17</th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>MB</th>
<th>ME</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[ n \leftarrow rB[27-31] \]
\[ r \leftarrow \text{ROTL}(rS, n) \]
\[ m \leftarrow \text{MASK}(MB, ME) \]
\[ rA \leftarrow (r \& m) \]

**DESCRIPTION**

The **rlwnm** (rotate left word then AND with mask) instruction rotates the contents of \( rS \) left by the number of bits specified by the low-order 5 bits of \( rB \). A mask is generated having 1 bits from bit \( MB \) through bit \( ME \) and 0 bits elsewhere. The rotated value is ANDed with the generated mask and the result is placed in destination register \( rA \). Note that **rlwnm** can be used to extract and rotate bit fields using the following methods:

- To extract an n-bit field that starts at variable bit position \( b \) of \( rS \), right-justified into \( rA \) (clearing the remaining 32-n bits of \( rA \)), but setting the low-order 5 bits of \( rB \) to \( b+n \), \( MB = 32-n \), and \( ME = 31 \).
- To extract an n-bit field that starts at variable bit position \( b \) of \( rS \), left-justified into \( rA \) (clearing the remaining 32-n bits of \( rA \)), by setting the low-order 5 bits of \( rB \) to \( b \), \( MB = 0 \), and \( ME = n-1 \).
- To rotate the contents of a register left (or right) by \( n \) bits, by setting the low-order 5 bits of \( rB \) to \( n(32-n) \), \( MB = 0 \), and \( ME = 31 \).

For each of the above uses, the high-order 32 bits of \( rA \) are cleared on 64-bit implementations such as the 620.

**REGISTERS AFFECTED**

- CR0[LGT,GT,EQ,SO](if \( Rc = 1 \))

**EXAMPLE**

See the example that accompanies the **clrlslwi** instruction.
**INTEGER UNIT**

**620**

**USER MODE**

**rotld**

**Rotate doubleword**

**REGISTER LEFT**

**FORMS**

\[ \text{rotld } rA, rS, rB = \text{ rldcl } rA, rS, rB, 0 \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x1e</th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>000000</th>
<th>0x08</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[ n \leftarrow rB[58-63] \]

\[ rA \leftarrow \text{ROTL}(rS,n) \]

**DESCRIPTION**

The rotld (rotate left doubleword) instruction rotates the contents of rS left the number of bits specified by the low-order 6 bits of rB and places the result into destination register rA. This instruction is a simplified form of the rldcl instruction.

**REGISTERS AFFECTED**

CR0[LT,GT,EQ,SO] (if Rc = 1)
**.rotldi**

**Rotate Register Left by Immediate**

**Forms**

\[ \text{rotldi } rA, rS, n = \text{rldcl } rA, rS, n, 0 \]

**Bit Definition**

<table>
<thead>
<tr>
<th></th>
<th>(S)</th>
<th>(A)</th>
<th>0x000000</th>
<th>0x00</th>
<th>(n^*)</th>
<th>(Rc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1e</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

*Note: This is a split field.*

**Pseudo Code**

\[ rA \leftarrow \text{ROTL}(rS, n) \]

**Description**

The rotldi (rotate left doubleword immediate) instruction rotates the contents of \(rS\) left by the number of bits specified by the \(n\) operand. The result is placed in destination register \(rA\).

**Registers Affected**

CR0[LT,GT,EQ,SO] (if \(Rc = 1\))
**INTEGER Unit**

**601/603/604/620**

**USER Mode**

**FORMS**

\[ \text{rotlw } rA, rS, rB \equiv \text{rlwnm } rA, rS, rB, 0, 31 \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>00000</th>
<th>31</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[ n \leftarrow rB[27-31] \]
\[ rA \leftarrow \text{ROTL}(rS,n) \]

**DESCRIPTION**

The **rotlw** (rotate left word) instruction is a simplified form of the **rlwnm** instruction. It rotates the contents of **rS** left by the number of bits specified by the low-order 5 bits of **rB**. The result is placed in destination register **rA**. This instruction is a simplified form of the **rlwnm** instruction.

**REGISTERS Affected**

CR0[LT,GT,EQ,SO](if Rc = 1)
**rotlwi**

**Rotate Register Left by Immediate**

**FORMS**

\[ \text{rotlwi } rA, rS, n = \text{rlwinm } rA, rS, n, 0, 31 \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x15</th>
<th>S</th>
<th>A</th>
<th>n</th>
<th>0</th>
<th>31</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[ rA \leftarrow \text{ROTL}(rS, n) \]

**DESCRIPTION**

The **rotlwi** (rotate left word immediate) instruction rotates the contents of \( rS \) left by the number of bits specified by the \( n \) operand. The result is placed in destination register \( rA \). This instruction is a simplified form of the **rlwinm** instruction.

**REGISTERS AFFECTED**

\( \text{CR0}[\text{LT,GT,EQ,SO}] \) (if \( \text{Rc} = 1 \))
INTEGER UNIT

620
USER MODE

FORMS
rotrdi rA,rS,n = rldicl rA,rS,64-n,0

BIT DEFINITION

<table>
<thead>
<tr>
<th>0x1e</th>
<th>64-n</th>
<th>000000</th>
<th>0x00</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>64-n</td>
</tr>
</tbody>
</table>

PSEUDO CODE

rA ← ROTR(rS, n)

DESCRIPTION

The rotrdi (rotate right doubleword immediate) instruction rotates the contents of rS left by the number of bits specified by the n operand. The result is placed in destination register rA. This instruction is a simplified form of the rldicl instruction.

REGISTERS AFFECTED

CR0[LT,GT,EQ,SO] (if Rc = 1)
**rotrwi**

**Rotate register right by immediate**

**FORMS**

\[ \text{rotrwi} \ rA, rS, n \quad \Rightarrow \quad \text{rlwinm} \ rA, rS, 32-n, 0, 31 \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ox15</td>
<td>S</td>
<td>0</td>
<td>Destination register rA</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>1</td>
<td>Source register rS</td>
</tr>
<tr>
<td></td>
<td>32 - n</td>
<td>15</td>
<td>Number of bits to rotate right</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rc</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[ rA \leftarrow \text{ROTR}(rS, n) \]

**DESCRIPTION**

The **rotrwi** (rotate right word immediate) instruction rotates the contents of rS right by the number of bits specified by the \( n \) operand. The result is placed in destination register rA. This instruction is a simplified form of the **rlwinm** instruction.

**REGISTERS AFFECTED**

CR0[LT,GT,EQ,SO] (if Rc = 1)
**SC**

**System Call**

### INTEGER UNIT

**601/603/604/620**

**User Mode**

#### FORMS

- `sc`

#### BIT DEFINITION

<table>
<thead>
<tr>
<th></th>
<th>Reserved</th>
<th>Reserved</th>
<th>Reserved</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x11</td>
<td>0000</td>
<td>0000</td>
<td>0000000000000000</td>
</tr>
</tbody>
</table>

|   |   0   |   1   |   2   |   3   |   4   |   5   |   6   |   7   |   8   |   9   |   10  |   11  |   12  |   13  |   14  |   15  |   16  |   17  |   18  |   19  |   20  |   21  |   22  |   23  |   24  |   25  |   26  |   27  |   28  |   29  |   30  |   31  |
---|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|

#### PSEUDO CODE

- `SRR0 ← ie a CIA+4`
- `SRR1[33-36,42-47] ← 0`
- `SRR1[0-32,37-41,48-63] ← MSR[0-32,37-41,48-63]`
- `MSR ← new_value (see Chapter 10)`
- `NIA ← ie a Vector_Base_EA + 0x00c00`

#### DESCRIPTION

The `sc` (system call) instruction calls the operating system to perform a service. When control is returned to the program that executed the system call, the content of the registers depends on the register conventions used by the program providing the system service. This instruction is context synchronizing.

The `sc` instruction generates an SC exception is generated, which causes the next instruction to be fetched from the system call vector. The vector is located at offset 0x00c00 from the vector base address, as defined by MSR[IP].

#### REGISTERS AFFECTED

Dependent on the system service
**slbia**

**INVALIDATE SEGMENT LOOKASIDE BUFFER**

**FORMS**
- slbia rB

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td>00000</td>
<td>00000</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

All SLB entries ← invalid

**DESCRIPTION**

The slbia (SLB invalidate all) instruction invalidates the entire segment lookaside buffer (SLB), that is, all entries are removed. The SLB is invalidated regardless of the settings of MSR[IR] and MSR[DR]. This instruction is optional in the PowerPC architecture. It is not necessary that the address space register (ASR) point to a valid segment table when issuing slbia.

**REGISTERS AFFECTED**

CR0[LT,GT,EQ,SO] (if Rc = 1)
**slbie**

**INVALIDATE SEGMENT LOOKASIDE BUFFER ENTRY**

**FORMS**
- slbie rB

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Res.</td>
<td>0x1f</td>
</tr>
<tr>
<td>1</td>
<td>Res.</td>
<td>0000</td>
</tr>
<tr>
<td>2</td>
<td>Res.</td>
<td>0000</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Res.</td>
<td>0x1B2</td>
</tr>
<tr>
<td>5</td>
<td>Res.</td>
<td>0</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

```plaintext
EA ← rB
if SLB entries exists for EA, then
    SLB entry ← invalid
```

**DESCRIPTION**

The slbie (SLB invalidate entry) instruction invalidates (that is, removes from the SLB) an entry corresponding to EA if contained in the segment lookaside buffer (SLB). The effective address (EA) is the contents of rB. The SLB search is done regardless of the settings of MSR[IR] and MSR[DR]. Block address translation for the EA, if any, is ignored.

Bits 11–15 of this instruction (ordinarily the position of an rA field) must be zero. This provides implementations the option of using (rA[0] + rB address arithmetic for this instruction. This instruction is supervisor-level and is optional in the PowerPC architecture. It is not necessary that the ASR point to a valid segment table when issuing slbie.

**REGISTERS AFFECTED**

None
**sldx**

**SHIFT DOUBLEWORD REGISTER LEFT**

**FORMS**

<table>
<thead>
<tr>
<th>Form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sld rA,rS,rB</td>
<td>Rc</td>
</tr>
<tr>
<td>sld. rA,rS,rB</td>
<td>0</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Res.</th>
<th>0x1f</th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>0x1b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

```plaintext```
```
function sld (rS, rB, n) {
  if (rB[57] == 0) {
    m = MASK(0, 63 - n);
  } else {
    m = (64)0;
  }
  rA = (r & m) << n;
}
```

**DESCRIPTION**

The **sld** (shift left doubleword) instruction shifts `rS` left by the number of bits specified by the low-order 7 bits of `rB`. Bits shifted out of position 0 are lost. Zeroes are shifted in on the right. The result is placed into `rA`. Shift amounts from 64 to 127 give a zero result.

**REGISTERS AFFECTED**

None
**INTEGER UNIT**

**620**

**USER MODE**

### sldi

**SHIFT REGISTER LEFT BY IMMEDIATE**

#### FORMS

\[ \text{sldi } rA, rS, n \text{ (n<64)} \equiv \text{ rldicr } rA, rS, n, 63-n \]

#### BIT DEFINITION

<table>
<thead>
<tr>
<th>0x1e</th>
<th>S</th>
<th>A</th>
<th>n*</th>
<th>63-n</th>
<th>0x01</th>
<th>n*</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### PSEUDO CODE

\[
r \leftarrow \text{ROTL}(rS, n) \\
m \leftarrow \text{MASK}(0, 63-n) \\
rA \leftarrow (r \& m)
\]

#### DESCRIPTION

The sldi (shift left doubleword immediate) instruction shifts the contents of rS left by the number of bits specified by the \( n \) operand. The result is placed in destination register \( rA \). This instruction is a simplified form of the rldicr instruction.

#### REGISTERS AFFECTED

\[ \text{CR0[LT,GT,EQ,SO]} \text{ (if Rc = 1)} \]
slwx
SHIFT REGISTER LEFT

FORMS

\[
\begin{align*}
& \text{slw} & & rA, rS, rB & & 0 \\
& \text{slw} & & rA, rS, rB & & 1
\end{align*}
\]

BIT DEFINITION

<table>
<thead>
<tr>
<th>S</th>
<th>A</th>
<th>B</th>
<th>0x18</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

PSEUDO CODE

\[
\begin{align*}
\text{if } rB[16] = 1 & \text{ then} \\
& rA \leftarrow 0 \\
\text{else} & \\
& n \leftarrow rB[27-31] \\
& rA \leftarrow \text{ROTL}(rS,n)
\end{align*}
\]

DESCRIPTION

The slw (shift left word) instruction shifts rS left. If \( rB[16] = 0 \), the contents of rS are shifted left by the number of bits specified by \( rB[26-31] \). Bits shifted out of position 0 are lost. Zeroes supplied vacated positions on the right. The 32-bit result is placed into rA. If bit \( rB[16] = 1 \), 32 zeroes are placed into rA. Shift amounts from 32 to 63 give a zero result.

REGISTERS AFFECTED

CR0[LT,GT,EQ,SO] (if Rc = 1)
### INTEGER UNIT

**601/603/604/620**  
**User Mode**

#### FORMS

\[ \text{slwi } rA, rS, n \quad (n<32) \quad \equiv \quad \text{rlwinm } rA, rS, n, 0, 31-n \]

#### BIT DEFINITION

<table>
<thead>
<tr>
<th>0x15</th>
<th>S</th>
<th>A</th>
<th>n</th>
<th>0</th>
<th>31-n</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### PSEUDO CODE

\[
\begin{align*}
    r & \leftarrow \text{ROTL}(rS, n) \\
    m & \leftarrow \text{MASK}(0, 31-n) \\
    rA & \leftarrow (r \& m)
\end{align*}
\]

#### DESCRIPTION

The **slwi** (shift left word immediate) instruction shifts \( rS \) left by the number of bits specified by the \( n \) operand. The result is placed in destination register \( rA \). This instruction is a simplified form of the **rlwinm** instruction.

#### REGISTERS AFFECTED

CR0[LT,GT,EQ,SO] (if \( Rc = 1 \))

#### EXAMPLE

if (word1 == 0x10)  
    word1 = (word1 << 1);

; Assumes:
; \( r3 = 32\)-bit word1
;
If Else:
    cmpwi r3,0x10 ; (IF) - compare immediate: \( r3 == 0x10 \)?
    bne Around1 ; branch if not equal to Around1
    slwi r3,r3,1 ; (STM T1) shift left immediate 1 bit
    Around1: ; execution continues as normal
**sradx**

**SHIFT REGISTER**

**RIGHT ALGEBRAIC**

**FORMS**

<table>
<thead>
<tr>
<th>Srad x</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>srad rA,rS,rB</td>
<td>0</td>
</tr>
<tr>
<td>srad x rA,rS,rB</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ox1f</th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>Ox31a</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

**PSEUDO CODE**

```
n ← rB[58-63]
r ← ROTR(rS,n)
if rB[57]=0 then
    m ← MASK(n,63)
else
    m ← (64)0
S ← rS[0]
rA ← (r & m) | (64)S & NOT m)
XER[CA] ← S & ((r & NOT m) = 0)
```

**DESCRIPTION**

The `sradx` (shift right algebraic doubleword) instruction shifts the contents of `rS` right the number of bits specified by the low-order 7 bits of `rB`. Bits shifted out of position 63 are lost. Bit 0 of `rS` is replicated to fill the vacated positions on the left. The result is placed into `rA`. `XER[CA]` is set if `rS` is negative and any 1 bits are shifted out of position 63; otherwise `XER[CA]` is cleared. Shift amounts from 64 to 127 give a result of 64 sign bits in `rA`, and cause `XER[CA]` to receive the sign bit of `rS`.

Note that the `sradx` instruction, followed by `addze`, can be used to divide quickly by $2^n$. The setting of the CA bit, by `sradx`, is mode independent.

**REGISTERS AFFECTED**

- CR0[LT,GT,EQ,SO] (if Rc = 1)
- XER[CA]
**INTEGER UNIT**

**620 USER MODE**

---

**sradix**

**SHIFT REGISTER RIGHT**

**ALGEBRAIC BY IMMEDIATE**

---

**FORMS**

\[
\begin{align*}
\text{srad} & \; \text{rA}, \text{rS}, \text{rB} & \text{Rc} \\
\text{srad} & . \; \text{rA}, \text{rS}, \text{rB} & 1
\end{align*}
\]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x1f</th>
<th>S</th>
<th>A</th>
<th>sh*</th>
<th>0x19d</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

*Note: This is a split field.*

---

**PSEUDO CODE**

\[
\begin{align*}
\text{n} & \leftarrow \text{sh}[5] \mid \mid \text{sh}[0-4] \\
\text{r} & \leftarrow \text{ROTR}(\text{rS}, \text{n}) \\
\text{m} & \leftarrow \text{MASK}(\text{n}, 63) \\
\text{S} & \leftarrow \text{rS}[0] \\
\text{rA} & \leftarrow (\text{r} \& \text{m}) \mid ((64 \& \text{S} \& \text{NOT} \text{m}) \\
\text{XER}[\text{CA}] & \leftarrow \text{S} \& ((\text{r} \& \text{NOT} \text{m}) \& 0)
\end{align*}
\]

**DESCRIPTION**

The srad (shift right algebraic doubleword immediate) instruction shifts the contents of rS right by SH bits. Bit 0 of rS is replicated to fill the vacated positions on the left. The result is placed into rA. XER[CA] is set if rS is negative and any 1 bits are shifted out of position 63; otherwise XER[CA] is cleared. A shift amount of zero causes rA to be set equal to rS, and XER[CA] to be cleared.

Note that the srad instruction, followed by addze, can be used to divide quickly by \(2^n\). The setting of the XER[CA] bit, by srad, is independent of mode.

**REGISTERS AFFECTED**

- CR0[LT,GT,EQ,SO] (if Rc = 1)
- XER[CA]
srawx
SHIFT REGISTER
RIGHT ALGEBRAIC

FORMS

\[
\begin{array}{ccc}
\text{sraw} & \text{rA,rS,rB} & 0 \\
\text{sraw.} & \text{rA,rS,rB} & 1
\end{array}
\]

BIT DEFINITION

<table>
<thead>
<tr>
<th>0x1f</th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>0x318</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

PSEUDO CODE

if rB[16] = 0 then
    n ← rB[27-31]
    rA ← EXTS(ROTR(rS,n))
else
    rA ← (32)rS[0]

DESCRIPTION

The sraw (shift right algebraic word) instruction shifts rS right as a signed integer. If rB[26] = 0, then the contents of rS are shifted right the number of bits specified by rB[27-31]. Bits shifted out of position 31 are lost. The result is padded on the left with sign bits before being placed into rA. If rB[26] = 1, then rA is filled with 32 sign bits (bit 0) from rS. CR0 is set based on the value written into rA.

REGISTERS AFFECTED

- CR0[LT,GT,EQ,SO](if Rc = 1)
- XER[CA]
**INTEGER UNIT**

601/603/604/620

*User Mode*

**FORMS**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>REGISTERS</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>srawi</td>
<td>rA,rS,SH</td>
<td>0</td>
</tr>
<tr>
<td>srawi</td>
<td>rA,rS,SH</td>
<td>1</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>0x1f</th>
<th>S</th>
<th>A</th>
<th>SH</th>
<th>0x338</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>30</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>29</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>28</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>27</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>26</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[
m \leftarrow \text{MASK}(SH,31) \\
b \leftarrow (rS \& m) \\
r \leftarrow \text{ROTR}(b,SH) \\
rA \leftarrow \text{EXTS}(r)
\]

**DESCRIPTION**

The `srawi` (shift right algebraic word immediate) instruction shifts `rS` to the right by `SH` bits. Bits shifted out of position 31 are lost. The shifted value is sign extended before being placed in `rA`. XER[CA] is set to 1 if `rS` contains a negative number and any 1 bits are shifted out of position 31; otherwise XER[CA] is cleared to 0. A shift amount of zero causes XER[CA] to be cleared to 0.

**REGISTERS AFFECTED**

- CR0[LT,GT,EQ,SO](if Rc = 1)
- XER[CA]
**srdx**

**SHIFT DOUBLEWORD REGISTER RIGHT**

**FORMS**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>srd</td>
<td>rA, rS, rB</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>srd</td>
<td>rA, rS, rB</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td>S</td>
<td>A</td>
<td>B</td>
<td>0x21b</td>
<td>Rc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[
\begin{align*}
    n & \leftarrow rB[50-63] \\
    r & \leftarrow \text{ROTR}(rS, n) \\
    \text{if } rB[57]=0 \text{ then} \\
    & \quad m \leftarrow \text{MASK}(n, 63) \\
    \text{else} \\
    & \quad m \leftarrow (64)0 \\
    rA & \leftarrow r \& m
\end{align*}
\]

**DESCRIPTION**

The srd (shift right doubleword) instruction shifts the contents of rS right by the number of bits specified by the low-order 7 bits of rB. Bits shifted out of position 63 are lost. Zeroes are supplied to the vacated positions on the left. The result is placed into rA. Shift amounts from 64 to 127 give a zero result.

**REGISTERS AFFECTED**

- CR0[LT,GT,EQ,SO] (if Rc = 1)
- XER[CA]
**INTEGER UNIT**

620

**User Mode**

**FORMS**

\[
\text{srdi } rA, rS, n \quad (n<64) \quad \equiv \quad \text{rldicl } rA, rS, 64-n, n
\]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x1e</th>
<th>S</th>
<th>A</th>
<th>64-n*</th>
<th>n</th>
<th>0x00</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

*Note: This is a split field.

**PSEUDO CODE**

\[
r \leftarrow \text{ROTR}(rS, n)
m \leftarrow \text{MASK}(n, 63)
rA \leftarrow (r \& m)
\]

**DESCRIPTION**

The `srdi` (shift right doubleword immediate) instruction shifts the contents of `rS` right by the number of bits specified by the `n` operand. The result is placed in destination register `rA`. This instruction is a simplified form of the `rldicl` instruction.

**REGISTERS AFFECTED**

CR0[LT,GT,EQ,SO] (if Rc = 1)
srwx
SHIFT REGISTER RIGHT

**FORMS**

<table>
<thead>
<tr>
<th>srw</th>
<th>rA,rS,rB</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>srwx</td>
<td>rA,rS,rB</td>
<td>1</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Ox1f</th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>Ox218</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

```
n ← rB[26-31]
rA ← ROTR(rS,n)
```

**DESCRIPTION**

The `srw` (shift right word) instruction shifts `rS` right.

If `rB[26] = 0`, the contents of `rA` are shifted right the number of bits specified by `rA[26-31]`. Bits shifted out of position 31 are lost. Zeroes are supplied to the vacated positions on the left. The 32-bit result is placed into `rA`. If `rB[26] = 1`, then `rA` is filled with zeroes. That is, shift amounts from 32 to 63 give a zero result.

**REGISTERS AFFECTED**

CR0[LT,GT,EQ,SO] (if Rc = 1)
The `srwi` (shift right word immediate) instruction shifts the contents of RS left by the number of bits specified by the `n` operand. The result is placed in destination register RA. This instruction is a simplified form of the `rlwinm` instruction.

**Description**

The `srwi` (shift right word immediate) instruction shifts the contents of RS left by the number of bits specified by the `n` operand. The result is placed in destination register RA. This instruction is a simplified form of the `rlwinm` instruction.

**Registers Affected**

CR0[LT,GT,EQ,SO] (if Rc = 1)
**stb**

**STORE BYTE FROM REGISTER TO MEMORY**

**FORMS**

\[ \text{stb} \quad r_S, d(r_A) \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Bit</th>
<th>S</th>
<th>A</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x26</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[
\begin{align*}
\text{if } r_A &= r_0 \text{ then } b \leftarrow 0 \\
\text{else } b &\leftarrow r_A \\
EA &\leftarrow b + \text{EXTS}(d) \\
\text{MEM}(EA, 1) &\leftarrow r_S[24-31]
\end{align*}
\]

**DESCRIPTION**

The **stb** (store byte) instruction stores a byte from \( r_S[24-31] \) into memory at the effective address (EA) specified by the sum \((r_A) + d\), where \( d \) is a 16-bit signed value that is sign-extended to 32 bits (64 bits on 64-bit implementations). Register \( r_S[24-31] \) is stored into the byte in memory addressed by EA. Register \( r_S \) is unchanged. On 64-bit implementations, \( r_S[56-63] \) is stored into the byte in memory addressed by EA.

**REGISTERS AFFECTED**

None

**EXAMPLE**

\[
\begin{align*}
\text{templ} &= 0; \\
\text{temp2} &= \text{templ}; \\
\text{li} \quad r_4, 0 &; \text{load zero into } r_4 \\
\text{stb} \quad r_4, 0(r_5) &; \text{store } 0 \text{ into } \text{templ}; \text{address in } r_5 \\
\text{stb} \quad r_4, 0(r_6) &; \text{store } 0 \text{ into } \text{temp2}; \text{address in } r_6
\end{align*}
\]
INTEGER UNIT

601/603/604/620
User Mode

FORMS
stbu rS, d(rA)

BIT DEFINITION

<table>
<thead>
<tr>
<th>0x27</th>
<th>S</th>
<th>A</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

PSEUDO CODE

EA ← rA + EXTS(d)
MEM(EA, 1) ← rS[24-31]

DESCRIPTION

The stbu (store byte with update) instruction stores a byte from rS[24-31] into memory at the effective address (EA) specified by the sum (rA + d), where d is a 16-bit signed value that is sign-extended to 32 bits (64 bits on 64-bit implementations). On 64-bit implementations, rS[56-63] is stored into the byte in memory addressed by EA. The EA is placed into rA.

The PowerPC architecture defines the instruction form as invalid if rA = 0, but the 601 supports execution with rA = 0 as shown above. On the 603, 604, and 620 rA = 0 is invalid.

REGISTERS AFFECTED

None

EXAMPLE

for (r=0; r<10; r++)
    bytel[r] = byte2[r];    // globally declared arrays of unsigned chars

; Assumes:
; r3 = contains address of byte2 array
; r4 = contains address of bytel array
;
    li  r5, 0 ; zero r5: used as 'r'
    subi r4, r4, 1 ; adjust index for use w/ update
    subi r3, r3, 1 ; adjust index for use w/ update

LOOP:
    lbzu r6, 1(r3) ; get byte from r3 and update r3
    addi r5, r5, 1 ; inc r5
    cmpi 0x6, 0x0, r5, 10 ; crf6, 32bit compare r5 to 10
    stbu r6, 1(r4) ; store byte to bytel array
    bc 0xc, 0x18, LOOP ; branch to LOOP if (r5 < 10)
**stbux**

**Store byte with EA update using indexed addressing**

**Forms**

stbux  rS,rA,rB

**Bit Definition**

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>0xff7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

**Pseudo Code**

EA ← rA + rB
MEM(EA,1) ← rS[24-31]
rA ← EA

**Description**

The `stbux` (store byte with update indexed) instruction stores a byte from rS[24-31] into memory at the effective address (EA) specified by the sum (rA) + rB. The EA is placed into rA. On 64-bit implementations, rS[56-63] is stored into the byte in memory addressed by EA.

The PowerPC architecture defines the instruction form as invalid if rA = 0, but the 601 supports execution with rA = 0 as shown above. On the 603, 604, and 620 rA = 0 is invalid.

**Registers Affected**

None
**INTEGER UNIT**

**601/603/604/620**

**USER**

**stbx**

*Store byte using indexed addressing*

**FORMS**

`stbx  rS, rA, rB`

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>0xd7</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>Res.</td>
<td>0</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

```
if rA = r0 then b ← 0
else b ← rA
EA ← b + rB
MEM(EA, 1) ← rS[24-31]
```

**DESCRIPTION**

The `stbx` (store byte indexed) instruction stores a byte from `rS[24-31]` into memory at the effective address (EA) specified by the sum `(rA)[0] + rB`. The contents of `rS[24-31]` are stored into the byte in memory addressed by EA. Register `rS` is unchanged. On 64-bit implementations, `rS[56-63]` is stored into the byte in memory addressed by EA.

**REGISTERS AFFECTED**

None
**std**

**STORE DOUBLEWORD FROM REGISTER TO MEMORY**

**FORMS**

std rS,ds(rA)

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>0x3e</th>
<th>S</th>
<th>A</th>
<th>ds</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

if rA = r0 then b ← 0
else b ← rA
EA ← b + EXTS(ds || 0b00)
(MEM(EA,b)) ← rS

**DESCRIPTION**

The std (store doubleword) instruction stores the contents of rS into the doubleword in memory addressed by EA. The effective address (EA) is the sum (rA|0) + (ds || 0b00). Note that ds is a 14-bit signed value which is concatenated on the right with 0b00; this 16-bit value is sign-extended to 64 bits.

**REGISTERS AFFECTED**

None
**INTEGER UNIT**

**620**  
**USER MODE**

**FORMS**  
stdcx. rS,rA,rB

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>S</th>
<th>A</th>
<th>B</th>
<th>0xd6</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

```plaintext
if rA = r0 then b ← 0
else b ← rA
EA ← b + rB
if RESERVE then
  if RESERVE_ADDR = physical_addr(EA)
    MEM(EA,8) ← rS
    CR0 EA ← (0b00 || 0b1 || XER[SO])
  else
    u ← undefined 1-bit value
    if u then MEM(EA,8) ← rS
      CR0 ← (0b00 || u || XER[SO])
    RESERVE ← 0
else
  CR0 ← (0b00 || 0b0 || XER[SO])
```

**DESCRIPTION**

The stdcx. (store doubleword conditional indexed) instruction stores the contents of rS at the effective address (EA) specified by the sum (rA[0] + rB).

If a reservation exists, and the memory address specified by the stdcx. instruction is the same as that specified by the load and reserve instruction that established the reservation, the contents of rS are stored into the doubleword in memory addressed by EA and the reservation is cleared.

If a reservation exists, but the memory address specified by the stdcx. instruction is not the same as that specified by the load and reserve instruction that established the reservation, the reservation is cleared, and it is undefined whether the contents of rS are stored into the doubleword in memory addressed by EA.

If no reservation exists, the instruction completes without altering memory.
CR0 Field is set to reflect whether the store operation was performed as follows.

\[
\text{CR0[LT GT EQ SO]} = 0b00 \| \text{store\_performed} \| \text{XER[SO]}
\]

EA must be a multiple of 8. If it is not, either the system alignment exception handler is invoked or the results are boundedly undefined. For additional information about alignment and DSI exception, see Chapter 10, "Exceptions and Interrupts."

Note that, when used correctly, the load and reserve and store conditional instructions can provide and atomic update function for a single aligned word (load word and reserve and store word conditional) or doubleword (load doubleword and reserve and store doubleword conditional) of memory.

In general, correct use requires that load word and reserve be paired with store word conditional, and load doubleword and reserve with store doubleword conditional, with the same memory address specified by both instructions of the pair. The only exception is that an unpaired store word conditional or store doubleword conditional instruction to any (scratch) EA can be used to clear any reservation held by the processor. Examples of correct uses of these instructions, to emulate primitives such as fetch and add, test and set, and compare and swap can be found in Chapter 11, "PowerPC Assembly Language Examples."

A reservation is cleared if any of the following events occur:

- The processor holding the reservation executes another load and reserve instruction; this clears the first reservation and establishes a new one.
- The processor holding the reservation executes a store conditional instruction to any address.
- Another processor executes any store instruction to the address associated with reservation.
- Any mechanism, other than the processor holding the reservation, stores to the address associated with the reservation.

**REGISTERS AFFECTED**

CR0[LT,GT,EQ,SO]
**INTEGER UNIT**

**620**

**User Mode**

**STDU**

**Store Doubleword with EA Update**

**Forms**

\[ \text{stdu } rS,ds(rA) \]

**Bit Definition**

<table>
<thead>
<tr>
<th>0x3e</th>
<th>S</th>
<th>A</th>
<th>ds</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pseudo Code**

EA ← rA + EXTS(ds || 0b00)
MEM(EA,B) ← rS
rA ← EA

**Description**

The stdu (store doubleword with update) instruction stores the contents of rS into the doubleword in memory addressed by the effective address (EA) specified by the sum rA + (ds || 0b00). Note that ds is a 14-bit signed value which is concatenated on the right with 0b00; this 16-bit value is sign-extended to 64-bits. If rA = 0, the instruction form is invalid.

**Registers Affected**

CR0[LT,GT,EQ,SO]
stdux
STORE DOUBLEWORD WITH
EA UPDATE USING
INDEXED ADDRESSING

FORMS
stdux rS,rA,rB

BIT DEFINITION

<table>
<thead>
<tr>
<th>0x1f</th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>0xb5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PSEUDO CODE
EA ← rA + rB
MEM(EA,B) ← rS
rA ← EA

DESCRIPTION
The stdux (store doubleword with update indexed) instruction stores the contents of rS into the doubleword in memory addressed by the effective address (EA) specified by the sum rA + rB. The EA is placed into rA. If rA = 0, the instruction form is invalid.

REGISTERS AFFECTED
None
INTEGER UNIT

620
USER MODE

FORMS
stdx rS,rA,rB

BIT DEFINITION

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>0x95</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PSEUDO CODE

if rA = r0 then b ← 0
else b ← rA
EA ← b + rB
MEM(EA,8) ← rS

DESCRIPTION

The stdx (store doubleword indexed) instruction stores the contents of rS into the doubleword in memory addressed by the effective address (EA) specified by the sum (rA[0]) + rB.

REGISTERS AFFECTED

None
**stfd**

**STORE FLOATING-POINT DOUBLE-PRECISION VALUE**

**FORMS**

\[ \text{stfd frS,d(rA)} \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x36</th>
<th>frS</th>
<th>A</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[
\begin{align*}
\text{if} & \ rA = r0 \ \text{then} \ b \leftarrow 0 \\
\text{else} & \ b \leftarrow rA \\
\text{EA} & \leftarrow b + \text{EXTS}(d) \\
\text{MEM(EA,8)} & \leftarrow \text{frS}
\end{align*}
\]

**DESCRIPTION**

The stfd (store floating-point double) instruction stores the contents of frS into the doubleword in memory addressed by EA. The effective address (EA) is the sum \((rA) + d\), where \(d\) is a 16-bit signed value that is sign-extended to 32 bits (64 bits on 64-bit implementations).

**REGISTERS AFFECTED**

None

**EXAMPLE**

```plaintext```
double1 = 3.1415;       // globally declared doubles
double2 *= double1;
```

; Assumes:
; r3 = contains address of constant data
; r4 = contains address of double1
; r5 = contains address of double2
;
; lfs f1, 0(r3) ; get 3.1415 const.
; stfd f1, 0(r4) ; do assignment
; lfd f2, 0(r5) ; get value of double2
; fmul f1, f1, f2 ; multiply them
; stfd f1, 0(r3) ; store double results
FLOATING-POINT UNIT

601/603/604/620

USER MODE

FORMS

\texttt{stfdu \ frS,d(rA)}

BIT DEFINITION

<table>
<thead>
<tr>
<th>0x37</th>
<th>frS</th>
<th>A</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

PSEUDO CODE

\begin{verbatim}
if rA = r0 then b ← 0
else b ← rA
EA ← b + d
MEM(EA,4) ← SINGLE(frS)
rA ← EA
\end{verbatim}

DESCRIPTION

The \texttt{stfdu} (store floating-point double with update) instruction stores the contents of \texttt{frS} into the doubleword in memory addressed by \texttt{EA}. The effective address (EA) is the sum (\texttt{rA}+d). The EA is placed into \texttt{rA}.

The PowerPC architecture defines the instruction form as invalid if \texttt{rA} = 0, but the 601 supports execution with \texttt{rA} = 0 as shown above. On the 603, 604, and 620 \texttt{rA} = 0 is invalid.

REGISTERS AFFECTED

None
**stfdux**

**STORE FP DOUBLE-PRECISION VALUE WITH EA UPDATE USING INDEXED ADDRESSING**

**FORMS**

stfdux frS,rA,rB

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>0x1f</th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>0x2f7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

EA ← rA + rB
MEM(EA, 8) ← (frS)
rA ← EA

**DESCRIPTION**

The stfdux (store floating-point double with update indexed) instruction stores the contents of frS into the doubleword in memory addressed by EA. The effective address (EA) is the sum (rA+0) + rB. The EA is placed into rA.

The PowerPC architecture defines the instruction form as invalid if rA = 0, but the 601 supports execution with rA = 0 as shown above. On the 603, 604, and 620 rA = 0 is invalid.

**REGISTERS AFFECTED**

None
**FLOATING-POINT UNIT**

**601/603/604/620**

**User Mode**

**FORMS**

stfdx frS, rA, rB

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>A</th>
<th>B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

if rA = r0 then b ← 0
else b ← rA
EA ← b + rB
MEM(EA, 8) ← (frS)

**DESCRIPTION**

The stfdx (store floating-point double indexed) instruction stores the contents of register frS into the doubleword in memory addressed by EA. The effective address (EA) is the sum (rA0) + rB.

**REGISTERS AFFECTED**

None
**stfiwx**

**STORE FLOATING-POINT VALUE AS INTEGER WORD USING INDEXED ADDRESSING**

**FORMS**

stfiwx frS,rA,rB

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>0x1f</th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>0x3d7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

if rA = r0 then b ← 0
else b ← rA
EA ← b + rB
MEM(EA,4) ← (frS)

**DESCRIPTION**

The stfiwx (store floating-point as integer word indexed) instruction stores the low-order 32 bits of frS, without conversion, into the word in memory addressed by EA. The effective address (EA) is the sum (rA+0) + rB.

The floating-point value in frS is not converted to an integer value; but rather stored, in floating-point format, to the word in memory addressed by EA. This instruction is optional in the PowerPC architecture.

If the contents of register frS were produced, either directly or indirectly, by an Ifs instruction, a single-precision arithmetic instruction, or frsp, then the value stored is undefined. The contents of frS are produced directly by such an instruction if frS is the target register for the instruction. The contents of frS are produced indirectly by such an instruction if frS is the final target register of a sequence of a floating-point move instruction, with the input to the sequence having been produced by such an instruction.

**REGISTERS AFFECTED**

None
**INTEGER UNIT AND FLOATING-POINT UNIT**

**601/603/604/620 USER MODE**

**FORMS**

\[ \text{stfs frS, d(rA)} \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x34</th>
<th>S</th>
<th>A</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

\[
\begin{align*}
\text{if } rA &= r0 \text{ then } b \leftarrow 0 \\
\text{else } b &\leftarrow rA \\
\text{EA} &\leftarrow b + \text{EXTS}(d) \\
\text{MEM}(\text{EA}, 4) &\leftarrow \text{SINGLE}(frS)
\end{align*}
\]

**DESCRIPTION**

The **stfs** (store floating-point single) instruction converts the contents of frS to single-precision and stores the result into the word in memory addressed by EA. The effective address (EA) is the sum \((rA0) + d\), where \(d\) is a 16-bit signed value that is sign-extended to 32 bits (64 bits on 64-bit implementations).

**REGISTERS AFFECTED**

None

**EXAMPLE**

\[
\text{tmpfl} = 3.14157; \text{tmpf2} = 10.5; \quad \text{// globally declared floats} \\
\text{tmpf2} *= \text{tmpfl};
\]

: Assumes
: \( r3 \) contains address of constant data
: \( r4 \) contains address of \( \text{tmpfl} \)
: \( r5 \) contains address of \( \text{tmpf2} \)

: lfs \( f1, 0(r3) \) ; load fp value from \( r3 \\
lfs \( f2, 4(r3) \) ; load fp value from \( r3 + 4 \\
stfs \( f1, 0(r4) \) ; initialize \( \text{tmpfl} \\
fmul \( f1, f1, f2 \) ; do multiply \\
stfs \( f1, 0(r5) \) ; save result in \( \text{tmpf2} \)
**stfsu**

**STORE FLOATING-POINT SINGLE-PRECISION VALUE WITH EA UPDATE**

**FORMS**

stfsu frS,d(rA)

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x35</th>
<th>S</th>
<th>A</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

EA ← b + EXTS(d)
MEM(EA,4) ← SINGLE(frS)
rA ← EA

**DESCRIPTION**

The stfsu (store floating-point single with update) instruction converts the contents of frS to single-precision and stores the result into the word in memory addressed by EA. The effective address (EA) is the sum (rA10) + d, where d is a 16-bit signed value that is sign-extended to 32 bits (64 bits on 64-bit implementations). The EA is placed into rA.

The PowerPC architecture defines the instruction form as invalid if rA = 0, but the 601 supports execution with rA = 0 as shown above. On the 603, 604, and 620 rA = 0 is invalid.

**REGISTERS AFFECTED**

None
**INTEGER UNIT AND FLOATING-POINT UNIT**

**601/603/604/620 User Mode**

**FORMS**

```plaintext
stfsux  frS,rA,rB
```

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td>0</td>
<td></td>
<td></td>
<td>0x2b7</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

EA ← b + rB  
MEM(EA,4) ← SINGLE(frS)  
rA ← EA

**DESCRIPTION**

The `stfsux` (store floating-point single with update indexed) instruction converts the contents of register frS to single-precision and stores the result into the word in memory addressed by EA. The effective address (EA) is the sum (rAlo) + rB. The EA is placed into rA.

The PowerPC architecture defines the instruction form as invalid if rA = 0, but the 601 supports execution with rA = 0 as shown above. On the 603, 604, and 620 rA = 0 is invalid.

**REGISTERS AFFECTED**

None
The `stfsx` (store floating-point single indexed) instruction converts the contents of register `frS` to single-precision and stores the result into the word in memory addressed by `EA`. The effective address (EA) is the sum `(rA + rB)`. The EA is placed into `rA`.

**REGISTERS AFFECTED**

None

**Example**

```cpp
for(r=0; r<10; r++)
    sf2[r] = sf1[r];  // sf1, sf2 are arrays of type float
```

; Assumes:
; r5 = contains address of sf1 array
; r3 = contains address of sf2 array
;
```
li r6, 0 ; zero r6; use as counter
li r4, 0 ; zero r4
```

```
LOOP:
    lfsx f1, r5, r4 ; load our float value into f1
    addi r6, r6, 1 ; inc for-loop counter
    cmpwi r6, 10 ; done yet? (are we at 10)
    stfsx f1, r3, r4 ; store float value into sf2[]
    addi r4, r4, 4 ; increment index value
    blt LOOP ; so are we done yet? If not, loop
```
INTEGER UNIT
601/603/604/620
User Mode

STORE A HALF-WORD TO MEMORY

**Forms**

\text{sth } rS,d(rA)

**Bit Definition**

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>A</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2c</td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

**Pseudo Code**

\[
\text{if } rA = r0 \text{ then } b \leftarrow 0 \\
\text{else } b \leftarrow rA \\
EA \leftarrow b + \text{EXTS}(d) \\
\text{MEM}(EA, 2) \leftarrow rS[16-31]
\]

**Description**

The \text{sth} (store half-word) instruction stores the low-order 16 bits of \text{rS} into the half-word in memory addressed by \text{EA}. The effective address (EA) is the sum \((rA0) + d\), where \(d\) is a 16-bit signed value that is sign-extended to 32 bits (64 bits on 64-bit implementations). On 32-bit implementations, \(rS[16-31]\) is stored into the half-word in memory addressed by \(\text{EA}\). On 64-bit implementations, \(rS[48-63]\) is stored into the half-word in memory addressed by \(\text{EA}\).

**Registers Affected**

None

**Example**

\text{short1} = 10; \\
\text{short2} = \text{short1}; \quad // \text{globally declared shorts (single-precision)}

; Assumes: \\
; \text{r3} = \text{initially contains address of short1} \\
; \text{r5} = \text{contains address of short2} \\
;
li \quad r4, 10 \quad ; \text{load r4 with immediate value 10} \\
sth \quad r4, 0(r3) \quad ; \text{store 10 into short1} \\
lha \quad r3, 0(r3) \quad ; \text{get that value back} \\
sth \quad r3, 0(r5) \quad ; \text{store value in short2}
STHBRX
STORE BYTE-REVERSED
HALF-WORD USING INDEXED
ADDRESSING

FORMS
sthbrx rS,rA,rB

BIT DEFINITION

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>A</th>
<th>B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x1f</td>
<td></td>
<td></td>
<td>Res. 0x396</td>
</tr>
</tbody>
</table>

PSEUDO CODE

if rA = r0 then b = 0
else b = rA
EA = b + rB
MEM(EA,2) = rS[24-31] rS[16-23]

DESCRIPTION

The sthbrx (store half-word byte-reversed indexed) instruction byte-reverses the low-order 16 bits of rS and stores that value into the half-word in memory addressed by EA. The effective address (EA) is the sum (rA+0) + rB.

On 32-bit implementations, the contents of rS[24-31] are stored into bits 0–7 of the half-word in memory addressed by EA. Bits rS[16-23] are stored into bits 8–15 of the half-word in memory addressed by EA. On 64-bit implementations, the contents of rS[56-63] are stored into bits 0–7 of the half-word in memory addressed by EA. Bits rS[48-55] are stored into bits 8–15 of the half-word in memory addressed by EA.

REGISTERS AFFECTED
None
**INTRODUCTION UNIT**

**601/603/604/620**

**USER MODE**

**FORMS**

sthu rS,d(rA)

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>Ox</th>
<th>2</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

EA ← rA + EXTS(d)
MEM(EA,2) ← rS[16-31]
rA ← EA

**DESCRIPTION**

The sthu (store half-word with update) instruction stores the low-order 16 bits of rS into the half-word in memory addressed by EA. The effective address (EA) is the sum (rA10) + d, where d is a 16-bit signed value that is sign-extended to 32 bits (64 bits on 64-bit implementations). The EA is placed into rA. On 32-bit implementations, rS[16-31] is stored into the half-word in memory addressed by EA. On 64-bit implementations, rS[48-63] is stored into the half-word in memory addressed by EA.

The PowerPC architecture defines the instruction form as invalid if rA = 0, but the 601 supports execution with rA = 0 as shown above. On the 603, 604, and 620, rA = 0 is invalid.

**REGISTERS AFFECTED**

None
**stlux**  
*STORE half-word with EA update using indexed addressing*

**FORMS**

stlux  rS,rA,rB

**BIT DEFINITION**

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 0x1f | S | A | B |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | 0x1b7 | 0 |

**PSEUDO CODE**

EA ← rA + rB  
MEM(EA, 2) ← rS[16-31]  
rA ← EA

**DESCRIPTION**

The stlux (store half-word with update indexed) instruction stores the low-order 16 bits of rS into the half-word in memory addressed by EA. The effective address (EA) is the sum (rA|0) + rB. The EA is placed into rA. On 32-bit implementations, rS[16-31] is stored into the half-word in memory addressed by EA. On 64-bit implementations, rS[48-63] is stored into the half-word in memory addressed by EA.

The PowerPC architecture defines the instruction form as invalid if rA = 0, but the 601 supports execution with rA = 0 as shown above. On the 603, 604, and 620 rA = 0 is invalid.

**REGISTERS AFFECTED**

None
**Integer Unit**

601/603/604/620

**User Mode**

**Forms**

sthx rS,rA,rB

**Bit Definition**

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>0x1f</th>
<th>0x197</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pseudo Code**

if rA = r0 then b ← 0
else b ← rA
EA ← b + rB
MEM(EA,2) ← rS[16-31]

**Description**

The sthx (store half-word indexed) instruction stores the low-order 16 bits of rS into the half-word in memory addressed by EA. The effective address (EA) is the sum (rA[0] + rB. On 32-bit implementations, rS[16-31] is stored into the half-word in memory addressed by EA. On 64-bit implementations, rS[48-63] is stored into the half-word in memory addressed by EA.

**Registers Affected**

None
**stmw**

**STORE MULTIPLE WORDS FROM REGISTERS TO MEMORY**

**FORMS**

```plaintext
stmw rS,d(rA)
```

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>A</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

```plaintext
if rA = r0 then b ← 0
else b ← rA
EA ← b + EXTS(d)
```

```plaintext
r ← rS
do while r ≤ 31
  MEM(EA,4) ← GPR(r)
  r ← r + 1
  EA ← EA + 4
```

**DESCRIPTION**

The **stmw** (store multiple word) instruction stores the contents of multiple GPRs to memory. The effective address (EA) is the sum (rA0) + d, where d is a 16-bit signed value that is sign-extended to 32 bits (64 bits on 64-bit implementations). \( n \) consecutive words starting at EA are stored from the GPRs rS through r31, where \( n = (32 - rS) \). For example, if rS = 30, two words are stored. EA must be a multiple of four; otherwise, the system alignment error handler is invoked or the results are bounded-ly undefined. Note that **stmw** will generate an alignment exception if the address is not WORD (32-bit) aligned.

The PowerPC architecture cautions programmers that some implementations may run this instructions with greater latency (perhaps much greater) than a sequence of individual load/store instructions that produce the same results.

**REGISTERS AFFECTED**

None
**INTEGER UNIT**

*601/603/604/620 User Mode*

**stswi**

*Store string word immediate*

**FORMS**

stswi rS,rA,NB

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>A</th>
<th>NB</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ox1f</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

if rA = r0 then EA ← 0
else EA ← rA
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 (mod 32)
   MEM(EA,1) ← GPR(r)[i-1+7]
   i ← i + 8
   if i = 31 then i ← 0
   EA ← EA + 1
   n ← n - 1

**DESCRIPTION**

The stswi (store string word immediate) instruction stores multiple words to memory. The effective address (EA) is (rA0). Let n = NB if NB ≠ 0, n = 32 if NB = 0; n is the number of bytes to store. Let nr = CEIL(n/4); nr is the number of registers to supply data. n consecutive bytes starting at EA are stored from GPRs rS through rS + nr - 1. Bytes are stored left to right from each register.

The sequence of registers wraps around through r0 if required. Under certain conditions (for example, segment boundary crossings) the data alignment exception handler may be invoked.

The PowerPC architecture cautions programmers that some implementations may run this instructions with greater latency (perhaps much greater) than a sequence of individual load/store instructions that produce the same results.

**REGISTERS AFFECTED**
None
stswx
STORE STRING WORD INDEXED

**FORMS**
stswx rS,rA,rB

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>A</th>
<th>B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
</tr>
<tr>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
</tr>
<tr>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
</tr>
<tr>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
</tr>
<tr>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
</tr>
<tr>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
</tr>
<tr>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

if rA = r0 then b ← 0
else b ← rA
EA ← b + rB
n ← XER[25-31]
r ← rS - 1
i ← 0
do while n > 0
    if i = 0 then r ← r + 1 (mod 32)
    MEM(EA,1) ← GPR(r)[i-i+7]
    i ← i + 8
    if i = 31 then i ← 0
    EA ← EA + 1
    n ← n - 1

**DESCRIPTION**
The stswi (store string word indexed) instruction stores multiple words to memory. The effective address (EA) is the sum (rA[0] + rB. Let n = XER[25-31]; n is the number of bytes to store. Let nr = CEIL(n/4), where nr is the number of registers to supply data. n consecutive bytes starting at EA are stored from GPRs rS through rS + nr - 1.

Bytes are stored left to right from each register. The sequence of registers wraps around through r0 if required. Under certain conditions (for example, segment boundary crossings) the data alignment exception handler may be invoked.

The PowerPC architecture cautions programmers that some implementations may run this instructions with greater latency (perhaps much greater) than a sequence of individual load/store instructions that produce the same results.

**REGISTERS AFFECTED**
None
INTEGER UNIT

601/603/604/620

USER MODE

FORMS

\texttt{stw \ rS, d(rA)}

BIT DEFINITION

<table>
<thead>
<tr>
<th>0x24</th>
<th>S</th>
<th>A</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

PSEUDO CODE

if \( rA = r0 \) then \( b \leftarrow 0 \)
else \( b \leftarrow rA \)
\( EA \leftarrow b + \text{EXTS}(d) \)
\( \text{MEM}(EA, 4) \leftarrow rS \)

DESCRIPTION

The \texttt{stw} (store word) instruction stores the contents of \( rS \) into the word in memory addressed by \( EA \). The effective address (EA) is the sum \( (rA10) + d \), where \( d \) is a 16-bit signed value that is sign-extended to 32 bits (64 bits on 64-bit implementations).

REGISTERS AFFECTED

None

EXAMPLE

templ = 0; \quad \text{// templ and temp2 are globally defined unsigned longs}

\begin{verbatim}
\text{temp2 = templ;}
\text{; Assumes:}
\text{; r4 = contains address of templ}
\text{; r5 = contains address of temp2}
\text{;}
\text{addi \ r3, r0, 0}
\text{stw \ r3, 0(r4)}
\text{or \ r4, r3, r3}
\text{stw \ r4, 0(r5)}
\end{verbatim}
**STWBRX**

**STORE BYTE-REVERSED WORD USING INDEXED ADDRESSING**

**FORMS**

STWBRX rS,rA,rB

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>0x1f</th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>0x296</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

if rA = r0 then b ← 0
else b ← rA
EA ← b + rB

**DESCRIPTION**

The **STWBRX** (store word byte-reversed indexed) instruction byte reverses the contents of rS and stores the result into the word in memory addressed by EA. The effective address (EA) is the sum (rA|0) + rB.

On 32-bit implementations, the contents of rS[24-31] are stored into bits 0-7 of the word in memory addressed by EA. Bits rS[16-23] are stored into bits 8-15 of the word in memory addressed by EA. Bits rS[8-15] are stored into bits 16-23 of the word in memory addressed by EA. Bits rS[0-7] are stored into bits 24-31 of the word in memory addressed by EA.

On 64-bit implementations, the contents of rS[56-63] are stored into bits 0-7 of the word in memory addressed by EA. Bits rS[48-55] are stored into bits 8-15 of the word in memory addressed by EA. Bits rS[40-47] are stored into bits 16-23 of the word in memory addressed by EA. Bits rS[32-39] are stored into bits 24-31 of the word in memory addressed by EA.

**REGISTERS AFFECTED**

None
**stwcx.**

**STORE WORD CONDITIONALLY USING INDEXED ADDRESSING**

**USER MODE**

**INTEGER UNIT**

**601/603/604/620**

**FORMS**

stwcx. rS,rA,rB

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x1f</th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>0x96</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

if rA = r0 then b = 0
else b = rA
EA = b + rB
if RESERVE then
    MEM(EA,4) = rS
    RESERVE ← 0
    CR0 ← 0b00 || 0b1 || XER[SO]
else
    CR0 ← 0b00 || 0b0 || XER[SO]

**DESCRIPTION**

The **stwcx.** (store word conditional indexed) instruction stores the contents of rS into the word in memory addressed by EA. The effective address (EA) is the sum (rA+0) + rB.

The PowerPC architecture defines the **lwarx** and **stwcx.** instructions as a means of atomic memory accesses. This is accomplished by setting a reservation on the load, and checking that the reservation is still valid before the store is performed. On the 603, 604, and 620, the reservations are made on behalf of aligned 32-byte sections of the memory address space. If a reservation exists, the contents of rS are stored into the word in memory addressed by EA and the reservation is cleared. If no reservation exists, the instruction completes without altering memory.

CR0 Field is set to reflect whether the store operation was performed (i.e., whether a reservation existed when the **stwcx.** instruction commenced execution) as follows:

CR0[LT GT EQ SO] = 0b00 || store_performed || XER[SO]

The CR0[EQ] bit in the condition register is modified to reflect whether the store operation was performed (whether a reservation existed when the **stwcx.** instruction began execution). If the store was completed successfully, the CR0[EQ] bit is set to 1.
If the EA is not a multiple of 4, the system alignment error handler may be invoked or the results may be undefined.

In general, the \texttt{stwcx} instruction always broadcasts on the external bus and will therefore operate with slightly worse performance characteristics compared to normal store instructions.

\textbf{REGISTERS AFFECTED}

CR0[LT,GT,EQ,SO]
INTEGER UNIT

601/603/604/620

USER MODE

FORMS

\texttt{stwu rS, d(rA)}

BIT DEFINITION

<table>
<thead>
<tr>
<th>0x25</th>
<th>S</th>
<th>A</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PSEUDO CODE

\begin{align*}
\text{EA} & \leftarrow \text{rA} + \text{EXTS}(d) \\
\text{MEM(EA, 4)} & \leftarrow \text{rS} \\
\text{rA} & \leftarrow \text{EA}
\end{align*}

DESCRIPTION

The \texttt{stwu} (store word with update) instruction stores the contents of \texttt{rS} into the word in memory addressed by \texttt{EA}. The effective address (EA) is the sum \((\text{rA} \cdot 0) + d\), where \(d\) is a 16-bit signed value that is sign-extended to 32 bits (64 bits on 64-bit implementations). The EA is placed into \texttt{rA}.

The PowerPC architecture defines the instruction form as invalid if \(\text{rA} = 0\), but the 601 supports execution with \(\text{rA} = 0\) as shown above. On the 603, 604, and 620 \(\text{rA} = 0\) is invalid.

REGISTERS AFFECTED

None
**STWUX**

**STORE WORD WITH EA**

**UPDATE USING INDEXED ADDRESSING**

**FORMS**

`stwux  rS, rA, rB`

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>0xb7</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

```plaintext
EA ← rA + rB
MEM(EA, 4) ← rS
rA ← EA
```

**DESCRIPTION**

The `stwux` (store word with update indexed) instruction stores the contents of `rS` into the word in memory addressed by `EA`. The effective address (EA) is the sum `(rA|0) + rB`. The EA is placed into `rA`.

The PowerPC architecture defines the instruction form as invalid if `rA = 0`, but the 601 supports execution with `rA = 0` as shown above. On the 603, 604, and 620 `rA = 0` is invalid.

**REGISTERS AFFECTED**

None
**INTEGER UNIT**

**601/603/604/620**

*User Mode*

**stwx**

*Store word using indexed addressing*

**FORMS**

stwx rS, rA, rB

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x1f</th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>0x97</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

if rA = r0 then b ← 0
else b ← rA
EA ← b + rB
MEM(EA, 4) ← rS

**DESCRIPTION**

The stwx (store word indexed) instruction stores the contents of rS into the word in memory addressed by EA. The effective address (EA) is the sum (rA|0) + rB.

**REGISTERS AFFECTED**

None
subfx
SUBTRACT REGISTERS

FORMS

<table>
<thead>
<tr>
<th>FORM</th>
<th>EFFECTS</th>
<th>OE</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>subf</td>
<td>rD, rA, rB</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>subf.</td>
<td>rD, rA, rB</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>subfo</td>
<td>rD, rA, rB</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>subfo.</td>
<td>rD, rA, rB</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

SIMPLIFIED MNEMONICS

sub rD, rA, rB       =       subf rD, rB, rA

BIT DEFINITION

<table>
<thead>
<tr>
<th>D</th>
<th>A</th>
<th>B</th>
<th>OE</th>
<th>0x28</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PSEUDO CODE

rD ← rB - rA

DESCRIPTION

The subf (subtract from) instruction subtracts rA from rB and stores the result into destination register rD. The subf instruction is preferred for subtraction because it sets few status bits.

Note that the setting of the affected bits in the XER is mode-dependent. For 32-bit implementations, the setting of the XER reflects overflow of the lower-order 32-bit result. For 64-bit implementations, the setting of the XER reflects overflow of the 64-bit result. An overflow condition exists if the carry out of the MSb of the result is not equal to the carry out of the MSb + 1.

REGISTERS AFFECTED

- CR0[LT, GT, EQ, SO] (if Rc = 1)
- XER[SO, OV] (if OE = 1)
INTEGER UNIT

601/603/604/620

USER MODE

**FORMS**

<table>
<thead>
<tr>
<th>Form</th>
<th>Description</th>
<th>OE</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>subfc</td>
<td>rD, rA, rB</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>subfc</td>
<td>rD, rA, rB</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>subfco</td>
<td>rD, rA, rB</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>subfco</td>
<td>rD, rA, rB</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**SIMPLIFIED MNEMONICS:**

subc rD, rA, rB = subfc rD, rB, rA

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>OE</th>
<th>0x08</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

rD ← rB - rA

**DESCRIPTION**

The subfc (subtract from carrying) instruction subtracts rA from rB and stores the result into destination register rD. If there is a carry out of the most significant bit (MSb) of the result, XER[CA] is set; otherwise, XER[CA] is cleared.

The setting of the affected bits in the XER is mode-dependent. For 32-bit implementations, the setting of the XER reflects overflow of the lower-order 32-bit result. For 64-bit implementations, the setting of the XER reflects overflow of the 64-bit result. An overflow condition exists if the carry out of the MSb of the result is not equal to the carry out of the MSb + 1.

**REGISTERS AFFECTED**

- CR0[LT, GT, EQ, SO] (if Rc = 1)
- XER[CA]
- XER[SO, OV] (if OE = 1)
**subfex**

**SUBTRACT REGISTERS**

**AND ADD CARRY BIT**

### FORMS

<table>
<thead>
<tr>
<th>subfex</th>
<th>rD, rA, rB</th>
<th>OE</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>subfe</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>subfe.</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>subfeo</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>subfeo.</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### BIT DEFINITION

<table>
<thead>
<tr>
<th>0xf</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>OE</th>
<th>0x88</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### PSEUDO CODE

\[ rD \leftarrow rB + (\text{NOT}(rA)) + \text{XER}[\text{CA}] \]

### DESCRIPTION

The subfex (subtract from extended) instruction sums \( rB \), the complement of \( rA \), and the carry bit and stores the result into destination register \( rD \). It is assumed that \( \text{XER}[\text{CA}] \) is set explicitly by a previous operation, such as the subfc (subtract from carrying) instruction.

The setting of the affected bits in the XER is mode-dependent. For 32-bit implementations, the setting of the XER reflects overflow of the lower-order 32-bit result. For 64-bit implementations, the setting of the XER reflects overflow of the 64-bit result. An overflow condition exists if the carry out of the MSb of the result is not equal to the carry out of the MSb + 1.

### REGISTERS AFFECTED

- CR0[LT, GT, EQ, SO] (if RC = 1)
- XER[CA]
- XER[SO, OV] (if OE = 1)
**INTEGER UNIT**

**601/603/604/620**

**User Mode**

**subfic**

SUBTRACT IMMEDIATE FROM
REGISTER AND SET CARRY BIT

**FORMS**

| subfic | rD, rA, SIMM |

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x08</th>
<th>D</th>
<th>A</th>
<th>SIMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

rD ← EXTS(SIMM) - rA

**DESCRIPTION**

The subfic (subtract from immediate carrying) instruction subtracts rA from a 16-bit signed value and stores the result into destination register rD. If there is a carry out of the most significant bit (MSb) of the result, XER[CA] is set; otherwise, XER[CA] is cleared.

The setting of the affected bits in the XER is mode-dependent. For 32-bit implementations, the setting of the XER reflects overflow of the lower-order 32-bit result. For 64-bit implementations, the setting of the XER reflects overflow of the 64-bit result. An overflow condition exists if the carry out of the MSb of the result is not equal to the carry out of the MSb + 1.

**REGISTERS AFFECTED**

XER[CA]
**subfmex**

**SUBTRACT REGISTER FROM MINUS ONE EXTENDED**

### FORMS

<table>
<thead>
<tr>
<th>Instruction</th>
<th>rD, rA, rB</th>
<th>OE</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>subfme</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>subfme.</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>subfmeo</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>subfmeo.</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### BIT DEFINITION

<table>
<thead>
<tr>
<th></th>
<th>Reserved</th>
<th>D</th>
<th>A</th>
<th>00000</th>
<th>OE</th>
<th>0xe8</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

### PSEUDO CODE

\[ rD \leftarrow \text{NOT}(rA) + \text{XER}[\text{CA}] - 1 \]

### DESCRIPTION

The subfme (subtract from minus one extended) instruction subtracts one from the complement of \(rA\), adds the carry bit, and stores the result into destination register \(rD\).

The setting of the affected bits in the XER is mode-dependent. For 32-bit implementations, the setting of the XER reflects overflow of the lower-order 32-bit result. For 64-bit implementations, the setting of the XER reflects overflow of the 64-bit result. An overflow condition exists if the carry out of the MSb of the result is not equal to the carry out of the MSb + 1.

### REGISTERS AFFECTED

- CR0[LT,GT,EQ,SO] (if \(Rc = 1\))
- XER[CA]
- XER[SO,OV] (if \(OE = 1\))
**subfzex**

**SUBTRACT REGISTER FROM ZERO EXTENDED**

### Forms

<table>
<thead>
<tr>
<th>subfze</th>
<th>rD, rA</th>
<th>OE</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>subfze.</td>
<td>rD, rA</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>subfzeo</td>
<td>rD, rA</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>subfzeo.</td>
<td>rD, rA</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Bit Definition

<table>
<thead>
<tr>
<th>0x1f</th>
<th>D</th>
<th>A</th>
<th>00000</th>
<th>OE</th>
<th>0xc8</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Pseudo Code

\[ rD \leftarrow \text{NOT}(rA) + \text{XER}[\text{CA}] \]

### Description

The subfze (subtract from zero extended) instruction adds the complement of \( rA \) to the carry bit and stores the result into destination register \( rD \).

The setting of the affected bits in the XER is mode-dependent. For 32-bit implementations, the setting of the XER reflects overflow of the lower-order 32-bit result. For 64-bit implementations, the setting of the XER reflects overflow of the 64-bit result. An overflow condition exists if the carry out of the MSb of the result is *not* equal to the carry out of the MSb + 1.

### Registers Affected

- CR0[LT, GT, EQ, SO] (if Rc = 1)
- XER[CA]
- XER[SO, OV] (if OE = 1)
The `sync` (synchronize) instruction provides an ordering function for the effects of all instructions executed by a given PowerPC microprocessor. Executing a `sync` instruction ensures that all instructions previously initiated by the given processor (except touch load and instruction fetches) have completed, at least to the point where they can no longer cause an exception, before any subsequent instructions are initiated by the given processor.

When the `sync` instruction completes, all external accesses initiated by the given processor prior to the `sync` will have been performed with respect to all other mechanisms that access memory. Additionally, all load and store cache/bus activities initiated by prior instructions are completed. Operations such as `dcbt` and `dcbst` are required to complete at least through address translation, but are not required to complete on the bus.

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

The `eieio` instruction may be more appropriate than `sync` for cases in which the only requirement is to control the order in which external references are seen by I/O devices. Since the 603 enforces that all loads and stores execute in order on the external bus, the `eieio` instruction is treated as a no-op on the 603.

**Registers Affected**

- CR0[LT,GT,EQ,SO] (if Rc = 1)
- XER[CA]
- XER[SO,OV] (if OE = 1)
**INTEGER UNIT**

**620 USER MODE**

**FORMS**

\[ \text{td TO,} rA, rB \]

**Simplified Mnemonics**

\[ \text{tdge} rA, rB = \text{td} 12, rA, rB \]
\[ \text{tdlnl} rA, rB = \text{td} 5, rA, rB \]

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x1f</th>
<th>TO</th>
<th>A</th>
<th>B</th>
<th>0x44</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pseudo Code**

if (rA < rB) & TO[0] then TRAP
if (rA > rB) & TO[1] then TRAP
if (rA = rB) & TO[2] then TRAP
if (rA <U rB) & TO[3] then TRAP
if (rA >U rB) & TO[4] then TRAP

**Description**

The **td** (trap doubleword) instruction compares the contents of rA and rB. If any bit in the TO field is set and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

**Registers Affected**

None
tdi
TRAP DOUBLEWORD IMMEDIATE

**FORMS**
tdi TO,rA,SIMM

**SIMPLIFIED MNEMONICS**
tdlriti rA,value  =  tdi 16,rA,value
tdnei rA,value  =  tdi 24,rA,value

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>0x02</th>
<th>TO</th>
<th>A</th>
<th>SIMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

if (rA < \text{EXTS(SIMM)}) \& \& \text{TO}[0] then TRAP
if (rA > \text{EXTS(SIMM)}) \& \& \text{TO}[1] then TRAP
if (rA = \text{EXTS(SIMM)}) \& \& \text{TO}[2] then TRAP
if (rA < \text{EXTS(SIMM)}) \& \& \text{TO}[3] then TRAP
if (rA > \text{EXTS(SIMM)}) \& \& \text{TO}[4] then TRAP

**DESCRIPTION**
The tdi (trap doubleword immediate) instruction compares the contents of rA with the 16-bit sign-extended value of the SIMM field. If any bit in the TO field is set and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

**REGISTERS AFFECTED**
None
**INTEGER UNIT**

**603/604/620**

**SUPERVISOR MODE**

**FORMS**

`tlbia`

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ox1f</td>
<td>00000</td>
</tr>
<tr>
<td>Ox0</td>
<td>00000</td>
</tr>
<tr>
<td>Ox0</td>
<td>00000</td>
</tr>
<tr>
<td>Ox172</td>
<td></td>
</tr>
</tbody>
</table>

**DESCRIPTION**

The `tlbia` (TLB invalidate all) instruction is optional in the PowerPC architecture. The entire translation lookaside buffer (TLB) is invalidated. That is, all entries are removed. The TLB is invalidated regardless of the settings of MSR[IR] and MSR[DR] (the address translation enable bits). Invalidation is done without reference to the SLB and segment table (on 64-bit implementations), or segment registers (on 32-bit implementations).

**REGISTERS AFFECTED**

None
The `tlbie` (TLB invalidate entry) instruction is optional in the PowerPC architecture. The effective address (EA) is the contents of rB. The translation lookaside buffer (referred to as the UTLB on the 601 and the ITLB/DTLB on the 603, 604, and 620) containing entries corresponding to the EA are made invalid (i.e., removed from the TLB). Block address translation for EA, if any, is ignored.

This instruction is optional in the PowerPC architecture. Because of the differences between processor implementations, the behavior of `tlbie` on each is examined independently below:

### 601 Processor

On the 601, a TLB invalidate operation is broadcast on the system interface. The UTLB search is done regardless of the settings of MSR[IT] and MSR[DT].

If the segment register for EA specifies SR[T] = 1 (an I/O controller interface segment), no UTLB entry invalidation is performed on the local processor and no TLB invalidate operation is broadcast on the system interface.

Because the 601 supports broadcast of TLB entry invalidate operations, then the following must be observed:

The `tlbie` instruction(s) must be contained in a critical section, controlled by software locking, so that `tlbie` is issued on only one processor at a time.

A `sync` instruction must be issued after every `tlbie` and at the end of the critical section. This causes the hardware to wait for the effects of the preceding `tlbie` instruction(s) to propagate to all processors.
A processor detecting a TLB invalidate broadcast performs the following:

1. Prevents execution of any new load, store, cache control or \texttt{tlbie} instruction and prevents any new reference or change bit updates.

2. Waits for completion of any outstanding memory operations (including updates to the reference and change bits associated with the entry to be invalidated).

3. Invalidates the two entries (both associativity classes) in the UTLB indexed by the matching address.

4. Resumes normal execution.

The software must ensure that SDR1 points to the page table when issuing \texttt{tlbie}, even when address translation is disabled. Nothing is guaranteed about instruction fetching in other processors if the \texttt{tlbie} instruction deletes the page in which some other processor is currently executing.

\textbf{603/604/620 Processors}

The effective address (EA) is the contents of rB. Both the instruction and data translation lookaside buffer (referred to as ITLB and DTLB, respectively) containing entries corresponding to the EA are made invalid both ways (that is, removed from both sets of the TLB) at the index provided within the EA. The TLB invalidation is performed regardless of settings of MSR[IR] and MSR[DR]. The index corresponds to bits 15–19 of the EA. To invalidate entries within both TLBs, 32 \texttt{tlbie} instructions must be executed, incrementing this field by one each time.

Nothing is guaranteed about instruction fetching in other processors if the \texttt{tlbie} instruction deletes the pages in which some other processor is currently executing.

\textbf{Registers Affected}

None
**tllbd**

**LOAD DATA TLB ENTRY**

**FORMS**

`tllbd` rB

**BIT DEFINITION**

<table>
<thead>
<tr>
<th></th>
<th>Reserved</th>
<th>Reserved</th>
<th>B</th>
<th>0x3d2</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3 4</td>
<td>5</td>
<td>0x1f</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>0000</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>0000</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>B</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

EA ← rB
TLB entry created from DCMP and RPA
DTLB entry selected by EA[15-19] and SRR1[WAY] ← created TLB entry

**DESCRIPTION**

The `tllbd` (load data TLB entry) instruction is optional in the PowerPC architecture. The `tllbd` instruction loads the contents of the data page table entry compare (DCMP) and required physical address (RPA) registers into the first word of the selected data TLB entry. The specific DTLB entry to be loaded is selected by EA and the SRR1[WAY] bit. The effective address (EA) is the contents of rB. The `tllbd` instruction should only be executed when address translation is disabled: MSR[IR] = 0 and MSR[DR] = 0.

**REGISTERS AFFECTED**

None
### INTEGER UNIT

**603**

**SUPERVISOR MODE**

---

**TLB ENTRY**

**tlbli**

**LOAD INSTRUCTION**

**Description**

The **tlbli** (load instruction TLB entry) instruction is optional in the PowerPC architecture. The **tlbli** instruction loads the contents of the data page table entry compare (DCMP) and required physical address (RPA) registers into the first word of the selected data TLB entry. The specific ITLB entry to be loaded is selected by EA and the SRR1[WAY] bit. The effective address (EA) is the contents of rB. The **tlbli** instruction should only be executed when address translation is disabled: MSR[IR] = 0 and MSR[DR] = 0.

**Registers Affected**

None
**tlbsync**

**SYNCHRONIZE TLB**

**FORMS**

tlbld rB

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Reserved</th>
<th>Reserved</th>
<th>Reserved</th>
<th>0x236</th>
<th>Res.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1f</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>0</td>
</tr>
</tbody>
</table>

**DESCRIPTION**

As defined by the PowerPC architecture, the `tlbsync` (synchronize translation lookaside buffer) instruction does not complete until all previous `tlbi` and `tlbli` instructions executed by the processor executing this `tlbsync` instruction have been received and completed.

On the 603/604/620, when the TLBISYNC signal is negated, instruction execution may continue or resume after the completion of a `tlbsync` instruction. When the TLBISYNC signal is asserted, instruction execution stops after the completion of a `tlbsync` instruction.

**REGISTERS AFFECTED**

None
**INTEGER UNIT**

**601/603/604/620**

**User Mode**

**FORMS**
- \( \text{tw} \text{ TO},rA,rB \)

**Simplified Mnemonics**
- \( t\text{weq } rA,rB \equiv \text{tw} 4,rA,rB \)
- \( tw\text{lge } rA,rB \equiv \text{tw} 5,rA,rB \)
- \( \text{trap} \equiv \text{tw} 31,r0,r0 \)

**Bit Definition**

<table>
<thead>
<tr>
<th>0x1f</th>
<th>TO</th>
<th>A</th>
<th>B</th>
<th>0x04</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pseudo Code**

\[
\begin{align*}
\text{a} & \leftarrow \text{EXTS}(rA) \\
\text{b} & \leftarrow \text{EXTS}(rB) \\
\text{if (a < b) \& TO[0] then TRAP} \\
\text{if (a > b) \& TO[1] then TRAP} \\
\text{if (a = b) \& TO[2] then TRAP} \\
\text{if (a < U b) \& TO[3] then TRAP} \\
\text{if (a > U b) \& TO[4] then TRAP}
\end{align*}
\]

**Description**
The \( \text{tw} \) (trap word) instruction invokes the system trap handler. The contents of \( rA \) are compared with the contents of \( rB \). If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

**Registers Affected**
None
**twi**
**Trap Word Immediate**

**Forms**
```
twi TO, rA, SIMM
```

**Bit Definition**

**Pseudo Code**
```
a ← EXTS(rA)
if (a < EXTS(SIMM)) & TO[0] then TRAP
```

<table>
<thead>
<tr>
<th>SIMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x03</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31</td>
</tr>
</tbody>
</table>

```
if (a > EXTS(SIMM)) & TO[1] then TRAP
if (a = EXTS(SIMM)) & TO[2] then TRAP
if (a <U EXTS(SIMM)) & TO[3] then TRAP
if (a >U EXTS(SIMM)) & TO[4] then TRAP
```

**Description**
The `twi` (trap word immediate) instruction invokes the system trap handler. The contents of `rA` are compared with the 16-bit sign-extended `SIMM` field. If any bit in the `TO` field is set to 1 and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

**Registers Affected**
None
**INTEGER UNIT**

601/603/604/620

**User Mode**

**FORMS**

<table>
<thead>
<tr>
<th>xor</th>
<th>rA, rS, rB</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>xor.</td>
<td>rA, rS, rB</td>
<td>1</td>
</tr>
</tbody>
</table>

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>Ox1f</th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>Ox13c</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
</tr>
</tbody>
</table>

**Pseudo Code**

RA ← RS XOR RB

**DESCRIPTION**

The xor instruction calculates the bit-wise exclusive-OR of RS with RB and places the result into destination register RA.

**REGISTERS AFFECTED**

CR0[LT, GT, EQ, SO] (if Rc = 1)

**EXAMPLE**

```c
unsigned char b1, b2, b3;        // 3 globally declared bytes
b1 = 0x5a;
b2 = 0x20;
b3 = b1 ^ b2;

; Assumes:
; r3 = contains address of b1
; r4 = contains address of b2
; r5 = contains address of b3
;
li r6, 0x5a                   ; load r6 with immediate value 0x5a
stb r6, 0(r3)                 ; store byte in r6 at r3 (b1=0x5a)
li r6, 0x20                   ; load r6 with immediate value 0x20
stb r6, 0(r4)                 ; store byte in r6 at r4 (b2=0x20)
lbz r3, 0(r4)                 ; get 0x5a value back from r3
xor r3, r3, r6               ; xor operation, result in r3
stb r3, 0(r5)                 ; store byte result into b3 (at r5)
```
**xori**

**XOR REGISTER WITH IMMEDIATE**

**FORMS**

xori rA,rS,UIMM

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0xa</th>
<th>S</th>
<th>A</th>
<th>UIMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

rA ← rS XOR (0x0000 || UIMM)

**DESCRIPTION**

The `xori` (xor immediate) instruction calculates the bit-wise exclusive-OR of `rS` with a 16-bit unsigned value and places the result into `rA`. The 16-bit UIMM field is zero-extended to 32 bits prior to the operation.

**REGISTERS AFFECTED**

None

**EXAMPLE**

```plaintext
b1 ^= 0x5a;       // b1 is a globally defined unsigned char

; Assumes:
; r4 = contains address of b1
;
lblz r3, 0(r4)    ; get byte from (r4)
xori r3, r3, 0x5a ; xor operation
stb r3, 0(r4)    ; store it back
```
**INTEGER UNIT**

**601/603/604/620**

**User Mode**

**xoris**

XOR REGISTER WITH SHIFTED IMMEDIATE

**FORMS**

xoris  rA,rS,UIMM

**BIT DEFINITION**

<table>
<thead>
<tr>
<th>0x1b</th>
<th>S</th>
<th>A</th>
<th>UIMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

**PSEUDO CODE**

rA ← rS XOR (UIMM << 16)

**DESCRIPTION**

The xoris (XOR immediate shifted) instruction calculates the bit-wise exclusive-OR of rS with a 16-bit unsigned value (shifted left by 16) and places the result into rA.

**REGISTERS AFFECTED**

None
This appendix is dedicated to correlating the x86 instruction set with the PowerPC instruction set. In some cases, there are few overlapping instructions. However, other instructions have close counterparts in each instruction set.

In the first section of this appendix, we'll examine the issues relating to the translation of x86 assembly language to PowerPC assembly language. Following the translation discussion, there is a table that relates every x86 instruction (through the i486) to its closest corresponding PowerPC instruction.

**Assembly Language Translation**

The computer industry is currently dominated by x86-based computers. So it makes sense that the PowerPC industry is interested in taking advantage of the existing software base. The translation of x86 code to PowerPC code represents one means of capitalizing on the existing software base.

In practice, there are a number of specific cases where translation works extremely well. For example, on-the-fly translation of x86 BIOS ROMs is useful when initializing legacy peripheral boards, such as ISA and PCI video adapters.

Additionally, the translation of x86 assembly language code to PowerPC assembly language code — suitable for native reassembly — represents a significant tool for porting existing software to PowerPC platforms. This form of translation is the topic of the following discussion.
The author wishes to thank MicroAPL for their help with this topic. For further information concerning MicroAPL and the PortAsm program (an x86-to-PowerPC assembly language translator), check out the version included on the CD-ROM with this book, or contact them at the address found in the Bibliography.

**Translating x86 Assembly Language**

There are four ways in which assembly language applications can be ported to different and incompatible processor types:

- Rewrite the code from scratch.
- Run the application under an emulator, as is done in Insignia Solutions SoftPC product.
- Translate the source code from one assembler language to another.
- Binary-to-binary translation, where access to the original source code is not available or required. A comprehensive disassembly of the program must be done first.

Rewriting source code is always an available option — but uninteresting for the specific purposes of this discussion.

Emulation requires no changes to the application itself, but gives poor performance compared with a native application. Each original machine instruction has to be read, decoded, and emulated on the target architecture. The emulator is effectively a language interpreter, where the language is the instruction set of the CPU for which the program was originally written. Emulators are useful to get old software to run on a new machine, or to run applications where high performance is not important. But they do not provide a means of allowing old programs to take full advantage of new high-performance processors such as the PowerPC.

Translation of assembly language, in contrast, produces a true native application. A native application can run very nearly as fast as applications written specifically for the target processor, and at worst will still be considerably faster than an emulator. The speed advantage over an emulator arises from a number of factors. First, there is no need to read and decode the instruction; even in a well-written emulator, this adds an overhead of several instructions for every source instruction executed.

Another emulator problem is that the emulator generally has no knowledge of what instruction is about to be executed, and therefore must main-
tain a completely valid machine state after every instruction is interpreted. This will have to include all side effects of the instruction including setting or clearing flags. A translator, on the other hand, like a well-written compiler, can see when the side effects of an instruction are not necessary for the program to execute correctly, thus avoiding unnecessary work.

In addition, the translator can apply optimizations on sequences of instructions, including instruction scheduling for RISC processors. Finally, source-level translation makes it very easy to include handwritten assembly language optimized for the target processor, which can improve performance considerably.

For reasons of maintainability and debugging, source-level translation which takes assembly language text as input and produces assembly language source as output is preferable to binary-binary translation. For these reasons, binary-binary translation will not be considered further.

How Translation Works

This section uses MicroAPL's PortAsm as the archetype for x86-to-PowerPC translation. The basic principle of translation is straightforward. If we ignore for the moment the various optimizations that a translator can apply, then its operation consists of reading an assembler source file, working out what each instruction does, and emitting an equivalent instruction or sequence of instructions in the target assembly language.

The primary difficulty with this is the determination of an efficient equivalent. In practice, if the generated code is to yield satisfactory performance, the instruction being translated has to be considered in context. Consider, for example, the following x86 instruction:

```
add ax,2
```

This instruction adds 2 to the register AX. If this instruction is translated to PowerPC assembly, then we need some way of representing the contents of AX, and the most obvious way is to use a PowerPC register. In fact, if we define RAX to be a PowerPC general-purpose register, we might simply translate the above instruction to:

```
addi RAX,RAX,2 ; Add 2 to register RAX, result to RAX
```
This is a very efficient translation, in that one x86 instruction becomes a single PowerPC instruction, which will execute very fast. However, the translation might not be correct as it stands. This is because the x86 addition does more than just add 2 to the contents of AX — it also sets the overflow, parity, carry, zero and other flags. If we know nothing about the context of the instruction, we have to ensure that all these side effects are also mirrored in the target environment. This will typically take several instructions. But if we see the sequence:

```
add    ax, 2
add    bx, 2
```

then we know for certain that the translation of the first add instruction does not need to set any flags, because whatever flags it sets will immediately be overwritten by the add instruction. Therefore, it is necessary for the translator to carry out a comprehensive analysis of the program flow so that it can achieve an efficient translation. This program flow analysis can become very complex, since in principle all potential paths through the program have to be considered, including subroutine calls, conditional branches, and indirect jumps.

Furthermore, the statement that the addition considered above adds 2 to the contents of AX is itself a simplification. On the 80386 or 80486, the register AX is a 16-bit subset of the full 32-bit EAX register, and, in turn, registers AH and AL are 8-bit subsets of AX. Therefore, simply representing AX by a PowerPC 32-bit register is not good enough. In fact, it looks more sensible to represent EAX by a PowerPC register, and consider AX to be just the low-order 16 bits of that register, and AH and AL to be 8-bit subsets in turn.

In this case we have to consider whether the effect of the translated instruction on the high bits of EAX is valid. If the register AX contains the value 0xffff immediately before the addition, then the simple one-instruction translation will not be valid, because it will alter the high 16 bits of EAX which is not what the x86 instruction does. To be sure of reproducing this behavior, the translator can use a spare PowerPC register (here denoted RTemp) to do the addition, and mask the low 16 bits of the result into the AX equivalent register:

```
addi   RTemp, REAX, 2          ; Add 2 to REAX, result to RTemp
rlwimi REAX, RTemp, 0, 0xffff  ; Insert 16-bit result into REAX
```
We now need two instructions rather than one, and some other cases may be even worse. Therefore, to avoid inefficient translation, PortAsm keeps track of when it is necessary to preserve the other bits of a register, and when it is safe to overwrite them. In fact, when it is run with optimizations on, it dynamically allocates registers over blocks of code to provide efficient translation of x86 assembly language.

These simple examples — plus other considerations which are mentioned in later sections — illustrate two important points. First, there is no single best translation of an instruction, but instead there are many possible translations, some of which are considerably more efficient than others. Second, in order to determine whether an efficient translation is also a safe one, the crucial test is whether the translator is able to follow the program flow unambiguously. The better the analysis of the program flow, the more efficient the generated code, and the safer the translation. In addition, optimizations which operate over more than one instruction are also important, as you’ll soon see.

Hints and Assumptions

One big advantage of source-code translation over other forms of automatic porting is that the translation does not have to be perfect in order to be useful. Although it is possible to model the x86 processor and translate every instruction in a way that reproduces all its side effects, such a translation would be very inefficient. This would defeat the object of the exercise, which is to produce a translated equivalent of the original program that runs as efficiently as possible on the target processor.

Translators must, therefore, deal with the details of translation by a combination of analysis, assumption, and occasional human intervention. The kinds of problems that can arise include:

- Passing condition codes into or out of routines which are called through jump tables
- Relying on side effects of external routines about which the translator does not know
- Self-modifying code
- Relying on the size of specific instruction opcodes
There are various techniques that can be used to ensure a valid translation to the target architecture. In almost all cases, changing the original source code is a simple and effective solution.

**x86 Translation Model**

Unfortunately, the x86 processor requires a complex set of translation rules. One of the primary difficulties is the x86's real-mode segmented addressing. An effective address in the x86 world has to be calculated taking into account one of the segment registers CS, DS, ES, FS, GS, or SS (FG and GS existing on 80386 and higher processors only). These segments explicitly or implicitly affect every memory reference.

In order to achieve efficient effective address calculation in translated code, the translated program holds the base address of each current segment in a PowerPC general-purpose register. This allows translation of memory accesses taking account of the implicit or explicit segment base:

```
mov  eax,[ebx]
lwzx reax,rds,rebx ; Load a 32-bit value from the address
          ; formed by adding the contents of rds
          ; and rebx
```

Only when segment registers are changed or accessed directly do we need to concern ourselves with mapping 16-bit segment addresses (in real mode) or segment selectors (in protected mode) from the x86's segmented world to the PowerPC processor's flat address space. Following is an example of such an instruction:

```
mov  ax,ds ;Move ds register to ax
```

This is handled by keeping a memory-based table of segments. Special considerations apply to the stack and instruction pointer. On the x86, segmentation applies to these in the same way as it does to other addresses. The next instruction to execute is pointed to by the instruction pointer plus the code segment, and the current stack position is the stack pointer plus the stack segment. For the instruction pointer (EIP or IP) it is essential, and for the stack pointer (SP or ESP) it is desirable, to factor out the segment address when translating to the PowerPC. This means that the PowerPC program counter corresponds to EIP plus the code segment address, and a PowerPC general-purpose register holds ESP plus the stack segment address.
Where it becomes necessary to determine the actual x86 ESP value, this is done by subtracting the current stack segment address.

The Little Endian Problem

The x86 architecture gives us one severe problem: the way it handles the ordering of bytes in memory. On some CPUs — notably IBM mainframes, the 680x0 family, and SPARC microprocessors — multibyte quantities are stored in memory with the most significant byte at the lowest address. Such processors are known as big endian. On others — including the DEC VAX and Intel processors — the most significant byte is placed at the highest address; these are little endian. Endianess is discussed in Chapter 3, "Of Eggs and Endians."

Applications vary in the degree to which they access data in an endian-dependent way; some applications will be very clean in this respect, whereas others will freely access subsets of larger data items and assume a specific byte ordering. Of course, this is a problem only if you are going from a little endian to big endian architecture or vice versa. It does not arise on porting 680x0 code to PowerPC.

Support for Little Endian Operation in the PowerPC

Although the PowerPC architecture is big endian, it includes two features that help in handling little endian data. Four instructions are provided for reading and writing 32- and 16-bit quantities with automatic byte reversal. These appear mainly to be intended for reading and writing data that is shared (perhaps over a network) with a processor that has the opposite byte ordering. These instructions are of the indexed displacement form; that is, an effective address is formed by adding two registers together. This is convenient for translating x86 code because you always have to offset and address by a segment base.

Also, the whole processor can be switched to run in little endian mode. At first sight, this seems to be an ideal solution to the problem; however, in practice it does little to make the job of translating x86 code easier. This is because, in little endian mode, 16- and 32-bit memory accesses have to be aligned on 16- or 32-bit boundaries, or else a machine exception occurs. The
x86 family does not require aligned memory accesses. In practice, x86 code commonly ignores alignment considerations.

Interesting Translation Situations

In practice, the choice of whether to run the PowerPC processor in little endian or big endian mode is one which will be dictated by the operating system. IBM’s AIX and the Apple Macintosh use big endian mode. It’s natural to expect that ports of operating systems with an x86 ancestry such as OS/2 and Microsoft’s Windows NT would run in little endian mode. To take account of these differences, PortAsm is designed to support three ways of handling the problem when translating Intel code.

The first option is full big endian mode. In this model, the PowerPC CPU runs in big endian mode, and the data is also held in big endian order. Data does not necessarily need to be aligned on natural boundaries, but the original source file has to be checked to ensure that any assumptions about byte ordering are signaled to PortAsm so that it can generate the correct code. PortAsm can automatically detect straightforward cases where data is being accessed in an endian-dependent way, but some manual hinting is required for more difficult cases.

This mode is most appropriate where the target machine architecture is essentially big endian, and the assembler code needs to fit into that architecture smoothly. It would apply, for example, to a section of handwritten assembler code which formed part of a bigger application written in C which was being ported to AIX.

In the next case, the processor operates in big endian mode but the data is in little endian order. For some programs, it will be more appropriate to hold the application’s data in little endian order even if the CPU itself is running in big endian mode. This would be sensible if, for example, the translated program was sharing data with a DOS program running under an emulator (such as SoftPC). In this mode, the translator uses the PowerPC byte-reversal instructions for every 32- or 16-bit memory access (except for stack operations). Minimal manual intervention is required.

Full little endian mode is best when PowerPC processors are operating in little endian mode (and data is stored in natural little endian order) and the byte-ordering problem described above does not exist. However, a different constraint appears: all accesses must be aligned on their natural boundaries. As a result, the translator tries to determine if a particular memory access is aligned. Where alignment can be detected unambiguously — for
example, if it referred directly to a label that had been placed on a 4-byte boundary — the translator generates an efficient direct-access as usual. For 16- or 32-bit accesses with ambiguous alignment, the translator generates less efficient code for accesses. Note that C compilers and other high-level languages have the same problem when they are passed arbitrary pointers to data which is then accessed using an arbitrary unit size. To maintain the quality of translated code, the programmer can mark sections of code with a hint to indicate that a section requires special treatment.

**Translation Optimizations**

So far, we have concentrated on how to translate code by looking at each instruction individually. Although we've emphasized the need for detailed program analysis to produce an efficient translation, this has been in the context of translating instructions one at a time. However, there are many cases where considerable improvements in the quality of the generated code — both in terms of speed and code expansion — can be achieved by considering larger blocks of code.

The first area of optimization works by identifying idioms, or common sequences of instructions, which can better be translated as a block than individually. As an example, consider an x86 sequence that pops a series of values off the stack into registers. Translating such a sequence of POP instructions one at a time is relatively inefficient on a PowerPC, because the stack pointer has to be adjusted each time. A translator might therefore wait until all the POPs have been done (adjusting the stack offset as necessary), and then adjust the stack pointer at the end.

Similarly, it is quite common to zero out a register, and then load an 8- or 16-bit quantity into it. This takes two instructions on a 680x0 or x86, but can be done in one instruction on the PowerPC.

Factoring out common temporary values can be another effective optimization. Where blocks of code use complex addressing modes or preserve high bits across a set of instructions, considerable translation improvements can be achieved. For example, this code fragment is extracted from *Programming the 80386*, by Crawford and Gelsinger, as part of an implementation of the bubble sort algorithm:

```
InnerLoop:
    cmp edx, ecx
    jge Bottom
    mov eax, [esi+edx*4+4]
```
A translator would be able to recognize that the effective address DS:[ESI+EDX*4] is repeatedly being used in this loop, and can therefore factor out part of the address calculation using a temporary register to hold it. Similarly, it can handle the preservation of the other bits of a register over a block of code.

**Conclusion**

Intel x86 assembly translation to PowerPC assembly is one solution for the issues associated with porting software to the PowerPC family of processors. While translation is not a complete solution for every porting situation, this discussion does provide an additional perspective on the relationship between x86 assembly language and PowerPC assembly language.

To experiment with an implementation of the preceding discussion, check out the PortAsm demo included on the CD-ROM that accompanies this book.
**INTEGER INSTRUCTIONS CROSS-REFERENCE**

It's common to think in your native language even when trying to program in a new one. This instruction cross-reference table is designed to help you in those times when you know the x86 instruction you want to use, but can’t think of the corresponding PowerPC instruction. Table B-1 lists the integer instructions that are somewhat analogous on both platforms.

**Table B-1**

<table>
<thead>
<tr>
<th>x86 Instruction</th>
<th>Related PowerPC Mnemonic and Operands</th>
<th>PowerPC Instruction Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>aad</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>aam</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>aas</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>addc (add two registers)</td>
<td>addcx rD,rA,rB</td>
<td>add carrying</td>
</tr>
<tr>
<td>addc (add register and immediate value)</td>
<td>addicx rD,rA,IMM</td>
<td>add immediate carrying</td>
</tr>
<tr>
<td>add (add two registers)</td>
<td>addx rD,rA,rB</td>
<td>add</td>
</tr>
<tr>
<td>add (add register and immediate value)</td>
<td>addi rD,rA,IMM</td>
<td>add immediate</td>
</tr>
<tr>
<td></td>
<td>addis rD,rA,IMM</td>
<td>add immediate shifted</td>
</tr>
<tr>
<td></td>
<td>addex rD,rA,rB</td>
<td>add extended</td>
</tr>
<tr>
<td></td>
<td>addmex rD,rA</td>
<td>add to minus one extended</td>
</tr>
<tr>
<td></td>
<td>addzex rD,rA</td>
<td>add to zero extended</td>
</tr>
<tr>
<td>and (and two registers)</td>
<td>andx rA,rS,rB</td>
<td>AND</td>
</tr>
<tr>
<td>and (and register and immediate value)</td>
<td>andi. rA,rS,UIIMM</td>
<td>AND immediate</td>
</tr>
<tr>
<td></td>
<td>andcx rA,rS,rB</td>
<td>AND with complement</td>
</tr>
<tr>
<td></td>
<td>andis. rA,rS,UIIMM</td>
<td>AND immediate shifted</td>
</tr>
<tr>
<td>arpl</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>x86 Instruction</td>
<td>Related PowerPC InstructionName and Operands</td>
<td>PowerPC InstructionName</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>bound</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>bsf,bsr</td>
<td>cntlzd x rA,rS</td>
<td>count leading zeros doubleword</td>
</tr>
<tr>
<td></td>
<td>cntlzw x rA,rS</td>
<td>count leading zeros word</td>
</tr>
<tr>
<td>bswap</td>
<td>lhbr x rD,rA,rB</td>
<td>load half-word byte-reverse indexed</td>
</tr>
<tr>
<td></td>
<td>lwbr x rD,rA,rB</td>
<td>load word byte-reverse indexed</td>
</tr>
<tr>
<td></td>
<td>sthbr x rS,rA,rB</td>
<td>store half-word byte-reverse indexed</td>
</tr>
<tr>
<td></td>
<td>stwbr x rS,rA,rB</td>
<td>store word byte-reverse indexed</td>
</tr>
<tr>
<td>bt,btc,btr,bts</td>
<td>See rlwinm, rlwimi</td>
<td></td>
</tr>
<tr>
<td>call</td>
<td>Equivalent to branching with link register update</td>
<td></td>
</tr>
<tr>
<td>cbw</td>
<td>extsb</td>
<td>extend sign byte</td>
</tr>
<tr>
<td>clc,cld,cmc</td>
<td>mcrf crfD,crfS</td>
<td>move condition register field</td>
</tr>
<tr>
<td></td>
<td>mtcrf CRM,rS</td>
<td>move to condition register fields</td>
</tr>
<tr>
<td></td>
<td>mcrxr crfD</td>
<td>move to condition register from XER</td>
</tr>
<tr>
<td></td>
<td>mfcr rD</td>
<td>move from condition register</td>
</tr>
<tr>
<td>cli</td>
<td>mtmsr rS</td>
<td>move to machine state register with appropriate bit mask</td>
</tr>
<tr>
<td></td>
<td>mfmsr rD</td>
<td>move from machine state register with appropriate bit mask</td>
</tr>
<tr>
<td>cls</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>cmp (compare register to immediate value)</td>
<td>cmpwi rA,SIMM</td>
<td>A signed comparison is made between rA and the sign-extended value of SIMM.</td>
</tr>
<tr>
<td></td>
<td>cmpi crfD,L,rA,SIMM</td>
<td></td>
</tr>
<tr>
<td>cmp (compare register to register)</td>
<td>cmpw rA,rB</td>
<td>A signed comparison is made between rA and rB.</td>
</tr>
<tr>
<td></td>
<td>cmp crfD,L,rA,rB</td>
<td></td>
</tr>
<tr>
<td>x86 Instruction</td>
<td>Related PowerPC Mnemonic and Operands</td>
<td>PowerPC InstructionName</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>cmpli crfD,L,rA,UIIMM</td>
<td>An unsigned comparison is made between rA and the zero-extended value of UIIMM.</td>
<td></td>
</tr>
<tr>
<td>cmpl crfD,L,rA,rB</td>
<td>An unsigned comparison is made between rA and rB.</td>
<td></td>
</tr>
<tr>
<td>cmpxchg</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>cwd, cwde</td>
<td>extsw</td>
<td>extend sign word</td>
</tr>
<tr>
<td>daa, das</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>dec</td>
<td>addi rA,rA,-1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>subi rA,rA,1</td>
<td></td>
</tr>
<tr>
<td>div, idiv</td>
<td>divdux rD,rA,rB</td>
<td>divide double word unsigned</td>
</tr>
<tr>
<td></td>
<td>divdx rD,rA,rB</td>
<td>divide double word</td>
</tr>
<tr>
<td>enter</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>hlt</td>
<td>Similar to the PowerPC's checkstop state.</td>
<td></td>
</tr>
<tr>
<td>imul</td>
<td>mulli rD,rA,SIMM</td>
<td>multiply low immediate</td>
</tr>
<tr>
<td>in, ins, insb, insw, insd</td>
<td>PowerPC memory mapped I/O is performed using load/store instructions.</td>
<td></td>
</tr>
<tr>
<td>inc</td>
<td>addi rA,rA,1</td>
<td>add immediate: one to register</td>
</tr>
<tr>
<td>int</td>
<td>sc, tw</td>
<td>system call, trap word</td>
</tr>
<tr>
<td>int 03</td>
<td>Equivalent to PowerPC definition of single-step exception</td>
<td></td>
</tr>
<tr>
<td>into</td>
<td>sc, tw</td>
<td>system call, trap word</td>
</tr>
<tr>
<td>invd</td>
<td>dcbz rA,rB</td>
<td>data cache block set to zero</td>
</tr>
<tr>
<td></td>
<td>icbi rA,rB</td>
<td>instruction cache block invalidate</td>
</tr>
<tr>
<td></td>
<td>dcbi rA,rB</td>
<td>data cache block invalidate</td>
</tr>
<tr>
<td>invlpg</td>
<td>tlbie rB</td>
<td>TLB invalidate all (64-bit only)</td>
</tr>
</tbody>
</table>
## Table B-1
Integer Instruction Cross-References (Continued)

<table>
<thead>
<tr>
<th>x86 Instruction</th>
<th>Related PowerPC Mnemonic and Operands</th>
<th>PowerPC InstructionName</th>
</tr>
</thead>
<tbody>
<tr>
<td>tbia</td>
<td>TLB invalidate all</td>
<td></td>
</tr>
<tr>
<td>tbsync</td>
<td>TLB synchronize</td>
<td></td>
</tr>
<tr>
<td>iret</td>
<td>rfi</td>
<td>return from interrupt</td>
</tr>
<tr>
<td>ja</td>
<td>bge</td>
<td>branch if greater than or equal</td>
</tr>
<tr>
<td>jb</td>
<td>blt</td>
<td>branch if less than</td>
</tr>
<tr>
<td>jbe</td>
<td>beq</td>
<td>branch if equal</td>
</tr>
<tr>
<td>jc</td>
<td>n/a</td>
<td>branch if equal</td>
</tr>
<tr>
<td>je</td>
<td>beq</td>
<td>branch if equal</td>
</tr>
<tr>
<td>jg</td>
<td>bgt</td>
<td>branch if greater than</td>
</tr>
<tr>
<td>jge</td>
<td>bge</td>
<td>branch if greater than or equal</td>
</tr>
<tr>
<td>jl</td>
<td>blt</td>
<td>branch if less than</td>
</tr>
<tr>
<td>jle</td>
<td>ble</td>
<td>branch if less than or equal</td>
</tr>
<tr>
<td>jmp</td>
<td>b</td>
<td>unconditional branch</td>
</tr>
<tr>
<td>jnb</td>
<td>bge</td>
<td>branch if greater than or equal to</td>
</tr>
<tr>
<td>jnc</td>
<td>n/a</td>
<td>branch if not equal</td>
</tr>
<tr>
<td>jne</td>
<td>bne</td>
<td>branch if not summary overflow</td>
</tr>
<tr>
<td>jno</td>
<td>bns</td>
<td>branch if not summary overflow</td>
</tr>
<tr>
<td>jnp</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>jns</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>jnz</td>
<td>bne</td>
<td>branch if not equal</td>
</tr>
<tr>
<td>jo</td>
<td>bso</td>
<td>branch if summary overflow</td>
</tr>
<tr>
<td>jp</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>jpe</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>jpo</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>js</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>jz</td>
<td>beq</td>
<td>branch if equal</td>
</tr>
</tbody>
</table>
### Table B-1
Integer Instruction Cross-References (Continued)

<table>
<thead>
<tr>
<th>x86 Instruction</th>
<th>Related PowerPC Mnemonic and Operands</th>
<th>PowerPC Instruction Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>lahf</td>
<td>Similar to mfcr instruction</td>
<td></td>
</tr>
<tr>
<td>lar</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>lds, les, lfs, lgs, lss</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>lea</td>
<td>la rD, d(rA)</td>
<td>load address</td>
</tr>
<tr>
<td></td>
<td>la rD, variable</td>
<td>load address</td>
</tr>
<tr>
<td>leave</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>lgdt</td>
<td>See Chapter 8</td>
<td></td>
</tr>
<tr>
<td>lidt</td>
<td>See Chapter 8</td>
<td></td>
</tr>
<tr>
<td>lidt</td>
<td>See Chapter 8</td>
<td></td>
</tr>
<tr>
<td>lmsw</td>
<td>mfmsr rD</td>
<td>move from machine state register</td>
</tr>
<tr>
<td>lock</td>
<td>ldarx rD, rA, rB</td>
<td>load double word and reserve indexed</td>
</tr>
<tr>
<td></td>
<td>lwax rD, rA, rB</td>
<td>load word and reserve indexed</td>
</tr>
<tr>
<td></td>
<td>stdcx. rS, rA, rB</td>
<td>store double word conditional indexed</td>
</tr>
<tr>
<td></td>
<td>stwcx. rS, rA, rB</td>
<td>store word conditional indexed</td>
</tr>
<tr>
<td>lods, lodsb, lodsw, lodsd</td>
<td>Similar to the load multiple word (lmw) instruction</td>
<td></td>
</tr>
<tr>
<td>loop, loope, loopne</td>
<td>Equivalent to particular forms of PowerPC branch instructions (see Chapter 6)</td>
<td></td>
</tr>
<tr>
<td>lsl</td>
<td>mtsr i rS</td>
<td>move to segment register (32-bit only)</td>
</tr>
<tr>
<td></td>
<td>mtsrin rS, rB</td>
<td>move to segment register indirect (32-bit only)</td>
</tr>
<tr>
<td></td>
<td>mfr rD, SR</td>
<td>move from segment register</td>
</tr>
<tr>
<td>x86 Instruction</td>
<td>Related PowerPC Mnemonic and Operands</td>
<td>PowerPC InstructionName</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>ltr</td>
<td>n/a</td>
<td>move from segment register indirect [32-bit only]</td>
</tr>
<tr>
<td>mov, movsb, movsw,</td>
<td>mr rA,rS</td>
<td>move register</td>
</tr>
<tr>
<td>movsd</td>
<td>lbz rD,d(rA)</td>
<td>load byte and zero</td>
</tr>
<tr>
<td></td>
<td>stb rS,d(rA)</td>
<td>store byte</td>
</tr>
<tr>
<td></td>
<td>lhz rD,d(rA)</td>
<td>load half-word and zero</td>
</tr>
<tr>
<td></td>
<td>sth rS,d(rA)</td>
<td>store half-word</td>
</tr>
<tr>
<td></td>
<td>lwz rD,d(rA)</td>
<td>load word and zero</td>
</tr>
<tr>
<td></td>
<td>stw rS,d(rA)</td>
<td>store word</td>
</tr>
<tr>
<td></td>
<td>ld rD,ds(rA)</td>
<td>load double word</td>
</tr>
<tr>
<td></td>
<td>std rS,ds(rA)</td>
<td>store double word</td>
</tr>
<tr>
<td>mul</td>
<td>mullwx rD,rA,rB</td>
<td>multiply low</td>
</tr>
<tr>
<td></td>
<td>mulldx rD,rA,rB</td>
<td>multiply low double word</td>
</tr>
<tr>
<td></td>
<td>mullhwx rD,rA,rB</td>
<td>multiply high word</td>
</tr>
<tr>
<td></td>
<td>mulhdx rD,rA,rB</td>
<td>multiply high double word</td>
</tr>
<tr>
<td></td>
<td>mullhwx rD,rA,rB</td>
<td>multiply high word unsigned</td>
</tr>
<tr>
<td></td>
<td>mulhdx rD,rA,rB</td>
<td>multiply high double word unsigned</td>
</tr>
<tr>
<td>neg</td>
<td>negx rD,rA</td>
<td>negate</td>
</tr>
<tr>
<td>nop</td>
<td>nop</td>
<td>no operation</td>
</tr>
<tr>
<td></td>
<td>ori r0,r0,0</td>
<td></td>
</tr>
<tr>
<td>x86 Instruction</td>
<td>Related PowerPC Mnemonic and Operands</td>
<td>PowerPC InstructionName</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>not</td>
<td>not rA,rS</td>
<td>not (complement register)</td>
</tr>
<tr>
<td>or</td>
<td>orx rA,rS,rB</td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td>orcx rA,rS,rB</td>
<td>OR with complement</td>
</tr>
<tr>
<td>ori</td>
<td>rA,rS,UIMM</td>
<td>OR immediate</td>
</tr>
<tr>
<td>oris</td>
<td>rA,rS,UIMM</td>
<td>OR immediate shifted</td>
</tr>
<tr>
<td></td>
<td>out, outs, outsb, outsw, outsd</td>
<td>PowerPC memory mapped I/O is performed using load/store instructions.</td>
</tr>
<tr>
<td></td>
<td>pop, popa, popf, push, pusha, pushf</td>
<td>n/a</td>
</tr>
<tr>
<td>rcl, rcr, rol, ror</td>
<td>ridicl rA,rS,SH,MB</td>
<td>rotate left double word immediate then clear left</td>
</tr>
<tr>
<td></td>
<td>ridicr rA,rS,SH,ME</td>
<td>rotate left double word immediate then clear right</td>
</tr>
<tr>
<td></td>
<td>ridic rA,rS,SH,MB</td>
<td>rotate left double word immediate then clear</td>
</tr>
<tr>
<td></td>
<td>riwinm rA,rS,SH,MB,ME</td>
<td>rotate left word immediate the AND with mask</td>
</tr>
<tr>
<td></td>
<td>ridcl rA,rS,rb,MB</td>
<td>rotate left double word then clear left</td>
</tr>
<tr>
<td></td>
<td>ridcr rA,rS,rb,ME</td>
<td>rotate left double word then clear right</td>
</tr>
<tr>
<td></td>
<td>riwnm rA,rS,rb,MB,ME</td>
<td>rotate left word then AND with mask</td>
</tr>
<tr>
<td></td>
<td>riwimi rA,rS,SH,MB,ME</td>
<td>rotate left word immediate then mask insert</td>
</tr>
<tr>
<td>x86 Instruction</td>
<td>Related PowerPC Mnemonic and Operands</td>
<td>PowerPC Instruction Name</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>lodsb/w/d</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>movsb/w/d</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>scasb/w/d</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>stosb/w/d</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>repe, repne</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>Generally equivalent to an unconditional branch to link register (blr)</td>
<td></td>
</tr>
<tr>
<td>reif</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>sasl</td>
<td>mcrl crfD,crfS</td>
<td>move condition register field</td>
</tr>
<tr>
<td></td>
<td>mcrlx crfD</td>
<td>move to condition register from XER</td>
</tr>
<tr>
<td></td>
<td>mfcr rD</td>
<td>move from condition register</td>
</tr>
<tr>
<td>sar</td>
<td>sradi rA,rS,SH</td>
<td>shift right algebraic double word immediate</td>
</tr>
<tr>
<td></td>
<td>srawi rA,rS,SH</td>
<td>shift right algebraic word immediate</td>
</tr>
<tr>
<td></td>
<td>srad rA,rS,rB</td>
<td>shift right algebraic double word</td>
</tr>
<tr>
<td></td>
<td>sraw rA,rS,rB</td>
<td>shift right algebraic word</td>
</tr>
<tr>
<td>sbb</td>
<td>See sub</td>
<td></td>
</tr>
<tr>
<td>scasb/w/d</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>setxx—(all forms of the set instruction)</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>x86 Instruction</td>
<td>Related PowerPC Mnemonic and Operands</td>
<td>PowerPC InstructionName</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>shl, shld</td>
<td>sldi rA,rS,n (n&lt;64)</td>
<td>shift left double immediate</td>
</tr>
<tr>
<td></td>
<td>slwi rA,rS,n (n&lt;32)</td>
<td>shift left word immediate</td>
</tr>
<tr>
<td></td>
<td>sli rA,n (same)</td>
<td></td>
</tr>
<tr>
<td>shr, shrd</td>
<td>srdi rA,rS,n (n&lt;64)</td>
<td>shift right double immediate</td>
</tr>
<tr>
<td></td>
<td>srwi rA,rS,n (n&lt;32)</td>
<td>shift right word immediate</td>
</tr>
<tr>
<td></td>
<td>sri rA,n (same)</td>
<td></td>
</tr>
<tr>
<td>sidt</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>sldt</td>
<td>See Chapter 8</td>
<td></td>
</tr>
<tr>
<td>smsw</td>
<td>mtmsr rD</td>
<td>move to machine state register</td>
</tr>
<tr>
<td>stc</td>
<td>mtcr</td>
<td>move to condition register</td>
</tr>
<tr>
<td>std</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>sti</td>
<td>mtmsr rD</td>
<td>move to machine state register</td>
</tr>
<tr>
<td>stosb, stosw, stosd</td>
<td>Similar to the store multiple word instruction (stmw)</td>
<td></td>
</tr>
<tr>
<td>str</td>
<td>n/a</td>
<td>See Chapter 8</td>
</tr>
<tr>
<td></td>
<td>See Chapter 8</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>subfcx rD,rA,rB</td>
<td>subtract from carrying</td>
</tr>
<tr>
<td></td>
<td>subfex rD,rA,rB</td>
<td>subtract from extended</td>
</tr>
<tr>
<td></td>
<td>subfic rD,rA,SI/MM</td>
<td>subtract from immediate carrying</td>
</tr>
<tr>
<td></td>
<td>subfmex rD,rA</td>
<td>subtract from minus one extended</td>
</tr>
<tr>
<td></td>
<td>subfx rD,rA,rb</td>
<td>subtract from</td>
</tr>
<tr>
<td></td>
<td>subfzex rD,rA</td>
<td>subtract from zero extended</td>
</tr>
<tr>
<td>verr, verw</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>xadd</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>xchg</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>xlat, xlatb</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>xor</td>
<td>xorir rA,rS,UI/MM</td>
<td>XOR immediate</td>
</tr>
<tr>
<td></td>
<td>xorix rA,rS,UI/MM</td>
<td>XOR immediate shifted</td>
</tr>
<tr>
<td></td>
<td>xorx rA,rS,rB</td>
<td>XOR</td>
</tr>
</tbody>
</table>
Unfortunately, there’s far less commonality with floating-point instructions between the two processors. Most x86 instructions don’t have equivalents on the PowerPC processors. The few floating-point instructions that do have correspondents are shown in Table B-2.

### Table B-2
Floating-Point Cross-References

<table>
<thead>
<tr>
<th>x86 Instruction</th>
<th>Related PowerPC Mnemonic and Operands</th>
<th>PowerPC Instruction Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>fabs</td>
<td>fabs frD,frB</td>
<td>floating absolute value</td>
</tr>
<tr>
<td></td>
<td>fnabs frD,frB</td>
<td>floating negative absolute value</td>
</tr>
<tr>
<td>fadd</td>
<td>fadd frD,frA,frB</td>
<td>floating add (double-precision)</td>
</tr>
<tr>
<td></td>
<td>fadd ds frD,frA,frB</td>
<td>floating add single</td>
</tr>
<tr>
<td>fcom</td>
<td>fcmpu crfD,frA,frB</td>
<td>floating compare unordered</td>
</tr>
<tr>
<td></td>
<td>fcmpo crfD,frA,frB</td>
<td>floating compare ordered</td>
</tr>
<tr>
<td>fdiv, fdivp</td>
<td>fdiv crfD,frA,frB</td>
<td>floating divide (double-precision)</td>
</tr>
<tr>
<td></td>
<td>fdvps crfD,frA,frB</td>
<td>floating divide single</td>
</tr>
<tr>
<td>fnesi, fneni</td>
<td>mtspr ra, FPSR</td>
<td>move to special-purpose register: floating-point status and control register</td>
</tr>
<tr>
<td>fmul, fmulp</td>
<td>fmul crfD,frA,frC</td>
<td>floating multiply (double-precision)</td>
</tr>
<tr>
<td></td>
<td>fmuls crfD,frA,frC</td>
<td>floating multiply single</td>
</tr>
<tr>
<td>fsqrt</td>
<td>fsqrte crfD,frB</td>
<td>floating reciprocal square root estimate</td>
</tr>
<tr>
<td></td>
<td>fsqrt crfD,frB</td>
<td>floating square root (double-precision)</td>
</tr>
<tr>
<td></td>
<td>fsqrts crfD,frB</td>
<td>floating square root single</td>
</tr>
<tr>
<td>fsub, fsubp, fsubr</td>
<td>fsub crfD,frA,frB</td>
<td>floating subtract (double-precision)</td>
</tr>
<tr>
<td></td>
<td>fsubs crfD,frA,frB</td>
<td>floating subtract single</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FLOATING POINT ON THE POWERPC

INTRODUCTION

Many x86 programmers have never dabbled with floating-point arithmetic. The i486 microprocessor was the first family member to include the floating-point coprocessor in the same package as the integer unit. In fact, floating point always seems to be particularly de-emphasized in the x86 architecture. Coupled with the fact that many current PC applications have little need for floating-point math, this area is generally neglected by x86 programmers.

When the PowerPC architects were contemplating the feature set that would give PowerPC processors an edge in the market and make them attractive to developers, one of the key design goals was floating-point performance. The emphasis on floating point has resulted in floating-point operations having a high spec performance rating. The following sections review and discuss floating-point numerical representation, PowerPC floating-point implementation, and practical floating-point code examples.

FLOATING-POINT REPRESENTATION — A REVIEW

In 1985, the IEEE (Institute of Electrical and Electronics Engineers) released a floating-point math standard is that used on both x86 and PowerPC processors. This standard, IEEE-754, defines two binary, fixed-length formats: single-precision and double-precision. Single-precision floating-point data is represented using a 32-bit value; double-precision requires 64 bits.
Figure C-1 shows the format for both single- and double-precision floating-point numbers.

The difference between single- and double-precision floating-point numbers is just that: their precision. The number of bits used to represent the exponent and fraction fields increase as the precision increases.

The three fields that are common to both floating-point formats are the sign bit (S), the exponent field (EXP), and the fraction field (FRACTION). Each component is described below.

- **Sign bit (S)**
  A positive sign (+) is represented by a zero (0) in this position; a negative sign (-) is represented by a one (1). On the PowerPC, the sign bit is bit zero.

- **Exponent (EXP)**
  The exponent in the IEEE format is a biased exponent. To obtain the actual exponent from the biased exponent, you must subtract 127 from a single-precision (32-bit) exponent and 1023 from a double-precision (64-bit) exponent.

  A biased exponent eliminates the need for a sign bit on the exponent. Two exponent values are reserved in each precision format: all bits set and all bits cleared. Refer to the discussion of infinities, NaNs, zeroes, and denormalized numbers later in this appendix for details on the reserved values.
• **Fraction (FRACTION)**
The fractional part of the floating-point number resides in the remaining bits. It is evaluated as shown in Figure C-5. The fraction and the implied bit are used together when determining the final value of a floating-point number.

• **Implied bit (not shown)**
The implied bit is used as the *whole number* portion of the fraction and is not explicitly represented by the bit values of the floating-point number. Depending on the type of floating-point number, the implied bit determines if the fractional component is of the form: 1.FRACTION or 0.FRACTION. Where FRACTION is the value calculated from the FRACTION field, as discussed previously, and 1 or 0 is the implied bit.

### Normals, Denormals, and Zeros

Processors such as the i486, Pentium, and PowerPC implementations require the ability to represent not only numerical floating-point values, but also non-numerical values such as infinity. To accomplish this, the bit settings of single- or double-precision numbers have definitions for all representable values, both numeric and non-numeric. Depending on the sign, exponent, and fraction fields, the floating-point number falls into one of several classifications of floating-point numbers.

Numeric floating-point values are categorized as *binary* floating-point numbers. Three types of binary floating-point numbers are supported by the PowerPC architecture: normals, denormals, and zeroes. Each type has a single-precision and double-precision representation.

Figure C-2 shows all representable floating-point values and the associated fields of the single- or double-precision data and their relationship to the real number line. Notice, for example, that the normals can represent numbers of greater magnitude than denormals. There is also a range of values on either side of zero that are too small to represent using either single- or double-precision denormals.

Figure C-2 contains a real number line to the right of the table. The top of the number line points in the direction of positive infinity; the bottom points towards negative infinity. Zero occurs at the very center of the number line. The position of each table entry corresponds to a position on the real number line.
For example, the top of the table represents the positive NaN values, followed by positive infinity. These correspond to the positive infinity area on the real number line. Next, there is a large (dependent on precision) range of normalized numbers. Denormalized numbers are smaller in magnitude than normalized and reside closer to zero on the real number line. Next, there is a range of numbers that is too small to represent using floating-point numbers. Positive zero comes next in the table. Each floating-point category is repeated as the real number line continues toward negative infinity.

The size and placement of floating-point categories in the number line of Figure C-2 is not to scale. However, it does provide a useful view of the relationship between the various categories of floating-point numbers.

### Normals

Normalization of a floating-point number refers to the process of manipulating the significand and exponent value to ensure that the leading significand bit is a 1. In particular, the significand shifted to the right while the exponent is incremented according to the number of bits shifted.
Normalized floating-point numbers (normals) can represent values in the range of the following magnitudes:

- Single-Precision: $1.2E-38 \leq \text{value} \leq 3.4E+38$
- Double-Precision: $2.2E-308 \leq \text{value} \leq 1.8E+308$

Single-precision normals can hold values with an unbiased exponent in the range $-126$ to $+127$. Double-precision normals can hold values with an unbiased exponent in the range: $-1022$ to $+1023$.

Figure C-3 shows the equation that is used to determine a normalized floating-point number from the 32-bit or 64-bit normalized floating-point value. The implied bit for normalized numbers is always one.

\[
\text{Normalized FP}_{32-\text{or} \ 64\text{-bit}} = (-1)^S 2^{UEXP} (1.\text{fraction})
\]

**Figure C-3**

*The normalized floating-point equation normalizes both single- and double-precision values.*

**Denormals**

A floating-point number is denormalized when its high-order bit (not including the implied bit) is not 1. Denormals feature the significand shifted to the right while the exponent is incremented according to the number of bits shifted until the exponent equals the format’s minimum value: $-126$ for single-precision and $-1022$ for double-precision. Numbers represented in this way use fewer significant bits to represent numbers, resulting in a loss of precision.

Denormals represent the set of floating-point numbers that are smaller in magnitude than normalized numbers yet still larger than zero. Refer to Figure C-2 for the relationship between denormalized floating-point numbers and other floating-point numbers.

The biased exponent on all denormals is zero. This means that the unbiased exponent, which is used in the equations below, is the minimum possible exponent value for the precision of the number. Calculation of denormalized floating-point numbers is performed using the equation shown in Figure C-3. However, single-precision denormals always have $UEXP = -126$; double-precision denormals always have $UEXP = -1022$. The implied bit is always zero on denormals.
Figure C-4 shows the equation used to calculate the floating-point number from the 32-bit or 64-bit denormalized floating-point value.

(a) The Single-precision Normalized Floating-point Equation

Denormalized FP<sub>32-bit</sub> = (-1)<sup>S</sup> 2<sup>-126</sup> (0.fraction)

(b) The Double-precision Normalized Floating-point Equation

Denormalized FP<sub>64-bit</sub> = (-1)<sup>S</sup> 2<sup>-1022</sup> (0.fraction)

<table>
<thead>
<tr>
<th>Figure C-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Denormalized Floating-Point equations.</td>
</tr>
</tbody>
</table>

Zeroes

It is possible to represent both +0 and -0 as floating-point values. For floating-point zeroes, EXPONENT = 0 and FRACTION = 0. On PowerPC implementations, floating-point comparisons disregard the sign of a zero value: +zero = -zero. Figure C-3 shows the floating-point data fields for the zero values.

Infinities and NaNs

Non-numeric floating-point values are categorized as either infinities (+oo or -oo) or Not A Number (NaN) values. Each non-numeric floating-point value has both a single-precision and double-precision representation.

Infinities

Two non-numeric values represented in floating-point format are positive and negative infinity. The infinity values always have the maximum biased exponent value, EXPONENT = 0b11111111 for single-precision and 0b111111111111 for double-precision, and a FRACTION of zero, as shown in Figure C-2.
Infinity values are used in a limited number of arithmetic operations. However, when using infinity values in the following situations, an invalid operation exception will be generated:

- Division of infinity by infinity
- Subtraction of infinity from infinity
- Multiplication of infinity by zero

The invalid operation exception is discussed in Chapter 10, “Exceptions and Interrupts.”

Not A Number (NaN) Values

The second category of non-numeric values that can be represented in floating-point format is NaN values. Like the infinities, NaNs always have the maximum exponent value. To distinguish NaNs from ±∞, NaN FRACTION fields can be any value except zero. Figure C-2 shows the floating-point data configuration of NaNs. Note that despite the placement of NaNs in Figure C-2, the sign bit for NaNs is ignored — they are, after all, not numbers.

NaNs result from invalid operations such as the illegal operations involving infinities, described previously. There are two types of NaN values: quiet NaNs and signaling NaNs. If the highest-order bit of the FRACTION field is set (bit 9 for single-precision and bit 12 for double-precision), the NaN is a quiet NaN. If the high-order FRACTION bit is cleared, the NaN is a signaling NaN. Here is the difference between the two NaNs:

- Quiet NaN (QNaN)
  Quiet NaNs result from invalid floating-point operations when the invalid operation exception is disabled (floating-point SCR[VE] = 0). When a floating-point operation results in a QNaN value, an exception is not generated. QNaN values can be used as diagnostic information for invalid floating-point operations when working with floating-point code.

- Signaling NaN (SNaN)
  Signaling NaNs generate an invalid operation exception when they are used in floating-point arithmetic operations. SNaNs can only be generated when the invalid operation exception is enabled in the floating-point status and control register (FPSRC[VE] = 1). More information on the invalid operation exception can be found in Chapter 10, “Exceptions and Interrupts.”
FLOATING POINT ON POWERPC PROCESSORS

The IEEE standard requires single-precision arithmetic for single-precision operand values. However, when using double-precision arithmetic, it is legal to mix and match the precision of the operands. In other words, double-precision arithmetic instructions will accept single-precision, double-precision, or both types of operands.

The PowerPC UISA requires that independent of the operand precision, double-precision floating-point arithmetic instructions will produce double-precision results. And, of course, single-precision arithmetic produces single-precision results.

PowerPC processors use the double-precision format internally. That means that every time you load a single-precision floating-point number from memory, the processor must convert it to double-precision format before operating on the value. Single-precision values that are converted to double-precision in 64-bit FPRs have bits FRACTION[35-63] set to zero. That same value must be converted back to single-precision by software when used with arithmetic instructions. However, the processor will convert from double- to single-precision when storing FP values back into memory.

Despite the conversion, single-precision floating-point operations are usually faster on all PowerPC implementations. The user’s manual for each PowerPC processor advises that if double-precision is not required, using single-precision operations could increase performance.

There is a set of rules that determine the sign of a valid arithmetic operation (one that does not generate an exception). Except for the NaN values, these rules apply to all floating-point operands and results. Most of the following rules comply with the most important rule in arithmetic: common sense.

- For floating-point addition operations, the result of the operation assumes the sign of the source operand that is greater in magnitude. If both source operands have the same sign, the result assumes the equivalent sign.

- The result of a floating-point subtraction operation assumes the sign of the source operand by treating the subtraction operation as the addition of a negative value. For example, (x−y) is the same as (x + (−y)); in this case, the rules for floating-point addition apply.
For addition or subtraction operations that result in a value of exactly zero, the sign of the result is positive. The single exception to this rule depends on the rounding mode: round toward negative infinity would cause a negative result in the above operation. Rounding modes are discussed below.

- For multiplication and division operations, the sign of the result is the exclusive or (XOR) of the signs of the two source operands.

- For round to single-precision or convert to/from integer operations, the sign of the result is the sign of the source operand.

- The sign of the result of \( \text{fsqrt} \) (floating-point square root) or \( \text{frsqrte} \) (reciprocal square root estimate) operations is always positive. The two exceptions to this rule are the following: \( \text{fsqrt}(-0) = -0 \) and \( \text{frsqrte}(-0) = -\infty \).

Note that multiply-add instructions require special attention because two floating-point operations are performed in one instruction. The above rules are first applied to the multiplication operation and then to the addition or subtraction operation since one of the source operands to the addition or subtraction operation is the result of the multiplication operation.

**Working With Single-Precision Data**

The 64-bit PowerPC floating-point registers (FPRs) always contain double-precision formatted data when in use. And floating-point arithmetic and move instructions always use double-precision format. In the previous discussion, we examined who is responsible for the conversion process. However, there are occasions when FP data needs to be interpreted as a single-precision value. There are a number of ways to force the use of the single-precision format:

- The load floating-point single-precision instructions will ensure that the processor interprets floating-point values using the single-precision format. In particular, these instructions access a single-precision operand in memory and convert it to double-precision before storing the value in an FPR.
The `frsp` (floating-point round to single-precision) instruction will round from double- to single-precision format. That is, the instruction will round the FP number such that it will fit into the 32-bit, single-precision representation. However, the FP number remains in double-precision format while contained in an FPR.

All single-precision FP arithmetic instructions accept double-precision operands from FPRs and perform the operation using double-precision. The intermediate result is subsequently converted to single-precision format before updating the destination register with the final result. Note that if either source operand cannot be represented using single-precision format, the resulting value will be undefined.

Any single-precision floating-point store instructions will convert from double- to single-precision before storing the operand into memory. If denormalization of the operand is required, it is performed by the processor automatically.

**Real World Examples**

Perhaps the best way to understand floating-point data is to reverse-engineer a floating-point number. In the following example, we'll decode the floating-point number π in both single-precision and double-precision format. Following that, we'll highlight some important points to remember about PowerPC floating point.

Original Number = π = 3.14159.

Single-precision (32-bit) representation of π = 0x40490fd0

The sign (S) bit is 0.
The biased exponent is 128.
The unbiased exponent is 128 - 127 = EXP = 1.

The Fraction is $1 \cdot (\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \frac{1}{64} + \frac{1}{128} + \frac{1}{256} + \frac{1}{512} + \cdots) = 1.57068$.

Using the incomplete fractional value above, the normalized single-precision number is $(-1)^S \cdot 2^{EXP} \cdot (1.57068) = 3.141357$.

**Figure C-5**

Decoding floating-point values.
Figure C-5 illustrates the process of converting a normalized, single-precision (32-bit) floating-point value from the floating-point data format to the floating-point number itself. The number that we’re trying to extract from the 32-bit floating-point value is \( \pi = 3.14159 \). The following code sequence was used to generate the number.

```c
float fl;
void main(void)
{
    fl = 3.14159; // load fl with abbreviated value of pi
}
```

The number that the compiler generated was 0x40490fd0, as shown in Figure C-5. We can separate each of the fields as follows: sign bit = 0b0, exponent field (EXP) = 0b10000000, and the FRACTION field = 0b10010010000111111010000. Using the formula in Figure C-3, we can start to calculate the components of the equation required to determine the value that this single-precision floating-point data represents.

Notice that the highest-order bit in the FRACTION field represents the fractional value one-half. Having a bit set in any of the FRACTION bit fields means that the associated fractional value for that field is summed to determine the fractional part of the equation. Recall that the implied bit for normalized numbers is one; therefore, the fractional part of this number will be of the form 1.FRACTION. In our example, we see that

\[
\frac{1}{2} + \frac{1}{16} + \frac{1}{128} + \frac{1}{4096} + \frac{1}{8192} + \ldots
\]

For every bit that is set in the FRACTION field, the corresponding fraction would be summed together. For the purposes of this example, we’ll stop with

\[
\frac{1}{8192}
\]

Note that the final value will be less precise than if we had completed the calculation of the fraction to completion. The final value of the FRACTION field (using the implied bit) is 1.57068. Solving the normalized number equation using this value, as shown in Figure C-5, results in: \( \pi = 1\times2\times1.57068 = 3.141357 \). The further we carry our addition of fractional values, the more precise the number. With that in mind, we see the dramatic increase in resolution that double-precision format provides.
**PowerPC Floating-Point Facts**

The following list describes aspects of floating-point operation that you should keep in mind when working with floating point on the PowerPC implementations:

- Floating-point division operations are still *very* costly in terms of execution times. In fact, when division operations are used with static values in a compiled program, many compilers will compute the reciprocal of the value and change the operation to a multiply — saving as many as 30 cycles per operation!

- The PowerPC instruction set architecture defines several multiply-add and multiply-subtract instructions. These instructions, which perform two commonly paired operations, represent an important floating-point optimization tool. There are multiply-add and multiply-subtract forms for both single- and double-precision values.

- Keep as much floating-point data in FPRs as possible during complex calculations. Doing so can reduce costly memory accesses.
The MicroAPL directory on the CD contains a demonstration copy of MicroAPL's PortAsm/86 80x86 Assembler to PowerPC Translator. The version found on the CD is a source-level translator which takes 80x86 assembler (in Microsoft's MASM format), and translates it to PowerPC assembler source code. This demonstration version is supplied for your personal evaluation and demonstration only; any commercial use requires a license from MicroAPL Ltd.

The files supplied are:

- **README**
  An introductory file. This file contains contact information for MicroAPL Ltd.

- **PADEMO.EXE**
  An MS-DOS executable of the demonstration version of PortAsm/86. This version is limited to a single source file, and 150 lines maximum. It requires an 80386 or higher processor, and 4MB of RAM. Target assemblers supported are those of IBM, Apple, Motorola and MetaWare.

- **PADEMO.TXT**
  Text-only, shortened version of the PortAsm/86 manual. You can read this on-line with an editor, or print it out (use a monospaced 10- or 12-point font).

- **PRIME.ASM**
  Source to a simple DOS program written in assembler, which you can use to try out PortAsm/86.
OPTIM.ASM
Source that shows how PortAsm optimizes certain types of code sequences.

PortAsm is a complex product that carries out an extensive analysis of the program in order to produce an efficient, optimized translation. However, if you want to get a quick idea of what PortAsm does, try translating the PRIME.ASM program to assemble under the MetaWare assembler, as follows:

```
pademo -a meta prime.asm
```

To see what code optimizations PortAsm can apply, try translating OPTIM.ASM:

```
pademo -nomain -a meta optim.asm
```

Note that executing “pademo” with no arguments prints out a list of the options.
For further information, see the pademo.txt file on the CD. It contains information about how PortAsm translates code, and how you can intervene using hints and hand-written assembler where necessary.

**METAWARE’S DEVELOPMENT TOOLS**

MetaWare has included a trial version of their High C/C++ compiler, assembler, and related tools for use by the readers of *PowerPC Programming for Intel Programmers*. Most of the features discussed below are included in the trial version. In the METAWARE directory, you’ll find the following directory structure:

- **DOCS** — Documentation on compiler options and directives. The hc.hlp file contains nearly 50 pages of compiler usage information.
- **BIN** — The compiler, assembler, and other tools.
- **INC** — Standard include files
- **INCC** — Additional include files
- **LIB** — Libraries
MetaWare delivers high-quality compilers to PowerPC and x86 platforms with its High C/C++ for PowerPC targeting IBM’s PowerPC operating systems (OS/2 and UNIX personalities) and OS/Open operating systems and IBM’s OpenBIOS ROM monitor, the latter two for embedded systems. The MetaWare compiler is based on the SVR4 ABI PowerPC Supplement dated July 25, 1994.

MetaWare’s attention to the evolving ANSI C++ Standard gives you cross-platform compatibility and predictability. Included is a multi-platform, source-level debugger that supports High C/C++ extensions. Also included are Rogue Wave’s Tools.h++ class library and fast libraries with ANSI conformance. High C/C++ compilers produce machine language that is compact, fast, and efficient, allowing large, complex applications to be developed and deployed.

The High C/C++ compiler is a true compiler, not a C to C++ translator. Some of MetaWare’s High C/C++ Compiler features include:

- Eight levels of global optimization
  You can optimize programs for execution speed and/or code size, or to achieve faster compile times, with a single command-line switch. High C/C++ supports the classic optimizations including retro-allocation via the graph coloring technique.

- Optional ANSI-standard conformance

- Wide variety of compiler features available through the use of toggles and pragmas

- Native floating-point code generation

- C or C++ compiler invoked based on user-definable source-file extension

- Source-annotated assembly listings

- Inline functions across compilation units

- IBM OS/Open and OpenBIOS embedded support for both OS/2 and UNIX and for personality-neutral servers on the microkernel

- Supports ELF and DWARF formats
Some last-minute additions to the compiler and tools are summarized below.

- `.endiand (little l big) assembly pseudo-op was added to facilitate writing little endian boot code in big endian environment. This switch causes the instructions and data to be emitted in the byte-order specified. It does not affect the byte-order (little or big endian) of the Elf header.

- By default, the compiler generates little endian targeting code (one advantage of using little endian mode is that you can actually simulate your environment on Solaris PowerPC, WorkPlace Unix, or WorkPlace OS/2). To generate big endian code, include `-HB` on your command line or driver’s `ARGS=` line.

- `cpu_603` is the default target (Note: the code generated `will` run on 601 and most 40x).

- new linker switches:
  - `-Bstart_addr=xxx`
    - `xxx` sets to default 0x10000000. This switch specifies starting virtual address of the binary.
  - `-Bpage_size=xxx`
    - `xxx` sets to default 0x1000. This switch specifies page size alignment.

- `-L$HCDIR/syslib` included in library search path in the driver configuration file. Place your system-specific libraries (e.g., `libgraph.a`, `libos.a`) there so that `-lgraph` and `-los` will automatically pick up the library.

- `-kernel` driver switch omits startup files in link line (required for most kernel program).

- `$HCDIR/lib/src` contains the minimum set of library source that is necessary to build your own C++ library (you don’t normally want to do that). There is also sample `crti.s` and `crtn.s` needed for C++ linking.

- The C library supplied is a generic Solaris PPC style library with several Solaris’s system calls (e.g. `write`, `read`, etc). For embedded applications, you can use the library as is (all you need to do is to supply several of your own system calls for your specific system). The C++ library is self-contained when linking with the C library.

- Users must supply their own system-specific `crt1_{be,le}.o` startup file and put it in the lib directory. In general, for C++ static initialization to work correctly, `crt1_{be,le}.o` should call `__init` and register `__fini` with `atexit` before calling to main.
What's On The CD

POWERPC NEWS ARCHIVE

When was the PowerPC 604 announced? When did Apple agree to license their technology? Find out in the PowerPC News archive. This archive is a tremendous resource for anyone interested in the history of the PowerPC. Spanning just over a year's worth of PowerPC-industry news, these stories cover each major event since the introduction of the first PowerPC-based Macintosh computers.

PowerPC News is an Internet-based news source that is free of charge to subscribers. PowerPC News is published by Internet Publishing which is part of the APT Group Plc. They have access to APT's world-wide publishing resources which allows them the opportunity to focus on developing quality electronic products over the long term.

The premier issue (March 15, 1994) contained stories such as:

- Apple Launches Power Macintosh Line
- Power Macintosh Garners Application Support
- German and Taiwanese Firms To Show PowerPC Machines at CEBIT

And one of the most recent issues (from March 27, 1995) contained stories such as:

- IBM Denies OS/2 Delay — Clings to (Late) Summer Release Date
- Taligent Releases Commonpoint Beta, but Delays OS Launch
- Motorola Denies it Wants 17% of Bull, But it Does Want Joint R&D

The following directory structures divide the information into two categories: feature stories and PowerPC News.

\PPC-NEWS\INDEX.TXT - News Index
\PPC-NEWS\FEATURES\ - Feature stories
\PPC-NEWS\PPCNEWS\ - PowerPC News Stories

PowerPC News has recently announced that news and information is accessible in Hypertext form via the World Wide Web.

To access the service, you will need a direct Internet connection and a Web browser. Many browsers are available, the most popular being Mosaic,
ported to Macintosh. MS-Windows and X-Windows. Mosaic is available on CompuServe (use Mosaic as a search keyword) or via anonymous FTP from:

ftp.ncsa.uiuc.edu

The NCSA machine is very busy so if you can find a local mirror of the site, so much the better.

The PowerPC News home page is located at:

http://power.globalnews.com/

MOTOROLA'S WORLD WIDE WEB INFORMATION

Motorola's acclaimed World Wide Web Site is on the CD in the moto-web directory. Just use your Mosaic browser to open the file `moto-web\start.htm` to access the latest information on PowerPC happenings from Motorola, Inc.

To look at the WWW information on the CD, you'll need a Mosaic browser. Mosaic browsers can be found on CompuServe (use Mosaic as a search keyword) or via anonymous FTP from:

ftp.ncsa.uiuc.edu

The NCSA machine is very busy so if you can find a local mirror of the site, so much the better.

Find out the latest PowerPC news in the HotNews area. Find out when and where PowerPC-related industry events are happening. Plus, get the latest HTML-based technical information on all PowerPC processors.

There have been some minor updates to Motorola's PowerPC Web Server. The source of Web information is Motorola's Server which is located at:

http://www.mot.com/PowerPC

They are interested in any feedback from users. For more information, you can contact the Motorola Webmaster at:

webmaster@risc.sps.mot.com
FirmWorks's Open Firmware Information

Open Firmware is a portable boot firmware system. Boot firmware is the ROM-based software that controls a computer from the time that it is turned on until the primary operating system has taken control of the machine. The main function of boot firmware is to initialize the hardware and then to "boot" (load and execute) the primary operating system. Secondary functions include testing the hardware, managing hardware configuration information, and providing tools for debugging in case of faulty hardware or software.

Open Firmware is portable in the sense that its design is not tied to any particular processor family nor to any particular expansion bus. Open Firmware was specifically designed to support a variety of different processor Instruction Set Architectures (ISAs) and different buses. Open Firmware is already in use on over a million machines, and is supported by several system vendors. A number of bus standards, including PCI, Futurebus+, VME-D, and SBus, include provisions for Open Firmware card identification and booting.

The PowerPC Reference Platform Specification requires that all compliant PowerPC computer systems adopt Open Firmware by mid-1995. Open Firmware provides similar functionality to the BIOS of x86 PCs — and much more. Find out more about Open Firmware by browsing through the information provided by FirmWorks, Inc.

FirmWorks, located in Mountain View, California, provides Open Firmware system ROM implementations, device drivers, training and consulting, as well as Forth ROM monitor products for those who want a Forth development environment instead of a complete Open Firmware system.

The \FIRMWARE directory contains descriptions of FirmWorks's various products and services, as well as some background information on Open Firmware.

The following four file formats are used:

.DOC — Microsoft Word 6.0 file
.PS — PostScript file
.TXT — ASCII text file
.WRI — Windows 3.1 Write file
BIBLIOGRAPHY


INDEX

A
add, 383
addc, 384
adde, 385
addi, 217, 328–330, 380, 386
addis, 11, 217, 389
addme, 390
address
recognition, 258–260
addressing modes, 151–168, 248–249, 256
addze, 391
aliasing, register, 238–239
alignment/misalignment, 162–163, 305–308
AllDone, 360
ALL TO field, 213
ALU (internal arithmetic logic), 44
and, 329, 392
andc, 393
andi, 161
andd, 394
andis, 395
API (abbreviated page index), 253, 265–266
Apple Computers, 1, 663. See also Macintosh
arithmetic operations, 10–11, 328–330
floating-point, 196–198
integer, 177–179
arrays, 17, 20–23
ASCII code, 14
ASR (address space register), 94, 148, 253, 263, 272–273
assembly language examples, 319–260
associative caches, 277–278
associativity, 379
atomic memory access, 174, 279, 356–358

B
B
BAT (block address translation) register, 45, 81–82, 94, 145, 306
basic description of, 116–118, 126–128
caches and, 280–281
manipulation, 354–356
memory management and, 254–265, 269, 273
bc, 397
bcctr, 224, 398
bcctrl, 224
bctrx, 92
bclr, 17, 359, 360, 399
bdnz, 338
BEPI (block effective page index), 253
bgt, 17
BHT (branch history table), 53, 236–237
BHTE field, 136
BI field, 205–207, 223, 224
binary numbers, prefixes for, 7
bit(s)
field/terminology conventions, 6
labels, 62–63, 69–70
master checkstop enable, 119, 120
bl, 327–328
Blackhawk (603/604-based systems), 2, 3
block(s), 37–39. See also BAT register
effective page index (BEPI), 253
logical page index (BLPI), 253
number, physical (PBM), 253
real page number (BRPN), 253, 259–260
blr, 333, 334
bne, 334, 336
bng, 339
BO field, 205–207, 223
BPU (branch processing unit), 38–39, 43, 53, 205–210, 236
branch(es), 205, 234–239, 327–333. See also instruction timing
folding, 43, 233, 235
instructions, 176, 205–210, 223–226
prediction, 43, 205, 227, 233, 234–237
breakpoints, instruction address, 316
BRPN (block real page number), 253, 259–260
bswap, 68
BTAC (branch target address cache), 53, 236–237
buffers
completion, 49–50, 238–239
read/write, 47
rename, 44, 241
translation lookaside (TLB), 217, 253, 268
Bulfinch, Thomas, 77
BUSCSR (bus status and control register), 144, 148
buses, 37–39
byte(s)
addressing, within multibyte operands, 65–67
endian schemes and, 59–60, 62
terminology conventions and, 6
C
C (changed) bit, 266
cache(s), 46, 49, 51, 55, 57
access attributes, 280–281
basic description of, 36, 275–286
endian schemes and, 71
i86, 30–32
lines, 278
management instructions, 215, 281–286
on-chip vs. off-chip, 277
call instructions, 327–328, 330–333
Carroll, Lewis, 151
cdecl, 319
checkstop, sources, 119–122. See also HD10
CIA (current instruction address), 82, 159, 170, 316
CISC (complex instruction set computer) processors, 1, 11, 30
addressing modes and, 151–152, 167
instruction timing and, 227
clrldi, 398
clrlsldi, 401
clrlswi, 402
crlwi, 403
crlrdi, 404
crwi, 405
cmp, 16, 180, 406
cmpd, 407
cmpdi, 408
cmpi, 16, 180, 231, 409
cmpl, 180, 410
cmpled, 411
cmplldi, 412
cmpli, 180, 413
cmplw, 414
cmplwi, 415
cmpw, 416
cmpwi, 326–327, 334, 339, 417
cntlzd, 418
cntlzw, 419
Cocke, John, 3
Cohen, Danny, 60–63
coherency, 280, 281
compare, 180, 197–198
completion buffers, 49–50, 238–239
configuration registers, 94. See also specific registers
constant, use of the term, 232
context synchronization, 102, 169–176, 213–215, 312, 364
context synchronizing exception, 291–292
conversion, endian, 74–75
CR (condition register), 6, 16
context synchronization of, 85–87
branch instructions and, 205–206
instruction timing and, 236–237
CRO–CR3 (control registers), 85–87, 89, 174, 251–252
crand, 420
crandc, 421
creqv, 422
creqd, 180
crnand, 423
crnor, 424
cror, 425
crorc, 426
crorx, 427
CSE (context synchronizing event), 172
CTR (counter register), 79, 224, 237, 337–339
basic description of, 92
branch instructions and, 205
D
DABR (data address breakpoint register/HID5), 94, 109–110, 112, 122
DAE (data access exception), 5, 109. See also DSI
DAR (data address register), 94, 111, 311
DBAT (data block address translation), 126–128, 171–172, 253, 257
dcbf, 284–285, 428
dcbi, 286, 429
dcrst, 284, 430
dcbt, 281–283, 431
dcbst, 281–283, 432
dcbz, 283–284, 433
DCMP (data TLB compare registers), 124, 131, 132
DEC (decrementer register), 94, 108, 134, 137
decremler exceptions, 71, 310
denormals, 678–681
direct-store exception, 310–313
direct-store segments, 5, 250, 274
dispatch, instruction, 39, 239–241, 244. See also instruction timing
divd, 434
divdu, 435
divw, 436–437
diww, 438
DMISS (data miss address registers), 124, 130–132, 133
DMMU (data memory management unit), 253
do while loops, 338–339
Doyle, Arthur Conan, 227, 247
doze mode, 50
DR (debug register), 81
Drexler, K. Eric, 169
DSI (data storage interrupt), 5, 109, 112, 233, 294, 296, 315. See also DAE
the alignment exception and, 305
basic description of, 302–303
the direct-store exception and, 311
memory management and, 262, 268
DSISR (data storage interrupt service register), 94, 112, 302–308
DTLB (data translation lookaside buffer) miss exception, 314–316
dynamic branch prediction, 53, 236–237. See also branch prediction
E
EA (effective address register), 153–155, 164–167, 264
EAR (external access register), 94, 110–111
EAX register, 7, 10, 18, 79–80, 156
strlen() and, 19
toupper() and, 15
ciwx, 110, 256
cowx, 110, 441
ECX (counting register), 12, 81
effective address, 254–255, 265–268.
See also EA
definition of, 249
termology equivalent for, 5
EFLAGS register, 81, 82, 85
eieio, 173, 279–280, 442–443
EIP register, 159, 170, 661
eiwx, 439–440
ELE (exception little endian mode), 70
e else-if operation, 334–336
endian schemes, 59–76, 131, 662–663
bi-endian memory and, 68–70, 74
brief history of, 59–62
endian conversion and, 74–75
eqv, 444
errors. See exceptions
categories/priorities, 292–294. See also specific exceptions
endian schemes, 59–76, 131, 662–663
bi-endian memory and, 68–70, 74
brief history of, 59–62
endian conversion and, 74–75
external, 71, 303
floating-point, 310, 313
model, basic description of, 35
terminalization equivalent for, 5
tracing, 299–301
types of, 288–292
vectors and, 295–318
exowx, 256
extended opcode, 5
extldi, 445
extlwi, 446
extrdi, 447
extrwi, 448
extsb, 449
extsh, 450
extsw, 451
fabs, 452
fadd, 453–454
fadds, 455
fcid, 456
fcmpo, 457
fcmpu, 458
fcidz, 459
fcw, 460
fdiv, 463
fdivs, 464
fetch, instruction, 239, 243–245
FirmWorks, 694
floating-point exceptions, 310, 313
floating-point instructions, 24, 176, 676–687
basic description of, 195–204
cross-reference for, 675
floating-point registers. See FPRs
floating-point units. See FPUs
fmadd, 465
fmdads, 466
fmr, 201–202, 467
fmsub, 468
fmsubs, 469
fmul, 470
fmls, 471
fnabs, 472
fneg, 473
fmadd, 474
fmdadds, 475
fmsub, 476
fmsubs, 477
for loop, 20, 336–338
Fourth programming language, 9
FPRs (floating-point registers), 23, 45, 83, 683–686
addressing modes and, 160
instruction timing and, 241
the PowerPC instruction set and, 187, 210
FPSCR (floating-point status and control register), 88, 200–201
FPUs (floating-point units), 32, 38–39, 44–45, 54–55, 57
fres, 478
frsp, 479
frsqrte, 480
fsel, 481
fsqrt, 482
fsqts, 483
fsub, 484
fsub, 485
full associative caches, 277–278
FXU (fixed-point unit), 5
GDTR (global descriptor table register), 81
getVersion(), 349
GPRs (general-purpose registers), 15–16, 23, 44, 79, 82, 320
addressing modes and, 160, 165–167
arithmetic/logical operations and, 328–330
basic description of, 83
DEC and, 108
instruction timing and, 239, 241
loading values into, 323–325
loop operations and, 336–338
the PowerPC instruction set and, 171, 176–177, 187, 195, 202, 210, 217, 220–221
switch/case operations and, 341
writing the contents of, back to memory, 326
GTone, 341
H
H (hash function identifier), 266
HID0 (checkstop sources and enables register), 71, 112, 119–122, 128–130, 144, 146–147
basic description of, 136–137
exceptions and, 302
HID1 (601 debug mode register), 112, 122–123, 316–318
HID2 (instruction address breakpoint register), 94, 110, 112, 122–124, 133–134, 137–138, 316
HID5 (data address breakpoint register), 94, 109–110, 112, 122
HID15 (processor identification register), 94, 112
hints, 208
Hippocrates, 319
I
I/O (input/output), 5, 81, 110
address space, 250
caches and, 279–280
exceptions and, 306, 310–311
external I/O instructions, 216
memory management and, 254, 274
segment registers and, 102
synchronization, 173
ib, 247, 250, 256, 295
i86, 30–34, 45, 58, 256
addressing modes/operand conventions and, 151, 153–155
bswap, 68
exceptions and, 287–288, 290–291
register set, 79–81
IBAT (instruction block address translation register), 116, 126–128, 253, 257, 304
IBM (International Business Machines), 2–5, 30, 37, 60, 71
AI, 663
WorkPlace OS register user conventions, 320
icbi, 285, 486
ICMP (instruction TLB compare registers), 124, 131–132
IDT (interrupt descriptor table), 290–291, 295
IDTR (interrupt descriptor table register), 81, 295
IEEE, 45, 195, 676–677, 683
if-else operation, 333–334
illegal instructions, 175-176
IMISS (instruction miss address
registers), 94, 124, 130-133
immediate operands, 152, 161
IMMU (instruction memory
management unit), 253
implementation-specific exception
vectors, 313-318
imprecise exception, 294
indexing registers, 12, 164-166
indirect (index) addressing, 154-155
infinities, 681-682
insert(), 344-347
inslwi, 487
insrdi, 488
insrwi, 489
instruction pipelining. See pipeline
integer(s). See also IUs
instruction categories, 176-195
instruction cross-reference, 666-674
operations, interleave, 362-363
Intel, 30-34, 60. See also i386; i486
interleave
integer operations, 362-363
memory accesses, 362
interrupts, 5, 287-318. See also
exceptions
definition of, 288-289
IP register, 170
IQ (instruction queue), 39-41
ISI (instruction storage exception),
253, 262, 264, 268, 296, 304, 315
issue logic, 43-44
ITLB (instruction translation
lookaside buffer) miss
exception, 314-315
IUs (integer units), 5, 32, 38-39, 44,
54, 57, 362
J
jag, 17
jb, 17
jmp, 327-328
K
Kennedy, Robert F., 1
Kp bits, 269, 274
Ks bits, 269, 274
L
L2CR, 144
L2SR, 144
la, 491
labels, bit, 62-63, 69-70
latency, 231-233
lbz, 164, 231, 325, 492
lbzu, 493
lbzux, 494
lbzux, 495
ld, 164, 496
ldarx, 174, 279, 356, 497
ldu, 498
ldux, 499
ldx, 500
lea, 19
lf, 501
lfdu, 502
lfdux, 503
ldx, 504
fs, 245, 505
fsu, 506
fssx, 507
fssx, 508
fha, 509
hrux, 510
hrux, 511
hrux, 512
hrbx, 513
hz, 164, 325, 514
hz, 515
hzux, 516
hzux, 517
li, 324-325, 336, 341, 518
linear address, definition of, 249
lis, 324, 332, 519
lmw, 163, 187, 363, 520
logical
address, definition of, 248
operations, 180-182, 328-330
loop operations, 20, 334-339
LR (link register), 91-92, 328,
331-332
LRU (least recently used) policy, 46, 55
LSb (least significant bit), 60, 64
lswi, 167, 187, 363, 521
lswx, 91, 187, 522
lwa, 253
lwarx, 174, 279, 356, 358, 524
lwaux, 525
lwaux, 526
lwbex, 527
lwx, 528
lwz, 164, 325, 358
lwz, 529
lwzux, 530
lwzx, 337, 531
M
machine check exception, 301-302
Macintosh, 60, 663
macros, 218
main(), 330-333
malloc(), 344-347
maskable exception, 292-293
matrix math, 351-354
MC (mask begin), 183, 186
mcrf, 85, 532
mcrfs, 86, 88, 533
mcx, 85, 534
ME (mask end), 183, 185, 186
memory. See also memory
management
access ordering, 279-280
endianness and, 63-65
model, basic description of, 35
operands, 161
physical, 247-248, 260, 265-266
terminology for, 5-6
units (MUs), 46-47
memory management, 94, 247-274
memory paging, 129-134, 250-251,
262-263
memory protection, 254-256
MMU, 45, 252-253-255
model, basic description of, 36
terminology, 248-249, 253
MESI protocol, 55, 57
MetaWare, 689-691
mfcr, 535
mfdbatl, 356
mfdbatu, 356
mffs, 536
mflr, 332
mffmsr, 537
mfspr, 91, 93, 114, 143, 257, 356,
537-540
mfsr, 541
mfsrsin, 542
mfbr, 114, 543
mftbu, 348
MicroAPL, 688
miscellaneous instructions, 210-217
MMCR0 (monitor mode control
register), 94, 138-144, 366-370
MMU, 45, 252-253-255
mnemonics, simplified, 5, 169,
217-220, 359-360, 375
modDBATpair(), 354
Morgan Kaufman, 4
Mosaic, 693
Motorola, 1, 260
Blackhawk, 2
endian schemes and, 60, 62, 63, 71
terminology, 71
World Wide Web information, 693
move instructions, 167, 201–202
MQ (multiple quotient register), 112, 115–116
mr, 336, 339, 544
MSb (most significant bit), 60, 64–65, 108
MSR (machine state register), 4, 70–72, 81, 94, 124, 144, 314, 316
basic description of, 95–100, 134–136
DABR and, 110
DEC and, 108
memory management and, 254–256, 269, 273
performance monitoring and, 24, 94, 138–144, 314, 365–370
physical address, 5, 254
physical memory, 247–248, 260, 265–266
pipeline, 13, 170
basic description of, 41–42, 228–230
resource management and, 230–231
PIR (processor identification register/HID15), 94, 112
PM (performance monitor) bit, 134
PMC1/PMC2 (performance monitor counter registers), 94, 138–144, 314, 366–370
PMI (performance monitor interrupt), 370
port access registers, 12
PortAsm, 663, 665, 688
POWER architecture, 3–4, 37, 91, 115–116, 274, 372
PowerMacs, 37
power management, 50–51, 55
PowerPC 601, 24–25, 29, 32, 78, 101
basic description of, 37–47
BPUs and, 43
caches and, 276–277
download schemes and, 72–73
exceptions and, 301–318
instruction timing and, 235, 239–240
integer operations and, 362–363
memory management and, 256–258, 260, 268
register set, overview of, 112–124
PowerPC 602, 277
PowerPC 603, 25, 29, 36, 176, 235
basic description of, 47–48
BPUs and, 43
caches and, 277
download schemes and, 72–73
exceptions and, 301–318
instruction timing and, 228, 232, 238–241
integer operations and, 362–363
memory management and, 252, 256–258, 260, 268
the PowerPC programming model and, 78, 94, 100–101
register set, 124–133
PowerPC 603e, 51, 100, 277
PowerPC 604, 24-25, 29, 32, 36, 176, 364
basic description of, 51-55
BAT registers and, 126-128
BPUs and, 43
caches and, 277, 278
endian schemes and, 72-73
IABR and, 137-138
instruction timing and, 228, 232, 236, 238, 241-245
memory management and, 252, 257-258, 260, 268
performance monitoring and, 365-370
the PowerPC programming model and, 78, 94, 101
register set, 134-144
PowerPC 620, 24-25, 29, 32, 36, 78, 94, 101-102
basic description of, 55-58
BAT registers and, 126-128
BPUs and, 43
caches and, 277
endian schemes and, 72-73
IABR and, 137-138
instruction timing and, 228, 232, 236, 238, 241
memory management and, 252, 257-258, 260, 268
performance monitoring and, 365-370
the PowerPC instruction set and,
176, 226
register set, 144-149
rename registers and, 364
PowerPC News archive, 692-693
PP (page protection) bits, 260-261, 269, 270
precedence, 379
precise/imprecise exceptions, 290
preferred form instructions, 176
privileged mode (privileged state), 5
privilege levels, 77-78, 290-291
program exception, 308-310
programming model
 basic description of, 35, 77-150
 definition of, 77
 protection boundaries, 71, 306
 PTE (page table entry), 132-133, 144, 253, 265-269, 271, 315
 basic description of, 265-266
 caches and, 280
 PTEG (page table entry group), 253
 ptr size designations, 153
PVR (processor version register), 94, 100-101
R
R (referenced) bit, 266
RAM (random-access units), 34
read-modify-write operations, 356
read/write buffers, 47
real mode addressing, 248-249, 256
reference events, 368
register(s). See also GPR
specific registers
aliasing, 238-239
basic description of, 78-79
field conventions, 6
indirect mode, 167
indirect with index mode, 166
names, 79
operands, 152, 160-161
user conventions, 320
relative addressing, 154
rename buffers, 44, 241
rename registers, 238-239, 364
reservations, 174, 231
resource management, 230-231
returns, 330-333
revision numbers, 100-101
rfi, 95, 172, 571
rings. See privilege levels
RISC processors, 1-2, 29, 151-155, 161, 167, 369
defining, 11-13
instruction timing and, 227
terminology conventions and, 6
rldecl, 572
rldcr, 573
rl dic, 574
rldcl, 575
rd eq, 576
rd idi, 577
rlimi, 578
rlvimi, 350
rlvlni, 359, 579
rlwim, 580
RM field, 122
rotate/shift instructions, 180-186, 220-222
rotld, 581
rotldi, 582
rotlw, 583
rotlwi, 584
ro trdi, 585
ro t rwi, 586
rounding instructions, floating-point, 198-200
RPA (required physical address register), 124, 131, 133
RPN (real page number), 253, 268, 271
RTC (real-time clock) register,
112-114
run mode exception bit field, 122-123
run mode/trace exception, 316-318
S
sc (system call), 172, 289, 587
SCIUs (single-cycle integer units), 364
scope, 35-37
SDA (sampled data address register), 94, 139, 144, 149, 369-370
SDA1 (table search description register), 94, 104-105, 144
segmentation, 32-34, 250-251, 262-265
segment descriptors, 32, 262-265, 274
segmented address, definition of, 248
serializing execution, 364
Shakespeare, William, 361
shift instructions, 180-186
SIA (sample instruction address register), 94, 139, 144, 149, 369-370
SIU (system interface unit), 46-47, 58
slbia, 588
slbie, 589
sld, 590
sldi, 591
sleep mode, 50
slw, 592
slwi, 334, 359, 593
SMI (system management interrupt), 50-51, 316
SoftPC, 663
software TLB search exceptions, 314
SPEC ratings, 55
speculative execution, 54, 234-236
SPGRs (general special-purpose registers), 105-106
SPRs (special-purpose registers), 82-84, 93-94, 114-115
SR (segment register), 82, 101-103
srd, 594
srdi, 595
sraw, 596
srawi, 597
sr d, 598
srdi, 599
IDG Books Worldwide License Agreement

Important — read carefully before opening the software packet. This is a legal agreement between you (either an individual or an entity) and IDG Books Worldwide, Inc. (IDG). By opening the accompanying sealed packet containing the software disc, you acknowledge that you have read and accept the following IDG License Agreement. If you do not agree and do not want to be bound by the terms of this Agreement, promptly return the book and the unopened software packet(s) to the place you obtained them for a full refund.

1. License. This License Agreement (Agreement) permits you to use one copy of the enclosed Software program(s) on a single computer. The Software is in "use" on a computer when it is loaded into temporary memory (i.e., RAM) or installed into permanent memory (e.g., hard disk, CD-ROM, or other storage device) of that computer.

2. Copyright. The entire contents of this disc and the compilation of the Software are copyrighted and protected by both United States copyright laws and international treaty provisions. You may only (a) make one copy of the Software for backup or archival purposes, or (b) transfer the Software to a single hard disk, provided that you keep the original for backup or archival purposes. The individual programs on the disc are copyrighted by the authors of each program respectively. Each program has its own use permissions and limitations. To use each program, you must follow the individual requirements and restrictions detailed for each in Appendix D of this Book. Do not use a program if you do not want to follow its Licensing Agreement. None of the material on this disc or listed in this Book may ever be distributed, in original or modified form, for commercial purposes.

3. Other Restrictions. You may not rent or lease the Software. You may transfer the Software and user documentation on a permanent basis provided you retain no copies and the recipient agrees to the terms of this Agreement. You may not reverse engineer, decompile, or disassemble the Software except to the extent that the foregoing restriction is expressly prohibited by applicable law. If the Software is an update or has been updated, any transfer must include the most recent update and all prior versions.

4. Limited Warranty. IDG Warrants that the Software and disc are free from defects in materials and workmanship for a period of sixty (60) days from the date of purchase of this Book. If IDG receives notification within the warranty period of defects in material or workmanship, IDG will replace the defective disc. IDG's entire liability and your exclusive remedy shall be limited to replacement of the Software, which is returned to IDG with a copy of your receipt. This Limited Warranty is void if failure of the Software has resulted from accident, abuse, or misapplication. Any replacement Software will be warranted for the remainder of the original warranty period or thirty (30) days, whichever is longer.

5. No Other Warranties. To the maximum extent permitted by applicable law, IDG and the author disclaim all other warranties, express or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose, with respect to the Software, the programs, the source code contained therein and/or the techniques described in this Book. This limited warranty gives you specific legal rights. You may have others which vary from state/jurisdiction to state/jurisdiction.

6. No Liability For Consequential Damages. To the extent permitted by applicable law, in no event shall IDG or the author be liable for any damages whatsoever (including without limitation, damages for loss of business profits, business interruption, loss of business information, or any other pecuniary loss) arising out of the use of or inability to use the Book or the Software, even if IDG has been advised of the possibility of such damages. Because some states/jurisdictions do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation may not apply to you.

7. U.S.Government Restricted Rights. Use, duplication, or disclosure of the Software by the U.S. Government is subject to restrictions stated in paragraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause of DFARS 252.227-7013, and in subparagraphs (a) through (d) of the Commercial Computer—Restricted Rights clause at FAR 52.227-19, and in similar clauses in the NASA FAR supplement, when applicable.
Title of this book: Power PC Programming For Intel Programmers

My overall rating of this book: □ Very good □ Good □ Satisfactory □ Fair □ Poor

How I first heard about this book:
□ Found in bookstore; name:
□ Advertisement:
□ Word of mouth; heard about book from friend, co-worker, etc.:
□ Other:

What I liked most about this book:

What I would change, add, delete, etc., in future editions of this book:

Other comments:

Number of computer books I purchase in a year: □ 1 □ 2-5 □ 6-10 □ More than 10

I would characterize my computer skills as: □ Beginner □ Intermediate □ Advanced □ Professional

I use □ DOS □ Windows □ OS/2 □ Unix □ Macintosh □ Other:

I would be interested in new books on the following subjects:
(please check all that apply, and use the spaces provided to identify specific software)

I use a PC at (please check all that apply): □ home □ work □ school □ Other:

The disks I prefer to use are □ 5.25 □ 3.5 □ Other:

I have a CD ROM: □ yes □ no

I plan to buy or upgrade computer hardware this year: □ yes □ no

I plan to buy or upgrade computer software this year: □ yes □ no

Name: Business title: Type of Business:

Address ( □ home □ work /Company name:)
Street/Suite#
City /State /Zipcode: Country

□ I liked this book! You may quote me by name in future
IDG Books Worldwide promotional materials.

My daytime phone number is
☐ YES!
Please keep me informed about IDG's World of Computer Knowledge. Send me the latest IDG Books catalog.
“If you’re an Intel assembly language programmer, this is the book you’ll want to keep handy for learning PowerPC programming from the ground up.” —Michael Abrash, Author of Zen of Code Optimization

Knowledge of the PowerPC architecture is the key to writing high-performance code like never before. Only PowerPC Programming for Intel Programmers provides an insider’s view of RISC architecture and PowerPC programming from the source — Motorola Systems Software Designer Kip McClanahan. Master PowerPC programming from the inside out with the code examples and comprehensive illustrations in this book — including a detailed view of the PowerPC family of microprocessors. With the Intel-specific programming tools included on the CD-ROM, you’ll enter the supercharged world of PowerPC programming in no time!

Inside, Find Everything You Need to Master the PowerPC Architecture:

- Find real-world programming examples in PowerPC assembly language
- Start programming today on your Intel-compatible PC using MetaWare’s tools
- Master techniques for writing efficient, optimized PowerPC code
- Get a top-level view of each PowerPC processor: illustrations show the components of each processor and how they interact

About the Author

Kip McClanahan is a systems software designer at Motorola and a member of the Motorola RISC Software Team. Kip has been programming in the PC industry for over 13 years and has written everything from firmware to device drivers to application programs.

Technical Review by Robert L. Hummel
Assembly Language Consultant & Author