Copyright Information

© 2007 Analog Devices, Inc., ALL RIGHTS RESERVED. This document may not be reproduced in any form without prior, express written consent from Analog Devices, Inc.

Printed in the USA.

Disclaimer

Analog Devices, Inc. reserves the right to change this product without prior notice. Information furnished by Analog Devices is believed to be accurate and reliable. However, no responsibility is assumed by Analog Devices for its use; nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under the patent rights of Analog Devices, Inc.

Trademark and Service Mark Notice

The Analog Devices logo, Blackfin, EZ-KIT Lite, SHARC, TigerSHARC, and VisualDSP++ are registered trademarks of Analog Devices, Inc.

All other brand and product names are trademarks or service marks of their respective owners.
CONTENTS

PREFACE

Purpose of This Manual .......................................................... xxv
Intended Audience ................................................................... xxv
Manual Contents ..................................................................... xxvi
What’s New in This Manual ....................................................... xxvii
Technical or Customer Support ................................................ xxviii
Supported Processors ............................................................ xxix
Product Information ............................................................... xxix
  MyAnalog.com .................................................................. xxx
Processor Product Information ............................................... xxx
Related Documents ............................................................... xxxi
Online Technical Documentation ............................................. xxxii
  Accessing Documentation From VisualDSP++ .................... xxxiii
  Accessing Documentation From Windows ......................... xxxiii
  Accessing Documentation From the Web ......................... xxxiv
## Contents

Printed Manuals ................................................................. xxxiv

VisualDSP++ Documentation Set ................................. xxxv

Hardware Tools Manuals ........................................... xxxv

Processor Manuals ....................................................... xxxv

Data Sheets ................................................................. xxxv

Conventions .................................................................... xxxvi

### INTRODUCTION

Core Architecture .......................................................... 1-1

Memory Architecture .................................................... 1-4

  Internal Memory .......................................................... 1-5

  External Memory .......................................................... 1-6

  I/O Memory Space ...................................................... 1-6

Event Handling ............................................................ 1-6

  Core Event Controller (CEC) ......................................... 1-8

  System Interrupt Controller (SIC) ............................... 1-8

Syntax Conventions ....................................................... 1-8

  Case Sensitivity .......................................................... 1-8

  Free Format ............................................................... 1-9

  Instruction Delimiting ............................................... 1-9

  Comments .................................................................. 1-10

Notation Conventions .................................................. 1-10

Behavior Conventions .................................................. 1-12
Contents

Using Multiplier Integer and Fractional Formats ....................  2-17
Rounding Multiplier Results ..................................................  2-19
  Unbiased Rounding ..........................................................  2-20
  Biased Rounding .............................................................  2-22
  Truncation .......................................................................  2-23
Special Rounding Instructions ...............................................  2-24
Using Computational Status .......................................................  2-24
ASTAT Register ..........................................................................  2-25
Arithmetic Logic Unit (ALU) .........................................................  2-26
  ALU Operations ...................................................................  2-26
    Single 16-Bit Operations ..................................................  2-27
    Dual 16-Bit Operations ....................................................  2-27
    Quad 16-Bit Operations ...................................................  2-28
    Single 32-Bit Operations ..................................................  2-29
    Dual 32-Bit Operations ....................................................  2-29
  ALU Instruction Summary ....................................................  2-30
  ALU Division Support Features .............................................  2-34
  Special SIMD Video ALU Operations ...................................  2-35
Multiply Accumulators (Multipliers) ...........................................  2-35
  Multiplier Operation .........................................................  2-36
    Placing Multiplier Results in Multiplier
      Accumulator Registers ....................................................  2-37
    Rounding or Saturating Multiplier Results .........................  2-37
    Saturating Multiplier Results on Overflow .........................  2-38
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier Instruction Summary</td>
<td>2-38</td>
</tr>
<tr>
<td>Multiplier Instruction Options</td>
<td>2-40</td>
</tr>
<tr>
<td>Multiplier Data Flow Details</td>
<td>2-42</td>
</tr>
<tr>
<td>Multiply Without Accumulate</td>
<td>2-44</td>
</tr>
<tr>
<td>Special 32-Bit Integer MAC Instruction</td>
<td>2-46</td>
</tr>
<tr>
<td>Dual MAC Operations</td>
<td>2-47</td>
</tr>
<tr>
<td>Barrel Shifter (Shifter)</td>
<td>2-48</td>
</tr>
<tr>
<td>Shifter Operations</td>
<td>2-48</td>
</tr>
<tr>
<td>Two-Operand Shifts</td>
<td>2-49</td>
</tr>
<tr>
<td>Immediate Shifts</td>
<td>2-49</td>
</tr>
<tr>
<td>Register Shifts</td>
<td>2-50</td>
</tr>
<tr>
<td>Three-Operand Shifts</td>
<td>2-50</td>
</tr>
<tr>
<td>Immediate Shifts</td>
<td>2-50</td>
</tr>
<tr>
<td>Register Shifts</td>
<td>2-51</td>
</tr>
<tr>
<td>Bit Test, Set, Clear, Toggle</td>
<td>2-52</td>
</tr>
<tr>
<td>Field Extract and Field Deposit</td>
<td>2-52</td>
</tr>
<tr>
<td>Shifter Instruction Summary</td>
<td>2-53</td>
</tr>
</tbody>
</table>

### OPERATING MODES AND STATES

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Mode</td>
<td>3-3</td>
</tr>
<tr>
<td>Protected Resources and Instructions</td>
<td>3-4</td>
</tr>
<tr>
<td>Protected Memory</td>
<td>3-5</td>
</tr>
<tr>
<td>Entering User Mode</td>
<td>3-5</td>
</tr>
<tr>
<td>Example Code to Enter User Mode Upon Reset</td>
<td>3-5</td>
</tr>
<tr>
<td>Return Instructions That Invoke User Mode</td>
<td>3-5</td>
</tr>
</tbody>
</table>
Contents

Supervisor Mode ................................................................. 3-7
Non-OS Environments ....................................................... 3-7
  Example Code for Supervisor Mode Coming
  Out of Reset ........................................................................ 3-8
Emulation Mode ................................................................. 3-9
Idle State .............................................................................. 3-9
  Example Code for Transition to Idle State ....................... 3-10
Reset State .......................................................................... 3-10
System Reset and Powerup .................................................. 3-12
  Hardware Reset ................................................................. 3-13
  SYSCR Register ............................................................... 3-14
  Software Resets and Watchdog Timer ............................. 3-14
  SWRST Register ............................................................... 3-15
  Core-Only Software Reset ................................................ 3-16
  Core and System Reset .................................................... 3-16

PROGRAM SEQUENCER

Introduction ......................................................................... 4-1
  Sequencer Related Registers ........................................... 4-5
Instruction Pipeline ............................................................. 4-7
Branches ............................................................................. 4-10
  Direct Short and Long Jumps .......................................... 4-11
  Direct Call ..................................................................... 4-12
  Indirect Branch and Call ................................................ 4-12
  PC-Relative Indirect Branch and Call .............................. 4-13
Contents

Subroutines ................................................................. 4-13

Stack Variables and Parameter Passing ....................... 4-15

Condition Code Flag ...................................................... 4-18

Conditional Branches ..................................................... 4-19

Conditional Register Move .......................................... 4-20

Branch Prediction ......................................................... 4-20

Hardware Loops ............................................................. 4-21

Two-Dimensional Loops ............................................... 4-24

Loop Unrolling ............................................................. 4-26

Saving and Resuming Loops ......................................... 4-27

Example Code for Using Hardware Loops in an ISR ....... 4-28

Events and Interrupts ..................................................... 4-29

System Interrupt Processing ......................................... 4-31

System Peripheral Interrupts ......................................... 4-33

SIC_IWR Register .......................................................... 4-34

SIC_ISR Register .......................................................... 4-35

SIC_IMASK Register ....................................................... 4-36

System Interrupt Assignment Registers (SIC_IARx) ....... 4-37

Core Event Controller Registers ..................................... 4-38

IMASK Register ............................................................ 4-38

ILAT Register ............................................................... 4-39

IPEND Register ............................................................ 4-40

Event Vector Table ......................................................... 4-41
Contents

Return Registers and Instructions ................................................. 4-42

Executing RTX, RTN, or RTE in a Lower Priority Event ................................................. 4-45

Emulation Interrupt .................................................................. 4-45

Reset Interrupt ........................................................................ 4-46

NMI (Nonmaskable Interrupt) .................................................... 4-46

Exceptions .................................................................................. 4-47

Hardware Error Interrupt ................................................................. 4-47

Core Timer Interrupt ................................................................. 4-47

General-purpose Interrupts (IVG7-IVG15) ........................................ 4-47

Interrupt Processing ...................................................................... 4-48

Global Enabling/Disabling of Interrupts ........................................ 4-48

Servicing Interrupts ................................................................. 4-48

Software Interrupts ...................................................................... 4-50

Nesting of Interrupts .................................................................... 4-51

Non-nested Interrupts ................................................................. 4-51

Nested Interrupts .......................................................................... 4-51

Example Prolog Code for Nested Interrupt Service Routine ................................. 4-53

Example Epilog Code for Nested Interrupt Service Routine ........................................ 4-54

Logging of Nested Interrupt Requests ........................................ 4-55

Self-Nesting of Core Interrupts ..................................................... 4-55
Contents

Additional Usability Issues ..................................................... 4-56
  Allocating the System Stack .................................................. 4-56
Latency in Servicing Events ...................................................... 4-56
Hardware Errors and Exception Handling ................................. 4-58
  SEQSTAT Register ................................................................ 4-59
  Hardware Error Interrupt ..................................................... 4-59
Exceptions ............................................................................. 4-61
  Exceptions While Executing an Exception Handler ................. 4-66
  Exceptions and the Pipeline ............................................... 4-67
  Deferring Exception Processing ......................................... 4-68
  Example Code for an Exception Handler ............................... 4-68
  Example Code for an Exception Routine ............................... 4-70

ADDRESS ARITHMETIC UNIT

  Addressing With the AAU ..................................................... 5-5
  Pointer Register File .......................................................... 5-6
  Frame and Stack Pointers ................................................... 5-6
  DAG Register Set .................................................................. 5-8
  Indexed Addressing With Index & Pointer Registers ............... 5-8
    Loads With Zero or Sign Extension ................................... 5-9
    Indexed Addressing With Immediate Offset ....................... 5-10
  Auto-increment and Auto-decrement Addressing .................. 5-10
  Pre-modify Stack Pointer Addressing ................................ 5-11
  Post-modify Addressing ..................................................... 5-11
Contents

Addressing Circular Buffers ................................................................. 5-12
Addressing With Bit-reversed Addresses .............................................. 5-15
Modifying DAG and Pointer Registers ............................................... 5-15
Memory Address Alignment .................................................................. 5-16
AAU Instruction Summary .................................................................... 5-19

MEMORY

Memory Architecture ............................................................................. 6-2
Overview of On-Chip Level 1 (L1) Memory ......................................... 6-2
Overview of Scratchpad Data SRAM .................................................... 6-4
Overview of On-Chip Level 2 (L2) Memory ......................................... 6-4
L1 Instruction Memory ........................................................................... 6-5
IMEM_CONTROL Register ................................................................. 6-5
L1 Instruction SRAM ............................................................................ 6-7
L1 Instruction Cache ............................................................................. 6-10
Cache Lines .......................................................................................... 6-10
Cache Hits and Misses .......................................................................... 6-13
Cache Line Fills .................................................................................... 6-14
Line Fill Buffer ....................................................................................... 6-15
Cache Line Replacement ...................................................................... 6-15
Instruction Cache Management ............................................................. 6-16
Instruction Cache Locking by Line ....................................................... 6-16
Instruction Cache Locking by Way ......................................................... 6-17
Instruction Cache Invalidation ............................................................... 6-18
Contents

Instruction Test Registers ................................................................. 6-19
  ITEST_COMMAND Register ......................................................... 6-21
  ITEST_DATA1 Register .......................................................... 6-22
  ITEST_DATA0 Register .......................................................... 6-23
L1 Data Memory .............................................................................. 6-24
  DMEM_CONTROL Register .................................................... 6-24
  L1 Data SRAM ........................................................................ 6-27
  L1 Data Cache ....................................................................... 6-29
    Example of Mapping Cacheable Address Space .................. 6-30
  Data Cache Access ............................................................. 6-33
  Cache Write Method .............................................................. 6-35
  IPRIO Register and Write Buffer Depth ................................. 6-35
  Data Cache Control Instructions ....................................... 6-37
  Data Cache Invalidation ...................................................... 6-38
Data Test Registers ........................................................................ 6-38
  DTEST_COMMAND Register .................................................. 6-39
  DTEST_DATA1 Register ......................................................... 6-41
  DTEST_DATA0 Register ......................................................... 6-42
On-chip Level 2 (L2) Memory ........................................................ 6-43
  On-chip L2 Bank Access ....................................................... 6-43
  Latency .................................................................................. 6-44
## Contents

- Memory Protection and Properties .............................................. 6-45
- Memory Management Unit ................................................... 6-45
- Memory Pages ....................................................................... 6-48
  - Memory Page Attributes ................................................... 6-48
- Page Descriptor Table ............................................................ 6-50
- CPLB Management ............................................................... 6-50
- MMU Application ................................................................. 6-52
- Examples of Protected Memory Regions ................................ 6-54
- ICPLB_DATAx Registers ...................................................... 6-55
- DCPLB_DATAx Registers ..................................................... 6-57
- DCPLB_ADDRx Registers .................................................... 6-59
- ICPLB_ADDRx Registers ..................................................... 6-60
- DCPLB_STATUS and ICPLB_STATUS Registers ................. 6-61
  - DCPLB_FAULT_ADDR and ICPLB_FAULT_ADDR Registers ............... 6-63
- Memory Transaction Model ........................................................ 6-65
- Load/Store Operation ................................................................. 6-66
  - Interlocked Pipeline ......................................................... 6-66
  - Ordering of Loads and Stores ................................................ 6-67
  - Synchronizing Instructions .................................................... 6-68
  - Speculative Load Execution ................................................... 6-69
  - Conditional Load Behavior ................................................... 6-70
- Working With Memory .............................................................. 6-71
  - Alignment ............................................................................. 6-71
  - Cache Coherency .................................................................. 6-71
## Contents

Atomic Operations ................................................................. 6-72  
Memory-mapped Registers .................................................... 6-72  
Core MMR Programming Code Example ......................... 6-73  
Terminology ............................................................................ 6-74  

### PROGRAM FLOW CONTROL

Jump .................................................................................. 7-2  
IF CC JUMP ........................................................................ 7-5  
Call .................................................................................... 7-8  
RTS, RTI, RTX, RTN, RTE (Return) .................................. 7-10  
LSETUP, LOOP .................................................................. 7-13  

### LOAD / STORE

Load Immediate .................................................................... 8-3  
Load Pointer Register ...................................................... 8-7  
Load Data Register .......................................................... 8-10  
Load Half-Word – Zero-Extended ........................................ 8-15  
Load Half-Word – Sign-Extended ........................................ 8-19  
Load High Data Register Half ............................................ 8-23  
Load Low Data Register Half ............................................. 8-27  
Load Byte – Zero-Extended ............................................... 8-31  
Load Byte – Sign-Extended ................................................ 8-34  
Store Pointer Register ..................................................... 8-37  
Store Data Register .......................................................... 8-40  
Store High Data Register Half .......................................... 8-45
Contents

Store Low Data Register Half ...................................................... 8-49
Store Byte ................................................................................... 8-54

MOVE

Move Register ............................................................................... 9-2
Move Conditional ........................................................................ 9-8
Move Half to Full Word – Zero-Extended ................................... 9-10
Move Half to Full Word – Sign-Extended .................................... 9-13
Move Register Half ..................................................................... 9-15
Move Byte – Zero-Extended ....................................................... 9-23
Move Byte – Sign-Extended ........................................................ 9-25

STACK CONTROL

--SP (Push) ................................................................................. 10-2
--SP (Push Multiple) ................................................................... 10-5
SP++ (Pop) .................................................................................. 10-8
SP++ (Pop Multiple) ..................................................................... 10-12
LINK, UNLINK ........................................................................... 10-17

CONTROL CODE BIT MANAGEMENT

Compare Data Register ............................................................... 11-2
Compare Pointer ......................................................................... 11-6
Compare Accumulator ............................................................... 11-9
Move CC ..................................................................................... 11-12
Negate CC .................................................................................. 11-15
LOGICAL OPERATIONS

& (AND) ....................................................................................... 12-2
~ (NOT One's-Complement) ...................................................... 12-4
| (OR) ...................................................................................... 12-6
^ (Exclusive-OR) ........................................................................ 12-8
BXORSHIFT, BXOR ................................................................ 12-10

BIT OPERATIONS

BITCLR ..................................................................................... 13-2
BITSET ...................................................................................... 13-4
BITTGL ..................................................................................... 13-6
BITTST ..................................................................................... 13-8
DEPOSIT ................................................................................... 13-10
EXTRACT ............................................................................... 13-16
BITMUX .................................................................................. 13-21
ONES (One’s-Population Count) .............................................. 13-26

SHIFT/ROTATE OPERATIONS

Add with Shift ............................................................................ 14-2
Shift with Add ............................................................................ 14-5
Arithmetic Shift .......................................................................... 14-7
Logical Shift ............................................................................... 14-14
ROT (Rotate) ........................................................................... 14-21
Contents

ARITHMETIC OPERATIONS

ABS ................................................................. 15-3
Add ............................................................... 15-6
Add/Subtract – Prescale Down ...................... 15-10
Add/Subtract – Prescale Up ......................... 15-13
Add Immediate ............................................. 15-16
DIVS, DIVQ (Divide Primitive) ................. 15-19
EXPADJ ....................................................... 15-26
MAX ............................................................ 15-30
MIN ............................................................. 15-32
Modify – Decrement ................................... 15-34
Modify – Increment ..................................... 15-37
Multiply 16-Bit Operands ......................... 15-43
Multiply 32-Bit Operands ......................... 15-51
Multiply and Multiply-Accumulate to Accumulator ........ 15-53
Multiply and Multiply-Accumulate to Half-Register ........ 15-58
Multiply and Multiply-Accumulate to Data Register ........ 15-67
Negate (Two's-Complement) ....................... 15-73
RND (Round to Half-Word) ......................... 15-77
Saturate ....................................................... 15-80
SIGNBITS .................................................. 15-83
Subtract ....................................................... 15-86
Subtract Immediate .................................... 15-90
EXTERNAL EVENT MANAGEMENT

Idle ................................................................. 16-3
Core Synchronize ............................................. 16-5
System Synchronize .......................................... 16-8
EMUEXCPT (Force Emulation) ......................... 16-11
Disable Interrupts ........................................... 16-13
Enable Interrupts ............................................. 16-15
RAISE (Force Interrupt / Reset) ......................... 16-17
EXCPT (Force Exception) ................................. 16-20
Test and Set Byte (Atomic) ............................... 16-22
No Op ............................................................ 16-25

CACHE CONTROL

PREFETCH ..................................................... 17-3
FLUSH .......................................................... 17-5
FLUSHINV ...................................................... 17-7
Iflush ............................................................ 17-9

VIDEO PIXEL OPERATIONS

ALIGN8, ALIGN16, ALIGN24 .............................. 18-3
DISALGNEXCPT ............................................. 18-6
BYTEOP3P (Dual 16-Bit Add / Clip) ..................... 18-8
Dual 16-Bit Accumulator Extraction with Addition ...... 18-13
BYTEOP16P (Quad 8-Bit Add) ............................ 18-15
BYTEOP1P (Quad 8-Bit Average – Byte) ............... 18-19
Contents

16-Bit Instructions .............................................................. 20-6
Examples ............................................................................... 20-8

DEBUG

Watchpoint Unit ............................................................... 21-1
  Instruction Watchpoints ................................................ 21-4
  WPIAn Registers ......................................................... 21-5
  WPIACNTn Registers ................................................... 21-6
  WPIACTL Register ..................................................... 21-7
  Data Address Watchpoints ......................................... 21-10
  WPDAAn Registers ..................................................... 21-10
  WPDACNTn Registers ................................................ 21-11
  WPDACTL Register ................................................... 21-12
  WPSTAT Register ..................................................... 21-14
Trace Unit ........................................................................ 21-15
  TBUFCTL Register ................................................... 21-16
  TBUFSTAT Register .................................................. 21-17
  TBUF Register .......................................................... 21-18
    Code to Recreate the Execution Trace in Memory ....... 21-18
Performance Monitoring Unit ............................................. 21-19
  PFCNTRn Registers ................................................... 21-20
  PFCTL Register ....................................................... 21-20
  Event Monitor Table ................................................... 21-21
## Contents

Cycle Counter ................................................................. 21-23  
CYCLES and CYCLES2 Registers ................................. 21-24  
SYSCFG Register ......................................................... 21-26  
Product Identification Register ................................. 21-27  
DSPID Register ......................................................... 21-27  

### ADSP-BF535 CONSIDERATIONS

- ADSP-BF535 Operating Modes and States .................... A-1  
- ADSP-BF535 Flags ............................................... A-2  

### CORE MMR ASSIGNMENTS

- L1 Data Memory Controller Registers ....................... B-1  
- L1 Instruction Memory Controller Registers ............. B-4  
- Interrupt Controller Registers ............................... B-6  
- Debug, MP, and Emulation Unit Registers .................. B-7  
- Trace Unit Registers ........................................... B-8  
- Watchpoint and Patch Registers ............................... B-8  
- Performance Monitor Registers ............................... B-9  

### INSTRUCTION OPCODES

- Introduction .......................................................... C-1  
- Appendix Organization ........................................... C-1  
- Glossary ............................................................... C-2  
  - Register Names ................................................... C-2  
  - Functional Units ............................................... C-3  

xxii ADSP-BF53x/BF56x Blackfin Processor Programming Reference
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notation Conventions</td>
<td>C-4</td>
</tr>
<tr>
<td>Arithmetic Status Flags</td>
<td>C-6</td>
</tr>
<tr>
<td>Core Register Encoding Map</td>
<td>C-8</td>
</tr>
<tr>
<td>Opcode Representation</td>
<td>C-8</td>
</tr>
<tr>
<td>Opcode Bit Terminology</td>
<td>C-10</td>
</tr>
<tr>
<td>Undefined Opcodes</td>
<td>C-10</td>
</tr>
<tr>
<td>Holes In Opcode Ranges</td>
<td>C-10</td>
</tr>
<tr>
<td>Opcode Representation In Listings, Memory Dumps</td>
<td>C-11</td>
</tr>
<tr>
<td>Program Flow Control Instructions</td>
<td>C-13</td>
</tr>
<tr>
<td>Load / Store Instructions</td>
<td>C-16</td>
</tr>
<tr>
<td>Move Instructions</td>
<td>C-28</td>
</tr>
<tr>
<td>Stack Control Instructions</td>
<td>C-37</td>
</tr>
<tr>
<td>Control Code Bit Management Instructions</td>
<td>C-39</td>
</tr>
<tr>
<td>Logical Operations Instructions</td>
<td>C-43</td>
</tr>
<tr>
<td>Bit Operations Instructions</td>
<td>C-44</td>
</tr>
<tr>
<td>Shift / Rotate Operations Instructions</td>
<td>C-46</td>
</tr>
<tr>
<td>Arithmetic Operations Instructions</td>
<td>C-55</td>
</tr>
<tr>
<td>External Event Management Instructions</td>
<td>C-99</td>
</tr>
<tr>
<td>Cache Control Instructions</td>
<td>C-101</td>
</tr>
<tr>
<td>Video Pixel Operations Instructions</td>
<td>C-102</td>
</tr>
<tr>
<td>Vector Operations Instructions</td>
<td>C-107</td>
</tr>
<tr>
<td>Instructions Listed By Operation Code</td>
<td>C-139</td>
</tr>
<tr>
<td>16-Bit Opcode Instructions</td>
<td>C-140</td>
</tr>
<tr>
<td>32-Bit Opcode Instructions</td>
<td>C-154</td>
</tr>
</tbody>
</table>
Contents

NUMERIC FORMATS

Unsigned or Signed: Two’s-complement Format ......................... D-1
Integer or Fractional Data Formats ............................................ D-1
Binary Multiplication ............................................................. D-5
  Fractional Mode And Integer Mode ...................................... D-6
  Block Floating-Point Format .............................................. D-6

INDEX
PREFACE

Thank you for purchasing and developing systems using an Analog Devices Blackfin® processor.

Purpose of This Manual

The ADSP-BF53x/BF56x Blackfin Processor Programming Reference contains information about the processor architecture and assembly language for Blackfin processors. This manual is applicable to single-core and dual-core Blackfin processors. In many ways, they are identical. The exceptions to this are noted in Chapter 6, “Memory.”

The manual provides information on how assembly instructions execute on the Blackfin processor’s architecture along with reference information about processor operations.

Intended Audience

The primary audience for this manual is programmers who are familiar with Analog Devices Blackfin processors. This manual assumes that the audience has a working knowledge of the appropriate Blackfin architecture and instruction set. Programmers who are unfamiliar with Analog Devices processors can use this manual but should supplement it with other texts (such as hardware reference manuals and data sheets that describe your target architecture).
Manual Contents

The manual consists of:

- Chapter 1, “Introduction”
  This chapter provides a general description of the instruction syntax and notation conventions.

- Chapter 2, “Computational Units”
  Describes the arithmetic/logic units (ALUs), multiplier/accumulator units (MACs), shifter, and the set of video ALUs. The chapter also discusses data formats, data types, and register files.

- Chapter 3, “Operating Modes and States”
  Describes the operating modes of the processor. The chapter also describes Idle state and Reset state.

- Chapter 4, “Program Sequencer”
  Describes the operation of the program sequencer, which controls program flow by providing the address of the next instruction to be executed. The chapter also discusses loops, subroutines, jumps, interrupts, and exceptions.

- Chapter 5, “Address Arithmetic Unit”
  Describes the Address Arithmetic Unit (AAU), including Data Address Generators (DAGs), addressing modes, how to modify DAG and Pointer registers, memory address alignment, and DAG instructions.

- Chapter 6, “Memory”
  Describes L1 memories. In particular, details their memory architecture, memory model, memory transaction model, and memory-mapped registers (MMRs). Discusses the instruction, data, and scratchpad memory, which are part of the Blackfin processor core.
Preface


- Chapter 20, “Issuing Parallel Instructions” Provides a description of parallel instruction operations and shows how to use parallel instruction syntax.

- Appendix A, “ADSP-BF535 Considerations” Provides a description of the status flag bits for the ADSP-BF535 processor only.

- Appendix B, “Core MMR Assignments” Lists the core memory-mapped registers, their addresses, and cross-references to text.

- Appendix C, “Instruction Opcodes” Identifies operation codes (opcodes) for instructions. Use this chapter to learn how to construct opcodes.

- Appendix D, “Numeric Formats” Describes various aspects of the 16-bit data format. The chapter also describes how to implement a block floating-point format in software.

What’s New in This Manual

This is the third edition (Revision 1.2) of the ADSP-BF53x/BF56x Blackfin Processor Programming Reference. Changes to this book from the second edition (Revision 1.1) include corrections of typographic errors and reported document errata.
Technical or Customer Support

You can reach Analog Devices, Inc. Customer Support in the following ways:

- Visit the Embedded Processing and DSP products Web site at http://www.analog.com/processors/manuals
- E-mail tools questions to processor.tools.support@analog.com
- E-mail processor questions to
  processor.support@analog.com (World wide support)
  processor.europe@analog.com (Europe support)
  processor.china@analog.com (China support)
- Phone questions to 1-800-ANALOGD
- Contact your Analog Devices, Inc. local sales office or authorized distributor
- Send questions by mail to:
  Analog Devices, Inc.
  One Technology Way
  P.O. Box 9106
  Norwood, MA 02062-9106
  USA
Supported Processors

The following is the list of Analog Devices, Inc. processors supported in VisualDSP++®.

**Blackfin (ADSP-BFxxx) Processors**

The name *Blackfin* refers to a family of 16-bit, embedded processors. VisualDSP++ currently supports the following Blackfin families:

- ADSP-BF53x, ADSP-BF54x, and ADSP-BF56x

**SHARC® (ADSP-21xxx) Processors**

The name *SHARC* refers to a family of high-performance, 32-bit, floating-point processors that can be used in speech, sound, graphics, and imaging applications. VisualDSP++ currently supports the following SHARC families:

- ADSP-2106x, ADSP-2116x, ADSP-2126x, ADSP-2136x, and ADSP-2137x

**TigerSHARC® (ADSP-TSxxx) Processors**

The name *TigerSHARC* refers to a family of floating-point and fixed-point [8-bit, 16-bit, and 32-bit] processors. VisualDSP++ currently supports the following TigerSHARC families:

- ADSP-TS101 and ADSP-TS20x

**Product Information**

You can obtain product information from the Analog Devices Web site, from the product CD-ROM, or from the printed publications (manuals).
Product Information

Analog Devices is online at www.analog.com. Our Web site provides information about a broad range of products—analog integrated circuits, amplifiers, converters, and digital signal processors.

MyAnalog.com

MyAnalog.com is a free feature of the Analog Devices Web site that allows customization of a Web page to display only the latest information on products you are interested in. You can also choose to receive weekly e-mail notifications containing updates to the Web pages that meet your interests. MyAnalog.com provides access to books, application notes, data sheets, code examples, and more.

Registration

Visit www.myanalog.com to sign up. Click Register to use MyAnalog.com. Registration takes about five minutes and serves as a means to select the information you want to receive.

If you are already a registered user, just log on. Your user name is your e-mail address.

Processor Product Information

For information on embedded processors and DSPs, visit our Web site at www.analog.com/processors, which provides access to technical publications, data sheets, application notes, product overviews, and product announcements.
Preface

You may also obtain additional information about Analog Devices and its products in any of the following ways.

- E-mail questions or requests for information to
  processor.support@analog.com (World wide support)
  processor.europe@analog.com (Europe support)
  processor.china@analog.com (China support)

- Fax questions or requests for information to
  1-781-461-3010 (North America)
  +49-89-76903-157 (Europe)

- Access the FTP Web site at
  ftp ftp.analog.com (or ftp 137.71.25.69)
  ftp://ftp.analog.com

Related Documents

The following publications that describe the ADSP-BF53x/BF56x processors (and related processors) can be ordered from any Analog Devices sales office:

- ADSP-BF533 Blackfin Processor Hardware Reference
- ADSP-BF535 Blackfin Processor Hardware Reference
- ADSP-BF561 Blackfin Processor Hardware Reference
- ADSP-BF537 Blackfin Processor Hardware Reference
- ADSP-BF538/ADSP-BF539 Blackfin Processor Hardware Reference
- ADSP-BF531/ADSP-BF532/ADSP-BF533 Blackfin Embedded Processor Data Sheet
- ADSP-BF534 Blackfin Embedded Processor Data Sheet
- ADSP-BF535 Blackfin Embedded Processor Data Sheet
Product Information

- ADSP-BF536/ADSP-BF537 Blackfin Embedded Processor Data Sheet
- ADSP-BF538 Blackfin Embedded Processor Data Sheet
- ADSP-BF539 Blackfin Embedded Processor Data Sheet

For information on product related development software and Analog Devices processors, see these publications:

- VisualDSP++ User's Guide
- VisualDSP++ C/C++ Compiler and Library Manual for Blackfin Processors
- VisualDSP++ Assembler and Preprocessor Manual
- VisualDSP++ Linker and Utilities Manual

Visit the Technical Library Web site to access all processor and tools manuals and data sheets:

http://www.analog.com/processors/manuals

Online Technical Documentation

Online documentation comprises the VisualDSP++ Help system, software tools manuals, hardware tools manuals, processor manuals, the Dinkum Abridged C++ library, and Flexible License Manager (FlexLM) network license manager software documentation. You can easily search across the entire VisualDSP++ documentation set for any topic of interest. For easy printing, supplementary .PDF files of most manuals are also provided.
Each documentation file type is described as follows.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.CHM</td>
<td>Help system files and manuals in Help format</td>
</tr>
<tr>
<td>.HTM or .HTML</td>
<td>Dinkum Abridged C++ library and FlexLM network license manager software documenta-</td>
</tr>
<tr>
<td></td>
<td>tion. Viewing and printing the .HTML files requires a browser, such as Internet Explor-</td>
</tr>
<tr>
<td></td>
<td>er 4.0 (or higher).</td>
</tr>
<tr>
<td>.PDF</td>
<td>VisualDSP++ and processor manuals in Portable Documentation Format (PDF).</td>
</tr>
<tr>
<td></td>
<td>Viewing and printing the .PDF files requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher).</td>
</tr>
</tbody>
</table>

If documentation is not installed on your system as part of the software installation, you can add it from the VisualDSP++ CD-ROM at any time by running the Tools installation. Access the online documentation from the VisualDSP++ environment, Windows® Explorer, or the Analog Devices Web site.

**Accessing Documentation From VisualDSP++**

From the VisualDSP++ environment:

- Access VisualDSP++ online Help from the Help menu’s **Contents**, **Search**, and **Index** commands.

- Open online Help from context-sensitive user interface items (tool- bar buttons, menu commands, and windows).

**Accessing Documentation From Windows**

In addition to any shortcuts you may have constructed, there are many ways to open VisualDSP++ online Help or the supplementary documentation from Windows.
Help system files (.CHM) are located in the Help folder, and .PDF files are located in the Docs folder of your VisualDSP++ installation CD-ROM. The Docs folder also contains the Dinkum Abridged C++ library and the FlexLM network license manager software documentation.

Using Windows Explorer

- Double-click the vdsp-help.chm file, which is the master Help system, to access all the other .CHM files.
- Double-click any file that is part of the VisualDSP++ documentation set.

Using the Windows Start Button

- Access VisualDSP++ online Help by clicking the Start button and choosing Programs, Analog Devices, VisualDSP++, and VisualDSP++ Documentation.
- Access the .PDF files by clicking the Start button and choosing Programs, Analog Devices, VisualDSP++, Documentation for Printing, and the name of the book.

Accessing Documentation From the Web

Download manuals at the following Web site:
http://www.analog.com/processors/manuals

Select a processor family and book title. Download archive (.ZIP) files, one for each manual. Use any archive management software, such as WinZip, to decompress downloaded files.

Printed Manuals

For general questions regarding literature ordering, call the Literature Center at 1-800-ANALOGD (1-800-262-5643) and follow the prompts.
Preface

VisualDSP++ Documentation Set

To purchase VisualDSP++ manuals, call 1-603-883-2430. The manuals may be purchased only as a kit.

If you do not have an account with Analog Devices, you are referred to Analog Devices distributors. For information on our distributors, log onto http://www.analog.com/salesdir.

Hardware Tools Manuals

To purchase EZ-KIT Lite® and In-Circuit Emulator (ICE) manuals, call 1-603-883-2430. The manuals may be ordered by title or by product number located on the back cover of each manual.

Processor Manuals

Hardware reference and instruction set reference manuals may be ordered through the Literature Center at 1-800-ANALOGD (1-800-262-5643), or downloaded from the Analog Devices Web site. Manuals may be ordered by title or by product number located on the back cover of each manual.

Data Sheets

All data sheets (preliminary and production) may be downloaded from the Analog Devices Web site. Only production (final) data sheets (Rev. 0, A, B, C, and so on) can be obtained from the Literature Center at 1-800-ANALOGD (1-800-262-5643); they also can be downloaded from the Web site.

To have a data sheet faxed to you, call the Analog Devices Faxback System at 1-800-446-6212. Follow the prompts and a list of data sheet code numbers will be faxed to you. If the data sheet you want is not listed, check for it on the Web site.
## Conventions

Text conventions used in this manual are identified and described as follows.

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Close command (File menu)</td>
<td>Titles in reference sections indicate the location of an item within the VisualDSP++ environment's menu system. For example, the Close command appears on the File menu.</td>
</tr>
<tr>
<td>this</td>
<td>that</td>
</tr>
<tr>
<td>this</td>
<td>that</td>
</tr>
<tr>
<td>[(()(S</td>
<td>SU)))]]</td>
</tr>
<tr>
<td>.SECTION</td>
<td>Commands, directives, keywords, and feature names are in text with letter gothic font.</td>
</tr>
<tr>
<td>filename</td>
<td>Non-keyword placeholders appear in text with italic style format.</td>
</tr>
<tr>
<td>SWRST Software Reset register</td>
<td>Register names appear in UPPERCASE and a special typeface. The descriptive names of registers are in mixed case and regular typeface.</td>
</tr>
<tr>
<td>TMR0E, RESET</td>
<td>Pin names appear in UPPERCASE and a special typeface. Active low signals appear with an OVERBAR.</td>
</tr>
<tr>
<td>DRx, SIC.IMASKx, I[3:0], SMS[3:0]</td>
<td>Register, bit, and pin names in the text may refer to groups of registers or pins: A lowercase x in a register name (DRx) indicates a set of registers (for example, DR2, DR1, and DR0) for those processors with more than one register of that name. For processors with only a single register of that name, the x can be disregarded (for example, SIC.IMASKx refers to SIC.IMASK in the ADSP-BF533 processor, and to SIC.IMASK0 and SIC.IMASK1 in the ADSP-BF561). A colon between numbers within brackets indicates a range of registers or pins (for example, I[3:0] indicates I3, I2, I1, and I0; SMS[3:0] indicates SMS3, SMS2, SMS1, and SMS0).</td>
</tr>
</tbody>
</table>
### Preface

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFBCD CBA9</td>
<td>Hexadecimal numbers use the 0x prefix and are typically shown with a space between the upper four and lower four digits.</td>
</tr>
<tr>
<td>b#1010 0101</td>
<td>Binary numbers use the b# prefix and are typically shown with a space between each four digit group.</td>
</tr>
</tbody>
</table>

**Note:** For correct operation, ...

A Note: provides supplementary information on a related topic. In the online version of this book, the word Note appears instead of this symbol.

**Caution:** Incorrect device operation may result if ...

Caution: Device damage may result if ...

A Caution: identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word Caution appears instead of this symbol.

**Warning:** Injury to device users may result if ...

A Warning: identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for devices users. In the online version of this book, the word Warning appears instead of this symbol.

Additional conventions, which apply only to specific chapters, may appear throughout this document.
Conventions
1 INTRODUCTION

This ADSP-BF53x/BF56x Blackfin Processor Programming Reference provides details on the assembly language instructions used by the Micro Signal Architecture (MSA) core developed jointly by Analog Devices, Inc. and Intel Corporation. This manual is applicable to all ADSP-BF53x and ADSP-BF56x processor derivatives. With the exception of the first-generation ADSP-BF535 processor, all devices provide an identical core architecture and instruction set. Specifics of the ADSP-BF535 processor are highlighted where applicable and are summarized in Appendix A. Dual-core derivatives and derivatives with on-chip L2 memory have slightly different system interfaces. Differences and commonalities at a global level are discussed in Chapter 6, "Memory." For a full description of the system architecture beyond the Blackfin core, refer to the specific Hardware Reference Manual for your derivative. This section points out some of the conventions used in this document.

The Blackfin processor combines a dual MAC signal processing engine, an orthogonal RISC-like microprocessor instruction set, flexible Single Instruction, Multiple Data (SIMD) capabilities, and multimedia features into a single instruction set architecture.

Core Architecture

The Blackfin processor core contains two 16-bit multipliers, two 40-bit accumulators, two 40-bit arithmetic logic units (ALUs), four 8-bit video ALUs, and a 40-bit shifter, shown in Figure 1-1. The process 8-, 16-, or 32-bit data from the register file.
The compute register file contains eight 32-bit registers. When performing compute operations on 16-bit operand data, the register file operates as 16 independent 16-bit registers. All operands for compute operations come from the multiported register file and instruction constant fields.
Introduction

Each MAC can perform a 16- by 16-bit multiply per cycle, with accumulation to a 40-bit result. Signed and unsigned formats, rounding, and saturation are supported.

The ALUs perform a traditional set of arithmetic and logical operations on 16-bit or 32-bit data. Many special instructions are included to accelerate various signal processing tasks. These include bit operations such as field extract and population count, modulo $2^{32}$ multiply, divide primitives, saturation and rounding, and sign/exponent detection. The set of video instructions include byte alignment and packing operations, 16-bit and 8-bit adds with clipping, 8-bit average operations, and 8-bit subtract/absolute value/accumulate (SAA) operations. Also provided are the compare/select and vector search instructions. For some instructions, two 16-bit ALU operations can be performed simultaneously on register pairs (a 16-bit high half and 16-bit low half of a compute register). By also using the second ALU, quad 16-bit operations are possible.

The 40-bit shifter can deposit data and perform shifting, rotating, normalization, and extraction operations.

A program sequencer controls the instruction execution flow, including instruction alignment and decoding. For program flow control, the sequencer supports PC-relative and indirect conditional jumps (with static branch prediction) and subroutine calls. Hardware is provided to support zero-overhead looping. The architecture is fully interlocked, meaning there are no visible pipeline effects when executing instructions with data dependencies.

The address arithmetic unit provides two addresses for simultaneous dual fetches from memory. It contains a multiported register file consisting of four sets of 32-bit Index, Modify, Length, and Base registers (for circular buffering) and eight additional 32-bit pointer registers (for C-style indexed stack manipulation).
Blackfin processors support a modified Harvard architecture in combination with a hierarchical memory structure. Level 1 (L1) memories typically operate at the full processor speed with little or no latency. At the L1 level, the instruction memory holds instructions only. The two data memories hold data, and a dedicated scratchpad data memory stores stack and local variable information.

In addition, multiple L1 memory blocks are provided, which may be configured as a mix of SRAM and cache. The Memory Management Unit (MMU) provides memory protection for individual tasks that may be operating on the core and may protect system registers from unintended access.

The architecture provides three modes of operation: User, Supervisor, and Emulation. User mode has restricted access to a subset of system resources, thus providing a protected software environment. Supervisor and Emulation modes have unrestricted access to the system and core resources.

The Blackfin processor instruction set is optimized so that 16-bit opcodes represent the most frequently used instructions. Complex DSP instructions are encoded into 32-bit opcodes as multifunction instructions. Blackfin products support a limited multi-issue capability, where a 32-bit instruction can be issued in parallel with two 16-bit instructions. This allows the programmer to use many of the core resources in a single instruction cycle.

The Blackfin processor assembly language uses an algebraic syntax. The architecture is optimized for use with the C compiler.

Memory Architecture

The Blackfin processor architecture structures memory as a single, unified 4G byte address space using 32-bit addresses, regardless of the specific Blackfin product. All resources, including internal memory, external memory, and I/O control registers, occupy separate sections of this
common address space. The memory portions of this address space are arranged in a hierarchical structure to provide a good cost/performance balance of some very fast, low latency on-chip memory as cache or SRAM, and larger, lower cost and lower performance off-chip memory systems.

The L1 memory system is the primary highest performance memory available to the core. The off-chip memory system, accessed through the External Bus Interface Unit (EBIU), provides expansion with SDRAM, flash memory, and SRAM, optionally accessing up to 132M bytes of physical memory.

The memory DMA controller provides high bandwidth data movement capability. It can perform block transfers of code or data between the internal memory and the external memory spaces.

Internal Memory

At a minimum, each Blackfin processors has three blocks of on-chip memory that provide high bandwidth access to the core:

- L1 instruction memory, consisting of SRAM and a 4-way set-associative cache. This memory is accessed at full processor speed.
- L1 data memory, consisting of SRAM and/or a 2-way set-associative cache. This memory block is accessed at full processor speed.
- L1 scratchpad RAM, which runs at the same speed as the L1 memories but is only accessible as data SRAM and cannot be configured as cache memory.

In addition, some Blackfin processors share a low latency, high bandwidth on-chip Level 2 (L2) memory. It forms an on-chip memory hierarchy with L1 memory and provides much more capacity than L1 memory, but the latency is higher. The on-chip L2 memory is SRAM and cannot be configured as cache. On-chip L2 memory is capable of storing both instructions and data and is accessible by both cores.
External Memory

External (off-chip) memory is accessed via the External Bus Interface Unit (EBIU). This 16-bit interface provides a glueless connection to a bank of synchronous DRAM (SDRAM) and as many as four banks of asynchronous memory devices including flash memory, EPROM, ROM, SRAM, and memory-mapped I/O devices.

The PC133-compliant SDRAM controller can be programmed to interface to up to 512M bytes of SDRAM (certain products have SDRAM up to 128M bytes).

The asynchronous memory controller can be programmed to control up to four banks of devices. Each bank occupies a 1M byte segment regardless of the size of the devices used, so that these banks are only contiguous if each is fully populated with 1M byte of memory.

I/O Memory Space

Blackfin processors do not define a separate I/O space. All resources are mapped through the flat 32-bit address space. Control registers for on-chip I/O devices are mapped into memory-mapped registers (MMRs) at addresses near the top of the 4G byte address space. These are separated into two smaller blocks: one contains the control MMRs for all core functions and the other contains the registers needed for setup and control of the on-chip peripherals outside of the core. The MMRs are accessible only in Supervisor mode. They appear as reserved space to on-chip peripherals.

Event Handling

The event controller on the Blackfin processor handles all asynchronous and synchronous events to the processor. The processor event handling supports both nesting and prioritization. Nesting allows multiple event service routines to be active simultaneously. Prioritization ensures that
servicing a higher priority event takes precedence over servicing a lower priority event. The controller provides support for five different types of events:

- **Emulation** – Causes the processor to enter Emulation mode, allowing command and control of the processor via the JTAG interface.

- **Reset** – Resets the processor.

- **Nonmaskable Interrupt (NMI)** – The software watchdog timer or the NMI input signal to the processor generates this event. The NMI event is frequently used as a power-down indicator to initiate an orderly shutdown of the system.

- **Exceptions** – Synchronous to program flow. That is, the exception is taken before the instruction is allowed to complete. Conditions such as data alignment violations and undefined instructions cause exceptions.

- **Interrupts** – Asynchronous to program flow. These are caused by input pins, timers, and other peripherals.

Each event has an associated register to hold the return address and an associated return-from-event instruction. When an event is triggered, the state of the processor is saved on the supervisor stack.

The processor event controller consists of two stages: the Core Event Controller (CEC) and the System Interrupt Controller (SIC). The CEC works with the SIC to prioritize and control all system events. Conceptually, interrupts from the peripherals arrive at the SIC and are routed directly into the general-purpose interrupts of the CEC.
Syntax Conventions

Core Event Controller (CEC)

The Core Event Controller supports nine general-purpose interrupts (IVG15–7), in addition to the dedicated interrupt and exception events. Of these general-purpose interrupts, the two lowest priority interrupts (IVG15–14) are recommended to be reserved for software interrupt handlers, leaving seven prioritized interrupt inputs to support peripherals.

System Interrupt Controller (SIC)

The System Interrupt Controller provides the mapping and routing of events from the many peripheral interrupt sources to the prioritized general-purpose interrupt inputs of the CEC. Although the processor provides a default mapping, the user can alter the mappings and priorities of interrupt events by writing the appropriate values into the Interrupt Assignment Registers (IAR).

Syntax Conventions

The Blackfin processor instruction set supports several syntactic conventions that appear throughout this document. Those conventions are given below.

Case Sensitivity

The instruction syntax is case insensitive. Upper and lower case letters can be used and intermixed arbitrarily.

The assembler treats register names and instruction keywords in a case-insensitive manner. User identifiers are case sensitive. Thus, R3.l, R3.L, r3.l, r3.L are all valid, equivalent input to the assembler.
Introduction

This manual shows register names and instruction keywords in examples using lower case. Otherwise, in explanations and descriptions, this manual uses upper case to help the register names and keywords stand out among text.

Free Format

Assembler input is free format, and may appear anywhere on the line. One instruction may extend across multiple lines, or more than one instruction may appear on the same line. White space (space, tab, comments, or newline) may appear anywhere between tokens. A token must not have embedded spaces. Tokens include numbers, register names, keywords, user identifiers, and also some multicharacter special symbols like “+=”, “/*”, or “||”.

Instruction Delimiting

A semicolon must terminate every instruction. Several instructions can be placed together on a single line at the programmer’s discretion, provided each instruction ends with a semicolon.

Each complete instruction must end with a semicolon. Sometimes, a complete instruction will consist of more than one operation. There are two cases where this occurs.

- Two general operations are combined. Normally a comma separates the different parts, as in
  \[a0 = r3.h * r2.l, a1 = r3.l * r2.h;\]

- A general instruction is combined with one or two memory references for joint issue. The latter portions are set off by a “||” token. For example,
  \[a0 = r3.h * r2.l || r1 = [p3++] || r4 = [i2++] ;\]
Comments

The assembler supports various kinds of comments, including the following.

- **End of line**: A double forward slash token (“//”) indicates the beginning of a comment that concludes at the next newline character.

- **General comment**: A general comment begins with the token “/*” and ends with “*/”. It may contain any characters and extend over multiple lines.

Comments are not recursive; if the assembler sees a “/*” within a general comment, it issues an assembler warning. A comment functions as white space.

Notation Conventions

This manual and the assembler use the following conventions.

- **Register names**: Are alphabetical, followed by a number in cases where there are more than one register in a logical group. Thus, examples include `ASTAT`, `FP`, `R3`, and `M2`.

- **Register names**: Are reserved and may not be used as program identifiers.

- **Some operations** (such as “Move Register”) require a register pair. Register pairs are always Data Registers and are denoted using a colon, for example, `R3:2`. The larger number must be written first. Note that the hardware supports only odd-even pairs, for example, `R7:6, R5:4, R3:2, and R1:0`. 

1-10    ADSP-BF53x/BF56x Blackfin Processor Programming Reference
Introduction

- Some instructions (such as "--SP (Push Multiple)") require a group of adjacent registers. Adjacent registers are denoted in syntax by the range enclosed in parentheses and separated by a colon, for example, \((R7:3)\). Again, the larger number appears first.

- Portions of a particular register may be individually specified. This is written in syntax with a dot (".") following the register name, then a letter denoting the desired portion. For 32-bit registers, "\(H\)" denotes the most-significant ("High") portion, "\(L\)" denotes the least-significant portion. The subdivisions of the 40-bit registers are described later.

Register names are reserved and may not be used as program identifiers.

This manual uses the following conventions.

- When there is a choice of any one register within a register group, this manual shows the register set using an en-dash ("–"). For example, "\(R7–0\)" in text means that any one of the eight data registers \((R7, R6, R5, R4, R3, R2, R1, \text{or} R0)\) can be used in syntax.

- Immediate values are designated as "\(imm\)" with the following modifiers.
  - "\(imm\)" indicates a signed value; for example, \(imm7\).
  - The "\(u\)" prefix indicates an unsigned value; for example, \(uimm4\).
  - The decimal number indicates how many bits the value can include; for example, \(imm5\) is a 5-bit value.
  - Any alignment requirements are designated by an optional "\(m\)" suffix followed by a number; for example, \(uimm16m2\) is an unsigned, 16-bit integer that must be an even number, and \(imm7m4\) is a signed, 7-bit integer that must be a multiple of 4.
Behavior Conventions

- PC-relative, signed values are designated as “pcrel” with the following modifiers:
  - the decimal number indicates how many bits the value can include; for example, pcrel5 is a 5-bit value.
  - any alignment requirements are designated by an optional “m” suffix followed by a number; for example, pcrel13m2 is a 13-bit integer that must be an even number.

- Loop PC-relative, signed values are designated as “lppcrel” with the following modifiers:
  - the decimal number indicates how many bits the value can include; for example, lppcrel5 is a 5-bit value.
  - any alignment requirements are designated by an optional “m” suffix followed by a number; for example, lppcrel11m2 is a 11-bit integer that must be an even number.

Behavior Conventions

All operations that produce a result in an Accumulator saturate to a 40-bit quantity unless noted otherwise. See “Saturation” on page 1-17 for a description of saturation behavior.
Glossary

The following terms appear throughout this document. Without trying to explain the Blackfin processor, here are the terms used with their definitions. See the *Blackfin Processor Hardware Reference* for your specific product for more details on the architecture.

Register Names

The architecture includes the registers shown in Table 1-1.

Table 1-1. Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accumulators</td>
<td>The set of 40-bit registers A1 and A0 that normally contain data that is being manipulated. Each Accumulator can be accessed in five ways: as one 40-bit register, as one 32-bit register (designated as A1.W or A0.W), as two 16-bit registers similar to Data Registers (designated as A1.H, A1.L, A0.H, or A0.L) and as one 8-bit register (designated A1.X or A0.X) for the bits that extend beyond bit 31.</td>
</tr>
<tr>
<td>Data Registers</td>
<td>The set of 32-bit registers (R0, R1, R2, R3, R4, R5, R6, and R7) that normally contain data for manipulation. Abbreviated D-register or Dreg. Data Registers can be accessed as 32-bit registers, or optionally as two independent 16-bit registers. The least significant 16 bits of each register is called the “low” half and is designated with “.L” following the register name. The most significant 16 bit is called the “high” half and is designated with “.H” following the name. Example: R7.L, r2.h, r4.L, R0.h.</td>
</tr>
<tr>
<td>Pointer Registers</td>
<td>The set of 32-bit registers (P0, P1, P2, P3, P4, P5, including SP and FP) that normally contain byte addresses of data structures. Accessed only as a 32-bit register. Abbreviated P-register or Preg. Example: p2, p5, fp, sp.</td>
</tr>
<tr>
<td>Stack Pointer</td>
<td>SP; contains the 32-bit address of the last occupied byte location in the stack. The stack grows by decrementing the Stack Pointer. A subset of the Pointer Registers.</td>
</tr>
<tr>
<td>Frame Pointer</td>
<td>FP; contains the 32-bit address of the previous Frame Pointer in the stack, located at the top of a frame. A subset of the Pointer Registers.</td>
</tr>
<tr>
<td>Loop Top</td>
<td>LT0 and LT1; contains 32-bit address of the top of a zero overhead loop.</td>
</tr>
</tbody>
</table>
Table 1-1. Registers (Cont’d)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop Count</td>
<td>LC0 and LC1; contains 32-bit counter of the zero overhead loop executions.</td>
</tr>
<tr>
<td>Loop Bottom</td>
<td>LB0 and LB1; contains 32-bit address of the bottom of a zero overhead loop.</td>
</tr>
<tr>
<td>Index Register</td>
<td>The set of 32-bit registers I0, I1, I2, I3 that normally contain byte addresses of data structures. Abbreviated I-register or Ireg.</td>
</tr>
<tr>
<td>Modify Registers</td>
<td>The set of 32-bit registers M0, M1, M2, M3 that normally contain offset values that are added or subtracted to one of the Index Registers. Abbreviated as Mreg.</td>
</tr>
<tr>
<td>Length Registers</td>
<td>The set of 32-bit registers L0, L1, L2, L3 that normally contain the length (in bytes) of the circular buffer. Abbreviated as Lreg. Clear Lreg to disable circular addressing for the corresponding Ireg. Example: Clear L3 to disable circular addressing for I3.</td>
</tr>
<tr>
<td>Base Registers</td>
<td>The set of 32-bit registers B0, B1, B2, B3 that normally contain the base address (in bytes) of the circular buffer. Abbreviated as Breg.</td>
</tr>
</tbody>
</table>

**Functional Units**

The architecture includes the three processor sections shown in Table 1-2.

Table 1-2. Processor Sections

<table>
<thead>
<tr>
<th>Processor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Address Generator (DAG)</td>
<td>Calculates the effective address for indirect and indexed memory accesses. Consists of two sections–DAG0 and DAG1.</td>
</tr>
<tr>
<td>Multiply and Accumulate Unit (MAC)</td>
<td>Performs the arithmetic functions on data. Consists of two sections (MAC0 and MAC1)–each associated with an Accumulator (A0 and A1, respectively).</td>
</tr>
<tr>
<td>Arithmetic Logical Unit (ALU)</td>
<td>Performs arithmetic computations and binary shifts on data. Operates on the Data Registers and Accumulators. Consists of two units (ALU0 and ALU1), each associated with an Accumulator (A0 and A1, respectively). Each ALU operates in conjunction with a Multiply and Accumulate Unit.</td>
</tr>
</tbody>
</table>
Introduction

Arithmetic Status Flags

The MSA includes 12 arithmetic status flags that indicate specific results of a prior operation. These flags reside in the Arithmetic Status (ASTAT) Register. A summary of the flags appears below. All flags are active high. Instructions regarding P-registers, I-registers, L-registers, M-registers, or B-registers do not affect flags.

See the Blackfin Processor Hardware Reference for your specific product for more details on the architecture.

Table 1-3. Arithmetic Status Flag Summary

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC0</td>
<td>Carry (ALU0)</td>
</tr>
<tr>
<td>AC0_COPY</td>
<td>Carry (ALU0), copy</td>
</tr>
<tr>
<td>AC1</td>
<td>Carry (ALU1)</td>
</tr>
<tr>
<td>AN</td>
<td>Negative</td>
</tr>
<tr>
<td>AQ</td>
<td>Quotient</td>
</tr>
<tr>
<td>AV0</td>
<td>Accumulator 0 Overflow</td>
</tr>
<tr>
<td>AVS0</td>
<td>Accumulator 0 Sticky Overflow</td>
</tr>
<tr>
<td></td>
<td>Set when AV0 is set, but remains set until explicitly cleared by user code.</td>
</tr>
<tr>
<td>AV1</td>
<td>Accumulator 1 Overflow</td>
</tr>
<tr>
<td>AVS1</td>
<td>Accumulator 1 Sticky Overflow</td>
</tr>
<tr>
<td></td>
<td>Set when AV1 is set, but remains set until explicitly cleared by user code.</td>
</tr>
<tr>
<td>AZ</td>
<td>Zero</td>
</tr>
<tr>
<td>CC</td>
<td>Control Code bit</td>
</tr>
<tr>
<td></td>
<td>Multipurpose flag set, cleared and tested by specific instructions.</td>
</tr>
<tr>
<td>V</td>
<td>Overflow for Data Register results</td>
</tr>
<tr>
<td>V_COPY</td>
<td>Overflow for Data Register results. copy</td>
</tr>
<tr>
<td>VS</td>
<td>Sticky Overflow for Data Register results</td>
</tr>
<tr>
<td></td>
<td>Set when V is set, but remains set until explicitly cleared by user code.</td>
</tr>
</tbody>
</table>
Glossary

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Fractional Convention

Fractional numbers include subinteger components less than ±1. Whereas decimal fractions appear to the right of a decimal point, binary fractions appear to the right of a binary point.

In DSP instructions that assume placement of a binary point, for example in computing sign bits for normalization or for alignment purposes, the binary point convention depends on the size of the register being used as shown in Table 1-4 and Figure 1-2 on page 1-17.

This processor does not represent fractional values in 8-bit registers.

Table 1-4. Fractional Conventions

<table>
<thead>
<tr>
<th>Registers Size</th>
<th>Format</th>
<th>Notation</th>
<th>Sign Bit</th>
<th>Extension Bits</th>
<th>Fractional Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>40-bit registers</td>
<td>Signed Fractional</td>
<td>9.31</td>
<td>1</td>
<td>8</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>Unsigned Fractional</td>
<td>8.32</td>
<td>0</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>32-bit registers</td>
<td>Signed Fractional</td>
<td>1.31</td>
<td>1</td>
<td>0</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>Unsigned Fractional</td>
<td>0.32</td>
<td>0</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>16-bit registers</td>
<td>Signed Fractional</td>
<td>1.15</td>
<td>1</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Unsigned Fractional</td>
<td>0.16</td>
<td>0</td>
<td>0</td>
<td>16</td>
</tr>
</tbody>
</table>
Introduction

Saturation

When the result of an arithmetic operation exceeds the range of the destination register, important information can be lost.

Saturation is a technique used to contain the quantity within the values that the destination register can represent. When a value is computed that exceeds the capacity of the destination register, then the value written to the register is the largest value that the register can hold with the same sign as the original.

- If an operation would otherwise cause a positive value to overflow and become negative, instead, saturation limits the result to the maximum positive value for the size register being used.
- Conversely, if an operation would otherwise cause a negative value to overflow and become positive, saturation limits the result to the maximum negative value for the register size.

The overflow arithmetic flag is never set by an operation that enforces saturation.
The maximum positive value in a 16-bit register is 0x7FFF. The maximum negative value is 0x8000. For a signed two’s-complement 1.15 fractional notation, the allowable range is –1 through (1–2–15).

The maximum positive value in a 32-bit register is 0x7FFFFF. The maximum negative value is 0x8000 0000. For a signed two’s-complement fractional data in 1.31 format, the range of values that the register can hold are –1 through (1–2–31).

The maximum positive value in a 40-bit register is 0x7FFF FFFF. The maximum negative value is 0x80 0000 0000. For a signed two’s-complement 9.31 fractional notation, the range of values that can be represented is –256 through (256–2–31).

For example, if a 16-bit register containing 0x1000 (decimal integer +4096) was shifted left 3 places without saturation, it would overflow to 0x8000 (decimal –32,768). With saturation, however, a left shift of 3 or more places would always produce the largest positive 16-bit number, 0x7FFF (decimal +32,767).

Another common example is copying the lower half of a 32-bit register into a 16-bit register. If the 32-bit register contains 0xFEED 0ACE and the lower half of this negative number is copied into a 16-bit register without saturation, the result is 0x0ACE, a positive number. But if saturation is enforced, the 16-bit result maintains its negative sign and becomes 0x8000.

The MSA implements 40-bit saturation for all arithmetic operations that write an Accumulator destination except as noted in the individual instruction descriptions when an optional 32-bit saturation mode can constrain a 40-bit Accumulator to the 32-bit register range. The MSA performs 32-bit saturation for 32-bit register destinations only as noted in the instruction descriptions.

Overflow is the alternative to saturation. The number is allowed to simply exceed its bounds and lose its most significant bit(s); only the lowest (least-significant) portion of the number can be retained. Overflow can
occur when a 40-bit value is written to a 32-bit destination. If there was any useful information in the upper 8 bits of the 40-bit value, then information is lost in the process. Some processor instructions report overflow conditions in the arithmetic flags, as noted in the instruction descriptions. The arithmetic flags reside in the Arithmetic Status (ASTAT) Register. See the Blackfin Processor Hardware Reference for your specific product for more details on the ASTAT Register.

Rounding and Truncating

Rounding is a means of reducing the precision of a number by removing a lower-order range of bits from that number’s representation and possibly modifying the remaining portion of the number to more accurately represent its former value. For example, the original number will have N bits of precision, whereas the new number will have only M bits of precision (where N>M), so N-M bits of precision are removed from the number in the process of rounding.

The round-to-nearest method returns the closest number to the original. By convention, an original number lying exactly halfway between two numbers always rounds up to the larger of the two. For example, when rounding the 3-bit, two’s-complement fraction 0.25 (binary 0.01) to the nearest 2-bit two’s-complement fraction, this method returns 0.5 (binary 0.1). The original fraction lies exactly midway between 0.5 and 0.0 (binary 0.0), so this method rounds up. Because it always rounds up, this method is called biased rounding.

The convergent rounding method also returns the closest number to the original. However, in cases where the original number lies exactly halfway between two numbers, this method returns the nearest even number, the one containing an LSB of 0. So for the example above, the result would be 0.0, since that is the even numbered choice of 0.5 and 0.0. Since it rounds up and down based on the surrounding values, this method is called unbiased rounding.
Some instructions for this processor support biased and unbiased rounding. The `RND_MOD` bit in the Arithmetic Status (`ASTAT`) Register determines which mode is used. See the *Blackfin Processor Hardware Reference* for your specific product for more details on the `ASTAT` Register.

Another common way to reduce the significant bits representing a number is to simply mask off the N-M lower bits. This process is known as *truncation* and results in a relatively large bias.

*Figure 1-3* shows other examples of rounding and truncation methods.

---

**Figure 1-3. 8-Bit Number Reduced to 4 Bits of Precision**
Automatic Circular Addressing

The Blackfin processor provides an optional circular (or “modulo”) addressing feature that increments an Index Register ($I_{reg}$) through a predefined address range, then automatically resets the $I_{reg}$ to repeat that range. This feature improves input/output loop performance by eliminating the need to manually reinitialize the address index pointer each time. Circular addressing is useful, for instance, when repetitively loading or storing a string of fixed-sized data blocks.

The circular buffer contents must meet the following conditions:

- The maximum length of a circular buffer (that is, the value held in any $L$ register) must be an unsigned number with magnitude less than $2^{31}$.
- The magnitude of the modifier should be less than the length of the circular buffer.
- The initial location of the pointer I should be within the circular buffer defined by the base $B$ and length $L$.

If any of these conditions is not satisfied, then processor behavior is not specified.

There are two elements of automatic circular addressing:

- Indexed address instructions
- Four sets of circular addressing buffer registers composed of one each $I_{reg}$, $B_{reg}$, and $L_{reg}$ (i.e., $I_0/B_0/L_0$, $I_1/B_1/L_1$, $I_2/B_2/L_2$, and $I_3/B_3/L_3$)

To qualify for circular addressing, the indexed address instruction must explicitly modify an Index Register. Some indexed address instructions use a Modify Register ($M_{reg}$) to increment the $I_{reg}$ value. In that case, any $M_{reg}$ can be used to increment any $I_{reg}$. The $I_{reg}$ used in the instruction specifies which of the four circular buffer sets to use.
Glossary

The circular buffer registers define the length \((L_{\text{reg}})\) of the data block in bytes and the base \((B_{\text{reg}})\) address to reinitialize the \(I_{\text{reg}}\).

Some instructions modify an Index Register without using it for addressing; for example, the Add Immediate and Modify – Decrement instructions. Such instructions are still affected by circular addressing, if enabled.

Disable circular addressing for an \(I_{\text{reg}}\) by clearing the \(L_{\text{reg}}\) that corresponds to the \(I_{\text{reg}}\) used in the instruction. For example, clear \(L_{2}\) to disable circular addressing for register \(I_{2}\). Any nonzero value in an \(L_{\text{reg}}\) enables circular addressing for its corresponding buffer registers.

See the Blackfin Processor Hardware Reference for your specific product for more details on circular addressing capabilities and operation.
2 COMPUTATIONAL UNITS

The processor’s computational units perform numeric processing for DSP and general control algorithms. The six computational units are two arithmetic/logic units (ALUs), two multiplier/accumulator (multiplier) units, a shifter, and a set of video ALUs. These units get data from registers in the Data Register File. Computational instructions for these units provide fixed-point operations, and each computational instruction can execute every cycle.

The computational units handle different types of operations. The ALUs perform arithmetic and logic operations. The multipliers perform multiplication and execute multiply/add and multiply/subtract operations. The shifter executes logical shifts and arithmetic shifts and performs bit packing and extraction. The video ALUs perform Single Instruction, Multiple Data (SIMD) logical operations on specific 8-bit data operands.

Data moving in and out of the computational units goes through the Data Register File, which consists of eight registers, each 32 bits wide. In operations requiring 16-bit operands, the registers are paired, providing sixteen possible 16-bit registers.

The processor’s assembly language provides access to the Data Register File. The syntax lets programs move data to and from these registers and specify a computation’s data format at the same time.

Figure 2-1 provides a graphical guide to the other topics in this chapter. An examination of each computational unit provides details about its operation and is followed by a summary of computational instructions. Studying the details of the computational units, register files, and data
buses leads to a better understanding of proper data flow for computations. Next, details about the processor's advanced parallelism reveal how to take advantage of multifunction instructions.

**Figure 2-1** shows the relationship between the Data Register File and the computational units—multipliers, ALUs, and shifter.

Single function multiplier, ALU, and shifter instructions have unrestricted access to the data registers in the Data Register File. Multifunction operations may have restrictions that are described in the section for that particular operation.

Two additional registers, A0 and A1, provide 40-bit accumulator results. These registers are dedicated to the ALUs and are used primarily for multiply-and-accumulate functions.

The traditional modes of arithmetic operations, such as fractional and integer, are specified directly in the instruction. Rounding modes are set from the ASTAT register, which also records status and conditions for the results of the computational operations.
Computational Units

Figure 2-1. Processor Core Architecture

ADSP-BF53x/BF56x Blackfin Processor Programming Reference 2-3
Using Data Formats

ADSP-BF53x/56x processors are primarily 16-bit, fixed-point machines. Most operations assume a two’s-complement number representation, while others assume unsigned numbers or simple binary strings. Other instructions support 32-bit integer arithmetic, with further special features supporting 8-bit arithmetic and block floating point. For detailed information about each number format, see Appendix D, “Numeric Formats.”

In the ADSP-BF53x/56x processor family arithmetic, signed numbers are always in two’s-complement format. These processors do not use signed-magnitude, one’s-complement, binary-coded decimal (BCD), or excess-n formats.

Binary String

The binary string format is the least complex binary notation; in it, 16 bits are treated as a bit pattern. Examples of computations using this format are the logical operations NOT, AND, OR, XOR. These ALU operations treat their operands as binary strings with no provision for sign bit or binary point placement.

Unsigned

Unsigned binary numbers may be thought of as positive and having nearly twice the magnitude of a signed number of the same length. The processor treats the least significant words of multiple precision numbers as unsigned numbers.
Computational Units

Signed Numbers: Two’s-Complement

In ADSP-BF53x/56x processor arithmetic, the word *signed* refers to two’s-complement numbers. Most ADSP-BF53x/56x processor family operations presume or support two’s-complement arithmetic.

Fractional Representation: 1.15

ADSP-BF53x processor arithmetic is optimized for numerical values in a fractional binary format denoted by 1.15 (“one dot fifteen”). In the 1.15 format, 1 sign bit (the Most Significant Bit (MSB)) and 15 fractional bits represent values from –1 to 0.999969.

Figure 2-2 shows the bit weighting for 1.15 numbers as well as some examples of 1.15 numbers and their decimal equivalents.

<table>
<thead>
<tr>
<th>1.15 NUMBER (HEXADECIMAL)</th>
<th>DECIMAL EQUIVALENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001</td>
<td>0.000031</td>
</tr>
<tr>
<td>0x7FFF</td>
<td>0.999969</td>
</tr>
<tr>
<td>0xFFFF</td>
<td>-0.000031</td>
</tr>
<tr>
<td>0x8000</td>
<td>-1.000000</td>
</tr>
</tbody>
</table>

Figure 2-2. Bit Weighting for 1.15 Numbers
The processor’s computational units have three definitive register groups—a Data Register File, a Pointer Register File, and set of Data Address Generation (DAG) registers.

- The Data Register File receives operands from the data buses for the computational units and stores computational results.
- The Pointer Register File has pointers for addressing operations.
- The DAG registers are dedicated registers that manage zero-overhead circular buffers for DSP operations.

For more information on Pointer and DAG registers, see Chapter 5, “Address Arithmetic Unit.”

In the processor, a word is 32 bits long; H denotes the high order 16 bits of a 32-bit register; L denotes the low order 16 bits of a 32-bit register; W denotes the low order 32 bits of a 40-bit accumulator register; and X denotes the high order 8 bits. For example, \( A_0.W \) contains the lower 32 bits of the 40-bit \( A_0 \) register; \( A_0.L \) contains the lower 16 bits of \( A_0.W \), and \( A_0.H \) contains the upper 16 bits of \( A_0.W \).
Computational Units

Data Register File

The Data Register File consists of eight registers, each 32 bits wide. Each register may be viewed as a pair of independent 16-bit registers. Each is denoted as the low half or high half. Thus the 32-bit register \( R_0 \) may be regarded as two independent register halves, \( R_0.L \) and \( R_0.H \).

For example, these instructions represent a 32-bit and a 16-bit operation:

\[
\begin{align*}
R2 &= R1 + R2; \quad /* 32-bit addition */ \\
R2.L &= R1.H \times R0.L; \quad /* 16-bit multiplication */
\end{align*}
\]

Figure 2-3. Register Files
Register Files

Three separate buses (two load, one store) connect the Register File to the L1 data memory, each bus being 32 bits wide. Transfers between the Data Register File and the data memory can move up to two 32-bit words of valid data in each cycle. Often, these represent four 16-bit words.

Accumulator Registers

In addition to the Data Register File, the processor has two dedicated, 40-bit accumulator registers, called $A0$ and $A1$. Each can be referred to as its 16-bit low half ($A_n.L$) or high half ($A_n.H$) plus its 8-bit extension ($A_n.X$). Each can also be referred to as a 32-bit register ($A_n.W$) consisting of the lower 32 bits, or as a complete 40-bit result register ($A_n$).

These examples illustrate this convention:

- $A0 = A1$: /* 40-bit move */
- $A0.H = R5.H$: /* 16-bit move */
- $R6.H = A0.X$: /* read 8-bit value and sign extend to 16 bits */

Figure 2-4. 40-Bit Accumulator Registers
Register File Instruction Summary

Table 2-1 lists the register file instructions. In Table 2-1, note the meaning of these symbols:

- **Allreg** denotes: 
  \[ R[7:0], P[5:0], SP, FP, I[3:0], M[3:0], \]
  \[ RETX, RETN, RETE, LC[1:0], LT[1:0], LB[1:0], USP, SEQSTAT, \]
  \[ SYSCFG, CYCLES, and CYCLES2. \]
- **An** denotes either ALU Result register \( A0 \) or \( A1 \).
- **Dreg** denotes any Data Register File register.
- **Sysreg** denotes the system registers: `ASTAT`, `SEQSTAT`, `SYSCFG`, `RETI`, `RETX`, `RETN`, `RETE`, or `RETS`, `LC[1:0]`, `LT[1:0]`, `LB[1:0]`, `USP`, `SEQSTAT`, `SYSCFG`, `CYCLES`, and `CYCLES2`.
- **Preg** denotes any Pointer register, \( FP \), or \( SP \) register.
- **Dreg_even** denotes \( R0,R2,R4, \) or \( R6 \).
- **Dreg_odd** denotes \( R1,R3,R5, \) or \( R7 \).
- **DPreg** denotes any Data Register File register or any Pointer register, \( FP \), or \( SP \) register.
- **Dreg_lo** denotes the lower 16 bits of any Data Register File register.
- **Dreg_hi** denotes the upper 16 bits of any Data Register File register.
- **An.L** denotes the lower 16 bits of Accumulator \( A0.W \) or \( A1.W \).
- **An.H** denotes the upper 16 bits of Accumulator \( A0.W \) or \( A1.W \).
- **Dreg_byte** denotes the low order 8 bits of each Data register.
Register Files

- Option (X) denotes sign extended.
- Option (Z) denotes zero extended.
- * Indicates the flag may be set or cleared, depending on the result of the instruction.
- ** Indicates the flag is cleared.
- – Indicates no effect.

Table 2-1. Register File Instruction Summary

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ASTAT Status Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AZ</td>
</tr>
<tr>
<td>allreg = allreg ;</td>
<td>1</td>
</tr>
<tr>
<td>An = An ;</td>
<td>1</td>
</tr>
<tr>
<td>An = Dreg ;</td>
<td>1</td>
</tr>
<tr>
<td>Dreg_even = A0 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_odd = A1 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_even = A0, Dreg_odd = A1 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_odd = A1, Dreg_even = A0 ;</td>
<td>*</td>
</tr>
<tr>
<td>IF CC DReg = DReg ;</td>
<td>1</td>
</tr>
<tr>
<td>IF ! CC DReg = DReg ;</td>
<td>1</td>
</tr>
<tr>
<td>Dreg = Dreg_lo (Z) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = Dreg_lo (X) ;</td>
<td>*</td>
</tr>
<tr>
<td>An.X = Dreg_lo ;</td>
<td>1</td>
</tr>
<tr>
<td>Dreg_lo = An.X ;</td>
<td>1</td>
</tr>
<tr>
<td>An.L = Dreg_lo ;</td>
<td>1</td>
</tr>
</tbody>
</table>
**Computational Units**

Data Types

The processor supports 32-bit words, 16-bit half words, and bytes. The 32- and 16-bit words can be integer or fractional, but bytes are always integers. Integer data types can be signed or unsigned, but fractional data types are always signed.

Table 2-3 illustrates the formats for data that resides in memory, in the register file, and in the accumulators. In the table, the letter $d$ represents one bit, and the letter $s$ represents one signed bit.

---

**Table 2-1. Register File Instruction Summary (Cont’d)**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ASTAT Status Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AZ</td>
</tr>
<tr>
<td>An.H = Dreg_hi ;</td>
<td>–</td>
</tr>
<tr>
<td>Dreg_lo = A0 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_hi = A1 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_hi = A1 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_lo = A0 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = Dreg_byte (Z) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = Dreg_byte (X) ;</td>
<td>*</td>
</tr>
</tbody>
</table>

1 Warning: Not all register combinations are allowed. For details, see the functional description of the Move Register instruction in Chapter 9, "Move."
Data Types

Some instructions manipulate data in the registers by sign-extending or zero-extending the data to 32 bits:

- Instructions zero-extend unsigned data
- Instructions sign-extend signed 16-bit half words and 8-bit bytes

Other instructions manipulate data as 32-bit numbers. In addition, two 16-bit half words or four 8-bit bytes can be manipulated as 32-bit values.

In Table 2-2, note the meaning of these symbols:

- $s =$ sign bit(s)
- $d =$ data bit(s)
- "." = decimal point by convention; however, a decimal point does not literally appear in the number.
- Italics denotes data from a source other than adjacent bits.
Computational Units

Table 2-2. Data Formats

<table>
<thead>
<tr>
<th>Format</th>
<th>Representation in Memory</th>
<th>Representation in 32-bit Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>32.0 Unsigned Word</td>
<td>dddd dddd dddd dddd dddd dddd dddd dddd</td>
<td>dddd dddd dddd dddd dddd dddd dddd dddd</td>
</tr>
<tr>
<td>32.0 Signed Word</td>
<td>sddd dddd dddd dddd dddd dddd dddd dddd</td>
<td>sddd dddd dddd dddd dddd dddd dddd dddd</td>
</tr>
<tr>
<td>16.0 Unsigned Half Word</td>
<td>dddd dddd dddd dddd</td>
<td>0000 0000 0000 0000 dddd dddd dddd dddd</td>
</tr>
<tr>
<td>16.0 Signed Half Word</td>
<td>sddd dddd dddd dddd</td>
<td>ssss ssss ssss ssss ssss dddd dddd dddd</td>
</tr>
<tr>
<td>8.0 Unsigned Byte</td>
<td>dddd dddd</td>
<td>0000 0000 0000 0000 0000 0000 dddd dddd</td>
</tr>
<tr>
<td>8.0 Signed Byte</td>
<td>sddd dddd</td>
<td>ssss ssss ssss ssss ssss ssss ssss ssss dddd dddd</td>
</tr>
<tr>
<td>1.15 Signed Fraction</td>
<td>s.ddd dddd dddd dddd</td>
<td>ssss ssss ssss ssss s.ddd dddd dddd dddd</td>
</tr>
<tr>
<td>1.31 Signed Fraction</td>
<td>s.ddd dddd dddd dddd dddd dddd dddd dddd</td>
<td>s.ddd dddd dddd dddd dddd dddd dddd dddd</td>
</tr>
<tr>
<td>Packed 8.0 Unsigned Byte</td>
<td>dddd dddd dddd dddd dddd dddd dddd dddd dddd</td>
<td>dddd dddd dddd dddd dddd dddd dddd dddd dddd</td>
</tr>
<tr>
<td>Packed 1.15 Signed Fraction</td>
<td>s.ddd dddd dddd dddd s.ddd dddd dddd dddd dddd</td>
<td>s.ddd dddd dddd dddd s.ddd dddd dddd dddd dddd</td>
</tr>
</tbody>
</table>

Endianess

Both internal and external memory are accessed in little endian byte order. For more information, see “Memory Transaction Model” on page 6-65.
Data Types

ALU Data Types

Operations on each ALU treat operands and results as either 16- or 32-bit binary strings, except the signed division primitive (DIVS). ALU result status bits treat the results as signed, indicating status with the overflow flags (AV0, AV1) and the negative flag (AN). Each ALU has its own sticky overflow flag, AV0S and AV1S. Once set, these bits remain set until cleared by writing directly to the ASTAT register. An additional V flag is set or cleared depending on the transfer of the result from both accumulators to the register file. Furthermore, the sticky VS bit is set with the V bit and remains set until cleared.

The logic of the overflow bits (V, VS, AV0, AV0S, AV1, AV1S) is based on two’s-complement arithmetic. A bit or set of bits is set if the Most Significant Bit (MSB) changes in a manner not predicted by the signs of the operands and the nature of the operation. For example, adding two positive numbers must generate a positive result; a change in the sign bit signifies an overflow and sets AVn, the corresponding overflow flags. Adding a negative and a positive number may result in either a negative or positive result, but cannot cause an overflow.

The logic of the carry bits (AC0, AC1) is based on unsigned magnitude arithmetic. The bit is set if a carry is generated from bit 16 (the MSB). The carry bits (AC0, AC1) are most useful for the lower word portions of a multiword operation.

ALU results generate status information. For more information about using ALU status, see “ALU Instruction Summary” on page 2-30.

Multiplier Data Types

Each multiplier produces results that are binary strings. The inputs are interpreted according to the information given in the instruction itself (whether it is signed multiplied by signed, unsigned multiplied by
unsigned, a mixture, or a rounding operation). The 32-bit result from the multipliers is assumed to be signed; it is sign-extended across the full 40-bit width of the $A0$ or $A1$ registers.

The processor supports two modes of format adjustment: the fractional mode for fractional operands (1.15 format with 1 sign bit and 15 fractional bits) and the integer mode for integer operands (16.0 format).

When the processor multiplies two 1.15 operands, the result is a 2.30 (2 sign bits and 30 fractional bits) number. In the fractional mode, the multiplier automatically shifts the multiplier product left one bit before transferring the result to the multiplier result register ($A0$, $A1$). This shift of the redundant sign bit causes the multiplier result to be in 1.31 format, which can be rounded to 1.15 format. The resulting format appears in Figure 2-5 on page 2-18.

In the integer mode, the left shift does not occur. For example, if the operands are in the 16.0 format, the 32-bit multiplier result would be in 32.0 format. A left shift is not needed and would change the numerical representation. This result format appears in Figure 2-6 on page 2-19.

Multiplier results generate status information when they update accumulators or when they are transferred to a destination register in the register file. For more information, see "Multiplier Instruction Summary" on page 2-38.

**Shifter Data Types**

Many operations in the shifter are explicitly geared to signed (two’s-complement) or unsigned values—logical shifts assume unsigned magnitude or binary string values, and arithmetic shifts assume two’s-complement values.

The exponent logic assumes two’s-complement numbers. The exponent logic supports block floating point, which is also based on two’s-complement fractions.
Data Types

Shifter results generate status information. For more information about using shifter status, see “Shifter Instruction Summary” on page 2-53.

Arithmetic Formats Summary

Table 2-3, Table 2-4, Table 2-5, and Table 2-6 summarize some of the arithmetic characteristics of computational operations.

Table 2-3. ALU Arithmetic Formats

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand Formats</th>
<th>Result Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>Signed or unsigned</td>
<td>Interpret flags</td>
</tr>
<tr>
<td>Subtraction</td>
<td>Signed or unsigned</td>
<td>Interpret flags</td>
</tr>
<tr>
<td>Logical</td>
<td>Binary string</td>
<td>Same as operands</td>
</tr>
<tr>
<td>Division</td>
<td>Explicitly signed or</td>
<td>Same as operands</td>
</tr>
</tbody>
</table>

Table 2-4. Multiplier Fractional Modes Formats

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand Formats</th>
<th>Result Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplication</td>
<td>1.15 explicitly signed or unsigned</td>
<td>2.30 shifted to 1.31</td>
</tr>
<tr>
<td>Multiplication/Addition</td>
<td>1.15 explicitly signed or unsigned</td>
<td>2.30 shifted to 1.31</td>
</tr>
<tr>
<td>Multiplication/Subtraction</td>
<td>1.15 explicitly signed or unsigned</td>
<td>2.30 shifted to 1.31</td>
</tr>
</tbody>
</table>

Table 2-5. Multiplier Arithmetic Integer Modes Formats

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand Formats</th>
<th>Result Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplication</td>
<td>16.0 explicitly signed or unsigned</td>
<td>32.0 not shifted</td>
</tr>
</tbody>
</table>
Using Multiplier Integer and Fractional Formats

For multiply-and-accumulate functions, the processor provides two choices—fractional arithmetic for fractional numbers (1.15) and integer arithmetic for integers (16.0).

For fractional arithmetic, the 32-bit product output is format adjusted—sign-extended and shifted one bit to the left—before being added to accumulator A0 or A1. For example, bit 31 of the product lines up with bit 32 of A0 (which is bit 0 of A0.X), and bit 0 of the product lines up with bit 1 of A0 (which is bit 1 of A0.W). The Least Significant Bit (LSB) is zero filled. The fractional multiplier result format appears in Figure 2-5.

For integer arithmetic, the 32-bit product register is not shifted before being added to A0 or A1. Figure 2-6 shows the integer mode result placement.
Data Types

With either fractional or integer operations, the multiplier output product is fed into a 40-bit adder/subtractor which adds or subtracts the new product with the current contents of the A₀ or A₁ register to produce the final 40-bit result.

Figure 2-5. Fractional Multiplier Results Format
Rounding Multiplier Results

On many multiplier operations, the processor supports multiplier results rounding (RND option). Rounding is a means of reducing the precision of a number by removing a lower order range of bits from that number’s representation and possibly modifying the remaining portion of the number to more accurately represent its former value. For example, the original number will have N bits of precision, whereas the new number will have only M bits of precision (where N>M). The process of rounding, then, removes N – M bits of precision from the number.

The RND_MOD bit in the ASTAT register determines whether the RND option provides biased or unbiased rounding. For unbiased rounding, set RND_MOD bit = 0. For biased rounding, set RND_MOD bit = 1.

For most algorithms, unbiased rounding is preferred.
Data Types

Unbiased Rounding

The convergent rounding method returns the number closest to the original. In cases where the original number lies exactly halfway between two numbers, this method returns the nearest even number, the one containing an LSB of 0. For example, when rounding the 3-bit, two’s-complement fraction 0.25 (binary 0.01) to the nearest 2-bit, two’s-complement fraction, the result would be 0.0, because that is the even-numbered choice of 0.5 and 0.0. Since it rounds up and down based on the surrounding values, this method is called unbiased rounding.

Unbiased rounding uses the ALU’s capability of rounding the 40-bit result at the boundary between bit 15 and bit 16. Rounding can be specified as part of the instruction code. When rounding is selected, the output register contains the rounded 16-bit result; the accumulator is never rounded.

The accumulator uses an unbiased rounding scheme. The conventional method of biased rounding adds a 1 into bit position 15 of the adder chain. This method causes a net positive bias because the midway value (when \( A0.L/A1.L = 0x8000 \)) is always rounded upward.

The accumulator eliminates this bias by forcing bit 16 in the result output to 0 when it detects this midway point. Forcing bit 16 to 0 has the effect of rounding odd \( A0.L/A1.L \) values upward and even values downward, yielding a large sample bias of 0, assuming uniformly distributed values.

The following examples use \( x \) to represent any bit pattern (not all zeros). The example in Figure 2-7 shows a typical rounding operation for \( A0 \); the example also applies for \( A1 \).
Computational Units

The compensation to avoid net bias becomes visible when all lower 15 bits are 0 and bit 15 is 1 (the midpoint value) as shown in Figure 2-7.

In Figure 2-8, A0 bit 16 is forced to 0. This algorithm is employed on every rounding operation, but is evident only when the bit patterns shown in the lower 16 bits of the next example are present.

Figure 2-7. Typical Unbiased Multiplier Rounding
Biased Rounding

The round-to-nearest method also returns the number closest to the original. However, by convention, an original number lying exactly halfway between two numbers always rounds up to the larger of the two. For example, when rounding the 3-bit, two’s-complement fraction 0.25 (binary 0.01) to the nearest 2-bit, two’s-complement fraction, this method returns 0.5 (binary 0.1). The original fraction lies exactly midway between 0.5 and 0.0 (binary 0.0), so this method rounds up. Because it always rounds up, this method is called biased rounding.
The **RND_MOD** bit in the **ASTAT** register enables biased rounding. When the **RND_MOD** bit is cleared, the **RND** option in multiplier instructions uses the normal, unbiased rounding operation, as discussed in “Unbiased Rounding” on page 2-20.

When the **RND_MOD** bit is set (=1), the processor uses biased rounding instead of unbiased rounding. When operating in biased rounding mode, all rounding operations with **A0.L/A1.L** set to 0x8000 round up, rather than only rounding odd values up. For an example of biased rounding, see Table 2-7.

**Table 2-7. Biased Rounding in Multiplier Operation**

<table>
<thead>
<tr>
<th>A0/A1 Before RND</th>
<th>Biased RND Result</th>
<th>Unbiased RND Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 0000 8000</td>
<td>0x00 0001 8000</td>
<td>0x00 0000 0000</td>
</tr>
<tr>
<td>0x00 0001 8000</td>
<td>0x00 0002 0000</td>
<td>0x00 0002 0000</td>
</tr>
<tr>
<td>0x00 0000 8001</td>
<td>0x00 0001 0001</td>
<td>0x00 0001 0001</td>
</tr>
<tr>
<td>0x00 0001 8001</td>
<td>0x00 0002 0001</td>
<td>0x00 0002 0001</td>
</tr>
<tr>
<td>0x00 0000 7FFF</td>
<td>0x00 0000 FFFF</td>
<td>0x00 0000 FFFF</td>
</tr>
<tr>
<td>0x00 0001 7FFF</td>
<td>0x00 0001 FFFF</td>
<td>0x00 0001 FFFF</td>
</tr>
</tbody>
</table>

Biased rounding affects the result only when the **A0.L/A1.L** register contains 0x8000; all other rounding operations work normally. This mode allows more efficient implementation of bit specified algorithms that use biased rounding (for example, the Global System for Mobile Communications (GSM) speech compression routines).

**Truncation**

Another common way to reduce the significant bits representing a number is to simply mask off the **N – M** lower bits. This process is known as **truncation** and results in a relatively large bias. Instructions that do not support rounding revert to truncation. The **RND_MOD** bit in **ASTAT** has no effect on truncation.
Using Computational Status

Special Rounding Instructions

The ALU provides the ability to round the arithmetic results directly into a data register with biased or unbiased rounding as described above. It also provides the ability to round on different bit boundaries. The options RND12, RND, and RND20 round at bit 12, bit 16, and bit 20, respectively, regardless of the state of the RND_MOD bit in ASTAT.

For example:

R3.L = R4 (RND) ;
performs biased rounding at bit 16, depositing the result in a half word.

R3.L = R4 + R5 (RND12) ;
performs an addition of two 32-bit numbers, biased rounding at bit 12, depositing the result in a half word.

R3.L = R4 + R5 (RND20) ;
performs an addition of two 32-bit numbers, biased rounding at bit 20, depositing the result in a half word.

Using Computational Status

The multiplier, ALU, and shifter update the overflow and other status flags in the processor’s Arithmetic Status (ASTAT) register. To use status conditions from computations in program sequencing, use conditional instructions to test the CC flag in the ASTAT register after the instruction executes. This method permits monitoring each instruction’s outcome. The ASTAT register is a 32-bit register, with some bits reserved. To ensure compatibility with future implementations, writes to this register should write back the values read from these reserved bits.
ASTAT Register

Figure 2-9 describes the Arithmetic Status (ASTAT) register. The processor updates the status bits in ASTAT, indicating the status of the most recent ALU, multiplier, or shifter operation.

Arithmetic Status Register (ASTAT)

Reset = 0x0000 0000

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>VS (Sticky Dreg Overflow)</td>
</tr>
<tr>
<td>30</td>
<td>V (Dreg Overflow)</td>
</tr>
<tr>
<td>29</td>
<td>0 - Last result written from ALU to Data Register File register has not overflowed</td>
</tr>
<tr>
<td>28</td>
<td>1 - Last result has overflow</td>
</tr>
<tr>
<td>27</td>
<td>AV1S (Sticky A1 Overflow)</td>
</tr>
<tr>
<td>26</td>
<td>AV0S (Sticky A0 Overflow)</td>
</tr>
<tr>
<td>24</td>
<td>0 - Last result written to A0 has not overflowed</td>
</tr>
<tr>
<td>23</td>
<td>1 - Last result written to A0 has overflow</td>
</tr>
<tr>
<td>22</td>
<td>V (Dreg Overflow)</td>
</tr>
<tr>
<td>21</td>
<td>0 - Last result written from ALU to Data Register File register has not overflowed</td>
</tr>
<tr>
<td>20</td>
<td>1 - Last result has overflow</td>
</tr>
<tr>
<td>19</td>
<td>AC1 (ALU1 Carry)</td>
</tr>
<tr>
<td>18</td>
<td>0 - Operation in ALU1 does not generate a carry</td>
</tr>
<tr>
<td>17</td>
<td>1 - Operation generates a carry</td>
</tr>
<tr>
<td>16</td>
<td>AC0 (ALU0 Carry)</td>
</tr>
<tr>
<td>15</td>
<td>0 - Operation in ALU0 does not generate a carry</td>
</tr>
<tr>
<td>14</td>
<td>1 - Operation generates a carry</td>
</tr>
<tr>
<td>13</td>
<td>RND_MOD (Rounding Mode)</td>
</tr>
<tr>
<td>12</td>
<td>0 - Unbiased rounding</td>
</tr>
<tr>
<td>11</td>
<td>1 - Biased rounding</td>
</tr>
<tr>
<td>10</td>
<td>AQ (Quotient)</td>
</tr>
<tr>
<td>9</td>
<td>Quotient bit</td>
</tr>
<tr>
<td>8</td>
<td>CC (Condition Code)</td>
</tr>
<tr>
<td>7</td>
<td>0 - Operation in ALU0 does not generate a carry</td>
</tr>
<tr>
<td>6</td>
<td>1 - Operation generates a carry</td>
</tr>
<tr>
<td>5</td>
<td>AN (Negative Result)</td>
</tr>
<tr>
<td>4</td>
<td>0 - Result from last ALU0, ALU1, or shifter operation is not negative</td>
</tr>
<tr>
<td>3</td>
<td>1 - Result is zero</td>
</tr>
<tr>
<td>2</td>
<td>AZ (Zero Result)</td>
</tr>
<tr>
<td>1</td>
<td>0 - Result from last ALU0, ALU1, or shifter operation is not negative</td>
</tr>
<tr>
<td>0</td>
<td>1 - Result is negative</td>
</tr>
</tbody>
</table>

Figure 2-9. Arithmetic Status Register
Arithmetic Logic Unit (ALU)

The two ALUs perform arithmetic and logical operations on fixed-point data. ALU fixed-point instructions operate on 16-, 32-, and 40-bit fixed-point operands and output 16-, 32-, or 40-bit fixed-point results. ALU instructions include:

- Fixed-point addition and subtraction of registers
- Addition and subtraction of immediate values
- Accumulation and subtraction of multiplier results
- Logical AND, OR, NOT, XOR, bitwise XOR, Negate
- Functions: ABS, MAX, MIN, Round, division primitives

ALU Operations

Primary ALU operations occur on ALU0, while parallel operations occur on ALU1, which performs a subset of ALU0 operations.

Table 2-8 describes the possible inputs and outputs of each ALU.

Table 2-8. Inputs and Outputs of Each ALU

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two or four 16-bit operands</td>
<td>One or two 16-bit results</td>
</tr>
<tr>
<td>Two 32-bit operands</td>
<td>One 32-bit result</td>
</tr>
<tr>
<td>32-bit result from the multiplier</td>
<td>Combination of 32-bit result from the multiplier with a 40-bit accumulation result</td>
</tr>
</tbody>
</table>

Combining operations in both ALUs can result in four 16-bit results, two 32-bit results, or two 40-bit results generated in a single instruction.
Computational Units

Single 16-Bit Operations

In single 16-bit operations, any two 16-bit register halves may be used as the input to the ALU. An addition, subtraction, or logical operation produces a 16-bit result that is deposited into an arbitrary destination register half. ALU0 is used for this operation, because it is the primary resource for ALU operations.

For example:

\[ R3.H = R1.H + R2.L \ (NS) ; \]

adds the 16-bit contents of \( R1.H \) (R1 high half) to the contents of \( R2.L \) (R2 low half) and deposits the result in \( R3.H \) (R3 high half) with no saturation.

Dual 16-Bit Operations

In dual 16-bit operations, any two 32-bit registers may be used as the input to the ALU, considered as pairs of 16-bit operands. An addition, subtraction, or logical operation produces two 16-bit results that are deposited into an arbitrary 32-bit destination register. ALU0 is used for this operation, because it is the primary resource for ALU operations.

For example:

\[ R3 = R1 +\mid - R2 \ (S) ; \]

adds the 16-bit contents of \( R2.H \) (R2 high half) to the contents of \( R1.H \) (R1 high half) and deposits the result in \( R3.H \) (R3 high half) with saturation.

The instruction also subtracts the 16-bit contents of \( R2.L \) (R2 low half) from the contents of \( R1.L \) (R1 low half) and deposits the result in \( R3.L \) (R3 low half) with saturation (see Figure 2-10 on page 2-39).
Arithmetic Logic Unit (ALU)

Quad 16-Bit Operations

In quad 16-bit operations, any two 32-bit registers may be used as the inputs to ALU0 and ALU1, considered as pairs of 16-bit operands. A small number of addition or subtraction operations produces four 16-bit results that are deposited into two arbitrary, 32-bit destination registers. Both ALU0 and ALU1 are used for this operation. Because there are only two 32-bit data paths from the Data Register File to the arithmetic units, the same two pairs of 16-bit inputs are presented to ALU1 as to ALU0. The instruction construct is identical to that of a dual 16-bit operation, and input operands must be the same for both ALUs.

For example:

R3 = R0 + R1, R2 = R0 – R1 (S);

performs four operations:

- Adds the 16-bit contents of R1.H (R1 high half) to the 16-bit contents of R0.H (R0 high half) and deposits the result in R3.H with saturation.
- Adds R1.L to R0.L and deposits the result in R3.L with saturation.
- Subtracts the 16-bit contents of R1.H (R1 high half) from the 16-bit contents of the R0.H (R0 high half) and deposits the result in R2.H with saturation.
- Subtracts R1.L from R0.L and deposits the result in R2.L with saturation.

Explicitly, the four equivalent instructions are:

R3.H = R0.H + R1.H (S);
R3.L = R0.L + R1.L (S);
R2.H = R0.H – R1.H (S);
R2.L = R0.L – R1.L (S);
### Computational Units

#### Single 32-Bit Operations

In single 32-bit operations, any two 32-bit registers may be used as the input to the ALU, considered as 32-bit operands. An addition, subtraction, or logical operation produces a 32-bit result that is deposited into an arbitrary 32-bit destination register. ALU0 is used for this operation, because it is the primary resource for ALU operations.

In addition to the 32-bit input operands coming from the Data Register File, operands may be sourced and deposited into the Pointer Register File, consisting of the eight registers $P[5:0], SP, FP$.

Instructions may not intermingle Pointer registers with Data registers.

For example:

- $R3 = R1 + R2 \text{ (NS)}$;
  - adds the 32-bit contents of $R2$ to the 32-bit contents of $R1$ and deposits the result in $R3$ with no saturation.

- $R3 = R1 + R2 \text{ (S)}$;
  - adds the 32-bit contents of $R1$ to the 32-bit contents of $R2$ and deposits the result in $R3$ with saturation.

#### Dual 32-Bit Operations

In dual 32-bit operations, any two 32-bit registers may be used as the input to ALU0 and ALU1, considered as a pair of 32-bit operands. An addition or subtraction produces two 32-bit results that are deposited into two 32-bit destination registers. Both ALU0 and ALU1 are used for this operation. Because only two 32-bit data paths go from the Data Register File to the arithmetic units, the same two 32-bit input registers are presented to ALU0 and ALU1.
Arithmetic Logic Unit (ALU)

For example:

\[ R3 = R1 + R2, \quad R4 = R1 - R2 \quad (\text{NS}) \]

adds the 32-bit contents of \( R2 \) to the 32-bit contents of \( R1 \) and deposits the result in \( R3 \) with no saturation.

The instruction also subtracts the 32-bit contents of \( R2 \) from that of \( R1 \) and deposits the result in \( R4 \) with no saturation.

A specialized form of this instruction uses the ALU 40-bit result registers as input operands, creating the sum and differences of the \( A0 \) and \( A1 \) registers.

For example:

\[ R3 = A0 + A1, \quad R4 = A0 - A1 \quad (\text{S}) \]

transfers to the result registers two 32-bit, saturated, sum and difference values of the ALU registers.

ALU Instruction Summary

Table 2-9 lists the ALU instructions. For more information about assembly language syntax and the effect of ALU instructions on the status flags, see Chapter 15, “Arithmetic Operations.”

In Table 2-9, note the meaning of these symbols:

- Dreg denotes any Data Register File register.
- Dreg_lo_hi denotes any 16-bit register half in any Data Register File register.
- Dreg_lo denotes the lower 16 bits of any Data Register File register.
- imm7 denotes a signed, 7-bit wide, immediate value.
Computational Units

- $A_n$ denotes either ALU Result register $A_0$ or $A_1$.
- DIVS denotes a Divide Sign primitive.
- DIVQ denotes a Divide Quotient primitive.
- MAX denotes the maximum, or most positive, value of the source registers.
- MIN denotes the minimum value of the source registers.
- ABS denotes the absolute value of the upper and lower halves of a single 32-bit register.
- RND denotes rounding a half word.
- RND12 denotes saturating the result of an addition or subtraction and rounding the result on bit 12.
- RND20 denotes saturating the result of an addition or subtraction and rounding the result on bit 20.
- SIGNBITS denotes the number of sign bits in a number, minus one.
- EXPADJ denotes the lesser of the number of sign bits in a number minus one, and a threshold value.
- * Indicates the flag may be set or cleared, depending on the results of the instruction.
- ** Indicates the flag is cleared.
- – Indicates no effect.
- $d$ indicates $A_0$ contains the dividend MSB Exclusive-OR divisor MSB.
## Arithmetic Logic Unit (ALU)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ASTAT Status Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AZ</td>
</tr>
<tr>
<td>Dreg = Dreg + Dreg ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = Dreg – Dreg (S) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = Dreg + Dreg, Dreg = Dreg – Dreg ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_lo_hi = Dreg_lo_hi + Dreg_lo_hi ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_lo_hi = Dreg_lo_hi – Dreg_lo_hi (S) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg ;</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>– Dreg ;</td>
</tr>
<tr>
<td>Dreg = Dreg –</td>
<td>+ Dreg ;</td>
</tr>
<tr>
<td>Dreg = Dreg –</td>
<td>– Dreg ;</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+Dreg, Dreg = Dreg –</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>– Dreg, Dreg = Dreg –</td>
</tr>
<tr>
<td>Dreg = An + An, Dreg = An – An ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg += imm7 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = ( A0 += A1 ) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_lo_hi = ( A0 += A1 ) ;</td>
<td>*</td>
</tr>
<tr>
<td>A0 += A1 ;</td>
<td>*</td>
</tr>
<tr>
<td>A0 –= A1 ;</td>
<td>*</td>
</tr>
<tr>
<td>DIVS ( Dreg, Dreg ) ;</td>
<td>*</td>
</tr>
<tr>
<td>DIVQ ( Dreg, Dreg ) ;</td>
<td>*</td>
</tr>
</tbody>
</table>
### Computational Units

**Table 2-9. ALU Instruction Summary (Cont'd)**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>AZ</th>
<th>AN</th>
<th>AC0</th>
<th>AC0_Copy</th>
<th>AV0</th>
<th>AV1</th>
<th>V</th>
<th>V_Copy</th>
<th>V_S</th>
<th>AQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Dreg = \text{MAX}( Dreg, Dreg )$ (V) ;</td>
<td>*</td>
<td>*</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>*</td>
<td>**/–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$Dreg = \text{MIN}( Dreg, Dreg )$ (V) ;</td>
<td>*</td>
<td>*</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>*</td>
<td>**/–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$Dreg = \text{ABS} Dreg$ (V) ;</td>
<td>*</td>
<td>**</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>*</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$An = \text{ABS} An$ ;</td>
<td>*</td>
<td>**</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>*</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$An = \text{ABS} An,$ $An = \text{ABS} An$ ;</td>
<td>*</td>
<td>**</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>*</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$An = –An$ ;</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$An = –An,$ $An = –An$ ;</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$An = An (S)$ ;</td>
<td>*</td>
<td>*</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>*</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$An = An (S),$ $An = An (S)$ ;</td>
<td>*</td>
<td>*</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>*</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$Dreg_{lo}hi = Dreg (RND)$ ;</td>
<td>*</td>
<td>*</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>*</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$Dreg_{lo}hi = Dreg + Dreg$ (RND12) ;</td>
<td>*</td>
<td>*</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>*</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$Dreg_{lo}hi = Dreg - Dreg$ (RND12) ;</td>
<td>*</td>
<td>*</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>*</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$Dreg_{lo}hi = Dreg + Dreg$ (RND20) ;</td>
<td>*</td>
<td>*</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>*</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$Dreg_{lo}hi = Dreg - Dreg$ (RND20) ;</td>
<td>*</td>
<td>*</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>*</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$Dreg_{lo} = \text{SIGNBITS} Dreg$ ;</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$Dreg_{lo} = \text{SIGNBITS} Dreg_{lo}hi$ ;</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$Dreg_{lo} = \text{SIGNBITS} An$ ;</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
Arithmetic Logic Unit (ALU)

Table 2-9. ALU Instruction Summary (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>AZ</th>
<th>AN</th>
<th>AC0</th>
<th>AC0_COPY</th>
<th>AV0</th>
<th>AV1</th>
<th>V</th>
<th>V_COPY</th>
<th>AQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dreg_lo = EXPADJ (Dreg, Dreg_lo) (V);</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Dreg_lo = EXPADJ (Dreg_lo_hi, Dreg_lo);</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Dreg = Dreg &amp; Dreg;</td>
<td>*</td>
<td>*</td>
<td>**</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>**/–</td>
<td>--</td>
</tr>
<tr>
<td>Dreg = - Dreg;</td>
<td>*</td>
<td>*</td>
<td>**</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>**/–</td>
<td>--</td>
</tr>
<tr>
<td>Dreg = Dreg</td>
<td>Dreg;</td>
<td>*</td>
<td>*</td>
<td>**</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>**/–</td>
</tr>
<tr>
<td>Dreg = Dreg ^ Dreg;</td>
<td>*</td>
<td>*</td>
<td>**</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>**/–</td>
<td>--</td>
</tr>
<tr>
<td>Dreg =– Dreg;</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>*</td>
<td>--</td>
</tr>
</tbody>
</table>

ALU Division Support Features

The ALU supports division with two special divide primitives. These instructions (DIVS, DIVQ) let programs implement a non-restoring, conditional (error checking), addition/subtraction/division algorithm.

The division can be either signed or unsigned, but both the dividend and divisor must be of the same type. Details about using division and programming examples are available in Chapter 15, “Arithmetic Operations.”
Computational Units

Special SIMD Video ALU Operations

Four 8-bit Video ALUs enable the processor to process video information with high efficiency. Each Video ALU instruction may take from one to four pairs of 8-bit inputs and return one to four 8-bit results. The inputs are presented to the Video ALUs in two 32-bit words from the Data Register File. The possible operations include:

- Quad 8-Bit Add or Subtract
- Quad 8-Bit Average
- Quad 8-Bit Pack or Unpack
- Quad 8-Bit Subtract-Absolute-Accumulate
- Byte Align

For more information about the operation of these instructions, see Chapter 18, “Video Pixel Operations.”

Multiply Accumulators (Multipliers)

The two multipliers (MAC0 and MAC1) perform fixed-point multiplication and multiply and accumulate operations. Multiply and accumulate operations are available with either cumulative addition or cumulative subtraction.

Multiplier fixed-point instructions operate on 16-bit fixed-point data and produce 32-bit results that may be added or subtracted from a 40-bit accumulator.
Multiply Accumulators (Multipliers)

Inputs are treated as fractional or integer, unsigned or two’s-complement. Multiplier instructions include:

- Multiplication
- Multiply and accumulate with addition, rounding optional
- Multiply and accumulate with subtraction, rounding optional
- Dual versions of the above

Multiplier Operation

Each multiplier has two 32-bit inputs from which it derives the two 16-bit operands. For single multiply and accumulate instructions, these operands can be any Data registers in the Data Register File. Each multiplier can accumulate results in its Accumulator register, $A_1$ or $A_0$. The accumulator results can be saturated to 32 or 40 bits. The multiplier result can also be written directly to a 16- or 32-bit destination register with optional rounding.

Each multiplier instruction determines whether the inputs are either both in integer format or both in fractional format. The format of the result matches the format of the inputs. In MAC0, both inputs are treated as signed or unsigned. In MAC1, there is a mixed-mode option.

If both inputs are fractional and signed, the multiplier automatically shifts the result left one bit to remove the redundant sign bit. Unsigned fractional, integer, and mixed modes do not perform a shift for sign bit correction. Multiplier instruction options specify the data format of the inputs. See “Multiplier Instruction Options” on page 2-40 for more information.
Computational Units

Placing Multiplier Results in Multiplier Accumulator Registers

As shown in Figure 2-10 on page 2-42, each multiplier has a dedicated accumulator, A0 or A1. Each Accumulator register is divided into three sections—A0.L/A1.L (bits 15:0), A0.H/A1.H (bits 31:16), and A0.X/A1.X (bits 39:32).

When the multiplier writes to its result Accumulator registers, the 32-bit result is deposited into the lower bits of the combined Accumulator register, and the MSB is sign-extended into the upper eight bits of the register (A0.X/A1.X).

Multiplier output can be deposited not only in the A0 or A1 registers, but also in a variety of 16- or 32-bit Data registers in the Data Register File.

Rounding or Saturating Multiplier Results

On a multiply and accumulate operation, the accumulator data can be saturated and, optionally, rounded for extraction to a register or register half. When a multiply deposits a result only in a register or register half, the saturation and rounding works the same way. The rounding and saturation operations work as follows.

- Rounding is applied only to fractional results except for the IH option, which applies rounding and high half extraction to an integer result.

For the IH option, the rounded result is obtained by adding 0x8000 to the accumulator (for MAC) or multiply result (for mult) and then saturating to 32-bits. For more information, see “Rounding Multiplier Results” on page 2-19.
Multiply Accumulators (Multipliers)

- If an overflow or underflow has occurred, the saturate operation sets the specified Result register to the maximum positive or negative value. For more information, see the following section.

Saturating Multiplier Results on Overflow

The following bits in \texttt{ASTAT} indicate multiplier overflow status:

- Bit 16 (\texttt{AV0}) and bit 18 (\texttt{AV1}) record overflow condition (whether the result has overflowed 32 bits) for the \texttt{A0} and \texttt{A1} accumulators, respectively.

  If the bit is cleared (=0), no overflow or underflow has occurred. If the bit is set (=1), an overflow or underflow has occurred. The \texttt{AV0S} and \texttt{AV1S} bits are sticky bits.

- Bit 24 (\texttt{V}) and bit 25 (\texttt{VS}) are set if overflow occurs in extracting the accumulator result to a register.

Multiplier Instruction Summary

Table 2-10 lists the multiplier instructions. For more information about assembly language syntax and the effect of multiplier instructions on the status flags, see Chapter 15, “Arithmetic Operations.”

In Table 2-10, note the meaning of these symbols:

- Dreg denotes any Data Register File register.
- Dreg\_lo\_hi denotes any 16-bit register half in any Data Register File register.
- Dreg\_lo denotes the lower 16 bits of any Data Register File register.
- Dreg\_hi denotes the upper 16 bits of any Data Register File register.
Computational Units

- $A_n$ denotes either MAC Accumulator register $A_0$ or $A_1$.
- * Indicates the flag may be set or cleared, depending on the results of the instruction.
- – Indicates no effect.

Multiplier instruction options are described on page 2-40.

Table 2-10. Multiplier Instruction Summary

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ASTAT Status Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AV0</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi;</td>
<td>–</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi;</td>
<td>–</td>
</tr>
<tr>
<td>Dreg = Dreg_lo_hi * Dreg_lo_hi;</td>
<td>–</td>
</tr>
<tr>
<td>An = Dreg_lo_hi * Dreg_lo_hi;</td>
<td>*</td>
</tr>
<tr>
<td>An += Dreg_lo_hi * Dreg_lo_hi;</td>
<td>*</td>
</tr>
<tr>
<td>An -= Dreg_lo_hi * Dreg_lo_hi;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_lo = ( A0 = Dreg_lo_hi * Dreg_lo_hi );</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_lo = ( A0 += Dreg_lo_hi * Dreg_lo_hi );</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_lo = ( A0 -= Dreg_lo_hi * Dreg_lo_hi );</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_hi = ( A1 = Dreg_lo_hi * Dreg_lo_hi );</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_hi = ( A1 += Dreg_lo_hi * Dreg_lo_hi );</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_hi = ( A1 -= Dreg_lo_hi * Dreg_lo_hi );</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = ( An = Dreg_lo_hi * Dreg_lo_hi );</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = ( An += Dreg_lo_hi * Dreg_lo_hi );</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = ( An -= Dreg_lo_hi * Dreg_lo_hi );</td>
<td>*</td>
</tr>
<tr>
<td>Dreg *= Dreg;</td>
<td>–</td>
</tr>
</tbody>
</table>
Multiply Accumulators (Multipliers)

Multiplier Instruction Options

The following descriptions of multiplier instruction options provide an overview. Not all options are available for all instructions. For information about how to use these options with their respective instructions, see Chapter 15, “Arithmetic Operations.”

**default**
No option; input data is signed fraction.

(IS)  Input data operands are signed integer. No shift correction is made.

(FU)  Input data operands are unsigned fraction. No shift correction is made.

(IU)  Input data operands are unsigned integer. No shift correction is made.

(T)   Input data operands are signed fraction. When copying to the destination half register, truncates the lower 16 bits of the Accumulator contents.

(TFU) Input data operands are unsigned fraction. When copying to the destination half register, truncates the lower 16 bits of the Accumulator contents.

(ISS2) If multiplying and accumulating to a register:

Input data operands are signed integer. When copying to the destination register, Accumulator contents are scaled (multiplied x2 by a one-place shift-left). If scaling produces a signed value larger than 32 bits, the number is saturated to its maximum positive or negative value.
Computational Units

If multiplying and accumulating to a half register:

When copying the lower 16 bits to the destination half register, the Accumulator contents are scaled. If scaling produces a signed value greater than 16 bits, the number is saturated to its maximum positive or negative value.

(IH)  This option indicates integer multiplication with high half word extraction. The Accumulator is saturated at 32 bits, and bits [31:16] of the Accumulator are rounded, and then copied into the destination half register.

(W32) Input data operands are signed fraction with no extension bits in the Accumulators at 32 bits. Left-shift correction of the product is performed, as required. This option is used for legacy GSM speech vocoder algorithms written for 32-bit Accumulators. For this option only, this special case applies: \(0x8000 \times 0x8000 = 0x7FFF\).

(M)  Operation uses mixed-multiply mode. Valid only for MAC1 versions of the instruction. Multiplies a signed fraction by an unsigned fractional operand with no left-shift correction. Operand one is signed; operand two is unsigned. MAC0 performs an unmixed multiply on signed fractions by default, or another format as specified. That is, MAC0 executes the specified signed/signed or unsigned/unsigned multiplication. The \((m)\) option can be used alone or in conjunction with one other format option.
Multiplier Data Flow Details

Figure 2-10 shows the Register files and ALUs, along with the multiplier/accumulators.

Each multiplier has two 16-bit inputs, performs a 16-bit multiplication, and stores the result in a 40-bit accumulator or extracts to a 16-bit or 32-bit register. Two 32-bit words are available at the MAC inputs, providing four 16-bit operands to choose from.

One of the operands must be selected from the low half or the high half of one 32-bit word. The other operand must be selected from the low half or the high half of the other 32-bit word. Thus, each MAC is presented with four possible input operand combinations. The two 32-bit words can
Computational Units

contain the same register information, giving the options for squaring and multiplying the high half and low half of the same register. Figure 2-11 show these possible combinations.

Figure 2-11. Four Possible Combinations of MAC Operations

The 32-bit product is passed to a 40-bit adder/subtractor, which may add or subtract the new product from the contents of the Accumulator Result register or pass the new product directly to the Data Register File Results register. For results, the A0 and A1 registers are 40 bits wide. Each of these registers consists of smaller 32- and 8-bit registers—A0.W, A1.W, A0.X, and A1.X.
Multiply Accumulators (Multipliers)

For example:

\[ A1 += R3.H \times R4.H ; \]

In this instruction, the MAC1 multiplier/accumulator performs a multiply and accumulates the result with the previous results in the A1 Accumulator.

Multiply Without Accumulate

The multiplier may operate without the accumulation function. If accumulation is not used, the result can be directly stored in a register from the Data Register File or the Accumulator register. The destination register may be 16 bits or 32 bits. If a 16-bit destination register is a low half, then MAC0 is used; if it is a high half, then MAC1 is used. For a 32-bit destination register, either MAC0 or MAC1 is used.

If the destination register is 16 bits, then the word that is extracted from the multiplier depends on the data type of the input.

- If the multiplication uses fractional operands or the fH option, then the high half of the result is extracted and stored in the 16-bit destination registers (see Figure 2-12).

- If the multiplication uses integer operands, then the low half of the result is extracted and stored in the 16-bit destination registers. These extractions provide the most useful information in the resultant 16-bit word for the data type chosen (see Figure 2-13).
Computational Units

For example, this instruction uses fractional, unsigned operands:

\[ R0.L = R1.L \times R2.L \text{ (FU)} ; \]

The instruction deposits the upper 16 bits of the multiply answer with rounding and saturation into the lower half of \( R0 \), using MAC0. This instruction uses unsigned integer operands:

\[ R0.H = R2.H \times R3.H \text{ (IU)} ; \]

The instruction deposits the lower 16 bits of the multiply answer with any required saturation into the high half of \( R0 \), using MAC1.

\[ R0 = R1.L \times R2.L ; \]

Regardless of operand type, the preceding operation deposits 32 bits of the multiplier answer with saturation into \( R0 \), using MAC0.
Multiply Accumulators (Multipliers)

The processor supports a multicycle 32-bit MAC instruction:

\[ \text{Dreg} \times= \text{Dreg} \]

The single instruction multiplies two 32-bit integer operands and provides a 32-bit integer result, destroying one of the input operands.

The instruction takes multiple cycles to execute. For more information about the exact operation of this instruction, refer to Chapter 15, “Arithmetic Operations.” This macro function is interruptable and does not modify the data in either Accumulator register A0 or A1.

Figure 2-13. Multiplication of Integer Operands

Special 32-Bit Integer MAC Instruction

The processor supports a multicycle 32-bit MAC instruction:

\[ \text{Dreg} \times= \text{Dreg} \]

The single instruction multiplies two 32-bit integer operands and provides a 32-bit integer result, destroying one of the input operands.

The instruction takes multiple cycles to execute. For more information about the exact operation of this instruction, refer to Chapter 15, “Arithmetic Operations.” This macro function is interruptable and does not modify the data in either Accumulator register A0 or A1.
Dual MAC Operations

The processor has two 16-bit MACs. Both MACs can be used in the same operation to double the MAC throughput. The same two 32-bit input registers are offered to each MAC unit, providing each with four possible combinations of 16-bit input operands. Dual MAC operations are frequently referred to as vector operations, because a program could store vectors of samples in the four input operands and perform vector computations.

An example of a dual multiply and accumulate instruction is

\[
A1 += R1.H \times R2.L, A0 += R1.L \times R2.H ;
\]

This instruction represents two multiply and accumulate operations.

- In one operation (MAC1) the high half of \( R1 \) is multiplied by the low half of \( R2 \) and added to the contents of the \( A1 \) Accumulator.

- In the second operation (MAC0) the low half of \( R1 \) is multiplied by the high half of \( R2 \) and added to the contents of \( A0 \).

The results of the MAC operations may be written to registers in a number of ways: as a pair of 16-bit halves, as a pair of 32-bit registers, or as an independent 16-bit half register or 32-bit register.

For example:

\[
R3.H = (A1 += R1.H \times R2.L), R3.L = (A0 += R1.L \times R2.L) ;
\]

In this instruction, the 40-bit Accumulator is packed into a 16-bit half register. The result from MAC1 must be transferred to a high half of a destination register and the result from MAC0 must be transferred to the low half of the same destination register.
Barrel Shifter (Shifter)

The operand type determines the correct bits to extract from the Accumulator and deposit in the 16-bit destination register. See “Multiply Without Accumulate” on page 2-44.

\[ R3 = (A1 += R1.H \times R2.L), \quad R2 = (A0 += R1.L \times R2.L) ; \]

In this instruction, the 40-bit Accumulators are packed into two 32-bit registers. The registers must be register pairs \((R[1:0], R[3:2], R[5:4], R[7:6])\).

\[ R3.H = (A1 += R1.H \times R2.L), \quad A0 += R1.L \times R2.L ; \]

This instruction is an example of one Accumulator—but not the other—being transferred to a register. Either a 16- or 32-bit register may be specified as the destination register.

Barrel Shifter (Shifter)

The shifter provides bitwise shifting functions for 16-, 32-, or 40-bit inputs, yielding a 16-, 32-, or 40-bit output. These functions include arithmetic shift, logical shift, rotate, and various bit test, set, pack, unpack, and exponent detection functions. These shift functions can be combined to implement numerical format control, including full floating-point representation.

Shifter Operations

The shifter instructions (\(\gg\gg\), \(\gg\), \(\ll\), ASHIFT, LSHIFT, ROT) can be used various ways, depending on the underlying arithmetic requirements. The ASHIFT and \(\gg\gg\) instructions represent the arithmetic shift. The LSHIFT, \(\ll\), and \(\gg\) instructions represent the logical shift.
Computational Units

The arithmetic shift and logical shift operations can be further broken into subsections. Instructions that are intended to operate on 16-bit single or paired numeric values (as would occur in many DSP algorithms) can use the instructions \texttt{ASHIFT} and \texttt{LSHIFT}. These are typically three-operand instructions.

Instructions that are intended to operate on a 32-bit register value and use two operands, such as instructions frequently used by a compiler, can use the \texttt{>>>} and \texttt{>>} instructions.

Arithmetic shift, logical shift, and rotate instructions can obtain the shift argument from a register or directly from an immediate value in the instruction. For details about shifter related instructions, see “Shifter Instruction Summary” on page 2-53.

Two-Operand Shifts

Two-operand shift instructions shift an input register and deposit the result in the same register.

Immediate Shifts

An immediate shift instruction shifts the input bit pattern to the right (downshift) or left (upshift) by a given number of bits. Immediate shift instructions use the data value in the instruction itself to control the amount and direction of the shifting operation.

The following example shows the input value downshifted.

\begin{verbatim}
R0 contains 0000 B6A3 ;
R0 >>= 0x04 ;
results in
R0 contains 0000 0B6A ;
\end{verbatim}
Barrel Shifter (Shifter)

The following example shows the input value upshifted.

R0 contains 0000 B6A3 ;
R0 <<= 0x04 ;

results in
R0 contains 000B 6A30 ;

Register Shifts

Register-based shifts use a register to hold the shift value. The entire 32-bit register is used to derive the shift value, and when the magnitude of the shift is greater than or equal to 32, then the result is either 0 or –1.

The following example shows the input value upshifted.

R0 contains 0000 B6A3 ;
R2 contains 0000 0004 ;
R0 <<= R2 ;

results in
R0 contains 000B 6A30 ;

Three-Operand Shifts

Three-operand shifter instructions shift an input register and deposit the result in a destination register.

Immediate Shifts

Immediate shift instructions use the data value in the instruction itself to control the amount and direction of the shifting operation.
The following example shows the input value downshifted.

R0 contains 0000 B6A3 ;
R1 = R0 >> 0x04 ;

results in

R1 contains 0000 0B6A ;

The following example shows the input value upshifted.

R0.L contains B6A3 ;
R1.H = R0.L << 0x04 ;

results in

R1.H contains 6A30 ;

Register Shifts

Register-based shifts use a register to hold the shift value. When a register is used to hold the shift value (for ASHIFT, LSHIFT or ROT), then the shift value is always found in the low half of a register (Rn.L). The bottom six bits of Rn.L are masked off and used as the shift value.

The following example shows the input value upshifted.

R0 contains 0000 B6A3 ;
R2.L contains 0004 ;
R1 = R0 ASHIFT by R2.L ;

results in

R1 contains 000B 6A30 ;
Barrel Shifter (Shifter)

The following example shows the input value rotated. Assume the Condition Code (CC) bit is set to 0. For more information about CC, see “Condition Code Flag” on page 4-18.

R0 contains ABCD EF12;
R2.L contains 0004;
R1 = R0 ROT by R2.L;

results in
R1 contains BCDE F125;

Note the CC bit is included in the result, at bit 3.

Bit Test, Set, Clear, Toggle

The shifter provides the method to test, set, clear, and toggle specific bits of a data register. All instructions have two arguments—the source register and the bit field value. The test instruction does not change the source register. The result of the test instruction resides in the CC bit.

The following examples show a variety of operations.

BITCLR ( R0, 6 );
BITSET ( R2, 9 );
BITTGL ( R3, 2 );
CC = BITTST ( R3, 0 );

Field Extract and Field Deposit

If the shifter is used, a source field may be deposited anywhere in a 32-bit destination field. The source field may be from 1 bit to 16 bits in length. In addition, a 1- to 16-bit field may be extracted from anywhere within a 32-bit source field.
Two register arguments are used for these functions. One holds the 32-bit destination or 32-bit source. The other holds the extract/deposit value, its length, and its position within the source.

**Shifter Instruction Summary**

Table 2-11 lists the shifter instructions. For more information about assembly language syntax and the effect of shifter instructions on the status flags, see Chapter 14, “Shift/Rotate Operations.”

In Table 2-11, note the meaning of these symbols:

- Dreg denotes any Data Register File register.
- Dreg_lo denotes the lower 16 bits of any Data Register File register.
- Dreg_hi denotes the upper 16 bits of any Data Register File register.
- * Indicates the flag may be set or cleared, depending on the results of the instruction.
- * 0 Indicates versions of the instruction that send results to Accumulator A0 set or clear AV0.
- * 1 Indicates versions of the instruction that send results to Accumulator A1 set or clear AV1.
- ** Indicates the flag is cleared.
- *** Indicates CC contains the latest value shifted into it.
- – Indicates no effect.
Table 2-11. Shifter Instruction Summary

<table>
<thead>
<tr>
<th>Instruction</th>
<th>AZ</th>
<th>AN</th>
<th>AC0</th>
<th>AC0_COPY</th>
<th>AC1</th>
<th>AV0</th>
<th>AV0S</th>
<th>AV1</th>
<th>AV1S</th>
<th>CC</th>
<th>V</th>
<th>V_COPY</th>
<th>VS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BITCLR (Dreg, uimm5)</td>
<td>*</td>
<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>BITSET (Dreg, uimm5)</td>
<td>**</td>
<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>BITTGL (Dreg, uimm5)</td>
<td>*</td>
<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>CC = BITTST (Dreg, uimm5)</td>
<td></td>
<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>CC = !BITTST (Dreg, uimm5)</td>
<td></td>
<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>Dreg = DEPOSIT (Dreg, Dreg)</td>
<td>*</td>
<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>Dreg = EXTRACT (Dreg, Dreg)</td>
<td>*</td>
<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>BITMUX (Dreg, Dreg, A0)</td>
<td></td>
<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>Dreg_lo = ONES Dreg</td>
<td></td>
<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>Dreg = PACK (Dreg_lo_hi,</td>
<td></td>
<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>Dreg_lo_hi);</td>
<td></td>
<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>Dreg &gt;&gt;&gt;= uimm5</td>
<td>*</td>
<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>Dreg &gt;&gt;= uimm5</td>
<td>*</td>
<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>Dreg &lt;&lt;= uimm5</td>
<td>*</td>
<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>Dreg = Dreg &gt;&gt;&gt; uimm5</td>
<td>*</td>
<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>Dreg = Dreg &gt;&gt; uimm5</td>
<td>*</td>
<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>Dreg = Dreg &lt;&lt; uimm5</td>
<td>*</td>
<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>Dreg = Dreg &gt;&gt;&gt; uimm4 (V)</td>
<td>*</td>
<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>Dreg = Dreg &gt;&gt; uimm4 (V)</td>
<td>*</td>
<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>Dreg = Dreg &lt;&lt; uimm4 (V)</td>
<td>*</td>
<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>
Table 2-11. Shifter Instruction Summary (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>AZ</th>
<th>AN</th>
<th>AC0</th>
<th>AC0_COPY</th>
<th>AV0</th>
<th>AV1</th>
<th>AV0S</th>
<th>AV1S</th>
<th>CC</th>
<th>V</th>
<th>V_COPY</th>
<th>VS</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A_n = A_n \ggg uimm5 )</td>
<td>*</td>
<td>*</td>
<td>–</td>
<td></td>
<td>** 0/</td>
<td>1/–</td>
<td>–</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( A_n = A_n \gg uimm5 )</td>
<td>*</td>
<td>*</td>
<td>–</td>
<td></td>
<td>** 0/</td>
<td>1/–</td>
<td>–</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( A_n = A_n \ll uimm5 )</td>
<td>*</td>
<td>*</td>
<td>–</td>
<td></td>
<td>* 0</td>
<td>1</td>
<td>–</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( Dreg_{lo_hi} = Dreg_{lo_hi} \ggg uimm4 )</td>
<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>( Dreg_{lo_hi} = Dreg_{lo_hi} \gg uimm4 )</td>
<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>( Dreg_{lo_hi} = Dreg_{lo_hi} \ll uimm4 )</td>
<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>( Dreg \gggg Dreg )</td>
<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>( Dreg \ggg Dreg )</td>
<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>( Dreg \ll Dreg )</td>
<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>( Dreg = \text{ASHIFT} Dreg ) \ggg Dreg_{lo}</td>
<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>( Dreg = \text{LSHIFT} Dreg ) \ggg Dreg_{lo}</td>
<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>( Dreg = \text{ROT} Dreg ) \ggg imm6 )</td>
<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>( Dreg = \text{ASHIFT} Dreg ) \ggg Dreg_{lo} (V)</td>
<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>( Dreg = \text{LSHIFT} Dreg ) \ggg Dreg_{lo} (V)</td>
<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>( Dreg_{lo_hi} = \text{ASHIFT} Dreg \ggg Dreg_{lo} )</td>
<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>
### Barrel Shifter (Shifter)

Table 2-11. Shifter Instruction Summary  (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ASTAT Status Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AZ</td>
</tr>
<tr>
<td>Dreg_lo_hi = LSHIFT Dreg_lo_hi BY Dreg_lo ;</td>
<td>*</td>
</tr>
<tr>
<td>An = An ASHIFT BY Dreg_lo ;</td>
<td>*</td>
</tr>
<tr>
<td>An = An ROT BY imm6 ;</td>
<td>–</td>
</tr>
<tr>
<td>Dreg = ( Dreg + Dreg ) &lt;&lt; 1 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = ( Dreg + Dreg ) &lt;&lt; 2 ;</td>
<td>*</td>
</tr>
</tbody>
</table>
3 OPERATING MODES AND STATES

The processor supports the following three processor modes:

- User mode
- Supervisor mode
- Emulation mode

Emulation and Supervisor modes have unrestricted access to the core resources. User mode has restricted access to certain system resources, thus providing a protected software environment.

User mode is considered the domain of application programs. Supervisor mode and Emulation mode are usually reserved for the kernel code of an operating system.

The processor mode is determined by the Event Controller. When servicing an interrupt, a nonmaskable interrupt (NMI), or an exception, the processor is in Supervisor mode. When servicing an emulation event, the processor is in Emulation mode. When not servicing any events, the processor is in User mode.

The current processor mode may be identified by interrogating the IPEND memory-mapped register (MMR), as shown in Table 3-1.

MMRs cannot be read while the processor is in User mode.
In addition, the processor supports the following two non-processing states:

- **Idle state**
- **Reset state**

*Figure 3-1* illustrates the processor modes and states as well as the transition conditions between them.

---

**Table 3-1. Identifying the Current Processor Mode**

<table>
<thead>
<tr>
<th>Event</th>
<th>Mode</th>
<th>IPEND</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt</td>
<td>Supervisor</td>
<td>≥ 0x10 but IPEND[0], IPEND[1], IPEND[2], and IPEND[3] = 0.</td>
</tr>
<tr>
<td>Exception</td>
<td>Supervisor</td>
<td>≥ 0x08 The core is processing an exception event if IPEND[0] = 0, IPEND[1] = 0, IPEND[2] = 0, IPEND[3] = 1, and IPEND[15:4] are 0’s or 1’s.</td>
</tr>
<tr>
<td>NMI</td>
<td>Supervisor</td>
<td>≥ 0x04 The core is processing an NMI event if IPEND[0] = 0, IPEND[1] = 0, IPEND[2] = 1, and IPEND[15:2] are 0’s or 1’s.</td>
</tr>
<tr>
<td>Reset</td>
<td>Supervisor</td>
<td>= 0x02 As the reset state is exited, IPEND is set to 0x02, and the reset vector runs in Supervisor mode.</td>
</tr>
<tr>
<td>Emulation</td>
<td>Emulator</td>
<td>= 0x01 The processor is in Emulation mode if IPEND[0] = 1, regardless of the state of the remaining bits IPEND[15:1].</td>
</tr>
<tr>
<td>None</td>
<td>User</td>
<td>= 0x00</td>
</tr>
</tbody>
</table>
Operating Modes and States

User Mode

The processor is in User mode when it is not in Reset or Idle state, and when it is not servicing an interrupt, NMI, exception, or emulation event. User mode is used to process application level code that does not require explicit access to system registers. Any attempt to access restricted system registers causes an exception event. Table 3-2 lists the registers that may be accessed in User mode.

Figure 3-1. Processor Modes and States

(1) Normal exit from Reset is to Supervisor mode. However, emulation hardware may have initiated a reset. If so, exit from Reset is to Emulation.
User Mode

Table 3-2. Registers Accessible in User Mode

<table>
<thead>
<tr>
<th>Processor Registers</th>
<th>Register Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Registers</td>
<td>R[7:0], A[1:0]</td>
</tr>
<tr>
<td>Pointer Registers</td>
<td>P[5:0], SP, FP, I[3:0], M[3:0], L[3:0], B[3:0]</td>
</tr>
<tr>
<td>Sequencer and Status Registers</td>
<td>RETS, LC[1:0], LT[1:0], LB[1:0], ASTAT, CYCLES, CYCLES2</td>
</tr>
</tbody>
</table>

Protected Resources and Instructions

System resources consist of a subset of processor registers, all MMRs, and a subset of protected instructions. These system and core MMRs are located starting at address 0xFFC0 0000. This region of memory is protected from User mode access. Any attempt to access MMR space in User mode causes an exception.

A list of protected instructions appears in Table 3-3. Any attempt to issue any of the protected instructions from User mode causes an exception event.

Table 3-3. Protected Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTI</td>
<td>Return from Interrupt</td>
</tr>
<tr>
<td>RTX</td>
<td>Return from Exception</td>
</tr>
<tr>
<td>RTN</td>
<td>Return from NMI</td>
</tr>
<tr>
<td>CLI</td>
<td>Disable Interrupts</td>
</tr>
<tr>
<td>STI</td>
<td>Enable Interrupts</td>
</tr>
<tr>
<td>RAISE</td>
<td>Force Interrupt/Reset</td>
</tr>
<tr>
<td>RTE</td>
<td>Return from Emulation</td>
</tr>
<tr>
<td></td>
<td>Causes an exception only if executed outside Emulation mode</td>
</tr>
</tbody>
</table>
Operating Modes and States

Protected Memory

Additional memory locations can be protected from User mode access. A Cacheability Protection Lookaside Buffer (CPLB) entry can be created and enabled. See “Memory Management Unit” on page 6-45 for further information.

Entering User Mode

When coming out of reset, the processor is in Supervisor mode because it is servicing a reset event. To enter User mode from the Reset state, two steps must be performed. First, a return address must be loaded into the RETI register. Second, an RTI must be issued. The following example code shows how to enter User mode upon reset.

Example Code to Enter User Mode Upon Reset

Listing 3-1 provides code for entering User mode from reset.

Listing 3-1. Entering User Mode from Reset

```
P1.L = START ;  /* Point to start of user code */
P1.H = START ;
RETI = P1 ;
RTI ;  /* Return from Reset Event */
START :  /* Place user code here */
```

Return Instructions That Invoke User Mode

Table 3-4 provides a summary of return instructions that can be used to invoke User mode from various processor event service routines. When these instructions are used in service routines, the value of the return address must be first stored in the appropriate event RETx register. In the
case of an interrupt routine, if the service routine is interruptible, the return address is stored on the stack. For this case, the address can be found by popping the value from the stack into RETI. Once RETI has been loaded, the RTI instruction can be issued.

Note the stack pop is optional. If the RETI register is not pushed/popped, then the interrupt service routine becomes non-interruptible, because the return address is not saved on the stack.

The processor remains in User mode until one of these events occurs:

- An interrupt, NMI, or exception event invokes Supervisor mode.
- An emulation event invokes Emulation mode.
- A reset event invokes the Reset state.

Table 3-4. Return Instructions That Can Invoke User Mode

<table>
<thead>
<tr>
<th>Current Process Activity</th>
<th>Return Instruction to Use</th>
<th>Execution Resumes at Address in This Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt Service Routine</td>
<td>RTI</td>
<td>RETI</td>
</tr>
<tr>
<td>Exception Service Routine</td>
<td>RTX</td>
<td>RETX</td>
</tr>
<tr>
<td>Nonmaskable Interrupt Service Routine</td>
<td>RTN</td>
<td>RETN</td>
</tr>
<tr>
<td>Emulation Service Routine</td>
<td>RTE</td>
<td>RETE</td>
</tr>
</tbody>
</table>
Supervisor Mode

The processor services all interrupt, NMI, and exception events in Supervisor mode.

Supervisor mode has full, unrestricted access to all processor system resources, including all emulation resources, unless a CPLB has been configured and enabled. See “Memory Management Unit” on page 6-45 for a further description. Only Supervisor mode can use the register alias USP, which references the User Stack Pointer in memory. This register alias is necessary because in Supervisor mode, SP refers to the kernel stack pointer rather than to the user stack pointer.

Normal processing begins in Supervisor mode from the Reset state. Deasserting the RESET signal switches the processor from the Reset state to Supervisor mode where it remains until an emulation event or Return instruction occurs to change the mode. Before the Return instruction is issued, the RETI register must be loaded with a valid return address.

Non-OS Environments

For non-OS environments, application code should remain in Supervisor mode so that it can access all core and system resources. When RESET is deasserted, the processor initiates operation by servicing the reset event. Emulation is the only event that can pre-empt this activity. Therefore, lower priority events cannot be processed.

One way of keeping the processor in Supervisor mode and still allowing lower priority events to be processed is to set up and force the lowest priority interrupt (IVG15). Events and interrupts are described further in “Events and Interrupts” on page 4-29. After the low priority interrupt has been forced using the RAISE 15 instruction, RETI can be loaded with a return address that points to user code that can execute until IVG15 is issued. After RETI has been loaded, the RTI instruction can be issued to return from the reset event.
Supervisor Mode

The interrupt handler for IVG15 can be set to jump to the application code starting address. An additional RTI is not required. As a result, the processor remains in Supervisor mode because IPEND[15] remains set. At this point, the processor is servicing the lowest priority interrupt. This ensures that higher priority interrupts can be processed.

Example Code for Supervisor Mode Coming Out of Reset

To remain in Supervisor mode when coming out of the Reset state, use code as shown in Listing 3-2.

Listing 3-2. Staying in Supervisor Mode Coming Out of Reset

```
P0.L = LO(EVT15) ;  /* Point to IVG15 in Event Vector Table */
P0.H = HI(EVT15) ;
P1.L = START ;    /* Point to start of User code */

P1.H = START ;
[P0] = P1 ;    /* Place the address of start code in IVG15 of EVT */

P0.L = LO(IMASK) ;

RO = [P0] ;
R1.L = EVT_IVG15 & 0xFFFF ;

RO = RO | R1 ;
[P0] = RO ;    /* Set (enable) IVG15 bit in Interrupt Mask Register */

RAISE 15 ;    /* Invoke IVG15 interrupt */
P0.L = WAIT_HERE ;
P0.H = WAIT_HERE ;
RETI = P0 ;    /* RETI loaded with return address */
```
Operating Modes and States

RTI : /* Return from Reset Event */
WAIT_HERE : /* Wait here till IVG15 interrupt is serviced */

JUMP WAIT_HERE :

START: /* IVG15 vectors here */
[--SP] = RETI ; /* Enables interrupts and saves return address to stack */

Emulation Mode

The processor enters Emulation mode if Emulation mode is enabled and either of these conditions is met:

- An external emulation event occurs.
- The EMUEXCEPT instruction is issued.

The processor remains in Emulation mode until the emulation service routine executes an RTE instruction. If no interrupts are pending when the RTE instruction executes, the processor switches to User mode. Otherwise, the processor switches to Supervisor mode to service the interrupt.

Emulation mode is the highest priority mode, and the processor has unrestricted access to all system resources.

Idle State

Idle state stops all processor activity at the user’s discretion, usually to conserve power during lulls in activity. No processing occurs during the Idle state. The Idle state is invoked by a sequential IDLE instruction. The IDLE instruction notifies the processor hardware that the Idle state is requested.
Reset State

The processor remains in the Idle state until a peripheral or external device, such as a SPORT or the Real-Time Clock (RTC), generates an interrupt that requires servicing.

In Listing 3-3, core interrupts are disabled and the IDLE instruction is executed. When all the pending processes have completed, the core disables its clocks. Since interrupts are disabled, Idle state can be terminated only by asserting a WAKEUP signal. For more information, see “SIC_IWR Register” on page 4-34. (While not required, an interrupt could also be enabled in conjunction with the WAKEUP signal.)

When the WAKEUP signal is asserted, the processor wakes up, and the STI instruction enables interrupts again.

Example Code for Transition to Idle State

To transition to the Idle state, use code shown in Listing 3-3.

Listing 3-3. Transitioning to Idle State

CLI R0 ; /* disable interrupts */
IDLE ; /* drain pipeline and send core into IDLE state */
STI R0 ; /* re-enable interrupts after wakeup */

Reset State

Reset state initializes the processor logic. During Reset state, application programs and the operating system do not execute. Clocks are stopped while in Reset state.

The processor remains in the Reset state as long as external logic asserts the external RESET signal. Upon deassertion, the processor completes the reset sequence and switches to Supervisor mode, where it executes code found at the reset event vector.
Operating Modes and States

Software in Supervisor or Emulation mode can invoke the Reset state without involving the external \texttt{RESET} signal. This can be done by issuing the Reset version of the \texttt{RAISE} instruction.

Application programs in User mode cannot invoke the Reset state, except through a system call provided by an operating system kernel. Table 3-5 summarizes the state of the processor upon reset.

Table 3-5. Processor State Upon Reset

<table>
<thead>
<tr>
<th>Item</th>
<th>Description of Reset State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Operating Mode</td>
<td>Supervisor mode in reset event, clocks stopped</td>
</tr>
<tr>
<td>Rounding Mode</td>
<td>Unbiased rounding</td>
</tr>
<tr>
<td>Cycle Counters</td>
<td>Disabled, zero</td>
</tr>
<tr>
<td>DAG Registers (I, L, B, M)</td>
<td>Random values (must be cleared at initialization)</td>
</tr>
<tr>
<td>Data and Address Registers</td>
<td>Random values (must be cleared at initialization)</td>
</tr>
<tr>
<td>IPEND, IMASK, ILAT</td>
<td>Cleared, interrupts globally disabled with IPEND bit 4</td>
</tr>
<tr>
<td>CPLBs</td>
<td>Disabled</td>
</tr>
<tr>
<td>L1 Instruction Memory</td>
<td>SRAM (cache disabled)</td>
</tr>
<tr>
<td>L1 Data Memory</td>
<td>SRAM (cache disabled)</td>
</tr>
<tr>
<td>Cache Validity Bits</td>
<td>Invalid</td>
</tr>
<tr>
<td>System</td>
<td></td>
</tr>
<tr>
<td>Booting Methods</td>
<td>Determined by the values of BMODE pins at reset</td>
</tr>
<tr>
<td>MSEL Clock Frequency</td>
<td>Reset value = 10</td>
</tr>
<tr>
<td>PLL Bypass Mode</td>
<td>Disabled</td>
</tr>
<tr>
<td>VCO/Core Clock Ratio</td>
<td>Reset value = 1</td>
</tr>
<tr>
<td>VCO/System Clock Ratio</td>
<td>Reset value = 5</td>
</tr>
<tr>
<td>Peripheral Clocks</td>
<td>Disabled</td>
</tr>
</tbody>
</table>
Table 3-6 describes the five types of resets. Note all resets, except System Software, reset the core.

Table 3-6. Resets

<table>
<thead>
<tr>
<th>Reset</th>
<th>Source</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Reset</td>
<td>The RESET pin causes a hardware reset.</td>
<td>Resets both the core and the peripherals, including the Dynamic Power Management Controller (DPMC). Resets the No Boot on Software Reset bit in SYSCR. For more information, see &quot;SYSCR Register&quot; on page 3-14.</td>
</tr>
<tr>
<td>System Software Reset</td>
<td>Writing b#111 to bits [2:0] in the system MMR SWRST at address 0xFFC00100 causes a System Software reset.</td>
<td>Resets only the peripherals, excluding the RTC (Real-Time Clock) block and most of the DPMC. The DPMC resets only the No Boot on Software Reset bit in SYSCR. Does not reset the core. Does not initiate a boot sequence.</td>
</tr>
<tr>
<td>Watchdog Timer Reset</td>
<td>Programming the watchdog timer appropriately causes a Watchdog Timer reset.</td>
<td>Resets both the core and the peripherals, excluding the RTC block and most of the DPMC. The Software Reset register (SWRST) can be read to determine whether the reset source was the watchdog timer.</td>
</tr>
</tbody>
</table>
Operating Modes and States

Table 3-6. Resets (Cont’d)

<table>
<thead>
<tr>
<th>Reset</th>
<th>Source</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Double-Fault Reset</td>
<td>If the core enters a double-fault state, a reset can be caused by unmasking the Core Double Fault Reset Mask bit in the System Interrupt Controller Interrupt Mask register (SIC_IMASK).</td>
<td>Resets both the core and the peripherals, excluding the RTC block and most of the DPMC. The SWRST register can be read to determine whether the reset source was Core Double Fault.</td>
</tr>
<tr>
<td>Core-Only Software Reset</td>
<td>This reset is caused by executing a RAISE1 instruction or by setting the Software Reset (SYSRST) bit in the core Debug Control register (DBGCTL) via emulation software through the JTAG port. The DBGCTL register is not visible to the memory map.</td>
<td>Resets only the core. The peripherals do not recognize this reset.</td>
</tr>
</tbody>
</table>

Hardware Reset

The processor chip reset is an asynchronous reset event. The RESET input pin must be deasserted to perform a hardware reset. For more information, see the product data sheet.

A hardware-initiated reset results in a system-wide reset that includes both core and peripherals. After the RESET pin is deasserted, the processor ensures that all asynchronous peripherals have recognized and completed a reset. After the reset, the processor transitions into the Boot mode sequence configured by the BMODE state.

The BMODE pins are dedicated mode control pins. No other functions are shared with these pins, and they may be permanently strapped by tying them directly to either VDD or VSS. The pins and the corresponding bits in SYSCR configure the Boot mode that is employed after hardware reset or
System Reset and Powerup

System Software reset. See “Reset Interrupt” on page 4-46, and Table 4-11, “Events That Cause Exceptions,” on page 4-63 for further information.

SYSCR Register

The values sensed from the BMODE pins are latched into the System Reset Configuration register (SYSCR) upon the deassertion of the RESET pin. The values are made available for software access and modification after the hardware reset sequence. Software can modify only the No Boot on Software Reset bit.

The various configuration parameters are distributed to the appropriate destinations from SYSCR. Refer to the Reset and Booting chapter of your Blackfin Processor Hardware Reference for details.

Software Resets and Watchdog Timer

A software reset may be initiated in three ways:

• By the watchdog timer, if appropriately configured
• By setting the System Software Reset field in the Software Reset register (see Figure 3-2 on page 3-16)
• By the RAISE1 instruction

The watchdog timer resets both the core and the peripherals. A System Software reset results in a reset of the peripherals without resetting the core and without initiating a booting sequence.

The System Software reset must be performed while executing from Level 1 memory (either as cache or as SRAM).

When L1 instruction memory is configured as cache, make sure the System Software reset sequence has been read into the cache.
Operating Modes and States

After either the watchdog or System Software reset is initiated, the processor ensures that all asynchronous peripherals have recognized and completed a reset.

For a reset generated by the watchdog timer, the processor transitions into the Boot mode sequence. The Boot mode is configured by the state of the BMODE and the No Boot on Software Reset control bits.

If the No Boot on Software Reset bit in SYSCR is cleared, the reset sequence is determined by the BMODE control bits.

**SWRST Register**

A software reset can be initiated by setting the System Software Reset field in the Software Reset register (SWRST). Bit 15 indicates whether a software reset has occurred since the last time SWRST was read. Bit 14 and Bit 13, respectively, indicate whether the Software Watchdog Timer or a Core Double Fault has generated a software reset. Bits [15:13] are read-only and cleared when the register is read. Bits [3:0] are read/write.

When the BMODE pins are not set to b#00 and the No Boot on Software Reset bit in SYSCR is set, the processor starts executing from the start of on-chip L1 memory. In this configuration, the core begins fetching instructions from the beginning of on-chip L1 memory.

When the BMODE pins are set to b#00 the core begins fetching instructions from address 0x2000 0000 (the beginning of ASYNC Bank 0).
System Reset and Powerup

Core-Only Software Reset

A Core-Only Software reset is initiated by executing the RAISE 1 instruction or by setting the Software Reset (SYSRST) bit in the core Debug Control register (DBGCTL) via emulation software through the JTAG port. (DBGCTL is not visible to the memory map.)

A Core-Only Software reset affects only the state of the core. Note the system resources may be in an undetermined or even unreliable state, depending on the system activity during the reset period.

Core and System Reset

To perform a system and core reset, use the code sequence shown in Listing 3-4.

---

Listing 3-4
Listing 3-4. Core and System Reset

/* Issue soft reset */
P0.L = LO(SWRST) ;
P0.H = HI(SWRST) ;
R0.L = 0x0007 ;
W[P0] = R0 ;
SSYNC ;

/* Clear soft reset */
P0.L = LO(SWRST) ;
P0.H = HI(SWRST) ;
R0.L = 0x0000 ;
W[P0] = R0 ;
SSYNC ;

/* Core reset - forces reboot */
RAISE 1 ;
System Reset and Powerup
This chapter describes the Blackfin processor program sequencing and interrupt processing modules. For information about instructions that control program flow, see Chapter 7, “Program Flow Control.” For information about instructions that control interrupt processing, see Chapter 16, “External Event Management.” Discussion of derivative-specific interrupt sources can be found in the Hardware Reference manual for the specific part.

Introduction

In the processor, the program sequencer controls program flow, constantly providing the address of the next instruction to be executed by other parts of the processor. Program flow in the chip is mostly linear, with the processor executing program instructions sequentially.

The linear flow varies occasionally when the program uses nonsequential program structures, such as those illustrated in Figure 4-1. Nonsequential structures direct the processor to execute an instruction that is not at the next sequential address. These structures include:

- **Loops.** One sequence of instructions executes several times with zero overhead.
- **Subroutines.** The processor temporarily interrupts sequential flow to execute instructions from another part of memory.
- **Jumps.** Program flow transfers permanently to another part of memory.
• **Interrupts and Exceptions.** A runtime event or instruction triggers the execution of a subroutine.

• **Idle.** An instruction causes the processor to stop operating and hold its current state until an interrupt occurs. Then, the processor services the interrupt and continues normal execution.

![Figure 4-1. Program Flow Variations](image)

The sequencer manages execution of these program structures by selecting the address of the next instruction to execute.
The fetched address enters the instruction pipeline, ending with the program counter (PC). The pipeline contains the 32-bit addresses of the instructions currently being fetched, decoded, and executed. The PC couples with the RETn registers, which store return addresses. All addresses generated by the sequencer are 32-bit memory instruction addresses.

To manage events, the event controller handles interrupt and event processing, determines whether an interrupt is masked, and generates the appropriate event vector address.

In addition to providing data addresses, the data address generators (DAGs) can provide instruction addresses for the sequencer’s indirect branches.

The sequencer evaluates conditional instructions and loop termination conditions. The loop registers support nested loops. The memory-mapped registers (MMRs) store information used to implement interrupt service routines.

Figure 4-2 shows the core Program Sequencer module and how it interconnects with the Core Event Controller and the System Event Controller.
Figure 4-2. Program Sequencing and Interrupt Processing Block Diagram
Sequencer Related Registers

Table 4-1 lists the non-memory-mapped registers within the processor that are related to the sequencer. Except for the PC and SEQSTAT registers, all sequencer-related registers are directly readable and writable by move instructions, for example:

SYSCFG = R0 ;
P0 = RETI ;

Manually pushing or popping registers to or from the stack is done using the explicit instructions:

[−−SP] = Rn ; /* for push */
Rn = [SP++] ; /* for pop */

Similarly, all non-memory-mapped sequencer registers can be pushed and popped to or from the system stack:

[−−SP] = CYCLES ;
SYSCFG = [SP++] ;

However, load/store operations and immediate loads are not supported.
Table 4-1. Non-memory-mapped Sequencer Registers

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQSTAT</td>
<td>Sequencer Status register: See “Hardware Errors and Exception Handling” on page 4-58.</td>
</tr>
<tr>
<td>RETX</td>
<td>Return Address registers: See “Events and Interrupts” on page 4-29.</td>
</tr>
<tr>
<td>RETN</td>
<td>Exception Return</td>
</tr>
<tr>
<td>RETI</td>
<td>NMI Return</td>
</tr>
<tr>
<td>RETE</td>
<td>Interrupt Return</td>
</tr>
<tr>
<td>RETS</td>
<td>Emulation Return</td>
</tr>
<tr>
<td>RETS</td>
<td>Subroutine Return</td>
</tr>
<tr>
<td>LC0, LC1</td>
<td>Zero-Overhead Loop registers: See “Hardware Loops” on page 4-21:</td>
</tr>
<tr>
<td>LT0, LT1</td>
<td>Loop Counters</td>
</tr>
<tr>
<td>LB0, LB1</td>
<td>Loop Tops</td>
</tr>
<tr>
<td>LB0, LB1</td>
<td>Loop Bottoms</td>
</tr>
<tr>
<td>FP, SP</td>
<td>Frame Pointer and Stack Pointer: See “Frame and Stack Pointers” on page 5-6</td>
</tr>
<tr>
<td>SYSCFG</td>
<td>System Configuration register: See “SYSCFG Register” on page 21-26</td>
</tr>
<tr>
<td>CYCLES, CYCLES2</td>
<td>Cycle Counters: See “CYCLES and CYCLES2 Registers” on page 21-24</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter. The PC is an embedded register. It is not directly accessible with program instructions.</td>
</tr>
</tbody>
</table>

In addition to these central sequencer registers, there is a set of memory-mapped registers that interact closely with the program sequencer. For information about the interrupt control registers, see “Events and Interrupts” on page 4-29. Although the registers of the Core Event Controller are memory-mapped, they still connect to the same 32-bit Register Access Bus (RAB) and perform in the same way. Registers of the System Interrupt Controller connect to the Peripheral Access Bus (PAB) which resides in the SCLK domain. On some derivatives the PAB bus is 16 bits wide; on others it is 32 bits wide. For debug and test registers see Chapter 21, “Debug.”
Program Sequencer

Instruction Pipeline

The program sequencer determines the next instruction address by examining both the current instruction being executed and the current state of the processor. If no conditions require otherwise, the processor executes instructions from memory in sequential order by incrementing the lookahead address.

The processor has a ten-stage instruction pipeline, shown in Table 4-2.

Table 4-2. Stages of Instruction Pipeline

<table>
<thead>
<tr>
<th>Pipeline Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch 1 (IF1)</td>
<td>Issue instruction address to IAB bus, start compare tag of instruction cache</td>
</tr>
<tr>
<td>Instruction Fetch 2 (IF2)</td>
<td>Wait for instruction data</td>
</tr>
<tr>
<td>Instruction Fetch 3 (IF3)</td>
<td>Read from IDB bus and align instruction</td>
</tr>
<tr>
<td>Instruction Decode (DEC)</td>
<td>Decode instructions</td>
</tr>
<tr>
<td>Address Calculation (AC)</td>
<td>Calculation of data addresses and branch target address</td>
</tr>
<tr>
<td>Data Fetch 1 (DF1)</td>
<td>Issue data address to DA0 and DA1 bus, start compare tag of data cache</td>
</tr>
<tr>
<td>Data Fetch 2 (DF2)</td>
<td>Read register files</td>
</tr>
<tr>
<td>Execute 1 (EX1)</td>
<td>Read data from LD0 and LD1 bus, start multiply and video instructions</td>
</tr>
<tr>
<td>Execute 2 (EX2)</td>
<td>Execute/Complete instructions (shift, add, logic, etc.)</td>
</tr>
<tr>
<td>Write Back (WB)</td>
<td>Writes back to register files, SD bus, and pointer updates (also referred to as the &quot;commit&quot; stage)</td>
</tr>
</tbody>
</table>
Instruction Pipeline

Figure 4-3 shows a diagram of the pipeline.

![Processor Pipeline Diagram](image)

The instruction fetch and branch logic generates 32-bit fetch addresses for the Instruction Memory Unit. The Instruction Alignment Unit returns instructions and their width information at the end of the IF3 stage.

For each instruction type (16, 32, or 64 bits), the Instruction Alignment Unit ensures that the alignment buffers have enough valid instructions to be able to provide an instruction every cycle. Since the instructions can be 16, 32, or 64 bits wide, the Instruction Alignment Unit may not need to fetch an instruction from the cache every cycle. For example, for a series of 16-bit instructions, the Instruction Alignment Unit gets an instruction from the Instruction Memory Unit once in four cycles. The alignment logic requests the next instruction address based on the status of the alignment buffers. The sequencer responds by generating the next fetch address in the next cycle, provided there is no change of flow.

The sequencer holds the fetch address until it receives a request from the alignment logic or until a change of flow occurs. The sequencer always increments the previous fetch address by 8 (the next 8 bytes). If a change of flow occurs, such as a branch or an interrupt, data in the Instruction Alignment Unit is invalidated. The sequencer decodes and distributes instruction data to the appropriate locations such as the register file and data memory.

The Execution Unit contains two 16-bit multipliers, two 40-bit ALUs, two 40-bit accumulators, one 40-bit shifter, a video unit (which adds 8-bit ALU support), and an 8-entry 32-bit Data Register File.
Program Sequencer

Register file reads occur in the DF2 pipeline stage (for operands).

Register file writes occur in the WB stage (for stores). The multipliers and the video units are active in the EX1 stage, and the ALUs and shifter are active in the EX2 stage. The accumulators are written at the end of the EX2 stage.

The program sequencer also controls stalling and invalidating the instructions in the pipeline. Multi-cycle instruction stalls occur between the IF3 and DEC stages. DAG and sequencer stalls occur between the DEC and AC stages. Computation and register file stalls occur between the DF2 and EX1 stages. Data memory stalls occur between the EX1 and EX2 stages.

The sequencer ensures that the pipeline is fully interlocked and that all the data hazards are hidden from the programmer.

Multi-cycle instructions behave as multiple single-cycle instructions being issued from the decoder over several clock cycles. For example, the Push Multiple or Pop Multiple instruction can push or pop from 1 to 14 DREGS and/or PREGS, and the instruction remains in the decode stage for a number of clock cycles equal to the number of registers being accessed.

Multi-issue instructions are 64 bits in length and consist of one 32-bit instruction and two 16-bit instructions. All three instructions execute in the same amount of time as the slowest of the three.

Any nonsequential program flow can potentially decrease the processor’s instruction throughput. Nonsequential program operations include:

- Jumps
- Subroutine calls and returns
- Interrupts and returns
- Loops
One type of nonsequential program flow that the sequencer supports is branching. A branch occurs when a **JUMP** or **CALL** instruction begins execution at a new location other than the next sequential address. For descriptions of how to use the **JUMP** and **CALL** instructions, see Chapter 7, “Program Flow Control.” Briefly:

- A **JUMP** or a **CALL** instruction transfers program flow to another memory location. The difference between a **JUMP** and a **CALL** is that a **CALL** automatically loads the return address into the **RETS** register. The return address is the next sequential address after the **CALL** instruction. This push makes the address available for the **CALL** instruction’s matching return instruction, allowing easy return from the subroutine.

- A return instruction causes the sequencer to fetch the instruction at the return address, which is stored in the **RETS** register (for subroutine returns). The types of return instructions include: return from subroutine (**RTS**), return from interrupt (**RTI**), return from exception (**RTX**), return from emulation (**RTE**), and return from nonmaskable interrupt (**RTN**). Each return type has its own register for holding the return address.

- A **JUMP** instruction can be conditional, depending on the status of the **CC** bit of the **ASTAT** register. These instructions are immediate and may not be delayed. The program sequencer can evaluate the **CC** status bit to decide whether to execute a branch. If no condition is specified, the branch is always taken.

- Conditional **JUMP** instructions use static branch prediction to reduce the branch latency caused by the length of the pipeline.
Program Sequencer

Branches can be direct or indirect. A direct branch address is determined solely by the instruction word (for example, `JUMP 0x30`), while an indirect branch gets its address from the contents of a DAG register (for example, `JUMP(P3)`).

All types of `JUMP` and `CALL` can be PC-relative. The indirect `JUMP` and `CALL` can be absolute or PC-relative.

Direct Short and Long Jumps

The sequencer supports both short and long jumps. The target of the branch is a PC-relative address from the location of the instruction, plus an offset. The PC-relative offset for the short jump is a 13-bit immediate value that must be a multiple of two (bit 0 must be a 0). The 13-bit value gives an effective dynamic range of –4096 to +4094 bytes.

The PC-relative offset for the long jump is a 25-bit immediate value that must also be a multiple of two (bit 0 must be a 0). The 25-bit value gives an effective dynamic range of –16,777,216 to +16,777,214 bytes.

If, at the time of writing the program, the destination is known to be less than a 13-bit offset from the current PC value, then the `JUMP.S 0xnnnn` instruction may be used. If the destination requires more than a 13-bit offset, then the `JUMP.L 0xnnnnnnnn` instruction must be used. If the destination offset is unknown and development tools must evaluate the offset, then use the instruction `JUMP 0xnnnnnnnn`. Upon disassembly, the instruction is replaced by the appropriate `JUMP.S` or `JUMP.L` instruction.

Rather than hard coding jump target addresses, use symbolic addresses in assembly source files. Symbolic addresses are called labels and are marked by a trailing colon. See the Visual DSP++ Assembler and Preprocessor Manual for details.
Branches

JUMP mylabel;
/* skip any code placed here */
mylabel:
/* continue to fetch and execute instruction here */

Direct Call

The CALL instruction is a branch instruction that copies the address of the instruction which would have executed next (had the CALL instruction not executed) into the RETS register. The direct CALL instruction has a 25-bit, PC-relative offset that must be a multiple of two (bit 0 must be a 0). The 25-bit value gives an effective dynamic range of –16,777,216 to +16,777,214 bytes. A direct CALL instruction is always a 4-byte instruction.

Indirect Branch and Call

The indirect JUMP and CALL instructions get their destination address from a data address generator (DAG) P-register. For the CALL instruction, the RETS register is loaded with the address of the instruction which would have executed next in the absence of the CALL instruction.

For example:

JUMP (P3);
CALL (P0);

To load a P-register with a symbolic target label you may use one of the following syntax styles. The syntax may differ in various assembly tools sets.

Modern style:

P4.H = HI(mytarget);
P4.L = LO(mytarget);
JUMP (P4);
Program Sequencer

mytarget:
     /* continue here */

Legacy style:

P4.H = mytarget;
P4.L = mytarget;
JUMP (P4);
mytarget:
     /* continue here */

PC-Relative Indirect Branch and Call

The PC-relative indirect JUMP and CALL instructions use the contents of a P-register as an offset to the branch target. For the CALL instruction, the RETS register is loaded with the address of the instruction which would have executed next (had the CALL instruction not executed).

For example:

JUMP (PC + P3);
CALL (PC + P0);

Subroutines

Subroutines are code sequences that are invoked by a CALL instruction. Assuming the stack pointer SP has been initialized properly, a typical scenario could look like the following:

/* parent function */
R0 = 0x1234 (Z); /* pass a parameter */
CALL myfunction;
/* continue here after the call */
[P0] = R0; /* save return value */
JUMP somewhereelse;
myfunction: /* subroutine label */
Due to the syntax of the multiple-push, multiple-pop instructions, often the upper R- and P-registers are used for local purposes, while lower registers pass the parameters. See the "Address Arithmetic Unit" chapter for more details on stack management.

The CALL instruction not only redirects the program flow to the myfunction routine, it also writes the return address into the RETS register. The RETS register holds the address where program execution resumes after the RTS instruction executes. In the example this is the location that holds the [P0]=R0; instruction.

The return address is not passed to any stack in the background. Rather, the RETS register functions as single-entry hardware stack. This scheme enables "leaf functions" (subroutines that do not contain further CALL instructions) to execute with less possible overhead, as no bus transfers need to be performed.

If a subroutine calls other functions, it must temporarily save the content of the RETS register explicitly. Most likely this is performed by stack operations as shown below.

/* parent function */
CALL function_a;
/* continue here after the call */
JUMP somewhereelse;
function_a: /* subroutine label */
[--SP] = (R7:7, P5:5); /* optional multiple push instruction */
Program Sequencer

[-SP] = RETS; /* save RETS onto stack */
CALL function_b; /* call further subroutines */
CALL function_c;
RETS = [SP++]; /* restore RETS */
(R7:7, P5:5) = [SP++]; /* optional multiple pop instruction */
RTS; /* return from subroutine */
function_a.end: /* closing subroutine label */
function_b:
/* do something */
RTS;
function_b.end:
function_c:
/* do something else */
RTS;
function_c.end:

Stack Variables and Parameter Passing

Many subroutines require input arguments from the calling function and need to return their results. Often, this is accomplished by project-wide conventions, that certain core registers are used for passing arguments, where others return the result. It is also recommended that assembly programs meet the conventions used by the C/C++ compiler. See the VisualDSP++ C/C++ Compiler and Library Manual for details.

Extensive arguments are typically passed over the stack rather than by registers. The following example passes and returns two 32-bit arguments:

_parent:
...
R0 = 1;
R1 = 3;
[-SP] = R0;
[-SP] = R1;
CALL _sub:

R1 = [SP++]; /* R1 = 4 */
R0 = [SP++]; /* R0 = 2 */
...
parent.end:

_sub:
[-SP] = FP; /* save frame pointer */
FP = SP; /* new frame */
[-SP] = (R7:5); /* multiple push */

R6 = [FP+4]; /* R6 = 3 */
R7 = [FP+8]; /* R7 = 1 */

R5 = R6 + R7; /* calculate anything */
R6 = R6 - R7;

[FP+4] = R5; /* R5 = 4 */
[FP+8] = R6; /* R6 = 2 */

(R7:5) = [SP++]; /* multiple pop */
FP = [SP++]; /* restore frame pointer */
RTS;
_sub.end:

Since the stack pointer SP is modified inside the subroutine for local stack operations, the frame pointer FP is used to save the original state of SP. Because the 32-bit frame pointer itself must be pushed onto the stack first, the FP is four bytes off the original SP value.
Program Sequencer

The Blackfin instruction set features a pair of instructions that provides cleaner and more efficient functionality than the above example: the `LINK` and `UNLINK` instructions. These multi-cycle instructions perform multiple operations that can be best explained by the equivalent code sequences:

Table 4-3. Link and Unlink Code Sequencer

<table>
<thead>
<tr>
<th>LINK n;</th>
<th>UNLINK;</th>
</tr>
</thead>
<tbody>
<tr>
<td>[--SP] = RETS;</td>
<td>SP = FP;</td>
</tr>
<tr>
<td>[--SP] = FP;</td>
<td>FP = [SP++];</td>
</tr>
<tr>
<td>FP = SP;</td>
<td>RETS = [SP++];</td>
</tr>
<tr>
<td>SP += -n;</td>
<td>SP = FP;</td>
</tr>
<tr>
<td></td>
<td>FP = [SP++];</td>
</tr>
<tr>
<td></td>
<td>SP = FP;</td>
</tr>
<tr>
<td></td>
<td>FP = [SP++];</td>
</tr>
<tr>
<td></td>
<td>RETS = [SP++];</td>
</tr>
</tbody>
</table>

The following subroutine does the same job as the one above, but it also saves the RETS register to enable nested subroutine calls. Therefore, the value stored to FP is 8 bytes off the original SP value. Since no local frame is required, the `LINK` instruction gets the parameter “0”.

```assembly
_sub2:
    LINK 0;
    [--SP] = (R7:5):

    R6 = [FP+8]; /* R6 = 3 */
    R7 = [FP+12]; /* R7 = 1 */

    R5 = R6 + R7;
    R6 = R6 - R7;

    [FP+8] = R5; /* R5 = 4 */
    [FP+12] = R6; /* R6 = 2 */

    (R7:5) = [SP++];
    UNLINK;
    RTS;
    _sub2.end:
```
Branches

If subroutines require local, private, and temporary variables beyond the capabilities of core registers, it is a good idea to place these variables on the stack as well. The \texttt{LINK} instruction takes a parameter that specifies the size of the stack memory required for this local purpose. The following example provides two local 32-bit variables and initializes them to zero when the routine is entered:

\begin{verbatim}
_sub3:
  LINK 8;
  [--SP] = (R7:0, P5:0):

  R7 = 0 (Z);
  [FP-4] = R7;
  [FP-8] = R7;
  ...
  (R7:0, P5:0) = [SP++];
  UNLINK;
  RTS;
_endsub3:
\end{verbatim}

For more information on the \texttt{LINK} and \texttt{UNLINK} instructions, see “\texttt{LINK, UNLINK}” on page 10-17.

Condition Code Flag

The processor supports a Condition Code (CC) flag bit, which is used to resolve the direction of a branch. This flag may be accessed eight ways:

- A conditional branch is resolved by the value in CC.
- A Data register value may be copied into CC, and the value in CC may be copied to a Data register.
- The \texttt{BITTST} instruction accesses the CC flag.
• A status flag may be copied into $CC$, and the value in $CC$ may be copied to a status flag.

• The $CC$ flag bit may be set to the result of a Pointer register comparison.

• The $CC$ flag bit may be set to the result of a Data register comparison.

• Some shifter instructions (rotate or $BXOR$) use $CC$ as a portion of the shift operand/result.

• Test and set instructions can set and clear the $CC$ bit.

These eight ways of accessing the $CC$ bit are used to control program flow. The branch is explicitly separated from the instruction that sets the arithmetic flags. A single bit resides in the instruction encoding that specifies the interpretation for the value of $CC$. The interpretation is to “branch on true” or “branch on false.”

The comparison operations have the form $CC = expr$ where $expr$ involves a pair of registers of the same type (for example, Data registers or Pointer registers, or a single register and a small immediate constant). The small immediate constant is a 3-bit ($-4$ through $3$) signed number for signed comparisons and a 3-bit ($0$ through $7$) unsigned number for unsigned comparisons.

The sense of $CC$ is determined by equal ($==$), less than ($<$), and less than or equal to ($<=$). There are also bit test operations that test whether a bit in a 32-bit R-register is set.

**Conditional Branches**

The sequencer supports conditional branches. Conditional branches are $JUMP$ instructions whose execution branches or continues linearly, depending on the value of the $CC$ bit. The target of the branch is a PC-relative address from the location of the instruction, plus an offset. The
Branches

PC-relative offset is an 11-bit immediate value that must be a multiple of two (bit 0 must be a 0). This gives an effective dynamic range of –1024 to +1022 bytes.

For example, the following instruction tests the \( \text{CC} \) flag and, if it is positive, jumps to a location identified by the label `dest_address`:

\[
\text{IF CC JUMP dest_address ;}
\]

Take care when conditional branches are followed by load operations. For more information, see “Load/Store Operation” on page 6-66.

Conditional Register Move

Register moves can be performed depending on whether the value of the \( \text{CC} \) flag is true or false (1 or 0). In some cases, using this instruction instead of a branch eliminates the cycles lost because of the branch. These conditional moves can be done between any R- or P-registers (including SP and FP).

Example code:

\[
\text{IF CC RO = PO ;}
\]

Branch Prediction

The sequencer supports static branch prediction to accelerate execution of conditional branches. These branches are executed based on the state of the \( \text{CC} \) bit.

In the EX2 stage, the sequencer compares the actual \( \text{CC} \) bit value to the predicted value. If the value was mispredicted, the branch is corrected, and the correct address is available for the WB stage of the pipeline.
Program Sequencer

The branch latency for conditional branches is as follows.

- If prediction was “not to take branch,” and branch was actually not taken: 0 CCLK cycles.
- If prediction was “not to take branch,” and branch was actually taken: 8 CCLK cycles.
- If prediction was “to take branch,” and branch was actually taken: 4 CCLK cycles.
- If prediction was “to take branch,” and branch was actually not taken: 8 CCLK cycles.

For all unconditional branches, the branch target address computed in the AC stage of the pipeline is sent to the Instruction Fetch Address bus at the beginning of the DF1 stage. All unconditional branches have a latency of 4 CCLK cycles.

Consider the example in Table 4-4.

Table 4-4. Branch Prediction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>If CC JUMP dest (bp)</td>
<td>This instruction tests the CC flag, and if it is set, jumps to a location, identified by the label, dest. If the CC flag is set, the branch is correctly predicted and the branch latency is reduced. Otherwise, the branch is incorrectly predicted and the branch latency increases.</td>
</tr>
</tbody>
</table>

Hardware Loops

The sequencer supports a mechanism of zero-overhead looping. The sequencer contains two loop units, each containing three registers. Each loop unit has a Loop Top register (LT0, LT1), a Loop Bottom register (LB0, LB1), and a Loop Count register (LC0, LC1).
Hardware Loops

Two sets of zero-overhead loop registers implement loops, using hardware counters instead of software instructions to evaluate loop conditions. After evaluation, processing branches to a new target address. Both sets of registers include the Loop Counter (LC), Loop Top (LT), and Loop Bottom (LB) registers.

Table 4-11 describes the 32-bit loop register sets.

Table 4-5. Loop Registers

<table>
<thead>
<tr>
<th>Registers</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC0, LC1</td>
<td>Loop Counters</td>
<td>Maintains a count of the remaining iterations of the loop</td>
</tr>
<tr>
<td>LT0, LT1</td>
<td>Loop Tops</td>
<td>Holds the address of the first instruction within a loop</td>
</tr>
<tr>
<td>LB0, LB1</td>
<td>Loop Bottoms</td>
<td>Holds the address of the last instruction of the loop</td>
</tr>
</tbody>
</table>

When an instruction at address $X$ is executed, and $X$ matches the contents of LB0, then the next instruction executed will be from the address in LT0. In other words, when $PC == LB0$, then an implicit jump to LT0 is executed.

A loopback only occurs when the count is greater than or equal to 2. If the count is nonzero, then the count is decremented by 1. For example, consider the case of a loop with two iterations. At the beginning, the count is 2. Upon reaching the first loop end, the count is decremented to 1 and the program flow jumps back to the top of the loop (to execute a second time). Upon reaching the end of the loop again, the count is decremented to 0, but no loopback occurs (because the body of the loop has already been executed twice).

The LSETUP instruction can be used to load all three registers of a loop unit at once. Each loop register can also be loaded individually with a register transfer, but this incurs a significant overhead if the loop count is nonzero (the loop is active) at the time of the transfer.

The following code example shows a loop that contains two instructions and iterates 32 times.
Listing 4-1. Loop Example

P5 = 0x20 :
LSETUP ( lp_start, lp_end ) LCO = P5 :
lp_start: R5 = R0 + R1(ns) || R2 = [P2++] || R3 = [I1++] :
lp_end: R5 = R5 + R2 :

When executing an LSETUP instruction, the program sequencer loads the address of the loop’s last instruction into \( \text{LBx} \) and the address of the loop’s first instruction into \( \text{LTx} \). The top and bottom addresses of the loop are computed as PC-relative addresses from the LSETUP instruction, plus an offset. In each case, the offset value is added to the location of the LSETUP instruction.

The \( \text{LCO} \) and \( \text{LC1} \) registers are unsigned 32-bit registers, each supporting \( 2^{32} - 1 \) iterations through the loop.

\( \square \) When \( \text{LCx} = 0 \), the loop is disabled, and a single pass of the code executes. If the loop counter is derived from a variable with a range that may include zero, it is recommended to guard the loop against the zero case.

P5 = [P4]:
CC = P5 == 0:
IF CC JUMP lp_skip:
LSETUP (lp_start, lp_end) LCO = P5:
lp_start: ...
lp_end: ...
lp_skip: /* first instruction outside the loop */
Hardware Loops

The processor supports a four-location instruction loop buffer that reduces instruction fetches while in loops. If the loop code contains four or fewer instructions, then no fetches to instruction memory are necessary for any number of loop iterations, because the instructions are stored locally. The loop buffer effectively eliminates the instruction fetch time in loops with more than four instructions by allowing fetches to take place while instructions in the loop buffer are being executed.

A four-cycle latency occurs on the first loopback when the LSETUP specifies a nonzero start offset (lp_start). Therefore, zero start offsets are preferred, that is, the lp_start label is next the LSETUP instruction.

The processor has no restrictions regarding which instructions can occur in a loop end position. Branches and calls are allowed in that position.

Two-Dimensional Loops

The processor features two loop units. Each provides its own set of loop registers.

- LC[1:0] – the Loop Count registers
- LT[1:0] – the Loop Top address registers
- LB[1:0] – the Loop Bottom address registers

Table 4-6. Loop Registers

<table>
<thead>
<tr>
<th>First/Last Address of the Loop</th>
<th>PC-Relative Offset Used to Compute the Loop Start Address</th>
<th>Effective Range of the Loop Start Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top / First</td>
<td>5-bit signed immediate; must be a multiple of 2.</td>
<td>0 to 30 bytes away from LSETUP instruction.</td>
</tr>
<tr>
<td>Bottom / Last</td>
<td>11-bit signed immediate; must be a multiple of 2.</td>
<td>0 to 2046 bytes away from LSETUP instruction (the defined loop can be 2046 bytes long).</td>
</tr>
</tbody>
</table>
Therefore, two-dimensional loops are supported directly in hardware, consisting of an outer loop and a nested inner loop.

The outer loop is always represented by loop unit 0 (LC0, LT0, LB0) while loop unit 1 (LC1, LT1, LB1) manages the inner loop.

To enable the two nested loops to end at the same instruction (LB1 equals LB0), loop unit 1 is assigned higher priority than loop unit 0. A loopback caused by loop unit 1 on a particular instruction (PC==LB1, LC1>=2) will prevent loop unit 0 from looping back on that same instruction, even if the address matches. Loop unit 0 is allowed to loop back only after the loop count 1 is exhausted. The following example shows a two-dimensional loop.

```c
#define M 32
#define N 1024
P4 = M (Z);
P5 = N-1 (Z);
LSETUP ( lpo_start, lpo_end ) LC0 = P4;
lpo_start: R7 = 0;
    MNOP || R2 = [I0++] || R3 = [I1++] ;
lpi_start: R5 = R2 + R3 (NS) || R2 = [I0] || R3 = [I1++] ;
lpi_end: R7 = R5 + R7 (NS) || [I0++] = R5;
    R5 = R2 + R3;
    R7 = R5 + R7 (NS) || [I0++] = R5;
lpo_end: [I2++] = R7;
```

The example processes an M by N data structure. The inner loop is unrolled and passes only N-1 times. The outer loop is not unrolled and still provides room for optimization.
Hardware Loops

Loop Unrolling

Typical DSP algorithms are coded for speed rather than for small code size. Especially when fetching data from circular buffers, loops are often unrolled in order to pass only N-1 times. The initial data fetch is executed before the loop is entered. Similarly, the final calculations are done after the loop terminates, for example:

```c
#define N 1024

void global_setup:
    I0.H = 0xFF80; I0.L = 0x0000; B0 = I0; L0 = N*2 (Z);
    I1.H = 0xFF90; I1.L = 0x0000; B1 = I1; L1 = N*2 (Z);
    P5 = N-1 (Z);

algorithm:
    A0 = 0 || R0.H = W[I0++] || R1.L = W[I1++];
    LSETUP (lp,lp) LC0 = P5;
    lp: A0+= R0.H * R1.L || R0.H = W[I0++] || R1.L = W[I1++];
        A0+= R0.H * R1.L;
```

This technique has the advantage that data is fetched exactly N times and the I-Registers have their initial value after processing. The “algorithm” sequence can be executed multiple times without any need to initialize DAG-Registers again.
Saving and Resuming Loops

Normally, loops can process and terminate without regard to system-level concepts. Even if interrupted by interrupts or exceptions, no special care is needed. There are, however, a few situations that require special attention—whenever a loop is interrupted by events that require the loop resources themselves, that is:

- If the loop is interrupted by an interrupt service routine that also contains a hardware loop and requires the same loop unit
- If the loop is interrupted by a preemptive task switch
- If the loop contains a CALL instruction that invokes an unknown subroutine that may have local loops

In scenarios like these, the loop environment can be saved and restored by pushing and popping the loop registers. For example, to save Loop Unit 0 onto the system stack, use this code:

\[
\begin{align*}
[\cdot \cdot \cdot SP] &= LCO; \\
[\cdot \cdot \cdot SP] &= LBO; \\
[\cdot \cdot \cdot SP] &= LTO;
\end{align*}
\]

To restore Loop Unit 0 from system stack, use:

\[
\begin{align*}
LTO &= [SP++]; \\
LBO &= [SP++]; \\
LCO &= [SP++];
\end{align*}
\]

It is obvious that writes or pops to the loop registers cause some internal side effects to re-initialize the loop hardware properly. The hardware does not force the user to save and restore all three loop registers, as there might be cases where saving one or two of them is sufficient. Consequently, every pop instruction in the example above may require the loop hardware to re-initialize again. This takes multiple cycles, as the loop buffers must also be prefilled again.
Hardware Loops

To avoid unnecessary penalty cycles, the loop hardware follows these rules:

- Restoring LC0 and LC1 registers always re-initializes the loop hardware and causes a ten-cycle “replay” penalty.
- Restoring LT0, LT1, LB0, and LB1 performs in a single cycle if the respective loop counter register is zero.
- If LCx is non-zero, every write to the LTx and LBx registers also attempts to re-initialize the loop hardware and causes a ten-cycle penalty.

In terms of performance, there is a difference depending on the order that the loop registers are popped. For best performance, restore the LCx registers last. Furthermore, it is recommended that interrupt service routines and global subroutines that contain hardware loops terminate their local loops cleanly, that is, do not artificially break the loops and do not execute return instructions within their loops. This guarantees that the LCx registers are 0 when LTx and LBx registers are popped.

Example Code for Using Hardware Loops in an ISR

The following code shows the optimal method of saving and restoring when using hardware loops in an interrupt service routine.

Listing 4-2. Saving and Restoring With Hardware Loops

```
lhandler:
  <Save other registers here>
  [--SP] = LC0; /* save loop 0 */
  [--SP] = LB0;
  [--SP] = LT0;
  <Handler code here>
```
/* If the handler uses loop 0, it is a good idea to have it leave LC0 equal to zero at the end. Normally, this will happen naturally as a loop is fully executed. If LC0 == 0, then LT0 and LB0 restores will not incur additional cycles. If LC0 != 0 when the following pops happen, each pop will incur a ten-cycle "replay" penalty. Popping or writing LC0 always incurs the penalty. */

LT0 = [SP++];
LB0 = [SP++];
LC0 = [SP++]; /* This will cause a “replay,” that is, a ten-cycle refetch. */

<Restore other registers here>

RTI;

Events and Interrupts

The Event Controller of the processor manages five types of activities or events:

- Emulation
- Reset
- Nonmaskable interrupts (NMI)
- Exceptions
- Interrupts

Note the word event describes all five types of activities. The Event Controller manages fifteen different events in all: Emulation, Reset, NMI, Exception, and eleven Interrupts.
Events and Interrupts

An interrupt is an event that changes normal processor instruction flow and is asynchronous to program flow. In contrast, an exception is a software initiated event whose effects are synchronous to program flow.

The event system is nested and prioritized. Consequently, several service routines may be active at any time, and a low priority event may be pre-empted by one of higher priority.

The processor employs a two-level event control mechanism. The processor System Interrupt Controller (SIC) works with the Core Event Controller (CEC) to prioritize and control all system interrupts. The SIC provides mapping between the many peripheral interrupt sources and the prioritized general-purpose interrupt inputs of the core. This mapping is programmable, and individual interrupt sources can be masked in the SIC.

The CEC supports nine general-purpose interrupts ($IVG_7$ – $IVG_{15}$) in addition to the dedicated interrupt and exception events that are described in Table 4-7. It is recommended that the two lowest priority interrupts ($IVG_{14}$ and $IVG_{15}$) be reserved for software interrupt handlers, leaving seven prioritized interrupt inputs ($IVG_7$ – $IVG_{13}$) to support the system. Refer to the product data sheet for the default system interrupt mapping.

Table 4-7. Core Event Mapping

<table>
<thead>
<tr>
<th>Event Source</th>
<th>Core Event Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Events</td>
<td></td>
</tr>
<tr>
<td>Emulation (highest priority)</td>
<td>EMU</td>
</tr>
<tr>
<td>Reset</td>
<td>RST</td>
</tr>
<tr>
<td>NMI</td>
<td>NMI</td>
</tr>
<tr>
<td>Exception</td>
<td>EVX</td>
</tr>
<tr>
<td>Reserved</td>
<td>–</td>
</tr>
<tr>
<td>Hardware Error</td>
<td>IVHW</td>
</tr>
<tr>
<td>Core Timer</td>
<td>IVTMR</td>
</tr>
</tbody>
</table>

Table 4-7. Core Event Mapping
Note the System Interrupt to Core Event mappings shown are the default values at reset and can be changed by software.

**System Interrupt Processing**

Referring to Figure 4-4 on page 4-33, note when an interrupt (Interrupt A) is generated by an interrupt-enabled peripheral:

1. SIC_ISR logs the request and keeps track of system interrupts that are asserted but not yet serviced (that is, an interrupt service routine hasn’t yet cleared the interrupt).

2. SIC_IWR checks to see if it should wake up the core from an idled state based on this interrupt request.

3. SIC_IMASK masks off or enables interrupts from peripherals at the system level. If Interrupt A is not masked, the request proceeds to Step 4.

4. The SIC_IARx registers, which map the peripheral interrupts to a smaller set of general-purpose core interrupts (IVG7 – IVG15), determine the core priority of Interrupt A.

5. ILAT adds Interrupt A to its log of interrupts latched by the core but not yet actively being serviced.

6. IMASK masks off or enables events of different core priorities. If the IVGx event corresponding to Interrupt A is not masked, the process proceeds to Step 7.

7. The Event Vector Table (EVT) is accessed to look up the appropriate vector for Interrupt A’s interrupt service routine (ISR).
8. When the event vector for Interrupt A has entered the core pipeline, the appropriate IPEND bit is set, which clears the respective ILAT bit. Thus, IPEND tracks all pending interrupts, as well as those being presently serviced.

9. When the interrupt service routine (ISR) for Interrupt A has been executed, the RTI instruction clears the appropriate IPEND bit. However, the relevant SIC_ISR bit is not cleared unless the interrupt service routine clears the mechanism that generated Interrupt A, or if the process of servicing the interrupt clears this bit.

It should be noted that emulation, reset, NMI, and exception events, as well as hardware error (IVHW) and core timer (IVTMR) interrupt requests, enter the interrupt processing chain at the ILAT level and are not affected by the system-level interrupt registers (SIC_IWR, SIC_ISR, SIC_IMASK, SIC_IARx).

If multiple interrupt sources share a single core interrupt, then the interrupt service routine (ISR) must identify the peripheral that generated the interrupt. The ISR may then need to interrogate the peripheral to determine the appropriate action to take.
System Peripheral Interrupts

The processor system has numerous peripherals, which therefore require many supporting interrupts.

The peripheral interrupt structure of the processor is flexible. By default upon reset, multiple peripheral interrupts share a single, general-purpose interrupt in the core, as shown in the System Interrupt Appendix of the Blackfin Processor Hardware Reference for your part.

An interrupt service routine that supports multiple interrupt sources must interrogate the appropriate system memory mapped registers (MMRs) to determine which peripheral generated the interrupt.
Events and Interrupts

If the default assignments shown in the System Interrupt Appendix of the Blackfin Processor Hardware Reference for your part are acceptable, then interrupt initialization involves only:

- Initialization of the core Event Vector Table (EVT) vector address entries
- Initialization of the IMASK register
- Unmasking the specific peripheral interrupts in SIC IMASK that the system requires

SIC_IWR Register

The System Interrupt Wakeup-Enable register (SIC_IWR) provides the mapping between the peripheral interrupt source and the Dynamic Power Management Controller (DPMC). Any of the peripherals can be configured to wake up the core from its idled state to process the interrupt, simply by enabling the appropriate bit in the System Interrupt Wakeup-enable register (SIC_IWR, refer to the System Interrupt Appendix of the Blackfin Processor Hardware Reference for your part). If a peripheral interrupt source is enabled in SIC_IWR and the core is idled, the interrupt causes the DPMC to initiate the core wakeup sequence in order to process the interrupt. Note this mode of operation may add latency to interrupt processing, depending on the power control state. For further discussion of power modes and the idled state of the core, see the Dynamic Power Management chapter of the Blackfin Processor Hardware Reference for your part.

By default, as shown in the System Interrupt Appendix of the Blackfin Processor Hardware Reference for your part, all interrupts generate a wakeup request to the core. However, for some applications it may be desirable to disable this function for some peripherals, such as for a SPORTx Transmit Interrupt.
Program Sequencer

The \texttt{SIC\_IWR} register has no effect unless the core is idled. The bits in this register correspond to those of the System Interrupt Mask (\texttt{SIC\_IMASK}) and Interrupt Status (\texttt{SIC\_ISR}) registers.

After reset, all valid bits of this register are set to 1, enabling the wakeup function for all interrupts that are not masked. Before enabling interrupts, configure this register in the reset initialization sequence. The \texttt{SIC\_IWR} register can be read from or written to at any time. To prevent spurious or lost interrupt activity, this register should be written to only when all peripheral interrupts are disabled.

\textbf{Note} the wakeup function is independent of the interrupt mask function. If an interrupt source is enabled in \texttt{SIC\_IWR} but masked off in \texttt{SIC\_IMASK}, the core wakes up if it is idled, but it does not generate an interrupt.

For a listing of the default System Interrupt Wakeup-Enable register settings, refer to the System Interrupt Appendix of the \textit{Blackfin Processor Hardware Reference} for your part.

\textbf{SIC\_ISR Register}

The System Interrupt Controller (SIC) includes a read-only status register, the System Interrupt Status register (\texttt{SIC\_ISR}), shown in the System Interrupt Appendix of the \textit{Blackfin Processor Hardware Reference} for your part. Each valid bit in this register corresponds to one of the peripheral interrupt sources. The bit is set when the SIC detects the interrupt is asserted and cleared when the SIC detects that the peripheral interrupt input has been deasserted. Note for some peripherals, such as programmable flag asynchronous input interrupts, many cycles of latency may pass from the time an interrupt service routine initiates the clearing of the interrupt (usually by writing a system MMR) to the time the SIC senses that the interrupt has been deasserted.
Events and Interrupts

Depending on how interrupt sources map to the general-purpose interrupt inputs of the core, the interrupt service routine may have to interrogate multiple interrupt status bits to determine the source of the interrupt. One of the first instructions executed in an interrupt service routine should read SIC_ISR to determine whether more than one of the peripherals sharing the input has asserted its interrupt output. The service routine should fully process all pending, shared interrupts before executing the RTI, which enables further interrupt generation on that interrupt input.

When an interrupt’s service routine is finished, the RTI instruction clears the appropriate bit in the IPEND register. However, the relevant SIC_ISR bit is not cleared unless the service routine clears the mechanism that generated the interrupt.

Many systems need relatively few interrupt-enabled peripherals, allowing each peripheral to map to a unique core priority level. In these designs, SIC_ISR will seldom, if ever, need to be interrogated.

The SIC_ISR register is not affected by the state of the System Interrupt Mask register (SIC_IMASK) and can be read at any time. Writes to the SIC_ISR register have no effect on its contents.

SIC_IMASK Register

The System Interrupt Mask register (SIC_IMASK, shown in the System Interrupt Appendix of the Blackfin Processor Hardware Reference for your part) allows masking of any peripheral interrupt source at the System Interrupt Controller (SIC), independently of whether it is enabled at the peripheral itself.

A reset forces the contents of SIC_IMASK to all 0s to mask off all peripheral interrupts. Writing a 1 to a bit location turns off the mask and enables the interrupt.
Program Sequencer

Although this register can be read from or written to at any time (in Supervisor mode), it should be configured in the reset initialization sequence before enabling interrupts.

System Interrupt Assignment Registers (SIC_IARx)

The relative priority of peripheral interrupts can be set by mapping the peripheral interrupt to the appropriate general-purpose interrupt level in the core. The mapping is controlled by the System Interrupt Assignment register settings, as detailed in the System Interrupt Appendix of the Blackfin Processor Hardware Reference for your part. If more than one interrupt source is mapped to the same interrupt, they are logically ORed, with no hardware prioritization. Software can prioritize the interrupt processing as required for a particular system application.

For general-purpose interrupts with multiple peripheral interrupts assigned to them, take special care to ensure that software correctly processes all pending interrupts sharing that input. Software is responsible for prioritizing the shared interrupts.

These registers can be read from or written to at any time in Supervisor mode. It is advisable, however, to configure them in the Reset interrupt service routine before enabling interrupts. To prevent spurious or lost interrupt activity, these registers should be written to only when all peripheral interrupts are disabled.
Core Event Controller Registers

The Event Controller uses three MMRs to coordinate pending event requests. In each of these MMRs, the 16 lower bits correspond to the 16 event levels (for example, bit 0 corresponds to “Emulator mode”). The registers are:

- **IMASK** - interrupt mask
- **ILAT** - interrupt latch
- **IPEND** - interrupts pending

These three registers are accessible in Supervisor mode only.

**IMASK Register**

The Core Interrupt Mask register (**IMASK**) indicates which interrupt levels are allowed to be taken. The **IMASK** register may be read and written in Supervisor mode. Bits [15:5] have significance; bits [4:0] are hard-coded to 1 and events of these levels are always enabled. If **IMASK[N]** = 1 and **ILAT[N]** = 1, then interrupt **N** will be taken if a higher priority is not already recognized. If **IMASK[N]** = 0, and **ILAT[N]** gets set by interrupt **N**, the interrupt will not be taken, and **ILAT[N]** will remain set.
Program Sequencer

Core Interrupt Mask Register (IMASK)
For all bits, 0 - Interrupt masked, 1 - Interrupt enabled

![Figure 4-5. Core Interrupt Mask Register](image)

**ILAT Register**

Each bit in the Core Interrupt Latch register (ILAT) indicates that the corresponding event is latched, but not yet accepted into the processor (see Figure 4-6). The bit is reset before the first instruction in the corresponding ISR is executed. At the point the interrupt is accepted, ILAT[N] will be cleared and IPEND[N] will be set simultaneously. The ILAT register can be read in Supervisor mode. Writes to ILAT are used to clear bits only (in Supervisor mode). To clear bit N from ILAT, first make sure that IMASK[N] == 0, and then write ILAT[N] = 1. This write functionality to ILAT is provided for cases where latched interrupt requests need to be cleared (cancelled) instead of serviced.

The RAISE instruction can be used to set ILAT[15] through ILAT[5], and also ILAT[2] or ILAT[1].

Only the JTAG TRST pin can clear ILAT[0].
Events and Interrupts

Core Interrupt Latch Register (ILAT)
Reset value for bit 0 is emulator-dependent. For all bits, 0 - Interrupt not latched, 1 - Interrupt latched

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>EMU (Emulation) - RO</td>
</tr>
<tr>
<td>14</td>
<td>RST (Reset) - RO</td>
</tr>
<tr>
<td>13</td>
<td>NMI (Nonmaskable Interrupt) - RO</td>
</tr>
<tr>
<td>12</td>
<td>EVX (Exception) - RO</td>
</tr>
<tr>
<td>11</td>
<td>IVHW (Hardware Error)</td>
</tr>
<tr>
<td>10</td>
<td>IVTMR (Core Timer)</td>
</tr>
<tr>
<td>9</td>
<td>IVG7</td>
</tr>
<tr>
<td>8</td>
<td>IVG8</td>
</tr>
<tr>
<td>7</td>
<td>IVG15</td>
</tr>
<tr>
<td>6</td>
<td>IVG14</td>
</tr>
<tr>
<td>5</td>
<td>IVG13</td>
</tr>
<tr>
<td>4</td>
<td>IVG12</td>
</tr>
<tr>
<td>3</td>
<td>IVG11</td>
</tr>
<tr>
<td>2</td>
<td>IVG10</td>
</tr>
<tr>
<td>1</td>
<td>IVG9</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4-6. Core Interrupt Latch Register

IPEND Register

The Core Interrupt Pending register (IPEND) keeps track of all currently nested interrupts (see Figure 4-7). Each bit in IPEND indicates that the corresponding interrupt is currently active or nested at some level. It may be read in Supervisor mode, but not written. The IPEND[4] bit is used by the Event Controller to temporarily disable interrupts on entry and exit to an interrupt service routine.

When an event is processed, the corresponding bit in IPEND is set. The least significant bit in IPEND that is currently set indicates the interrupt that is currently being serviced. At any given time, IPEND holds the current status of all nested events.
Program Sequencer

Event Vector Table

The Event Vector Table (EVT) is a hardware table with sixteen entries that are each 32 bits wide. The EVT contains an entry for each possible core event. Entries are accessed as MMRs, and each entry can be programmed at reset with the corresponding vector address for the interrupt service routine. When an event occurs, instruction fetch starts at the address location in the EVT entry for that event.

The processor architecture allows unique addresses to be programmed into each of the interrupt vectors; that is, interrupt vectors are not determined by a fixed offset from an interrupt vector table base address. This approach minimizes latency by not requiring a long jump from the vector table to the actual ISR code.
Events and Interrupts

Table 4-8 lists events by priority. Each event has a corresponding bit in the event state registers ILAT, IMASK, and IPEND.

Table 4-8. Core Event Vector Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Event Class</th>
<th>Event Vector Register</th>
<th>MMR Location</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMU</td>
<td>Emulation</td>
<td>EVT0</td>
<td>0xFFE0 2000</td>
<td>Highest priority. Vector address is provided by JTAG.</td>
</tr>
<tr>
<td>RST</td>
<td>Reset</td>
<td>EVT1</td>
<td>0xFFE0 2004</td>
<td></td>
</tr>
<tr>
<td>NMI</td>
<td>NMI</td>
<td>EVT2</td>
<td>0xFFE0 2008</td>
<td></td>
</tr>
<tr>
<td>EVX</td>
<td>Exception</td>
<td>EVT3</td>
<td>0xFFE0 200C</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>Reserved</td>
<td>EVT4</td>
<td>0xFFE0 2010</td>
<td>Reserved vector</td>
</tr>
<tr>
<td>IVHW</td>
<td>Hardware Error</td>
<td>EVT5</td>
<td>0xFFE0 2014</td>
<td></td>
</tr>
<tr>
<td>IVTMR</td>
<td>Core Timer</td>
<td>EVT6</td>
<td>0xFFE0 2018</td>
<td></td>
</tr>
<tr>
<td>IVG7</td>
<td>Interrupt 7</td>
<td>EVT7</td>
<td>0xFFE0 201C</td>
<td>System interrupt</td>
</tr>
<tr>
<td>IVG8</td>
<td>Interrupt 8</td>
<td>EVT8</td>
<td>0xFFE0 2020</td>
<td>System interrupt</td>
</tr>
<tr>
<td>IVG9</td>
<td>Interrupt 9</td>
<td>EVT9</td>
<td>0xFFE0 2024</td>
<td>System interrupt</td>
</tr>
<tr>
<td>IVG10</td>
<td>Interrupt 10</td>
<td>EVT10</td>
<td>0xFFE0 2028</td>
<td>System interrupt</td>
</tr>
<tr>
<td>IVG11</td>
<td>Interrupt 11</td>
<td>EVT11</td>
<td>0xFFE0 202C</td>
<td>System interrupt</td>
</tr>
<tr>
<td>IVG12</td>
<td>Interrupt 12</td>
<td>EVT12</td>
<td>0xFFE0 2030</td>
<td>System interrupt</td>
</tr>
<tr>
<td>IVG13</td>
<td>Interrupt 13</td>
<td>EVT13</td>
<td>0xFFE0 2034</td>
<td>System interrupt</td>
</tr>
<tr>
<td>IVG14</td>
<td>Interrupt 14</td>
<td>EVT14</td>
<td>0xFFE0 2038</td>
<td>System interrupt</td>
</tr>
<tr>
<td>IVG15</td>
<td>Interrupt 15</td>
<td>EVT15</td>
<td>0xFFE0 203C</td>
<td>Software interrupt</td>
</tr>
</tbody>
</table>

Return Registers and Instructions

Similarly to the RETS register controlled by CALL and RTS instructions, interrupts and exceptions also use single-entry hardware stack registers. If an interrupt is serviced, the program sequencer saves the return address.
Program Sequencer

into the RETI register prior to jumping to the event vector. A typical interrupt service routine terminates with an RTI instruction that instructs the sequencer to reload the Program Counter, PC, from the RETI register. The following example shows a simple interrupt service routine.

```
isr:
    [~SP] = (R7:0, P5:0); /* push core registers */
    [~SP] = ASTAT; /* push arithmetic status */
    /* place core of service routine here */
    ASTAT = [SP++]; /* pop arithmetic status */
    (R7:0, P5:0) = [SP++]; /* pop core registers */
    RTI; /* return from interrupt */
isr.end:
```

There is no need to manage the RETI register when interrupt nesting is not enabled. If however, nesting is enabled and the respective service routine must be interruptible by an interrupt of higher priority, the RETI register must be saved, most likely onto the stack.

Instructions that access the RETI register do have an implicit site effect—reading the RETI register enables interrupt nesting. Writing to it disables nesting again. This enables the service routine to break itself down into interruptible and non-interruptible sections. For example:

```
isr:
    [~SP] = (R7:0, P5:0); /* push core registers */
    [~SP] = ASTAT; /* push arithmetic status */
    /* place critical or atomic code here */
    [~SP] = RETI; /* enable nesting */
    /* place core of service routine here */
    RETI = [SP++]; /* disable nesting */
    /* more critical or atomic instructions */
    ASTAT = [SP++]; /* pop arithmetic status */
    (R7:0, P5:0) = [SP++]; /* pop core registers */
    RTI; /* return from interrupt */
isr.end:
```
Events and Interrupts

If there is not a need for non-interruptible code inside the service routine, it is good programming practice to enable nesting immediately. This avoids unnecessary delay to high priority interrupt routines. For example:

```assembly
isr:
[--SP] = RETI; /* enable nesting */
[--SP] = (R7:0, P5:0); /* push core registers */
[--SP] = ASTAT; /* push arithmetic status */
/* place core of service routine here */
ASTAT = [SP++]; /* pop arithmetic status */
(R7:0, P5:0) = [SP++]; /* pop core registers */
RTI = [SP++]; /* disable nesting */
RTI; /* return from interrupt */
isr.end:
```

See “Nesting of Interrupts” on page 4-51 for more details on interrupt nesting.

Emulation Events, NMI, and Exceptions use a technique similar to “normal” interrupts. However, they have their own return register and return instruction counterparts. Table 4-9 provides an overview.

Table 4-9. Return Registers and Instructions

<table>
<thead>
<tr>
<th>Name</th>
<th>Event Class</th>
<th>Return Register</th>
<th>Return Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMU</td>
<td>Emulation</td>
<td>RETE</td>
<td>RTE</td>
</tr>
<tr>
<td>RST</td>
<td>Reset</td>
<td>RETI</td>
<td>RTI</td>
</tr>
<tr>
<td>NMI</td>
<td>NMI</td>
<td>RETN</td>
<td>RTN</td>
</tr>
<tr>
<td>EVX</td>
<td>Exception</td>
<td>RETX</td>
<td>RTX</td>
</tr>
<tr>
<td>Reserved</td>
<td>Reserved</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IVHW</td>
<td>Hardware Error</td>
<td>RETI</td>
<td>RTI</td>
</tr>
<tr>
<td>IVTMR</td>
<td>Core Timer</td>
<td>RETI</td>
<td>RTI</td>
</tr>
<tr>
<td>IVG7</td>
<td>Interrupt 7</td>
<td>RETI</td>
<td>RTI</td>
</tr>
</tbody>
</table>
Program Sequencer

Table 4-9. Return Registers and Instructions (Cont’d)

<table>
<thead>
<tr>
<th>Name</th>
<th>Event Class</th>
<th>Return Register</th>
<th>Return Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>IVG8</td>
<td>Interrupt 8</td>
<td>RETI</td>
<td>RTI</td>
</tr>
<tr>
<td>IVG9</td>
<td>Interrupt 9</td>
<td>RETI</td>
<td>RTI</td>
</tr>
<tr>
<td>IVG10</td>
<td>Interrupt 10</td>
<td>RETI</td>
<td>RTI</td>
</tr>
<tr>
<td>IVG11</td>
<td>Interrupt 11</td>
<td>RETI</td>
<td>RTI</td>
</tr>
<tr>
<td>IVG12</td>
<td>Interrupt 12</td>
<td>RETI</td>
<td>RTI</td>
</tr>
<tr>
<td>IVG13</td>
<td>Interrupt 13</td>
<td>RETI</td>
<td>RTI</td>
</tr>
<tr>
<td>IVG14</td>
<td>Interrupt 14</td>
<td>RETI</td>
<td>RTI</td>
</tr>
<tr>
<td>IVG15</td>
<td>Interrupt 15</td>
<td>RETI</td>
<td>RTI</td>
</tr>
</tbody>
</table>

Executing RTX, RTN, or RTE in a Lower Priority Event

Instructions RTX, RTN, and RTE are designed to return from an exception, NMI, or emulator event, respectively. Do not use them to return from a lower priority event. To return from an interrupt, use the RTI instruction. Failure to use the correct instruction may produce unintended results.

In the case of RTX, bit IPEND[3] is cleared. In the case of RTI, the bit of the highest priority interrupt in IPEND is cleared.

Emulation Interrupt

An emulation event causes the processor to enter Emulation mode, where instructions are read from the JTAG interface. It is the highest priority interrupt to the core.

For detailed information about emulation, see the Blackfin Processor Debug chapter of the Blackfin Processor Hardware Reference for your part.
Events and Interrupts

Reset Interrupt

The reset interrupt (\texttt{RST}) can be initiated via the \texttt{RESET} pin or through expiration of the watchdog timer. This location differs from that of other interrupts in that its content is read-only. Writes to this address change the register but do not change where the processor vectors upon reset. The processor always vectors to the reset vector address upon reset. For more information, see “Reset State” on page 3-10.

The core has an output that indicates that a double fault has occurred. This is a nonrecoverable state. The system (via the \texttt{SWRST} register) can be programmed to send a reset request if a double fault condition is detected. Subsequently, the reset request forces a system reset for core and peripherals.

The reset vector is determined by the processor system. It points to the start of the on-chip boot ROM, or to the start of external asynchronous memory, depending on the state of the \texttt{BMODE} pins.

NMI (Nonmaskable Interrupt)

The NMI entry is reserved for a nonmaskable interrupt, which can be generated by the Watchdog timer or by the NMI input signal to the processor. An example of an event that requires immediate processor attention, and thus is appropriate as an NMI, is a powerdown warning.

⚠️ If an exception occurs in an event handler that is already servicing an Exception, NMI, Reset, or Emulation event, this will trigger a double fault condition, and the address of the excepting instruction will be written to \texttt{RETX}.

If unused, the \texttt{NMI} pin should always be pulled to its deasserted state. On some derivatives, the \texttt{NMI} input is active high and on some it is active low. Please refer to the specific data sheet for your processor.
Exceptions

Exceptions are discussed in “Hardware Errors and Exception Handling” on page 4-58.

Hardware Error Interrupt

Hardware Errors are discussed in “Hardware Errors and Exception Handling” on page 4-58.

Core Timer Interrupt

The Core Timer Interrupt (IVTMR) is triggered when the core timer value reaches zero. For more information about the core timer, see the Hardware Reference Manual for your processor.

General-purpose Interrupts (IVG7-IVG15)

General-purpose interrupts are used for any event that requires processor attention. For instance, a DMA controller may use them to signal the end of a data transmission, or a serial communications device may use them to signal transmission errors.

Software can also trigger general-purpose interrupts by using the RAISE instruction. The RAISE instruction forces events for interrupts IVG15-IVG7, IVTMR, IVHW, NMI, and RST, but not for exceptions and emulation (EVX and EMU, respectively).

It is recommended to reserve the two lowest priority interrupts (IVG15 and IVG14) for software interrupt handlers.

For system interrupts available on specific Blackfin processors, see the Hardware Reference Manual for that processor.
Interrupt Processing

The following sections describe interrupt processing.

Global Enabling/Disabling of Interrupts

General-purpose interrupts can be globally disabled with the `CLI Dreg` instruction and re-enabled with the `STI Dreg` instruction, both of which are only available in Supervisor mode. Reset, NMI, emulation, and exception events cannot be globally disabled. Globally disabling interrupts clears `IMASK[15:5]` after saving `IMASK`’s current state.

```
CLI R5;    /* save IMASK to R5 and mask all */
/* place critical instructions here */
STI R5;    /* restore IMASK from R5 again */
```

See “Enable Interrupts” and “Disable Interrupts” in Chapter 16, “External Event Management.”

When multiple instructions need to be atomic or are too time-critical to be delayed by an interrupt, disable the general-purpose interrupts, but be sure to re-enable them at the conclusion of the code sequence.

Servicing Interrupts

The Core Event Controller (CEC) has a single interrupt queueing element per event—a bit in the `ILAT` register. The appropriate `ILAT` bit is set when an interrupt rising edge is detected (which takes two core clock cycles) and cleared when the respective `IPEND` register bit is set. The `IPEND` bit indicates that the event vector has entered the core pipeline. At this point, the CEC recognizes and queues the next rising edge event on the corresponding interrupt input. The minimum latency from the rising edge transition of the general-purpose interrupt to the `IPEND` output assertion is three core clock cycles. However, the latency can be much higher, depending on the core’s activity level and state.
To determine when to service an interrupt, the controller logically ANDs the three quantities in \textit{ILAT}, \textit{IMASK}, and the current processor priority level.

Servicing the highest priority interrupt involves these actions:

1. The interrupt vector in the Event Vector Table (EVT) becomes the next fetch address.

   On an interrupt, most instructions currently in the pipeline are aborted. On a service exception, all instructions after the excepting instruction are aborted. On an error exception, the excepting instruction and all instructions after it are aborted.

2. The return address is saved in the appropriate return register.

   The return register is \texttt{RETI} for interrupts, \texttt{RETX} for exceptions, \texttt{RETN} for NMIs, and \texttt{RETE} for debug emulation. The return address is the address of the instruction after the last instruction executed from normal program flow.

3. Processor mode is set to the level of the event taken.

   If the event is an NMI, exception, or interrupt, the processor mode is Supervisor. If the event is an emulation exception, the processor mode is Emulation.

4. Before the first instruction starts execution, the corresponding interrupt bit in \textit{ILAT} is cleared and the corresponding bit in \textit{IPEND} is set.

   Bit \texttt{IPEND[4]} is also set to disable all interrupts until the return address in \texttt{RETI} is saved.
Interrupt Processing

Software Interrupts

Software cannot set bits of the ILAT register directly, as writes to ILAT cause write-1-to-clear (W1C) operation. Instead, use the RAISE instruction to set individual ILAT bits by software. It safely sets any of the ILAT bits without affecting the rest of the register.

RAISE 1; /* fire reset interrupt request */

The RAISE instruction must not be used to fire emulation events or exceptions, which are managed by the related EMUEXCPT and EXCPT instructions. For details, see Chapter 16, “External Event Management.”

Often, the RAISE instruction is executed in interrupt service routines to degrade the interrupt priority. This enables less urgent parts of the service routine to be interrupted even by low priority interrupts.

isr7: /* service routine for IVG7 */
...
/* execute high priority instructions here */
/* handshake with signalling peripheral */
RAISE 14;
RTI;
isr7.end:
isr14: /* service routine for IVG14 */
...
/* further process event initiated by IVG7 */
RTI;
isr14.end:

The example above may read data from any receiving interface, post it to a queue, and let the lower priority service routine process the queue after the isr7 routine returns. Since IVG15 is used for normal program execution in non-multi-tasking system, IVG14 is often dedicated to software interrupt purposes.
“Example Code for an Exception Handler” on page 4-68 uses the same principle to handle an exception with normal interrupt priority level.

Nesting of Interrupts

Interrupts are handled either with or without nesting, individually. For more information, see “Return Registers and Instructions” on page 4-42.

Non-nested Interrupts

If interrupts do not require nesting, all interrupts are disabled during the interrupt service routine. Note, however, that emulation, NMI, and exceptions are still accepted by the system.

When the system does not need to support nested interrupts, there is no need to store the return address held in RETI. Only the portion of the machine state used in the interrupt service routine must be saved in the Supervisor stack. To return from a non-nested interrupt service routine, only the RTI instruction must be executed, because the return address is already held in the RETI register.

Figure 4-8 shows an example of interrupt handling where interrupts are globally disabled for the entire interrupt service routine.

Nested Interrupts

If interrupts require nesting, the return address to the interrupted point in the original interrupt service routine must be explicitly saved and subsequently restored when execution of the nested interrupt service routine has completed. The first instruction in an interrupt service routine that supports nesting must save the return address currently held in RETI by pushing it onto the Supervisor stack (SP = RETI). This clears the global interrupt disable bit IPEND[4], enabling interrupts. Next, all registers that are modified by the interrupt service routine are saved onto the
Interrupt Processing

Supervisor stack. Processor state is stored in the Supervisor stack, not in the User stack. Hence, the instructions to push RETI (\[-\text{SP}\] = RETI) and pop RETI (RETI = [SP++]) use the Supervisor stack.

---

**Figure 4-8. Non-nested Interrupt Handling**

Supervisor stack. Processor state is stored in the Supervisor stack, not in the User stack. Hence, the instructions to push RETI (\[-\text{SP}\] = RETI) and pop RETI (RETI = [SP++]) use the Supervisor stack.
Program Sequencer

Figure 4-9 illustrates that by pushing RETI onto the stack, interrupts can be re-enabled during an interrupt service routine, resulting in a short duration where interrupts are globally disabled.

Example Prolog Code for Nested Interrupt Service Routine

Listing 4-3. Prolog Code for Nested ISR

/* Prolog code for nested interrupt service routine. Push return address in RETI into Supervisor stack, ensuring that interrupts are back on. Until now, interrupts have been suspended. */
ISR:
[-SP] = RETI ; /* Enables interrupts and saves return address to stack */
[-SP] = ASTAT ;

<table>
<thead>
<tr>
<th>CYCLE</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>
<th>10</th>
<th>m+1</th>
<th>m+2</th>
<th>m+3</th>
<th>m+4</th>
<th>m+5</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF1</td>
<td>A9</td>
<td>A10</td>
<td>PUSH</td>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
<td>I6</td>
<td>RE</td>
<td>A3</td>
<td>A4</td>
<td>A5</td>
<td>A6</td>
<td>A7</td>
</tr>
<tr>
<td>IF2</td>
<td>A8</td>
<td>A9</td>
<td>PUSH</td>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
<td>I6</td>
<td>RE</td>
<td>A3</td>
<td>A4</td>
<td>A5</td>
<td>A6</td>
<td>A7</td>
</tr>
<tr>
<td>IF3</td>
<td>A7</td>
<td>A8</td>
<td>PUSH</td>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
<td>I6</td>
<td>RE</td>
<td>A3</td>
<td>A4</td>
<td>A5</td>
<td>A6</td>
<td>A7</td>
</tr>
<tr>
<td>DEC</td>
<td>A6</td>
<td>A7</td>
<td>PUSH</td>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
<td>I6</td>
<td>RE</td>
<td>A3</td>
<td>A4</td>
<td>A5</td>
<td>A6</td>
<td>A7</td>
</tr>
<tr>
<td>AC</td>
<td>A5</td>
<td>A6</td>
<td>PUSH</td>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
<td>I6</td>
<td>RE</td>
<td>A3</td>
<td>A4</td>
<td>A5</td>
<td>A6</td>
<td>A7</td>
</tr>
<tr>
<td>DF1</td>
<td>A4</td>
<td>A5</td>
<td>PUSH</td>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
<td>I6</td>
<td>RETI</td>
<td>POP</td>
<td>RTI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DF2</td>
<td>A3</td>
<td>A4</td>
<td>PUSH</td>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
<td>I6</td>
<td>RETI</td>
<td>POP</td>
<td>RTI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX1</td>
<td>A2</td>
<td>A3</td>
<td>POP</td>
<td>I1n</td>
<td>I2n</td>
<td>I3n</td>
<td>I4n</td>
<td>I5n</td>
<td>I6n</td>
<td>POP</td>
<td>RTI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX2</td>
<td>A1</td>
<td>A2</td>
<td>POP</td>
<td>I1n</td>
<td>I2n</td>
<td>I3n</td>
<td>I4n</td>
<td>I5n</td>
<td>I6n</td>
<td>POP</td>
<td>RTI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>A0</td>
<td>A1</td>
<td>POP</td>
<td>I1n</td>
<td>I2n</td>
<td>I3n</td>
<td>I4n</td>
<td>I5n</td>
<td>I6n</td>
<td>POP</td>
<td>RTI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PIPELINE STAGE**

**CYCLE 1:** INTERRUPT IS LATCHED. ALL POSSIBLE INTERRUPT SOURCES DETERMINED.
CYCLE 2: INTERRUPT IS PRIORITIZED.
CYCLE 3: ALL INSTRUCTIONS ABOVE A2 ARE KILLED. A2 IS KILLED IF IT IS AN RTI OR CLI INSTRUCTION. ISR STARTING ADDRESS LOOKUP OCCURS.
CYCLE 4: I0 (INSTRUCTION AT START OF ISR) ENTERS PIPELINE. ASSUME IT IS A PUSH RETI INSTRUCTION (TO ENABLE NESTING).
CYCLE 10: WHEN PUSH REACHES DF2 STAGE, INTERRUPTS ARE RE-ENABLED.
CYCLE M+1: WHEN THE POP RETI INSTRUCTION REACHES THE DF2 STAGE, INTERRUPTS ARE DISABLED.
CYCLE M+5: WHEN RTI REACHES THE WB STAGE, INTERRUPTS ARE RE-ENABLED.
Interrupt Processing

[---SP] = FP ;
[--- SP] = (R7:0, P5:0) ;
/* Body of service routine. Note none of the processor resources
(accumulators, DAGs, loop counters and bounds) have been saved.
It is assumed this interrupt service routine does not use the
processor resources. */

Example Epilog Code for Nested Interrupt Service Routine

Listing 4-4. Epilog Code for Nested ISR

/* Epilog code for nested interrupt service routine.
Restore ASTAT, Data and Pointer registers. Popping RETI from
Supervisor stack ensures that interrupts are suspended between
load of return address and RTI. */
(R7:0, P5:0) = [SP++] ;
FP    = [SP++] ;
ASTAT = [SP++] ;
RETI  = [SP++] ;
/* Execute RTI, which jumps to return address, re-enables inter-
upts, and switches to User mode if this is the last nested
interrupt in service. */
RTI;

The RTI instruction causes the return from an interrupt. The return
address is popped into the RETI register from the stack, an action that sus-
pends interrupts from the time that RETI is restored until RTI finishes
executing. The suspension of interrupts prevents a subsequent interrupt
from corrupting the RETI register.

Next, the RTI instruction clears the highest priority bit that is currently set
in IPEND. The processor then jumps to the address pointed to by the value
in the RETI register and re-enables interrupts by clearing IPEND[4].

4-54   ADSP-BF53x/BF56x Blackfin Processor Programming Reference
Logging of Nested Interrupt Requests

The System Interrupt Controller (SIC) detects level-sensitive interrupt requests from the peripherals. The Core Event Controller (CEC) provides edge-sensitive detection for its general-purpose interrupts (IVG7-IVG15). Consequently, the SIC generates a synchronous interrupt pulse to the CEC and then waits for interrupt acknowledgement from the CEC. When the interrupt has been acknowledged by the core (via assertion of the appropriate IPEND output), the SIC generates another synchronous interrupt pulse to the CEC if the peripheral interrupt is still asserted. This way, the system does not lose peripheral interrupt requests that occur during servicing of another interrupt.

Multiple interrupt sources can map to a single core processor general-purpose interrupt. Because of this, multiple pulse assertions from the SIC can occur simultaneously, before, or during interrupt processing for an interrupt event that is already detected on this interrupt input. For a shared interrupt, the IPEND interrupt acknowledge mechanism described above re-enables all shared interrupts. If any of the shared interrupt sources are still asserted, at least one pulse is again generated by the SIC. The Interrupt Status registers indicate the current state of the shared interrupt sources.

Self-Nesting of Core Interrupts

Interrupts that are “self-nested” can be interrupted by events at the same priority level. When the SNEN bit of the SYSCFG register is set, self-nesting of core interrupts is supported. Self-nesting is supported for any interrupt level generated with the RAISE instruction, as well as for core level interrupts.

As an example, assume that the SNEN bit is set and the processor is servicing an interrupt generated by the RAISE 14; instruction. Once the RETI register has been saved to the stack within the service routine, a second RAISE 14; instruction would allow the processor to service the second interrupt.
Interrupt Processing

Self-nesting is not supported for system level peripheral interrupts such as the SPORT or SPI.

The SYSCFG register is discussed in “SYSCFG Register” on page 21-26.

Additional Usability Issues

The following sections describe additional usability issues.

Allocating the System Stack

The software stack model for processing exceptions implies that the Supervisor stack must never generate an exception while the exception handler is saving its state. However, if the Supervisor stack grows past a CPLB entry or SRAM block, it may, in fact, generate an exception.

To guarantee that the Supervisor stack never generates an exception—never overflows past a CPLB entry or SRAM block while executing the exception handler—calculate the maximum space that all interrupt service routines and the exception handler occupy while they are active, and then allocate this amount of SRAM memory.

Latency in Servicing Events

In some processor architectures, if instructions are executed from external memory and an interrupt occurs while the instruction fetch operation is underway, then the interrupt is held off from being serviced until the current fetch operation has completed. Consider a processor operating at 300 MHz and executing code from external memory with 100 ns access times. Depending on when the interrupt occurs in the instruction fetch operation, the interrupt service routine may be held off for around 30 instruction clock cycles. When cache line fill operations are taken into account, the interrupt service routine could be held off for many hundreds of cycles.
In order for high priority interrupts to be serviced with the least latency possible, the processor allows any high latency fill operation to be completed at the system level, while an interrupt service routine executes from L1 memory. See Figure 4-10.

Figure 4-10. Minimizing Latency in Servicing an ISR

If an instruction load operation misses the L1 instruction cache and generates a high latency line fill operation, then when an interrupt occurs, it is not held off until the fill has completed. Instead, the processor executes the interrupt service routine in its new context, and the cache fill operation completes in the background.
Hardware Errors and Exception Handling

Note the interrupt service routine must reside in L1 cache or SRAM memory and must not generate a cache miss, an L2 memory access, or a peripheral access, as the processor is already busy completing the original cache line fill operation. If a load or store operation is executed in the interrupt service routine requiring one of these accesses, then the interrupt service routine is held off while the original external access is completed, before initiating the new load or store.

If the interrupt service routine finishes execution before the load operation has completed, then the processor continues to stall, waiting for the fill to complete.

This same behavior is also exhibited for stalls involving reads of slow data memory or peripherals.

Writes to slow memory generally do not show this behavior, as the writes are deemed to be single cycle, being immediately transferred to the write buffer for subsequent execution.

For detailed information about cache and memory structures, see Chapter 6, “Memory.”

Hardware Errors and Exception Handling

The following sections describe hardware errors and exception handling.
Program Sequencer

SEQSTAT Register

The Sequencer Status register (SEQSTAT) contains information about the current state of the sequencer as well as diagnostic information from the last event. SEQSTAT is a read-only register and is accessible only in Supervisor mode.

Sequencer Status Register (SEQSTAT)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>EXCAUSE[5:0]</td>
</tr>
<tr>
<td></td>
<td>Holds information about the last executed exception. See Table 4-11.</td>
</tr>
<tr>
<td>24</td>
<td>HWERRCAUSE[4:2]</td>
</tr>
<tr>
<td></td>
<td>Holds cause of last hardware error generated by the core. Hardware errors trigger interrupt number 5 (IVHW). See Table 4-10.</td>
</tr>
<tr>
<td>16</td>
<td>SFTRESET</td>
</tr>
<tr>
<td></td>
<td>0 - Last core reset was not a reset triggered by software</td>
</tr>
<tr>
<td></td>
<td>1 - Last core reset was a reset triggered by software, rather than a hardware powerup reset</td>
</tr>
<tr>
<td>8</td>
<td>HWERRCAUSE[1:0]</td>
</tr>
<tr>
<td></td>
<td>Holds cause of last hardware error generated by the core. Hardware errors trigger interrupt number 5 (IVHW). See Table 4-10.</td>
</tr>
</tbody>
</table>

Figure 4-11. Sequencer Status Register

Hardware Error Interrupt

The Hardware Error Interrupt indicates a hardware error or system malfunction. Hardware errors occur when logic external to the core, such as a memory bus controller, is unable to complete a data transfer (read or write) and asserts the core’s error input signal. Such hardware errors invoke the Hardware Error Interrupt (interrupt IVHW in the Event Vector Table (EVT) and ILAT, IMASK, and IPEND registers). The Hardware Error
Hardware Errors and Exception Handling

Interrupt service routine can then read the cause of the error from the 5-bit \texttt{HWERRCAUSE} field appearing in the Sequencer Status register (\texttt{SEQSTAT}) and respond accordingly.

The Hardware Error Interrupt is generated by:

- Bus parity errors
- Internal error conditions within the core, such as Performance Monitor overflow
- Peripheral errors
- Bus timeout errors

The list of supported hardware conditions, with their related \texttt{HWERRCAUSE} codes, appears in Table 4-10. The bit code for the most recent error appears in the \texttt{HWERRCAUSE} field. If multiple hardware errors occur simultaneously, only the last one can be recognized and serviced. The core does not support prioritizing, pipelining, or queuing multiple error codes. The Hardware Error Interrupt remains active as long as any of the error conditions remain active.
Program Sequencer

Table 4-10. Hardware Conditions Causing Hardware Error Interrupts

<table>
<thead>
<tr>
<th>Hardware Condition</th>
<th>HWERRCAUSE (Binary)</th>
<th>HWERRCAUSE (Hexadecimal)</th>
<th>Notes / Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>System MMR Error</td>
<td>0b00010</td>
<td>0x02</td>
<td>An error can occur if an invalid System MMR location is accessed, if a 32-bit register is accessed with a 16-bit instruction, or if a 16-bit register is accessed with a 32-bit instruction.</td>
</tr>
<tr>
<td>External Memory Addressing Error</td>
<td>0b00011</td>
<td>0x03</td>
<td></td>
</tr>
<tr>
<td>Performance Monitor Overflow</td>
<td>0b10010</td>
<td>0x12</td>
<td></td>
</tr>
<tr>
<td>RAISE 5 instruction</td>
<td>0b11000</td>
<td>0x18</td>
<td>Software issued a RAISE 5 instruction to invoke the Hardware Error Interrupt (IVHW).</td>
</tr>
<tr>
<td>Reserved</td>
<td>All other bit combinations</td>
<td>All other values.</td>
<td></td>
</tr>
</tbody>
</table>

Exceptions

Exceptions are synchronous to the instruction stream. In other words, a particular instruction causes an exception when it attempts to finish execution. No instructions after the offending instruction are executed before the exception handler takes effect.

Many of the exceptions are memory related. For example, an exception is given when a misaligned access is attempted, or when a cacheability protection lookaside buffer (CPLB) miss or protection violation occurs. Exceptions are also given when illegal instructions or illegal combinations of registers are executed.
Hardware Errors and Exception Handling

An excepting instruction may or may not commit before the exception event is taken, depending on if it is a service type or an error type exception.

An instruction causing a service type event will commit, and the address written to the `RETX` register will be the next instruction after the excepting one. An example of a service type exception is the single step.

An instruction causing an error type event cannot commit, so the address written to the `RETX` register will be the address of the offending instruction. An example of an error type event is a CPLB miss.

| Usually the `RETX` register contains the correct address to return to. To skip over an excepting instruction, take care in case the next address is not simply the next linear address. This could happen when the excepting instruction is a loop end. In that case, the proper next address would be the loop top. |

The `EXCAUSE[5:0]` field in the Sequencer Status register (`SEOSTAT`) is written whenever an exception is taken, and indicates to the exception handler which type of exception occurred. Refer to Table 4-11 for a list of events that cause exceptions.

| If an exception occurs in an event handler that is already servicing an Exception, NMI, Reset, or Emulation event, this will trigger a double fault condition, and the address of the excepting instruction will be written to `RETX`. |
### Table 4-11. Events That Cause Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>EXCAUSE [5:0]</th>
<th>Type: (E) Error (S) Service See note 1.</th>
<th>Notes/Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Force Exception instruction EXCPT with 4-bit m field</td>
<td>m field</td>
<td>S</td>
<td>Instruction provides 4 bits of EXCAUSE.</td>
</tr>
<tr>
<td>Single step</td>
<td>0x10</td>
<td>S</td>
<td>When the processor is in single step mode, every instruction generates an exception. Primarily used for debugging.</td>
</tr>
<tr>
<td>Exception caused by a trace buffer full condition</td>
<td>0x11</td>
<td>S</td>
<td>The processor takes this exception when the trace buffer overflows (only when enabled by the Trace Unit Control register).</td>
</tr>
<tr>
<td>Undefined instruction</td>
<td>0x21</td>
<td>E</td>
<td>May be used to emulate instructions that are not defined for a particular processor implementation.</td>
</tr>
<tr>
<td>Illegal instruction combination</td>
<td>0x22</td>
<td>E</td>
<td>See section for multi-issue rules in the <em>ADSP-BF53x/BF56x Blackfin Processor Programming Reference</em>.</td>
</tr>
<tr>
<td>Data access CPLB protection violation</td>
<td>0x23</td>
<td>E</td>
<td>Attempted read or write to Supervisor resource, or illegal data memory access. Supervisor resources are registers and instructions that are reserved for Supervisor use: Supervisor only registers, all MMRs, and Supervisor only instructions. (A simultaneous, dual access to two MMRs using the data address generators generates this type of exception.) In addition, this entry is used to signal a protection violation caused by disallowed memory access, and it is defined by the Memory Management Unit (MMU) cacheability protection lookaside buffer (CPLB).</td>
</tr>
</tbody>
</table>

---

ADSP-BF53x/BF56x Blackfin Processor Programming Reference 4-63
## Hardware Errors and Exception Handling

Table 4-11. Events That Cause Exceptions (Cont’d)

<table>
<thead>
<tr>
<th>Exception</th>
<th>EXCAUSE [5:0]</th>
<th>Type: (E) Error (S) Service See note 1</th>
<th>Notes/Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data access misaligned address violation</td>
<td>0x24</td>
<td>E</td>
<td>Attempted misaligned data memory or data cache access.</td>
</tr>
<tr>
<td>Unrecoverable event</td>
<td>0x25</td>
<td>E</td>
<td>For example, an exception generated while processing a previous exception.</td>
</tr>
<tr>
<td>Data access CPLB miss</td>
<td>0x26</td>
<td>E</td>
<td>Used by the MMU to signal a CPLB miss on a data access.</td>
</tr>
<tr>
<td>Data access multiple CPLB hits</td>
<td>0x27</td>
<td>E</td>
<td>More than one CPLB entry matches data fetch address.</td>
</tr>
<tr>
<td>Exception caused by an emulation watchpoint match</td>
<td>0x28</td>
<td>E</td>
<td>There is a watchpoint match, and one of the EMUSW bits in the Watchpoint Instruction Address Control register (WPIACTL) is set.</td>
</tr>
<tr>
<td>Instruction fetch misaligned address violation</td>
<td>0x2A</td>
<td>E</td>
<td>Attempted misaligned instruction cache fetch. On a misaligned instruction fetch exception, the return address provided in RETX is the destination address which is misaligned, rather than the address of the offending instruction. For example, if an indirect branch to a misaligned address held in P0 is attempted, the return address in RETX is equal to P0, rather than to the address of the branch instruction. (Note this exception can never be generated from PC-relative branches, only from indirect branches.)</td>
</tr>
<tr>
<td>Instruction fetch CPLB protection violation</td>
<td>0x2B</td>
<td>E</td>
<td>Illegal instruction fetch access (memory protection violation).</td>
</tr>
<tr>
<td>Instruction fetch CPLB miss</td>
<td>0x2C</td>
<td>E</td>
<td>CPLB miss on an instruction fetch.</td>
</tr>
</tbody>
</table>
Program Sequencer

Table 4-11. Events That Cause Exceptions (Cont’d)

<table>
<thead>
<tr>
<th>Exception</th>
<th>EXCAUSE</th>
<th>Type: (E) Error (S) Service</th>
<th>Notes/Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch multiple CPLB hits</td>
<td>0x2D</td>
<td>E</td>
<td>More than one CPLB entry matches instruction fetch address.</td>
</tr>
<tr>
<td>Illegal use of supervisor resource</td>
<td>0x2E</td>
<td>E</td>
<td>Attempted to use a Supervisor register or instruction from User mode. Supervisor resources are registers and instructions that are reserved for Supervisor use: Supervisor only registers, all MMRs, and Supervisor only instructions.</td>
</tr>
</tbody>
</table>

Note 1: For services (S), the return address is the address of the instruction that follows the exception. For errors (E), the return address is the address of the excepting instruction.

If an instruction causes multiple exceptions, only the exception with the highest priority is taken. The following table ranks exceptions by descending priority.

Table 4-12. Exceptions by Descending Priority

<table>
<thead>
<tr>
<th>Priority</th>
<th>Exception</th>
<th>EXCAUSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Unrecoverable Event</td>
<td>0x25</td>
</tr>
<tr>
<td>2</td>
<td>I-Fetch Multiple CPLB Hits</td>
<td>0x2D</td>
</tr>
<tr>
<td>3</td>
<td>I-Fetch Misaligned Access</td>
<td>0x2A</td>
</tr>
<tr>
<td>4</td>
<td>I-Fetch Protection Violation</td>
<td>0x2B</td>
</tr>
<tr>
<td>5</td>
<td>I-Fetch CPLB Miss</td>
<td>0x2C</td>
</tr>
<tr>
<td>6</td>
<td>I-Fetch Access Exception</td>
<td>0x29</td>
</tr>
<tr>
<td>7</td>
<td>Watchpoint Match</td>
<td>0x28</td>
</tr>
<tr>
<td>8</td>
<td>Undefined Instruction</td>
<td>0x21</td>
</tr>
</tbody>
</table>
Hardware Errors and Exception Handling

Table 4-12. Exceptions by Descending Priority (Cont’d)

<table>
<thead>
<tr>
<th>Priority</th>
<th>Exception</th>
<th>EXCAUSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Illegal Combination</td>
<td>0x22</td>
</tr>
<tr>
<td>10</td>
<td>Illegal Use of Protected Resource</td>
<td>0x2E</td>
</tr>
<tr>
<td>11</td>
<td>DAG0 Multiple CPLB Hits</td>
<td>0x27</td>
</tr>
<tr>
<td>12</td>
<td>DAG0 Misaligned Access</td>
<td>0x24</td>
</tr>
<tr>
<td>13</td>
<td>DAG0 Protection Violation</td>
<td>0x23</td>
</tr>
<tr>
<td>14</td>
<td>DAG0 CPLB Miss</td>
<td>0x26</td>
</tr>
<tr>
<td>15</td>
<td>DAG1 Multiple CPLB Hits</td>
<td>0x27</td>
</tr>
<tr>
<td>16</td>
<td>DAG1 Misaligned Access</td>
<td>0x24</td>
</tr>
<tr>
<td>17</td>
<td>DAG1 Protection Violation</td>
<td>0x23</td>
</tr>
<tr>
<td>18</td>
<td>DAG1 CPLB Miss</td>
<td>0x26</td>
</tr>
<tr>
<td>19</td>
<td>EXCPT Instruction</td>
<td>m field</td>
</tr>
<tr>
<td>20</td>
<td>Single Step</td>
<td>0x10</td>
</tr>
<tr>
<td>21</td>
<td>Trace Buffer</td>
<td>0x11</td>
</tr>
</tbody>
</table>

Exceptions While Executing an Exception Handler

While executing the exception handler, avoid issuing an instruction that generates another exception. If an exception is caused while executing code within the exception handler, the NMI handler, the reset vector, or in emulator mode:

- The excepting instruction is not committed. All writebacks from the instruction are prevented.
- The generated exception is not taken.
Program Sequencer

- The \textsc{excause} field in \textsc{seqstat} is updated with an unrecoverable event code.

- The address of the offending instruction is saved in \textsc{retx}. Note if the processor were executing, for example, the NMI handler, the \textsc{retn} register would not have been updated; the excepting instruction address is always stored in \textsc{retx}.

To determine whether an exception occurred while an exception handler was executing, check \textsc{seqstat} at the end of the exception handler for the code indicating an “unrecoverable event” (\textsc{excause} = 0x25). If an unrecoverable event occurred, register \textsc{retx} holds the address of the most recent instruction to cause an exception. This mechanism is not intended for recovery, but rather for detection.

Exceptions and the Pipeline

Interrupts and exceptions treat instructions in the pipeline differently.

- When an interrupt occurs, all instructions in the pipeline are aborted.

- When an exception occurs, all instructions in the pipeline after the excepting instruction are aborted. For error exceptions, the excepting instruction is also aborted.

Because exceptions, NMIs, and emulation events have a dedicated return register, guarding the return address is optional. Consequently, the \texttt{push} and \texttt{pop} instructions for exceptions, NMIs, and emulation events do not affect the interrupt system.

Note, however, the return instructions for exceptions (\texttt{rtx}, \texttt{rtn}, and \texttt{rte}) do clear the Least Significant Bit (LSB) currently set in \textsc{ipend}.
Deferring Exception Processing

Exception handlers are usually long routines, because they must discriminate among several exception causes and take corrective action accordingly. The length of the routines may result in long periods during which the interrupt system is, in effect, suspended.

To avoid lengthy suspension of interrupts, write the exception handler to identify the exception cause, but defer the processing to a low priority interrupt. To set up the low priority interrupt handler, use the Force Interrupt / Reset instruction (RAISE).

When deferring the processing of an exception to lower priority interrupt IVGx, the system must guarantee that IVGx is entered before returning to the application-level code that issued the exception. If a pending interrupt of higher priority than IVGx occurs, it is acceptable to enter the high priority interrupt before IVGx.

Example Code for an Exception Handler

The following code is for an exception routine handler with deferred processing.

Listing 4-5. Exception Routine Handler With Deferred Processing

```c
/* Determine exception cause by examining EXCAUSE field in
SEQSTAT (first save contents of R0, P0, P1 and ASTAT in Supervisor SP) */
[--SP] = R0 ;
[--SP] = P0 ;
[--SP] = P1 ;
[--SP] = ASTAT ;
R0 = SEQSTAT ;
/* Mask the contents of SEQSTAT, and leave only EXCAUSE in R0 */
R0 <<= 26 ;
R0 >>= 26 ;
```
/* Using jump table EVTABLE, jump to the event pointed to by R0 */
P0 = R0 ;
P1 = _EVTABLE ;
P0 = P1 + ( P0 << 1 ) ;
R0 = W [ P0 ] (Z) ;
P1 = R0 ;
JUMP (PC + P1) :
/* The entry point for an event is as follows. Here, processing is deferred to low priority interrupt IVG15. Also, parameter passing would typically be done here. */
_EVENT1:
RAISE 15 ;
JUMP.S _EXIT ;
/* Entry for event at IVG14 */
_EVENT2:
RAISE 14 ;
JUMP.S _EXIT ;
/* Comments for other events */
/* At the end of handler, restore R0, P0, P1 and ASTAT, and return. */
_EXIT:
ASTAT = [SP++] ;
P1 = [SP++] ;
P0 = [SP++] ;
R0 = [SP++] ;
RTX ;
_EVENTABLE:
.byte2 addr_event1;
.byte2 addr_event2;
...
.byte2 addr_eventN;
Hardware Errors and Exception Handling

/* The jump table EVTABLE holds 16-bit address offsets for each event. With offsets, this code is position independent and the table is small. */
+--------------+
| addr_event1  | _EVTABLE
+--------------+
| addr_event2  | _EVTABLE + 2
+--------------+
| . . .        |
+--------------+
| addr_eventN  | _EVTABLE + 2N
+--------------+
*/

Example Code for an Exception Routine

The following code provides an example framework for an interrupt routine jumped to from an exception handler such as that described above.

Listing 4-6. Interrupt Routine for Handling Exception

[---SP] = RETI ; /* Push return address on stack. */

/* Put body of routine here. */

RETI = [SP++] ; /* To return, pop return address and jump. */

RTI ; /* Return from interrupt. */
Like most DSP and RISC platforms, the Blackfin processors have a load/store architecture. Computation operands and results are always represented by core registers. Prior to computation, data is loaded from memory into core registers and results are stored back by explicit move operations. The Address Arithmetic Unit (AAU) provides all the required support to keep data transport between memory and core registers efficient and seamless. Having a separate arithmetic unit for address calculations prevents the data computation block from being burdened by address operations. Not only can the load and store operations occur in parallel to data computations, but memory addresses can also be calculated at the same time.

The AAU uses Data Address Generators (DAGs) to generate addresses for data moves to and from memory. By generating addresses, the DAGs let programs refer to addresses indirectly, using a DAG register instead of an absolute address. Figure 5-1 shows the AAU block diagram.
The AAU architecture supports several functions that minimize overhead in data access routines. These functions include:

- Supply address – Provides an address during a data access
- Supply address and post-modify – Provides an address during a data move and auto-increments/decrements the stored address for the next move
- Supply address with offset – Provides an address from a base with an offset without incrementing the original address pointer
- Modify address – Increments or decrements the stored address without performing a data move
- Bit-reversed carry address – Provides a bit-reversed carry address during a data move without reversing the stored address
Address Arithmetic Unit

The AAU comprises two DAGs, nine Pointer registers, four Index registers and four complete sets of related Modify, Base, and Length registers. These registers, shown in Figure 5-2 on page 5-4, hold the values that the DAGs use to generate addresses. The types of registers are:

- **Index registers, I[3:0]**. Unsigned 32-bit Index registers hold an address pointer to memory. For example, the instruction $R3 = [I0]$ loads the data value found at the memory location pointed to by the register $I0$. Index registers can be used for 16- and 32-bit memory accesses.

- **Modify registers, M[3:0]**. Signed 32-bit Modify registers provide the increment or step size by which an Index register is post-modified during a register move. For example, the $R0 = [I0 \pm M1]$ instruction directs the DAG to:
  - Output the address in register $I0$
  - Load the contents of the memory location pointed to by $I0$ into $R0$
  - Modify the contents of $I0$ by the value contained in the $M1$ register

- **Base and Length registers, B[3:0] and L[3:0]**. Unsigned 32-bit Base and Length registers set up the range of addresses and the starting address of a. Each $B, L$ pair is always coupled with a corresponding $I$-register, for example, $I3, B3, L3$. For more information on circular buffers, see “Addressing Circular Buffers” on page 5-12.

- **Pointer registers, P[5:0], FP, USP, and SP**. 32-bit Pointer registers hold an address pointer to memory. The $P[5:0]$ field, $FP$ (Frame Pointer) and $SP/USP$ (Stack Pointer/User Stack Pointer) can be manipulated and used in various instructions. For example, the instruction $R3 = [P0]$ loads the register $R3$ with the data value found at the memory location pointed to by the register $P0$. The Pointer registers have no effect on circular buffer addressing. They
can be used for 8-, 16-, and 32-bit memory accesses. For added mode protection, $SP$ is accessible only in Supervisor mode, while $USP$ is accessible in User mode.

\[ \text{Figure 5-2. Address Arithmetic Unit} \]

\begin{center}

\begin{tabular}{|c|c|c|}
\hline
Data Address Registers & & Pointer Registers \\
\hline
I0 & L0 & B0 & M0 & P0 \\
I1 & L1 & B1 & M1 & P1 \\
I2 & L2 & B2 & M2 & P2 \\
I3 & L3 & B3 & M3 & P3 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|}
\hline
User SP & Supervisor SP \\
\hline
FP & \\
\hline
\end{tabular}

Supervisor only register. Attempted read or write in User mode causes an exception error.

\end{center}

Do not assume the L-registers are automatically initialized to zero for linear addressing. The I-, M-, L-, and B-registers contain random values after reset. For each I-register used, programs must initialize the corresponding L-registers to zero for linear addressing or to the buffer length for circular buffer addressing.

Note all data address registers must be initialized individually. Initializing a B-register does not automatically initialize the I-register.
Address Arithmetic Unit

Addressing With the AAU

The DAGs can generate an address that is incremented by a value or by a register. In post-modify addressing, the DAG outputs the I-register value unchanged; then the DAG adds an M-register or immediate value to the I-register.

In indexed addressing, the DAG adds a small offset to the value in the P-register, but does not update the P-register with this new value, thus providing an offset for that particular memory access.

The processor is byte addressed. All data accesses must be aligned to the data size. In other words, a 32-bit fetch must be aligned to 32 bits, but an 8-bit store can be aligned to any byte. Depending on the type of data used, increments and decrements to the address registers can be by 1, 2, or 4 to match the 8-, 16-, or 32-bit accesses.

For example, consider the following instruction:

R0 = [ P3++ ];

This instruction fetches a 32-bit word, pointed to by the value in P3, and places it in R0. It then post-increments P3 by four, maintaining alignment with the 32-bit access.

R0.L = W [ I3++ ];

This instruction fetches a 16-bit word, pointed to by the value in I3, and places it in the low half of the destination register, R0.L. It then post-increments I3 by two, maintaining alignment with the 16-bit access.

R0 = B [ P3++ ] (2);

This instruction fetches an 8-bit word, pointed to by the value in P3, and places it in the destination register, R0. It then post-increments P3 by one, maintaining alignment with the 8-bit access. The byte value may be zero extended (as shown) or sign extended into the 32-bit data register.
Addressing With the AAU

Instructions using Index registers use an M-register or a small immediate value (+/- 2 or 4) as the modifier. Instructions using Pointer registers use a small immediate value or another P-register as the modifier. For details, see Table 5-3, "AAU Instruction Summary," on page 5-20.

Pointer Register File

The general-purpose Address Pointer registers, also called P-registers, are organized as:

- 6-entry, P-register file \( P[5:0] \)
- Frame Pointer (\( FP \)) used to point to the current procedure’s activation record
- Stack Pointer (\( SP \)) used to point to the last used location on the runtime stack.

P-registers are 32 bits wide. Although P-registers are primarily used for address calculations, they may also be used for general integer arithmetic with a limited set of arithmetic operations; for instance, to maintain counters. However, unlike the Data registers, P-register arithmetic does not affect the Arithmetic Status (\( ASTAT \)) register status flags.

Frame and Stack Pointers

In many respects, the Frame and Stack Pointer registers perform like the other P-registers, \( P[5:0] \). They can act as general pointers in any of the load/store instructions, for example, \( R1 = B[SP] (Z) \). However, \( FP \) and \( SP \) have additional functionality.

The Stack Pointer registers include:

- a User Stack Pointer (\( USP \) in Supervisor mode, \( SP \) in User mode)
- a Supervisor Stack Pointer (\( SP \) in Supervisor mode)
Address Arithmetic Unit

The User Stack Pointer register and the Supervisor Stack Pointer register are accessed using the register alias $SP$. Depending on the current processor operating mode, only one of these registers is active and accessible as $SP$:

- In User mode, any reference to $SP$ (for example, stack pop $RO = [SP++]$) implicitly uses the $USP$ as the effective address.

- In Supervisor mode, the same reference to $SP$ (for example, $RO = [SP++]$) implicitly uses the Supervisor Stack Pointer as the effective address.

To manipulate the User Stack Pointer for code running in Supervisor mode, use the register alias $USP$. When in Supervisor mode, a register move from $USP$ (for example, $RO = USP$) moves the current User Stack Pointer into $RD$. The register alias $USP$ can only be used in Supervisor mode.

Some load/store instructions use $FP$ and $SP$ implicitly:

- $FP$-indexed load/store, which extends the addressing range for 16-bit encoded load/stores

- Stack push/pop instructions, including those for pushing and popping multiple registers

- Link/unlink instructions, which control stack frame space and manage the Frame Pointer register ($FP$) for that space
Addressing With the AAU

DAG Register Set

DSP instructions primarily use the Data Address Generator (DAG) register set for addressing. The data address register set consists of these registers:

- \( I[3:0] \) contain index addresses
- \( M[3:0] \) contain modify values
- \( B[3:0] \) contain base addresses
- \( L[3:0] \) contain length values

All data address registers are 32 bits wide.

The I (Index) registers and B (Base) registers always contain addresses of 8-bit bytes in memory. The Index registers contain an effective address. The M (Modify) registers contain an offset value that is added to one of the Index registers or subtracted from it.

The B and L (Length) registers define circular buffers. The B register contains the starting address of a buffer, and the L register contains the length in bytes. Each L and B register pair is associated with the corresponding I register. For example, \( L0 \) and \( B0 \) are always associated with \( I0 \). However, any M register may be associated with any I register. For example, \( I0 \) may be modified by \( M3 \).

Indexed Addressing With Index & Pointer Registers

Indexed addressing uses the value in the Index or Pointer register as an effective address. This instruction can load or store 16- or 32-bit values. The default is a 32-bit transfer. If a 16-bit transfer is required, then the \( W \) designator is used to preface the load or store.
Address Arithmetic Unit

For example:

\[ \text{R0} = [\text{I2}] ; \]

loads a 32-bit value from an address pointed to by I2 and stores it in the destination register R0.

\[ \text{R0.H} = \text{W}[\text{I2}] ; \]

loads a 16-bit value from an address pointed to by I2 and stores it in the 16-bit destination register \text{R0.H}.

\[ [\text{P1}] = \text{R0} ; \]

is an example of a 32-bit store operation.

Pointer registers can be used for 8-bit loads and stores.

For example:

\[ \text{B}[\text{P1++}] = \text{R0} ; \]

stores the 8-bit value from the \text{R0} register in the address pointed to by the \text{P1} register, then increments the \text{P1} register.

Loads With Zero or Sign Extension

When a 32-bit register is loaded by an 8-bit or 16-bit memory read, the value can be extended to the full register width. A trailing Z character in parenthesis is used to zero-extend the loaded value. An X character forces sign extension. The following examples assume that \text{P1} points to a memory location that contains a value of 0x8080.

\[ \text{R0} = \text{W}[\text{P1}] (Z) ; \quad \text{/* R0 = 0x0000 8080 */} \]
\[ \text{R1} = \text{W}[\text{P1}] (X) ; \quad \text{/* R1 = 0xFFFF 8080 */} \]
\[ \text{R2} = \text{B}[\text{P1}] (Z) ; \quad \text{/* R2 = 0x0000 0080 */} \]
\[ \text{R3} = \text{B}[\text{P1}] (X) ; \quad \text{/* R3 = 0xFFFF FF80 */} \]
Indexed Addressing With Immediate Offset

Indexed addressing allows programs to obtain values from data tables, with reference to the base of that table. The Pointer register is modified by the immediate field and then used as the effective address. The value of the Pointer register is not updated.

Alignment exceptions are triggered when a final address is unaligned.

For example, if $P1 = 0x13$, then $[P1 + 0x11]$ would effectively be equal to $[0x24]$, which is aligned for all accesses.

Auto-increment and Auto-decrement Addressing

Auto-increment addressing updates the Pointer and Index registers after the access. The amount of increment depends on the word size. An access of 32-bit words results in an update of the Pointer by 4. A 16-bit word access updates the Pointer by 2, and an access of an 8-bit word updates the Pointer by 1. Both 8- and 16-bit read operations may specify either to sign-extend or zero-extend the contents into the destination register. Pointer registers may be used for 8-, 16-, and 32-bit accesses while Index registers may be used only for 16- and 32-bit accesses.

For example:

```c
R0 = W [ P1++ ] (Z);
```

loads a 16-bit word into a 32-bit destination register from an address pointed to by the $P1$ Pointer register. The Pointer is then incremented by 2 and the word is zero extended to fill the 32-bit destination register.

Auto-decrement works the same way by decrementing the address after the access.
Address Arithmetic Unit

For example:

\begin{verbatim}
R0 = [ I2-- ];
\end{verbatim}

loads a 32-bit value into the destination register and decrements the Index register by 4.

**Pre-modify Stack Pointer Addressing**

The only pre-modify instruction in the processor uses the Stack Pointer register, SP. The address in SP is decremented by 4 and then used as an effective address for the store. The instruction \([--SP] = R0\) is used for stack push operations and can support only a 32-bit word transfer.

**Post-modify Addressing**

Post-modify addressing uses the value in the Index or Pointer registers as the effective address and then modifies it by the contents of another register. Pointer registers are modified by other Pointer registers. Index registers are modified by Modify registers. Post-modify addressing does not support the Pointer registers as destination registers, nor does it support byte-addressing.

For example:

\begin{verbatim}
R5 = [ P1++P2 ];
\end{verbatim}

loads a 32-bit value into the R5 register, found in the memory location pointed to by the P1 register.

The value in the P2 register is then added to the value in the P1 register.
Addressing With the AAU

For example:

\[ R2 = W \[ P4++P5 \] (Z) ; \]

loads a 16-bit word into the low half of the destination register \( R2 \) and zero-extends it to 32 bits. The value of the pointer \( P4 \) is incremented by the value of the pointer \( P5 \).

For example:

\[ R2 = [ I2++M1 ] ; \]

loads a 32-bit word into the destination register \( R2 \). The value in the Index register, \( I2 \), is updated by the value in the Modify register, \( M1 \).

Addressing Circular Buffers

The DAGs support addressing circular buffers. Circular buffers are a range of addresses containing data that the DAG steps through repeatedly, wrapping around to repeat stepping through the same range of addresses in a circular pattern.

The DAGs use four types of data address registers for addressing circular buffers. For circular buffering, the registers operate this way:

- The Index (I) register contains the value that the DAG outputs on the address bus.
- The Modify (M) register contains the post-modify amount (positive or negative) that the DAG adds to the I-register at the end of each memory access.

Any M-register can be used with any I-register. The modify value can also be an immediate value instead of an M-register. The size of the modify value must be less than or equal to the length (L-register) of the circular buffer.
Address Arithmetic Unit

- The Length (L) register sets the size of the circular buffer and the address range through which the DAG circulates the I-register.

  L is positive and cannot have a value greater than $2^{32} - 1$. If an L-register’s value is zero, its circular buffer operation is disabled.

- The Base (B) register or the B-register plus the L-register is the value with which the DAG compares the modified I-register value after each access.

To address a circular buffer, the DAG steps the Index pointer (I-register) through the buffer values, post-modifying and updating the index on each access with a positive or negative modify value from the M-register.

If the Index pointer falls outside the buffer range, the DAG subtracts the length of the buffer (L-register) from the value or adds the length of the buffer to the value, wrapping the Index pointer back to a point inside the buffer.

The starting address that the DAG wraps around is called the buffer’s base address (B-register). There are no restrictions on the value of the base address for circular buffers that contains 8-bit data. Circular buffers that contain 16- and 32-bit data must be 16-bit aligned and 32-bit aligned, respectively. Exceptions can be made for video operations. For more information, see “Memory Address Alignment” on page 5-16. Circular buffering uses post-modify addressing.
As seen in Figure 5-3, on the first post-modify access to the buffer, the DAG outputs the I-register value on the address bus, then modifies the address by adding the modify value.

- If the updated index value is within the buffer length, the DAG writes the value to the I-register.
- If the updated index value exceeds the buffer length, the DAG subtracts (for a positive modify value) or adds (for a negative modify value) the L-register value before writing the updated index value to the I-register.

Figure 5-3. Circular Data Buffers
Address Arithmetic Unit

In equation form, these post-modify and wraparound operations work as follows, shown for “I+M” operations.

- If M is positive:

\[
I_{\text{new}} = I_{\text{old}} + M
\]
if \(I_{\text{old}} + M < \text{buffer base} + \text{length} \) (end of buffer)

\[
I_{\text{new}} = I_{\text{old}} + M - L
\]
if \(I_{\text{old}} + M \geq \text{buffer base} + \text{length} \) (end of buffer)

- If M is negative:

\[
I_{\text{new}} = I_{\text{old}} + M
\]
if \(I_{\text{old}} + M \geq \text{buffer base} \) (start of buffer)

\[
I_{\text{new}} = I_{\text{old}} + M + L
\]
if \(I_{\text{old}} + M < \text{buffer base} \) (start of buffer)

Addressing With Bit-reversed Addresses

To obtain results in sequential order, programs need bit-reversed carry addressing for some algorithms, particularly Fast Fourier Transform (FFT) calculations. To satisfy the requirements of these algorithms, the DAG’s bit-reversed addressing feature permits repeatedly subdividing data sequences and storing this data in bit-reversed order. For detailed information about bit-reversed addressing, see “Modify – Increment” on page 15-37.

Modifying DAG and Pointer Registers

The DAGs support operations that modify an address value in an Index register without outputting an address. The operation, address-modify, is useful for maintaining pointers.
The address-modify operation modifies addresses in any Index and Pointer register (I[3:0], P[5:0], FP, SP) without accessing memory. If the Index register’s corresponding B- and L-registers are set up for circular buffering, the address-modify operation performs the specified buffer wraparound (if needed).

The syntax is similar to post-modify addressing (index += modifier). For Index registers, an M-register is used as the modifier. For Pointer registers, another P-register is used as the modifier.

Consider the example, \( I_1 += M_2 \);

This instruction adds \( M_2 \) to \( I_1 \) and updates \( I_1 \) with the new value.

The processor requires proper memory alignment to be maintained for the data size being accessed. Unless exceptions are disabled, violations of memory alignment cause an alignment exception. Some instructions—for example, many of the Video ALU instructions—automatically disable alignment exceptions because the data may not be properly aligned when stored in memory. Alignment exceptions may be disabled by issuing the \texttt{DISALGNEXCPT} instruction in parallel with a load/store operation.

Normally, the memory system requires two address alignments:

- 32-bit word load/stores are accessed on four-byte boundaries, meaning the two least significant bits of the address are \( \text{b}\#00 \).

- 16-bit word load/stores are accessed on two-byte boundaries, meaning the least significant bit of the address must be \( \text{b}\#0 \).
Table 5-1 summarizes the types of transfers and transfer sizes supported by the addressing modes.

Table 5-1. Types of Transfers Supported and Transfer Sizes

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Types of Transfers Supported</th>
<th>Transfer Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto-increment</td>
<td>To and from Data Registers</td>
<td>LOADS:</td>
</tr>
<tr>
<td>Auto-decrement</td>
<td></td>
<td>32-bit word</td>
</tr>
<tr>
<td>Indirect</td>
<td></td>
<td>16-bit, zero extended half word</td>
</tr>
<tr>
<td>Indexed</td>
<td></td>
<td>16-bit, sign extended half word</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8-bit, zero extended byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8-bit, sign extended byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STORES:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32-bit word</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16-bit half word</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8-bit byte</td>
</tr>
<tr>
<td>Post-increment</td>
<td>To and from Data Registers</td>
<td>LOAD:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32-bit word</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STORE:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32-bit word</td>
</tr>
</tbody>
</table>

Be careful when using the **DISALGNEXCPT** instruction, because it disables automatic detection of memory alignment errors. The **DISALGNEXCPT** instruction only affects misaligned loads that use I-register indirect addressing. Misaligned loads using P-register addressing will still cause an exception.
Memory Address Alignment

Table 5-2 summarizes the addressing modes. In the table, an asterisk (*) indicates the processor supports the addressing mode.

Table 5-2. Addressing Modes

<table>
<thead>
<tr>
<th>Address Mode</th>
<th>32-bit word</th>
<th>16-bit half-word</th>
<th>8-bit byte</th>
<th>Sign/Zero extend</th>
<th>Data Register</th>
<th>Pointer register</th>
<th>Data Register Half</th>
</tr>
</thead>
<tbody>
<tr>
<td>P Auto-inc [P0++]</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>P Auto-dec [P0--]</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>P Indirect [P0]</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>P Indexed [P0+im]</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>FP indexed [FP+im]</td>
<td>*</td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>P Post-inc [P0++P1]</td>
<td>*</td>
<td>*</td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>I Auto-inc [I0++]</td>
<td>*</td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>I Auto-dec [I0--]</td>
<td>*</td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>I Indirect [I0]</td>
<td>*</td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>I Post-inc [I0++M0]</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>
AAU Instruction Summary

Table 5-3 lists the AAU instructions. In Table 5-3, note the meaning of these symbols:

- Dreg denotes any Data Register File register.
- Dreg_lo denotes the lower 16 bits of any Data Register File register.
- Dreg_hi denotes the upper 16 bits of any Data Register File register.
- Preg denotes any Pointer register, FP, or SP register.
- Ireg denotes any Index register.
- Mreg denotes any Modify register.
- W denotes a 16-bit wide value.
- B denotes an 8-bit wide value.
- immA denotes a signed, A-bits wide, immediate value.
- uimmAmB denotes an unsigned, A-bits wide, immediate value that is an even multiple of B.
- Z denotes the zero-extension qualifier.
- X denotes the sign-extension qualifier.
- BREV denotes the bit-reversal qualifier.
AAU Instruction Summary

AAU instructions do not affect the \texttt{ASTAT} Status flags.

Table 5-3. AAU Instruction Summary

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Preg} = [ \text{Preg} ]$ ;</td>
</tr>
<tr>
<td>$\text{Preg} = [ \text{Preg} ++ ]$ ;</td>
</tr>
<tr>
<td>$\text{Preg} = [ \text{Preg} -- ]$ ;</td>
</tr>
<tr>
<td>$\text{Preg} = [ \text{Preg} + \text{uumm6m4} ]$ ;</td>
</tr>
<tr>
<td>$\text{Preg} = [ \text{Preg} + \text{uumm17m4} ]$ ;</td>
</tr>
<tr>
<td>$\text{Preg} = [ \text{Preg} - \text{uumm17m4} ]$ ;</td>
</tr>
<tr>
<td>$\text{Preg} = [ \text{FP} - \text{uumm7m4} ]$ ;</td>
</tr>
<tr>
<td>$\text{Dreg} = [ \text{Preg} ]$ ;</td>
</tr>
<tr>
<td>$\text{Dreg} = [ \text{Preg} ++ ]$ ;</td>
</tr>
<tr>
<td>$\text{Dreg} = [ \text{Preg} -- ]$ ;</td>
</tr>
<tr>
<td>$\text{Dreg} = [ \text{Preg} + \text{uumm6m4} ]$ ;</td>
</tr>
<tr>
<td>$\text{Dreg} = [ \text{Preg} + \text{uumm17m4} ]$ ;</td>
</tr>
<tr>
<td>$\text{Dreg} = [ \text{Preg} - \text{uumm17m4} ]$ ;</td>
</tr>
<tr>
<td>$\text{Dreg} = [ \text{Preg} ++ \text{Preg} ]$ ;</td>
</tr>
<tr>
<td>$\text{Dreg} = [ \text{FP} - \text{uumm7m4} ]$ ;</td>
</tr>
<tr>
<td>$\text{Dreg} = [ \text{Ireg} ]$ ;</td>
</tr>
<tr>
<td>$\text{Dreg} = [ \text{Ireg} ++ ]$ ;</td>
</tr>
<tr>
<td>$\text{Dreg} = [ \text{Ireg} -- ]$ ;</td>
</tr>
<tr>
<td>$\text{Dreg} = [ \text{Ireg} ++ \text{Mreg} ]$ ;</td>
</tr>
<tr>
<td>$\text{Dreg} = \text{W}[ \text{Preg} ] (Z)$ ;</td>
</tr>
<tr>
<td>$\text{Dreg} = \text{W}[ \text{Preg} ++ ] (Z)$ ;</td>
</tr>
<tr>
<td>$\text{Dreg} = \text{W}[ \text{Preg} -- ] (Z)$ ;</td>
</tr>
<tr>
<td>$\text{Dreg} = \text{W}[ \text{Preg} + \text{uumm5m2} ] (Z)$ ;</td>
</tr>
</tbody>
</table>
Address Arithmetic Unit

Table 5-3. AAU Instruction Summary (Cont'd)

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dreg = W [ Preg + uimm16m2 ] (Z) ;</td>
</tr>
<tr>
<td>Dreg = W [ Preg – uimm16m2 ] (Z) ;</td>
</tr>
<tr>
<td>Dreg = W [ Preg ++ Preg ] (Z) ;</td>
</tr>
<tr>
<td>Dreg = W [ Preg ] (X) ;</td>
</tr>
<tr>
<td>Dreg = W [ Preg ++ ] (X) ;</td>
</tr>
<tr>
<td>Dreg = W [ Preg -- ] (X) ;</td>
</tr>
<tr>
<td>Dreg = W [ Preg + uimm5m2 ] (X) ;</td>
</tr>
<tr>
<td>Dreg = W [ Preg + uimm16m2 ] (X) ;</td>
</tr>
<tr>
<td>Dreg = W [ Preg – uimm16m2 ] (X) ;</td>
</tr>
<tr>
<td>Dreg = W [ Preg ++ Preg ] (X) ;</td>
</tr>
<tr>
<td>Dreg_hi = W [ Ireg ] ;</td>
</tr>
<tr>
<td>Dreg_hi = W [ Ireg ++ ] ;</td>
</tr>
<tr>
<td>Dreg_hi = W [ Ireg -- ] ;</td>
</tr>
<tr>
<td>Dreg_hi = W [ Preg ] ;</td>
</tr>
<tr>
<td>Dreg_hi = W [ Preg ++ Preg ] ;</td>
</tr>
<tr>
<td>Dreg_lo = W [ Ireg ] ;</td>
</tr>
<tr>
<td>Dreg_lo = W [ Ireg ++ ] ;</td>
</tr>
<tr>
<td>Dreg_lo = W [ Ireg -- ] ;</td>
</tr>
<tr>
<td>Dreg_lo = W [ Preg ] ;</td>
</tr>
<tr>
<td>Dreg_lo = W [ Preg ++ Preg ] ;</td>
</tr>
<tr>
<td>Dreg = B [ Preg ] (Z) ;</td>
</tr>
<tr>
<td>Dreg = B [ Preg ++ ] (Z) ;</td>
</tr>
<tr>
<td>Dreg = B [ Preg -- ] (Z) ;</td>
</tr>
<tr>
<td>Dreg = B [ Preg + uimm15 ] (Z) ;</td>
</tr>
</tbody>
</table>
## AAU Instruction Summary

### Table 5-3. AAU Instruction Summary (Cont'd)

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dreg = B [ Preg – uimm15 ] (Z) ;</td>
</tr>
<tr>
<td>Dreg = B [ Preg ] (X) ;</td>
</tr>
<tr>
<td>Dreg = B [ Preg ++ ] (X) ;</td>
</tr>
<tr>
<td>Dreg = B [ Preg -- ] (X) ;</td>
</tr>
<tr>
<td>Dreg = B [ Preg + uimm15 ] (X) ;</td>
</tr>
<tr>
<td>Dreg = B [ Preg – uimm15 ] (X) ;</td>
</tr>
<tr>
<td>[ Preg ] = Preg ;</td>
</tr>
<tr>
<td>[ Preg ++ ] = Preg ;</td>
</tr>
<tr>
<td>[ Preg -- ] = Preg ;</td>
</tr>
<tr>
<td>[ Preg + uimm6m4 ] = Preg ;</td>
</tr>
<tr>
<td>[ Preg + uimm17m4 ] = Preg ;</td>
</tr>
<tr>
<td>[ Preg – uimm17m4 ] = Preg ;</td>
</tr>
<tr>
<td>[ FP – uimm7m4 ] = Preg ;</td>
</tr>
<tr>
<td>[ Preg ] = Dreg ;</td>
</tr>
<tr>
<td>[ Preg ++ ] = Dreg ;</td>
</tr>
<tr>
<td>[ Preg -- ] = Dreg ;</td>
</tr>
<tr>
<td>[ Preg + uimm6m4 ] = Dreg ;</td>
</tr>
<tr>
<td>[ Preg + uimm17m4 ] = Dreg ;</td>
</tr>
<tr>
<td>[ Preg – uimm17m4 ] = Dreg ;</td>
</tr>
<tr>
<td>[ Preg ++ Preg ] = Dreg ;</td>
</tr>
<tr>
<td>[FP – uimm7m4 ] = Dreg ;</td>
</tr>
<tr>
<td>[ Ireg ] = Dreg ;</td>
</tr>
<tr>
<td>[ Ireg ++ ] = Dreg ;</td>
</tr>
<tr>
<td>[ Ireg -- ] = Dreg ;</td>
</tr>
</tbody>
</table>
## Address Arithmetic Unit

### Table 5-3. AAU Instruction Summary (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ Ireg ++ Mreg ] = Dreg ;</td>
</tr>
<tr>
<td>W [ Ireg ] = Dreg_hi ;</td>
</tr>
<tr>
<td>W [ Ireg ++ ] = Dreg_hi ;</td>
</tr>
<tr>
<td>W [ Ireg -- ] = Dreg_hi ;</td>
</tr>
<tr>
<td>W [ Preg ] = Dreg_hi ;</td>
</tr>
<tr>
<td>W [ Preg ++ Preg ] = Dreg_hi ;</td>
</tr>
<tr>
<td>W [ Ireg ] = Dreg_lo ;</td>
</tr>
<tr>
<td>W [ Ireg ++ ] = Dreg_lo ;</td>
</tr>
<tr>
<td>W [ Ireg -- ] = Dreg_lo ;</td>
</tr>
<tr>
<td>W [ Preg ] = Dreg_lo ;</td>
</tr>
<tr>
<td>W [ Preg -- ] = Dreg ;</td>
</tr>
<tr>
<td>W [ Preg + uimm5m2 ] = Dreg ;</td>
</tr>
<tr>
<td>W [ Preg + uimm16m2 ] = Dreg ;</td>
</tr>
<tr>
<td>W [ Preg – uimm16m2 ] = Dreg ;</td>
</tr>
<tr>
<td>W [ Preg ++ Preg ] = Dreg_lo ;</td>
</tr>
<tr>
<td>B [ Preg ] = Dreg ;</td>
</tr>
<tr>
<td>B [ Preg ++ ] = Dreg ;</td>
</tr>
<tr>
<td>B [ Preg -- ] = Dreg ;</td>
</tr>
<tr>
<td>Preg = imm7 (X) ;</td>
</tr>
<tr>
<td>Preg = imm16 (X) ;</td>
</tr>
</tbody>
</table>
Many of the AAU instructions can be part of multi-issue operations. Data can be loaded and stored in parallel to arithmetical operations. For details, see Chapter 20, “Issuing Parallel Instructions.”
Blackfin processors support a hierarchical memory model with different performance and size parameters, depending on the memory location within the hierarchy. Level 1 (L1) memories interconnect closely and efficient with the Blackfin core for best performance. Separate blocks of L1 memory can be accessed simultaneously through multiple bus systems. Instruction memory is separated from data memory, but unlike classical Harvard architectures, all L1 memory blocks are accessed by one unified addressing scheme. Portions of L1 memory can be configured to function as cache memory. Some Blackfin derivatives also feature on-chip Level 2 (L2) memories. Based on a Von-Neumann architecture, L2 memories have a unified purpose and can freely store instructions and data. Although L2 memories still reside inside the \texttt{CCLK} clock domain, they take multiple \texttt{CCLK} cycles to access. The processors also provide support of an external memory space that includes asynchronous memory space for static RAM devices and synchronous memory space for dynamic RAM such as SDRAM devices.

This chapter discusses the architecture and principles of on-chip memories as well as memory protection and caching mechanisms. For memory size, population, and off-chip memory interfaces, refer to the specific \textit{Blackfin Processor Hardware Reference} manual for your derivative.
Memory Architecture

Blackfin processors have a unified 4G byte address range that spans a combination of on-chip and off-chip memory and memory-mapped I/O resources. Of this range, some of the address space is dedicated to internal, on-chip resources. The processor populates portions of this internal memory space with:

- L1 Static Random Access Memories (SRAM)
- L2 Static Random Access Memories (SRAM)
- A set of memory-mapped registers (MMRs)
- A boot Read-Only Memory (ROM)

Figure 6-1 on page 6-3 shows a processor memory architecture typical of most Blackfin processors.

Overview of On-Chip Level 1 (L1) Memory

The L1 memory system performance provides high bandwidth and low latency. Because SRAMs provide deterministic access time and very high throughput, DSP systems have traditionally achieved performance improvements by providing fast SRAM on the chip.

The addition of instruction and data caches (SRAMs with cache control hardware) provides both high performance and a simple programming model. Caches eliminate the need to explicitly manage data movement into and out of L1 memories. Code can be ported to or developed for the processor quickly without requiring performance optimization for the memory organization.

Figure 6-1 shows the typical bus architecture of single-core Blackfin devices that do not feature L2 memories on-chip. The bus widths on the system side may vary.
The L1 memory provides:

- A modified Harvard architecture, allowing up to four core memory accesses per clock cycle (one 64-bit instruction fetch, two 32-bit data loads, and one pipelined 32-bit data store)
- Simultaneous system DMA, cache maintenance, and core accesses
- SRAM access at processor clock rate \( \text{CCLK} \) for critical DSP algorithms and fast context switching

Figure 6-1. Processor Memory Architecture
Memory Architecture

- Instruction and data cache options for microcontroller code, excellent High Level Language (HLL) support, and ease of programming cache control instructions, such as \texttt{PREFETCH} and \texttt{FLUSH}
- Memory protection

![Note]

The L1 memories operate at the core clock frequency (\texttt{CCLK}).

Overview of Scratchpad Data SRAM

The processor provides a dedicated 4K byte bank of scratchpad data SRAM. The scratchpad is independent of the configuration of the other L1 memory banks and cannot be configured as cache or targeted by DMA. Typical applications use the scratchpad data memory where speed is critical. For example, the User and Supervisor stacks should be mapped to the scratchpad memory for the fastest context switching during interrupt handling.

![Note]

The scratchpad data SRAM, like the other L1 blocks, operates at core clock frequency (\texttt{CCLK}). It can be accessed by the core at full performance. However, it cannot be accessed by the DMA controller.

Overview of On-Chip Level 2 (L2) Memory

Some Blackfin derivatives feature a Level 2 (L2) memory on chip. The L2 memory provides low latency, high-bandwidth capacity. This memory system is referred to as on-chip L2 because it forms an on-chip memory hierarchy with L1 memory. On-chip L2 memory provides more capacity than L1 memory, but the latency is higher. The on-chip L2 memory is SRAM and can not be configured as cache. It is capable of storing both instructions and data. The L1 caches can be configured to cache instructions and data located in the on-chip L2 memory. On-chip L2 memory operates at \texttt{CCLK} frequency.
L1 Instruction Memory

L1 Instruction Memory consists of a combination of dedicated SRAM and banks which can be configured as SRAM or cache. For the 16K byte bank that can be either cache or SRAM, control bits in the IMEM_CONTROL register can be used to organize all four subbanks of the L1 Instruction Memory as:

- A simple SRAM
- A 4-Way, set associative instruction cache
- A cache with as many as four locked Ways

ℹ️ L1 Instruction Memory can be used only to store instructions.

IMEM_CONTROL Register

The Instruction Memory Control register (IMEM_CONTROL) contains control bits for the L1 Instruction Memory. By default after reset, cache and Cacheability Protection Lookaside Buffer (CPLB) address checking is disabled (see “L1 Instruction Cache” on page 6-10).

When the LRUPRIORST bit is set to 1, the cached states of all CPLB_LRPRIORO bits (see “ICPLB_DATAx Registers” on page 6-55) are cleared. This simultaneously forces all cached lines to be of equal (low) importance. Cache replacement policy is based first on line importance indicated by the cached states of the CPLB_LRUPRIORO bits, and then on LRU (least recently used). See “Instruction Cache Locking by Line” on page 6-16 for complete details. This bit must be 0 to allow the state of the CPLB_LRUPRIORO bits to be stored when new lines are cached.

The ILOC[3:0] bits provide a useful feature only after code has been manually loaded into cache. See “Instruction Cache Locking by Way” on page 6-17. These bits specify which Ways to remove from the cache replacement policy. This has the effect of locking code present in...
nonparticipating Ways. Code in nonparticipating Ways can still be removed from the cache using an **IFLUSH** instruction. If an **ILOC[3:0]** bit is 0, the corresponding Way is not locked and that Way participates in cache replacement policy. If an **ILOC[3:0]** bit is 1, the corresponding Way is locked and does not participate in cache replacement policy.

The **IMC** bit reserves a portion of L1 instruction SRAM to serve as cache. Note reserving memory to serve as cache will not alone enable L2 memory accesses to be cached. CPLBs must also be enabled using the **EN_ICPLB** bit and the CPLB descriptors (**ICPLB_DATAx** and **ICPLB_ADDRx** registers) must specify desired memory pages as cache-enabled.

Instruction CPLBs are disabled by default after reset. When disabled, only minimal address checking is performed by the L1 memory interface. This minimal checking generates an exception to the processor whenever it attempts to fetch an instruction from:

- Reserved (nonpopulated) L1 instruction memory space
- L1 data memory space
- MMR space

CPLBs must be disabled using this bit prior to updating their descriptors (**DCPLB_DATAx** and **DCPLB_ADDRx** registers). Note since load store ordering is weak (see “Ordering of Loads and Stores” on page 6-67), disabling of CPLBs should be proceeded by a **CSYNC**.

When enabling or disabling cache or CPLBs, immediately follow the write to **IMEM_CONTROL** with a **SSYNC** to ensure proper behavior.

To ensure proper behavior and future compatibility, all reserved bits in this register must be set to 0 whenever this register is written.
L1 Instruction Memory Control Register (IMEM_CONTROL)

The processor core reads the instruction memory through the 64-bit wide instruction fetch bus. All addresses from this bus are 64-bit aligned. Each instruction fetch can return any combination of 16-, 32- or 64-bit instructions (for example, four 16-bit instructions, two 16-bit instructions and one 32-bit instruction, or one 64-bit instruction).

The pointer registers and index registers, which are described in Chapter 5, cannot access L1 Instruction Memory directly. A direct access to an address in instruction memory SRAM space generates an exception.
Write access to the L1 Instruction SRAM Memory must be made through the 64-bit wide system DMA port. Because the SRAM is implemented as a collection of single ported subbanks, the instruction memory is effectively dual ported.

Figure 6-3 on page 6-9 describes the bank architecture of the L1 Instruction Memory. As the figure shows, each 16K byte bank is made up of four 4K byte subbanks. In the figure, dotted lines indicate features that exist only on some Blackfin processors. Please refer to the hardware reference manual for your particular processor for more details.

While on some processors the EAB and DCB buses shown in Figure 6-3 connect directly to the EBIU and DMA controllers, on derivatives that feature multiple cores or on-chip L2 memories they must cross additional arbitration units. Also, these buses are wider than 16 bits on some parts. For details, refer to the specific Blackfin Processor Hardware Reference manual for your derivative.
Figure 6-3. L1 Instruction Memory Bank Architecture

The shaded blocks are not present on all derivatives. For more information, please refer to the corresponding hardware reference.
L1 Instruction Memory

L1 Instruction Cache

For information about cache terminology, see “Terminology” on page 6-74.

The L1 Instruction Memory may also be configured to contain a, 4-Way set associative instruction 16K byte cache. To improve the average access latency for critical code sections, each Way or line of the cache can be locked independently. When the memory is configured as cache, it cannot be accessed directly.

When cache is enabled, only memory pages further specified as cacheable by the CPLBs will be cached. When CPLBs are enabled, any memory location that is accessed must have an associated page definition available, or a CPLB exception is generated. CPLBs are described in “Memory Protection and Properties” on page 6-45.

Figure 6-4 on page 6-12 shows the overall Blackfin processor instruction cache organization.

Cache Lines

As shown in Figure 6-4, the cache consists of a collection of cache lines. Each cache line is made up of a tag component and a data component.

- The tag component incorporates a 20-bit address tag, least recently used (LRU) bits, a Valid bit, and a Line Lock bit.

- The data component is made up of four 64-bit words of instruction data.

The tag and data components of cache lines are stored in the tag and data memory arrays, respectively.
Memory

The address tag consists of the upper 18 bits plus bits 11 and 10 of the physical address. Bits 12 and 13 of the physical address are not part of the address tag. Instead, these bits are used to identify the 4K byte memory subbank targeted for the access.

The LRU bits are part of an LRU algorithm used to determine which cache line should be replaced if a cache miss occurs.

The Valid bit indicates the state of a cache line. A cache line is always valid or invalid.

- Invalid cache lines have their Valid bit cleared, indicating the line will be ignored during an address-tag compare operation.

- Valid cache lines have their Valid bit set, indicating the line contains valid instruction/data that is consistent with the source memory.

The tag and data components of a cache line are illustrated in Figure 6-5. Each 4K byte subbank provides the same structure.
Figure 6-4. Instruction Cache Organization Per Subbank
Memory

Memory

A cache hit occurs when the address for an instruction fetch request from the core matches a valid entry in the cache. Specifically, a cache hit is determined by comparing the upper 18 bits and bits 11 and 10 of the instruction fetch address to the address tags of valid lines currently stored in a cache set. The cache set (cache line across ways) is selected, using bits 9 through 5 of the instruction fetch address. If the address-tag compare operation results in a match in any of the four ways and the respective cache line is valid, a cache hit occurs. If the address-tag compare operation does not result in a match in any of the four ways or the respective line is not valid, a cache miss occurs.

When a cache miss occurs, the instruction memory unit generates a cache line fill access to retrieve the missing cache line from memory that is external to the core. The address for the external memory access is the address of the target instruction word. When a cache miss occurs, the core halts until the target instruction word is returned from external memory.

Figure 6-5. Cache Line – Tag and Data Portions

<table>
<thead>
<tr>
<th>TAG</th>
<th>LRUPRIO</th>
<th>LRU</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>WD 3</td>
<td>WD 2</td>
<td>WD 1</td>
<td>WD 0</td>
</tr>
</tbody>
</table>

WD - 64-BIT DATA WORD

TAG - 20-BIT ADDRESS TAG
LRUPRIO - LRU PRIORITY BIT FOR LINE LOCKING
LRU - LRU STATE
V - VALID BIT
L1 Instruction Memory

Cache Line Fills

A cache line fill consists of fetching 32 bytes of data from memory. The operation starts when the instruction memory unit requests a line-read data transfer on its external read-data port. This is a burst of four 64-bit words of data from the line fill buffer. The line fill buffer translates then to the bus width of the External Access Bus (EAB).

The address for the read transfer is the address of the target instruction word. When responding to a line-read request from the instruction memory unit, the external memory returns the target instruction word first. After it has returned the target instruction word, the next three words are fetched in sequential address order. This fetch wraps around if necessary, as shown in Table 6-1.

Table 6-1. Cache Line Word Fetching Order

<table>
<thead>
<tr>
<th>Target Word</th>
<th>Fetching Order for Next Three Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>WD0</td>
<td>WD0, WD1, WD2, WD3</td>
</tr>
<tr>
<td>WD1</td>
<td>WD1, WD2, WD3, WD0</td>
</tr>
<tr>
<td>WD2</td>
<td>WD2, WD3, WD0, WD1</td>
</tr>
<tr>
<td>WD3</td>
<td>WD3, WD0, WD1, WD2</td>
</tr>
</tbody>
</table>

Once the line fill has completed, the four 64-bit words have fixed order in the cache as shown in Figure 6-4. This avoids the need to save the lower 5 bits (byte select) of the address word along with the cache entry.
Memory

Line Fill Buffer

As the new cache line is retrieved from external memory, each 64-bit word is buffered in a four-entry line fill buffer before it is written to a 4K byte memory bank within L1 memory. The line fill buffer allows the core to access the data from the new cache line as the line is being retrieved from external memory, rather than having to wait until the line has been written into the cache. While the L1 port of the fill buffer is always 64 bits wide, the width of port to external or L2 memory varies between derivatives.

Cache Line Replacement

When the instruction memory unit is configured as cache, bits 9 through 5 of the instruction fetch address are used as the index to select the cache set for the tag-address compare operation. If the tag-address compare operation results in a cache miss, the Valid and LRU bits for the selected set are examined by a cache line replacement unit to determine the entry to use for the new cache line, that is, whether to use Way0, Way1, Way2, or Way3. See Figure 6-4, “Instruction Cache Organization Per Subbank,” on page 6-12.

The cache line replacement unit first checks for invalid entries (that is, entries having its Valid bit cleared). If only a single invalid entry is found, that entry is selected for the new cache line. If multiple invalid entries are found, the replacement entry for the new cache line is selected based on the following priority:

- Way0 first
- Way1 next
- Way2 next
- Way3 last
L1 Instruction Memory

For example:

- If Way3 is invalid and Ways0, 1, 2 are valid, Way3 is selected for the new cache line.
- If Ways0 and 1 are invalid and Ways2 and 3 are valid, Way0 is selected for the new cache line.
- If Ways2 and 3 are invalid and Ways0 and 1 are valid, Way2 is selected for the new cache line.

When no invalid entries are found, the cache replacement logic uses an LRU algorithm.

Instruction Cache Management

The system DMA controller and the core DAGs cannot access the instruction cache directly. By a combination of instructions and the use of core MMRs, it is possible to initialize the instruction tag and data arrays indirectly and provide a mechanism for instruction cache test, initialization, and debug.

The coherency of instruction cache must be explicitly managed. To accomplish this and ensure that the instruction cache fetches the latest version of any modified instruction space, invalidate instruction cache line entries, as required.

See “Instruction Cache Invalidation” on page 6-18.

Instruction Cache Locking by Line

The CPLB_LRUPRIO bits in the ICPLB_DATAx registers (see “Memory Protection and Properties” on page 6-45) are used to enhance control over which code remains resident in the instruction cache. When a cache line is filled, the state of this bit is stored along with the line’s tag. It is then used in conjunction with the LRU (least recently used) policy to determine which Way is victimized when all cache Ways are occupied when a new
cacheable line is fetched. This bit indicates that a line is of either “low” or “high” importance. In a modified LRU policy, a high can replace a low, but a low cannot replace a high. If all Ways are occupied by highs, an otherwise cacheable low will still be fetched for the core, but will not be cached. Fetched highs seek to replace unoccupied Ways first, then least recently used lows next, and finally other highs using the LRU policy. Lows can only replace unoccupied Ways or other lows, and do so using the LRU policy. If all previously cached highs ever become less important, they may be simultaneously transformed into lows by writing to the LRU-PRIEST bit in the IMEM_CONTROL register (see page 6-5).

Instruction Cache Locking by Way

The instruction cache has four independent lock bits (ILOC[3:0]) that control each of the four Ways of the instruction cache. When the cache is enabled, L1 Instruction Memory has four Ways available. Setting the lock bit for a specific Way prevents that Way from participating in the LRU replacement policy. Thus, a cached instruction with its Way locked can only be removed using an IFLUSH instruction, or a “back door” MMR assisted manipulation of the tag array.

An example sequence is provided below to demonstrate how to lock down Way0:

- If the code of interest may already reside in the instruction cache, invalidate the entire cache first (for an example, see “Instruction Cache Invalidation” on page 6-18).
- Disable interrupts, if required, to prevent interrupt service routines (ISRs) from potentially corrupting the locked cache.
- Set the locks for the other Ways of the cache by setting ILOC[3:1]. Only Way0 of the instruction cache can now be replaced by new code.
L1 Instruction Memory

- Execute the code of interest. Any cacheable exceptions, such as exit code, traversed by this code execution are also locked into the instruction cache.

- Upon exit of the critical code, clear ILOC[3:1] and set ILOC[0]. The critical code (and the instructions which set ILOC[0]) is now locked into Way0.

- Re-enable interrupts, if required.

If all four Ways of the cache are locked, then further allocation into the cache is prevented.

Instruction Cache Invalidation

The instruction cache can be invalidated by address, cache line, or complete cache. The IFLUSH instruction can explicitly invalidate cache lines based on their line addresses. The target address of the instruction is generated from the P-registers. Because the instruction cache should not contain modified (dirty) data, the cache line is simply invalidated, and not “flushed.”

In the following example, the P2 register contains the address of a valid memory location. If this address has been brought into cache, the corresponding cache line is invalidated after the execution of this instruction.

Example of ICACHE instruction:

```assembly
iflush [ p2 ] ; /* Invalidate cache line containing address that P2 points to */
```

Because the IFLUSH instruction is used to invalidate a specific address in the memory map and its corresponding cache-line, it is most useful when the buffer being invalidated is less than the cache size. For more information about the IFLUSH instruction, see Chapter 17, “Cache Control.” A second technique can be used to invalidate larger portions of the cache directly. This second technique directly invalidates Valid bits by setting...
the Invalid bit of each cache line to the invalid state. To implement this technique, additional MMRs (ITEST_COMMAND and ITEST_DATA[1:0]) are available to allow arbitrary read/write of all the cache entries directly. This method is explained in the next section.

For invalidating the complete instruction cache, a third method is available. By clearing the IMC bit in the IMEM_CONTROL register (see Figure 6-2, “L1 Instruction Memory Control Register,” on page 6-7), all Valid bits in the instruction cache are set to the invalid state. A second write to the IMEM_CONTROL register to set the IMC bit configures the instruction memory as cache again. An SSYNC instruction should be run before invalidating the cache and a CSYNC instruction should be inserted after each of these operations.

Instruction Test Registers

The Instruction Test registers allow arbitrary read/write of all L1 cache entries directly. They make it possible to initialize the instruction tag and data arrays and to provide a mechanism for instruction cache test, initialization, and debug.

When the Instruction Test Command register (ITEST_COMMAND) is used, the L1 cache data or tag arrays are accessed, and data is transferred through the Instruction Test Data registers (ITEST_DATA[1:0]). The ITEST_DATAx registers contain either the 64-bit data that the access is to write to or the 64-bit data that was read during the access. The lower 32 bits are stored in the ITEST_DATA[0] register, and the upper 32 bits are stored in the ITEST_DATA[1] register. When the tag arrays are accessed, ITEST_DATA[0] is used. Graphical representations of the ITEST registers begin with Figure 6-6 on page 6-21.
Instruction Test Registers

The following figures describe the ITEST registers:

- Figure 6-6, “Instruction Test Command Register,” on page 6-21
- Figure 6-7, “Instruction Test Data 1 Register,” on page 6-22
- Figure 6-8, “Instruction Test Data 0 Register,” on page 6-23

Access to these registers is possible only in Supervisor or Emulation mode. When writing to ITEST registers, always write to the ITEST_DATAx registers first, then the ITEST_COMMAND register. When reading from ITEST registers, reverse the sequence—read the ITEST_COMMAND register first, then the ITEST_DATAx registers.
**Memory**

**ITEST_COMMAND Register**

When the Instruction Test Command register (ITEST_COMMAND) is written to, the L1 cache data or tag arrays are accessed, and the data is transferred through the Instruction Test Data registers (ITEST_DATA[1:0]).

![Figure 6-6. Instruction Test Command Register](image-url)

**Instruction Test Command Register (ITEST_COMMAND)**

- **Reset = 0x0000 0000**
- **WAYSEL[1:0] (Access Way)**
  - 00 - Access Way0
  - 01 - Access Way1
  - 10 - Access Way2
  - 11 - Access Way3
  (Address bits [11:10] in SRAM)
- **SBNK[1:0] (Subbank Access)**
  - 00 - Access subbank 0
  - 01 - Access subbank 1
  - 10 - Access subbank 2
  - 11 - Access subbank 3
  (Address bits [13:12] in SRAM)
- **RW (Read/Write Access)**
  - 0 - Read access
  - 1 - Write access
- **TAGSELB (Array Access)**
  - 0 - Access tag array
  - 1 - Access data array
- **SET[4:0] (Set Index)**
  - Selects one of 32 sets
  (Address bits [9:5] in SRAM)
- **DW[1:0] (Double Word Index)**
  - Selects one of four 64-bit double words in a 256-bit line (Address bits [4:3] in SRAM)
Instruction Test Registers

**ITEST_DATA1 Register**

Instruction Test Data registers ([ITEST_DATA][1:0]) are used to access L1 cache data arrays. They contain either the 64-bit data that the access is to write to or the 64-bit data that the access is to read from. The Instruction Test Data 1 register ([ITEST_DATA1]) stores the upper 32 bits.

**Instruction Test Data 1 Register (ITEST_DATA1)**

Used to access L1 cache data arrays and tag arrays. When accessing a data array, stores the upper 32 bits of 64-bit words of instruction data to be written to or read from by the access. See "Cache Lines" on page 6-10.

![Diagram of Instruction Test Data 1 Register](image)

**Figure 6-7. Instruction Test Data 1 Register**


Memory

**ITEST_DATA0 Register**

The Instruction Test Data 0 register (ITEST_DATA0) stores the lower 32 bits of the 64-bit data to be written to or read from by the access. The ITEST_DATA0 register is also used to access tag arrays. This register also contains the Valid and Dirty bits, which indicate the state of the cache line.

**Instruction Test Data 0 Register (ITEST_DATA0)**

Used to access L1 cache data arrays and tag arrays. When accessing a data array, stores the lower 32 bits of 64-bit words of instruction data to be written to or read from by the access. See "Cache Lines" on page 6-10.

![Instruction Test Data 0 Register](image)

0xFFF0 1400

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16</th>
<th>Reset = Undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>X X X X X X X X X X X X X X X X X X X X X X X X</td>
<td>Data[31:16]</td>
</tr>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
<tr>
<td>X X X X X X X X X X X X X X X X X X X X X X X X</td>
<td>Data[15:0]</td>
</tr>
</tbody>
</table>

Used to access the L1 cache tag arrays. The address tag consists of the upper 18 bits and bits 11 and 10 of the physical address. See "Cache Lines" on page 6-10.

![Instruction Test Data 0 Register](image)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16</th>
<th>Reset = Undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>X X X X X X X X X X X X X X X X X X X X X X X X</td>
<td>Tag[19:4]</td>
</tr>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
<tr>
<td>X X X X X X X X X X X X X X X X X X X X X X X X</td>
<td>Valid</td>
</tr>
</tbody>
</table>

**ITEST_DATA0 Register (Continued)**

Valid

0 - Cache line is not valid
1 - Cache line contains valid data

LRUPRIO

0 - LRUPRIO is cleared for this entry
1 - LRUPRIO is set for this entry. See "ICPLB_DATAx Registers" on page 6-55 and "IMEM_CONTROL Register" on page 6-5.

---

Figure 6-8. Instruction Test Data 0 Register
L1 Data Memory

The L1 data SRAM/cache is constructed from single-ported subsections, but organized to reduce the likelihood of access collisions. This organization results in apparent multi-ported behavior. When there are no collisions, this L1 data traffic could occur in a single core clock cycle:

- Two 32-bit data loads
- One pipelined 32-bit data store
- One DMA I/O, up to 64 bits
- One 64-bit cache fill/victim access

L1 Data Memory can be used only to store data.

DMEM_CONTROL Register

The Data Memory Control register (DMEM_CONTROL) contains control bits for the L1 Data Memory.

The PORT_PREF1 bit selects the data port used to process DAG1 non-cacheable L2 fetches. Cacheable fetches are always processed by the data port physically associated with the targeted cache memory. Steering DAG0, DAG1, and cache traffic to different ports optimizes performance by keeping the queue to L2 memory full.
Memory

Data Memory Control Register (DMEM_CONTROL)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>ENDCPLB (Data Cacheability Protection Lookaside Buffer Enable)</td>
<td>0x0000 1001</td>
</tr>
<tr>
<td>15</td>
<td>DMC[1:0] (L1 Data Memory Configure)</td>
<td>0xFFE0 0004</td>
</tr>
<tr>
<td>0-14</td>
<td>DCBS (L1 Data Cache Bank Select)</td>
<td></td>
</tr>
<tr>
<td>0-14</td>
<td>PORT_PREF1 (DAG1 Port Preference)</td>
<td></td>
</tr>
<tr>
<td>0-14</td>
<td>PORT_PREF0 (DAG0 Port Preference)</td>
<td></td>
</tr>
</tbody>
</table>

0 - DAG1 non-cacheable fetches use port A
1 - DAG1 non-cacheable fetches use port B

0 - DAG0 non-cacheable fetches use port A
1 - DAG0 non-cacheable fetches use port B


0 - Address bit 14 is used to select Bank A or B for cache access. If bit 14 of address is 1, select L1 Data Memory Data Bank A; if bit 14 of address is 0, select L1 Data Memory Data Bank B.
1 - Address bit 23 is used to select Bank A or B for cache access. If bit 23 of address is 1, select L1 Data Memory Data Bank A; if bit 23 of address is 0, select L1 Data Memory Data Bank B.

See "Example of Mapping Cacheable Address Space" on page 6-30.

Figure 6-9. L1 Data Memory Control Register
L1 Data Memory

The `PORT_PREF0` bit selects the data port used to process DAG0 non-cacheable L2 fetches. Cacheable fetches are always processed by the data port physically associated with the targeted cache memory. Steering DAG0, DAG1, and cache traffic to different ports optimizes performance by keeping the queue to L2 memory full.

For optimal performance with dual DAG reads, DAG0 and DAG1 should be configured for different ports. For example, if `PORT_PREF0` is configured as 1, then `PORT_PREF1` should be programmed to 0.

The `DCBS` bit provides some control over which addresses alias into the same set. This bit can be used to affect which addresses tend to remain resident in cache by avoiding victimization of repetitively used sets. It has no affect unless both Data Bank A and Data Bank B are serving as cache (bits `DMC[1:0]` in this register are set to 11).

The `ENDCPLB` bit is used to enable/disable the 16 Cacheability Protection Lookaside Buffers (CPLBs) used for data (see “L1 Data Cache” on page 6-29). Data CPLBs are disabled by default after reset. When disabled, only minimal address checking is performed by the L1 memory interface. This minimal checking generates an exception when the processor:

- Addresses nonexistent (reserved) L1 memory space
- Attempts to perform a nonaligned memory access
- Attempts to access MMR space either using DAG1 or when in User mode

CPLBs must be disabled using this bit prior to updating their descriptors (registers `DCPLB_DATAx` and `DCPLB_ADDRx`). Note that since load store ordering is weak (see “Ordering of Loads and Stores” on page 6-67), disabling CPLBs should be preceded by a `CSYNC` instruction.

When enabling or disabling cache or CPLBs, immediately follow the write to `DMEM_CONTROL` with a `SSYNC` to ensure proper behavior.
By default after reset, all L1 Data Memory serves as SRAM. The $\text{DMC}[1:0]$ bits can be used to reserve portions of this memory to serve as cache instead. Reserving memory to serve as cache does not enable L2 memory accesses to be cached. To do this, CPLBs must also be enabled (using the $\text{ENDCPLB}$ bit) and CPLB descriptors (registers $\text{DCPLB\_DATAx}$ and $\text{DCPLB\_ADDRx}$) must specify chosen memory pages as cache-enabled.

By default after reset, cache and CPLB address checking is disabled.

To ensure proper behavior and future compatibility, all reserved bits in this register must be set to 0 whenever this register is written.

**L1 Data SRAM**

Accesses to SRAM do not collide unless all of the following are true: the accesses are to the same 32-bit word polarity (address bits 2 match), the same 4K byte subbank (address bits 13 and 12 match), the same 16K byte half bank (address bits 16 match), and the same bank (address bits 21 and 20 match). When an address collision is detected, access is nominally granted first to the DAGs, then to the store buffer, and finally to the DMA and cache fill/victim traffic. To ensure adequate DMA bandwidth, DMA is given highest priority if it has been blocked for more than 16 sequential core clock cycles, or if a second DMA I/O is queued before the first DMA I/O is processed.

Figure 6-10 shows the L1 Data Memory architecture. In the figure, dotted lines indicate features that exist only on some Blackfin processors. Please refer to the hardware reference manual for your particular processor for more details. While on some processors the EAB and DCB buses shown in Figure 6-10 connect directly to EBIU and DMA controllers, on derivatives that feature multiple cores or on-chip L2 memories they have to cross additional arbitration units. Also, these buses are wider than 16 bits on some parts. For details, refer to the specific Blackfin Processor Hardware Reference manual for your derivative.
Figure 6-10. L1 Data Memory Architecture

The shaded blocks are not present on all derivatives. For more information, please refer to the corresponding processor hardware reference.
## L1 Data Cache

For definitions of cache terminology, see “Terminology” on page 6-74.

Unlike instruction cache, which is 4-Way set associative, data cache is 2-Way set associative. When two banks are available and enabled as cache, additional sets rather than Ways are created. When both Data Bank A and Data Bank B have memory serving as cache, the DCBS bit in the DMEM_CONTROL register may be used to control which half of all address space is handled by which bank of cache memory. The DCBS bit selects either address bit 14 or 23 to steer traffic between the cache banks. This provides some control over which addresses alias into the same set. It may therefore be used to affect which addresses tend to remain resident in cache by avoiding victimization of repetitively used sets.

Accesses to cache do not collide unless they are to the same 4K byte sub-bank, the same half bank, and to the same bank. Cache has less apparent multi-ported behavior than SRAM due to the overhead in maintaining tags. When cache addresses collide, access is granted first to the DTEST register accesses, then to the store buffer, and finally to cache fill/victim traffic.

Three different cache modes are available.

- Write-through with cache line allocation only on reads
- Write-through with cache line allocation on both reads and writes
- Write-back which allocates cache lines on both reads and writes

Cache mode is selected by the DCPLB descriptors (see “Memory Protection and Properties” on page 6-45). Any combination of these cache modes can be used simultaneously since cache mode is selectable for each memory page independently.
L1 Data Memory

If cache is enabled (controlled by bits $\text{DMC}[1:0]$ in the $\text{DMEM\_CONTROL}$ register), data CPLBs should also be enabled (controlled by $\text{ENDCPLB}$ bit in the $\text{DMEM\_CONTROL}$ register). Only memory pages specified as cacheable by data CPLBs will be cached. The default behavior when data CPLBs are disabled is for nothing to be cached.

⚠️ Erroneous behavior can result when MMR space is configured as cacheable by data CPLBs, or when data banks serving as L1 SRAM are configured as cacheable by data CPLBs.

Example of Mapping Cacheable Address Space

An example of how the cacheable address space maps into two data banks follows.

When both banks are configured as cache they operate as two independent, 16K byte, 2-Way set associative caches that can be independently mapped into the Blackfin processor address space.

If both data banks are configured as cache, the $\text{DCBS}$ bit in the $\text{DMEM\_CONTROL}$ register designates Address bit $A[14]$ or $A[23]$ as the cache selector. Address bit $A[14]$ or $A[23]$ selects the cache implemented by Data Bank A or the cache implemented by Data Bank B.

- If $\text{DCBS} = 0$, then $A[14]$ is part of the address index, and all addresses in which $A[14] = 0$ use Data Bank B. All addresses in which $A[14] = 1$ use Data Bank A.

In this case, $A[23]$ is treated as merely another bit in the address that is stored with the tag in the cache and compared for hit/miss processing by the cache.
Memory

- If \( \text{DCBS} = 1 \), then \( A[23] \) is part of the address index, and all addresses where \( A[23] = 0 \) use Data Bank B. All addresses where \( A[23] = 1 \) use Data Bank A.

In this case, \( A[14] \) is treated as merely another bit in the address that is stored with the tag in the cache and compared for hit/miss processing by the cache.

The result of choosing \( \text{DCBS} = 0 \) or \( \text{DCBS} = 1 \) is:

- If \( \text{DCBS} = 0 \), \( A[14] \) selects Data Bank A instead of Data Bank B.

Alternating 16K byte pages of memory map into each of the two 16K byte caches implemented by the two data banks. Consequently:

Any data in the first 16K byte of memory could be stored only in Data Bank B.

Any data in the next address range (16K byte through 32K byte) – 1 could be stored only in Data Bank A.

Any data in the next range (32K byte through 48K byte) – 1 would be stored in Data Bank B.

Alternate mapping would continue.

As a result, the cache operates as if it were a single, contiguous, 2-Way set associative 32K byte cache. Each Way is 16K byte long, and all data elements with the same first 14 bits of address index to a unique set in which up to two elements can be stored (one in each Way).
L1 Data Memory

- If $DCBS = 1$, $A[23]$ selects Data Bank A instead of Data Bank B.

With $DCBS = 1$, the system functions more like two independent caches, each a 2-Way set associative 16K byte cache. Each Bank serves an alternating set of 8M byte blocks of memory.

For example, Data Bank B caches all data accesses for the first 8M byte of memory address range. That is, every 8M byte of range vies for the two line entries (rather than every 16K byte repeat). Likewise, Data Bank A caches data located above 8M byte and below 16M byte.

For example, if the application is working from a data set that is 1M byte long and located entirely in the first 8M byte of memory, it is effectively served by only half the cache, that is, by Data Bank B (a 2-Way set associative 16K byte cache). In this instance, the application never derives any benefit from Data Bank A.

For most applications, it is best to operate with $DCBS = 0$.

However, if the application is working from two data sets, located in two memory spaces at least 8M byte apart, closer control over how the cache maps to the data is possible. For example, if the program is doing a series of dual MAC operations in which both DAGs are accessing data on every cycle, by placing DAG0’s data set in one block of memory and DAG1’s data set in the other, the system can ensure that:

- DAG0 gets its data from Data Bank A for all of its accesses and
- DAG1 gets its data from Data Bank B.

This arrangement causes the core to use both data buses for cache line transfer and achieves the maximum data bandwidth between the cache and the core.
Figure 6-11 shows an example of how mapping is performed when \( \text{DCBS} = 1 \).

The \( \text{DCBS} \) selection can be changed dynamically; however, to ensure that no data is lost, first flush and invalidate the entire cache.

**Data Cache Access**

The Cache Controller tests the address from the DAGs against the tag bits. If the logical address is present in L1 cache, a cache hit occurs, and the data is accessed in L1. If the logical address is not present, a cache miss occurs, and the memory transaction is passed to the next level of memory via the system interface. The line index and replacement policy for the Cache Controller determines the cache tag and data space that are allocated for the data coming back from external memory.
A data cache line is in one of three states: invalid, exclusive (valid and clean), and modified (valid and dirty). If valid data already occupies the allocated line and the cache is configured for write-back storage, the controller checks the state of the cache line and treats it accordingly:

- If the state of the line is exclusive (clean), the new tag and data write over the old line.
- If the state of the line is modified (dirty), then the cache contains the only valid copy of the data.

If the line is dirty, the current contents of the cache are copied back to external memory before the new data is written to the cache.

The processor provides victim buffers and line fill buffers. These buffers are used if a cache load miss generates a victim cache line that should be replaced. The line fill operation goes to external memory. The data cache performs the line fill request to the system as critical (or requested) word first, and forwards that data to the waiting DAG as it updates the cache line. In other words, the cache performs critical word forwarding.

The data cache supports hit-under-a-store miss, and hit-under-a-prefetch miss. In other words, on a write-miss or execution of a \texttt{PREFETCH} instruction that misses the cache (and is to a cacheable region), the instruction pipeline incurs a minimum of a 4-cycle stall. Furthermore, a subsequent load or store instruction can hit in the L1 cache while the line fill completes.

Interrupts of sufficient priority (relative to the current context) cancel a stalled load instruction. Consequently, if the load operation misses the L1 Data Memory cache and generates a high latency line fill operation on the system interface, it is possible to interrupt the core, causing it to begin processing a different context. The system access to fill the cache line is not cancelled, and the data cache is updated with the new data before any further cache miss operations to the respective data bank are serviced. For more information see “Exceptions” on page 4-47.
Cache Write Method

Cache write memory operations can be implemented by using either a write-through method or a write-back method:

- For each store operation, write-through caches initiate a write to external memory immediately upon the write to cache.

  If the cache line is replaced or explicitly flushed by software, the contents of the cache line are invalidated rather than written back to external memory.

- A write-back cache does not write to external memory until the line is replaced by a load operation that needs the line.

The L1 Data Memory employs a full cache line width copyback buffer on each data bank. In addition, a two-entry write buffer in the L1 Data Memory accepts all stores with cache inhibited or store-through protection. An SSYNC instruction flushes the write buffer.

IPRIO Register and Write Buffer Depth

The Interrupt Priority register (IPRIO) can be used to control the size of the write buffer on Port A (see “L1 Data Memory Architecture” on page 6-28).

The IPRIO[3:0] bits can be programmed to reflect the low priority interrupt watermark. When an interrupt occurs, causing the processor to vector from a low priority interrupt service routine to a high priority interrupt service routine, the size of the write buffer increases from two to eight 32-bit words deep. This allows the interrupt service routine to run and post writes without an initial stall, in the case where the write buffer was already filled in the low priority interrupt routine. This is most useful
when posted writes are to a slow external memory device. When returning from a high priority interrupt service routine to a low priority interrupt service routine or user mode, the core stalls until the write buffer has completed the necessary writes to return to a two-deep state. By default, the write buffer is a fixed two-deep FIFO.

**Interrupt Priority Register (IPRIO)**

![Figure 6-12. Interrupt Priority Register](image-url)

- **IPRIO_MARK[0:3]** (Priority Watermark)
  - 0000 - Default, all interrupts are low priority
  - 0001 - Interrupts 15 through 1 are low priority, interrupt 0 is considered high priority
  - 0010 - Interrupts 15 through 2 are low priority, interrupts 1 and 0 are considered high priority
  - ...  
  - 1110 - Interrupts 15 and 14 are low priority, interrupts 13 through 0 are considered high priority
  - 1111 - Interrupt 15 is low priority, all others are considered high priority
Data Cache Control Instructions

The processor defines three data cache control instructions that are accessible in User and Supervisor modes. The instructions are `PREFETCH`, `FLUSH`, and `FLUSHINV`. Examples of each of these instructions can be found in Chapter 17, “Cache Control.”

- **PREFETCH** (Data Cache Prefetch) attempts to allocate a line into the L1 cache. If the prefetch hits in the cache, generates an exception, or addresses a cache inhibited region, `PREFETCH` functions like a `NOP`. It can be used to begin a data fetch prior to when the processor needs the data, to improve performance.

- **FLUSH** (Data Cache Flush) causes the data cache to synchronize the specified cache line with external memory. If the cached data line is dirty, the instruction writes the line out and marks the line clean in the data cache. If the specified data cache line is already clean or does not exist, `FLUSH` functions like a `NOP`.

- **FLUSHINV** (Data Cache Line Flush and Invalidate) causes the data cache to perform the same function as the `FLUSH` instruction and then invalidate the specified line in the cache. If the line is in the cache and dirty, the cache line is written out to external memory. The Valid bit in the cache line is then cleared. If the line is not in the cache, `FLUSHINV` functions like a `NOP`.

If software requires synchronization with system hardware, place an `SSYNC` instruction after the `FLUSH` instruction to ensure that the flush operation has completed. If ordering is desired to ensure that previous stores have been pushed through all the queues, place an `SSYNC` instruction before the `FLUSH`.
Data Test Registers

Data Cache Invalidation

Besides the FLUSHINV instruction, explained in the previous section, two additional methods are available to invalidate the data cache when flushing is not required. The first technique directly invalidates Valid bits by setting the Invalid bit of each cache line to the invalid state. To implement this technique, additional MMRs (DTEST_COMMAND and DTEST_DATA[1:0]) are available to allow arbitrary reads/writes of all the cache entries directly. This method is explained in the next section.

For invalidating the complete data cache, a second method is available. By clearing the DMC[1:0] bits in the DMEM_CONTROL register (see Figure 6-9, “L1 Data Memory Control Register,” on page 6-25), all Valid bits in the data cache are set to the invalid state. A second write to the DMEM_CONTROL register to set the DMC[1:0] bits to their previous state then configures the data memory back to its previous cache/SRAM configuration. An SSYNC instruction should be run before invalidating the cache and a CSYNC instruction should be inserted after each of these operations.

Data Test Registers

Like L1 Instruction Memory, L1 Data Memory contains additional MMRs to allow arbitrary reads/writes of all cache entries directly. The registers provide a mechanism for data cache test, initialization, and debug.

When the Data Test Command register (DTEST_COMMAND) is written to, the L1 cache data or tag arrays are accessed and data is transferred through the Data Test Data registers (DTEST_DATA[1:0]). The DTEST_DATA[1:0] registers contain the 64-bit data to be written, or they contain the destination for the 64-bit data read. The lower 32 bits are stored in the DTEST_DATA[0] register and the upper 32 bits are stored in the DTEST_DATA[1] register. When the tag arrays are being accessed, then the DTEST_DATA[0] register is used.

A CSYNC instruction is required after writing the DTEST_COMMAND MMR.
These figures describe the `DTEST` registers.

- Figure 6-13, “Data Test Command Register,” on page 6-40
- Figure 6-14, “Data Test Data 1 Register,” on page 6-41
- Figure 6-15, “Data Test Data 0 Register,” on page 6-42

Access to these registers is possible only in Supervisor or Emulation mode. When writing to `DTEST` registers, always write to the `DTEST_DATA` registers first, then the `DTEST_COMMAND` register.

**DTEST_COMMAND Register**

When the Data Test Command register (`DTEST_COMMAND`) is written to, the L1 cache data or tag arrays are accessed, and the data is transferred through the Data Test Data registers (`DTEST_DATA[1:0]`).

![Info icon] The Data/Instruction Access bit allows direct access via the `DTEST_COMMAND` MMR to L1 instruction SRAM.
Figure 6-13. Data Test Command Register
Memory

DTEST_DATA1 Register

Data Test Data registers (DTEST_DATA[1:0]) contain the 64-bit data to be written, or they contain the destination for the 64-bit data read. The Data Test Data 1 register (DTEST_DATA1) stores the upper 32 bits.

Data Test Data 1 Register (DTEST_DATA1)

![Data Test Data 1 Register Diagram]

When accessing tag arrays, all bits are reserved.

![Tag Array Bit Diagram]

Figure 6-14. Data Test Data 1 Register
Data Test Registers

DTEST_DATA0 Register

The Data Test Data 0 register (DTEST_DATA0) stores the lower 32 bits of the 64-bit data to be written, or it contains the lower 32 bits of the destination for the 64-bit data read. The DTEST_DATA0 register is also used to access the tag arrays and contains the Valid and Dirty bits, which indicate the state of the cache line.

![Figure 6-15. Data Test Data 0 Register](image-url)

Used to access the L1 cache tag arrays. The address tag consists of the upper 18 bits and bit 11 of the physical address. See “Cache Lines” on page 6-10.

![Figure 6-15. Data Test Data 0 Register](image-url)

Used to access the L1 cache tag arrays. The address tag consists of the upper 18 bits and bit 11 of the physical address. See “Cache Lines” on page 6-10.

Figure 6-15. Data Test Data 0 Register
On-chip Level 2 (L2) Memory

Some Blackfin processors provide additional low-latency and high-bandwidth SRAM on chip, called Level 2 (L2) memory. L2 memory runs at CCLK clock rate, but takes multiple CCLK cycles to access.

Simultaneous access to the multibanked, on-chip L2 memory architecture from the core(s) and system DMA can occur in parallel, provided that they access different banks. A fixed-priority arbitration scheme resolves conflicts. The on-chip system DMA controllers share a dedicated 32-bit data path into the L2 memory system. This interface operates at the SCLK frequency. Dedicated L2 access from the processor core is also supported.

Derivatives with on-chip L2 memory provide not only the plain memory itself. They also provide proper bus and DMA infrastructure. Wide buses between L1 and L2 memory guarantee high data throughput. A dedicated DMA controller, called IMDMA, supports data exchange between internal memories.

The cores and IMDMA share a dedicated, low latency, 64-bit data path into the L2 SRAM memory. At a core clock frequency of 600 MHz, the peak data transfer rate across this interface is 4.8 GB/second.

On-chip L2 Bank Access

Two L2 access ports, a processor core port and a system port, are provided to allow concurrent access to the L2 memory, provided that the two ports access different memory sub-banks. If simultaneous access to the same memory sub-bank is attempted, collision detection logic in the L2 provides arbitration. This is a fixed priority arbiter; the DMA port always has the highest priority, unless the core is granted access to the sub-bank for a burst transfer. In this case, the L2 finishes the burst transfer before the system bus is granted access.
On-chip Level 2 (L2) Memory

Latency

When cache is enabled, the bus between the core and L2 memory is fully pipelined for contiguous burst transfers. The cache line fill from on-chip memory behaves the same for instruction and data fetches. Operations that miss the cache trigger a cache line replacement. This replacement fills one 256-bit (32-byte) line with four 64-bit reads. Under this condition, the L1 cache line fills from the L2 SRAM in \(9+2+2+2=15\) core cycles. In other words, after nine core cycles, the first 64-bit (8-byte) fill is available for the processor. Figure 6-16 on page 6-44 shows an example of L2 latency with cache on.

![Figure 6-16. L2 Latency With Cache On](image)

In this example, at the end of 15 core cycles, 32 bytes of instructions or data have been brought into cache and are available to the sequencer. If all the instructions contain 16 bits, sixteen instructions are brought into cache at the end of 15 core cycles. In addition, the first instruction that is
part of the cache-line fill executes on the tenth cycle; the second instruction executes on the eleventh cycle, and the third instruction executes on the twelfth cycle—all of them in parallel with the cache line fill.

Each cache line fill is aligned on a 32-byte boundary. When the requested instruction or data is not 32-byte aligned, the requested item is always loaded in the first read; each read is forwarded to the core as the line is filled. Sequential memory accesses miss the cache only when they reach the end of a cache line.

When on-chip L2 memory is configured as non-cacheable, instruction fetches and data fetches occur in 64-bit fills. In this case, each fill takes seven core cycles to complete. As shown in Figure 6-17 on page 6-46, on-chip L2 memory is configured as non-cacheable. To illustrate the concept of L2 latency with cache off, simple instructions are used that do not require additional external data fetches. In this case, consecutive instructions are issued on consecutive core cycles if multiple instructions are brought into the core in a given fetch.

**Memory Protection and Properties**

This section describes the Memory Management Unit (MMU), memory pages, CPLB management, MMU management, and CPLB registers.

**Memory Management Unit**

The Blackfin processor contains a page based Memory Management Unit (MMU). This mechanism provides control over cacheability of memory ranges, as well as management of protection attributes at a page level. The MMU provides great flexibility in allocating memory and I/O resources between tasks, with complete control over access rights and cache behavior.
The MMU is implemented as two 16-entry Content Addressable Memory (CAM) blocks. Each entry is referred to as a Cacheability Protection Lookaside Buffer (CPLB) descriptor. When enabled, every valid entry in the MMU is examined on any fetch, load, or store operation to determine whether there is a match between the address being requested and the page described by the CPLB entry. If a match occurs, the cacheability and protection attributes contained in the descriptor are used for the memory transaction with no additional cycles added to the execution of the instruction.

Because the L1 memories are separated into instruction and data memories, the CPLB entries are also divided between instruction and data CPLBs. Sixteen CPLB entries are used for instruction fetch requests; these are called ICPLBs. Another sixteen CPLB entries are used for data transactions; these are called DCPLBs. The ICPLBs and DCPLBs are enabled by setting the appropriate bits in the L1 Instruction Memory Control
(IMEM_CONTROL) and L1 Data Memory Control (DMEM_CONTROL) registers, respectively. These registers are shown in Figure 6-2 on page 6-7 and Figure 6-9 on page 6-25, respectively.

Each CPLB entry consists of a pair of 32-bit values. For instruction fetches:

- \( ICPLB\_ADDR[n] \) defines the start address of the page described by the CPLB descriptor.
- \( ICPLB\_DATA[n] \) defines the properties of the page described by the CPLB descriptor.

For data operations:

- \( DCPLB\_ADDR[m] \) defines the start address of the page described by the CPLB descriptor.
- \( DCPLB\_DATA[m] \) defines the properties of the page described by the CPLB descriptor.

There are two default CPLB descriptors for data accesses to the scratchpad data memory and to the system and core MMR space. These default descriptors define the above space as non-cacheable, so that additional CPLBs do not need to be set up for these regions of memory.

If valid CPLBs are set up for this space, the default CPLBs are ignored.
Memory Protection and Properties

Memory Pages

The 4G byte address space of the processor can be divided into smaller ranges of memory or I/O referred to as memory pages. Every address within a page shares the attributes defined for that page. The architecture supports four different page sizes:

- 1K byte
- 4K byte
- 1M byte
- 4M byte

Different page sizes provide a flexible mechanism for matching the mapping of attributes to different kinds of memory and I/O.

Memory Page Attributes

Each page is defined by a two-word descriptor, consisting of an address descriptor word $\text{xCPLB_ADDR}[n]$ and a properties descriptor word $\text{xCPLB_DATA}[n]$. The address descriptor word provides the base address of the page in memory. Pages must be aligned on page boundaries that are an integer multiple of their size. For example, a 4M byte page must start on an address divisible by 4M byte; whereas a 1K byte page can start on any 1K byte boundary. The second word in the descriptor specifies the other properties or attributes of the page. These properties include:

- Page size
  - 1K byte, 4K byte, 1M byte, 4M byte
- Cacheable/non-cacheable
  
  Accesses to this page use the L1 cache or bypass the cache.
Memory

- If cacheable: write-through/write-back

  Data writes propagate directly to memory or are deferred until the cache line is reallocated. If write-through, allocate on read only, or read and write.

- Dirty/modified

  The data in this page in memory has changed since the CPLB was last loaded. This must be managed by software and does not change status automatically.

- Supervisor write access permission

  - Enables or disables writes to this page when in Supervisor mode.
  - Data pages only.

- User write access permission

  - Enables or disables writes to this page when in User mode.
  - Data pages only.

- User read access permission

  Enables or disables reads from this page when in User mode.

- Valid

  Check this bit to determine whether this is valid CPLB data.

- Lock

  Keep this entry in MMR; do not participate in CPLB replacement policy.
Page Descriptor Table

For memory accesses to utilize the cache when CPLBs are enabled for instruction access, data access, or both, a valid CPLB entry must be available in an MMR pair. The MMR storage locations for CPLB entries are limited to 16 descriptors for instruction fetches and 16 descriptors for data load and store operations.

For small and/or simple memory models, it may be possible to define a set of CPLB descriptors that fit into these 32 entries, cover the entire addressable space, and never need to be replaced. This type of definition is referred to as a static memory management model.

However, operating environments commonly define more CPLB descriptors to cover the addressable memory and I/O spaces than will fit into the available on-chip CPLB MMRs. When this happens, a memory-based data structure, called a Page Descriptor Table, is used; in it can be stored all the potentially required CPLB descriptors. The specific format for the Page Descriptor Table is not defined as part of the Blackfin processor architecture. Different operating systems, which have different memory management models, can implement Page Descriptor Table structures that are consistent with the OS requirements. This allows adjustments to be made between the level of protection afforded versus the performance attributes of the memory-management support routines.

CPLB Management

When the Blackfin processor issues a memory operation for which no valid CPLB (cacheability protection lookaside buffer) descriptor exists in an MMR pair, an exception occurs. This exception places the processor into Supervisor mode and vectors to the MMU exception handler (see
“Exceptions” on page 4-47 for more information). The handler is typically part of the operating system (OS) kernel that implements the CPLB replacement policy.

Before CPLBs are enabled, valid CPLB descriptors must be in place for both the Page Descriptor Table and the MMU exception handler. The LOCK bits of these CPLB descriptors are commonly set so they are not inadvertently replaced in software.

The handler uses the faulting address to index into the Page Descriptor Table structure to find the correct CPLB descriptor data to load into one of the on-chip CPLB register pairs. If all on-chip registers contain valid CPLB entries, the handler selects one of the descriptors to be replaced, and the new descriptor information is loaded. Before loading new descriptor data into any CPLBs, the corresponding group of sixteen CPLBs must be disabled using:

- The Enable DCPLB (ENDCPLB) bit in the DMEM_CONTROL register for data descriptors, or
- The Enable ICPLB (ENICPLB) bit in the IMEM_CONTROL register for instruction descriptors

The CPLB replacement policy and algorithm to be used are the responsibility of the system MMU exception handler. This policy, which is dictated by the characteristics of the operating system, usually implements a modified LRU (Least Recently Used) policy, a round robin scheduling method, or pseudo random replacement.

After the new CPLB descriptor is loaded, the exception handler returns, and the faulting memory operation is restarted. This operation should now find a valid CPLB descriptor for the requested address, and it should proceed normally.
Memory Protection and Properties

A single instruction may generate an instruction fetch as well as one or two data accesses. It is possible that more than one of these memory operations references data for which there is no valid CPLB descriptor in an MMR pair. In this case, the exceptions are prioritized and serviced in this order:

- Instruction page miss
- A page miss on DAG0
- A page miss on DAG1

MMU Application

Memory management is an optional feature in the Blackfin processor architecture. Its use is predicated on the system requirements of a given application. Upon reset, all CPLBs are disabled, and the Memory Management Unit (MMU) is not used.

The MMU does not support automatic address translation in hardware.

If all L1 memory is configured as SRAM, then the data and instruction MMU functions are optional, depending on the application’s need for protection of memory spaces either between tasks or between User and Supervisor modes. To protect memory between tasks, the operating system can maintain separate tables of instruction and/or data memory pages available for each task and make those pages visible only when the relevant task is running. When a task switch occurs, the operating system can ensure the invalidation of any CPLB descriptors on chip that should not be available to the new task. It can also preload descriptors appropriate to the new task.

For many operating systems, the application program is run in User mode while the operating system and its services run in Supervisor mode. It is desirable to protect code and data structures used by the operating system...
from inadvertent modification by a running User mode application. This protection can be achieved by defining CPLB descriptors for protected memory ranges that allow write access only when in Supervisor mode. If a write to a protected memory region is attempted while in User mode, an exception is generated before the memory is modified. Optionally, the User mode application may be granted read access for data structures that are useful to the application. Even Supervisor mode functions can be blocked from writing some memory pages that contain code that is not expected to be modified. Because CPLB entries are MMRs that can be written only while in Supervisor mode, user programs cannot gain access to resources protected in this way.

If either the L1 Instruction Memory or the L1 Data Memory is configured partially or entirely as cache, the corresponding CPLBs must be enabled. When an instruction generates a memory request and the cache is enabled, the processor first checks the ICPLBs to determine whether the address requested is in a cacheable address range. If no valid ICPLB entry in an MMR pair corresponds to the requested address, an MMU exception is generated to obtain a valid ICPLB descriptor to determine whether the memory is cacheable or not. As a result, if the L1 Instruction Memory is enabled as cache, then any memory region that contains instructions must have a valid ICPLB descriptor defined for it. These descriptors must either reside in MMRs at all times or be resident in a memory-based Page Descriptor Table that is managed by the MMU exception handler. Likewise, if either or both L1 data banks are configured as cache, all potential data memory ranges must be supported by DCPLB descriptors.

Before caches are enabled, the MMU and its supporting data structures must be set up and enabled.
Examples of Protected Memory Regions

In Figure 6-18, a starting point is provided for basic CPLB allocation for Instruction and Data CPLBs. Note some ICPLBs and DCPLBs have common descriptors for the same address space.

![Diagram showing examples of protected memory regions](image)

Figure 6-18. Examples of Protected Memory Regions
ICPLB_DATAx Registers

Figure 6-19 describes the ICPLB Data registers (ICPLB_DATAx).

To ensure proper behavior and future compatibility, all reserved bits in this register must be set to 0 whenever this register is written.

ICPLB Data Registers (ICPLB_DATAx)

For Memory-mapped addresses, see Table 6-2.

Figure 6-19. ICPLB Data Registers
Table 6-2. ICPLB Data Register Memory-mapped Addresses

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICPLB_DATA0</td>
<td>0xFFE0 1200</td>
</tr>
<tr>
<td>ICPLB_DATA1</td>
<td>0xFFE0 1204</td>
</tr>
<tr>
<td>ICPLB_DATA2</td>
<td>0xFFE0 1208</td>
</tr>
<tr>
<td>ICPLB_DATA3</td>
<td>0xFFE0 120C</td>
</tr>
<tr>
<td>ICPLB_DATA4</td>
<td>0xFFE0 1210</td>
</tr>
<tr>
<td>ICPLB_DATA5</td>
<td>0xFFE0 1214</td>
</tr>
<tr>
<td>ICPLB_DATA6</td>
<td>0xFFE0 1218</td>
</tr>
<tr>
<td>ICPLB_DATA7</td>
<td>0xFFE0 121C</td>
</tr>
<tr>
<td>ICPLB_DATA8</td>
<td>0xFFE0 1220</td>
</tr>
<tr>
<td>ICPLB_DATA9</td>
<td>0xFFE0 1224</td>
</tr>
<tr>
<td>ICPLB_DATA10</td>
<td>0xFFE0 1228</td>
</tr>
<tr>
<td>ICPLB_DATA11</td>
<td>0xFFE0 122C</td>
</tr>
<tr>
<td>ICPLB_DATA12</td>
<td>0xFFE0 1230</td>
</tr>
<tr>
<td>ICPLB_DATA13</td>
<td>0xFFE0 1234</td>
</tr>
<tr>
<td>ICPLB_DATA14</td>
<td>0xFFE0 1238</td>
</tr>
<tr>
<td>ICPLB_DATA15</td>
<td>0xFFE0 123C</td>
</tr>
</tbody>
</table>
Memory

DCPLB_DATAx Registers

Figure 6-20 shows the DCPLB Data registers (DCPLB_DATAx).

To ensure proper behavior and future compatibility, all reserved bits in this register must be set to 0 whenever this register is written.

**DCPLB Data Registers (DCPLB_DATAx)**

For Memory-mapped addresses, see Table 6-3.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>PAGE_SIZE[1:0]</td>
</tr>
<tr>
<td>30</td>
<td>00 - 1K byte page size</td>
</tr>
<tr>
<td>29</td>
<td>01 - 4K byte page size</td>
</tr>
<tr>
<td>28</td>
<td>10 - 1M byte page size</td>
</tr>
<tr>
<td>27</td>
<td>11 - 4M byte page size</td>
</tr>
<tr>
<td>26</td>
<td>00 - Invalid (disabled) CPLB entry</td>
</tr>
<tr>
<td>25</td>
<td>1 - Valid (enabled) CPLB entry</td>
</tr>
<tr>
<td>24</td>
<td>CPLB_VALID</td>
</tr>
<tr>
<td>23</td>
<td>0 - User mode read access generates protection violation exception</td>
</tr>
<tr>
<td>22</td>
<td>1 - User mode read access permitted</td>
</tr>
<tr>
<td>21</td>
<td>CPLB_USER_RD</td>
</tr>
<tr>
<td>20</td>
<td>0 - User mode write access generates protection violation exception</td>
</tr>
<tr>
<td>19</td>
<td>1 - User mode write access permitted</td>
</tr>
<tr>
<td>18</td>
<td>CPLB_USER_WR</td>
</tr>
<tr>
<td>17</td>
<td>0 - Supervisor mode write access generates protection violation exception</td>
</tr>
<tr>
<td>16</td>
<td>1 - Supervisor mode write access permitted</td>
</tr>
<tr>
<td>15</td>
<td>CPLB_VALID</td>
</tr>
<tr>
<td>14</td>
<td>0 - Invalid (disabled) CPLB entry</td>
</tr>
<tr>
<td>13</td>
<td>1 - Valid (enabled) CPLB entry</td>
</tr>
<tr>
<td>12</td>
<td>CPLB_LOCK</td>
</tr>
<tr>
<td>11</td>
<td>Can be used by software in CPLB replacement algorithms</td>
</tr>
<tr>
<td>10</td>
<td>0 - Unlocked, CPLB entry can be replaced</td>
</tr>
<tr>
<td>9</td>
<td>1 - Locked, CPLB entry should not be replaced</td>
</tr>
<tr>
<td>8</td>
<td>CPLB_USER_RD</td>
</tr>
<tr>
<td>7</td>
<td>0 - User mode read access generates protection violation exception</td>
</tr>
<tr>
<td>6</td>
<td>1 - User mode read access permitted</td>
</tr>
<tr>
<td>5</td>
<td>CPLB_USER_WR</td>
</tr>
<tr>
<td>4</td>
<td>0 - Supervisor mode write access generates protection violation exception</td>
</tr>
<tr>
<td>3</td>
<td>1 - Supervisor mode write access permitted</td>
</tr>
<tr>
<td>2</td>
<td>CPLB_DIRTY</td>
</tr>
<tr>
<td>1</td>
<td>Valid only if write back cachable (CPLB_VALID = 1, CPLB_WT = 0, and CPLB_L1_CHBL = 1)</td>
</tr>
<tr>
<td>0</td>
<td>0 - Clean</td>
</tr>
<tr>
<td></td>
<td>1 - Dirty</td>
</tr>
</tbody>
</table>

A protection violation exception is generated on store accesses to this page when this bit is 0. The state of this bit is modified only by writes to this register. The exception service routine must set this bit.

Figure 6-20. DCPLB Data Registers
## Memory Protection and Properties

Table 6-3. DCPLB Data Register Memory-mapped Addresses

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCPLB_DATA0</td>
<td>0xFFE0 0200</td>
</tr>
<tr>
<td>DCPLB_DATA1</td>
<td>0xFFE0 0204</td>
</tr>
<tr>
<td>DCPLB_DATA2</td>
<td>0xFFE0 0208</td>
</tr>
<tr>
<td>DCPLB_DATA3</td>
<td>0xFFE0 020C</td>
</tr>
<tr>
<td>DCPLB_DATA4</td>
<td>0xFFE0 0210</td>
</tr>
<tr>
<td>DCPLB_DATA5</td>
<td>0xFFE0 0214</td>
</tr>
<tr>
<td>DCPLB_DATA6</td>
<td>0xFFE0 0218</td>
</tr>
<tr>
<td>DCPLB_DATA7</td>
<td>0xFFE0 021C</td>
</tr>
<tr>
<td>DCPLB_DATA8</td>
<td>0xFFE0 0220</td>
</tr>
<tr>
<td>DCPLB_DATA9</td>
<td>0xFFE0 0224</td>
</tr>
<tr>
<td>DCPLB_DATA10</td>
<td>0xFFE0 0228</td>
</tr>
<tr>
<td>DCPLB_DATA11</td>
<td>0xFFE0 022C</td>
</tr>
<tr>
<td>DCPLB_DATA12</td>
<td>0xFFE0 0230</td>
</tr>
<tr>
<td>DCPLB_DATA13</td>
<td>0xFFE0 0234</td>
</tr>
<tr>
<td>DCPLB_DATA14</td>
<td>0xFFE0 0238</td>
</tr>
<tr>
<td>DCPLB_DATA15</td>
<td>0xFFE0 023C</td>
</tr>
</tbody>
</table>
**DCPLB_ADDRx Registers**

Figure 6-21 shows the DCPLB Address registers (DCPLB_ADDRx).

**DCPLB Address Registers (DCPLB_ADDRx)**

![Diagram showing DCPLB Address Registers](image)

**Table 6-4. DCPLB Address Register Memory-mapped Addresses**

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCPLB_ADDR0</td>
<td>0xFFE0 0100</td>
</tr>
<tr>
<td>DCPLB_ADDR1</td>
<td>0xFFE0 0104</td>
</tr>
<tr>
<td>DCPLB_ADDR2</td>
<td>0xFFE0 0108</td>
</tr>
<tr>
<td>DCPLB_ADDR3</td>
<td>0xFFE0 010C</td>
</tr>
<tr>
<td>DCPLB_ADDR4</td>
<td>0xFFE0 0110</td>
</tr>
<tr>
<td>DCPLB_ADDR5</td>
<td>0xFFE0 0114</td>
</tr>
<tr>
<td>DCPLB_ADDR6</td>
<td>0xFFE0 0118</td>
</tr>
<tr>
<td>DCPLB_ADDR7</td>
<td>0xFFE0 011C</td>
</tr>
<tr>
<td>DCPLB_ADDR8</td>
<td>0xFFE0 0120</td>
</tr>
<tr>
<td>DCPLB_ADDR9</td>
<td>0xFFE0 0124</td>
</tr>
</tbody>
</table>
Memory Protection and Properties

Table 6-4. DCPLB Address Register Memory-mapped Addresses (Cont’d)

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCPLB_ADDR10</td>
<td>0xFFE0 0128</td>
</tr>
<tr>
<td>DCPLB_ADDR11</td>
<td>0xFFE0 012C</td>
</tr>
<tr>
<td>DCPLB_ADDR12</td>
<td>0xFFE0 0130</td>
</tr>
<tr>
<td>DCPLB_ADDR13</td>
<td>0xFFE0 0134</td>
</tr>
<tr>
<td>DCPLB_ADDR14</td>
<td>0xFFE0 0138</td>
</tr>
<tr>
<td>DCPLB_ADDR15</td>
<td>0xFFE0 013C</td>
</tr>
</tbody>
</table>

ICPLB_ADDRx Registers

Figure 6-22 shows the ICPLB Address registers (ICPLB_ADDRx).

ICPLB Address Registers (ICPLB_ADDRx)

For Memory-mapped addresses, see Table 6-5.

Figure 6-22. ICPLB Address Registers
Bits in the DCPLB Status register (DCPLB_STATUS) and ICPLB Status register (ICPLB_STATUS) identify the CPLB entry that has triggered CPLB-related exceptions. The exception service routine can infer the cause of the fault by examining the CPLB entries.

The DCPLB_STATUS and ICPLB_STATUS registers are valid only while in the faulting exception service routine.

### DCPLB_STATUS and ICPLB_STATUS Registers

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICPLB_ADDR0</td>
<td>0xFFE0 1100</td>
</tr>
<tr>
<td>ICPLB_ADDR1</td>
<td>0xFFE0 1104</td>
</tr>
<tr>
<td>ICPLB_ADDR2</td>
<td>0xFFE0 1108</td>
</tr>
<tr>
<td>ICPLB_ADDR3</td>
<td>0xFFE0 110C</td>
</tr>
<tr>
<td>ICPLB_ADDR4</td>
<td>0xFFE0 1110</td>
</tr>
<tr>
<td>ICPLB_ADDR5</td>
<td>0xFFE0 1114</td>
</tr>
<tr>
<td>ICPLB_ADDR6</td>
<td>0xFFE0 1118</td>
</tr>
<tr>
<td>ICPLB_ADDR7</td>
<td>0xFFE0 111C</td>
</tr>
<tr>
<td>ICPLB_ADDR8</td>
<td>0xFFE0 1120</td>
</tr>
<tr>
<td>ICPLB_ADDR9</td>
<td>0xFFE0 1124</td>
</tr>
<tr>
<td>ICPLB_ADDR10</td>
<td>0xFFE0 1128</td>
</tr>
<tr>
<td>ICPLB_ADDR11</td>
<td>0xFFE0 112C</td>
</tr>
<tr>
<td>ICPLB_ADDR12</td>
<td>0xFFE0 1130</td>
</tr>
<tr>
<td>ICPLB_ADDR13</td>
<td>0xFFE0 1134</td>
</tr>
<tr>
<td>ICPLB_ADDR14</td>
<td>0xFFE0 1138</td>
</tr>
<tr>
<td>ICPLB_ADDR15</td>
<td>0xFFE0 113C</td>
</tr>
</tbody>
</table>

Table 6-5. ICPLB Address Register Memory-mapped Addresses
Memory Protection and Properties

Bits `FAULT_DAG`, `FAULT_USERSUPV` and `FAULT_RW` in the DCPLB Status register (`DCPLB_STATUS`) are used to identify the CPLB entry that has triggered the CPLB-related exception (see Figure 6-23).

**DCPLB Status Register (DCPLB_STATUS)**

```plaintext
0xFFE0 0008

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<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>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>XXX</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**FAULT_ILLADDR**
0 - No fault
1 - Attempted access to nonexistent memory

**FAULT_DAG**
0 - Access was made by DAG0
1 - Access was made by DAG1

**FAULT_RW**
0 - Access was read
1 - Access was write

**FAULT_USERSUPV**
0 - Access was made in User mode
1 - Access was made in Supervisor mode

Each bit indicates the hit/miss status of the associated CPLB entry

Figure 6-23. DCPLB Status Register

Bit `FAULT_USERSUPV` in the ICPLB Status register (`ICPLB_STATUS`) is used to identify the CPLB entry that has triggered the CPLB-related exception (see Figure 6-24).
The DCPLB Address register (`DCPLB_FAULT_ADDR`) and ICPLB Fault Address register (`ICPLB_FAULT_ADDR`) hold the address that has caused a fault in the L1 Data Memory or L1 Instruction Memory, respectively. See Figure 6-25 and Figure 6-26.

The `DCPLB_FAULT_ADDR` and `ICPLB_FAULT_ADDR` registers are valid only while in the faulting exception service routine.
Memory Protection and Properties

DCPLB Address Register (DCPLB_FAULT_ADDR)

```
0xFFE0 000C
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
```

Reset = Undefined

FAULT_ADDR[31:16]
Data address that has caused a fault in L1 Data Memory

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
```

FAULT_ADDR[15:0]
Data address that has caused a fault in the L1 Data Memory

Figure 6-25. DCPLB Address Register

ICPLB Fault Address Register (ICPLB_FAULT_ADDR)

```
0xFFE0 100C
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
```

Reset = Undefined

FAULT_ADDR[31:16]
Instruction address that has caused a fault in the L1 Instruction Memory

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
```

FAULT_ADDR[15:0]
Instruction address that has caused a fault in the L1 Instruction Memory

Figure 6-26. ICPLB Fault Address Register
Memory Transaction Model

Both internal and external memory locations are accessed in little endian byte order. Figure 6-27 shows a data word stored in register R0 and in memory at address location addr. B0 refers to the least significant byte of the 32-bit word.

Figure 6-27. Data Stored in Little Endian Order

Figure 6-28 shows 16- and 32-bit instructions stored in memory. The diagram on the left shows 16-bit instructions stored in memory with the most significant byte of the instruction stored in the high address (byte B1 in addr+1) and the least significant byte in the low address (byte B0 in addr).

Figure 6-28. Instructions Stored in Little Endian Order

The diagram on the right shows 32-bit instructions stored in memory. Note the most significant 16-bit half word of the instruction (bytes B3 and B2) is stored in the low addresses (addr+1 and addr), and the least significant half word (bytes B1 and B0) is stored in the high addresses (addr+3 and addr+2).
Load/Store Operation

The Blackfin processor architecture supports the RISC concept of a Load/Store machine. This machine is the characteristic in RISC architectures whereby memory operations (loads and stores) are intentionally separated from the arithmetic functions that use the targets of the memory operations. The separation is made because memory operations, particularly instructions that access off-chip memory or I/O devices, often take multiple cycles to complete and would normally halt the processor, preventing an instruction execution rate of one instruction per cycle.

In write operations, the store instruction is considered complete as soon as it executes, even though many cycles may execute before the data is actually written to an external memory or I/O location. This arrangement allows the processor to execute one instruction per clock cycle, and it implies that the synchronization between when writes complete and when subsequent instructions execute is not guaranteed. Moreover, this synchronization is considered unimportant in the context of most memory operations.

Interlocked Pipeline

In the execution of instructions, the Blackfin processor architecture implements an interlocked pipeline. When a load instruction executes, the target register of the read operation is marked as busy until the value is returned from the memory system. If a subsequent instruction tries to access this register before the new value is present, the pipeline will stall until the memory operation completes. This stall guarantees that instructions that require the use of data resulting from the load do not use the previous or invalid data in the register, even though instructions are allowed to start execution before the memory read completes.

This mechanism allows the execution of independent instructions between the load and the instructions that use the read target without requiring the programmer or compiler to know how many cycles are actually needed for
the memory-read operation to complete. If the instruction immediately following the load uses the same register, it simply stalls until the value is returned. Consequently, it operates as the programmer expects. However, if four other instructions are placed after the load but before the instruction that uses the same register, all of them execute, and the overall throughput of the processor is improved.

**Ordering of Loads and Stores**

The relaxation of synchronization between memory access instructions and their surrounding instructions is referred to as weak ordering of loads and stores. Weak ordering implies that the timing of the actual completion of the memory operations—even the order in which these events occur—may not align with how they appear in the sequence of the program source code. All that is guaranteed is:

- Load operations will complete before the returned data is used by a subsequent instruction.
- Load operations using data previously written will use the updated values.
- Store operations will eventually propagate to their ultimate destination.

Because of weak ordering, the memory system is allowed to prioritize reads over writes. In this case, a write that is queued anywhere in the pipeline, but not completed, may be deferred by a subsequent read operation, and the read is allowed to be completed before the write. Reads are prioritized over writes because the read operation has a dependent operation waiting on its completion, whereas the processor considers the write operation complete, and the write does not stall the pipeline if it takes more cycles to propagate the value out to memory. This behavior could cause a read that occurs in the program source code after a write in the program flow to actually return its value before the write has been completed.
Load/Store Operation

This ordering provides significant performance advantages in the operation of most memory instructions. However, it can cause side effects that the programmer must be aware of to avoid improper system operation.

When writing to or reading from nonmemory locations such as off-chip I/O device registers, the order of how read and write operations complete is often significant. For example, a read of a status register may depend on a write to a control register. If the address is the same, the read would return a value from the store buffer rather than from the actual I/O device register, and the order of the read and write at the register may be reversed. Both these effects could cause undesirable side effects in the intended operation of the program and peripheral. To ensure that these effects do not occur in code that requires precise (strong) ordering of load and store operations, synchronization instructions (CSYNC or SSYNC) should be used.

Synchronizing Instructions

When strong ordering of loads and stores is required, as may be the case for sequential writes to an I/O device for setup and control, use the core or system synchronization instructions, CSYNC or SSYNC, respectively.

The CSYNC instruction ensures all pending core operations have completed and the store buffer (between the processor core and the L1 memories) has been flushed before proceeding to the next instruction. Pending core operations may include any pending interrupts, speculative states (such as branch predictions), or exceptions.

Consider the following example code sequence:

```assembly
IF CC JUMP away_from_here:
   CSYNC:
   R0 = [P0]:
away_from_here:
```
In the preceding example code, the CSYNC instruction ensures:

- The conditional branch (IF CC JUMP away_from_here) is resolved, forcing stalls into the execution pipeline until the condition is resolved and any entries in the processor store buffer have been flushed.
- All pending interrupts or exceptions have been processed before CSYNC completes.
- The load is not fetched from memory speculatively.

The SSYNC instruction ensures that all side effects of previous operations are propagated out through the interface between the L1 memories and the rest of the chip. In addition to performing the core synchronization functions of CSYNC, the SSYNC instruction flushes any write buffers between the L1 memory and the system domain and generates a sync request to the system that requires acknowledgement before SSYNC completes.

**Speculative Load Execution**

Load operations from memory do not change the state of the memory value. Consequently, issuing a speculative memory-read operation for a subsequent load instruction usually has no undesirable side effect. In some code sequences, such as a conditional branch instruction followed by a load, performance may be improved by speculatively issuing the read request to the memory system before the conditional branch is resolved. For example,

```assembly
IF CC JUMP away_from_here
RO = [P2];
...
away_from_here:
```
Load/Store Operation

If the branch is taken, then the load is flushed from the pipeline, and any results that are in the process of being returned can be ignored. Conversely, if the branch is not taken, the memory will have returned the correct value earlier than if the operation were stalled until the branch condition was resolved.

However, in the case of an off-chip I/O device, this could cause an undesirable side effect for a peripheral that returns sequential data from a FIFO or from a register that changes value based on the number of reads that are requested. To avoid this effect, use synchronizing instructions (CSYNC or SSYNC) to guarantee the correct behavior between read operations.

Store operations never access memory speculatively, because this could cause modification of a memory value before it is determined whether the instruction should have executed.

On-chip peripherals are guarded against destruction due to speculative reads. There, a separate strobe triggers the read side-effect when the instruction actually executes.

Conditional Load Behavior

The synchronization instructions force all speculative states to be resolved before a load instruction initiates a memory reference. However, the load instruction itself may generate more than one memory-read operation, because it is interruptible. If an interrupt of sufficient priority occurs between the completion of the synchronization instruction and the completion of the load instruction, the sequencer cancels the load instruction. After execution of the interrupt, the interrupted load is executed again. This approach minimizes interrupt latency. However, it is possible that a memory-read cycle was initiated before the load was canceled, and this would be followed by a second read operation after the load is executed again. For most memory accesses, multiple reads of the same memory address have no side effects. However, for some off-chip memory-mapped
devices, such as peripheral data FIFOs, reads are destructive. Each time the device is read, the FIFO advances, and the data cannot be recovered and re-read.

When accessing off-chip memory-mapped devices that have state dependencies on the number of read operations on a given address location, disable interrupts before performing the load operation.

On-chip peripherals are protected against this issue.

Working With Memory

This section contains information about alignment of data in memory and memory operations that support semaphores between tasks. It also contains a brief discussion of MMR registers and a core MMR programming example.

Alignment

Nonaligned memory operations are not directly supported. A nonaligned memory reference generates a Maligned Access exception event (see “Exceptions” on page 4-47). However, because some datastreams (such as 8-bit video data) can properly be nonaligned in memory, alignment exceptions may be disabled by using the DISALGNEXCPT instruction. Moreover, some instructions in the quad 8-bit group automatically disable alignment exceptions.

Cache Coherency

For shared data, software must provide cache coherency support as required. To accomplish this, use the FLUSH instruction (see “Data Cache Control Instructions” on page 6-37), and/or explicit line invalidation through the core MMRs (see “Data Test Registers” on page 6-38).
Atomic Operations

The processor provides a single atomic operation: TESTSET. Atomic operations are used to provide noninterruptible memory operations in support of semaphores between tasks. The TESTSET instruction loads an indirectly addressed memory half word, tests whether the low byte is zero, and then sets the most significant bit (MSB) of the low memory byte without affecting any other bits. If the byte is originally zero, the instruction sets the CC bit. If the byte is originally nonzero, the instruction clears the CC bit. The sequence of this memory transaction is atomic—hardware bus locking insures that no other memory operation can occur between the test and set portions of this instruction. The TESTSET instruction can be interrupted by the core. If this happens, the TESTSET instruction is executed again upon return from the interrupt.

The TESTSET instruction can address the entire 4G byte memory space, but should not target on-core memory (L1 or MMR space) since atomic access to this memory is not supported.

The memory architecture always treats atomic operations as cache inhibited accesses even if the CPLB descriptor for the address indicates cache enabled access. However, executing TESTSET operations on cacheable regions of memory is not recommended since the architecture cannot guarantee a cacheable location of memory is coherent when the TESTSET instruction is executed.

Memory-mapped Registers

The MMR reserved space is located at the top of the memory space (0xFFC0 0000). This region is defined as non-cacheable and is divided between the system MMRs (0xFFC0 0000–0xFFE0 0000) and core MMRs (0xFFE0 0000–0xFFFF FFFF).

Like non-memory mapped registers, the core MMRs connect to the 32-bit wide Register Access Bus (RAB). They operate at CCLK frequency.
Memory

System MMRs connect to the Peripheral Access Bus (PAB), which is implemented as either a 16-bit or a 32-bit wide bus on specific derivatives. The PAB bus operates at SCLK rate. Writes to system MMRs do not go through write buffers nor through store buffers. Rather, there is a simple bridge between the RAB and the PAB bus that translates between clock domains (and bus width) only.

On ADSP-BF535 products only, the system MMRs do reside behind store and write buffers. There, system MMRs behave like off-chip I/O devices as described in “Load/Store Operation” on page 6-66. Consequently, SSYNC instructions are required after store instructions to guarantee strong ordering of MMR accesses.

All MMRs are accessible only in Supervisor mode. Access to MMRs in User mode generates a protection violation exception.

All core MMRs are read and written using 32-bit aligned accesses. However, some MMRs have fewer than 32 bits defined. In this case, the unused bits are reserved. System MMRs may be 16 bits.

Accesses to nonexistent MMRs generate an illegal access exception. The system ignores writes to read-only MMRs.

Hardware raises an exception when a multi-issue instruction attempts to simultaneously perform two accesses to MMR space.

Appendix B provides a summary of all Core MMRs.

Core MMR Programming Code Example

Core MMRs may be accessed only as aligned 32-bit words. Nonaligned access to MMRs generates an exception event. Listing 6-1 shows the instructions required to manipulate a generic core MMR.
Listing 6-1. Core MMR Programming

CLI RO; /* stop interrupts and save IMASK */
P0 = MMR_BASE; /* 32-bit instruction to load base of MMRs */
R1 = [P0 + TIMER_CONTROL_REG]; /* get value of control reg */
BITSET R1, #N; /* set bit N */
[P0 + TIMER_CONTROL_REG] = R1; /* restore control reg */
CSYNC; /* assures that the control reg is written */
STI RO; /* enable interrupts */

The CLI instruction saves the contents of the IMASK register and disables interrupts by clearing IMASK. The STI instruction restores the contents of the IMASK register, thus enabling interrupts. The instructions between CLI and STI are not interruptible.

Terminology

The following terminology is used to describe memory.

**cache block.** The smallest unit of memory that is transferred to/from the next level of memory from/to a cache as a result of a cache miss.

**cache hit.** A memory access that is satisfied by a valid, present entry in the cache.

**cache line.** Same as cache block. In this chapter, cache line is used for cache block.

**cache miss.** A memory access that does not match any valid entry in the cache.

**direct-mapped.** Cache architecture in which each line has only one place in which it can appear in the cache. Also described as 1-Way associative.
dirty or modified. A state bit, stored along with the tag, indicating whether the data in the data cache line has been changed since it was copied from the source memory and, therefore, needs to be updated in that source memory.

exclusive, clean. The state of a data cache line, indicating that the line is valid and that the data contained in the line matches that in the source memory. The data in a clean cache line does not need to be written to source memory before it is replaced.

fully associative. Cache architecture in which each line can be placed anywhere in the cache.

index. Address portion that is used to select an array element (for example, a line index).

invalid. Describes the state of a cache line. When a cache line is invalid, a cache line match cannot occur.

least recently used (LRU) algorithm. Replacement algorithm, used by cache, that first replaces lines that have been unused for the longest time.

Level 1 (L1) memory. Memory that is directly accessed by the core with no intervening memory subsystems between it and the core.

little endian. The native data store format of the Blackfin processor. Words and half words are stored in memory (and registers) with the least significant byte at the lowest byte address and the most significant byte in the highest byte address of the data storage location.

replacement policy. The function used by the processor to determine which line to replace on a cache miss. Often, an LRU algorithm is employed.

set. A group of N-line storage locations in the Ways of an N-Way cache, selected by the INDEX field of the address (see Figure 6-4 on page 6-12).
Terminology

**set associative.** Cache architecture that limits line placement to a number of sets (or Ways).

**tag.** Upper address bits, stored along with the cached data line, to identify the specific address source in memory that the cached line represents.

**valid.** A state bit, stored with the tag, indicating that the corresponding tag and data are current and correct and can be used to satisfy memory access requests.

**victim.** A dirty cache line that must be written to memory before it can be replaced to free space for a cache line allocation.

**Way.** An array of line storage elements in an N-Way cache (see Figure 6-4 on page 6-12).

**write back.** A cache write policy, also known as *copyback*. The write data is written only to the cache line. The modified cache line is written to source memory only when it is replaced. Cache lines are allocated on both reads and writes.

**write through.** A cache write policy (also known as store through). The write data is written to both the cache line and to the source memory. The modified cache line is not written to the source memory when it is replaced. Cache lines must be allocated on reads, and may be allocated on writes (depending on mode).
7 PROGRAM FLOW CONTROL

Instruction Summary

- “Jump” on page 7-2
- “IF CC JUMP” on page 7-5
- “Call” on page 7-8
- “RTS, RTI, RTX, RTN, RTE (Return)” on page 7-10
- “LSETUP, LOOP” on page 7-13

Instruction Overview

This chapter discusses the instructions that control program flow. Users can take advantage of these instructions to force new values into the Program Counter and change program flow, branch conditionally, set up loops, and call and return from subroutines.
Instruction Overview

Jump

General Form

\[
\text{JUMP (destination_indirect)}
\]
\[
\text{JUMP (PC + offset)}
\]
\[
\text{JUMP offset}
\]
\[
\text{JUMP.S offset}
\]
\[
\text{JUMP.L offset}
\]

Syntax

\[
\text{JUMP (Preg)}; \quad /* \text{indirect to an absolute (not PC-relative) address (a)} */
\]
\[
\text{JUMP (PC + Preg)}; \quad /* \text{PC-relative, indexed (a)} */
\]
\[
\text{JUMP pcrel25m2}; \quad /* \text{PC-relative, immediate (a) or (b)} */
\]
\[
\text{see "Functional Description" on page 7-3}
\]
\[
\text{JUMP.S pcrel13m2}; \quad /* \text{PC-relative, immediate, short (a)} */
\]
\[
\text{JUMP.L pcrel25m2}; \quad /* \text{PC-relative, immediate, long (b)} */
\]
\[
\text{JUMP user_label}; \quad /* \text{user-defined absolute address label, resolved by the assembler/linker to the appropriate PC-relative instruction (a) or (b)} */
\]

Syntax Terminology

\textit{Preg}: P5–0, SP, FP

\textit{pcrelm2}: undetermined 25-bit or smaller signed, even relative offset, with a range of \(-16,777,216\) through \(16,777,214\) bytes (\(0xFF00\ 0000\) to \(0x00FF\ FFFE\))

\textit{pcrel13m2}: 13-bit signed, even relative offset, with a range of \(-4096\) through \(4094\) bytes (\(0xF000\) to \(0x0FFE\))

1 This instruction can be used in assembly-level programs when the final distance to the target is unknown at coding time. The assembler substitutes the opcode for JUMP.S or JUMP.L depending on the final target. Disassembled code shows the mnemonic JUMP.S or JUMP.L.
Program Flow Control

*pcrel25m2*: 25-bit signed, even relative offset, with a range of –16,777,216 through 16,777,214 bytes (0xFF00 0000 to 0x00FF FFFE)

*user_label*: valid assembler address label, resolved by the assembler/linker to a valid PC-relative offset

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

Functional Description

The Jump instruction forces a new value into the Program Counter (PC) to change program flow.

In the Indirect and Indexed versions of the instruction, the value in Preg must be an even number (bit0=0) to maintain 16-bit address alignment. Otherwise, an odd offset in Preg causes the processor to invoke an alignment exception.

Flags Affected

None

Required Mode

User & Supervisor

Parallel Issue

The Jump instruction cannot be issued in parallel with other instructions.
Instruction Overview

Example

```asm
jump get_new_sample;  /* assembler resolved target, abstract offsets */
jump (p5);  /* P5 contains the absolute address of the target */
jump (pc + p2);  /* P2 relative absolute address of the target and then a presentation of the absolute values for target */
jump 0x224;  /* offset is positive in 13 bits, so target address is PC + 0x224, a forward jump */
jump.s 0x224;  /* same as above with jump “short” syntax */
jump.l 0xFFFACE86;  /* offset is negative in 25 bits, so target address is PC + 0x1FA CE86, a backwards jump */
```

Also See

Call, IF CC JUMP

Special Applications

None
Program Flow Control

IF CC JUMP

General Form

IF CC JUMP destination
IF !CC JUMP destination

Syntax

IF CC JUMP pcrel11m2 : /* branch if CC=1, branch predicted as not taken (a) */¹
IF CC JUMP pcrel11m2 (bp) : /* branch if CC=1, branch predicted as taken (a) */¹
IF !CC JUMP pcrel11m2 : /* branch if CC=0, branch predicted as not taken (a) */²
IF !CC JUMP pcrel11m2 (bp) : /* branch if CC=0, branch predicted as taken (a) */²
IF CC JUMP user_label : /* user-defined absolute address label, resolved by the assembler/linker to the appropriate PC-relative instruction (a) */¹
IF CC JUMP user_label (bp) : /* user-defined absolute address label, resolved by the assembler/linker to the appropriate PC-relative instruction (a) */¹
IF !CC JUMP user_label : /* user-defined absolute address label, resolved by the assembler/linker to the appropriate PC-relative instruction (a) */¹
IF !CC JUMP user_label (bp) : /* user-defined absolute address label, resolved by the assembler/linker to the appropriate PC-relative instruction (a) */¹

¹ CC bit = 1 causes a branch to an address, computed by adding the signed, even offset to the current PC value.
² CC bit = 0 causes a branch to an address, computed by adding the signed, even relative offset to the current PC value.
Instruction Overview

Syntax Terminology

pcrel11m2: 11-bit signed even relative offset, with a range of –1024 through 1022 bytes (0xFC00 to 0x03FE). This value can optionally be replaced with an address label that is evaluated and replaced during linking.

user_label: valid assembler address label, resolved by the assembler/linker to a valid PC-relative offset

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Conditional JUMP instruction forces a new value into the Program Counter (PC) to change the program flow, based on the value of the CC bit.

The range of valid offset values is –1024 through 1022.

Option

The Branch Prediction appendix (bp) helps the processor improve branch instruction performance. The default is branch predicted-not-taken. By appending (bp) to the instruction, the branch becomes predicted-taken.

Typically, code analysis shows that a good default condition is to predict branch-taken for branches to a prior address (backwards branches), and to predict branch-not-taken for branches to subsequent addresses (forward branches).

Flags Affected

None
Program Flow Control

Required Mode

User & Supervisor

Parallel Issue

This instruction cannot be issued in parallel with other instructions.

Example

if cc jump 0xFFFFFE08 (bp) ; /* offset is negative in 11 bits,
so target address is a backwards branch, branch predicted */
if cc jump 0x0B4 ; /* offset is positive, so target offset
address is a forwards branch, branch not predicted */
if !cc jump 0xFFFFFC22 (bp) ; /* negative offset in 11 bits, so
target address is a backwards branch, branch predicted */
if !cc jump 0x120 ; /* positive offset, so target address is a
forwards branch, branch not predicted */
if cc jump dest_label ; /* assembler resolved target, abstract
offsets */

Also See

Jump, Call

Special Applications

None
**Instruction Overview**

**Call**

**General Form**

CALL (destination_indirect  
CALL (PC + offset)  
CALL offset

**Syntax**

CALL ( Preg ) ;  /* indirect to an absolute (not PC-relative)  
   address (a) */  
CALL ( PC + Preg ) ;  /* PC-relative, indexed (a) */  
CALL pcrel25m2 ;  /* PC-relative, immediate (b) */  
CALL user_label ;  /* user-defined absolute address label,  
   resolved by the assembler/linker to the appropriate PC-relative  
   instruction (a) or (b) */

**Syntax Terminology**

Preg: P5-0 (SP and FP are not allowed as the source register for this  
   instruction.)

pcrel25m2: 25-bit signed, even, PC-relative offset; can be specified as a  
   symbolic address label, with a range of –16,777,216 through 16,777,214  
   (0xFF00 0000 to 0x00FF FFFE) bytes.

user_label: valid assembler address label, resolved by the assembler/linker  
   to a valid PC-relative offset

**Instruction Length**

In the syntax, comment (a) identifies 16-bit instruction length. Comment  
(b) identifies 32-bit instruction length.
Program Flow Control

Functional Description

The CALL instruction calls a subroutine from an address that a P-register points to or by using a PC-relative offset. After the CALL instruction executes, the RETS register contains the address of the next instruction.

The value in the Preg must be an even value to maintain 16-bit alignment.

Flags Affected

None

Required Mode

User & Supervisor

Parallel Issue

This instruction cannot be issued in parallel with other instructions.

Example

```c
  call ( p5 ) ;
  call ( pc + p2 ) ;
  call 0x123456 ;
  call get_next_sample ;
```

Also See

RTS, RTI, RTX, RTN, RTE (Return), Jump, IF CC JUMP

Special Applications

None
Instruction Overview

RTS, RTI, RTX, RTN, RTE (Return)

General Form

RTS, RTI, RTX, RTN, RTE

Syntax

RTS : // Return from Subroutine (a)
RTI : // Return from Interrupt (a)
RTX : // Return from Exception (a)
RTN : // Return from NMI (a)
RTE : // Return from Emulation (a)

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Return instruction forces a return from a subroutine, maskable or NMI interrupt routine, exception routine, or emulation routine (see Table 7-1).

Flags Affected

None

Required Mode

Table 7-2 identifies the modes required by the Return instruction.

Parallel Issue

This instruction cannot be issued in parallel with other instructions.
Program Flow Control

Table 7-1. Types of Return Instruction

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTS</td>
<td>Forces a return from a subroutine by loading the value of the RETS Register into the Program Counter (PC), causing the processor to fetch the next instruction from the address contained in RETS. For nested subroutines, you must save the value of the RETS Register. Otherwise, the next subroutine CALL instruction overwrites it.</td>
</tr>
<tr>
<td>RTI</td>
<td>Forces a return from an interrupt routine by loading the value of the RETI Register into the PC. When an interrupt is generated, the processor enters a non-interruptible state. Saving RETI to the stack re-enables interrupt detection so that subsequent, higher priority interrupts can be serviced (or &quot;nested&quot;) during the current interrupt service routine. If RETI is not saved to the stack, higher priority interrupts are recognized but not serviced until the current interrupt service routine concludes. Restoring RETI back off the stack at the conclusion of the interrupt service routine masks subsequent interrupts until the RTI instruction executes. In any case, RETI is protected against inadvertent corruption by higher priority interrupts.</td>
</tr>
<tr>
<td>RTX</td>
<td>Forces a return from an exception routine by loading the value of the RETX Register into the PC.</td>
</tr>
<tr>
<td>RTN</td>
<td>Forces a return from a non-maskable interrupt (NMI) routine by loading the value of the RETN Register into the PC.</td>
</tr>
<tr>
<td>RTE</td>
<td>Forces a return from an emulation routine and emulation mode by loading the value of the RETE Register into the PC. Because only one emulation routine can run at a time, nesting is not an issue, and saving the value of the RETE Register is unnecessary.</td>
</tr>
</tbody>
</table>

Table 7-2. Required Mode for the Return Instruction

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Required Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTS</td>
<td>User &amp; Supervisor</td>
</tr>
<tr>
<td>RTI, RTX, and RTN</td>
<td>Supervisor only. Any attempt to execute in User mode produces a protection violation exception.</td>
</tr>
<tr>
<td>RTE</td>
<td>Emulation only. Any attempt to execute in User mode or Supervisor mode produces an exception.</td>
</tr>
</tbody>
</table>
Instruction Overview

Example

\texttt{rts;}
\texttt{rti;}
\texttt{rtx;}
\texttt{rtn;}
\texttt{rte;}

Also See

Call, \texttt{--SP (Push), SP++ (Pop)}

Special Applications

None
LSETUP, LOOP

General Form

There are two forms of this instruction. The first is:

LOOP loop_name loop_counter
LOOP_BEGIN loop_name
LOOP_END loop_name

The second form is:

LSETUP (Begin_Loop, End_Loop)Loop_Counter

Syntax

For Loop0

LOOP loop_name LC0 ; /* (b) */
LOOP loop_name LC0 = Preg ; /* autoinit LCO (b) */
LOOP loop_name LC0 = Preg >> 1 ; /* autoinit LCO(b) */
LOOP_BEGIN loop_name ; /* define the 1st instruction of loop(b) */
LOOP_END loop_name ; /* define the last instruction of the loop (b) */

/* use any one of the LOOP syntax versions with a LOOP_BEGIN and a LOOP_END instruction. The name of the loop ("loop_name" in the syntax) relates the three instructions together. */

LSETUP ( pcrel15m2 , lppcrel11m2 ) LC0 ; /* (b) */
LSETUP ( pcrel15m2 , lppcrel11m2 ) LC0 = Preg ; /* autoinitalize LCO (b) */
LSETUP ( pcrel15m2 , lppcrel11m2 ) LC0 = Preg >> 1 ; /* autoiniitalize LCO (b) */
Instruction Overview

For Loop1

```assembly
LOOP loop_name LC1 ; /* (b) */
LOOP loop_name LC1 = Preg ; /* autoinitialize LC1 (b) */
LOOP loop_name LC1 = Preg >> 1 ; /* autoinitialize LC1 (b) */
LOOP_BEGIN loop_name ; /* define the first instruction of the loop (b) */
LOOP_END loop_name ; /* define the last instruction of the loop (b) */
/* Use any one of the LOOP syntax versions with a LOOP_BEGIN and a LOOP_END instruction. The name of the loop (“loop_name” in the syntax) relates the three instructions together. */

LSETUP ( pcrel5m2 , lppcrel11m2 ) LC1 ; /* (b) */
LSETUP ( pcrel5m2 , lppcrel11m2 ) LC1 = Preg ; /* autoinitialize LC1 (b) */
LSETUP ( pcrel5m2 , lppcrel11m2 ) LC1 = Preg >> 1 ; /* autoinitialize LC1 (b) */
```

Syntax Terminology

- **Preg**: P5–0 (SP and FP are not allowed as the source register for this instruction.)

- **pcrel5m2**: 5-bit unsigned, even, PC-relative offset; can be replaced by a symbolic label. The range is 4 to 30, or \(2^5 - 2\).

- **lppcrel11m2**: 11-bit unsigned, even, PC-relative offset for a loop; can be replaced by a symbolic label. The range is 4 to 2046 (0x0004 to 0x07FE), or \(2^{11} - 2\).

- **loop_name**: a symbolic identifier

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.
Program Flow Control

Functional Description

The Zero-Overhead Loop Setup instruction provides a flexible, counter-based, hardware loop mechanism that provides efficient, zero-overhead software loops. In this context, zero-overhead means that the software in the loops does not incur a performance or code size penalty by decrementing a counter, evaluating a loop condition, then calculating and branching to a new target address.

When the Begin_Loop address is the next sequential address after the LSETUP instruction, the loop has zero overhead. If the Begin_Loop address is not the next sequential address after the LSETUP instruction, there is some overhead that is incurred on loop entry only.

The architecture includes two sets of three registers each to support two independent, nestable loops. The registers are Loop_Top (LTn), Loop_Bottom (LBn) and Loop_Count (LCn). Consequently, LT0, LB0, and LC0 describe Loop0, and LT1, LB1, and LC1 describe Loop1.

The LOOP and LSETUP instructions are a convenient way to initialize all three registers in a single instruction. The size of the LOOP and LSETUP instructions only supports a finite number of bits, so the loop range is limited. However, LT0 and LT1, LB0 and LB1, and LC0 and LC1 can be initialized manually using Move instructions if loop length and repetition count need to be beyond the limits supported by the LOOP and LSETUP syntax. Thus, a single loop can span the entire 4 GB of memory space.

When initializing LT0 and LT1, LB0 and LB1, and LC0 and LC1 manually, make sure that Loop_Top (LTn) and Loop_Bottom (LBn) are configured before setting Loop_Count (LCn) to the desired loop count value.

The instruction syntax supports an optional initialization value from a P-register or P-register divided by 2.
Instruction Overview

The LOOP, LOOP_BEGIN, LOOP_END syntax is generally more readable and user friendly. The LSETUP syntax contains the same information, but in a more compact form.

If LCn is nonzero when the fetch address equals LBn, the processor decrements LCn and places the address in LTn into the PC. The loop always executes once through because Loop_Count is evaluated at the end of the loop.

There are two special cases for small loop count values. A value of 0 in Loop_Count causes the hardware loop mechanism to neither decrement or loopback, causing the instructions enclosed by the loop pointers to be executed as straight-line code. A value of 1 in Loop_Count causes the hardware loop mechanism to decrement only (not loopback), also causing the instructions enclosed by the loop pointers to be executed as straight-line code.

In the instruction syntax, the designation of the loop counter—LC0 or LC1—determines which loop level is initialized. Consequently, to initialize Loop0, code LC0; to initialize Loop1, code LC1.

In the case of nested loops that end on the same instruction, the processor requires Loop0 to describe the outer loop and Loop1 to describe the inner loop. The user is responsible for meeting this requirement.

For example, if LB0=LB1, then the processor assumes loop 1 is the inner loop and loop 0 the outer loop.

Just like entries in any other register, loop register entries can be saved and restored. If nesting beyond two loop levels is required, the user can explicitly save the outermost loop register values, re-use the registers for an inner loop, and then restore the outermost loop values before terminating the inner loop. In such a case, remember that loop 0 must always be outside of loop 1. Alternately, the user can implement the outermost loop in software with the Conditional Jump structure.
Program Flow Control

Begin_Loop, the value loaded into LTn, is a 5-bit, PC-relative, even offset from the current instruction to the first instruction in the loop. The user is required to preserve half-word alignment by maintaining even values in this register. The offset is interpreted as a one’s-complement, unsigned number, eliminating backwards loops.

End_Loop, the value loaded into LBi, is an 11-bit, unsigned, even, PC-relative offset from the current instruction to the last instruction of the loop.

When using the LSETUP instruction, Begin_Loop and End_Loop are typically address labels. The linker replaces the labels with offset values.

A loop counter register (LC0 or LC1) counts the trips through the loop. The register contains a 32-bit unsigned value, supporting as many as 4,294,967,294 trips through the loop. The loop is disabled (subsequent executions of the loop code pass through without reiterating) when the loop counter equals 0.

ADSP-BF535 Execution Note

The following information about instructions that are permissible as the last instruction on a loop applies only to the ADSP-BF535 processor, not to all ADSP-BF53x/BF56x processors.

The last instruction of the loop must not be any of the following instructions.

- Jump
- Conditional Branch
- Call
- CSYNC
- SSYNC
- Return (RTS, RTN, etc.)
Instruction Overview

As long as the hardware loop is active (Loop_Count is nonzero), any of these forbidden instructions at the End_Loop address produces undefined execution, and no exception is generated. Forbidden End_Loop instructions that appear anywhere else in the defined loop execute normally. Branch instructions that are located anywhere else in the defined loop execute normally.

Also, the last instruction in the loop must not modify the registers that define the currently active loop (LCn, LTn, or LBn). User modifications to those registers while the hardware accesses them produces undefined execution. Software can legally modify the loop counter at any other location in the loop.

Flags Affected

None

Required Mode

User & Supervisor

Parallel Issue

This instruction cannot be issued in parallel with other instructions.

Example

lsetup ( 4, 4 ) lc0 ;
lsetup ( poll_bit, end_poll_bit ) lc0 ;
lsetup ( 4, 6 ) lc1 ;
lsetup ( FIR_filter, bottom_of_FIR_filter ) lc1 ;
lsetup ( 4, 8 ) lc0 = p1 ;
lsetup ( 4, 8 ) lc0 = p1>>1 ;
Program Flow Control

loop DoItSome LC0 ; /* define loop ‘DoItSome’ with Loop Counter 0 */
loop_begin DoItSome ; /* place before the first instruction in the loop */
loop_end DoItSome ; /* place after the last instruction in the loop */
loop MyLoop LC1 ; /* define loop ‘MyLoop’ with Loop Counter 1 */
loop_begin MyLoop ; /* place before the first instruction in the loop */
loop_end MyLoop ; /* place after the last instruction in the loop */

Also See

IF CC JUMP, Jump

Special Applications

None
8 LOAD / STORE

Instruction Summary

- “Load Immediate” on page 8-3
- “Load Pointer Register” on page 8-7
- “Load Data Register” on page 8-10
- “Load Half-Word – Zero-Extended” on page 8-15
- “Load Half-Word – Sign-Extended” on page 8-19
- “Load High Data Register Half” on page 8-23
- “Load Low Data Register Half” on page 8-27
- “Load Byte – Zero-Extended” on page 8-31
- “Load Byte – Sign-Extended” on page 8-34
- “Store Pointer Register” on page 8-37
- “Store Data Register” on page 8-40
- “Store High Data Register Half” on page 8-45
- “Store Low Data Register Half” on page 8-49
- “Store Byte” on page 8-54
Instruction Overview

This chapter discusses the load/store instructions. Users can take advantage of these instructions to load and store immediate values, pointer registers, data registers or data register halves, and half words (zero or sign extended).
Load Immediate

General Form

\[
\text{register} = \text{constant} \\
A1 = A0 = 0
\]

Syntax

Half-Word Load

\[
\text{reg}_\text{lo} = \text{uimm16} \\
\text{reg}_\text{hi} = \text{uimm16}
\]

Zero Extended

\[
\text{reg} = \text{uimm16} (Z) \\
A0 = 0 \\
A1 = 0 \\
A1 = A0 = 0
\]

Sign Extended

\[
\text{Dreg} = \text{imm7} (X) \\
\text{Preg} = \text{imm7} (X) \\
\text{reg} = \text{imm16} (X)
\]

Syntax Terminology

\[
\text{Dreg: R7-0} \\
\text{Preg: P5-0, SP, FP}
\]
Instruction Overview

*reg_lo*: R7–0.L, P5–0.L, SP.L, FP.L, I3–0.L, M3–0.L, B3–0.L, L3–0.L


*reg*: R7–0, P5–0, SP, FP, I3–0, M3–0, B3–0, L3–0

*imm7*: 7-bit signed field, with a range of –64 through 63

*imm16*: 16-bit signed field, with a range of –32,768 through 32,767 (0x800 through 0x7FFF)

*uiimm16*: 16-bit unsigned field, with a range of 0 through 65,535 (0x0000 through 0xFFFF)

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

Functional Description

The Load Immediate instruction loads immediate values, or explicit constants, into registers.

The instruction loads a 7-bit or 16-bit quantity, depending on the size of the immediate data. The range of constants that can be loaded is 0x8000 through 0x7FFF, equivalent to –32768 through +32767.

The only values that can be immediately loaded into 40-bit Accumulator registers are zeros.

Sixteen-bit half-words can be loaded into either the high half or low half of a register. The load operation leaves the unspecified half of the register intact.
Loading a 32-bit value into a register using Load Immediate requires two separate instructions—one for the high and one for the low half. For example, to load the address “foo” into register P3, write:

\[
p3.h = \text{foo} ;
p3.1 = \text{foo} ;
\]

The assembler automatically selects the correct half-word portion of the 32-bit literal for inclusion in the instruction word.

The zero-extended versions fill the upper bits of the destination register with zeros. The sign-extended versions fill the upper bits with the sign of the constant value.

**Flags Affected**

None

**Required Mode**

User & Supervisor

**Parallel Issue**

The accumulator version of the Load Immediate instruction can be issued in parallel with other instructions.

**Example**

\[
\begin{align*}
    r7 &= 63 \ (z) ; \\
    p3 &= 12 \ (z) ; \\
    r0 &= -344 \ (x) ; \\
    r7 &= 436 \ (z) ; \\
    m2 &= 0x89ab \ (z) ; \\
    p1 &= 0x1234 \ (z) ; \\
    m3 &= 0x3456 \ (x) ; \\
    l3.h &= 0xbcde ;
\end{align*}
\]
Instruction Overview

\[ a0 = 0 \]
\[ a1 = 0 \]
\[ a1 = a0 = 0 \]

Also See

Load Pointer Register

Special Applications

Use the Load Immediate instruction to initialize registers.
Load / Store

Load Pointer Register

General Form

\[ P\text{-register} = \lbrack \text{indirect\_address} \rbrack \]

Syntax

\[
\begin{align*}
P\text{reg} & = \lbrack P\text{reg} \rbrack ; \quad /* \text{indirect (a) */} \\
P\text{reg} & = \lbrack P\text{reg} ++ \rbrack ; \quad /* \text{indirect, post-increment (a) */} \\
P\text{reg} & = \lbrack P\text{reg} -- \rbrack ; \quad /* \text{indirect, post-decrement (a) */} \\
P\text{reg} & = \lbrack P\text{reg} + \text{uimm6m4} \rbrack ; \quad /* \text{indexed with small offset (a) */} \\
P\text{reg} & = \lbrack P\text{reg} + \text{uimm17m4} \rbrack ; \quad /* \text{indexed with large offset (b) */} \\
P\text{reg} & = \lbrack P\text{reg} - \text{uimm17m4} \rbrack ; \quad /* \text{indexed with large offset (b) */} \\
P\text{reg} & = \lbrack FP - \text{uimm7m4} \rbrack ; \quad /* \text{indexed FP-relative (a) */}
\end{align*}
\]

Syntax Terminology

\[ P\text{reg}: P5-0, SP, FP \]

\[ \text{uimm6m4}: \text{6-bit unsigned field that must be a multiple of 4, with a range of 0 through 60 bytes} \]

\[ \text{uimm7m4}: \text{7-bit unsigned field that must be a multiple of 4, with a range of 4 through 128 bytes} \]

\[ \text{uimm17m4}: \text{17-bit unsigned field that must be a multiple of 4, with a range of 0 through 131,068 bytes (0x0000 0000 through 0x0001 FFFC)} \]

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.
Instruction Overview

Functional Description

The Load Pointer Register instruction loads a 32-bit P-register with a 32-bit word from an address specified by a P-register.

The indirect address and offset must yield an even multiple of 4 to maintain 4-byte word address alignment. Failure to maintain proper alignment causes a misaligned memory access exception.

Options

The Load Pointer Register instruction supports the following options.

- Post-increment the source pointer by 4 bytes.
- Post-decrement the source pointer by 4 bytes.
- Offset the source pointer with a small (6-bit), word-aligned (multiple of 4), unsigned constant.
- Offset the source pointer with a large (18-bit), word-aligned (multiple of 4), signed constant.
- Frame Pointer (FP) relative and offset with a 7-bit, word-aligned (multiple of 4), negative constant.

The indexed FP-relative form is typically used to access local variables in a subroutine or function. Positive offsets relative to FP (useful to access arguments from a called function) can be accomplished using one of the other versions of this instruction. Preg includes the Frame Pointer and Stack Pointer.

Auto-increment or auto-decrement pointer registers cannot also be the destination of a Load instruction. For example, $sp=[sp++]$ is not a valid instruction because it prescribes two competing values for the Stack Pointer—the data returned from memory, and post-incremented $sp++$. Similarly, $P0=[P0++]$ and $P1=[P1++]$, etc. are invalid. Such an instruction causes an undefined instruction exception.
Flags Affected

None

Required Mode

User & Supervisor

Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For more information, see “Issuing Parallel Instructions” on page 20-1.

The 32-bit versions of this instruction cannot be issued in parallel with other instructions.

Example

```
p3 = [ p2 ] ;
p5 = [ p0 ++ ] ;
p2 = [ sp -- ] ;
p3 = [ p2 + 8 ] ;
p0 = [ p2 + 0x4008 ] ;
p1 = [ fp - 16 ] :
```

Also See

Load Immediate, SP++ (Pop), SP++ (Pop Multiple)

Special Applications

None
Instruction Overview

Load Data Register

General Form

\[ D\text{-register} = [ \text{indirect}\_address]\]

Syntax

\[
\begin{align*}
D\text{reg} & = [\ P\text{reg} \ ]; \quad /* \text{indirect (a)} */ \\
D\text{reg} & = [\ P\text{reg }++ \ ]; \quad /* \text{indirect, post-increment (a)} */ \\
D\text{reg} & = [\ P\text{reg }-- \ ]; \quad /* \text{indirect, post-decrement (a)} */ \\
D\text{reg} & = [\ P\text{reg }+ uimm6m4 \ ]; \quad /* \text{indexed with small offset (a)} */ \\
D\text{reg} & = [\ P\text{reg }+ uimm17m4 \ ]; \quad /* \text{indexed with large offset (b)} */ \\
D\text{reg} & = [\ P\text{reg }- uimm17m4 \ ]; \quad /* \text{indexed with large offset (b)} */ \\
D\text{reg} & = [\ P\text{reg }++ P\text{reg} \ ]; \quad /* \text{indirect, post-increment index (a)} */^1 \\
D\text{reg} & = [\ FP - uimm7m4 \ ]; \quad /* \text{indexed FP-relative (a)} */ \\
D\text{reg} & = [\ I\text{reg} \ ]; \quad /* \text{indirect (a)} */ \\
D\text{reg} & = [\ I\text{reg }++ \ ]; \quad /* \text{indirect, post-increment (a)} */ \\
D\text{reg} & = [\ I\text{reg }-- \ ]; \quad /* \text{indirect, post-decrement (a)} */ \\
D\text{reg} & = [\ I\text{reg }++ M\text{reg} \ ]; \quad /* \text{indirect, post-increment index (a)} */^1 \\
\end{align*}
\]

Syntax Terminology

\[D\text{reg}: \ R7-0\]

\[P\text{reg}: \ P5-0, \ SP, \ FP\]

\[I\text{reg}: \ I3-0\]

\[M\text{reg}: \ M3-0\]

---

1 See “Indirect and Post-Increment Index Addressing” on page 8-12.
Load / Store

\textit{uimm6m4:} 6-bit unsigned field that must be a multiple of 4, with a range of 0 through 60 bytes

\textit{uimm7m4:} 7-bit unsigned field that must be a multiple of 4, with a range of 4 through 128 bytes

\textit{uimm17m4:} 17-bit unsigned field that must be a multiple of 4, with a range of 0 through 131,068 bytes (0x0000 0000 through 0x0001 FFFC)

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

Functional Description

The Load Data Register instruction loads a 32-bit word into a 32-bit D-register from a memory location. The Source Pointer register can be a P-register, I-register, or the Frame Pointer.

The indirect address and offset must yield an even multiple of 4 to maintain 4-byte word address alignment. Failure to maintain proper alignment causes a misaligned memory access exception.

The instruction versions that explicitly modify \textit{Ireg} support optional circular buffering. See “Automatic Circular Addressing” on page 1-21 for more details. Unless circular buffering is desired, disable it prior to issuing this instruction by clearing the Length Register (\textit{Lreg}) corresponding to the \textit{Ireg} used in this instruction. Example: If you use \textit{I2} to increment your address pointer, first clear \textit{L2} to disable circular buffering. Failure to explicitly clear \textit{Lreg} beforehand can result in unexpected Ireg values.

The circular address buffer registers (Index, Length, and Base) are not initialized automatically by Reset. Traditionally, user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes them later, if needed.
Instruction Overview

Options

The Load Data Register instruction supports the following options.

- Post-increment the source pointer by 4 bytes to maintain word alignment.
- Post-decrement the source pointer by 4 bytes to maintain word alignment.
- Offset the source pointer with a small (6-bit), word-aligned (multiple of 4), unsigned constant.
- Offset the source pointer with a large (18-bit), word-aligned (multiple of 4), signed constant.
- Frame Pointer (FP) relative and offset with a 7-bit, word-aligned (multiple of 4), negative constant.

The indexed FP-relative form is typically used to access local variables in a subroutine or function. Positive offsets relative to FP (useful to access arguments from a called function) can be accomplished using one of the other versions of this instruction. \textit{Preg} includes the Frame Pointer and Stack Pointer.

Indirect and Post-Increment Index Addressing

The syntax of the form:

\[ \text{Dest} = [ \text{Src}_1 ++ \text{Src}_2 ] \]

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

\begin{verbatim}
Dest = [Src_1] ; /* load the 32-bit destination, indirect*/
Src_1 += Src_2 ; /* post-increment Src_1 by a quantity indexed by Src_2*/
\end{verbatim}
Load / Store

where:

- **Dest** is the destination register. *(Dreg* in the syntax example).

- **Src_1** is the first source register on the right-hand side of the equation.

- **Src_2** is the second source register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate P-registers for the input operands. If a common Preg is used for the inputs, the auto-increment feature does not work.

**Flags Affected**

None

**Required Mode**

User & Supervisor

**Parallel Issue**

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For more information, see “Issuing Parallel Instructions” on page 20-1.

The 32-bit versions of this instruction cannot be issued in parallel with other instructions.

**Example**

```assembly
r3 = [ p0 ] ;
r7 = [ p1 ++ ] ;
r2 = [ sp -- ] ;
r6 = [ p2 + 12 ] ;
r0 = [ p4 + 0x800C ] ;
```
Instruction Overview

```
r1 = [ p0 ++ p1 ];
r5 = [ fp -12 ];
r2 = [ i2 ];
r0 = [ i0 ++ ];
r0 = [ i0 -- ];
/* Before indirect post-increment indexed addressing*/
r7 = 0;
i3 = 0x4000; /* Memory location contains 15, for example.*/
m0 = 4;
r7 = [i3 ++ m0];
/* Afterwards . . .*/
/* r7 = 15 from memory location 0x4000*/
/* i3 = i3 + m0 = 0x4004*/
/* m0 still equals 4*/
```

Also See

Load Immediate

Special Applications

None
Load / Store

Load Half-Word – Zero-Extended

General Form

D-register = W [ indirect_address ] (Z)

Syntax

Dreg = W [ Preg ] (Z) ; /* indirect (a)*/
Dreg = W [ Preg ++ ] (Z) ; /* indirect, post-increment (a)*/
Dreg = W [ Preg -- ] (Z) ; /* indirect, post-decrement (a)*/
Dreg = W [ Preg + uimm5m2 ] (Z) ; /* indexed with small offset (a) */
Dreg = W [ Preg + uimm16m2 ] (Z) ; /* indexed with large offset (b) */
Dreg = W [ Preg - uimm16m2 ] (Z) ; /* indexed with large offset (b) */
Dreg = W [ Preg ++ Preg ] (Z) ; /* indirect, post-increment index (a) */\(^1\)

Syntax Terminology

Dreg: R7-0
Preg: P5-0, SP, FP

uimm5m2: 5-bit unsigned field that must be a multiple of 2, with a range of 0 through 30 bytes
uimm16m2: 16-bit unsigned field that must be a multiple of 2, with a range of 0 through 65,534 bytes (0x0000 through 0xFFFC)

\(^1\) See "Indirect and Post-Increment Index Addressing" on page 8-17.
Instruction Overview

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

Functional Description

The Load Half-Word – Zero-Extended instruction loads 16 bits from a memory location into the lower half of a 32-bit data register. The instruction zero-extends the upper half of the register. The Pointer register is a P-register.

The indirect address and offset must yield an even numbered address to maintain 2-byte half-word address alignment. Failure to maintain proper alignment causes a misaligned memory access exception.

Options

The Load Half-Word – Zero-Extended instruction supports the following options.

- Post-increment the source pointer by 2 bytes.
- Post-decrement the source pointer by 2 bytes.
- Offset the source pointer with a small (5-bit), half-word-aligned (even), unsigned constant.
- Offset the source pointer with a large (17-bit), half-word-aligned (even), signed constant.
Indirect and Post-Increment Index Addressing

The syntax of the form:

\[ \text{Dest} = \text{W}[\text{Src}_1 ++ \text{Src}_2] \]

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

\[
\begin{align*}
\text{Dest} &= [\text{Src}_1] ; \quad /* \text{load the 32-bit destination, indirect} */ \\
\text{Src}_1 &= \text{Src}_2 ; \quad /* \text{post-increment Src}_1 \text{ by a quantity indexed by Src}_2 */
\end{align*}
\]

where:

- \( \text{Dest} \) is the destination register. (\text{Dreg} in the syntax example).
- \( \text{Src}_1 \) is the first source register on the right-hand side of the equation.
- \( \text{Src}_2 \) is the second source register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate P-registers for the input operands. If a common \text{Preg} is used for the inputs, the instruction functions as a simple, non-incrementing load. For example, \( r0 = \text{W}[p2++p2](z) \) functions as \( r0 = \text{W}[p2](z) \).

Flags Affected

None

Required Mode

User & Supervisor
Instruction Overview

Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For more information, see “Issuing Parallel Instructions” on page 20-1.

The 32-bit versions of this instruction cannot be issued in parallel with other instructions.

Example

\[
\begin{align*}
    r3 &= w [ \ p0 \ ] (z) ; \\
    r7 &= w [ \ p1 ++ \ ] (z) ; \\
    r2 &= w [ \ sp -- \ ] (z) ; \\
    r6 &= w [ \ p2 + 12 \ ] (z) ; \\
    r0 &= w [ \ p4 + 0x8004 \ ] (z) ; \\
    r1 &= w [ \ p0 ++ p1 \ ] (z) ;
\end{align*}
\]

Also See

Load Half-Word – Sign-Extended, Load Low Data Register Half, Load High Data Register Half, Load Data Register

Special Applications

To read consecutive, aligned 16-bit values for high-performance DSP operations, use the Load Data Register instructions instead of these Half-Word instructions. The Half-Word Load instructions use only half the available 32-bit data bus bandwidth, possibly imposing a bottleneck constriction in the data flow rate.
Load Half-Word – Sign-Extended

General Form

\[ D\text{-register} = W[\text{indirect\_address}](X) \]

Syntax

\[ Dreg = W[\text{Preg}](X); // \text{indirect} (a) \]
\[ Dreg = W[\text{Preg ++}](X); // \text{indirect, post-increment} (a) \]
\[ Dreg = W[\text{Preg --}](X); // \text{indirect, post-decrement} (a) \]
\[ Dreg = W[\text{Preg + uimm5m2}](X); /* \text{indexed with small offset} (a) */ \]
\[ Dreg = W[\text{Preg + uimm16m2}](X); /* \text{indexed with large offset} (b) */ \]
\[ Dreg = W[\text{Preg - uimm16m2}](X); /* \text{indexed with large offset} (b) */ \]
\[ Dreg = W[\text{Preg ++ Preg}](X); /* \text{indirect, post-increment index} (a) */^1 \]

Syntax Terminology

\[ Dreg: R7-0 \]
\[ Preg: P5-0, SP, FP \]
\[ uimm5m2: 5\text{-bit unsigned field that must be a multiple of 2, with a range of 0 through 30 bytes} \]
\[ uimm16m2: 16\text{-bit unsigned field that must be a multiple of 2, with a range of –0 through 65,534 bytes (0x0000 through 0xFFFE)} \]

---

^1 See "Indirect and Post-Increment Index Addressing" on page 8-21.
Instruction Overview

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

Functional Description

The Load Half-Word – Sign-Extended instruction loads 16 bits sign-extended from a memory location into a 32-bit data register. The Pointer register is a P-register. The MSB of the number loaded is replications in the whole upper-half word of the destination D-register.

The indirect address and offset must yield an even numbered address to maintain 2-byte half-word address alignment. Failure to maintain proper alignment causes a misaligned memory access exception.

Options

The Load Half-Word – Sign-Extended instruction supports the following options.

- Post-increment the source pointer by 2 bytes.
- Post-decrement the source pointer by 2 bytes.
- Offset the source pointer with a small (5-bit), half-word-aligned (even), unsigned constant.
- Offset the source pointer with a large (17-bit), half-word-aligned (even), signed constant.
Indirect and Post-Increment Index Addressing

The syntax of the form:

\( \text{Dest} = W[\text{Src}_1 ++ \text{Src}_2 ](X) \)

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

\( \text{Dest} = [\text{Src}_1]; /* load the 32-bit destination, indirect*/ \\
\text{Src}_1 += \text{Src}_2; /* post-increment \text{Src}_1 by a quantity indexed by \text{Src}_2 */ \)

where:

- \( \text{Dest} \) is the destination register. (\( \text{Dreg} \) in the syntax example).
- \( \text{Src}_1 \) is the first source register on the right-hand side of the equation.
- \( \text{Src}_2 \) is the second source register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate P-registers for the input operands. If a common Preg is used for the inputs, the instruction functions as a simple, non-incrementing load. For example, \( r0 = W[p2++p2] \) functions as \( r0 = W[p2] \).

Flags Affected

None

Required Mode

User & Supervisor
Instruction Overview

Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For more information, see “Issuing Parallel Instructions” on page 20-1.

The 32-bit versions of this instruction cannot be issued in parallel with other instructions.

Example

r3 = w [ p0 ] (x);
r7 = w [ p1 ++ ] (x);
r2 = w [ sp -- ] (x);
r6 = w [ p2 + 12 ] (x);
r0 = w [ p4 + 0x800E ] (x);
r1 = w [ p0 ++ p1 ] (x);

Also See

Load Half-Word – Zero-Extended, Load Low Data Register Half, Load High Data Register Half

Special Applications

To read consecutive, aligned 16-bit values for high-performance DSP operations, use the Load Data Register instructions instead of these Half-Word instructions. The Half-Word Load instructions use only half the available 32-bit data bus bandwidth, possibly imposing a bottleneck constriction in the data flow rate.
Load / Store

Load High Data Register Half

General Form

\[ Dreg\_hi = W \[ \text{indirect\_address} \] \]

Syntax

\[
\begin{align*}
Dreg\_hi & = W \[ Ireg \] ; /* indirect data addressing (a) */ \\
Dreg\_hi & = W \[ Ireg ++ \] ; /* indirect, post-increment data addressing (a) */ \\
Dreg\_hi & = W \[ Ireg -- \] ; /* indirect, post-decrement data addressing (a) */ \\
Dreg\_hi & = W \[ Preg \] ; /* indirect (a) */ \\
Dreg\_hi & = W \[ Preg ++ Preg \] ; /* indirect, post-increment index (a) */^{1}
\end{align*}
\]

Syntax Terminology

- \( Dreg\_hi \): R7-0.H
- \( Preg \): P5-0, SP, FP
- \( Ireg \): I3-0

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Load High Data Register Half instruction loads 16 bits from a memory location indicated by an I-register or a P-register into the most significant half of a 32-bit data register. The operation does not affect the least significant half.

---

1 See "Indirect and Post-Increment Index Addressing" on page 8-25.
The indirect address must be even to maintain 2-byte half-word address alignment. Failure to maintain proper alignment causes a misaligned memory access exception.

The instruction versions that explicitly modify Ireg support optional circular buffering. See “Automatic Circular Addressing” on page 1-21 for more details. Unless circular buffering is desired, disable it prior to issuing this instruction by clearing the Length Register (Lreg) corresponding to the Ireg used in this instruction. Example: If you use I2 to increment your address pointer, first clear L2 to disable circular buffering. Failure to explicitly clear Lreg beforehand can result in unexpected Ireg values.

The circular address buffer registers (Index, Length, and Base) are not initialized automatically by Reset. Traditionally, user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes them later, if needed.

Options

The Load High Data Register Half instruction supports the following options.

- Post-increment the source pointer I-register by 2 bytes to maintain half-word alignment.
- Post-decrement the source pointer I-register by 2 bytes to maintain half-word alignment.
Load / Store

Indirect and Post-Increment Index Addressing

\[ \text{Dst}_\text{hi} = [\text{Src}_1 \, \text{++} \, \text{Src}_2 ] \]

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

\[ \text{Dst}_\text{hi} = [\text{Src}_1] ; \quad /* \text{load the half-word into the upper half of} \]
\[ \quad \text{the destination register, indirect*/} \]
\[ \text{Src}_1 \, \text{+} = \text{Src}_2 ; \quad /* \text{post-increment } \text{Src}_1 \text{ by a quantity indexed} \]
\[ \quad \text{by } \text{Src}_2 */ \]

where:

- \( \text{Dst}_\text{hi} \) is the most significant half of the destination register. (\( \text{Dreg}_\text{hi} \) in the syntax example).
- \( \text{Src}_1 \) is the memory source pointer register on the right-hand side of the syntax.
- \( \text{Src}_2 \) is the increment pointer register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate P-registers for the input operands. If a common Preg is used for the inputs, the instruction functions as a simple, non-incrementing load. For example, \( r0.h = W[p2++p2] \) functions as \( r0.h = W[p2] \).

Flags Affected

None

Required Mode

User & Supervisor
Instruction Overview

Parallel Issue

This instruction can be issued in parallel with specific other instructions. For more information, see “Issuing Parallel Instructions” on page 20-1.

Example

r3.h = w [ i1 ] ;
r7.h = w [ i3 ++ ] ;
r1.h = w [ i0 -- ] ;
r2.h = w [ p4 ] ;
r5.h = w [ p2 ++ p0 ] ;

Also See

Load Low Data Register Half, Load Half-Word – Zero-Extended, Load Half-Word – Sign-Extended

Special Applications

To read consecutive, aligned 16-bit values for high-performance DSP operations, use the Load Data Register instructions instead of these Half-Word instructions. The Half-Word Load instructions use only half the available 32-bit data bus bandwidth, possibly imposing a bottleneck constriction in the data flow rate.
Load Low Data Register Half

General Form

\[ Dreg_{lo} = W \ [ \text{indirect\_address} \] \]

Syntax

\[ Dreg_{lo} = W \ [ \text{Ireg} \] ; /* indirect data addressing (a)*/ \]
\[ Dreg_{lo} = W \ [ \text{Ireg} \ ++ \ ] ; /* indirect, post-increment data addressing (a) */ \]
\[ Dreg_{lo} = W \ [ \text{Ireg} \ -- \ ] ; /* indirect, post-decrement data addressing (a) */ \]
\[ Dreg_{lo} = W \ [ \text{Preg} \] ; /* indirect (a) */ \]
\[ Dreg_{lo} = W \ [ \text{Preg} \ ++ \ \text{Preg} \] ; /* indirect, post-increment index (a) */ \]

Syntax Terminology

\[ Dreg_{lo}: R7-0.L \]
\[ Preg: P5-0, SP, FP \]
\[ Ireg: I3-0 \]

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Load Low Data Register Half instruction loads 16 bits from a memory location indicated by an I-register or a P-register into the least significant half of a 32-bit data register. The operation does not affect the most significant half of the data register.

\[ \text{1 See "Indirect and Post-Increment Index Addressing" on page 8-29.} \]
Instruction Overview

The indirect address must be even to maintain 2-byte half-word address alignment. Failure to maintain proper alignment causes an misaligned memory access exception.

The instruction versions that explicitly modify Ireg support optional circular buffering. See “Automatic Circular Addressing” on page 1-21 for more details. Unless circular buffering is desired, disable it prior to issuing this instruction by clearing the Length Register (Lreg) corresponding to the Ireg used in this instruction. Example: If you use I2 to increment your address pointer, first clear L2 to disable circular buffering. Failure to explicitly clear Lreg beforehand can result in unexpected Ireg values.

The circular address buffer registers (Index, Length, and Base) are not initialized automatically by Reset. Traditionally, user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes them later, if needed.

Options

The Load Low Data Register Half instruction supports the following options.

- Post-increment the source pointer I-register by 2 bytes.
- Post-decrement the source pointer I-register by 2 bytes.
Indirect and Post-Increment Index Addressing

The syntax of the form:

\[ Dst_{lo} = [\ Src_1 +\ Src_2 ] \]

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

\[ Dst_{lo} = [\ Src_1 ]; \quad /* load the half-word into the lower half of the destination register, indirect*/ \]
\[ Src_1 += Src_2 ; \quad /* post-increment Src_1 by a quantity indexed by Src_2 */ \]

where:

- \( Dst_{lo} \) is the least significant half of the destination register. (\( Dreg_{lo} \) in the syntax example).
- \( Src_1 \) is the memory source pointer register on the right side of the syntax.
- \( Src_2 \) is the increment index register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate P-registers for the input operands. If a common Preg is used for the inputs, the instruction functions as a simple, non-incrementing load. For example, \( r0.l = W[p2++p2] \) functions as \( r0.l = W[p2] \).

Flags Affected

None

Required Mode

User & Supervisor
Instruction Overview

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Parallel Issue

This instruction can be issued in parallel with specific other instructions. For more information, see “Issuing Parallel Instructions” on page 20-1.

Example

\[ r3.1 = w[i1] ; \]
\[ r7.1 = w[i3 ++] ; \]
\[ r1.1 = w[i0 --] ; \]
\[ r2.1 = w[p4] ; \]
\[ r5.1 = w[p2 ++ p0] ; \]

Also See

Load High Data Register Half, Load Half-Word – Zero-Extended, Load Half-Word – Sign-Extended

Special Applications

To read consecutive, aligned 16-bit values for high-performance DSP operations, use the Load Data Register instructions instead of these Half-Word instructions. The Half-Word Load instructions use only half of the available 32-bit data bus bandwidth, possibly imposing a bottleneck constriction in the data flow rate.
Load Byte – Zero-Extended

General Form

\[ D\text{-register} = B \left[ \text{indirect\_address} \right] (Z) \]

Syntax

\[
\begin{align*}
D\text{reg} &= B \left[ \text{Preg} \right] (Z) ; \quad /* \text{indirect (a)}*/ \\
D\text{reg} &= B \left[ \text{Preg ++} \right] (Z) ; \quad /* \text{indirect, post-increment (a)}*/ \\
D\text{reg} &= B \left[ \text{Preg --} \right] (Z) ; \quad /* \text{indirect, post-decrement (a)}*/ \\
D\text{reg} &= B \left[ \text{Preg + uimm15} \right] (Z) ; \quad /* \text{indexed with offset (b)}*/ \\
D\text{reg} &= B \left[ \text{Preg - uimm15} \right] (Z) ; \quad /* \text{indexed with offset (b)}*/
\end{align*}
\]

Syntax Terminology

\[
\begin{align*}
D\text{reg}: \text{R7–0} \\
P\text{reg}: \text{P5–0, SP, FP}
\end{align*}
\]

\[ uimm15: 15\text{-bit unsigned field, with a range of 0 through 32,767 bytes (}0x0000\text{ through }0x7FFF) \]

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

Functional Description

The Load Byte – Zero-Extended instruction loads an 8-bit byte, zero-extended to 32 bits indicated by an I-register or a P-register, from a memory location into a 32-bit data register. Fill the D-register bits 31–8 with zeros.

The indirect address and offset have no restrictions for memory address alignment.
Instruction Overview

Options

The Load Byte – Zero-Extended instruction supports the following options.

- Post-increment the source pointer by 1 byte.
- Post-decrement the source pointer by 1 byte.
- Offset the source pointer with a 16-bit signed constant.

Flags Affected

None

Required Mode

User & Supervisor

Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For more information, see “Issuing Parallel Instructions” on page 20-1.

The 32-bit versions of this instruction cannot be issued in parallel with other instructions.
Load / Store

Example

\[
\begin{align*}
  r3 &= b [ p0 ] (z) ; \\
  r7 &= b [ p1 ++ ] (z) ; \\
  r2 &= b [ sp -- ] (z) ; \\
  r0 &= b [ p4 + 0xFFFFB00F ] (z) ;
\end{align*}
\]

Also See

Load Byte – Sign-Extended

Special Applications

None
Instruction Overview

Load Byte – Sign-Extended

General Form

\[ D\text{-register} = B [ \text{indirect\_address} ] (X) \]

Syntax

\[
\begin{align*}
D\text{reg} &= B [ \text{Preg} ] (X) ; \quad /* \text{indirect (a)*/} \\
D\text{reg} &= B [ \text{Preg ++} ] (X) ; \quad /* \text{indirect, post-increment (a)*/} \\
D\text{reg} &= B [ \text{Preg --} ] (X) ; \quad /* \text{indirect, post-decrement (a)*/} \\
D\text{reg} &= B [ \text{Preg + uimm15} ] (X) ; \quad /* \text{indexed with offset (b)*/} \\
D\text{reg} &= B [ \text{Preg - uimm15} ] (X) ; \quad /* \text{indexed with offset (b)*/}
\end{align*}
\]

Syntax Terminology

\[ D\text{reg}: \text{R7–0} \]
\[ \text{Preg}: \text{P5–0, SP, FP} \]

\[ uimm15: \text{15-bit unsigned field, with a range of 0 through 32,767 bytes (0x0000 through 0x7FFF)} \]

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

Functional Description

The Load Byte – Sign-Extended instruction loads an 8-bit byte, sign-extended to 32 bits, from a memory location indicated by a P-register into a 32-bit data register. The Pointer register is a P-register. Fill the D-register bits 31–8 with the most significant bit of the loaded byte.

The indirect address and offset have no restrictions for memory address alignment.
Options

The Load Byte – Sign-Extended instruction supports the following options.

- Post-increment the source pointer by 1 byte.
- Post-decrement the source pointer by 1 byte.
- Offset the source pointer with a 16-bit signed constant.

Flags Affected

None

Required Mode

User & Supervisor

Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For more information, see “Issuing Parallel Instructions” on page 20-1.

The 32-bit versions of this instruction cannot be issued in parallel with other instructions.
Instruction Overview

Example

\[
\begin{align*}
r3 &= b[\ p0\ ](x) \\
r7 &= b[\ p1++\ ](x) \\
r2 &= b[\ sp--\ ](x) \\
r0 &= b[\ p4 + 0xFFFF800F\ ](x)
\end{align*}
\]

Also See

Load Byte – Zero-Extended

Special Applications

None
Store Pointer Register

General Form

\[
\text{[ indirect_address \]} = \text{P-register}
\]

Syntax

\[
\begin{align*}
\text{[ Preg \]} & = \text{Preg} ; \quad /* \text{indirect (a)*}/ \\
\text{[ Preg ++ \]} & = \text{Preg} ; \quad /* \text{indirect, post-increment (a)*}/ \\
\text{[ Preg -- \]} & = \text{Preg} ; \quad /* \text{indirect, post-decrement (a)*}/ \\
\text{[ Preg + uimm6m4 \]} & = \text{Preg} ; \quad /* \text{indexed with small offset (a)*}/ \\
\text{[ Preg + uimm17m4 \]} & = \text{Preg} ; \quad /* \text{indexed with large offset (b)*}/ \\
\text{[ Preg - uimm17m4 \]} & = \text{Preg} ; \quad /* \text{indexed with large offset (b)*}/ \\
\text{[ FP - uimm7m4 \]} & = \text{Preg} ; \quad /* \text{indexed FP-relative (a)*}/ \\
\end{align*}
\]

Syntax Terminology

\text{Preg}: \text{P5-0, SP, FP}

\text{uimm6m4}: 6-bit unsigned field that must be a multiple of 4, with a range of 0 through 60 bytes

\text{uimm7m4}: 7-bit unsigned field that must be a multiple of 4, with a range of 4 through 128 bytes

\text{uimm17m4}: 17-bit unsigned field that must be a multiple of 4, with a range of 0 through 131,068 bytes (0x000 0000 through 0x0001 FFFC)

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.
Instruction Overview

Functional Description

The Store Pointer Register instruction stores the contents of a 32-bit P-register to a 32-bit memory location. The Pointer register is a P-register.

The indirect address and offset must yield an even multiple of 4 to maintain 4-byte word address alignment. Failure to maintain proper alignment causes a misaligned memory access exception.

Options

The Store Pointer Register instruction supports the following options.

- Post-increment the destination pointer by 4 bytes.
- Post-decrement the destination pointer by 4 bytes.
- Offset the source pointer with a small (6-bit), word-aligned (multiple of 4), unsigned constant.
- Offset the source pointer with a large (18-bit), word-aligned (multiple of 4), signed constant.
- Frame Pointer (FP) relative and offset with a 7-bit, word-aligned (multiple of 4), negative constant.

The indexed FP-relative form is typically used to access local variables in a subroutine or function. Positive offsets relative to FP (useful to access arguments from a called function) can be accomplished using one of the other versions of this instruction. Preg includes the Frame Pointer and Stack Pointer.

Flags Affected

None
Load / Store

Required Mode

User & Supervisor

Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For more information, see “Issuing Parallel Instructions” on page 20-1.

The 32-bit versions of this instruction cannot be issued in parallel with other instructions.

Example

\[
\begin{align*}
[ \ p2 \ ] &= p3 ; \\
[ \ sp ++ ] &= p5 ; \\
[ \ p0 -- ] &= p2 ; \\
[ \ p2 + 8 ] &= p3 ; \\
[ \ p2 + 0x4444 ] &= p0 ; \\
[ \ fp -12 ] &= p1 ;
\end{align*}
\]

Also See

--SP (Push), --SP (Push Multiple)

Special Applications

None
Instruction Overview

Store Data Register

General Form

\[
\text{[ indirect_address ]} = \text{D-register}
\]

Syntax

Using Pointer Registers

\[
\begin{align*}
\text{[ Preg ]} &= \text{Dreg} ; \quad /* \text{indirect (a)*}/ \\
\text{[ Preg ++ ]} &= \text{Dreg} ; \quad /* \text{indirect, post-increment (a)*}/ \\
\text{[ Preg -- ]} &= \text{Dreg} ; \quad /* \text{indirect, post-decrement (a)*}/ \\
\text{[ Preg + uimm6m4 ]} &= \text{Dreg} ; \quad /* \text{indexed with small offset (a)*}/ \\
\text{[ Preg + uimm17m4 ]} &= \text{Dreg} ; \quad /* \text{indexed with large offset (b)*}/ \\
\text{[ Preg - uimm17m4 ]} &= \text{Dreg} ; \quad /* \text{indexed with large offset (b)*}/ \\
\text{[ Preg ++ Preg ]} &= \text{Dreg} ; \quad /* \text{indirect, post-increment index (a)*/1} \\
\text{[ FP - uimm7m4 ]} &= \text{Dreg} ; \quad /* \text{indexed FP-relative (a)*}/
\end{align*}
\]

Using Data Address Generator (DAG) Registers

\[
\begin{align*}
\text{[ Ireg ]} &= \text{Dreg} ; \quad /* \text{indirect (a)*}/ \\
\text{[ Ireg ++ ]} &= \text{Dreg} ; \quad /* \text{indirect, post-increment (a)*}/ \\
\text{[ Ireg -- ]} &= \text{Dreg} ; \quad /* \text{indirect, post-decrement (a)*}/ \\
\text{[ Ireg ++ Mreg ]} &= \text{Dreg} ; \quad /* \text{indirect, post-increment index (a)*/} \\
\end{align*}
\]

Syntax Terminology

\begin{align*}
\text{Dreg: R7-0} \\
\text{Preg: P5-0, SP, FP} \\
\text{Ireg: I3-0}
\end{align*}

\[1 \text{ See "Indirect and Post-Increment Index Addressing" on page 8-43.}\]
Load / Store

Mreg: M3-0

uimm6m4: 6-bit unsigned field that must be a multiple of 4, with a range of 0 through 60 bytes

uimm7m4: 7-bit unsigned field that must be a multiple of 4, with a range of 4 through 128 bytes

uimm17m4: 17-bit unsigned field that must be a multiple of 4, with a range of 0 through 131,068 bytes (0x0000 through 0xFFFF)

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

Functional Description

The Store Data Register instruction stores the contents of a 32-bit D-register to a 32-bit memory location. The destination Pointer register can be a P-register, I-register, or the Frame Pointer.

The indirect address and offset must yield an even multiple of 4 to maintain 4-byte word address alignment. Failure to maintain proper alignment causes a misaligned memory access exception.

The instruction versions that explicitly modify Ireg support optional circular buffering. See “Automatic Circular Addressing” on page 1-21 for more details. Unless circular buffering is desired, disable it prior to issuing this instruction by clearing the Length Register (Lreg) corresponding to the Ireg used in this instruction.
Instruction Overview

Example: If you use L2 to increment your address pointer, first clear L2 to disable circular buffering. Failure to explicitly clear Lreg beforehand can result in unexpected Ireg values.

The circular address buffer registers (Index, Length, and Base) are not initialized automatically by Reset. Traditionally, user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes them later, if needed.

Options

The Store Data Register instruction supports the following options.

- Post-increment the destination pointer by 4 bytes.
- Post-decrement the destination pointer by 4 bytes.
- Offset the source pointer with a small (6-bit), word-aligned (multiple of 4), unsigned constant.
- Offset the source pointer with a large (18-bit), word-aligned (multiple of 4), signed constant.
- Frame Pointer (FP) relative and offset with a 7-bit, word-aligned (multiple of 4), negative constant.

The indexed FP-relative form is typically used to access local variables in a subroutine or function. Positive offsets relative to FP (such as is useful to access arguments from a called function) can be accomplished using one of the other versions of this instruction. Preg includes the Frame Pointer and Stack Pointer.
Indirect and Post-Increment Index Addressing

The syntax of the form:

\[
[\text{Dst}_1 ++ \text{Dst}_2] = \text{Src}
\]

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

\[
[\text{Dst}_1] = \text{Src} ; \quad /* \text{load the 32-bit source, indirect} */
\text{Dst}_1 += \text{Dst}_2 ; \quad /* \text{post-increment Dst}_1 \text{ by a quantity indexed by Dst}_2 */
\]

where:

- \(\text{Src}\) is the source register. (\(\text{Dreg}\) in the syntax example).
- \(\text{Dst}_1\) is the memory destination register on the left side of the equation.
- \(\text{Dst}_2\) is the increment index register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate P-registers for the input operands. If a common \(\text{Preg}\) is used for the inputs, the auto-increment feature does not work.

Flags Affected

None

Required Mode

User & Supervisor
Instruction Overview

Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For more information, see “Issuing Parallel Instructions” on page 20-1.

The 32-bit versions of this instruction cannot be issued in parallel with other instructions.

Example

\[
\begin{align*}
[p_0] &= r3 \\
[p_1++] &= r7 \\
[sp--] &= r2 \\
[p_2+12] &= r6 \\
[p_4-0x1004] &= r0 \\
[p_0++] &= r1 \\
[fp-28] &= r5 \\
i2 &= r2 \\
i0++] &= r0 \\
i0-- &= r0 \\
i3[++m0] &= r7
\end{align*}
\]

Also See

Load Immediate

Special Applications

None
Store High Data Register Half

General Form

\[ W \ [ \text{indirect\_address} ] = \text{Dreg\_hi} \]

Syntax

\[ W \ [ \text{Ireg} ] = \text{Dreg\_hi} \; \text{/* indirect data addressing (a)*/} \]
\[ W \ [ \text{Ireg} \; \text{++} ] = \text{Dreg\_hi} \; \text{/* indirect, post-increment data addressing (a)*/} \]
\[ W \ [ \text{Ireg} \; \text{--} ] = \text{Dreg\_hi} \; \text{/* indirect, post-decrement data addressing (a)*/} \]
\[ W \ [ \text{Preg} ] = \text{Dreg\_hi} \; \text{/* indirect (a)*/} \]
\[ W \ [ \text{Preg} \; \text{++} \; \text{Preg} ] = \text{Dreg\_hi} \; \text{/* indirect, post-increment index (a)*/}^1 \]

Syntax Terminology

\text{Dreg\_hi}: \text{P7–0.H}
\text{Preg}: \text{P5–0, SP, FP}
\text{Ireg}: \text{I3–0}

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Store High Data Register Half instruction stores the most significant 16 bits of a 32-bit data register to a 16-bit memory location. The Pointer register is either an I-register or a P-register.

---

\(^1\) See “Indirect and Post-Increment Index Addressing” on page 8-47.
Instruction Overview

The indirect address and offset must yield an even number to maintain 2-byte half-word address alignment. Failure to maintain proper alignment causes a misaligned memory access exception.

The instruction versions that explicitly modify $I_{reg}$ support optional circular buffering. See “Automatic Circular Addressing” on page 1-21 for more details. Unless circular buffering is desired, disable it prior to issuing this instruction by clearing the Length Register ($L_{reg}$) corresponding to the $I_{reg}$ used in this instruction.

Example: If you use $I_2$ to increment your address pointer, first clear $L_2$ to disable circular buffering. Failure to explicitly clear $L_{reg}$ beforehand can result in unexpected $I_{reg}$ values.

The circular address buffer registers (Index, Length, and Base) are not initialized automatically by Reset. Traditionally, user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes them later, if needed.

Options

The Store High Data Register Half instruction supports the following options.

- Post-increment the destination pointer $I_{register}$ by 2 bytes.
- Post-decrement the destination pointer $I_{register}$ by 2 bytes.
Indirect and Post-Increment Index Addressing

The syntax of the form:

\[ Dst_1 \mathrel{++} Dst_2 = Src_{hi} \]

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

\[ Dst_1 = Src_{hi} ; \quad /* \text{store the upper half of the source register, indirect} */ \]
\[ Dst_1 += Dst_2 ; \quad /* \text{post-increment } Dst_1 \text{ by a quantity indexed by } Dst_2 */ \]

where:

- \( Src_{hi} \) is the most significant half of the source register. (\( Dreg_{hi} \) in the syntax example).
- \( Dst_1 \) is the memory destination pointer register on the left side of the syntax.
- \( Dst_2 \) is the increment index register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate P-registers for the input operands. If a common \( Preg \) is used for the inputs, the auto-increment feature does not work.

Flags Affected

None

Required Mode

User & Supervisor
Instruction Overview

Parallel Issue

This instruction can be issued in parallel with specific other instructions. For more information, see “Issuing Parallel Instructions” on page 20-1.

Example

\[ w[i1] = r3.h ; \]
\[ w[i3 ++] = r7.h ; \]
\[ w[i0 --] = r1.h ; \]
\[ w[p4] = r2.h ; \]
\[ w[p2 ++ p0] = r5.h ; \]

Also See

Store Low Data Register Half

Special Applications

To write consecutive, aligned 16-bit values for high-performance DSP operations, use the Store Data Register instructions instead of these Half-Word instructions. The Half-Word Store instructions use only half the available 32-bit data bus bandwidth, possibly imposing a bottleneck constriction in the data flow rate.
Load / Store

Store Low Data Register Half

General Form

\[
W \ [ \text{indirect_address} ] = \text{Dreg}_{lo}
\]

\[
W \ [ \text{indirect_address} ] = \text{D-register}
\]

Syntax

\[
W \ [ \text{Ireg} ] = \text{Dreg}_{lo} \quad /* \text{indirect data addressing (a)}/
\]

\[
W \ [ \text{Ireg} ++ ] = \text{Dreg}_{lo} \quad /* \text{indirect, post-increment data}
\]

\[
W \ [ \text{Ireg} -- ] = \text{Dreg}_{lo} \quad /* \text{indirect, post-decrement data}
\]

\[
W \ [ \text{Preg} ] = \text{Dreg}_{lo} \quad /* \text{indirect (a)}*/
\]

\[
W \ [ \text{Preg} ] = \text{Dreg} \quad /* \text{indirect (a)}*/
\]

\[
W \ [ \text{Preg} ++ ] = \text{Dreg} \quad /* \text{indirect, post-increment (a)}*/
\]

\[
W \ [ \text{Preg} -- ] = \text{Dreg} \quad /* \text{indirect, post-decrement (a)}*/
\]

\[
W \ [ \text{Preg} + \text{uimm}_{5m2} ] = \text{Dreg} \quad /* \text{indexed with small offset (a)}/
\]

\[
W \ [ \text{Preg} + \text{uimm}_{16m2} ] = \text{Dreg} \quad /* \text{indexed with large offset (b)}/
\]

\[
W \ [ \text{Preg} - \text{uimm}_{16m2} ] = \text{Dreg} \quad /* \text{indexed with large offset (b)}/
\]

\[
W \ [ \text{Preg} ++ \text{Preg} ] = \text{Dreg}_{lo} \quad /* \text{indirect, post-increment}
\]

\[
\text{Syntax Terminology}
\]

\[
\text{Dreg}_{lo}: \text{R7-0.L}
\]

\[
\text{Preg}: \text{P5-0, SP, FP}
\]

\[
\text{Ireg}: \text{I3-0}
\]

---

1 See "Indirect and Post-Increment Index Addressing" on page 8-51.
**Instruction Overview**

*Dreg:* R7-0

*uimm5m2:* 5-bit unsigned field that must be a multiple of 2, with a range of 0 through 30 bytes

*uimm16m2:* 16-bit unsigned field that must be a multiple of 2, with a range of 0 through 65,534 bytes (0x0000 through 0xFFFE)

**Instruction Length**

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

**Functional Description**

The Store Low Data Register Half instruction stores the least significant 16 bits of a 32-bit data register to a 16-bit memory location. The Pointer register is either an I-register or a P-register.

The indirect address and offset must yield an even number to maintain 2-byte half-word address alignment. Failure to maintain proper alignment causes an misaligned memory access exception.

ℹ️ The instruction versions that explicitly modify Ireg support optional circular buffering. See “Automatic Circular Addressing” on page 1-21 for more details. Unless circular buffering is desired, disable it prior to issuing this instruction by clearing the Length Register (Lreg) corresponding to the Ireg used in this instruction. Example: If you use I2 to increment your address pointer, first clear L2 to disable circular buffering. Failure to explicitly clear Lreg beforehand can result in unexpected Ireg values.

The circular address buffer registers (Index, Length, and Base) are not initialized automatically by Reset. Traditionally, user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes them later, if needed.
Options

The Store Low Data Register Half instruction supports the following options.

- Post-increment the destination pointer by 2 bytes.
- Post-decrement the destination pointer by 2 bytes.
- Offset the source pointer with a small (5-bit), half-word-aligned (even), unsigned constant.
- Offset the source pointer with a large (17-bit), half-word-aligned (even), signed constant.

Indirect and Post-Increment Index Addressing

The syntax of the form:

\[ Dst_1 ++ Dst_2 = Src \]

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

\[
[Dst_1] = Src_{lo} \quad /* \text{store the lower half of the source register, indirect} */
\]

\[
Dst_1 += Dst_2 \quad /* \text{post-increment } Dst_1 \text{ by a quantity indexed by } Dst_2 */
\]

where:

- \( Src \) is the least significant half of the source register. (\( Dreg \) or \( Dreg_{lo} \) in the syntax example).
- \( Dst_1 \) is the memory destination pointer register on the left side of the syntax.
- \( Dst_2 \) is the increment index register.
Instruction Overview

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate P-registers for the input operands. If a common Preg is used for the inputs, the auto-increment feature does not work.

Flags Affected

None

Required Mode

User & Supervisor

Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For more information, see “Issuing Parallel Instructions” on page 20-1.

The 32-bit versions of this instruction cannot be issued in parallel with other instructions.

Example

\[
\begin{align*}
\text{w} & \quad [ \text{i1} ] = \text{r3}.l \\
\text{w} & \quad [ \text{p0} ] = \text{r3} \\
\text{w} & \quad [ \text{i3} ++ ] = \text{r7}.l \\
\text{w} & \quad [ \text{i0} -- ] = \text{r1}.l \\
\text{w} & \quad [ \text{p4} ] = \text{r2}.l \\
\text{w} & \quad [ \text{p1} ++ ] = \text{r7} \\
\text{w} & \quad [ \text{sp} -- ] = \text{r2} \\
\text{w} & \quad [ \text{p2} + 12 ] = \text{r6} \\
\text{w} & \quad [ \text{p4} - \text{0x200C} ] = \text{r0} \\
\text{w} & \quad [ \text{p2} ++ \text{p0} ] = \text{r5}.l
\end{align*}
\]
Load / Store

Also See

Store High Data Register Half, Store Data Register

Special Applications

To write consecutive, aligned 16-bit values for high-performance DSP operations, use the Store Data Register instructions instead of these Half-Word instructions. The Half-Word Store instructions use only half the available 32-bit data bus bandwidth, possibly imposing a bottleneck constriction in the data flow rate.
Instruction Overview

**Store Byte**

**General Form**

\[
B \ [ \text{indirect\_address} ] = D\text{-register}
\]

**Syntax**

\[
\begin{align*}
B \ [ \text{Preg} ] &= D\text{reg} ; /* \text{indirect (a)*/} \\
B \ [ \text{Preg ++ } ] &= D\text{reg} ; /* \text{indirect, post-increment (a)*/} \\
B \ [ \text{Preg -- } ] &= D\text{reg} ; /* \text{indirect, post-decrement (a)*/} \\
B \ [ \text{Preg + uimm15 } ] &= D\text{reg} ; /* \text{indexed with offset (b)*/} \\
B \ [ \text{Preg - uimm15 } ] &= D\text{reg} ; /* \text{indexed with offset (b)*/}
\end{align*}
\]

**Syntax Terminology**

- **Dreg**: R7–0
- **Preg**: P5–0, SP, FP
- **uimm15**: 15-bit unsigned field, with a range of 0 through 32,767 bytes (0x0000 through 0x7FFF)

**Instruction Length**

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

**Functional Description**

The Store Byte instruction stores the least significant 8-bit byte of a data register to an 8-bit memory location. The Pointer register is a P-register.

The indirect address and offset have no restrictions for memory address alignment.
Options

The Store Byte instruction supports the following options.

- Post-increment the destination pointer by 1 byte to maintain byte alignment.
- Post-decrement the destination pointer by 1 byte to maintain byte alignment.
- Offset the destination pointer with a 16-bit signed constant.

Flags Affected

None

Required Mode

User & Supervisor

Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For more information, see “Issuing Parallel Instructions” on page 20-1.

The 32-bit versions of this instruction cannot be issued in parallel with other instructions.

Example

\[
\begin{align*}
&b[p0] = r3; \\
&b[p1++] = r7; \\
&b[sp--] = r2; \\
&b[p4+0x100F] = r0; \\
&b[p4-0x53F] = r0;
\end{align*}
\]
Instruction Overview

Also See

None

Special Applications

To write consecutive, 8-bit values for high-performance DSP operations, use the Store Data Register instructions instead of these byte instructions. The byte store instructions use only one fourth the available 32-bit data bus bandwidth, possibly imposing a bottleneck constriction in the data flow rate.
9 MOVE

Instruction Summary

- “Move Register” on page 9-2
- “Move Conditional” on page 9-8
- “Move Half to Full Word – Zero-Extended” on page 9-10
- “Move Half to Full Word – Sign-Extended” on page 9-13
- “Move Register Half” on page 9-15
- “Move Byte – Zero-Extended” on page 9-23
- “Move Byte – Sign-Extended” on page 9-25

Instruction Overview

This chapter discusses the move instructions. Users can take advantage of these instructions to move registers (or register halves), move half words (zero or sign extended), move bytes, and perform conditional moves.
Instruction Overview

Move Register

General Form

\[ \text{dest} = \text{src} \]

Syntax

\[
\begin{align*}
genreg &= \text{genreg} ; \quad /* (a) */ 
genreg &= \text{dagreg} ; \quad /* (a) */ 
dagreg &= \text{genreg} ; \quad /* (a) */ 
dagreg &= \text{dagreg} ; \quad /* (a) */ 
genreg &= \text{USP} ; \quad /* (a) */ 
\text{USP} &= \text{genreg} ; \quad /* (a) */ 
\text{Dreg} &= \text{sysreg} ; \quad /* \text{sysreg to 32-bit D-register (a)} */ 
\text{Preg} &= \text{sysreg} ; \quad /* \text{sysreg to P-register (c)} */ 
\text{sysreg} &= \text{Dreg} ; \quad /* 32-bit D-register to sysreg (a) */ 
\text{sysreg} &= \text{Preg} ; \quad /* 32-bit P-register to sysreg (a) */ 
\text{sysreg} &= \text{USP} ; \quad /* (a) */ 
\text{A0} &= \text{A1} ; \quad /* \text{move 40-bit Accumulator value (b)} */ 
\text{A1} &= \text{A0} ; \quad /* \text{move 40-bit Accumulator value (b)} */ 
\text{A0} &= \text{Dreg} ; \quad /* 32-bit D-register to 40-bit A0, sign extended (b) */ 
\text{A1} &= \text{Dreg} ; \quad /* 32-bit D-register to 40-bit A1, sign extended (b) */ 
\end{align*}
\]

Accumulator to D-register Move:

\[
\begin{align*}
\text{Dreg\_even} &= \text{A0} (\text{opt\_mode}) ; \quad /* \text{move 32-bit A0.W to even Dreg (b)} */ 
\text{Dreg\_odd} &= \text{A1} (\text{opt\_mode}) ; \quad /* \text{move 32-bit A1.W to odd Dreg (b)} */ 
\text{Dreg\_even} &= \text{A0}, \text{Dreg\_odd} &= \text{A1} (\text{opt\_mode}) ; \quad /* \text{move both Accumulators to a register pair (b)} */ 
\text{Dreg\_odd} &= \text{A1}, \text{Dreg\_even} &= \text{A0} (\text{opt\_mode}) ; \quad /* \text{move both Accumulators to a register pair (b)} */ 
\end{align*}
\]
Move

Syntax Terminology

- **genreg**: R7-0, P5-0, SP, FP, A0.X, A0.W, A1.X, A1.W
- **dagreg**: I3-0, M3-0, B3-0, L3-0
- **sysreg**: ASTAT, SEQSTAT, SYSCFG, RETI, RETX, RETN, RETE, RETS, LCO and LC1, LTO and LT1, LBO and LB1, CYCLES, CYCLES2, and EMUDAT
- **USP**: The User Stack Pointer Register
  - **Dreg**: R7-0
  - **Preg**: P5-0, SP, FP
  - **Dreg_even**: R0, R2, R4, R6
  - **Dreg_odd**: R1, R3, R5, R7

When combining two moves in the same instruction, the **Dreg_even** and **Dreg_odd** operands must be members of the same register pair, for example from the set R1:0, R3:2, R5:4, R7:6.

**opt_mode**: Optionally (FU), (S2RND), or (ISS2) (See Table 9-1 on page 9-4).

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length. Comment (c) indicates an instruction that is not valid on the ADSP-BF535 processor.

Functional Description

The Move Register instruction copies the contents of the source register into the destination register. The operation does not affect the source register contents.

All moves from smaller to larger registers are sign extended.
**Instruction Overview**

All moves from 40-bit Accumulators to 32-bit D-registers support saturation.

**Options**

The Accumulator to Data Register Move instruction supports the options listed in the table below.

<table>
<thead>
<tr>
<th>Option</th>
<th>Accumulator Copy Formatting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>Signed fraction. Copy Accumulator 9.31 format to register 1.31 format. Saturate results between minimum -1 and maximum 1-2^{-31}. Signed integer. Copy Accumulator 40.0 format to register 32.0 format. Saturate results between minimum -2^{31} and maximum 2^{31-1}. In either case, the resulting hexadecimal range is minimum 0x8000 0000 through maximum 0x7FFF FFFF. The Accumulator is unaffected by extraction.</td>
</tr>
<tr>
<td>(FU)</td>
<td>Unsigned fraction. Copy Accumulator 8.32 format to register 0.32 format. Saturate results between minimum 0 and maximum 1-2^{-32}. Unsigned integer. Copy Accumulator 40.0 format to register 32.0 format. Saturate results between minimum 0 and maximum 2^{32-1}. In either case, the resulting hexadecimal range is minimum 0x0000 0000 through maximum 0xFFFF FFFF. The Accumulator is unaffected by extraction.</td>
</tr>
</tbody>
</table>
Move

See “Saturation” on page 1-17 for a description of saturation behavior.

Flags Affected

The ASTAT register that contains the flags can be explicitly modified by this instruction.

The Accumulator to D-register Move versions of this instruction affect the following flags.

- $V$ is set if the result written to the D-register file saturates 32 bits; cleared if no saturation. In the case of two simultaneous operations, $V$ represents the logical “OR” of the two.

- $VS$ is set if $V$ is set; unaffected otherwise.

Table 9-1. Accumulator to Data Register Move (Cont’d)

<table>
<thead>
<tr>
<th>Option</th>
<th>Accumulator Copy Formatting</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S2RND)</td>
<td>Signed fraction with scaling. Shift the Accumulator contents one place to the left (multiply x 2). Saturate result to 1.31 format. Copy to destination register. Results range between minimum -1 and maximum $1 - 2^{-31}$. Signed integer with scaling. Shift the Accumulator contents one place to the left (multiply x 2). Saturate result to 32.0 format. Copy to destination register. Results range between minimum -1 and maximum $2^{31} - 1$. In either case, the resulting hexadecimal range is minimum 0x8000 0000 through maximum 0x7FFF FFFF. The Accumulator is unaffected by extraction.</td>
</tr>
<tr>
<td>(ISS2)</td>
<td>Signed fraction with scaling. Shift the Accumulator contents one place to the left (multiply x 2). Saturate result to 1.31 format. Copy to destination register. Results range between minimum -1 and maximum $1 - 2^{-31}$. Signed integer with scaling. Shift the Accumulator contents one place to the left (multiply x 2). Saturate result to 32.0 format. Copy to destination register. Results range between minimum -1 and maximum $2^{31} - 1$. In either case, the resulting hexadecimal range is minimum 0x8000 0000 through maximum 0x7FFF FFFF. The Accumulator is unaffected by extraction.</td>
</tr>
</tbody>
</table>
Instruction Overview

- \( A_Z \) is set if result is zero; cleared if nonzero. In the case of two simultaneous operations, \( A_Z \) represents the logical “OR” of the two.

- \( A_N \) is set if result is negative; cleared if non-negative. In the case of two simultaneous operations, \( A_N \) represents the logical “OR” of the two.

- All other flags are unaffected.

The ADSP-BF535 processor has fewer \( A_{STAT} \) flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor for most cases.

Explicit accesses to \( U_{SP}, S_{EQSTAT}, S_{YSCFG}, R_{ETI}, R_{ETX}, R_{ETN} \) and \( R_{ETE} \) require Supervisor mode. If any of these registers are explicitly accessed from User mode, an Illegal Use of Protected Resource exception occurs.

Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For more information, see “Issuing Parallel Instructions” on page 20-1.

The 16-bit versions of this instruction cannot be issued in parallel with other instructions.

Example

\[
\begin{align*}
  r_3 &= r_0 ; \\
  r_7 &= p_2 ; \\
  r_2 &= a_0 ; \\
  a_0 &= a_1 ;
\end{align*}
\]

9-6 ADSP-BF53x/BF56x Blackfin Processor Programming Reference
Move

a1 = a0 ;
a0 = r7 ; /* move R7 to 32-bit A0.W */
a1 = r3 ; /* move R3 to 32-bit A1.W */
retn = p0 ; /* must be in Supervisor mode */
r2 = a0 ; /* 32-bit move with saturation */
r7 = a1 ; /* 32-bit move with saturation */
r0 = a0 (iss2) ; /* 32-bit move with scaling, truncation and saturation */

Also See

Load Immediate to initialize registers.

Move Register Half to move values explicitly into the A0.X and A1.X registers.

LSETUP, LOOP to implicitly access registers LC0, LT0, LB0, LC1, LT1 and LB1.

Call, RAISE (Force Interrupt / Reset) and RTS, RTI, RTX, RTN, RTE (Return) to implicitly access registers RETI, RETN, and RETS.

Force Exception and Force Emulation to implicitly access registers RETX and RETE.

Special Applications

None
Instruction Overview

Move Conditional

General Form

IF CC dest_reg = src_reg
IF ! CC dest_reg = src_reg

Syntax

IF CC DPreg = DPreg ; /* move if CC = 1 (a) */
IF ! CC DPreg = DPreg ; /* move if CC = 0 (a) */

Syntax Terminology

DPreg: R7-0, P5-0, SP, FP

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Move Conditional instruction moves source register contents into a destination register, depending on the value of CC.

IF CC DPreg = DPreg, the move occurs only if CC = 1.
IF ! CC DPreg = DPreg, the move occurs only if CC = 0.

The source and destination registers are any D-register or P-register.

Flags Affected

None

Required Mode

User & Supervisor
Move

Parallel Issue

The Move Conditional instruction cannot be issued in parallel with other instructions.

Example

```
if cc r3 = r0 ; /* move if CC=1 */
if cc r2 = p4 ;
if cc p0 = r7 ;
if cc p2 = p5 ;
if ! cc r3 = r0 ; /* move if CC=0 */
if ! cc r2 = p4 ;
if ! cc p0 = r7 ;
if ! cc p2 = p5 ;
```

Also See

Compare Accumulator, Move CC, Negate CC, IF CC JUMP

Special Applications

None
Instruction Overview

Move Half to Full Word – Zero-Extended

General Form

\[ \text{dest} \_\text{reg} = \text{src} \_\text{reg} \,(Z) \]

Syntax

\[ \text{Dreg} = \text{Dreg} \_\text{lo} \,(Z) ; \quad /* \,(a) */ \]

Syntax Terminology

\[ \text{Dreg: } R7-0 \]
\[ \text{Dreg} \_\text{lo: } R7-0.L \]

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Move Half to Full Word – Zero-Extended instruction converts an unsigned half word (16 bits) to an unsigned word (32 bits).

The instruction copies the least significant 16 bits from a source register into the lower half of a 32-bit register and zero-extends the upper half of the destination register. The operation supports only D-registers. Zero extension is appropriate for unsigned values. If used with signed values, a small negative 16-bit value will become a large positive value.
Flags Affected

The following flags are affected by the Move Half to Full Word – Zero-Extended instruction.

- $AZ$ is set if result is zero; cleared if nonzero.
- $AN$ is cleared.
- $AC0$ is cleared.
- $V$ is cleared.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer $ASTAT$ flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction cannot be issued in parallel with other instructions.

Example

```c
/* If r0.l = 0xFFFF */
r4 = r0.l (z); /* Equivalent to r4.l = r0.l and r4.h = 0 */
/* . . . then r4 = 0x0000FFFF */
```
Instruction Overview

Also See

Move Half to Full Word – Sign-Extended, Move Register Half

Special Applications

None
**Move Half to Full Word – Sign-Extended**

**General Form**

\[ \text{dest} = \text{src} \ (X) \]

**Syntax**

\[ \text{Dreg} = \text{Dreg}_{lo} \ (X) \] ; /* (a)*/

**Syntax Terminology**

- \( \text{Dreg} \): R7-0
- \( \text{Dreg}_{lo} \): R7-0.L

**Instruction Length**

In the syntax, comment (a) identifies 16-bit instruction length.

**Functional Description**

The Move Half to Full Word – Sign-Extended instruction converts a signed half word (16 bits) to a signed word (32 bits). The instruction copies the least significant 16 bits from a source register into the lower half of a 32-bit register and sign-extends the upper half of the destination register. The operation supports only D-registers.

**Flags Affected**

The following flags are affected by the Move Half to Full Word – Sign-Extended instruction.

- **AZ** is set if result is zero; cleared if nonzero.
- **AN** is set if result is negative; cleared if non-negative.
- **AC0** is cleared.
Instruction Overview

- \( v \) is cleared.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction cannot be issued in parallel with any other instructions.

Example

\[
\begin{align*}
    r4 &= r0.1(x) ; \\
    r4 &= r0.1 ;
\end{align*}
\]

Also See

Move Half to Full Word – Zero-Extended, Move Register Half

Special Applications

None
Move Register Half

General Form

\[
\begin{align*}
\text{dest} \_ \text{reg} \_ \text{half} &= \text{src} \_ \text{reg} \_ \text{half} \\
\text{dest} \_ \text{reg} \_ \text{half} &= \text{accumulator (opt_mode)}
\end{align*}
\]

Syntax

\[
\begin{align*}
\text{A0.X} &= \text{Dreg}_\text{lo} ; & \text{/* least significant 8 bits of Dreg into A0.X (b) */}^1 \\
\text{A1.X} &= \text{Dreg}_\text{lo} ; & \text{/* least significant 8 bits of Dreg into A1.X (b) */} \\
\text{Dreg}_\text{lo} &= \text{A0.X} ; & \text{/* 8-bit A0.X, sign-extended, into least significant 16 bits of Dreg (b) */} \\
\text{Dreg}_\text{lo} &= \text{A1.X} ; & \text{/* 8-bit A1.X, sign-extended, into least significant 16 bits of Dreg (b) */} \\
\text{A0.L} &= \text{Dreg}_\text{lo} ; & \text{/* least significant 16 bits of Dreg into least significant 16 bits of A0.W (b) */} \\
\text{A1.L} &= \text{Dreg}_\text{lo} ; & \text{/* least significant 16 bits of Dreg into least significant 16 bits of A1.W (b) */} \\
\text{A0.H} &= \text{Dreg}_\text{hi} ; & \text{/* most significant 16 bits of Dreg into most significant 16 bits of A0.W (b) */} \\
\text{A1.H} &= \text{Dreg}_\text{hi} ; & \text{/* most significant 16 bits of Dreg into most significant 16 bits of A1.W (b) */}
\end{align*}
\]

---

^1 The Accumulator Extension registers A0.X and A1.X are defined only for the 8 low-order bits 7 through 0 of A0.X and A1.X. This instruction truncates the upper byte of Dreg_lo before moving the value into the Accumulator Extension register (A0.X or A1.X).
Instruction Overview

Accumulator to Half D-register Moves

\[ D_{\text{reg\_lo}} = A_0 \text{ (opt\_mode)} \]; /* move A0 to lower half of Dreg (b) */
\[ D_{\text{reg\_hi}} = A_1 \text{ (opt\_mode)} \]; /* move A1 to upper half of Dreg (b) */

\[ D_{\text{reg\_lo}} = A_0, D_{\text{reg\_hi}} = A_1 \text{ (opt\_mode)} \]; /* move both values at once; must go to the lower and upper halves of the same Dreg (b) */
\[ D_{\text{reg\_hi}} = A_1, D_{\text{reg\_lo}} = A_0 \text{ (opt\_mode)} \]; /* move both values at once; must go to the upper and lower halves of the same Dreg (b) */

Syntax Terminology

\[ D_{\text{reg\_lo}} \]: R7-0.L
\[ D_{\text{reg\_hi}} \]: R7-0.H
\[ A_0.L \]: the least significant 16 bits of Accumulator A0.W
\[ A_1.L \]: the least significant 16 bits of Accumulator A1.W
\[ A_0.H \]: the most significant 16 bits of Accumulator A0.W
\[ A_1.H \]: the most significant 16 bits of Accumulator A1.W

\[ \text{opt\_mode} \]: Optionally (FU), (IS), (IU), (T), (S2RND), (ISS2), or (IH) (See Table 9-2 on page 9-19).

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.
Move

Functional Description

The Move Register Half instruction copies 16 bits from a source register into half of a 32-bit register. The instruction does not affect the unspecified half of the destination register. It supports only D-registers and the Accumulator.

One version of the instruction simply copies the 16 bits (saturated at 16 bits) of the Accumulator into a data half-register. This syntax supports truncation and rounding beyond a simple Move Register Half instruction.

The fraction version of this instruction (the default option) transfers the Accumulator result to the destination register according to the diagrams in Figure 9-1. Accumulator \( A0.H \) contents transfer to the lower half of the destination D-register. \( A1.H \) contents transfer to the upper half of the destination D-register.

![Diagram of result to destination register (default option)](image)

Figure 9-1. Result to Destination Register (Default Option)
Instruction Overview

The integer version of this instruction (the \((IS)\) option) transfers the Accumulator result to the destination register according to the diagrams, shown in Figure 9-2. Accumulator \(A0.L\) contents transfer to the lower half of the destination D-register. \(A1.L\) contents transfer to the upper half of the destination D-register.

Some versions of this instruction are affected by the \(\text{RND\_MOD}\) bit in the \(\text{ASTAT}\) register when they copy the results into the destination register. \(\text{RND\_MOD}\) determines whether biased or unbiased rounding is used. \(\text{RND\_MOD}\) controls rounding for all versions of this instruction except the \((IS)\), \((ISS2)\), \((IU)\), and \((T)\) options.

See “Rounding and Truncating” on page 1-19 for a description of rounding behavior.
The Accumulator to Half D-Register Move instructions support the copy options in Table 9-2.

Table 9-2. Accumulator to Half D-Register Move Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Accumulator Copy Formatting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>Signed fraction format. Round Accumulator 9.31 format value at bit 16. (RND.MOD bit in the ASTAT register controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum 1-2^-15 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). The Accumulator is unaffected by extraction.</td>
</tr>
<tr>
<td>(FU)</td>
<td>Unsigned fraction format. Round Accumulator 8.32 format value at bit 16. (RND.MOD bit in the ASTAT register controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum 1-2^-16 (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). The Accumulator is unaffected by extraction.</td>
</tr>
<tr>
<td>(IS)</td>
<td>Signed integer format. Extract the lower 16 bits of the Accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^15 and maximum 2^15-1 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). The Accumulator is unaffected by extraction.</td>
</tr>
<tr>
<td>(IU)</td>
<td>Unsigned integer format. Extract the lower 16 bits of the Accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum 0 and maximum 2^16-1 (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). The Accumulator is unaffected by extraction.</td>
</tr>
<tr>
<td>(T)</td>
<td>Signed fraction with truncation. Truncate Accumulator 9.31 format value at bit 16. (Perform no rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum 1-2^-15 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). The Accumulator is unaffected by extraction.</td>
</tr>
</tbody>
</table>
To truncate the result, the operation eliminates the least significant bits
that do not fit into the destination register.

When necessary, saturation is performed after the rounding.

See “Saturation” on page 1-17 for a description of saturation behavior.

**Flags Affected**

The Accumulator to Half D-register Move versions of this instruction
affect the following flags.

- $V$ is set if the result written to the half D-register file saturates 16
  bits; cleared if no saturation.
- $V_S$ is set if $V$ is set; unaffected otherwise.
Move

- AZ is set if result is zero; cleared if nonzero.
- AN is set if result is negative; cleared if non-negative.
- All other flags are unaffected.

Flags are not affected by other versions of this instruction.

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For more information, see “Issuing Parallel Instructions” on page 20-1.

Example

a0.x = r1.l ;
a1.x = r4.l ;
r7.l = a0.x ;
r0.l = a1.x ;
a0.l = r2.l ;
a1.l = r1.l ;
a0.l = r5.l ;
a1.l = r3.l ;
a0.h = r7.h ;
a1.h = r0.h ;
r7.l = a0 ; /* copy A0.H into R7.L with saturation. */
r2.h = a0 ; /* copy A0.H into R2.H with saturation. */
Instruction Overview

r3.l = a0, r3.h = a1 ; /* copy both half words; must go to the lower and upper halves of the same Dreg. */
r1.h = a1, r1.l = a0 ; /* copy both half words; must go to the upper and lower halves of the same Dreg.
r0.h = a1 (is) ; /* copy A1.L into R0.H with saturation. */
r5.l = a0 (t) ; /* copy A0.H into R5.L; truncate A0.L; no saturation. */
r1.l = a0 (s2rnd) ; /* copy A0.H into R1.L with scaling, rounding & saturation. */
r2.h = a1 (iss2) ; /* copy A1.L into R2.H with scaling and saturation. */
r6.l = a0 (ih) ; /* copy A0.H into R6.L with saturation, then rounding. */

Also See

Move Half to Full Word – Zero-Extended, Move Half to Full Word – Sign-Extended

Special Applications

None
**Move**

### Move Byte – Zero-Extended

#### General Form

\[ \text{dest} = \text{src} \text{\_reg} \text\_byte} \ (Z) \]

#### Syntax

\[ \text{Dreg} = \text{Dreg} \text\_byte} \ (Z) ; \ /* (a)*/ \]

#### Syntax Terminology

\[ \text{Dreg\_byte}: R7-0.B, \text{the low-order 8 bits of each Data Register} \]

#### Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

#### Functional Description

The Move Byte – Zero-Extended instruction converts an unsigned byte to an unsigned word (32 bits). The instruction copies the least significant 8 bits from a source register into the least significant 8 bits of a 32-bit register. The instruction zero-extends the upper bits of the destination register. This instruction supports only D-registers.

#### Flags Affected

The following flags are affected by the Move Byte – Zero-Extended instruction:

- \( AZ \) is set if result is zero; cleared if nonzero.
- \( AN \) is cleared.
- \( ACO \) is cleared.
Instruction Overview

- \( V \) is cleared.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer \( \text{ASTAT} \) flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction cannot be issued in parallel with any other instructions.

Example

\[ r7 = r2.b(z) ; \]

Also See

Move Register Half to explicitly access the Accumulator Extension registers \( A0.X \) and \( A1.X \).

Move Byte – Sign-Extended

Special Applications

None
Move Byte – Sign-Extended

General Form

\[
dest\_reg = src\_reg\_byte (X)
\]

Syntax

\[
Dreg = Dreg\_byte (X) ; /* (a) */
\]

Syntax Terminology

\emph{Dreg\_byte}: R7-0.B, the low-order 8 bits of each Data Register

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Move Byte – Sign-Extended instruction converts a signed byte to a signed word (32 bits). It copies the least significant 8 bits from a source register into the least significant 8 bits of a 32-bit register. The instruction sign-extends the upper bits of the destination register. This instruction supports only D-registers.

Flags Affected

The following flags are affected by the Move Byte – Sign-Extended instruction.

- \emph{AZ} is set if result is zero; cleared if nonzero.
- \emph{AN} is set if result is negative; cleared if non-negative.
- \emph{AC0} is cleared.
Instruction Overview

- $V$ is cleared.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer $\text{ASTAT}$ flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction cannot be issued in parallel with any other instructions.

Example

\[
\begin{align*}
    r7 &= r2.b ; \\
    r7 &= r2.b(x) ;
\end{align*}
\]

Also See

Move Byte – Zero-Extended

Special Applications

None
10 STACK CONTROL

Instruction Summary

- “--SP (Push)” on page 10-2
- “--SP (Push Multiple)” on page 10-5
- “SP++ (Pop)” on page 10-8
- “SP++ (Pop Multiple)” on page 10-12
- “LINK, UNLINK” on page 10-17

Instruction Overview

This chapter discusses the instructions that control the stack. Users can take advantage of these instructions to save the contents of single or multiple registers to the stack or to control the stack frame space on the stack and the Frame Pointer (FP) for that space.
Instruction Overview

--SP (Push)

General Form

[ -- SP ] = src_reg

Syntax

[ -- SP ] = allreg ;       /* predecrement SP (a) */

Syntax Terminology

allreg: R7-0, P5-0, FP, I3-0, M3-0, B3-0, L3-0, A0.X, A0.W, A1.X, A1.W,
ASTAT, RETS, RETI, RETX, RETN, RETE, LC0, LC1, LT0, LT1, LB0, LB1, CYCLES,
CYCLES2, EMUDAT, USP, SEQSTAT, and SYSCFG

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Push instruction stores the contents of a specified register in the stack. The instruction pre-decrements the Stack Pointer to the next available location in the stack first. Push and Push Multiple are the only instructions that perform pre-modify functions.

The stack grows down from high memory to low memory. Consequently, the decrement operation is used for pushing, and the increment operation is used for popping values. The Stack Pointer always points to the last used location. Therefore, the effective address of the push is SP-4.

The following illustration shows what the stack would look like when a series of pushes occur.
Stack Control

higher memory

\[
\begin{array}{l}
\text{P5} \quad \text{[-sp]=p5 ;} \\
\text{P1} \quad \text{[-sp]=p1 ;} \\
\text{R3} \quad \leftarrow \text{SP} \text{[-sp]=r3 ;} \\
\vdots
\end{array}
\]

lower memory

The Stack Pointer must already be 32-bit aligned to use this instruction. If an unaligned memory access occurs, an exception is generated and the instruction aborts.

Push/pop on RETS has no effect on the interrupt system.

Push/pop on RETI does affect the interrupt system.

Pushing RETI enables the interrupt system, whereas popping RETI disables the interrupt system.

Pushing the Stack Pointer is meaningless since it cannot be retrieved from the stack. Using the Stack Pointer as the destination of a pop instruction (as in the fictional instruction `SP=[SP++]`) causes an undefined instruction exception. (Refer to “Register Names” on page 1-13 for more information.)

Flags Affected

None

Required Mode

User & Supervisor for most cases.

Explicit accesses to USP, SEQSTAT, SYSCFG, RETI, RETX, RETN, and RETE requires Supervisor mode. A protection violation exception results if any of these registers are explicitly accessed from User mode.
Instruction Overview

Parallel Issue

This instruction cannot be issued in parallel with other instructions.

Example

\[
[\quad sp \quad] = r0 \\
[\quad sp \quad] = r1 \\
[\quad sp \quad] = p0 \\
[\quad sp \quad] = i0 
\]

Also See

--SP (Push Multiple), SP++ (Pop)

Special Applications

None
Stack Control

--SP (Push Multiple)

General Form

\[[ -- SP ] = (src\_reg\_range)\]

Syntax

\[[ -- SP ] = ( R7 : Dreglim , P5 : Preglim ) ; /* Dregs and indexed Pregs (a) */\]
\[[ -- SP ] = ( R7 : Dreglim ) ; /* Dregs, only (a) */\]
\[[ -- SP ] = ( P5 : Preglim ) ; /* indexed Pregs, only (a) */\]

Syntax Terminology

- **Dreglim**: any number in the range 7 through 0
- **Preglim**: any number in the range 5 through 0

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Push Multiple instruction saves the contents of multiple data and/or Pointer registers to the stack. The range of registers to be saved always includes the highest index register (R7 and/or P5) plus any contiguous lower index registers specified by the user down to and including R0 and/or P0. Push and Push Multiple are the only instructions that perform pre-modify functions.

The instructions start by saving the register having the lowest index then advance to the register with the highest index. The index of the first register saved in the stack is specified by the user in the instruction syntax.

Data registers are pushed before Pointer registers if both are specified in one instruction.
Instruction Overview

The instruction pre-decrements the Stack Pointer to the next available location in the stack first.

The stack grows down from high memory to low memory, therefore the decrement operation is the same used for pushing, and the increment operation is used for popping values. The Stack Pointer always points to the last used location. Therefore, the effective address of the push is $SP-4$.

The following illustration shows what the stack would look like when a push multiple occurs.

higher memory

\[
\begin{array}{c}
\text{P3} \\
\text{P4} \\
\text{P5} \\
\vdots
\end{array} \quad [-sp]=(p5:3) ;
\]

lower memory

Because the lowest-indexed registers are saved first, it is advisable that a runtime system be defined to have its compiler scratch registers as the lowest-indexed registers. For instance, data registers $R0, P0$ would be the return value registers for a simple calling convention.

Although this instruction takes a variable amount of time to complete depending on the number of registers to be saved, it reduces compiled code size.

This instruction is not interruptible. Interrupts asserted after the first issued stack write operation are appended until all the writes complete. However, exceptions that occur while this instruction is executing cause it to abort gracefully. For example, a load/store operation might cause a protection violation while Push Multiple is executing. The $SP$ is reset to its value before the execution of this instruction. This measure ensures that
the instruction can be restarted after the exception. Note that when a Push Multiple operation is aborted due to an exception, the memory state is changed by the stores that have already completed before the exception.

The Stack Pointer must already be 32-bit aligned to use this instruction. If an unaligned memory access occurs, an exception is generated and the instruction aborts, as described above.

Only pointer registers P5–0 can be operands for this instruction; SP and FP cannot. All data registers R7–0 can be operands for this instruction.

Flags Affected

None

Required Mode

User & Supervisor

Parallel Issue

This instruction cannot be issued in parallel with other instructions.

Example

\[ \[ \quad \text{sp} \quad \] = (r7:5, p5:0) ; \quad /* \text{D-registers R4:0 excluded} */ \]
\[ \[ \quad \text{sp} \quad \] = (r7:2) ; \quad /* \text{R1:0 excluded} */ \]
\[ \[ \quad \text{sp} \quad \] = (p5:4) ; \quad /* \text{P3:0 excluded} */ \]

Also See

--SP (Push), SP++ (Pop), SP++ (Pop Multiple)

Special Applications

None
Instruction Overview

SP++ (Pop)

General Form

\[ \text{dest	extunderscore reg} = [\ SP \ ++ \ ] \]

Syntax

\[
\begin{align*}
\text{mostreg} = [\ SP \ ++ \ ] : & \quad /* \text{post-increment SP; does not apply to Data Registers and Pointer Registers (a)} */ \\
\text{Dreg} = [\ SP \ ++ \ ] : & \quad /* \text{Load Data Register instruction (repeated here for user convenience) (a)} */ \\
\text{Preg} = [\ SP \ ++ \ ] : & \quad /* \text{Load Pointer Register instruction (repeated here for user convenience) (a)} */
\end{align*}
\]

Syntax Terminology

\[
\begin{align*}
\text{mostreg:} & \quad \text{I3–0, M3–0, B3–0, L3–0, A0.X, A0.W, A1.X, A1.W, ASTAT, RETS, RETI, RETX, RETN, RETE, LCO, LC1, LTO, LT1, LBO, LBI, USP, SEQSTAT, and SYSCFG} \\
\text{Dreg:} & \quad \text{R7–0} \\
\text{Preg:} & \quad \text{P5–0, FP}
\end{align*}
\]

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Pop instruction loads the contents of the stack indexed by the current Stack Pointer into a specified register. The instruction post-increments the Stack Pointer to the next occupied location in the stack before concluding.
Stack Control

The stack grows down from high memory to low memory, therefore the decrement operation is used for pushing, and the increment operation is used for popping values. The Stack Pointer always points to the last used location. When a pop operation is issued, the value pointed to by the Stack Pointer is transferred and the $SP$ is replaced by $SP+4$.

The illustration below shows what the stack would look like when a pop such as $R3 = [SP++]$ occurs.

higher memory

```
Word0
Word1
Word2 <-------- SP
...
```

BEGINNING STATE

lower memory

higher memory

```
Word0
Word1
Word2 <-------- SP
...
```

LOAD REGISTER R3 FROM STACK

```
Word0
Word1
Word2 <-------- SP
...
```

R3 = Word2

lower memory

higher memory

```
Word0
Word1
Word2 <-------- SP
...
```

POST-INCREMENT STACK POINTER

```
Word0
Word1
Word2 <-------- SP
...
```

lower memory

The value just popped remains on the stack until another push instruction overwrites it.
Instruction Overview

Of course, the usual intent for Pop and these specific Load Register instructions is to recover register values that were previously pushed onto the stack. The user must exercise programming discipline to restore the stack values back to their intended registers from the first-in, last-out structure of the stack. Pop or load exactly the same registers that were pushed onto the stack, but pop them in the opposite order.

The Stack Pointer must already be 32-bit aligned to use this instruction. If an unaligned memory access occurs, an exception is generated and the instruction aborts.

A value cannot be popped off the stack directly into the Stack Pointer. $SP = [SP++]$ is an invalid instruction. Refer to “Register Names” on page 1-13 for more information.

Flags Affected

The $ASTAT = [SP++]$ version of this instruction explicitly affects arithmetic flags.

Flags are not affected by other versions of this instruction.

Required Mode

User & Supervisor for most cases

Explicit access to $USP, SEQSTAT, SYSCFG, RETI, RETX, RETN, and RETE$ requires Supervisor mode. A protection violation exception results if any of these registers are explicitly accessed from User mode.

Parallel Issue

The 16-bit versions of the Load Data Register and Load Pointer Register instructions can be issued in parallel with specific other instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

The Pop instruction cannot be issued in parallel with other instructions.
Stack Control

Example

\[ r0 = [sp++] ; /* Load Data Register instruction */ \]
\[ p4 = [sp++] ; /* Load Pointer Register instruction */ \]
\[ i1 = [sp++] ; /* Pop instruction */ \]
\[ reti = [sp++] ; /* Pop instruction: supervisor mode required */ \]

Also See

Load Pointer Register, Load Data Register, --SP (Push), --SP (Push Multiple), SP++ (Pop Multiple)

Special Applications

None
Instruction Overview

SP++ (Pop Multiple)

General Form

\[(\text{dest\_reg\_range}) = [\text{SP}++]\]

Syntax

\[
(\text{R7 : Dreglim, P5 : Preglim}) = [\text{SP}++] ; /* Dregs and indexed Pregs (a) */
\]
\[
(\text{R7 : Dreglim}) = [\text{SP}++] ; /* Dregs, only (a) */
\]
\[
(\text{P5 : Preglim}) = [\text{SP}++] ; /* indexed Pregs, only (a) */
\]

Syntax Terminology

*\text{Dreglim}: any number in the range 7 through 0*

*\text{Preglim}: any number in the range 5 through 0*

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Pop Multiple instruction restores the contents of multiple data and/or Pointer registers from the stack. The range of registers to be restored always includes the highest index register (\text{R7 and/or P5}) plus any contiguous lower index registers specified by the user down to and including \text{R0 and/or P0}.

The instructions start by restoring the register having the highest index then descend to the register with the lowest index. The index of the last register restored from the stack is specified by the user in the instruction syntax. Pointer registers are popped before Data registers, if both are specified in the same instruction.
The instruction post-increments the Stack Pointer to the next occupied location in the stack before concluding.

The stack grows down from high memory to low memory, therefore the decrement operation is used for pushing, and the increment operation is used for popping values. The Stack Pointer always points to the last used location. When a pop operation is issued, the value pointed to by the Stack Pointer is transferred and the \( SP \) is replaced by \( SP+4 \).

The following graphic shows what the stack would look like when a Pop Multiple such as \((R7:5) = [SP++]\) occurs.

higher memory

\[
\begin{array}{c}
\text{Word0} \\
\text{Word1} \\
\text{Word2} \\
\text{Word3} \\
\ldots \\
\end{array}
\begin{array}{c}
\text{BEGINNING STATE} \\
\text{SP} \\
\end{array}
\]

lower memory

higher memory

\[
\begin{array}{c}
\text{R3} \\
\text{R4} \\
\text{R6} \\
\text{R7} \\
\ldots \\
\end{array}
\begin{array}{c}
\text{LOAD REGISTER R7 FROM STACK} \\
\text{SP} \\
\text{R7 = Word3} \\
\end{array}
\]

lower memory
Instruction Overview

higher memory

\[
\begin{align*}
R4 & \\
R5 & \\
R6 & \leftarrow SP \quad \Rightarrow R6 = \text{Word2} \\
R7 & \\
& \ldots
\end{align*}
\]

lower memory

higher memory

\[
\begin{align*}
& \ldots \\
R5 & \leftarrow SP \quad \Rightarrow R5 = \text{Word1} \\
R6 & \\
R7 & \ldots
\end{align*}
\]

lower memory

higher memory

\[
\begin{align*}
& \ldots \\
& \ldots \quad \text{POST-INCREMENT STACK POINTER} \\
\text{Word0} & \leftarrow SP \\
\text{Word1} & \\
\text{Word2} & \\
\end{align*}
\]

lower memory

The value(s) just popped remain on the stack until another push instruction overwrites it.

Of course, the usual intent for Pop Multiple is to recover register values that were previously pushed onto the stack. The user must exercise programming discipline to restore the stack values back to their intended
Stack Control

registers from the first-in, last-out structure of the stack. Pop exactly the same registers that were pushed onto the stack, but pop them in the opposite order.

Although this instruction takes a variable amount of time to complete depending on the number of registers to be saved, it reduces compiled code size.

This instruction is not interruptible. Interrupts asserted after the first issued stack read operation are appended until all the reads complete. However, exceptions that occur while this instruction is executing cause it to abort gracefully. For example, a load/store operation might cause a protection violation while Pop Multiple is executing. In that case, \( SP \) is reset to its original value prior to the execution of this instruction. This measure ensures that the instruction can be restarted after the exception.

Note that when a Pop Multiple operation aborts due to an exception, some of the destination registers are changed as a result of loads that have already completed before the exception.

The Stack Pointer must already be 32-bit aligned to use this instruction. If an unaligned memory access occurs, an exception is generated and the instruction aborts, as described above.

Only Pointer registers \( P5-0 \) can be operands for this instruction; \( SP \) and \( FP \) cannot. All data registers \( R7-0 \) can be operands for this instruction.

Flags Affected

None

Required Mode

User & Supervisor
Instruction Overview

Parallel Issue

This instruction cannot be issued in parallel with other instructions.

Example

```
(p5:4) = [ sp ++ ];   /* P3 through P0 excluded */
(r7:2) = [ sp ++ ];   /* R1 through R0 excluded */
(r7:5, p5:0) = [ sp ++ ];  /* D-registers R4 through R0
optionally excluded */
```

Also See

--SP (Push), --SP (Push Multiple), SP++ (Pop)

Special Applications

None
Stack Control

**LINK, UNLINK**

**General Form**

```
LINK, UNLINK
```

**Syntax**

```
LINK uimm18m4;  /* allocate a stack frame of specified size (b) */
UNLINK;  /* de-allocate the stack frame (b) */
```

**Syntax Terminology**

`uimm18m4`: 18-bit unsigned field that must be a multiple of 4, with a range of 8 through 262,152 bytes (0x00000 through 0x3FFFC)

**Instruction Length**

In the syntax, comment (b) identifies 32-bit instruction length.

**Functional Description**

The Linkage instruction controls the stack frame space on the stack and the Frame Pointer (FP) for that space. **LINK** allocates the space and **UNLINK** de-allocates the space.

**LINK** saves the current RETS and FP registers to the stack, loads the FP register with the new frame address, then decrements the SP by the user-supplied frame size value.

Typical applications follow the **LINK** instruction with a Push Multiple instruction to save pointer and data registers to the stack.
Instruction Overview

The user-supplied argument for \texttt{LINK} determines the size of the allocated stack frame. \texttt{LINK} always saves \texttt{RETS} and \texttt{FP} on the stack, so the minimum frame size is 2 words when the argument is zero. The maximum stack frame size is $2^{18} + 8 = 262152$ bytes in 4-byte increments.

\texttt{UNLINK} performs the reciprocal of \texttt{LINK}, de-allocating the frame space by moving the current value of \texttt{FP} into \texttt{SP} and restoring previous values into \texttt{FP} and \texttt{RETS} from the stack.

The \texttt{UNLINK} instruction typically follows a \texttt{Pop Multiple} instruction that restores pointer and data registers previously saved to the stack.

The frame values remain on the stack until a subsequent \texttt{Push}, \texttt{Push Multiple} or \texttt{LINK} operation overwrites them.

Of course, \texttt{FP} must not be modified by user code between \texttt{LINK} and \texttt{UNLINK} to preserve stack integrity.

Neither \texttt{LINK} nor \texttt{UNLINK} can be interrupted. However, exceptions that occur while either of these instructions is executing cause the instruction to abort. For example, a load/store operation might cause a protection violation while \texttt{LINK} is executing. In that case, \texttt{SP} and \texttt{FP} are reset to their original values prior to the execution of this instruction. This measure ensures that the instruction can be restarted after the exception.

Note that when a \texttt{LINK} operation aborts due to an exception, the stack memory may already be changed due to stores that have already completed before the exception. Likewise, an aborted \texttt{UNLINK} operation may leave the \texttt{FP} and \texttt{RETS} registers changed because of a load that has already completed before the interruption.

The illustrations below show the stack contents after executing a \texttt{LINK} instruction followed by a \texttt{Push Multiple} instruction.
Stack Control

higher memory

\[
\begin{array}{|c|}
\hline
\ldots \\
\ldots \\
\text{AFTER LINK EXECUTES} \\
\text{Saved RETS} \leftarrow \text{FP} \\
\text{Prior FP} \leftarrow \text{FP} \\
\text{Allocated words for local subroutine variables} \leftarrow \text{SP} = \text{FP} + \text{frame_size} \\
\ldots \\
\hline
\end{array}
\]

lower memory

higher memory

\[
\begin{array}{|c|}
\hline
\ldots \\
\ldots \\
\text{AFTER A PUSH MULTIPLE EXECUTES} \\
\text{Saved RETS} \\
\text{Prior FP} \leftarrow \text{FP} \\
\text{Allocated words for local subroutine variables} \\
\text{R0} \\
\text{R1} \\
\vdots \\
\text{R7} \\
\text{P0} \\
\vdots \\
\text{P5} \leftarrow \text{SP} \\
\hline
\end{array}
\]

lower memory

The Stack Pointer must already be 32-bit aligned to use this instruction. If an unaligned memory access occurs, an exception is generated and the instruction aborts, as described above.
Instruction Overview

Flags Affected

None

Required Mode

User & Supervisor

Parallel Issue

This instruction cannot be issued in parallel with other instructions.

Example

```
link 8 ;   /* establish frame with 8 words allocated for local
variables */
[ -- sp ] = (r7:0, p5:0) ;   /* save D- and P-registers */
(r7:0, p5:0) = [ sp ++ ] ;   /* restore D- and P-registers */
unlink ;   /* close the frame */
```

Also See

--SP (Push Multiple) SP++ (Pop Multiple)

Special Applications

The Linkage instruction is used to set up and tear down stack frames for a high-level language like C.
11 CONTROL CODE BIT MANAGEMENT

Instruction Summary

- “Compare Data Register” on page 11-2
- “Compare Pointer” on page 11-6
- “Compare Accumulator” on page 11-9
- “Move CC” on page 11-12
- “Negate CC” on page 11-15

Instruction Overview

This chapter discusses the instructions that affect the Control Code (cc) bit in the ASTAT register. Users can take advantage of these instructions to set the cc bit based on a comparison of values from two registers, pointers, or accumulators. In addition, these instructions can move the status of the cc bit to and from a data register or arithmetic status bit, or they can negate the status of the cc bit.
Compare Data Register

General Form

CC = operand_1 == operand_2
CC = operand_1 < operand_2
CC = operand_1 <= operand_2
CC = operand_1 < operand_2 (IU)
CC = operand_1 <= operand_2 (IU)

Syntax

CC = Dreg == Dreg ; /* equal, register, signed (a) */
CC = Dreg == imm3 ; /* equal, immediate, signed (a) */
CC = Dreg < Dreg ; /* less than, register, signed (a) */
CC = Dreg < imm3 ; /* less than, immediate, signed (a) */
CC = Dreg <= Dreg ; /* less than or equal, register, signed (a) */
CC = Dreg <= imm3 ; /* less than or equal, immediate, signed (a) */
CC = Dreg < Dreg (IU) ; /* less than, register, unsigned (a) */
CC = Dreg < uimm3 (IU) ; /* less than, immediate, unsigned (a) */
CC = Dreg <= Dreg (IU) ; /* less than or equal, register, unsigned (a) */
CC = Dreg <= uimm3 (IU) ; /* less than or equal, immediate unsigned (a) */

Syntax Terminology

Dreg: R7-0
imm3: 3-bit signed field, with a range of –4 through 3
uimm3: 3-bit unsigned field, with a range of 0 through 7
Control Code Bit Management

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Compare Data Register instruction sets the Control Code (CC) bit based on a comparison of two values. The input operands are D-registers.

The compare operations are nondestructive on the input operands and affect only the CC bit and the flags. The value of the CC bit determines all subsequent conditional branching.

The various forms of the Compare Data Register instruction perform 32-bit signed compare operations on the input operands or an unsigned compare operation, if the (IU) optional mode is appended. The compare operations perform a subtraction and discard the result of the subtraction without affecting user registers. The compare operation that you specify determines the value of the CC bit.

Flags Affected

The Compare Data Register instruction uses the values shown in Table 11-1 in signed and unsigned compare operations.

Table 11-1. Compare Data Register Values

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal</td>
<td>AZ=1</td>
<td>n/a</td>
</tr>
<tr>
<td>Less than</td>
<td>AN=1</td>
<td>AC0=0</td>
</tr>
<tr>
<td>Less than or equal</td>
<td>AN or AZ=1</td>
<td>AC0=0 or AZ=1</td>
</tr>
</tbody>
</table>
Instruction Overview

The following flags are affected by the Compare Data Register instruction.

- CC is set if the test condition is true; cleared if false.
- AZ is set if result is zero; cleared if nonzero.
- AN is set if result is negative; cleared if non-negative.
- ACO is set if result generated a carry; cleared if no carry.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode
User & Supervisor

Parallel Issue
This instruction cannot be issued in parallel with other instructions.

Example

cc = r3 == r2 ;
c = r7 == 1 ;
   /* If r0 = 0x8FFFFF and r3 = 0x00000001, then the signed
      operation . . . */
c = r0 < r3 ;
   /* . . . produces cc = 1, because r0 is treated as a negative
      value */
c = r2 < -4 ;
c = r6 <= r1 ;
c = r4 <= 3 ;
Control Code Bit Management

/* If r0 = 0x8FFF FFFF and r3 = 0x0000 0001, then the unsigned operation . . . */
c = r0 < r3 (iu);
/* . . . produces CC = 0, because r0 is treated as a large unsigned value */
c = r1 < 0x7 (iu);
c = r2 <= r0 (iu);
c = r3 <= 2 (iu);

Also See

Compare Pointer, Compare Accumulator, IF CC JUMP, BITTST

Special Applications

None
Instruction Overview

Compare Pointer

General Form

\[ CC = \text{operand}_1 == \text{operand}_2 \]
\[ CC = \text{operand}_1 < \text{operand}_2 \]
\[ CC = \text{operand}_1 <= \text{operand}_2 \]
\[ CC = \text{operand}_1 < \text{operand}_2 \text{ (IU)} \]
\[ CC = \text{operand}_1 <= \text{operand}_2 \text{ (IU)} \]

Syntax

\[ CC = \text{Preg} == \text{Preg} ; \quad /* \text{equal, register, signed (a)} */ \]
\[ CC = \text{Preg} == \text{imm3} ; \quad /* \text{equal, immediate, signed (a)} */ \]
\[ CC = \text{Preg} < \text{Preg} ; \quad /* \text{less than, register, signed (a)} */ \]
\[ CC = \text{Preg} < \text{imm3} ; \quad /* \text{less than, immediate, signed (a)} */ \]
\[ CC = \text{Preg} <= \text{Preg} ; \quad /* \text{less than or equal, register, signed (a)} */ \]
\[ CC = \text{Preg} <= \text{imm3} ; \quad /* \text{less than or equal, immediate, signed (a)} */ \]
\[ CC = \text{Preg} < \text{Preg} \text{ (IU)} ; \quad /* \text{less than, register, unsigned (a)} */ \]
\[ CC = \text{Preg} < \text{uimm3} \text{ (IU)} ; \quad /* \text{less than, immediate, unsigned (a)} */ \]
\[ CC = \text{Preg} <= \text{Preg} \text{ (IU)} ; \quad /* \text{less than or equal, register, unsigned (a)} */ \]
\[ CC = \text{Preg} <= \text{uimm3} \text{ (IU)} ; \quad /* \text{less than or equal, immediate unsigned (a)} */ \]

Syntax Terminology

Preg: P5-0, SP, FP

imm3: 3-bit signed field, with a range of –4 through 3

uimm3: 3-bit unsigned field, with a range of 0 through 7
Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Compare Pointer instruction sets the Control Code (CC) bit based on a comparison of two values. The input operands are P-registers.

The compare operations are nondestructive on the input operands and affect only the CC bit and the flags. The value of the CC bit determines all subsequent conditional branching.

The various forms of the Compare Pointer instruction perform 32-bit signed compare operations on the input operands or an unsigned compare operation, if the (IU) optional mode is appended. The compare operations perform a subtraction and discard the result of the subtraction without affecting user registers. The compare operation that you specify determines the value of the CC bit.

Flags Affected

- CC is set if the test condition is true; cleared if false.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction cannot be issued in parallel with other instructions.
Instruction Overview

Example

\[
\begin{align*}
cc &= p3 == p2 ; \\
cc &= p0 == 1 ; \\
cc &= p0 < p3 ; \\
cc &= p2 < -4 ; \\
cc &= p1 <= p0 ; \\
cc &= p4 <= 3 ; \\
cc &= p5 < p3 (iu) ; \\
cc &= p1 < 0x7 (iu) ; \\
cc &= p2 <= p0 (iu) ; \\
cc &= p3 <= 2 (iu) ;
\end{align*}
\]

Also See

Compare Data Register, Compare Accumulator, IF CC JUMP

Special Applications

None
Control Code Bit Management

Compare Accumulator

General Form

\[ CC = A0 == A1 \]
\[ CC = A0 < A1 \]
\[ CC = A0 <= A1 \]

Syntax

\[ CC = A0 == A1 ; /* equal, signed (a) */ \]
\[ CC = A0 < A1 ; /* less than, Accumulator, signed (a) */ \]
\[ CC = A0 <= A1 ; /* less than or equal, Accumulator, signed (a) */ \]

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Compare Accumulator instruction sets the Control Code (CC) bit based on a comparison of two values. The input operands are Accumulators.

These instructions perform 40-bit signed compare operations on the Accumulators. The compare operations perform a subtraction and discard the result of the subtraction without affecting user registers. The compare operation that you specify determines the value of the CC bit.

No unsigned compare operations or immediate compare operations are performed for the Accumulators.

The compare operations are nondestructive on the input operands, and affect only the CC bit and the flags. All subsequent conditional branching is based on the value of the CC bit.
Instruction Overview

Flags Affected

The Compare Accumulator instruction uses the values shown in Table 11-2 in compare operations.

Table 11-2. Compare Accumulator Instruction Values

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal</td>
<td>AZ=1</td>
</tr>
<tr>
<td>Less than</td>
<td>AN=1</td>
</tr>
<tr>
<td>Less than or equal</td>
<td>AN or AZ=1</td>
</tr>
</tbody>
</table>

The following arithmetic status bits reside in the ASTAT register.

- **CC** is set if the test condition is true; cleared if false.
- **AZ** is set if result is zero; cleared if nonzero.
- **AN** is set if result is negative; cleared if non-negative.
- **AC0** is set if result generated a carry; cleared if no carry.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction cannot be issued in parallel with other instructions.
Control Code Bit Management

Example

\[
\text{cc} = a0 == a1 ; \\
\text{cc} = a0 < a1 ; \\
\text{cc} = a0 <= a1 ;
\]

Also See

Compare Pointer, Compare Data Register, IF CC JUMP

Special Applications

None
Instruction Overview

Move CC

General Form

\[
\begin{align*}
\text{dest} &= \text{CC} \\
\text{dest} &= \text{CC} \\
\text{dest} &= \text{CC} \\
\text{dest} &= \text{CC} \\
\text{CC} &= \text{source} \\
\text{CC} &= \text{source} \\
\text{CC} &= \text{source} \\
\text{CC} &= \text{source}
\end{align*}
\]

Syntax

\[
\begin{align*}
\text{Dreg} &= \text{CC} ; /* \text{CC} \text{ into 32-bit data register, zero-extended (a)} */ \\
\text{statbit} &= \text{CC} ; /* \text{status bit equals CC (a)} */ \\
\text{statbit} &= \text{CC} ; /* \text{status bit equals status bit OR CC (a)} */ \\
\text{statbit} &= \text{CC} ; /* \text{status bit equals status bit AND CC (a)} */ \\
\text{statbit} &= \text{CC} ; /* \text{status bit equals status bit XOR CC (a)} */ \\
\text{CC} &= \text{Dreg} ; /* \text{CC set if the register is non-zero (a)} */ \\
\text{CC} &= \text{statbit} ; /* \text{CC equals status bit (a)} */ \\
\text{CC} &= \text{statbit} ; /* \text{CC equals CC OR status bit (a)} */ \\
\text{CC} &= \text{statbit} ; /* \text{CC equals CC AND status bit (a)} */ \\
\text{CC} &= \text{statbit} ; /* \text{CC equals CC XOR status bit (a)} */
\end{align*}
\]

Syntax Terminology

\[
\begin{align*}
\text{Dreg: R7-0} \\
\text{statbit: AZ, AN, AC0, AC1, V, VS, AVO, AVOS, AV1, AV1S, AQ}
\end{align*}
\]

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.
Control Code Bit Management

Functional Description

The Move $cc$ instruction moves the status of the Control Code ($cc$) bit to and from a data register or arithmetic status bit.

When copying the $cc$ bit into a 32-bit register, the operation moves the $cc$ bit into the least significant bit of the register, zero-extended to 32 bits. The two cases are as follows.

- If $cc = 0$, Dreg becomes 0x00000000.
- If $cc = 1$, Dreg becomes 0x00000001.

When copying a data register to the $cc$ bit, the operation sets the $cc$ bit to 1 if any bit in the source data register is set; that is, if the register is non-zero. Otherwise, the operation clears the $cc$ bit.

Some versions of this instruction logically set or clear an arithmetic status bit based on the status of the Control Code.

The use of the $cc$ bit as source and destination in the same instruction is disallowed. See the Negate $cc$ instruction to change $cc$ based solely on its own value.

Flags Affected

- The Move $cc$ instruction affects flags $cc$, $az$, $an$, $aco$, $ac1$, $v$, $vs$, $avo$, $avos$, $av1$, $av1s$, $aq$, according to the status bit and syntax used, as described in “Syntax” on page 11-12.
- All other flags not explicitly specified by the syntax are unaffected.

The ADSP-BF535 processor has fewer $astat$ flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.
Instruction Overview

Required Mode

User & Supervisor

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Parallel Issue

This instruction cannot be issued in parallel with other instructions.

Example

\[
\begin{align*}
& r0 = cc ; \\
& az = cc ; \\
& an |= cc ; \\
& ac0 &= cc ; \\
& av0 ^= cc ; \\
& cc = r4 ; \\
& cc = av1 ; \\
& cc |= aq ; \\
& cc &= an ; \\
& cc ^= ac1 ;
\end{align*}
\]

Also See

Negate CC

Special Applications

None
Control Code Bit Management

**Negate CC**

**General Form**

\[ CC = ! CC \]

**Syntax**

\[ CC = ! CC ; /* (a) */ \]

**Instruction Length**

In the syntax, comment (a) identifies 16-bit instruction length.

**Functional Description**

The Negate CC instruction inverts the logical state of CC.

**Flags Affected**

- CC is toggled from its previous value by the Negate CC instruction.
- All other flags are unaffected.

ℹ️ The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

**Required Mode**

User & Supervisor

**Parallel Issue**

This instruction cannot be issued in parallel with other instructions.
Instruction Overview

Example

```c
cc = ! cc ;
```

Also See

- Move CC

Special Applications

- None
12 LOGICAL OPERATIONS

Instruction Summary

- “& (AND)” on page 12-2
- “~ (NOT One’s-Complement)” on page 12-4
- “| (OR)” on page 12-6
- “^ (Exclusive-OR)” on page 12-8
- “BXORSHIFT, BXOR” on page 12-10

Instruction Overview

This chapter discusses the instructions that specify logical operations. Users can take advantage of these instructions to perform logical AND, NOT, OR, exclusive-OR, and bit-wise exclusive-OR (BXORSHIFT) operations.
Instruction Overview

& (AND)

General Form

dest_reg = src_reg_0 & src_reg_1

Syntax

Dreg = Dreg & Dreg ;  /* (a) */

Syntax Terminology

Dreg: R7-0

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The AND instruction performs a 32-bit, bit-wise logical AND operation on the two source registers and stores the results into the dest_reg.

The instruction does not implicitly modify the source registers. The dest_reg and one src_reg can be the same D-register. This would explicitly modifies the src_reg.

Flags Affected

The AND instruction affects flags as follows.

- AZ is set if the final result is zero, cleared if nonzero.
- AN is set if the result is negative, cleared if non-negative.
Logical Operations

- \( ACO \) and \( V \) are cleared.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

**Required Mode**

User & Supervisor

**Parallel Issue**

This instruction cannot be issued in parallel with other instructions.

**Example**

\[
\text{r4 = r4 & r3 ;}
\]

**Also See**

\( | \) (OR)

**Special Applications**

None
Instruction Overview

~ (NOT One’s-Complement)

General Form

\[
dest\_reg = \sim src\_reg
\]

Syntax

\[
Dreg = \sim Dreg ; /* (a)*/
\]

Syntax Terminology

\[
Dreg: R7-0
\]

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The NOT One’s-Complement instruction toggles every bit in the 32-bit register.

The instruction does not implicitly modify the \textit{src\_reg}. The \textit{dest\_reg} and \textit{src\_reg} can be the same D-register. Using the same D-register as the \textit{dest\_reg} and \textit{src\_reg} would explicitly modify the \textit{src\_reg}.

Flags Affected

The NOT One’s-Complement instruction affects flags as follows.

- \( AZ \) is set if the final result is zero, cleared if nonzero.
- \( AN \) is set if the result is negative, cleared if non-negative.
Logical Operations

- \( AC = 0 \) and \( V \) are cleared.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction cannot be issued in parallel with other instructions.

Example

\[ r_3 = \sim r_4 ; \]

Also See

Negate (Two’s-Complement)

Special Applications

None
Instruction Overview

| (OR)

General Form

\[ \text{dest	extunderscore reg} = \text{src	extunderscore reg}_0 \mid \text{src	extunderscore reg}_1 \]

Syntax

\[ D\text{reg} = D\text{reg} \mid D\text{reg} ; \quad /* \text{(a)} */ \]

Syntax Terminology

\[ D\text{reg}: R7-0 \]

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The OR instruction performs a 32-bit, bit-wise logical OR operation on the two source registers and stores the results into the \text{dest	extunderscore reg}.

The instruction does not implicitly modify the source registers. The \text{dest	extunderscore reg} and one \text{src	extunderscore reg} can be the same D-register. This would explicitly modifies the \text{src	extunderscore reg}.

Flags Affected

The OR instruction affects flags as follows.

- \text{AZ} is set if the final result is zero, cleared if nonzero.
- \text{AN} is set if the result is negative, cleared if non-negative.
Logical Operations

- $AC0$ and $V$ are cleared.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer $ASTAT$ flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction cannot be issued in parallel with other instructions.

Example

$r4 = r4 | r3$;

Also See

$^(\text{Exclusive-OR}),$ BXORSHIFT, BXOR

Special Applications
None
Instruction Overview

^ (Exclusive-OR)

General Form

\[ \text{dest\_reg} = \text{src\_reg\_0} \ ^ \ ^ \ \text{src\_reg\_1} \]

Syntax

\[ \text{Dreg} = \text{Dreg} \ ^ \ ^ \ \text{Dreg} ; \quad /* (a) */ \]

Syntax Terminology

\[ \text{Dreg: R7-0} \]

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Exclusive-OR (XOR) instruction performs a 32-bit, bit-wise logical exclusive OR operation on the two source registers and loads the results into the \text{dest\_reg}.

The XOR instruction does not implicitly modify source registers. The \text{dest\_reg} and one \text{src\_reg} can be the same D-register. This would explicitly modifies the \text{src\_reg}.

Flags Affected

The XOR instruction affects flags as follows.

- \text{AZ} is set if the final result is zero, cleared if nonzero.
- \text{AN} is set if the result is negative, cleared if non-negative.
Logical Operations

- AC0 and V are cleared.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction cannot be issued in parallel with other instructions.

Example

```
r4 = r4 ^ r3 ;
```

Also See

| (OR), BXORSHIFT, BXOR

Special Applications

None
Instruction Overview

BXORSHIFT, BXOR

General Form

dest_reg = CC = BXORSHIFT ( A0, src_reg )
dest_reg = CC = BXOR ( A0, src_reg )
dest_reg = CC = BXOR ( A0, A1, CC )
A0 = BXORSHIFT ( A0, A1, CC )

Syntax

LFSR Type I (Without Feedback)

Dreg_lo = CC = BXORSHIFT ( A0, Dreg ) ; /* (b) */
Dreg_lo = CC = BXOR ( A0, Dreg ) ; /* (b) */

LFSR Type I (With Feedback)

Dreg_lo = CC = BXOR ( A0, A1, CC ) ; /* (b) */
A0 = BXORSHIFT ( A0, A1, CC ) ; /* (b) */

Syntax Terminology

Dreg: R7-0
Dreg_lo: R7-0.L

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

Four Bit-Wise Exclusive-OR (BXOR) instructions support two different types of linear feedback shift register (LFSR) implementations.
Logical Operations

The Type I LFSRs (no feedback) applies a 32-bit registered mask to a 40-bit state residing in Accumulator A0, followed by a bit-wise XOR reduction operation. The result is placed in CC and a destination register half.

The Type I LFSRs (with feedback) applies a 40-bit mask in Accumulator A1 to a 40-bit state residing in A0. The result is shifted into A0.

In the following circuits describing the BXOR instruction group, a bit-wise XOR reduction is defined as:

\[
Out = (((((B_0 \oplus B_1) \oplus B_2) \oplus B_3) \oplus \ldots) \oplus B_{n-1})
\]

where B_0 through B_{N-1} represent the N bits that result from masking the contents of Accumulator A0 with the polynomial stored in either A1 or a 32-bit register. The instruction descriptions are shown in Figure 12-1.

![Figure 12-1. Bit-Wise Exclusive-OR Reduction](image)

In the figure above, the bits A0 bit 0 and A0 bit 1 are logically AND’ed with bits D[0] and D[1]. The result from this operation is XOR reduced according to the following formula.

\[
s(D) = (A0[0] \& D[0]) \oplus (A0[1] \& D[1])
\]
Instruction Overview

Modified Type I LFSR (without feedback)

Two instructions support the LFSR with no feedback.

\[ D_{reg\_lo} = CC = BXORSHIFT(A0, dreg) \]
\[ D_{reg\_lo} = CC = BXOR(A0, dreg) \]

In the first instruction the Accumulator \( A0 \) is left-shifted by 1 prior to the XOR reduction. This instruction provides a bit-wise XOR of \( A0 \) logically AND’ed with a \( dreg \). The result of the operation is placed into both the \( CC \) flag and the least significant bit of the destination register. The operation is shown in Figure 12-2.

The upper 15 bits of \( dreg\_lo \) are overwritten with zero, and \( dr[0] = IN \) after the operation.

![Figure 12-2. A0 Left-Shifted by 1 Followed by XOR Reduction](image-url)
Logical Operations

The second instruction in this class performs a bit-wise XOR of $A0$ logically AND’ed with the $dreg$. The output is placed into the least significant bit of the destination register and into the $CC$ bit. The Accumulator $A0$ is not modified by this operation. This operation is illustrated in Figure 12-3.

The upper 15 bits of $dreg_{lo}$ are overwritten with zero, and $dr[0] = IN$ after the operation.

![XOR Reduction Diagram](image)

Figure 12-3. XOR of A0, Logical AND with the D-Register

**Modified Type I LFSR (with feedback)**

Two instructions support the LFSR with feedback.

$$A0 = BXORSHIFT(A0, A1, CC)$$

$$dreg_{lo} = CC = BXOR(A0, A1, CC)$$
Instruction Overview

The first instruction provides a bit-wise XOR of $A_0$ logically AND’ed with $A_1$. The resulting intermediate bit is XOR’ed with the $CC$ flag. The result of the operation is left-shifted into the least significant bit of $A_0$ following the operation. This operation is illustrated in Figure 12-4. The $CC$ bit is not modified by this operation.

![Figure 12-4. XOR of A0 AND A1, Left-Shifted into LSB of A0](image)

The second instruction in this class performs a bit-wise XOR of $A_0$ logically AND’ed with $A_1$. The resulting intermediate bit is XOR’ed with the $CC$ flag. The result of the operation is placed into both the $CC$ flag and the least significant bit of the destination register.

This operation is illustrated in Figure 12-5.

The Accumulator $A_0$ is not modified by this operation. The upper 15 bits of $dreg_{10}$ are overwritten with zero, and $dr[0] = IN$. 

---

12-14 ADSP-BF53x/BF56x Blackfin Processor Programming Reference
Flags Affected

The following flags are affected by the Four Bit-Wise Exclusive-OR instructions.

- **CC** is set or cleared according to the Functional Description for the BXOR and the nonfeedback version of the BXORSHIFT instruction. The feedback version of the BXORSHIFT instruction affects no flags.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor
Instruction Overview

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

\[
\begin{align*}
    r0.l &= cc = bxorshift (a0, r1) ; \\
    r0.l &= cc = bxor (a0, r1) ; \\
    r0.l &= cc = bxor (a0, a1, cc) ; \\
    a0 &= bxorshift (a0, a1, cc) ;
\end{align*}
\]

Also See

None

Special Applications

Linear feedback shift registers (LFSRs) can multiply and divide polynomials and are often used to implement cyclical encoders and decoders.

LFSRs use the set of Bit-Wise XOR instructions to compute bit XOR reduction from a state masked by a polynomial.
13 BIT OPERATIONS

Instruction Summary

- “BITCLR” on page 13-2
- “BITSET” on page 13-4
- “BITTGL” on page 13-6
- “BITTST” on page 13-8
- “DEPOSIT” on page 13-10
- “EXTRACT” on page 13-16
- “BITMUX” on page 13-21
- “ONES (One’s-Population Count)” on page 13-26

Instruction Overview

This chapter discusses the instructions that specify bit operations. Users can take advantage of these instructions to set, clear, toggle, and test bits. They can also merge bit fields and save the result, extract specific bits from a register, merge bit streams, and count the number of ones in a register.
Instruction Overview

BITCLR

General Form

BITCLR ( register, bit_position )

Syntax

BITCLR ( Dreg, uimm5 ) ; /* (a) */

Syntax Terminology

Dreg: R7-0
uimm5: 5-bit unsigned field, with a range of 0 through 31

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Bit Clear instruction clears the bit designated by bit_position in the specified D-register. It does not affect other bits in that register.

The bit_position range of values is 0 through 31, where 0 indicates the LSB, and 31 indicates the MSB of the 32-bit D-register.

Flags Affected

The Bit Clear instruction affects flags as follows.

- AZ is set if result is zero; cleared if nonzero.
- AN is set if result is negative; cleared if non-negative.
- ACO is cleared.
Bit Operations

- $V$ is cleared.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction cannot be issued in parallel with other instructions.

Example

```c
bitclr (r2, 3); /* clear bit 3 (the fourth bit from LSB) in R2 */
```

For example, if $R2$ contains 0xFFFFFFFF before this instruction, it contains 0xFFFFFFFF7 after the instruction.

Also See

BITSET, BITTST, BITTGL

Special Applications

None
**Instruction Overview**

**BITSET**

**General Form**

BITSET ( register, bit_position )

**Syntax**

BITSET ( Dreg, uimm5 ) ; /* (a) */

**Syntax Terminology**

*Dreg*: R7-0

*uimm5*: 5-bit unsigned field, with a range of 0 through 31

**Instruction Length**

In the syntax, comment (a) identifies 16-bit instruction length.

**Functional Description**

The Bit Set instruction sets the bit designated by *bit_position* in the specified D-register. It does not affect other bits in the D-register.

The *bit_position* range of values is 0 through 31, where 0 indicates the LSB, and 31 indicates the MSB of the 32-bit D-register.

**Flags Affected**

The Bit Set instruction affects flags as follows.

- *AZ* is cleared.
- *AN* is set if result is negative; cleared if non-negative.
- *AC0* is cleared.
Bit Operations

- V is cleared.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction cannot be issued in parallel with other instructions.

Example

```c
bitset (r2, 7); /* set bit 7 (the eighth bit from LSB) in R2 */
```

For example, if R2 contains 0x00000000 before this instruction, it contains 0x00000080 after the instruction.

Also See

BITCLR, BITTST, BITTGL

Special Applications

None
Instruction Overview

BITTGL

General Form

BITTGL ( register, bit_position )

Syntax

BITTGL ( Dreg, uimm5 ) ; /* (a) */

Syntax Terminology

Dreg: R7-0

uimm5: 5-bit unsigned field, with a range of 0 through 31

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Bit Toggle instruction inverts the bit designated by bit_position in the specified D-register. The instruction does not affect other bits in the D-register.

The bit_position range of values is 0 through 31, where 0 indicates the LSB, and 31 indicates the MSB of the 32-bit D-register.

Flags Affected

The Bit Toggle instruction affects flags as follows.

- AZ is set if result is zero; cleared if nonzero.
- AN is set if result is negative; cleared if non-negative.
- ACO is cleared.
Bit Operations

- $V$ is cleared.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer $\text{ASTAT}$ flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction cannot be issued in parallel with other instructions.

Example

```c
bittgl (r2, 24); /* toggle bit 24 (the 25th bit from LSB in R2 */
```

For example, if $R2$ contains 0xF1FFFFFF before this instruction, it contains 0xF0FFFFFF after the instruction. Executing the instruction a second time causes the register to contain 0xF1FFFFFF.

Also See

- BITSET, BITTST, BITCLR

Special Applications

None
Instruction Overview

BITTST

General Form

\[
CC = \text{BITTST} \left( \text{register, bit\_position} \right) \\
CC = ! \text{BITTST} \left( \text{register, bit\_position} \right)
\]

Syntax

\[
CC = \text{BITTST} \left( \text{Dreg, uimm5} \right); \quad /* \text{set CC if bit} = 1 \text{ (a)*/} \\
CC = ! \text{BITTST} \left( \text{Dreg, uimm5} \right); \quad /* \text{set CC if bit} = 0 \text{ (a)**/}
\]

Syntax Terminology

- \text{Dreg}: \text{R7-0}
- \text{uimm5}: 5\text{-bit unsigned field, with a range of 0 through 31}

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Bit Test instruction sets or clears the \( CC \) bit, based on the bit designated by \( \text{bit\_position} \) in the specified D-register. One version tests whether the specified bit is set; the other tests whether the bit is clear. The instruction does not affect other bits in the D-register.

The \( \text{bit\_position} \) range of values is 0 through 31, where 0 indicates the LSB, and 31 indicates the MSB of the 32-bit D-register.
Bit Operations

Flags Affected

The Bit Test instruction affects flags as follows.

- \( CC \) is set if the tested bit is 1; cleared otherwise.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction cannot be issued in parallel with other instructions.

Example

\[
cc = \text{bittst}(r7, 15) ; \quad /* \text{test bit 15 TRUE in R7} */
\]

For example, if \( r7 \) contains 0xFFFFFFFF before this instruction, \( CC \) is set to 1, and \( r7 \) still contains 0xFFFFFFFF after the instruction.

\[
cc = ! \text{bittst}(r3, 0) ; \quad /* \text{test bit 0 FALSE in R3} */
\]

If \( r3 \) contains 0xFFFFFFFF, this instruction clears \( CC \) to 0.

Also See

\text{BITCLR}, \text{BITSET}, \text{BITTGL}

Special Applications

None
Instruction Overview

DEPOSIT

General Form

\[
\text{dest\_reg} = \text{DEPOSIT ( backgnd\_reg, foregnd\_reg )}
\]

Syntax

\[
\text{Dreg} = \text{DEPOSIT ( Dreg, Dreg )} ; /* \text{no extension (b)} */
\]
\[
\text{Dreg} = \text{DEPOSIT ( Dreg, Dreg ) (X)} ; /* \text{sign-extended (b)} */
\]

Syntax Terminology

\[
\text{Dreg: R7-0}
\]

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Bit Field Deposit instruction merges the background bit field in \text{backgnd\_reg} with the foreground bit field in the upper half of \text{foregnd\_reg} and saves the result into \text{dest\_reg}. The user determines the length of the foreground bit field and its position in the background field.

The input register bit field definitions appear in Table 13-1.
Bit Operations

Table 13-1. Input Register Bit Field Definitions

<table>
<thead>
<tr>
<th>31.............24</th>
<th>23.............16</th>
<th>15.............8</th>
<th>7.............0</th>
</tr>
</thead>
<tbody>
<tr>
<td>backgnd_reg: 1</td>
<td>bbbb bbbb</td>
<td>bbbb bbbb</td>
<td>bbbb bbbb</td>
</tr>
<tr>
<td>foregnd_reg: 2</td>
<td>nnnn nnnn</td>
<td>nnnn nnnn</td>
<td>xxxL LLLL</td>
</tr>
</tbody>
</table>

1. where b = background bit field (32 bits)
2. where:
   - n = foreground bit field (16 bits); the L field determines the actual number of foreground bits used.
   - p = intended position of foreground bit field LSB in dest_reg (valid range 0 through 31)
   - L = length of foreground bit field (valid range 0 through 16)

The operation writes the foreground bit field of length L over the background bit field with the foreground LSB located at bit p of the background. See “Example,” below, for more.

Boundary Cases

Consider the following boundary cases.

- **Unsigned syntax, L = 0**: The architecture copies backgnd_reg contents without modification into dest_reg. By definition, a foreground of zero length is transparent.

- **Sign-extended, L = 0 and p = 0**: This case loads 0x0000 0000 into dest_reg. The sign of a zero length, zero position foreground is zero; therefore, sign-extended is all zeros.
Instruction Overview

- Sign-extended, L = 0 and p = 0: The architecture copies the lower order bits of `backgnd_reg` below position `p` into `dest_reg`, then sign-extends that number. The foreground value has no effect. For instance, if:

  \[
  \text{backgnd}_\text{reg} = 0x0000 \ 8123, \\
  \text{L} = 0, \text{ and} \\
  \text{p} = 16, \\
  \text{then:} \\
  \text{dest}_\text{reg} = 0xFFFF \ 8123.
  \]

  In this example, the architecture copies bits 15–0 from `backgnd_reg` into `dest_reg`, then sign-extends that number.

- Sign-extended, (L + p) > 32: Any foreground bits that fall outside the range 31–0 are truncated.

  The Bit Field Deposit instruction does not modify the contents of the two source registers. One of the source registers can also serve as `dest_reg`.

Options

The (X) syntax sign-extends the deposited bit field. If you specify the sign-extended syntax, the operation does not affect the `dest_reg` bits that are less significant than the deposited bit field.

Flags Affected

This instruction affects flags as follows.

- \( AZ \) is set if result is zero; cleared if nonzero.
- \( AN \) is set if result is negative; cleared if non-negative.
- \( AC0 \) is cleared.
Bit Operations

- V is cleared.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode
User & Supervisor

Parallel Issue
This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example
Bit Field Deposit Unsigned

\[ r7 = \text{deposit} \ (r4, r3) ; \]

- If
  - \( R4=0b1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \) where this is the background bit field
  - \( R3=0b0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0111 \ 0111 \ 0011 \) where bits 31–16 are the foreground bit field, bits 15–8 are the position, and bits 7–0 are the length

then the Bit Field Deposit (unsigned) instruction produces:

- \( R7=0b1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1100 \ 0111 \ 1111 \)
Instruction Overview

- If

  - R4=0b1111 1111 1111 1111 1111 1111 1111 1111
    where this is the background bit field
  - R3=0b0000 0000 1111 1010 0000 1101 0000 1001
    where bits 31–16 are the foreground bit field, bits 15–8 are
    the position, and bits 7–0 are the length

  then the Bit Field Deposit (unsigned) instruction produces:

  - R7=0b1111 1111 1111 1111 0111 1111 1111 1111

Bit Field Deposit Sign-Extended

r7 = deposit (r4, r3) (x);  /* sign-extended*/

- If

  - R4=0b1111 1111 1111 1111 1111 1111 1111 1111
    where this is the background bit field
  - R3=0b0101 0101 0101 1010 0000 0111 0000 0011
    where bits 31–16 are the foreground bit field, bits 15–8 are
    the position, and bits 7–0 are the length

  then the Bit Field Deposit (unsigned) instruction produces:

  - R7=0b0000 0000 0000 0000 0000 0001 0111 1111
Bit Operations

- If
  - R4=0b1111 1111 1111 1111 1111 1111 1111 1111
    where this is the background bit field
  - R3=0b0000 1010 1100 0000 1101 0000 1001
    where bits 31–16 are the foreground bit field, bits 15–8 are the position, and bits 7–0 are the length

then the Bit Field Deposit (unsigned) instruction produces:

- R7=0b1111 1111 1111 0101 1001 1111 1111 1111

Also See

EXTRACT

Special Applications

Video image overlay algorithms
Instruction Overview

EXTRACT

General Form

\[
dest\_reg = \text{EXTRACT} \left( \text{scene\_reg, pattern\_reg} \right) (Z)
\]

\[
dest\_reg = \text{EXTRACT} \left( \text{scene\_reg, pattern\_reg} \right) (X)
\]

Syntax

\[
Dreg = \text{EXTRACT} \left( Dreg, Dreg\_lo \right) (Z); \quad /* \text{zero-extended (b)} */
\]

\[
Dreg = \text{EXTRACT} \left( Dreg, Dreg\_lo \right) (X); \quad /* \text{sign-extended (b)} */
\]

Syntax Terminology

\[Dreg: R7-0\]

\[Dreg\_lo: R7-0.L\]

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Bit Field Extraction instruction moves only specific bits from the \text{scene\_reg} into the low-order bits of the \text{dest\_reg}. The user determines the length of the pattern bit field and its position in the scene field.

The input register bit field definitions appear in Table 13-2.
Bit Operations

Table 13-2. Input Register Bit Field Definitions

| scene_reg: where s = scene bit field (32 bits) | pattern_reg: where:
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31..................24 23..................16 15...................8 7.................0</td>
<td>-p = position of pattern bit field LSB in scene_reg (valid range 0 through 31)</td>
</tr>
<tr>
<td>ssss ssss ssss ssss ssss ssss ssss ssss</td>
<td>L = length of pattern bit field (valid range 0 through 31)</td>
</tr>
<tr>
<td>xxx  pppp  xxxL LLLL</td>
<td></td>
</tr>
</tbody>
</table>

The operation reads the pattern bit field of length $L$ from the scene bit field, with the pattern LSB located at bit $p$ of the scene. See “Example”, below, for more.

Boundary Case

If $(p + L) > 32$: In the zero-extended and sign-extended versions of the instruction, the architecture assumes that all bits to the left of the scene_reg are zero. In such a case, the user is trying to access more bits than the register actually contains. Consequently, the architecture fills any undefined bits beyond the MSB of the scene_reg with zeros.

The Bit Field Extraction instruction does not modify the contents of the two source registers. One of the source registers can also serve as dest_reg.

Options

The user has the choice of using the (X) syntax to perform sign-extend extraction or the (Z) syntax to perform zero-extend extraction.

Flags Affected

This instruction affects flags as follows.

- $AZ$ is set if result is zero; cleared if nonzero.
- $AN$ is set if result is negative; cleared if non-negative.
Instruction Overview

- \( ACO \) is cleared.
- \( V \) is cleared.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer \( \text{ASTAT} \) flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

Bit Field Extraction Unsigned

\[ r7 = \text{extract} \ (r4, \ r3.l) \ (z) \ ; \ /* \text{zero-extended}*\/
\]

- If

  - \( R4=0b1010 \ 0101 \ 1010 \ 0101 \ 1100 \ 0011 \ 1010 \ 1010 \)

  where this is the scene bit field

  - \( R3=0bxxxx \ xxxx \ xxxx \ xxxx \ 0000 \ 0111 \ 0000 \ 0100 \)

  where bits 15–8 are the position, and bits 7–0 are the length

then the Bit Field Extraction (unsigned) instruction produces:

  - \( R7=0b0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0111 \)
Bit Operations

- If

  - \( R4=0b1011\ 0101\ 1010\ 0101\ 1100\ 0011\ 1010\ 1010 \)
    where this is the scene bit field
  - \( R3=0bxxxx\ xxxx\ xxxx\ xxxx\ 0000\ 1101\ 0000\ 1001 \)
    where bits 15–8 are the position, and bits 7–0 are the length

  then the Bit Field Extraction (unsigned) instruction produces:

  - \( R7=0b0000\ 0000\ 0000\ 0000\ 0000\ 0001\ 0010\ 1110 \)

Bit Field Extraction Sign-Extended

\[
r7 = \text{extract}(r4, r3.l) \quad \text{(x)} ; \quad /* \text{sign-extended}*/
\]

- If

  - \( R4=0b1010\ 0101\ 1010\ 0101\ 1100\ 0011\ 1010\ 1010 \)
    where this is the scene bit field
  - \( R3=0bxxxx\ xxxx\ xxxx\ xxxx\ 0000\ 0111\ 0000\ 0100 \)
    where bits 15–8 are the position, and bits 7–0 are the length

  then the Bit Field Extraction (sign-extended) instruction produces:

  - \( R7=0b0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111 \)
Instruction Overview

- If
  - R4=0b1010 0101 1010 0101 1100 0011 1010 1010
    where this is the scene bit field
  - R3=0bxxxx xxxx xxxx xxxx 0000 1101 0000 1001
    where bits bits 15–8 are the position, and bits 7–0 are the length

  Then the Bit Field Extraction (sign-extended) instruction produces:
  - R7=0b1111 1111 1111 1111 1111 1110 1010 1110

Also See

DEPOSIT

Special Applications

  Video image pattern recognition and separation algorithms
Bit Operations

BITMUX

General Form

BITMUX (source_1, source_0, A0) (ASR)
BITMUX (source_1, source_0, A0) (ASL)

Syntax

BITMUX (Dreg, Dreg, A0) (ASR); /* shift right, LSB is
shifted out (b) */
BITMUX (Dreg, Dreg, A0) (ASL); /* shift left, MSB is
shifted out (b) */

Syntax Terminology

Dreg: R7-0

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Bit Multiplex instruction merges bit streams.

The instruction has two versions, Shift Right and Shift Left. This instruction overwrites the contents of source_1 and source_0. See Table 13-3, Table 13-4, and Table 13-5.

In the Shift Right version, the processor performs the following sequence.

1. Right shift Accumulator A0 by one bit. Right shift the LSB of source_1 into the MSB of the Accumulator.

2. Right shift Accumulator A0 by one bit. Right shift the LSB of source_0 into the MSB of the Accumulator.
Instruction Overview

In the Shift Left version, the processor performs the following sequence.

1. Left shift Accumulator A0 by one bit. Left shift the MSB of source_0 into the LSB of the Accumulator.

2. Left shift Accumulator A0 by one bit. Left shift the MSB of source_1 into the LSB of the Accumulator.

source_1 and source_0 must not be the same D-register.

Table 13-3. Contents Before Shift

<table>
<thead>
<tr>
<th>IF</th>
<th>39........32</th>
<th>31........24</th>
<th>23........16</th>
<th>15........8</th>
<th>7........0</th>
</tr>
</thead>
<tbody>
<tr>
<td>source_1:</td>
<td>xxxxxxxx</td>
<td>xxxxxxxx</td>
<td>xxxxxxxx</td>
<td>xxxxxxxx</td>
<td></td>
</tr>
<tr>
<td>source_0:</td>
<td>yyyy yyyy</td>
<td>yyyy yyyy</td>
<td>yyyy yyyy</td>
<td>yyyy yyyy</td>
<td></td>
</tr>
<tr>
<td>Accumulator A0:</td>
<td>zzzzz zzzzz</td>
<td>zzzzz zzzzz</td>
<td>zzzzz zzzzz</td>
<td>zzzzz zzzzz</td>
<td></td>
</tr>
</tbody>
</table>

Table 13-4. A Shift Right Instruction

<table>
<thead>
<tr>
<th>IF</th>
<th>39........32</th>
<th>31........24</th>
<th>23........16</th>
<th>15........8</th>
<th>7........0</th>
</tr>
</thead>
<tbody>
<tr>
<td>source_1:¹</td>
<td>0xxxx xxxxx</td>
<td>xxxxxxx</td>
<td>xxxxxxx</td>
<td>xxxxxxx</td>
<td></td>
</tr>
<tr>
<td>source_0:²</td>
<td>0yyyy yyyy</td>
<td>yyyy yyyy</td>
<td>yyyy yyyy</td>
<td>yyyy yyyy</td>
<td></td>
</tr>
<tr>
<td>Accumulator A0:³</td>
<td>yyyyy yyyy</td>
<td>zzzzz zzzzz</td>
<td>zzzzz zzzzz</td>
<td>zzzzz zzzzz</td>
<td></td>
</tr>
</tbody>
</table>

¹ source_1 is shifted right 1 place
² source_0 is shifted right 1 place
³ Accumulator A0 is shifted right 2 places
Table 13-5. A Shift Left Instruction

<table>
<thead>
<tr>
<th>IF</th>
<th>39........32 31........24 23........16 15........8 7..........0</th>
</tr>
</thead>
<tbody>
<tr>
<td>source_1:1</td>
<td>xxxx xxxx xxxx xxxx xxxx xxx0</td>
</tr>
<tr>
<td>source_0:2</td>
<td>yyyy yyyy yyyy yyyy yyyy yyyy yyyy yyyy</td>
</tr>
<tr>
<td>Accumulator A0:3</td>
<td>zzzz zzzz zzzz zzzz zzzz zzzz zzzz zzzz zzzz zzyx</td>
</tr>
</tbody>
</table>

1 source_1 is shifted left 1 place
2 source_0 is shifted left 1 place
3 Accumulator A0 is shifted left 2 places

Flags Affected

None

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.
Instruction Overview

Example

\[
\text{bitmux (r2, r3, a0) (asr) : /* right shift*/}
\]
- \text{If}
  - \text{R2}=0b1010 0101 1010 0101 1100 0011 1010 1010
  - \text{R3}=0b1100 0011 1010 1010 1010 0101 1010 0101
  - \text{A0}=0b0000 0000 0000 0000 0000 0000 0000 0000 0000 0111

then the Shift Right instruction produces:
- \text{R2}=0b0101 0010 1101 0010 1110 0001 1101 0101
- \text{R3}=0b0110 0001 1101 0101 0101 0010 1101 0010
- \text{A0}=0b0100 0000 0000 0000 0000 0000 0000 0000 0000 0001

\[
\text{bitmux (r3, r2, a0) (asl) : /* left shift*/}
\]
- \text{If}
  - \text{R3}=0b1010 0101 1010 0101 1100 0011 1010 1010
  - \text{R2}=0b1100 0011 1010 1010 1010 0101 1010 0101
  - \text{A0}=0b0000 0000 0000 0000 0000 0000 0000 0000 0000 0111

then the Shift Left instruction produces:
- \text{R2}=0b0100 0111 0101 0101 0100 1011 0100 1010
- \text{R3}=0b0100 1011 0100 1011 1000 0111 0101 0100
- \text{A0}=0b0000 0000 0000 0000 0000 0000 0000 0000 0000 0111

---

13-24 ADSP-BF53x/BF56x Blackfin Processor Programming Reference
Bit Operations

Also See
None

Special Applications
Convolutional encoder algorithms
Instruction Overview

ONES (One's-Population Count)

General Form

\[ \text{dest} = \text{ONES src} \]

Syntax

\[ \text{Dreg}_l = \text{ONES Dreg} ; /* (b) */ \]

Syntax Terminology

\[ \text{Dreg}: R7–0 \]
\[ \text{Dreg}_l: R7–0.L \]

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The One's-Population Count instruction loads the number of 1's contained in the src_reg into the lower half of the dest_reg.

The range of possible values loaded into dest_reg is 0 through 32.

The dest_reg and src_reg can be the same D-register. Otherwise, the One's-Population Count instruction does not modify the contents of src_reg.
Flags Affected

None

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

r3.l = ones r7 ;

If R7 contains 0xA5A5A5A5, R3.L contains the value 16, or 0x0010.

If R7 contains 0x00000081, R3.L contains the value 2, or 0x0002.

Also See

None

Special Applications

Software parity testing
14 SHIFT/ROTATE OPERATIONS

Instruction Summary

- “Add with Shift” on page 14-2
- “Shift with Add” on page 14-5
- “Arithmetic Shift” on page 14-7
- “Logical Shift” on page 14-14
- “ROT (Rotate)” on page 14-21

Instruction Overview

This chapter discusses the instructions that manipulate bit operations. Users can take advantage of these instructions to perform logical and arithmetic shifts, combine addition operations with shifts, and rotate a registered number through the Control Code (CC) bit.
Instruction Overview

Add with Shift

General Form

\[ \text{dest\_ptr} = (\text{dest\_ptr} + \text{src\_reg}) \ll 1 \]
\[ \text{dest\_ptr} = (\text{dest\_ptr} + \text{src\_reg}) \ll 2 \]
\[ \text{dest\_reg} = (\text{dest\_reg} + \text{src\_reg}) \ll 1 \]
\[ \text{dest\_reg} = (\text{dest\_reg} + \text{src\_reg}) \ll 2 \]

Syntax

Pointer Operations

\[ P\text{reg} = (P\text{reg} + P\text{reg}) \ll 1 ; \quad /* \text{dest\_reg} = (\text{dest\_reg} + \text{src\_reg}) \times 2 \quad (a) */ \]
\[ P\text{reg} = (P\text{reg} + P\text{reg}) \ll 2 ; \quad /* \text{dest\_reg} = (\text{dest\_reg} + \text{src\_reg}) \times 4 \quad (a) */ \]

Data Operations

\[ D\text{reg} = (D\text{reg} + D\text{reg}) \ll 1 ; \quad /* \text{dest\_reg} = (\text{dest\_reg} + \text{src\_reg}) \times 2 \quad (a) */ \]
\[ D\text{reg} = (D\text{reg} + D\text{reg}) \ll 2 ; \quad /* \text{dest\_reg} = (\text{dest\_reg} + \text{src\_reg}) \times 4 \quad (a) */ \]

Syntax Terminology

\[ P\text{reg}: P5-0 \]
\[ D\text{reg}: R7-0 \]

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.
Shift/Rotate Operations

Functional Description

The Add with Shift instruction combines an addition operation with a one- or two-place logical shift left. Of course, a left shift accomplishes a x2 multiplication on sign-extended numbers. Saturation is not supported.

The Add with Shift instruction does not intrinsically modify values that are strictly input. However, dest_reg serves as an input as well as the result, so dest_reg is intrinsically modified.

Flags Affected

The D-register versions of this instruction affect flags as follows.

- \( A_Z \) is set if result is zero; cleared if nonzero.
- \( A_N \) is set if result is negative; cleared if non-negative.
- \( V \) is set if result overflows; cleared if no overflow.
- \( V_S \) is set if \( V \) is set; unaffected otherwise.
- All other flags are unaffected.

The ADSP-BF53x processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

The P-register versions of this instruction do not affect any flags.

Required Mode

User & Supervisor

Parallel Issue

This instruction cannot be issued in parallel with other instructions.
Instruction Overview

Example

```
p3 = (p3+p2)<<1 ;  /* p3 = (p3 + p2) * 2 */
p3 = (p3+p2)<<2 ;  /* p3 = (p3 + p2) * 4 */
r3 = (r3+r2)<<1 ;  /* r3 = (r3 + r2) * 2 */
r3 = (r3+r2)<<2 ;  /* r3 = (r3 + r2) * 4 */
```

Also See

- Shift with Add, Logical Shift, Arithmetic Shift, Add, Multiply 32-Bit Operands

Special Applications

- None
Shift with Add

General Form

\[\text{dest\_pntr} = \text{adder\_pntr} + (\text{src\_pntr} \ll 1)\]
\[\text{dest\_pntr} = \text{adder\_pntr} + (\text{src\_pntr} \ll 2)\]

Syntax

Preg = Preg + (Preg \ll 1); /* adder\_pntr + (src\_pntr x 2) (a) */
Preg = Preg + (Preg \ll 2); /* adder\_pntr + (src\_pntr x 4) (a) */

Syntax Terminology

Preg: P5-0

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Shift with Add instruction combines a one- or two-place logical shift left with an addition operation.

The instruction provides a shift-then-add method that supports a rudimentary multiplier sequence useful for array pointer manipulation.

Flags Affected

None

Required Mode

User & Supervisor
Instruction Overview

Parallel Issue

This instruction cannot be issued in parallel with other instructions.

Example

\[
p3 = p0 + (p3 << 1); \quad /* p3 = (p3 \times 2) + p0 */
p3 = p0 + (p3 << 2); \quad /* p3 = (p3 \times 4) + p0 */
\]

Also See

Add with Shift, Logical Shift, Arithmetic Shift, Add, Multiply 32-Bit Operands

Special Applications

None
Shift/Rotate Operations

Arithmetic Shift

General Form

\[
\begin{align*}
\text{dest} & \ggg = \text{shift}\_\text{magnitude} \\
\text{dest} & = \text{src} \ggg \text{shift}\_\text{magnitude} \ (\text{opt}\_\text{sat}) \\
\text{dest} & = \text{src} \ll \text{shift}\_\text{magnitude} \ (S) \\
\text{accumulator} & = \text{accumulator} \ggg \text{shift}\_\text{magnitude} \\
\text{dest} & = \text{ASHIFT src} \ BY \ \text{shift}\_\text{magnitude} \ (\text{opt}\_\text{sat}) \\
\text{accumulator} & = \text{ASHIFT accumulator} \ BY \ \text{shift}\_\text{magnitude}
\end{align*}
\]

Syntax

Constant Shift Magnitude

\[
\begin{align*}
\text{Dreg} & \ggg = \text{uimm5} ; \quad /* \text{arithmetic right shift (a)} */ \\
\text{Dreg} & \ll = \text{uimm5} ; \quad /* \text{logical left shift (a)} */ \\
\text{Dreg}\_\text{lo}\_\text{hi} & = \text{Dreg}\_\text{lo}\_\text{hi} \ggg \text{uimm4} ; \quad /* \text{arithmetic right shift (b)} */ \\
\text{Dreg}\_\text{lo}\_\text{hi} & = \text{Dreg}\_\text{lo}\_\text{hi} \ll \text{uimm4} \ (S) ; \quad /* \text{arithmetic left shift (b)} */ \\
\text{Dreg} & = \text{Dreg} \ggg \text{uimm5} ; \quad /* \text{arithmetic right shift (b)} */ \\
\text{Dreg} & = \text{Dreg} \ll \text{uimm5} ; \quad /* \text{arithmetic left shift (b)} */ \\
\text{A0} & = \text{A0} \ggg \text{uimm5} ; \quad /* \text{arithmetic right shift (b)} */ \\
\text{A0} & = \text{A0} \ll \text{uimm5} ; \quad /* \text{logical left shift (b)} */ \\
\text{A1} & = \text{A1} \ggg \text{uimm5} ; \quad /* \text{arithmetic right shift (b)} */ \\
\text{A1} & = \text{A1} \ll \text{uimm5} ; \quad /* \text{logical left shift (b)} */ \\
\end{align*}
\]

Registered Shift Magnitude

\[
\begin{align*}
\text{Dreg} & = \text{Dreg} ; \quad /* \text{arithmetic right shift (a)} */ \\
\text{Dreg} & = \text{Dreg} ; \quad /* \text{logical left shift (a)} */ \\
\text{Dreg}\_\text{lo}\_\text{hi} & = \text{ASHIFT Dreg}\_\text{lo}\_\text{hi} \ BY \ \text{Dreg}\_\text{lo} \ (\text{opt}\_\text{sat}) ; \quad /* \text{arithmetic right shift (b)} */ \\
\text{Dreg} & = \text{ASHIFT Dreg} \ BY \ \text{Dreg}\_\text{lo} \ (\text{opt}\_\text{sat}) ; \quad /* \text{arithmetic right shift (b)} */ \\
\end{align*}
\]
Instruction Overview

A0 = ASHIFT A0 BY Dreg_lo ;  /* arithmetic right shift (b)*/
A1 = ASHIFT A1 BY Dreg_lo ;  /* arithmetic right shift (b)*/

Syntax Terminology

Dreg: R7–0
Dreg_lo_hi: R7–0.L, R7–0.H
Dreg_lo: R7–0.L
uimm4: 4-bit unsigned field, with a range of 0 through 15
uimm5: 5-bit unsigned field, with a range of 0 through 31
opt_sat: optional “(S)” (without the quotes) to invoke saturation of the result. Not optional on versions that show “(S)” in the syntax.

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

Functional Description

The Arithmetic Shift instruction shifts a registered number a specified distance and direction while preserving the sign of the original number. The sign bit value back-fills the left-most bit positions vacated by the arithmetic right shift.

Specific versions of arithmetic left shift are supported, too. Arithmetic left shift saturates the result if the value is shifted too far. A left shift that would otherwise lose nonsign bits off the left-hand side saturates to the maximum positive or negative value instead.
Shift/Rotate Operations

The “**ASHIFT**” versions of this instruction support two modes.

1. Default–arithmetic right shifts and logical left shifts. Logical left shifts do not guarantee sign bit preservation. The “**ASHIFT**” versions automatically select arithmetic and logical shift modes based on the sign of the `shift_magnitude`.

2. Saturation mode–arithmetic right and left shifts that saturate if the value is shifted left too far.

The “**>>>**” and “**>>**” versions of this instruction supports only arithmetic right shifts. If left shifts are desired, the programmer must explicitly use arithmetic “**<<**” (saturating) or logical “**<<**” (non-saturating) instructions.

Logical left shift instructions are duplicated in the Syntax section for programmer convenience. See the Logical Shift instruction for details on those operations.

The Arithmetic Shift instruction supports 16-bit and 32-bit instruction length.

- The “**>>>**” syntax instruction is 16 bits in length, allowing for smaller code at the expense of flexibility.
- The “**>>>**”, “**<<**”, and “**ASHIFT**” syntax instructions are 32 bits in length, providing a separate source and destination register, alternative data sizes, and parallel issue with Load/Store instructions.

Both syntaxes support constant and registered shift magnitudes.

For the **ASHIFT** versions, the sign of the shift magnitude determines the direction of the shift.

- Positive shift magnitudes produce Logical Left shifts.
- Negative shift magnitudes produce Arithmetic Right shifts.
In essence, the magnitude is the power of 2 multiplied by the src_reg number. Positive magnitudes cause multiplication (N \times 2^n) whereas negative magnitudes produce division (N \times 2^{-n} or N / 2^n).

The dest_reg and src_reg can be a 16-, 32-, or 40-bit register. Some versions of the Arithmetic Shift instruction support optional saturation.

See “Saturation” on page 1-17 for a description of saturation behavior.

For 16-bit src_reg, valid shift magnitudes are –16 through +15, zero included. For 32- and 40-bit src_reg, valid shift magnitudes are –32 through +31, zero included.

The D-register versions of this instruction shift 16 or 32 bits for half-word and word registers, respectively. The Accumulator versions shift all 40 bits of those registers.

The D-register versions of this instruction do not implicitly modify the src_reg values. Optionally, dest_reg can be the same D-register as src_reg. Doing this explicitly modifies the source register.

The Accumulator versions always modify the Accumulator source value.
Options

Option (S) invokes saturation of the result.

In the default case—without the saturation option—numbers can be left-shifted so far that all the sign bits overflow and are lost. However, when the saturation option is enabled, a left shift that would otherwise shift nonsign bits off the left-hand side saturates to the maximum positive or negative value instead. Consequently, with saturation enabled, the result always keeps the same sign as the original number.

See “Saturation” on page 1-17 for a description of saturation behavior.

Flags Affected

The versions of this instruction that send results to a Dreg set flags as follows.

- $A_Z$ is set if result is zero; cleared if nonzero.
- $A_N$ is set if result is negative; cleared if non-negative.
- $V$ is set if result overflows; cleared if no overflow.
- $V_S$ is set if $V$ is set; unaffected otherwise.
- All other flags are unaffected.

The versions of this instruction that send results to an Accumulator A0 set flags as follows.

- $A_Z$ is set if result is zero; cleared if nonzero.
- $A_N$ is set if result is negative; cleared if non-negative.
- $A_{V0}$ is set if result is zero; cleared if nonzero.
- $A_{V0S}$ is set if $A_{V0}$ is set; unaffected otherwise.
- All other flags are unaffected.
Instruction Overview

The versions of this instruction that send results to an Accumulator A1 set flags as follows.

- $AZ$ is set if result is zero; cleared if nonzero.
- $AN$ is set if result is negative; cleared if non-negative.
- $AV1$ is set if result is zero; cleared if nonzero.
- $AV1S$ is set if $AV1$ is set; unaffected otherwise.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer $ASTAT$ flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

The 16-bit versions of this instruction cannot be issued in parallel with other instructions.

Example

```c
r0 >>>= 19; /* 16-bit instruction length arithmetic right shift */
r3.l = r0.h >>> 7; /* arithmetic right shift, half-word */
r3.h = r0.h >>> 5; /* same as above; any combination of upper and lower half-words is supported */
```
Shift/Rotate Operations

r3.l = r0.h >>> 7(s) ;  /* arithmetic right shift, half-word, saturated */
r4 = r2 >>> 20 ;  /* arithmetic right shift, word */
A0 = A0 >>> 1 ;  /* arithmetic right shift, Accumulator */
r0 >>>= r2 ;  /* 16-bit instruction length arithmetic right shift */
r3.l = r0.h << 12 (S) ;  /* arithmetic left shift */
r5 = r2 << 24(S) ;  /* arithmetic left shift */
r3.l = ashift r0.h by r7.l ;  /* shift, half-word */
r3.h = ashift r0.l by r7.l ;
r3.h = ashift r0.h by r7.l ;
r3.l = ashift r0.l by r7.l ;
r3.l = ashift r0.h by r7.l(s) ;  /* shift, half-word, saturated */
r3.h = ashift r0.l by r7.l(s) ;  /* shift, half-word, saturated */
r3.h = ashift r0.h by r7.l(s) ;
r3.l = ashift r0.l by r7.l(s) ;
r4 = ashift r2 by r7.l ;  /* shift, word */
r4 = ashift r2 by r7.1 (s) ;  /* shift, word, saturated */
A0 = ashift A0 by r7.1 ;  /* shift, Accumulator */
A1 = ashift A1 by r7.1 ;  /* shift, Accumulator */
  // If r0.h = -64, then performing . . .
r3.h = r0.h >>> 4 ;  /* . . . produces r3.h = -4, preserving the sign */

Also See

Vector Arithmetic Shift, Vector Logical Shift, Logical Shift, Shift with Add, ROT (Rotate)

Special Applications

Multiply, divide, and normalize signed numbers
Instruction Overview

Logical Shift

General Form

\[
\begin{align*}
\text{dest}_\text{pntr} &= \text{src}_\text{pntr} \gg 1 \\
\text{dest}_\text{pntr} &= \text{src}_\text{pntr} \gg 2 \\
\text{dest}_\text{pntr} &= \text{src}_\text{pntr} \ll 1 \\
\text{dest}_\text{pntr} &= \text{src}_\text{pntr} \ll 2 \\
\text{dest}_\text{reg} &= \text{shift}_\text{magnitude} \\
\text{dest}_\text{reg} &= \text{shift}_\text{magnitude} \\
\text{dest}_\text{reg} &= \text{src}_\text{reg} \gg \text{shift}_\text{magnitude} \\
\text{dest}_\text{reg} &= \text{src}_\text{reg} \ll \text{shift}_\text{magnitude} \\
\text{dest}_\text{reg} &= \text{LSHIFT src}_\text{reg} \text{ BY shift}_\text{magnitude}
\end{align*}
\]

Syntax

**Pointer Shift, Fixed Magnitude**

\[
\begin{align*}
P_{\text{reg}} &= P_{\text{reg}} \gg 1; \quad /* \text{right shift by 1 bit (a)} */ \\
P_{\text{reg}} &= P_{\text{reg}} \gg 2; \quad /* \text{right shift by 2 bit (a)} */ \\
P_{\text{reg}} &= P_{\text{reg}} \ll 1; \quad /* \text{left shift by 1 bit (a)} */ \\
P_{\text{reg}} &= P_{\text{reg}} \ll 2; \quad /* \text{left shift by 2 bit (a)} */
\end{align*}
\]

**Data Shift, Constant Shift Magnitude**

\[
\begin{align*}
\text{Dreg} &= \text{uimm5} \gg 0; \quad /* \text{right shift (a)} */ \\
\text{Dreg} &= \text{uimm5} \ll 0; \quad /* \text{left shift (a)} */ \\
\text{Dreg}_\text{lo} &= \text{Dreg}_\text{hi} \gg \text{uimm4}; \quad /* \text{right shift (b)} */ \\
\text{Dreg}_\text{lo} &= \text{Dreg}_\text{hi} \ll \text{uimm4}; \quad /* \text{left shift (b)} */ \\
\text{Dreg} &= \text{Dreg} \gg \text{uimm5}; \quad /* \text{right shift (b)} */ \\
\text{Dreg} &= \text{Dreg} \ll \text{uimm5}; \quad /* \text{left shift (b)} */ \\
\text{A0} &= \text{A0} \gg \text{uimm5}; \quad /* \text{right shift (b)} */ \\
\text{A0} &= \text{A0} \ll \text{uimm5}; \quad /* \text{left shift (b)} */ \\
\text{A1} &= \text{A1} \ll \text{uimm5}; \quad /* \text{left shift (b)} */ \\
\text{A1} &= \text{A1} \gg \text{uimm5}; \quad /* \text{right shift (b)} */
\end{align*}
\]
Shift/Rotate Operations

Data Shift, Registered Shift Magnitude

\[ \text{Dreg} >>= \text{Dreg} ; /* \text{right shift (a)} */ \]
\[ \text{Dreg} <<= \text{Dreg} ; /* \text{left shift (a)} */ \]
\[ \text{Dreg}_{lo\_hi} = \text{LSHIFT} \text{Dreg}_{lo\_hi} \text{BY} \text{Dreg}_{lo} ; /* (b) */ \]
\[ \text{Dreg} = \text{LSHIFT} \text{Dreg} \text{BY} \text{Dreg}_{lo} ; /* (b) */ \]
\[ \text{A0} = \text{LSHIFT} \text{A0} \text{BY} \text{Dreg}_{lo} ; /* (b) */ \]
\[ \text{A1} = \text{LSHIFT} \text{A1} \text{BY} \text{Dreg}_{lo} ; /* (b) */ \]

Syntax Terminology

\[ \text{Dreg}: \text{R7–0} \]
\[ \text{Dreg}_{lo}: \text{R7–0.L} \]
\[ \text{Dreg}_{lo\_hi}: \text{R7–0.L}, \text{R7–0.H} \]
\[ \text{Preg}: \text{P5–0} \]
\[ u_{imm4}: 4\text{-bit unsigned field, with a range of 0 through 15} \]
\[ u_{imm5}: 5\text{-bit unsigned field, with a range of 0 through 31} \]

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

Functional Description

The Logical Shift instruction logically shifts a register by a specified distance and direction.

Logical shifts discard any bits shifted out of the register and backfill vacated bits with zeros.
Instruction Overview

Four versions of the Logical Shift instruction support pointer shifting. The instruction does not implicitly modify the input src_pntr value. For the P-register versions of this instruction, dest_pntr can be the same P-register as src_pntr. Doing so explicitly modifies the source register.

The rest of this description applies to the data shift versions of this instruction relating to D-registers and Accumulators.

The Logical Shift instruction supports 16-bit and 32-bit instruction length.

- The ">>=" and "<<=" syntax instruction is 16 bits in length, allowing for smaller code at the expense of flexibility.

- The ">>", "<<", and "LSHIFT" syntax instruction is 32 bits in length, providing a separate source and destination register, alternative data sizes, and parallel issue with Load/Store instructions.

Both syntaxes support constant and registered shift magnitudes.

Table 14-2. Logical Shifts

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;&gt;&gt;=&quot; and &quot;&lt;&lt;=&quot;</td>
<td>The value in dest_reg is shifted by the number of places specified by shift_magnitude. The data size is always 32 bits long. The entire 32 bits of the shift_magnitude determine the shift value. Shift magnitudes larger than 0x1F produce a 0x00000000 result.</td>
</tr>
<tr>
<td>&quot;&gt;&gt;&quot;, &quot;&lt;&lt;&quot;, and &quot;LSHIFT&quot;</td>
<td>The value in src_reg is shifted by the number of places specified in shift_magnitude, and the result is stored into dest_reg. The LSHIFT versions can shift 32-bit Dreg and 40-bit Accumulator registers by up to –32 through +31 places.</td>
</tr>
</tbody>
</table>

For the LSHIFT version, the sign of the shift magnitude determines the direction of the shift.

- Positive shift magnitudes produce Left shifts.
- Negative shift magnitudes produce Right shifts.
Shift/Rotate Operations

The `dest_reg` and `src_reg` can be a 16-, 32-, or 40-bit register.

For the `LSHIFT` instruction, the shift magnitude is the lower 6 bits of the `Dreg_lo`, sign extended. The `Dreg >>= Dreg` and `Dreg <<= Dreg` instructions use the entire 32 bits of magnitude.

The D-register versions of this instruction shift 16 or 32 bits for half-word and word registers, respectively. The Accumulator versions shift all 40 bits of those registers.

Forty-bit Accumulator values can be shifted by up to –32 to +31 bit places.

Shift magnitudes that exceed the size of the destination register produce all zeros in the result. For example, shifting a 16-bit register value by 20 bit places (a valid operation) produces 0x0000.

A shift magnitude of zero performs no shift operation at all.

The D-register versions of this instruction do not implicitly modify the `src_reg` values. Optionally, `dest_reg` can be the same D-register as `src_reg`. Doing this explicitly modifies the source register.

Flags Affected

The P-register versions of this instruction do not affect any flags.

The versions of this instruction that send results to a `Dreg` set flags as follows.

- `AZ` is set if result is zero; cleared if nonzero.
- `AN` is set if result is negative; cleared if non-negative.
- `V` is cleared.
- All other flags are unaffected.
Instruction Overview

The versions of this instruction that send results to an Accumulator A0 set flags as follows.

- \(\text{AZ}\) is set if result is zero; cleared if nonzero.
- \(\text{AN}\) is set if result is negative; cleared if non-negative.
- \(\text{AV0}\) is cleared.
- All other flags are unaffected.

The versions of this instruction that send results to an Accumulator A1 set flags as follows.

- \(\text{AZ}\) is set if result is zero; cleared if nonzero.
- \(\text{AN}\) is set if result is negative; cleared if non-negative.
- \(\text{AV1}\) is cleared.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer \(\text{ASTAT}\) flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

The 16-bit versions of this instruction cannot be issued in parallel with other instructions.
Shift/Rotate Operations

Example

```c
p3 = p2 >> 1;  /* pointer right shift by 1 */
p3 = p3 >> 2;  /* pointer right shift by 2 */
p4 = p5 << 1;  /* pointer left shift by 1 */
p0 = p1 << 2;  /* pointer left shift by 2 */
r3 >>= 17;  /* data right shift */
r3 <<= 17;  /* data left shift */
r3.1 = r0.1 >> 4;  /* data right shift, half-word register */
r3.1 = r0.h >> 4;  /* same as above; half-word register combinations are arbitrary */
r3.h = r0.1 << 12;  /* data left shift, half-word register */
r3.h = r0.h << 14;  /* same as above; half-word register combinations are arbitrary */
r3 = r6 >> 4;  /* right shift, 32-bit word */
r3 = r6 << 4;  /* left shift, 32-bit word */
a0 = a0 >> 7;  /* Accumulator right shift */
a1 = a1 >> 25;  /* Accumulator right shift */
a0 = a0 << 7;  /* Accumulator left shift */
a1 = a1 << 14;  /* Accumulator left shift */
r3 >>= r0;  /* data right shift */
r3 <<= r1;  /* data left shift */
r3.1 = lshift r0.1 by r2.1;  /* shift direction controlled by sign of R2.L */
r3.h = lshift r0.1 by r2.1;
a0 = lshift a0 by r7.1;
a1 = lshift a1 by r7.1;
    /* If r0.h = -64 (or 0xFFC0), then performing . . . */
r3.h = r0.h >> 4;  /* . . . produces r3.h = 0x0FFC (or 4092), losing the sign */
```

/* If r0.h = -64 (or 0xFFC0), then performing . . . */
r3.h = r0.h >> 4;  /* . . . produces r3.h = 0x0FFC (or 4092), losing the sign */
Instruction Overview

Also See

Arithmetic Shift, ROT (Rotate), Shift with Add, Vector Arithmetic Shift, Vector Logical Shift

Special Applications

None
Shift/Rotate Operations

**ROT (Rotate)**

**General Form**

\[
\text{dest\_reg} = \text{ROT} \text{ src\_reg BY rotate\_magnitude} \\
\text{accumulator\_new} = \text{ROT} \text{ accumulator\_old BY rotate\_magnitude}
\]

**Syntax**

**Constant Rotate Magnitude**

\[
\text{Dreg} = \text{ROT} \text{ Dreg BY imm6} : /* (b) */ \\
\text{A0} = \text{ROT} \text{ A0 BY imm6} : /* (b) */ \\
\text{A1} = \text{ROT} \text{ A1 BY imm6} : /* (b) */
\]

**Registered Rotate Magnitude**

\[
\text{Dreg} = \text{ROT} \text{ Dreg BY Dreg\_lo} : /* (b) */ \\
\text{A0} = \text{ROT} \text{ A0 BY Dreg\_lo} : /* (b) */ \\
\text{A1} = \text{ROT} \text{ A1 BY Dreg\_lo} : /* (b) */
\]

**Syntax Terminology**

\[
\text{Dreg: R7–0} \\
\text{imm6: 6-bit signed field, with a range of –32 through +31}
\]

**Instruction Length**

In the syntax, comment (b) identifies 32-bit instruction length.

**Functional Description**

The Rotate instruction rotates a register through the \( \text{CC} \) bit a specified distance and direction. The \( \text{CC} \) bit is in the rotate chain. Consequently, the first value rotated into the register is the initial value of the \( \text{CC} \) bit.
Rotation shifts all the bits either right or left. Each bit that rotates out of the register (the LSB for rotate right or the MSB for rotate left) is stored in the CC bit, and the CC bit is stored into the bit vacated by the rotate on the opposite end of the register.

### Rotate left 1 bit

<table>
<thead>
<tr>
<th>If</th>
<th>31 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-register:</td>
<td>1010 1111 0000 0000 0000 0000 0001 1010</td>
</tr>
<tr>
<td>CC bit:</td>
<td>N (“1” or “0”)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rotate left 1 bit again</th>
<th>31 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-register:</td>
<td>0111 1100 0000 0000 0000 0000 0110 10N1</td>
</tr>
<tr>
<td>CC bit:</td>
<td>0</td>
</tr>
</tbody>
</table>

### Rotate right 1 bit

<table>
<thead>
<tr>
<th>If</th>
<th>31 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-register:</td>
<td>1010 1111 0000 0000 0000 0000 0001 1010</td>
</tr>
<tr>
<td>CC bit:</td>
<td>N (“1” or “0”)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rotate right 1 bit again</th>
<th>31 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-register:</td>
<td>0101 0111 1000 0000 0000 0000 0110 1110</td>
</tr>
<tr>
<td>CC bit:</td>
<td>0</td>
</tr>
</tbody>
</table>
Shift/Rotate Operations

The sign of the rotate magnitude determines the direction of the rotation.

- Positive rotate magnitudes produce Left rotations.
- Negative rotate magnitudes produce Right rotations.

Valid rotate magnitudes are –32 through +31, zero included. The Rotate instruction masks and ignores bits that are more significant than those allowed. The distance is determined by the lower 6 bits (sign extended) of the `shift_magnitude`.

Unlike shift operations, the Rotate instruction loses no bits of the source register data. Instead, it rearranges them in a circular fashion. However, the last bit rotated out of the register remains in the `CC` bit, and is not returned to the register. Because rotates are performed all at once and not one bit at a time, rotating one direction or another regardless of the rotate magnitude produces no advantage. For instance, a rotate right by two bits is no more efficient than a rotate left by 30 bits. Both methods produce identical results in identical execution time.

The D-register versions of this instruction rotate all 32 bits. The Accumulator versions rotate all 40 bits of those registers.

The D-register versions of this instruction do not implicitly modify the `src_reg` values. Optionally, `dest_reg` can be the same D-register as `src_reg`. Doing this explicitly modifies the source register.
Instruction Overview

Flags Affected

The following flags are affected by the Rotate instruction.

- $CC$ contains the latest value shifted into it.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer $ASTAT$ flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

```c
r4 = rot r1 by 8;  /* rotate left */
r4 = rot r1 by -5;  /* rotate right */
a0 = rot a0 by 22;  /* rotate Accumulator left */
a1 = rot a1 by -31;  /* rotate Accumulator right */
r4 = rot r1 by r2.l;
a0 = rot a0 by r3.l;
a1 = rot a1 by r7.l;
```
Shift/Rotate Operations

Also See

Arithmetic Shift, Logical Shift

Special Applications

None
Instruction Overview
15 ARITHMETIC OPERATIONS

Instruction Summary

- “ABS” on page 15-3
- “Add” on page 15-6
- “Add/Subtract – Prescale Down” on page 15-10
- “Add/Subtract – Prescale Up” on page 15-13
- “Add Immediate” on page 15-16
- “DIVS, DIVQ (Divide Primitive)” on page 15-19
- “EXPADJ” on page 15-26
- “MAX” on page 15-30
- “MIN” on page 15-32
- “Modify – Decrement” on page 15-34
- “Modify – Increment” on page 15-37
- “Multiply 16-Bit Operands” on page 15-43
- “Multiply 32-Bit Operands” on page 15-51
- “Multiply and Multiply-Accumulate to Accumulator” on page 15-53
- “Multiply and Multiply-Accumulate to Half-Register” on page 15-58
Instruction Overview

- “Multiply and Multiply-Accumulate to Data Register” on page 15-67
- “Negate (Two’s-Complement)” on page 15-73
- “RND (Round to Half-Word)” on page 15-77
- “Saturate” on page 15-80
- “SIGNBITS” on page 15-83
- “Subtract” on page 15-86
- “Subtract Immediate” on page 15-90

Instruction Overview

This chapter discusses the instructions that specify arithmetic operations. Users can take advantage of these instructions to add, subtract, divide, and multiply, as well as to calculate and store absolute values, detect exponents, round, saturate, and return the number of sign bits.
Arithmetic Operations

ABS

General Form

\[ \text{dest\_reg} = \text{ABS}\ \text{src\_reg} \]

Syntax

\[
\begin{align*}
A0 &= \text{ABS}\ A0 \quad /* (b) */ \\
A0 &= \text{ABS}\ A1 \quad /* (b) */ \\
A1 &= \text{ABS}\ A0 \quad /* (b) */ \\
A1 &= \text{ABS}\ A1 \quad /* (b) */ \\
A1 &= \text{ABS}\ A1, \ A0 &= \text{ABS}\ A0 \quad /* (b) */ \\
Dreg &= \text{ABS}\ Dreg \quad /* (b) */
\end{align*}
\]

Syntax Terminology

\[ Dreg: R7-0 \]

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.
Instruction Overview

Functional Description

The Dreg form of the Absolute Value instruction calculates the absolute value of a 32-bit register and stores it into a 32-bit dest_reg. The accumulator form of this instruction takes the absolute value of a 40-bit input value in a register and produces a 40-bit result. Calculation is done according to the following rules.

- If the input value is positive or zero, copy it unmodified to the destination.
- If the input value is negative, subtract it from zero and store the result in the destination. Saturation is automatically performed with the instruction, so taking the absolute value of the largest-magnitude negative number returns the largest-magnitude positive number.

The ABS operation can also be performed on both Accumulators by a single instruction.

Flags Affected

This instruction affects flags as follows.

- AZ is set if result is zero; cleared if nonzero. In the case of two simultaneous operations, AZ represents the logical “OR” of the two.
- AN is cleared.
- V is set if the maximum negative value is saturated to the maximum positive value and the dest_reg is a Dreg; cleared if no saturation.
- VS is set if V is set; unaffected otherwise.
- AVO is set if result overflows and the dest_reg is A0; cleared if no overflow.
- AVOS is set if AVO is set; unaffected otherwise.
Arithmetic Operations

- AV1 is set if result overflows and the dest_reg is A1; cleared if no overflow.
- AV1S is set if AV1 is set; unaffected otherwise.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

```c
a0 = abs a0 ;
a0 = abs a1 ;
a1 = abs a0 ;
a1 = abs a1 ;
a1 = abs a1, a0=abs a0 ;
r3 = abs r1 ;
```

Also See

Vector ABS

Special Applications

None
Instruction Overview

Add

General Form

\[ \text{dest\_reg} = \text{src\_reg\_1} + \text{src\_reg\_2} \]

Syntax

Pointer Registers — 32-Bit Operands, 32-Bit Result

\[ \text{Preg} = \text{Preg} + \text{Preg}; \quad /* (a) */ \]

Data Registers — 32-Bit Operands, 32-bit Result

\[ \text{Dreg} = \text{Dreg} + \text{Dreg}; \quad /* \text{no saturation support but shorter} \]
\[ \text{instruction length (a) */} \]
\[ \text{Dreg} = \text{Dreg} + \text{Dreg} (\text{sat\_flag}); \quad /* \text{saturation optionally sup-} \]
\[ \text{ported, but at the cost of longer instruction length (b) */} \]

Data Registers — 16-Bit Operands, 16-Bit Result

\[ \text{Dreg\_lo\_hi} = \text{Dreg\_lo\_hi} + \text{Dreg\_lo\_hi} (\text{sat\_flag}); \quad /* (b) */ \]

Syntax Terminology

\textit{Preg}: P5–0, SP, FP

\textit{Dreg}: R7–0

\textit{Dreg\_lo\_hi}: R7–0.L, R7–0.H

\textit{sat\_flag}: nonoptional saturation flag, (S) or (NS)

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.
Arithmetic Operations

Functional Description

The Add instruction adds two source values and places the result in a destination register.

There are two ways to specify addition on 32-bit data in D-registers:

- One does not support saturation (16-bit instruction length)
- The other supports optional saturation (32-bit instruction length)

The shorter 16-bit instruction takes up less memory space. The larger 32-bit instruction can sometimes save execution time because it can be issued in parallel with certain other instructions. See “Parallel Issue” on page 15-5.

The D-register version that accepts 16-bit half-word operands stores the result in a half-word data register. This version accepts any combination of upper and lower half-register operands, and places the results in the upper or lower half of the destination register at the user’s discretion.

All versions that manipulate 16-bit data are 32 bits long.

Options

In the syntax, where sat_flag appears, substitute one of the following values.

(S) – saturate the result

(NS) – no saturation

See “Saturation” on page 1-17 for a description of saturation behavior.
Instruction Overview

Flags Affected

D-register versions of this instruction set flags as follows.

- $A_{Z}$ is set if result is zero; cleared if nonzero.
- $A_{N}$ is set if result is negative; cleared if non-negative.
- $A_{C0}$ is set if the operation generates a carry; cleared if no carry.
- $V$ is set if result overflows; cleared if no overflow.
- $V_{S}$ is set if $V$ is set; unaffected otherwise.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer $A_{STAT}$ flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

The P-register versions of this instruction do not affect any flags.

Required Mode

User & Supervisor

Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

The 16-bit versions of this instruction cannot be issued in parallel with other instructions.
Arithmetic Operations

Example

```
r5 = r2 + r1 ;  /* 16-bit instruction length add, no
    saturation */
r5 = r2 + r1(ns) ;  /* same result as above, but 32-bit
    instruction length */
r5 = r2 + r1(s) ;  /* saturate the result */
p5 = p3 + p0 ;
    /* If r0.l = 0x7000 and r7.l = 0x2000, then . . . */
r4.l = r0.l + r7.l (ns) ;  /* . . . produces r4.l = 0x9000,
    because no saturation is enforced */
    /* If r0.l = 0x7000 and r7.h = 0x2000, then . . . */
r4.l = r0.l + r7.h (s) ;  /* . . . produces r4.l = 0x7FFF, satu-
    rated to the maximum positive value */
r0.l = r2.h + r4.l(ns) ;
r1.l = r3.h + r7.h(ns) ;
r4.h = r0.l + r7.l (ns) ;
r4.h = r0.l + r7.h (ns) ;
r0.h = r2.h + r4.l(s) ;  /* saturate the result */
r1.h = r3.h + r7.h(ns) ;
```

Also See

Modify – Increment, Add with Shift, Shift with Add, Vector Add / Subtract

Special Applications

None
Instruction Overview

Add/Subtract – Prescale Down

General Form

\[
\text{dest} = \text{src}_0 + \text{src}_1 \text{ (RND20)} \\
\text{dest} = \text{src}_0 - \text{src}_1 \text{ (RND20)}
\]

Syntax

\[
\text{Dreg}_0\text{.hi} = \text{Dreg} + \text{Dreg} \text{ (RND20)}; // (b) \\
\text{Dreg}_0\text{.hi} = \text{Dreg} - \text{Dreg} \text{ (RND20);} // (b)
\]

Syntax Terminology

\(Dreg: R7-0\)

\(Dreg\_0\_hi: R7-0.L, R7-0.H\)

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Add/Subtract – Prescale Down instruction combines two 32-bit values to produce a 16-bit result as follows:

- Prescale down both input operand values by arithmetically shifting them four places to the right
- Add or subtract the operands, depending on the instruction version used
- Round the upper 16 bits of the result
- Extract the upper 16 bits to the \textit{dest\_reg}
Arithmetic Operations

The instruction supports only biased rounding. The \texttt{RND_MOD} bit in the \texttt{ASTAT} register has no bearing on the rounding behavior of this instruction.

See “Rounding and Truncating” on page 1-19 for a description of rounding behavior.

Flags Affected

The following flags are affected by this instruction:

- \texttt{AZ} is set if result is zero; cleared if nonzero.
- \texttt{AN} is set if result is negative; cleared if non-negative.
- \texttt{V} is cleared.

All other flags are unaffected.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

\begin{verbatim}
r1.l = r6+r7(rnd20) ; 
r1.l = r6-r7(rnd20) ; 
r1.h = r6+r7(rnd20) ; 
r1.h = r6-r7(rnd20) ;
\end{verbatim}
Instruction Overview

Also See

   Add/Subtract – Prescale Up, RND (Round to Half-Word), Add

Special Applications

   Typically, use the Add/Subtract – Prescale Down instruction to provide an IEEE 1180–compliant 2D 8x8 inverse discrete cosine transform.
Add/Subtract – Prescale Up

General Form

\[
\begin{align*}
\text{dest\_reg} &= \text{src\_reg\_0} + \text{src\_reg\_1} \text{ (RND12)} \\
\text{dest\_reg} &= \text{src\_reg\_0} - \text{src\_reg\_1} \text{ (RND12)}
\end{align*}
\]

Syntax

\[
\begin{align*}
\text{Dreg\_lo\_hi} &= \text{Dreg} + \text{Dreg} \text{ (RND12)} ; // (b) \\
\text{Dreg\_lo\_hi} &= \text{Dreg} - \text{Dreg} \text{ (RND12)} ; // (b)
\end{align*}
\]

Syntax Terminology

\[
\begin{align*}
\text{Dreg: R7-0} \\
\text{Dreg\_lo\_hi: R7-0.L, R7-0.H}
\end{align*}
\]

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Add/Subtract – Prescale Up instruction combines two 32-bit values to produce a 16-bit result as follows:

- Prescale up both input operand values by shifting them four places to the left
- Add or subtract the operands, depending on the instruction version used
- Round and saturate the upper 16 bits of the result
- Extract the upper 16 bits to the \textit{dest\_reg}
Instruction Overview

The instruction supports only biased rounding. The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction.

See “Saturation” on page 1-17 for a description of saturation behavior.

See “Rounding and Truncating” on page 1-19 for a description of rounding behavior.

Flags Affected

The following flags are affected by this instruction:

- AZ is set if result is zero; cleared if nonzero.
- AN is set if result is negative; cleared if non-negative.
- V is set if result saturates; cleared if no saturation.
- VS is set if V is set; unaffected otherwise.

All other flags are unaffected.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

r1.l = r6+r7(rnd12) ;  
r1.l = r6-r7(rnd12) ;  
r1.h = r6+r7(rnd12) ;  
r1.h = r6-r7(rnd12) ;
Arithmetic Operations

Also See

RND (Round to Half-Word), Add/Subtract – Prescale Down, Add

Special Applications

Typically, use the Add/Subtract – Prescale Up instruction to provide an IEEE 1180–compliant 2D 8x8 inverse discrete cosine transform.
Instruction Overview

Add Immediate

General Form

register += constant

Syntax

Dreg += imm7;  /* Dreg = Dreg + constant (a) */
Preg += imm7;  /* Preg = Preg + constant (a) */
Ireg += 2;  /* increment Ireg by 2, half-word address pointer increment (a) */
Ireg += 4;  /* word address pointer increment (a) */

Syntax Terminology

Dreg: R7–0
Preg: P5–0, SP, FP
Ireg: I3–0

imm7: 7-bit signed field, with the range of –64 through +63

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Add Immediate instruction adds a constant value to a register without saturation.

To subtract immediate values from I-registers, use the Subtract Immediate instruction.

The instruction versions that explicitly modify Ireg support optional circular buffering. See “Automatic Circular Addressing”
on page 1-21 for more details. Unless circular buffering is desired, disable it prior to issuing this instruction by clearing the Length Register \((\text{Lreg})\) corresponding to the \(\text{Ireg}\) used in this instruction. Example: If you use \(\text{I2}\) to increment your address pointer, first clear \(\text{L2}\) to disable circular buffering. Failure to explicitly clear \(\text{Lreg}\) beforehand can result in unexpected \(\text{Ireg}\) values.

The circular address buffer registers (Index, Length, and Base) are not initialized automatically by Reset. Traditionally, user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes them later, if needed.

Flags Affected

D-register versions of this instruction set flags as follows.

- \(\text{AZ}\) is set if result is zero; cleared if nonzero.
- \(\text{AN}\) is set if result is negative; cleared if non-negative.
- \(\text{ACO}\) is set if the operation generates a carry; cleared if no carry.
- \(\text{V}\) is set if result overflows; cleared if no overflow.
- \(\text{VS}\) is set if \(\text{V}\) is set; unaffected otherwise.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer \(\text{ASTAT}\) flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

The P-register and I-register versions of this instruction do not affect any flags.
Instruction Overview

Required Mode

User & Supervisor

Parallel Issue

The Index Register versions of this instruction can be issued in parallel with specific other instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

The Data Register and Pointer Register versions of this instruction cannot be issued in parallel with other instructions.

Example

```c
r0 += 40;
p5 += -4;  /* decrement by adding a negative value */
i0 += 2;
i1 += 4;
```

Also See

Subtract Immediate

Special Applications

None
Arithmetic Operations

DIVS, DIVQ (Divide Primitive)

General Form

DIVS ( dividend_register, divisor_register )
DIVQ ( dividend_register, divisor_register )

Syntax

DIVS ( Dreg, Dreg ) ; /* Initialize for DIVQ. Set the AQ flag based on the signs of the 32-bit dividend and the 16-bit divisor. Left shift the dividend one bit. Copy AQ into the dividend LSB. (a) */
DIVQ ( Dreg, Dreg ) ; /* Based on AQ flag, either add or subtract the divisor from the dividend. Then set the AQ flag based on the MSBs of the 32-bit dividend and the 16-bit divisor. Left shift the dividend one bit. Copy the logical inverse of AQ into the dividend LSB. (a) */

Syntax Terminology

Dreg: R7-0

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Divide Primitive instruction versions are the foundation elements of a nonrestoring conditional add-subtract division algorithm. See “Example” on page 15-24 for such a routine.

The dividend (numerator) is a 32-bit value. The divisor (denominator) is a 16-bit value in the lower half of divisor_register. The high-order half-word of divisor_register is ignored entirely.
The division can either be signed or unsigned, but the dividend and divisor must both be of the same type. The divisor cannot be negative. A signed division operation, where the dividend may be negative, begins the sequence with the DIVS (“divide-sign”) instruction, followed by repeated execution of the DIVQ (“divide-quotient”) instruction. An unsigned division omits the DIVS instruction. In that case, the user must manually clear the AQ flag of the ASTAT register before issuing the DIVQ instructions.

Up to 16 bits of signed quotient resolution can be calculated by issuing DIVS once, then repeating the DIVQ instruction 15 times. A 16-bit unsigned quotient is calculated by omitting DIVS, clearing the AQ flag, then issuing 16 DIVQ instructions.

Less quotient resolution is produced by executing fewer DIVQ iterations.

The result of each successive addition or subtraction appears in dividend_register, aligned and ready for the next addition or subtraction step. The contents of divisor_register are not modified by this instruction.

The final quotient appears in the low-order half-word of dividend_register at the end of the successive add/subtract sequence.

DIVS computes the sign bit of the quotient based on the signs of the dividend and divisor. DIVS initializes the AQ flag based on that sign, and initializes the dividend for the first addition or subtraction. DIVS performs no addition or subtraction.

DIVQ either adds (dividend + divisor) or subtracts (dividend – divisor) based on the AQ flag, then reinitializes the AQ flag and dividend for the next iteration. If AQ is 1, addition is performed; if AQ is 0, subtraction is performed.

See “Flags Affected” on page 15-4 for the conditions that set and clear the AQ flag.
Arithmetic Operations

Both instruction versions align the dividend for the next iteration by left shifting the dividend one bit to the left (without carry). This left shift accomplishes the same function as aligning the divisor one bit to the right, such as one would do in manual binary division.

The format of the quotient for any numeric representation can be determined by the format of the dividend and divisor. Let:

- **NL** represent the number of bits to the left of the binary point of the dividend, and
- **NR** represent the number of bits to the right of the binary point of the dividend (numerator);
- **DL** represent the number of bits to the left of the binary point of the divisor, and
- **DR** represent the number of bits to the right of the binary point of the divisor (denominator).

Then the quotient has \( NL - DL + 1 \) bits to the left of the binary point and \( NR - DR - 1 \) bits to the right of the binary point. See the following example.

| Dividend (numerator) | BBBB B . | BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
<table>
<thead>
<tr>
<th>NL bits</th>
<th>NR bits</th>
</tr>
</thead>
</table>
| Divisor (denominator) | BB . | BB BBB BBB BBB BBB
| DL bits | DR bits |
| Quotient | BBBB . | BBB BBB BBB BBB
| NL - DL +1 | NR - DR - 1 |
| (5 - 2 + 1) | (27 - 14 - 1) |
| 4.12 format |

Some format manipulation may be necessary to guarantee the validity of the quotient. For example, if both operands are signed and fully fractional (dividend in 1.31 format and divisor in 1.15 format), the result is fully

ADSP-BF53x/BF56x Blackfin Processor Programming Reference 15-21
fractional (in 1.15 format) and therefore the upper 16 bits of the dividend must have a smaller magnitude than the divisor to avoid a quotient overflow beyond 16 bits. If an overflow occurs, AV0 is set. User software is able to detect the overflow, rescale the operand, and repeat the division.

Dividing two integers (32.0 dividend by a 16.0 divisor) results in an invalid quotient format because the result will not fit in a 16-bit register. To divide two integers (dividend in 32.0 format and divisor in 16.0 format) and produce an integer quotient (in 16.0 format), one must shift the dividend one bit to the left (into 31.1 format) before dividing. This requirement to shift left limits the usable dividend range to 31 bits. Violations of this range produce an invalid result of the division operation.

The algorithm overflows if the result cannot be represented in the format of the quotient as calculated above, or when the divisor is zero or less than the upper 16 bits of the dividend in magnitude (which is tantamount to multiplication).

Error Conditions

Two special cases can produce invalid or inaccurate results. Software can trap and correct both cases.

1. The Divide Primitive instructions do not support signed division by a negative divisor. Attempts to divide by a negative divisor result in a quotient that is, in most cases, one LSB less than the correct value. If division by a negative divisor is required, follow the steps below.
   
   • Before performing the division, save the sign of the divisor in a scratch register.
   
   • Calculate the absolute value of the divisor and use that value as the divisor operand in the Divide Primitive instructions.
After the divide sequence concludes, multiply the resulting quotient by the original divisor sign.

The quotient then has the correct magnitude and sign.

2. The Divide Primitive instructions do not support unsigned division by a divisor greater than 0x7FFF. If such divisions are necessary, prescale both operands by shifting the dividend and divisor one bit to the right prior to division. The resulting quotient will be correctly aligned.

Of course, prescaling the operands decreases their resolution, and may introduce one LSB of error in the quotient. Such error can be detected and corrected by the following steps.

- Save the original (unscaled) dividend and divisor in scratch registers.
- Prescale both operands as described and perform the division as usual.
- Multiply the resulting quotient by the unscaled divisor. Do not corrupt the quotient by the multiplication step.
- Subtract the product from the unscaled dividend. This step produces an error value.
- Compare the error value to the unscaled divisor.
  - If error > divisor, add one LSB to the quotient.
  - If error < divisor, subtract one LSB from the quotient.
  - If error = divisor, do nothing.

Tested examples of these solutions are planned to be added in a later edition of this document.
Instruction Overview

Flags Affected

This instruction affects flags as follows.

- \( AQ \) equals \( dividend_{MSB} \) Exclusive-OR \( divisor_{MSB} \) where dividend is a 32-bit value and divisor is a 16-bit value.

- All other flags are unaffected.

The ADSP-BF535 processor has fewer \( ASTAT \) flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction cannot be issued in parallel with other instructions.

Example

/* Evaluate given a signed integer dividend and divisor */
p0 = 15 ; /* Evaluate the quotient to 16 bits. */
r0 = 70 ; /* Dividend, or numerator */
r1 = 5 ; /* Divisor, or denominator */
r0 <<= 1 ; /* Left shift dividend by 1 needed for integer division */
divs (r0, r1) ; /* Evaluate quotient MSB. Initialize AQ flag and dividend for the DIVQ loop. */
loop .div_prim lc0=p0 ; /* Evaluate DIVQ p0=15 times. */
loop_begin .div_prim ;
divq (r0, r1) ;
loop_end .div_prim ;
Arithmetic Operations

r0 = r0.l(x); /* Sign extend the 16-bit quotient to 32 bits. */
/* r0 contains the quotient (70/5 = 14). */

Also See

LSETUP, LOOP, Multiply 32-Bit Operands

Special Applications

None
Instruction Overview

EXPADJ

General Form

dest_reg = EXPADJ ( sample_register, exponent_register )

Syntax

Dreg_lo = EXPADJ ( Dreg, Dreg_lo ) ; /* 32-bit sample (b) */
Dreg_lo = EXPADJ ( Dreg_lo_hi, Dreg_lo ) ; /* one 16-bit sample (b) */
Dreg_lo = EXPADJ ( Dreg, Dreg_lo ) (V) ; /* two 16-bit samples (b) */

Syntax Terminology

Dreg_lo_hi: R7-0.L, R7-0.H
Dreg_lo: R7-0.L
Dreg: R7-0

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Exponent Detection instruction identifies the largest magnitude of two or three fractional numbers based on their exponents. It compares the magnitude of one or two sample values to a reference exponent and returns the smallest of the exponents.

The exponent is the number of sign bits minus one. In other words, the exponent is the number of redundant sign bits in a signed number.
Arithmetic Operations

Exponents are unsigned integers. The Exponent Detection instruction accommodates the two special cases (0 and –1) and always returns the smallest exponent for each case.

The reference exponent and destination exponent are 16-bit half-word unsigned values. The sample number can be either a word or half-word. The Exponent Detection instruction does not implicitly modify input values. The dest_reg and exponent_register can be the same D-register. Doing this explicitly modifies the exponent_register.

The valid range of exponents is 0 through 31, with 31 representing the smallest 32-bit number magnitude and 15 representing the smallest 16-bit number magnitude.

Exponent Detection supports three types of samples—one 32-bit sample, one 16-bit sample (either upper-half or lower-half word), and two 16-bit samples that occupy the upper-half and lower-half words of a single 32-bit register.

Flags Affected

None

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.
Instruction Overview

Example

```c
r5.l = expadj (r4, r2.l) ;
```

- Assume R4 = 0x0000 0052 and R2.L = 12. Then R5.L becomes 12.
- Assume R4 = 0xFFFF 0052 and R2.L = 12. Then R5.L becomes 12.
- Assume R4 = 0x0000 0052 and R2.L = 27. Then R5.L becomes 24.
- Assume R4 = 0xF000 0052 and R2.L = 27. Then R5.L becomes 3.

```c
r5.l = expadj (r4.l, r2.l) ;
```


```c
r5.l = expadj (r4.h, r2.l) ;
```


```c
r5.l = expadj (r4, r2.l)(v) ;
```


Also See

SIGNBITS
Arithmetic Operations

Special Applications

EXPADJ detects the exponent of the largest magnitude number in an array. The detected value may then be used to normalize the array on a subsequent pass with a shift operation. Typically, use this feature to implement block floating-point capabilities.
Instruction Overview

MAX

General Form

\[
\text{dest\_reg} = \text{MAX} \left( \text{src\_reg\_0}, \text{src\_reg\_1} \right)
\]

Syntax

\[
\text{Dreg} = \text{MAX} \left( \text{Dreg}, \text{Dreg} \right) ; /* 32-bit operands (b) */
\]

Syntax Terminology

\text{Dreg}: R7-0

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Maximum instruction returns the maximum, or most positive, value of the source registers. The operation subtracts \text{src\_reg\_1} from \text{src\_reg\_0} and selects the output based on the signs of the input values and the arithmetic flags.

The Maximum instruction does not implicitly modify input values. The \text{dest\_reg} can be the same D-register as one of the source registers. Doing this explicitly modifies the source register.

Flags Affected

This instruction affects flags as follows.

- \text{AZ} is set if result is zero; cleared if nonzero.
- \text{AN} is set if result is negative; cleared if non-negative.
Arithmetic Operations

- V is cleared.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

\[ r5 = \text{max} \ (r2, r3) ; \]

- Assume \( r2 = 0x00000000 \) and \( r3 = 0x0000000F \), then \( r5 = 0x0000000F \).
- Assume \( r2 = 0x80000000 \) and \( r3 = 0x0000000F \), then \( r5 = 0x0000000F \).
- Assume \( r2 = 0xFFFFFFFF \) and \( r3 = 0x0000000F \), then \( r5 = 0x0000000F \).

Also See

MIN, Vector MAX, Vector MIN, VIT_MAX (Compare-Select)

Special Applications

None
Instruction Overview

MIN

General Form

\[ \text{dest\_reg} = \text{MIN} ( \text{src\_reg\_0}, \text{src\_reg\_1} ) \]

Syntax

\[ Dreg = \text{MIN} ( Dreg, Dreg ); \quad /* \text{32-bit operands (b)} */ \]

Syntax Terminology

\( Dreg: R7-0 \)

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Minimum instruction returns the minimum value of the source registers to the \( \text{dest\_reg} \). (The minimum value of the source registers is the value closest to \(-\infty\).) The operation subtracts \( \text{src\_reg\_1} \) from \( \text{src\_reg\_0} \) and selects the output based on the signs of the input values and the arithmetic flags.

The Minimum instruction does not implicitly modify input values. The \( \text{dest\_reg} \) can be the same D-register as one of the source registers. Doing this explicitly modifies the source register.

Flags Affected

This instruction affects flags as follows.

- \( AZ \) is set if result is zero; cleared if nonzero.

- \( AN \) is set if result is negative; cleared if non-negative.
**Arithmetic Operations**

- $v$ is cleared.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer `ASTAT` flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

**Required Mode**

User & Supervisor

**Parallel Issue**

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

**Example**

$r5 = \text{min} (r2, r3)$;

- Assume $R2 = 0x00000000$ and $R3 = 0x0000000F$, then $R5 = 0x00000000$.
- Assume $R2 = 0x80000000$ and $R3 = 0x0000000F$, then $R5 = 0x80000000$.
- Assume $R2 = 0xFFFFFFFF$ and $R3 = 0x0000000F$, then $R5 = 0xFFFFFFFF$.

**Also See**

MAX, Vector MAX, Vector MIN

**Special Applications**

None
Instruction Overview

Modify – Decrement

General Form

\[ \text{dest}_\text{reg} -= \text{src}_\text{reg} \]

Syntax

40-Bit Accumulators

\[ A0 \ -= \ A1 ; \quad /* \text{dest}_\text{reg}_\text{new} = \text{dest}_\text{reg}_\text{old} - \text{src}_\text{reg}, \text{saturate} \text{the result at 40 bits (b) */} \]
\[ A0 \ -= \ A1 \ (\text{W32}) ; \quad /* \text{dest}_\text{reg}_\text{new} = \text{dest}_\text{reg}_\text{old} - \text{src}_\text{reg}, \text{decrement and saturate the result at 32 bits, sign extended (b) */} \]

32-Bit Registers

\[ \text{Preg} \ -= \ \text{Preg} \ ; \quad /* \text{dest}_\text{reg}_\text{new} = \text{dest}_\text{reg}_\text{old} - \text{src}_\text{reg} \ (a) */ \]
\[ \text{Ireg} \ -= \ \text{Mreg} \ ; \quad /* \text{dest}_\text{reg}_\text{new} = \text{dest}_\text{reg}_\text{old} - \text{src}_\text{reg} \ (a) */ \]

Syntax Terminology

\[ \text{Preg}: \ P5-0, \ SP, \ FP \]
\[ \text{Ireg}: \ I3-0 \]
\[ \text{Mreg}: \ M3-0 \]

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

Functional Description

The Modify – Decrement instruction decrements a register by a user-defined quantity.
Arithmetic Operations

See “Saturation” on page 1-17 for a description of saturation behavior.

The instruction versions that explicitly modify $I_{reg}$ support optional circular buffering. See “Automatic Circular Addressing” on page 1-21 for more details. Unless circular buffering is desired, disable it prior to issuing this instruction by clearing the Length Register ($L_{reg}$) corresponding to the $I_{reg}$ used in this instruction. Example: If you use $I_2$ to increment your address pointer, first clear $L_2$ to disable circular buffering. Failure to explicitly clear $L_{reg}$ beforehand can result in unexpected $I_{reg}$ values.

The circular address buffer registers (Index, Length, and Base) are not initialized automatically by Reset. Traditionally, user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes them later, if needed.

Flags Affected

The Accumulator versions of this instruction affect the flags as follows.

- $A_Z$ is set if result is zero; cleared if nonzero.
- $A_N$ is set if result is negative; cleared if non-negative.
- $A_{C0}$ is set if the operation generates a carry; cleared if no carry.
- $A_{V0}$ is set if result saturates; cleared if no saturation.
- $A_{VO}$ is set if $A_{V0}$ is set; unaffected otherwise.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer $A_{STAT}$ flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

The P-register and I-register versions do not affect any flags.
Instruction Overview

Required Mode

User & Supervisor

Parallel Issue

The 32-bit versions of this instruction and the 16-bit versions that use Ireg can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

All other 16-bit versions of this instruction cannot be issued in parallel with other instructions.

Example

a0 -= a1 ;
a0 -= a1 (w32) ;
p3 -= p0 ;
i1 -= m2 ;

Also See

Modify – Increment, Subtract, Shift with Add

Special Applications

Typically, use the Index Register and Pointer Register versions of the Modify – Decrement instruction to decrement indirect address pointers for load or store operations.
Arithmetic Operations

Modify – Increment

General Form

\[ \text{dest}_\text{reg} += \text{src}_\text{reg} \]
\[ \text{dest}_\text{reg} = ( \text{src}_\text{reg}_0 += \text{src}_\text{reg}_1 ) \]

Syntax

40-Bit Accumulators

\[ A0 += A1 ; /* \text{dest}_\text{reg}_\text{new} = \text{dest}_\text{reg}_\text{old} + \text{src}_\text{reg}, \text{saturate the result at 40 bits (b)} */ \]
\[ A0 += A1 (W32) ; /* \text{dest}_\text{reg}_\text{new} = \text{dest}_\text{reg}_\text{old} + \text{src}_\text{reg}, \text{signed saturate the result at 32 bits, sign extended (b)} */ \]

32-Bit Registers

\[ Preg += Preg (BREV) ; /* \text{dest}_\text{reg}_\text{new} = \text{dest}_\text{reg}_\text{old} + \text{src}_\text{reg}, \text{bit reversed carry, only (a)} */ \]
\[ Ireg += Mreg (opt_brev) ; /* \text{dest}_\text{reg}_\text{new} = \text{dest}_\text{reg}_\text{old} + \text{src}_\text{reg}, \text{optional bit reverse (a)} */ \]
\[ Dreg = ( A0 += A1 ) ; /* \text{increment 40-bit A0 by A1 with saturation at 40 bits, then extract the result into a 32-bit register with saturation at 32 bits (b)} */ \]

16-Bit Half-Word Data Registers

\[ Dreg_{lo\_hi} = ( A0 += A1 ) ; /* \text{increment 40-bit A0 by A1 with saturation at 40 bits, then extract the result into a half register. The extraction step involves first rounding the 40-bit} \]

ADSP-BF53x/BF56x Blackfin Processor Programming Reference 15-37
Instruction Overview

result at bit 16 (according to the RND.MOD bit in the ASTAT register), then saturating at 32 bits and moving bits 31:16 into the half register. (b) */

Syntax Terminology

\[ Dreg: R7-0 \]
\[ Preg: P5-0, SP, FP \]
\[ Ireg: I3-0 \]
\[ Mreg: M3-0 \]

opt\_brev: optional bit reverse syntax; replace with (brev)

\[ Dreg\_lo\_hi: R7-0.L, R7-0.H \]

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

Functional Description

The Modify – Increment instruction increments a register by a user-defined quantity. In some versions, the instruction copies the result into a third register.

The 16-bit Half-Word Data Register version increments the 40-bit \( A0 \) by \( A1 \) with saturation at 40 bits, then extracts the result into a half register. The extraction step involves first rounding the 40-bit result at bit 16 (according to the RND.MOD bit in the ASTAT register), then saturating at 32 bits and moving bits 31–16 into the half register.

See “Saturation” on page 1-17 for a description of saturation behavior.
Arithmetic Operations

See “Rounding and Truncating” on page 1-19 for a description of rounding behavior.

The instruction versions that explicitly modify Ireg support optional circular buffering. See “Automatic Circular Addressing” on page 1-21 for more details. Unless circular buffering is desired, disable it prior to issuing this instruction by clearing the Length Register (Lreg) corresponding to the Ireg used in this instruction. Example: If you use I2 to increment your address pointer, first clear L2 to disable circular buffering. Failure to explicitly clear Lreg beforehand can result in unexpected Ireg values.

The circular address buffer registers (Index, Length, and Base) are not initialized automatically by Reset. Traditionally, user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes them later, if needed.

Options

(BREV)—bit reverse carry adder. When specified, the carry bit is propagated from left to right, as shown in Figure 15-1, instead of right to left.

When bit reversal is used on the Index Register version of this instruction, circular buffering is disabled to support operand addressing for FFT, DCT and DFT algorithms. The Pointer Register version does not support circular buffering in any case.

Table 15-1. Bit Addition Flow for the Bit Reverse (BREV) Case

<table>
<thead>
<tr>
<th>an</th>
<th>cn</th>
<th>a2</th>
<th>c2</th>
<th>a1</th>
<th>c1</th>
<th>a0</th>
<th>c0</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bn</td>
<td>b2</td>
<td>b1</td>
<td>b0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Instruction Overview

Flags Affected

The versions of the Modify – Increment instruction that store the results in an Accumulator affect flags as follows.

- AZ is set if Accumulator result is zero; cleared if nonzero.
- AN is set if Accumulator result is negative; cleared if non-negative.
- ACO is set if the operation generates a carry; cleared if no carry.
- V is set if result saturates and the dest_reg is a Dreg; cleared if no saturation.
- VS is set if V is set; unaffected otherwise.
- AVO is set if result saturates and the dest_reg is A0; cleared if no saturation.
- AVOS is set if AVO is set; unaffected otherwise.
- All other flags are unaffected.

The versions of the Modify – Increment instruction that store the results in a Data Register affect flags as follows.

- AZ is set if Data Register result is zero; cleared if nonzero.
- AN is set if Data Register result is negative; cleared if non-negative.
- ACO is set if the operation generates a carry; cleared if no carry.
- V is set if result saturates and the dest_reg is a Dreg; cleared if no saturation.
- VS is set if V is set; unaffected otherwise.
- AVO is set if result saturates and the dest_reg is A0; cleared if no saturation.
Arithmetic Operations

- `AVOS` is set if `AV0` is set; unaffected otherwise.
- All other flags are unaffected.

ℹ️ The ADSP-BF535 processor has fewer `ASTAT` flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

The Pointer Register, Index Register, and Modify Register versions of the instruction do not affect the flags.

**Required Mode**

User & Supervisor

**Parallel Issue**

The 32-bit versions of this instruction and the 16-bit versions that use `Ireg` can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

All other 16-bit versions of this instruction cannot be issued in parallel with other instructions.

**Example**

```c
a0 += a1 ;
a0 += a1 (w32) ;
p3 += p0 (brev) ;
i1 += m1 ;
i0 += m0 (brev) ;  /* optional carry bit reverse mode */
r5 = (a0 += a1) ;
r2.l = (a0 += a1) ;
r5.h = (a0 += a1) ;
```
Instruction Overview

Also See

*Modify – Decrement, Add, Shift with Add*

Special Applications

Typically, use the Index Register and Pointer Register versions of the Modify – Increment instruction to increment indirect address pointers for load or store operations.
Arithmetic Operations

Multiply 16-Bit Operands

General Form

\[
\text{dest\_reg} = \text{src\_reg}_0 \times \text{src\_reg}_1 \ (\text{opt\_mode})
\]

Syntax

Multiply-And-Accumulate Unit 0 (MAC0)

\[
\text{Dreg\_lo} = \text{Dreg\_lo\_hi} \times \text{Dreg\_lo\_hi} \ (\text{opt\_mode}_1) ; \ /* 16-bit
\text{result into the destination lower half-word register (b) */}
\text{Dreg\_even} = \text{Dreg\_lo\_hi} \times \text{Dreg\_lo\_hi} \ (\text{opt\_mode}_2) ; \ /* 32-bit
\text{result (b) */}
\]

Multiply-And-Accumulate Unit 1 (MAC1)

\[
\text{Dreg\_hi} = \text{Dreg\_lo\_hi} \times \text{Dreg\_lo\_hi} \ (\text{opt\_mode}_1) ; \ /* 16-bit
\text{result into the destination upper half-word register (b) */}
\text{Dreg\_odd} = \text{Dreg\_lo\_hi} \times \text{Dreg\_lo\_hi} \ (\text{opt\_mode}_2) ; \ /* 32-bit
\text{result (b) */}
\]

Syntax Terminology

\[
\text{Dreg: R7-0}
\]

\[
\text{Dreg\_lo: R7-0.L}
\]

\[
\text{Dreg\_hi: R7-0.H}
\]

\[
\text{Dreg\_lo\_hi: R7-0.L, R7-0.H}
\]

\text{opt\_mode}_1: \text{Optionally} \ (\text{FU}), (\text{IS}), (\text{IU}), (\text{T}), (\text{TFU}), (\text{S2RND}), (\text{ISS2}) \text{ or } (\text{IH}). \text{Optionally, (M) can be used with MAC1 versions either alone or with any of these other options. When used together, the option flags must be enclosed in one set of parentheses and separated by a comma. Example: (M, IS)}
Instruction Overview

opt_mode_2: Optionally (FU), (IS), or (ISS2). Optionally, (M) can be used with MAC1 versions either alone or with any of these other options. When used together, the option flags must be enclosed in one set of parenthesis and separated by a comma. Example: (M, IS)

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Multiply 16-Bit Operands instruction multiplies the two 16-bit operands and stores the result directly into the destination register with saturation.

The instruction is like the Multiply-Accumulate instructions, except that Multiply 16-Bit Operands does not affect the Accumulators.

Operations performed by the Multiply-and-Accumulate Unit 0 (MAC0) portion of the architecture load their 16-bit results into the lower half of the destination data register; 32-bit results go into an even numbered Dreg. Operations performed by MAC1 load their results into the upper half of the destination data register or an odd numbered Dreg. In 32-bit result syntax, the MAC performing the operation will be determined by the destination Dreg. Even-numbered Dregs (R6, R4, R2, R0) invoke MAC0. Odd-numbered Dregs (R7, R5, R3, R1) invoke MAC1. Therefore, 32-bit result operations using the (M) option can only be performed on odd-numbered Dreg destinations.

In 16-bit result syntax, the MAC performing the operation will be determined by the destination Dreg half. Low-half Dregs (R7-0.L) invoke MAC0. High-half Dregs (R7-0.H) invoke MAC1. Therefore, 16-bit result operations using the (M) option can only be performed on high-half Dreg destinations.
The versions of this instruction that produce 16-bit results are affected by the `RND_MOD` bit in the `ASTAT` register when they copy the results into the 16-bit destination register. `RND_MOD` determines whether biased or unbiased rounding is used. `RND_MOD` controls rounding for all versions of this instruction that produce 16-bit results except the `(IS)`, `(IU)` and `(ISS2)` options.

See “Saturation” on page 1-17 for a description of saturation behavior.

See “Rounding and Truncating” on page 1-19 for a description of rounding behavior.

The versions of this instruction that produce 32-bit results do not perform rounding and are not affected by the `RND_MOD` bit in the `ASTAT` register.

**Options**

The Multiply 16-Bit Operands instruction supports the following options. Saturation is supported for every option.

To truncate the result, the operation eliminates the least significant bits that do not fit into the destination register.

In fractional mode, the product of the smallest representable fraction times itself (for example, 0x8000 times 0x8000) is saturated to the maximum representable positive fraction (0x7FFF).
## Instruction Overview

### Table 15-2. Multiply 16-Bit Operands Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description for Register Half Destination</th>
<th>Description for 32-Bit Register Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>Signed fraction. Multiply 1.15 * 1.15 to produce 1.31 results after left-shift correction. Round 1.31 format value at bit 16. (RND_MOD bit in the ASTAT register controls the rounding.) Saturate the result to 1.15 precision in destination register half. Result is between minimum -1 and maximum 1-2^-15 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).</td>
<td>Signed fraction. Multiply 1.15 * 1.15 to produce 1.31 results after left-shift correction. Saturate results between minimum -1 and maximum 1-2^-31. The resulting hexadecimal range is minimum 0x8000 0000 through maximum 0x7FFF FFFF.</td>
</tr>
<tr>
<td>(FU)</td>
<td>Unsigned fraction. Multiply 0.16 * 0.16 to produce 0.32 results. No shift correction. Round 0.32 format value at bit 16. (RND_MOD bit in the ASTAT register controls the rounding.) Saturate the result to 0.16 precision in destination register half. Result is between minimum 0 and maximum 1-2^-16 (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).</td>
<td>Unsigned fraction. Multiply 0.16 * 0.16 to produce 0.32 results. No shift correction. Saturate results between minimum 0 and maximum 1-2^-32. Unsigned integer. Multiply 16.0 * 16.0 to produce 32.0 results. No shift correction. Saturate results between minimum 0 and maximum 2^32-1. In either case, the resulting hexadecimal range is minimum 0x0000 0000 through maximum 0xFFFF FFFF.</td>
</tr>
<tr>
<td>(IS)</td>
<td>Signed integer. Multiply 16.0 * 16.0 to produce 32.0 results. No shift correction. Extract the lower 16 bits. Saturate for 16.0 precision in destination register half. Result is between minimum -2^{15} and maximum 2^{15}-1 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).</td>
<td>Signed integer. Multiply 16.0 * 16.0 to produce 32.0 results. No shift correction. Saturate integer results between minimum -2^{31} and maximum 2^{31}-1.</td>
</tr>
<tr>
<td>(IU)</td>
<td>Unsigned integer. Multiply 16.0 * 16.0 to produce 32.0 results. No shift correction. Extract the lower 16 bits. Saturate for 16.0 precision in destination register half. Result is between minimum 0 and maximum 2^{16}-1 (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).</td>
<td>Not applicable. Use (IS).</td>
</tr>
</tbody>
</table>
Arithmetic Operations

Table 15-2. Multiply 16-Bit Operands Options (Cont’d)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description for Register Half Destination</th>
<th>Description for 32-Bit Register Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T)</td>
<td>Signed fraction with truncation. Truncate Accumulator 9.31 format value at bit 16. (Perform no rounding.) Saturate the result to 1.15 precision in destination register half. Result is between minimum -1 and maximum 1-2^-15 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).</td>
<td>Not applicable. Truncation is meaningless for 32-bit register destinations.</td>
</tr>
<tr>
<td>(TFU)</td>
<td>Unsigned fraction with truncation. Multiply 1.15 * 1.15 to produce 1.31 results after left-shift correction. (Identical to Default.) Truncate 1.32 format value at bit 16. (Perform no rounding.) Saturate the result to 0.16 precision in destination register half. Result is between minimum 0 and maximum 1-2^-16 (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).</td>
<td>Not applicable.</td>
</tr>
<tr>
<td>(S2RND)</td>
<td>Signed fraction with scaling and rounding. Multiply 1.15 * 1.15 to produce 1.31 results after left-shift correction. (Identical to Default.) Shift the result one place to the left (multiply x 2). Round 1.31 format value at bit 16. (RND_MOD bit in the ASTAT register controls the rounding.) Saturate the result to 1.15 precision in destination register half. Result is between minimum -1 and maximum 1-2^-15 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).</td>
<td>Not applicable.</td>
</tr>
</tbody>
</table>
### Instruction Overview

Table 15-2. Multiply 16-Bit Operands Options (Cont’d)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description for Register Half Destination</th>
<th>Description for 32-Bit Register Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>(ISS2)</td>
<td>Signed integer with scaling. Multiply 16.0 * 16.0 to produce 32.0 results. No shift correction. Shift the lower 16 bits. Shift them one place to the left (multiply x 2). Saturate the result for 16.0 format in destination register half. Result is between minimum $2^{-15}$ and maximum $2^{15} - 1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).</td>
<td>Signed integer with scaling. Multiply 16.0 * 16.0 to produce 32.0 results. No shift correction. Shift the results one place to the left (multiply x 2). Saturate result to 32.0 format. Copy to destination register. Results range between minimum -1 and maximum $2^{31} - 1$. The resulting hexadecimal range is minimum 0x8000 0000 through maximum 0x7FFF FFFF.</td>
</tr>
<tr>
<td>(IH)</td>
<td>Signed integer, high word extract. Multiply 16.0 * 16.0 to produce 32.0 results. No shift correction. Round 32.0 format value at bit 16. (RND_MOD bit in the ASTAT register controls the rounding.) Saturate to 32.0 result. Extract the upper 16 bits of that value to the destination register half. Result is between minimum $2^{-15}$ and maximum $2^{15} - 1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).</td>
<td>Not applicable.</td>
</tr>
<tr>
<td>(M)</td>
<td>Mixed mode multiply (valid only for MAC1). When issued in a fraction mode instruction (with Default, FU, T, TFU, or S2RND mode), multiply 1.15 * 0.16 to produce 1.31 results. When issued in an integer mode instruction (with IS, ISS2, or IH mode), multiply 16.0 * 16.0 (signed * unsigned) to produce 32.0 results. No shift correction in either case. Src_reg_0 is the signed operand and Src_reg_1 is the unsigned operand. All other operations proceed according to the other mode flag or Default.</td>
<td></td>
</tr>
</tbody>
</table>

15-48 ADSP-BF53x/BF56x Blackfin Processor Programming Reference
Arithmetic Operations

Flags Affected

This instruction affects flags as follows.

- V is set if result saturates; cleared if no saturation.
- VS is set if V is set; unaffected otherwise.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

```
r3.l=r3.h*r2.h ; /* MAC0. Both operands are signed fractions. */
r3.h=r6.h*r4.l (fu) ; /* MAC1. Both operands are unsigned fractions. */
r6=r3.h*r4.h ; /* MAC0. Signed fraction operands, results saved as 32 bits. */
```
Instruction Overview

Also See

Multiply 32-Bit Operands, Multiply and Multiply-Accumulate to Accumulator, Multiply and Multiply-Accumulate to Half-Register, Multiply and Multiply-Accumulate to Data Register, Vector Multiply, Vector Multiply and Multiply-Accumulate

Special Applications

None
Arithmetic Operations

Multiply 32-Bit Operands

General Form

\[ \text{dest\_reg} *\text{= multiplier\_register} \]

Syntax

\[ \text{Dreg} *\text{= Dreg} ; /\ast \text{32 x 32 integer multiply (a) */} \]

Syntax Terminology

\[ \text{Dreg}: \text{R7-0} \]

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Multiply 32-Bit Operands instruction multiplies two 32-bit data registers (\text{dest\_reg} and \text{multiplier\_register}) and saves the product in \text{dest\_reg}. The instruction mimics multiplication in the C language and effectively performs \( \text{Dreg1} = (\text{Dreg1} \times \text{Dreg2}) \mod 2^{32} \). Since the integer multiply is modulo 2^{32}, the result always fits in a 32-bit \text{dest\_reg}, and overflows are possible but not detected. The overflow flag in the \text{ASTAT} register is never set.

Users are required to limit input numbers to ensure that the resulting product does not exceed the 32-bit \text{dest\_reg} capacity. If overflow notification is required, users should write their own multiplication macro with that capability.

Accumulators \text{A0} and \text{A1} are unchanged by this instruction.

The Multiply 32-Bit Operands instruction does not implicitly modify the number in \text{multiplier\_register}. 
**Instruction Overview**

This instruction might be used to implement the congruence method of random number generation according to:

\[ X[n + a] = (a \times X[n]) \mod 2^{32} \]

where:

- \( X[n] \) is the seed value,
- \( a \) is a large integer, and
- \( X[n+1] \) is the result that can be multiplied again to further the pseudo-random sequence.

**Flags Affected**

None

**Required Mode**

User & Supervisor

**Parallel Issue**

This instruction cannot be issued in parallel with any other instructions.

**Example**

\[ r3 \times= r0 ; \]

**Also See**

DIVS, DIVQ (Divide Primitive), Arithmetic Shift, Shift with Add, Add with Shift, Vector Multiply and Multiply-Accumulate, Vector Multiply

**Special Applications**

None
Arithmetic Operations

Multiply and Multiply-Accumulate to Accumulator

General Form

accumulator = src_reg_0 * src_reg_1 (opt_mode)
accumulator += src_reg_0 * src_reg_1 (opt_mode)
accumulator -= src_reg_0 * src_reg_1 (opt_mode)

Syntax

Multiply-And-Accumulate Unit 0 (MAC0) Operations

A0 = Dreg_lo_hi * Dreg_lo_hi (opt_mode) ; /* multiply and store (b) */
A0 += Dreg_lo_hi * Dreg_lo_hi (opt_mode) ; /* multiply and add (b) */
A0 -= Dreg_lo_hi * Dreg_lo_hi (opt_mode) ; /* multiply and subtract (b) */

Multiply-And-Accumulate Unit 1 (MAC1) Operations

A1 = Dreg_lo_hi * Dreg_lo_hi (opt_mode) ; /* multiply and store (b) */
A1 += Dreg_lo_hi * Dreg_lo_hi (opt_mode) ; /* multiply and add (b) */
A1 -= Dreg_lo_hi * Dreg_lo_hi (opt_mode) ; /* multiply and subtract (b) */

Syntax Terminology

Dreg_lo_hi: R7-0.L, R7-0.H

opt_mode: Optionally (FU), (IS), or (W32). Optionally, (M) can be used on MAC1 versions either alone or with (W32). If multiple options are specified together for a MAC, the options must be separated by commas and enclosed within a single set of parenthesis. Example: (M, W32)
Instruction Overview

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Multiply and Multiply-Accumulate to Accumulator instruction multiplies two 16-bit half-word operands. It stores, adds or subtracts the product into a designated Accumulator with saturation.

The Multiply-and-Accumulate Unit 0 (MAC0) portion of the architecture performs operations that involve Accumulator A0. MAC1 performs A1 operations.

By default, the instruction treats both operands of both MACs as signed fractions with left-shift correction as required.

Options

The Multiply and Multiply-Accumulate to Accumulator instruction supports the following options. Saturation is supported for every option.

When the (M) and (W32) options are used together, both MACs saturate their Accumulator products at 32 bits. MAC1 multiplies signed fractions by unsigned fractions and MAC0 multiplies signed fractions.

When used together, the order of the options in the syntax makes no difference.

In fractional mode, the product of the most negative representable fraction times itself (for example, 0x8000 times 0x8000) is saturated to the maximum representable positive fraction (0x7FFF) before accumulation.

See “Saturation” on page 1-17 for a description of saturation behavior.
## Arithmetic Operations

Table 15-3. Multiply and Multiply-Accumulate to Accumulator Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>Signed fraction. Multiply 1.15 x 1.15 to produce 1.31 format data after shift correction. Sign extend the result to 9.31 format before passing it to the Accumulator. Saturate the Accumulator after copying or accumulating to maintain 9.31 precision. Result is between minimum -1 and maximum 1-2^-31 (or, expressed in hex, between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF).</td>
</tr>
<tr>
<td>(FU)</td>
<td>Unsigned fraction. Multiply 0.16 x 0.16 to produce 0.32 format data. Perform no shift correction. Zero extend the result to 8.32 format before passing it to the Accumulator. Saturate the Accumulator after copying or accumulating to maintain 8.32 precision. Unsigned integer. Multiply 16.0 x 16.0 to produce 32.0 format data. Perform no shift correction. Zero extend the result to 40.0 format before passing it to the Accumulator. Saturate the Accumulator after copying or accumulating to maintain 40.0 precision. In either case, the resulting hexadecimal range is minimum 0x00 0000 0000 through maximum 0xFF FFFF FFFF.</td>
</tr>
<tr>
<td>(IS)</td>
<td>Signed integer. Multiply 16.0 x 16.0 to produce 32.0 format data. Perform no shift correction. Sign extend the result to 40.0 format before passing it to the Accumulator. Saturate the Accumulator after copying or accumulating to maintain 40.0 precision. Result is between minimum -2^39 and maximum 2^39-1 (or, expressed in hex, between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF).</td>
</tr>
<tr>
<td>(W32)</td>
<td>Signed fraction with 32-bit saturation. Multiply 1.15 x 1.15 to produce 1.31 format data after shift correction. Sign extend the result to 9.31 format before passing it to the Accumulator. Saturate the Accumulator after copying or accumulating at bit 31 to maintain 1.31 precision. Result is between minimum -1 and maximum 1-2^-31 (or, expressed in hex, between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF).</td>
</tr>
<tr>
<td>(M)</td>
<td>Mixed mode multiply (valid only for MAC1). When issued in a fraction mode instruction (with Default, FU, T, TFU, or S2RND mode), multiply 1.15 * 0.16 to produce 1.31 results. When issued in an integer mode instruction (with IS, ISS2, or IH mode), multiply 16.0 * 16.0 (signed * unsigned) to produce 32.0 results. No shift correction in either case. Src_reg_0 is the signed operand and Src_reg_1 is the unsigned operand. Accumulation and extraction proceed according to the other mode flag or Default.</td>
</tr>
</tbody>
</table>
Instruction Overview

Flags Affected

This instruction affects flags as follows.

- \( AV0 \) is set if result in Accumulator \( A0 \) (MAC0 operation) saturates; cleared if \( A0 \) result does not saturate.
- \( AV0S \) is set if \( AV0 \) is set; unaffected otherwise.
- \( AV1 \) is set if result in Accumulator \( A1 \) (MAC1 operation) saturates; cleared if \( A1 \) result does not saturate.
- \( AV1S \) is set if \( AV1 \) is set; unaffected otherwise.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer \( ASTAT \) flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

\[
\begin{align*}
a0 &= r3.h \times r2.h ; \quad /* \text{MAC0, only. Both operands are signed fractions. Load the product into } A0. */ \\
a1 &= r6.h \times r4.l (\text{fu}) ; \quad /* \text{MAC1, only. Both operands are unsigned fractions. Accumulate into } A1 */
\end{align*}
\]
Arithmetic Operations

Also See

Multiply 16-Bit Operands, Multiply 32-Bit Operands, Multiply and Multiply-Accumulate to Half-Register, Multiply and Multiply-Accumulate to Data Register, Vector Multiply, Vector Multiply and Multiply-Accumulate

Special Applications

DSP filter applications often use the Multiply and Multiply-Accumulate to Accumulator instruction to calculate the dot product between two signal vectors.
Instruction Overview

Multiply and Multiply-Accumulate to Half-Register

General Form

\[
\text{dest\_reg\_half} = (\text{accumulator} = \text{src\_reg\_0} \ast \text{src\_reg\_1}) \text{(opt\_mode)} \]

\[
\text{dest\_reg\_half} = (\text{accumulator} += \text{src\_reg\_0} \ast \text{src\_reg\_1}) \text{(opt\_mode)} \]

\[
\text{dest\_reg\_half} = (\text{accumulator} -= \text{src\_reg\_0} \ast \text{src\_reg\_1}) \text{(opt\_mode)} \]

Syntax

Multiply-And-Accumulate Unit 0 (MAC0)

\[
Dreg\_lo = (A0 = Dreg\_lo\_hi \ast Dreg\_lo\_hi) \text{(opt\_mode)} ; /* multiply and store (b) */
\]

\[
Dreg\_lo = (A0 += Dreg\_lo\_hi \ast Dreg\_lo\_hi) \text{(opt\_mode)} ; /* multiply and add (b) */
\]

\[
Dreg\_lo = (A0 -= Dreg\_lo\_hi \ast Dreg\_lo\_hi) \text{(opt\_mode)} ; /* multiply and subtract (b) */
\]

Multiply-And-Accumulate Unit 1 (MAC1)

\[
Dreg\_hi = (A1 = Dreg\_lo\_hi \ast Dreg\_lo\_hi) \text{(opt\_mode)} ; /* multiply and store (b) */
\]

\[
Dreg\_hi = (A1 += Dreg\_lo\_hi \ast Dreg\_lo\_hi) \text{(opt\_mode)} ; /* multiply and add (b) */
\]

\[
Dreg\_hi = (A1 -= Dreg\_lo\_hi \ast Dreg\_lo\_hi) \text{(opt\_mode)} ; /* multiply and subtract (b) */
\]

Syntax Terminology

\[Dreg\_lo\_hi: R7-0.L, R7-0.H\]

\[Dreg\_lo: R7-0.L\]

\[Dreg\_hi: R7-0.H\]
Arithmetic Operations

**opt_mode:** Optionally (FU), (IS), (IU), (T), (TFU), (S2RND), (ISS2) or (IH). Optionally, (M) can be used with MAC1 versions either alone or with any of these other options. If multiple options are specified together for a MAC, the options must be separated by commas and enclosed within a single set of parentheses. Example: (M, TFU)

### Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

### Functional Description

The Multiply and Multiply-Accumulate to Half-Register instruction multiplies two 16-bit half-word operands. The instruction stores, adds or subtracts the product into a designated Accumulator. It then copies 16 bits (saturated at 16 bits) of the Accumulator into a data half-register.

The fraction versions of this instruction (the default and “(FU)” options) transfer the Accumulator result to the destination register according to the diagrams in Figure 15-1.

The integer versions of this instruction (the “(IS)” and “(IU)” options) transfer the Accumulator result to the destination register according to the diagrams in Figure 15-2.

The Multiply-and-Accumulate Unit 0 (MAC0) portion of the architecture performs operations that involve Accumulator A0 and loads the results into the lower half of the destination data register. MAC1 performs A1 operations and loads the results into the upper half of the destination data register.

All versions of this instruction that support rounding are affected by the RND_MOD bit in the ASTAT register when they copy the results into the destination register. RND_MOD determines whether biased or unbiased rounding is used.
Instruction Overview

Figure 15-1. Result to Destination Register (Default and (FU) Options)

Figure 15-2. Result to Destination Register ((IS) and (IU) Options)

15-60        ADSP-BF53x/BF56x Blackfin Processor Programming Reference
Arithmetic Operations

See “Rounding and Truncating” on page 1-19 for a description of rounding behavior.

Options

The Multiply and Multiply-Acumulate to Half-Register instruction supports operand and Accumulator copy options.

The options are listed in Table 15-4.

Table 15-4. Multiply and Multiply-Acumulate to Half-Register Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>Signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 9.31 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To extract to half-register, round Accumulator 9.31 format value at bit 16. (RND_MOD bit in the ASTAT register controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum 1-2^{-15} (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).</td>
</tr>
<tr>
<td>(FU)</td>
<td>Unsigned fraction format. Multiply 0.16 * 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 8.32 precision; Accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To extract to half-register, round Accumulator 8.32 format value at bit 16. (RND_MOD bit in the ASTAT register controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum 1-2^{-16} (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).</td>
</tr>
</tbody>
</table>
### Instruction Overview

Table 15-4. Multiply and Multiply-Accumulate to Half-Register Options (Cont’d)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
</table>
| (IS)   | Signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction.  
Sign extend 32.0 result to 40.0 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 40.0 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF.  
Extract the lower 16 bits of the Accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum 2^{15}-1 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). |
| (IU)   | Unsigned integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction.  
Zero extend 32.0 result to 40.0 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 40.0 precision; Accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF.  
Extract the lower 16 bits of the Accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum 0 and maximum 2^{16}-1 (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). |
| (T)    | Signed fraction with truncation. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the Default mode.)  
Sign extend 1.31 result to 9.31 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 9.31 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF.  
To extract to half-register, truncate Accumulator 9.31 format value at bit 16. (Perform no rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum 1-2^{-15} (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). |
Arithmetic Operations

Table 15-4. Multiply and Multiply-Accumulate to Half-Register Options (Cont’d)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(TFU)</td>
<td><strong>Unsigned fraction with truncation.</strong> Multiply 0.16 * 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. (Same as the FU mode.) Zero extend 0.32 result to 8.32 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 8.32 precision; Accumulator result is between minimum 0x00 0000 0000 and maximum 0xFFFF FFFF. To extract to half-register, truncate Accumulator 8.32 format value at bit 16. (Perform no rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum 1-2^{-16} (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).</td>
</tr>
<tr>
<td>(S2RND)</td>
<td><strong>Signed fraction with scaling and rounding.</strong> Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the Default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 9.31 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To extract to half-register, shift the Accumulator contents one place to the left (multiply x 2). Round Accumulator 9.31 format value at bit 16. (RND_MOD bit in the ASTAT register controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum 1-2^{-15} (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).</td>
</tr>
<tr>
<td>(ISS2)</td>
<td><strong>Signed integer with scaling.</strong> Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 40.0 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. Extract the lower 16 bits of the Accumulator. Shift them one place to the left (multiply x 2). Saturate the result for 16.0 format and copy to the destination register half. Result is between minimum -2^{15} and maximum 2^{15-1} (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).</td>
</tr>
</tbody>
</table>
Table 15-4. Multiply and Multiply-Accumulate to Half-Register Options (Cont’d)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(IH)</td>
<td>Signed integer, high word extract. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 32.0 precision; Accumulator result is between minimum 0x00 8000 0000 and maximum 0x00 7FFF FFFF. To extract to half-register, round Accumulator 40.0 format value at bit 16. (RND_MOD bit in the ASTAT register controls the rounding.) Saturate to 32.0 result. Copy the upper 16 bits of that value to the destination register half. Result is between minimum -2^{15} and maximum 2^{15}-1 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).</td>
</tr>
<tr>
<td>(M)</td>
<td>Mixed mode multiply (valid only for MAC1). When issued in a fraction mode instruction (with Default, FU, T, TFU, or S2RND mode), multiply 1.15 * 0.16 to produce 1.31 results. When issued in an integer mode instruction (with IS, ISS2, or IH mode), multiply 16.0 * 16.0 (signed * unsigned) to produce 32.0 results. No shift correction in either case. Src_reg_0 is the signed operand and Src_reg_1 is the unsigned operand. Accumulation and extraction proceed according to the other mode flag or Default.</td>
</tr>
</tbody>
</table>

To truncate the result, the operation eliminates the least significant bits that do not fit into the destination register.

When necessary, saturation is performed after the rounding.

The accumulator is unaffected by extraction.

If you want to keep the unaltered contents of the Accumulator, use a simple Move instruction to copy An.X or An.W to or from a register.

See “Saturation” on page 1-17 for a description of saturation behavior.
Arithmetic Operations

Flags Affected

This instruction affects flags as follows.

- $V$ is set if the result extracted to the $Dreg$ saturates; cleared if no saturation.
- $VS$ is set if $V$ is set; unaffected otherwise.
- $AV0$ is set if result in Accumulator $A0$ (MAC0 operation) saturates; cleared if $A0$ result does not saturate.
- $AV0S$ is set if $AV0$ is set; unaffected otherwise.
- $AV1$ is set if result in Accumulator $A1$ (MAC1 operation) saturates; cleared if $A1$ result does not saturate.
- $AV1S$ is set if $AV1$ is set; unaffected otherwise.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer $ASTAT$ flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.
Instruction Overview

Example

```
r3.l=(a0=r3.h*r2.h) ; /* MACO, only. Both operands are signed fractions. Load the product into A0, then copy to r3.l. */
r3.h=(a1+=r6.h*r4.l) (fu) ; /* MAC1, only. Both operands are unsigned fractions. Add the product into A1, then copy to r3.h */
```

Also See

Multiply 32-Bit Operands, Multiply and Multiply-Accumulate to Accumulator, Multiply and Multiply-Accumulate to Data Register, Vector Multiply, Vector Multiply and Multiply-Accumulate

Special Applications

DSP filter applications often use the Multiply and Multiply-Accumulate to Half-Register instruction to calculate the dot product between two signal vectors.
## Arithmetic Operations

### Multiply and Multiply-Accumulate to Data Register

**General Form**

\[
\text{dest\_reg} = (\text{accumulator} = \text{src\_reg\_0} \times \text{src\_reg\_1}) \text{ (opt\_mode)} \\
\text{dest\_reg} = (\text{accumulator} += \text{src\_reg\_0} \times \text{src\_reg\_1}) \text{ (opt\_mode)} \\
\text{dest\_reg} = (\text{accumulator} -= \text{src\_reg\_0} \times \text{src\_reg\_1}) \text{ (opt\_mode)}
\]

**Syntax**

**Multiply-And-Accumulate Unit 0 (MAC0)**

\[
\text{Dreg\_even} = (\text{A0} = \text{Dreg\_lo\_hi} \times \text{Dreg\_lo\_hi}) \text{ (opt\_mode)} \\
\text{Dreg\_even} = (\text{A0} += \text{Dreg\_lo\_hi} \times \text{Dreg\_lo\_hi}) \text{ (opt\_mode)} \\
\text{Dreg\_even} = (\text{A0} -= \text{Dreg\_lo\_hi} \times \text{Dreg\_lo\_hi}) \text{ (opt\_mode)}
\]

**Multiply-And-Accumulate Unit 1 (MAC1)**

\[
\text{Dreg\_odd} = (\text{A1} = \text{Dreg\_lo\_hi} \times \text{Dreg\_lo\_hi}) \text{ (opt\_mode)} \\
\text{Dreg\_odd} = (\text{A1} += \text{Dreg\_lo\_hi} \times \text{Dreg\_lo\_hi}) \text{ (opt\_mode)} \\
\text{Dreg\_odd} = (\text{A1} -= \text{Dreg\_lo\_hi} \times \text{Dreg\_lo\_hi}) \text{ (opt\_mode)}
\]

**Syntax Terminology**

\[
\text{Dreg\_lo\_hi}: \text{R7}\_0.L, \text{R7}\_0.H \\
\text{Dreg\_even}: \text{R0}, \text{R2}, \text{R4}, \text{R6} \\
\text{Dreg\_odd}: \text{R1}, \text{R3}, \text{R5}, \text{R7}
\]
Instruction Overview

opt_mode: Optionally (FU), (IS), (S2RND), or (ISS2). Optionally, (M) can be used with MAC1 versions either alone or with any of these other options. If multiple options are specified together for a MAC, the options must be separated by commas and enclosed within a single set of parenthesis. Example: (M, IS)

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

This instruction multiplies two 16-bit half-word operands. The instruction stores, adds or subtracts the product into a designated Accumulator. It then copies 32 bits of the Accumulator into a data register. The 32 bits are saturated at 32 bits.

The Multiply-and-Accumulate Unit 0 (MAC0) portion of the architecture performs operations that involve Accumulator A0; it loads the results into an even-numbered data register. MAC1 performs A1 operations and loads the results into an odd-numbered data register.

Combinations of these instructions can be combined into a single instruction. See “Vector Multiply and Multiply-Accumulate” on page 19-41.

Options

The Multiply and Multiply-Accumulate to Data Register instruction supports operand and Accumulator copy options.

These options are as shown in Table 15-5.

The syntax supports only biased rounding. The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction.

See “Rounding and Truncating” on page 1-19 for a description of rounding behavior.
Arithmetic Operations

Table 15-5. Multiply and Multiply-Accumulate to Data Register Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>Signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 9.31 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To extract, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum 1-2-31 (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).</td>
</tr>
<tr>
<td>(FU)</td>
<td>Unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 8.32 precision; Accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To extract, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum 1-2-32 (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).</td>
</tr>
<tr>
<td>(IS)</td>
<td>Signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 40.0 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To extract, saturate for 32.0 precision and copy to the destination register. Result is between minimum -231 and maximum 231-1 (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).</td>
</tr>
<tr>
<td>(S2RND)</td>
<td>Signed fraction with scaling and rounding. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the Default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 9.31 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To extract, shift the Accumulator contents one place to the left (multiply x 2), saturate the result to 1.31 precision, and copy it to the destination register. Result is between minimum -1 and maximum 1-2-31 (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).</td>
</tr>
</tbody>
</table>
The accumulator is unaffected by extraction.

In fractional mode, the product of the most negative representable fraction times itself (for example, 0x8000 times 0x8000) is saturated to the maximum representable positive fraction (0x7FFF) before accumulation.

If you want to keep the unaltered contents of the Accumulator, use a simple Move instruction to copy An.X or An.W to or from a register.

See “Saturation” on page 1-17 for a description of saturation behavior.
Arithmetic Operations

Flags Affected

This instruction affects flags as follows.

- $V$ is set if the result extracted to the Dreg saturates; cleared if no saturation.
- $V_S$ is set if $V$ is set; unaffected otherwise.
- $A_{V0}$ is set if result in Accumulator A0 (MAC0 operation) saturates; cleared if A0 result does not saturate.
- $A_{V0S}$ is set if $A_{V0}$ is set; unaffected otherwise.
- $A_{V1}$ is set if result in Accumulator A1 (MAC1 operation) saturates; cleared if A1 result does not saturate.
- $A_{V1S}$ is set if $A_{V1}$ is set; unaffected otherwise.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.
Instruction Overview

Example

\[
\begin{align*}
r_4 &= (a_0 = r_3.h \times r_2.h) ; \quad /* \text{MAC0, only. Both operands are signed fractions. Load the product into A0, then into } r_4. */ \\
r_3 &= (a_1 += r_6.h \times r_4.l) \ (fu) ; \quad /* \text{MAC1, only. Both operands are unsigned fractions. Add the product into A1, then into } r_3. */
\end{align*}
\]

Also See

Move Register, Move Register Half, Multiply 32-Bit Operands, Multiply and Multiply-Accumulate to Accumulator, Multiply and Multiply-Accumulate to Half-Register, Vector Multiply, Vector Multiply and Multiply-Accumulate

Special Applications

DSP filter applications often use the Multiply and Multiply-Accumulate to Data Register instruction or the vector version (“Vector Multiply and Multiply-Accumulate” on page 19-41) to calculate the dot product between two signal vectors.
Arithmetic Operations

Negate (Two's-Complement)

General Form

\[ \text{dest\_reg} = - \text{src\_reg} \]
\[ \text{dest\_accumulator} = - \text{src\_accumulator} \]

Syntax

\[ D\text{reg} = - D\text{reg} ; \quad /\ast \text{ (a) } /\ast \]
\[ D\text{reg} = - D\text{reg} (\text{sat\_flag}) ; \quad /\ast \text{ (b) } /\ast \]
\[ A0 = - A0 ; \quad /\ast \text{ (b) } /\ast \]
\[ A0 = - A1 ; \quad /\ast \text{ (b) } /\ast \]
\[ A1 = - A0 ; \quad /\ast \text{ (b) } /\ast \]
\[ A1 = - A1 ; \quad /\ast \text{ (b) } /\ast \]
\[ A1 = - A1, A0 = - A0 ; \quad /\ast \text{ negate both Accumulators simultaneously in one 32-bit length instruction (b) } /\ast \]

Syntax Terminology

\[ D\text{reg}: R7-0 \]
\[ \text{sat\_flag}: \text{nonoptional saturation flag, (S) or (NS)} \]

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

Functional Description

The Negate (Two’s-Complement) instruction returns the same magnitude with the opposite arithmetic sign. The Accumulator versions saturate the result at 40 bits. The instruction calculates by subtracting from zero.
Instruction Overview

The Dreg version of the Negate (Two’s-Complement) instruction is offered with or without saturation. The only case where the nonsaturating Negate would overflow is when the input value is 0x8000 0000. The saturating version returns 0x7FFF FFFF; the nonsaturating version returns 0x8000 0000.

In the syntax, where sat_flag appears, substitute one of the following values.

- (S) saturate the result
- (NS) no saturation

See “Saturation” on page 1-17 for a description of saturation behavior.

Flags Affected

This instruction affects the flags as follows.

- AZ is set if result is zero; cleared if nonzero.
- AN is set if result is negative; cleared if non-negative.
- V is set if result overflows or saturates and the dest_reg is a Dreg; cleared if no overflow or saturation.
- VS is set if V is set; unaffected otherwise.
- AV0 is set if result saturates and the dest_reg is A0; cleared if no saturation.
- AV0S is set if AV0 is set; unaffected otherwise.
- AV1 is set if result saturates and the dest_reg is A1; cleared if no saturation.
- AV1S is set if AV1 is set; unaffected otherwise.
Arithmetic Operations

- AC0 is set if src_reg is zero; otherwise it is cleared.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode
User & Supervisor

Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

The 16-bit versions of this instruction cannot be issued in parallel with other instructions.

Example

r5 = -r0 ;
a0 = -a0 ;
a0 = -a1 ;
a1 = -a0 ;
a1 = -a1 ;
a1 = -a1, a0=-a0 ;
Instruction Overview

Also See

Vector Negate (Two’s-Complement)

Special Applications

None
Arithmetic Operations

RND (Round to Half-Word)

General Form

\[ \text{dest}_\text{reg} = \text{src}_\text{reg} (\text{RND}) \]

Syntax

\[ \text{Dreg}\_\text{lo}\_\text{hi} = \text{Dreg} (\text{RND}) ; \quad /* \text{round and saturate the source to} \]
\[ \text{16 bits. (b) */} \]

Syntax Terminology

- \text{Dreg}: R7–0
- \text{Dreg}\_\text{lo}\_\text{hi}: R7–0.L, R7–0.H

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Round to Half-Word instruction rounds a 32-bit, normalized-fraction number into a 16-bit, normalized-fraction number by extracting and saturating bits 31–16, then discarding bits 15–0. The instruction supports only biased rounding, which adds a half LSB (in this case, bit 15) before truncating bits 15–0. The ALU performs the rounding. The \text{RND\_MOD} bit in the \text{ASTAT} register has no bearing on the rounding behavior of this instruction.

Fractional data types such as the operands used in this instruction are always signed.

See “Saturation” on page 1-17 for a description of saturation behavior.

See “Rounding and Truncating” on page 1-19 for a description of rounding behavior.
Instruction Overview

Flags Affected

The following flags are affected by this instruction.

- **AZ** is set if result is zero; cleared if nonzero.
- **AN** is set if result is negative; cleared if non-negative.
- **V** is set if result saturates; cleared if no saturation.
- **VS** is set if **V** is set; unaffected otherwise.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer `ASTAT` flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

```c
/* If r6 = 0xFFFC FFFF, then rounding to 16-bits with . . . */
r1.l = r6 (rnd);  // . . . produces r1.l = 0xFFFFD
// If r7 = 0x0001 8000, then rounding . . .
r1.h = r7 (rnd);  // . . . produces r1.h = 0x0002
```
Arithmetic Operations

Also See

Add, Add/Subtract – Prescale Up, Add/Subtract – Prescale Down

Special Applications

None
Instruction Overview

Saturate

General Form

\[
\text{dest\_reg} = \text{src\_reg} (S)
\]

Syntax

\[
\begin{align*}
A0 &= A0 (S) ; /* (b) */ \\
A1 &= A1 (S) ; /* (b) */ \\
A1 &= A1 (S), A0 &= A0 (S) ; /* signed saturate both Accumulators at the 32-bit boundary (b) */
\end{align*}
\]

Syntax Terminology

None

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Saturate instruction saturates the 40-bit Accumulators at 32 bits. The resulting saturated value is sign extended into the Accumulator extension bits.

See “Saturation” on page 1-17 for a description of saturation behavior.
Arithmetic Operations

Flags Affected

This instruction affects flags as follows.

- $A_{Z}$ is set if result is zero; cleared if nonzero. In the case of two simultaneous operations, $A_{Z}$ represents the logical “OR” of the two.

- $A_{N}$ is set if result is negative; cleared if non-negative. In the case of two simultaneous operations, $A_{N}$ represents the logical “OR” of the two.

- $A_{V0}$ is set if result saturates and the $dest_{reg}$ is $A_{0}$; cleared if no overflow.

- $A_{V0S}$ is set if $A_{V0}$ is set; unaffected otherwise.

- $A_{V1}$ is set if result saturates and the $dest_{reg}$ is $A_{1}$; cleared if no overflow.

- $A_{V1S}$ is set if $A_{V1}$ is set; unaffected otherwise.

- All other flags are unaffected.

The ADSP-BF535 processor has fewer $ASTAT$ flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see "Issuing Parallel Instructions" on page 20-1.
Instruction Overview

Example

\[ a0 = a0 (s) ; \]
\[ a1 = a1 (s) ; \]
\[ a1 = a1 (s), a0 = a0 (s) ; \]

Also See

Subtract (saturate options), Add (saturate options)

Special Applications

None
Arithmetic Operations

SIGNBITS

General Form

\[ \text{dest\_reg} = \text{SIGNBITS sample\_register} \]

Syntax

\[
\begin{align*}
Dreg\_lo &= \text{SIGNBITS} \ Dreg ; \quad /* \text{32-bit sample (b)} */ \\
Dreg\_lo &= \text{SIGNBITS} \ Dreg\_lo\_hi ; \quad /* \text{16-bit sample (b)} */ \\
Dreg\_lo &= \text{SIGNBITS} \ A0 ; \quad /* \text{40-bit sample (b)} */ \\
Dreg\_lo &= \text{SIGNBITS} \ A1 ; \quad /* \text{40-bit sample (b)} */
\end{align*}
\]

Syntax Terminology

\[
\begin{align*}
Dreg &: R7-0 \\
Dreg\_lo &: R7-0.L \\
Dreg\_lo\_hi &: R7-0.L, R7-0.H
\end{align*}
\]

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Sign Bit instruction returns the number of sign bits in a number, and can be used in conjunction with a shift to normalize numbers. This instruction can operate on 16-bit, 32-bit, or 40-bit input numbers.

- For a 16-bit input, Sign Bit returns the number of leading sign bits minus one, which is in the range 0 through 15. There are no special cases. An input of all zeros returns +15 (all sign bits), and an input of all ones also returns +15.
Instruction Overview

- For a 32-bit input, Sign Bit returns the number of leading sign bits minus one, which is in the range 0 through 31. An input of all zeros or all ones returns +31 (all sign bits).

- For a 40-bit Accumulator input, Sign Bit returns the number of leading sign bits minus 9, which is in the range –8 through +31. A negative number is returned when the result in the Accumulator has expanded into the extension bits; the corresponding normalization will shift the result down to a 32-bit quantity (losing precision). An input of all zeros or all ones returns +31.

The result of the `SIGNBITS` instruction can be used directly as the argument to `ASHIFT` to normalize the number. Resultant numbers will be in the following formats (S == signbit, M == magnitude bit).

16-bit: S.MM MMMM MMMM MMMM
32-bit: S.MM MMMM MMMM MMMM MMMM MMMM MMMM MMMM
40-bit: SSSS SSSS S.MM MMMM MMMM MMMM MMMM MMMM MMMM MMMM MMMM

In addition, the `SIGNBITS` instruction result can be subtracted directly to form the new exponent.

The Sign Bit instruction does not implicitly modify the input value. For 32-bit and 16-bit input, the dest_reg and sample_register can be the same D-register. Doing this explicitly modifies the sample_register.

Flags Affected

None

The ADSP-BF535 processor has fewer `ASTAT` flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.
Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

\[
\begin{align*}
  r2.l &= \text{signbits } r7 ; \\
  r1.l &= \text{signbits } r5.l ; \\
  r0.l &= \text{signbits } r4.h ; \\
  r6.l &= \text{signbits } a0 ; \\
  r5.l &= \text{signbits } a1 ;
\end{align*}
\]

Also See

EXPADJ

Special Applications

You can use the exponent as shift magnitude for array normalization. You can accomplish normalization by using the ASHIFT instruction directly, without using special normalizing instructions, as required on other architectures.
Instruction Overview

Subtract

General Form

\[ \text{dest\_reg} = \text{src\_reg\_1} - \text{src\_reg\_2} \]

Syntax

32-Bit Operands, 32-Bit Result

\[ \text{Dreg} = \text{Dreg} - \text{Dreg}; \quad /* \text{no saturation support but shorter} \]
\[ \text{instruction length (a) */} \]
\[ \text{Dreg} = \text{Dreg} - \text{Dreg (sat\_flag)}; \quad /* \text{saturation optionally sup-} \]
\[ \text{ported, but at the cost of longer instruction length (b) */} \]

16-Bit Operands, 16-Bit Result

\[ \text{Dreg\_lo\_hi} = \text{Dreg\_lo\_hi} - \text{Dreg\_lo\_hi (sat\_flag)}; /* (b) */ \]

Syntax Terminology

\[ \text{Dreg}: \text{R7-0} \]
\[ \text{Dreg\_lo\_hi}: \text{R7-0.L, R7-0.H} \]
\[ \text{sat\_flag}: \text{nonoptional saturation flag, (S) or (NS)} \]

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

Functional Description

The Subtract instruction subtracts src\_reg\_2 from src\_reg\_1 and places the result in a destination register.
Arithmetic Operations

There are two ways to specify subtraction on 32-bit data. One instruction that is 16-bit instruction length does not support saturation. The other instruction, which is 32-bit instruction length, optionally supports saturation. The larger DSP instruction can sometimes save execution time because it can be issued in parallel with certain other instructions. See “Parallel Issue” on page 15-5.

The instructions for 16-bit data use half-word data register operands and store the result in a half-word data register.

All the instructions for 16-bit data are 32-bit instruction length.

In the syntax, where sat_flag appears, substitute one of the following values.

- $(S)$ saturate the result
- $(NS)$ no saturation

See “Saturation” on page 1-17 for a description of saturation behavior.

The Subtract instruction has no subtraction equivalent of the addition syntax for P-registers.

Flags Affected

This instruction affects flags as follows.

- $AZ$ is set if result is zero; cleared if nonzero.
- $AN$ is set if result is negative; cleared if non-negative.
- $AC0$ is set if the operation generates a carry; cleared if no carry.
- $V$ is set if result overflows; cleared if no overflow.
Instruction Overview

- Vs is set if V is set; unaffected otherwise.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

The 16-bit versions of this instruction cannot be issued in parallel with other instructions.

Example

```
r5 = r2 - r1 ;  /* 16-bit instruction length subtract, no saturation */
r5 = r2 - r1(ns) ;  /* same result as above, but 32-bit instruction length */
r5 = r2 - r1(s) ;  /* saturate the result */
r4.l = r0.1 - r7.l (ns) ;
r4.l = r0.1 - r7.h (s) ;  /* saturate the result */
r0.l = r2.h - r4.l(ns) ;
r1.l = r3.h - r7.h(ns) ;
r4.h = r0.1 - r7.l (ns) ;
r4.h = r0.1 - r7.h (ns) ;
r0.h = r2.h - r4.l(ns) ;  /* saturate the result */
r1.h = r3.h - r7.h(ns) ;
```
Arithmetic Operations

Also See

   Modify – Decrement, Vector Add / Subtract

Special Applications

   None
Instruction Overview

Subtract Immediate

General Form

register -= constant

Syntax

\[ Ireg -= 2 ; \quad /* \text{decrement } Ireg \text{ by } 2, \text{half-word address pointer increment (a)} */ \]
\[ Ireg -= 4 ; \quad /* \text{word address pointer decrement (a)} */ \]

Syntax Terminology

\( Ireg: 13-0 \)

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Subtract Immediate instruction subtracts a constant value from an Index register without saturation.

The instruction versions that explicitly modify \( Ireg \) support optional circular buffering. See “Automatic Circular Addressing” on page 1-21 for more details. Unless circular buffering is desired, disable it prior to issuing this instruction by clearing the Length Register (\( Lreg \)) corresponding to the \( Ireg \) used in this instruction.
Example: If you use $I_2$ to increment your address pointer, first clear $L_2$ to disable circular buffering. Failure to explicitly clear $Lreg$ beforehand can result in unexpected $Ireg$ values.

The circular address buffer registers (Index, Length, and Base) are not initialized automatically by Reset. Traditionally, user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes them later, if needed.

To subtract immediate values from D-registers or P-registers, use a negative constant in the Add Immediate instruction.

Flags Affected

None

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

```c
i0 -= 4 ;
i2 -= 2 ;
```

Also See

Add Immediate, Subtract

Special Applications

None
Instruction Overview
16 EXTERNAL EVENT MANAGEMENT

Instruction Summary

- “Idle” on page 16-3
- “Core Synchronize” on page 16-5
- “System Synchronize” on page 16-8
- “EMUEXCPT (Force Emulation)” on page 16-11
- “Disable Interrupts” on page 16-13
- “Enable Interrupts” on page 16-15
- “RAISE (Force Interrupt / Reset)” on page 16-17
- “EXCPT (Force Exception)” on page 16-20
- “Test and Set Byte (Atomic)” on page 16-22
- “No Op” on page 16-25
Instruction Overview

This chapter discusses the instructions that manage external events. Users can take advantage of these instructions to enable interrupts, force a specific interrupt or reset to occur, or put the processor in idle state. The Core Synchronize instruction resolves all pending operations and flushes the core store buffer before proceeding to the next instruction. The System Synchronize instruction forces all speculative, transient states in the core and system to complete before processing continues. Other instructions in this chapter force an emulation exception, placing the processor in Emulation mode; test the value of a specific, indirectly-addressed byte; or increment the Program Counter (PC) without performing useful work.
External Event Management

Idle

General Form

IDLE

Syntax

IDLE : /* (a) */

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

Typically, the Idle instruction is part of a sequence to place the Blackfin processor in a quiescent state so that the external system can switch between core clock frequencies.

The IDLE instruction requests an idle state by setting the idle_req bit in SEQSTAT register. Setting the idle_req bit precedes placing the Blackfin processor in a quiescent state. If you intend to place the processor in Idle mode, the IDLE instruction must immediately precede an SSYNC instruction.

The first instruction following the SSYNC is the first instruction to execute when the processor recovers from Idle mode.

The Idle instruction is the only way to set the idle_req bit in SEQSTAT. The architecture does not support explicit writes to SEQSTAT.

Flags Affected

None
Instruction Overview

Required Mode

The Idle instruction executes only in Supervisor mode. If execution is attempted in User mode, the instruction produces an Illegal Use of Protected Resource exception.

Parallel Issue

This instruction cannot be issued in parallel with other instructions.

Example

\texttt{idle ;}

Also See

System Synchronize

Special Applications

None
External Event Management

Core Synchronize

General Form

CSYNC

Syntax

CSYNC ;  /* (a) */

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Core Synchronize (CSYNC) instruction ensures resolution of all pending core operations and the flushing of the core store buffer before proceeding to the next instruction. Pending core operations include any speculative states (for example, branch prediction) or exceptions. The core store buffer lies between the processor and the L1 cache memory.

CCSYNC is typically used after core MMR writes to prevent imprecise behavior.

Flags Affected

None

Required Mode

User & Supervisor
Instruction Overview

Parallel Issue

The Core Synchronize instruction cannot be issued in parallel with other instructions.

Example

Consider the following example code sequence.

```c
if cc jump away_from_here ;  /* produces speculative branch prediction */
    csync ;
    r0 = [p0] ;  /* load */
```

In this example, the `CSYNC` instruction ensures that the load instruction is not executed speculatively. `CSYNC` ensures that the conditional branch is resolved and any entries in the processor store buffer have been flushed. In addition, all speculative states or exceptions complete processing before `CSYNC` completes.

Also See

`System Synchronize`

Special Applications

Use `CSYNC` to enforce a strict execution sequence on loads and stores or to conclude all transitional core states before reconfiguring the core modes. For example, issue `CSYNC` before configuring memory-mapped registers (MMRs). `CSYNC` should also be issued after stores to MMRs to make sure the data reaches the MMR before the next instruction is fetched.

Typically, the Blackfin processor executes all load instructions strictly in the order that they are issued and all store instructions in the order that they are issued. However, for performance reasons, the architecture relaxes ordering between load and store operations. It usually allows load operations to access memory out of order with respect to store operations.
Further, it usually allows loads to access memory speculatively. The core may later cancel or restart speculative loads. By using the Core Synchronize or System Synchronize instructions and managing interrupts appropriately, you can restrict out-of-order and speculative behavior.

Stores never access memory speculatively.
Instruction Overview

System Synchronize

General Form

SSYNC

Syntax

SSYNC ; /* (a) */

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The System Synchronize (SSYNC) instruction forces all speculative, transient states in the core and system to complete before processing continues. Until SSYNC completes, no further instructions can be issued to the pipeline.

The SSYNC instruction performs the same function as Core Synchronize (CSYNC). In addition, SSYNC flushes any write buffers (between the L1 memory and the system interface) and generates a Synch request signal to the external system. The operation requires an acknowledgement Synch_Ack signal by the system before completing the instruction.

If the idle_req bit of the SEQSTAT register is set when SSYNC is executed, the processor enters Idle state and asserts the external Idle signal after receiving the external Synch_Ack signal. After the external Idle signal is asserted, exiting the Idle state requires an external Wakeup signal.

SSYNC should be issued immediately before and after writing to a system MMR. Otherwise, the MMR change can take effect at an indeterminate time while other instructions are executing, resulting in imprecise behavior.
External Event Management

Flags Affected

None

Required Mode

User & Supervisor

Parallel Issue

The SSYNC instruction cannot be issued in parallel with other instructions.

Example

Consider the following example code sequence.

```c
if cc jump away_from_here ; /* produces speculative branch prediction */
ssync ;
r0 = [p0] ; /* load */
```

In this example, SSYNC ensures that the load instruction will not be executed speculatively. The instruction ensures that the conditional branch is resolved and any entries in the processor store buffer and write buffer have been flushed. In addition, all exceptions complete processing before SSYNC completes.

Also See

Core Synchronize, Idle
Instruction Overview

Special Applications

Typically, SSYNC prepares the architecture for clock cessation or frequency change. In such cases, the following instruction sequence is typical.

:  
instruction...
instruction...
CLI r0 ; /* disable interrupts */
idle ; /* enable Idle state */
ssync ; /* conclude all speculative states, assert external
Sync signal, await Synch_Ack, then assert external Idle signal
and stall in the Idle state until the Wakeup signal. Clock input
can be modified during the stall. */
sti r0 ; /* re-enable interrupts when Wakeup occurs */
instruction...
instruction...
External Event Management

EMUEXCPT (Force Emulation)

General Form

EMUEXCPT

Syntax

EMUEXCPT ; /* (a) */

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Force Emulation instruction forces an emulation exception, thus allowing the processor to enter emulation mode.

When emulation is enabled, the processor immediately takes an exception into emulation mode. When emulation is disabled, EMUEXCPT generates an illegal instruction exception.

An emulation exception is the highest priority event in the processor.

Flags Affected

None

Required Mode

User & Supervisor

Parallel Issue

The Force Emulation instruction cannot be issued in parallel with other instructions.
**Instruction Overview**

**Example**

```
emuexcpt;
```

**Also See**

- RAISE (Force Interrupt / Reset)

**Special Applications**

- None
External Event Management

Disable Interrupts

General Form

CLI

Syntax

CLI Dreg ; /* previous state of IMASK moved to Dreg (a) */

Syntax Terminology

Dreg: R7-0

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Disable Interrupts instruction globally disables general interrupts by setting IMASK to all zeros. In addition, the instruction copies the previous contents of IMASK into a user-specified register in order to save the state of the interrupt system.

The Disable Interrupts instruction does not mask NMI, reset, exceptions and emulation.

Flags Affected

None

Required Mode

The Disable Interrupts instruction executes only in Supervisor mode. If execution is attempted in User mode, the instruction produces an Illegal Use of Protected Resource exception.
Instruction Overview

Parallel Issue

The Disable Interrupts instruction cannot be issued in parallel with other instructions.

Example

c l ir3 ;

Also See

Enable Interrupts

Special Applications

This instruction is often issued immediately before an IDLE instruction.
External Event Management

Enable Interrupts

General Form

STI

Syntax

STI Dreg : /* previous state of IMASK restored from Dreg (a) */

Syntax Terminology

Dreg: R7-0

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Enable Interrupts instruction globally enables interrupts by restoring the previous state of the interrupt system back into IMASK.

Flags Affected

None

Required Mode

The Enable Interrupts instruction executes only in Supervisor mode. If execution is attempted in User mode, the instruction produces an Illegal Use of Protected Resource exception.

Parallel Issue

The Enable Interrupts instruction cannot be issued in parallel with other instructions.
Instruction Overview

Example

stir3;

Also See

Disable Interrupts

Special Applications

This instruction is often located after an IDLE instruction so that it will execute after a wake-up event from the idle state.
External Event Management

RAISE (Force Interrupt / Reset)

General Form

RAISE

Syntax

RAISE uimm4 ; /* (a) */

Syntax Terminology

uimm4: 4-bit unsigned field, with the range of 0 through 15

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Force Interrupt / Reset instruction forces a specified interrupt or reset to occur. Typically, it is a software method of invoking a hardware event for debug purposes.

When the RAISE instruction is issued, the processor sets a bit in the ILAT register corresponding to the interrupt vector specified by the uimm4 constant in the instruction. The interrupt executes when its priority is high enough to be recognized by the processor. The RAISE instruction causes these events to occur given the uimm4 arguments shown in Table 16-1.

Table 16-1. uimm4 Arguments and Events

<table>
<thead>
<tr>
<th>uimm4</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;reserved&gt;</td>
</tr>
<tr>
<td>1</td>
<td>RST</td>
</tr>
<tr>
<td>2</td>
<td>NMI</td>
</tr>
</tbody>
</table>
The Force Interrupt / Reset instruction cannot invoke Exception (EXC) or Emulation (EMU) events; use the EXCPT and EMUEXCPT instructions, respectively, for those events.

The RAISE instruction does not take effect before the write-back stage in the pipeline.

**Flags Affected**

None
External Event Management

Required Mode

The Force Interrupt / Reset instruction executes only in Supervisor mode. If execution is attempted in User mode, the Force Interrupt / Reset instruction produces an Illegal Use of Protected Resource exception.

Parallel Issue

The Force Interrupt / Reset instruction cannot be issued in parallel with other instructions.

Example

raise 1 ; /* Invoke RST */
raise 6 ; /* Invoke IVTMR timer interrupt */

Also See

EXCPT (Force Exception), EMUEXCPT (Force Emulation)

Special Applications

None
Instruction Overview

EXCPT (Force Exception)

General Form

EXCPT

Syntax

EXCPT uimm4 ; /* (a) */

Syntax Terminology

uimm4: 4-bit unsigned field, with the range of 0 through 15

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Force Exception instruction forces an exception with code uimm4. When the EXCPT instruction is issued, the sequencer vectors to the exception handler that the user provides.

Application-level code uses the Force Exception instruction for operating system calls. The instruction does not set the EVSW bit (bit 3) of the ILAT register.

Flags Affected

None

Required Mode

User & Supervisor
External Event Management

Parallel Issue

The Force Exception instruction cannot be issued in parallel with other instructions.

Example

except 4 ;

Also See

None

Special Applications

None
Instruction Overview

Test and Set Byte (Atomic)

General Form

\texttt{TESTSET}

Syntax

\texttt{TESTSET ( Preg ) ; /* (a) */}

Syntax Terminology

\textit{Preg}: \texttt{P5–0} (\texttt{SP} and \texttt{FP} are not allowed as the register for this instruction)

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Test and Set Byte (Atomic) instruction loads an indirectly addressed memory byte, tests whether it is zero, then sets the most significant bit of the memory byte without affecting any other bits. If the byte is originally zero, the instruction sets the \texttt{CC} bit. If the byte is originally nonzero the instruction clears the \texttt{CC} bit. The sequence of this memory transaction is \textit{atomic}.

\texttt{TESTSET} accesses the entire logical memory space except the core Memory-Mapped Register (MMR) address region. The system design must ensure atomicity for all memory regions that \texttt{TESTSET} may access. The hardware does not perform atomic access to L1 memory space configured as SRAM. Therefore, semaphores must not reside in on-core memory.

The memory architecture always treats atomic operations as cache-inhibited accesses, even if the \texttt{CPLB} descriptor for the address indicates a cache-enabled access. If a cache hit is detected, the operation flushes and invalidates the line before allowing the \texttt{TESTSET} to proceed.
External Event Management

The software designer is responsible for executing atomic operations in the proper cacheable / non-cacheable memory space. Typically, these operations should execute in non-cacheable, off-core memory. In a chip implementation that requires tight temporal coupling between processors or processes, the design should implement a dedicated, non-cacheable block of memory that meets the data latency requirements of the system.

TESTSET can be interrupted before the load portion of the instruction completes. If interrupted, the TESTSET will be re-executed upon return from the interrupt. After the test or load portion of the TESTSET completes, the TESTSET sequence cannot be interrupted. For example, any exceptions associated with the CPLB lookup for both the load and store operations must be completed before the load of the TESTSET completes.

The integrity of the TESTSET atomicity depends on the L2 memory resource-locking mechanism. If the L2 memory does not support atomic locking for the address region you are accessing, your software has no guarantee of correct semaphore behavior. See the processor L2 memory documentation for more on the locking support.

Flags Affected

This instruction affects flags as follows.

- CC is set if addressed value is zero; cleared if nonzero.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor
Instruction Overview

Parallel Issue

The TESTSET instruction cannot be issued in parallel with other instructions.

Example

testset (p1);

The TESTSET instruction may be preceded by a CSYNC or SSYNC instruction to ensure that all previous exceptions or interrupts have been processed before the atomic operation begins.

Also See

Core Synchronize, System Synchronize

Special Applications

Typically, use TESTSET as a semaphore sampling method between coprocessors or coprocesses.
External Event Management

No Op

General Form

NOP
MNOP

Syntax

NOP ; /* (a) */
MNOP ; /* (b) */

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

Functional Description

The No Op instruction increments the PC and does nothing else.

Typically, the No Op instruction allows previous instructions time to complete before continuing with subsequent instructions. Other uses are to produce specific delays in timing loops or to act as hardware event timers and rate generators when no timers and rate generators are available.

Flags Affected

None

Required Mode

User & Supervisor
Instruction Overview

Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

```
nop ;
mnop ;
mnop || /* a 16-bit instr. */ || /* a 16-bit instr. */ ;
```

Also See

None

Special Applications

MNOP can be used to issue loads or store instructions in parallel without invoking a 32-bit MAC or ALU operation. Refer to “Issuing Parallel Instructions” on page 20-1 for more information.
17 CACHE CONTROL

Instruction Summary

- “PREFETCH” on page 17-3
- “FLUSH” on page 17-5
- “FLUSHINV” on page 17-7
- “IFLUSH” on page 17-9

Instruction Overview

This chapter discusses the instructions that are used to flush, invalidate, and prefetch data cache lines as well as the instruction used to invalidate a line in the instruction cache.

As part of the data-cache related instructions, the `PREFETCH` instruction can be used to improve performance by initiating a data cache-line fill in advance of when the desired data is actually required for processing. The `FLUSH` instruction is useful when data cache is configured in the write-back mode (which is described in further detail in the “Memory” chapter). This instruction forces data in the cache line that has been changed by the processor (and thus has been marked as “dirty”) to be written to its source memory.

There is no single instruction that can be used to invalidate a data cache-line. The `FLUSHINV` instruction provides a way to directly flush and invalidate a data cache-line. The `FLUSHINV` instruction is commonly used...
to invalidate a buffer, but the instruction also performs a flush of data marked as “dirty.” The ITEST and DTEST registers, which are described in the “Memory” chapter, can also be used to directly invalidate a line in cache. Buffers in source memory need to be invalidated when a DMA channel is filling the buffer and data cache has been enabled and the source memory has been defined as cacheable. By invalidating the cache-lines associated with the buffer, “coherency” is maintained between the contents stored in cache and the actual values in source memory. When the buffer size is less than or equal in size to the actual cache on the processor, it is better to use the FLUSH INV instruction in a loop to invalidate the cache-lines. When the buffer is larger in size than the cache, it is better to use the DTEST registers described in the “Memory” chapter to invalidate the cache-lines.

The IFLUSH instruction is used to invalidate an instruction cache-line.

On the Blackfin processors, the cache-line size is 32 bytes.
Cache Control

PREFETCH

General Form

PREFETCH

Syntax

PREFETCH [ Preg ] ; /* indexed (a) */
PREFETCH [ Preg ++ ] ; /* indexed, post increment (a) */

Syntax Terminology

Preg: P5-0, SP, FP

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Data Cache Prefetch instruction causes the data cache to prefetch the cache line that is associated with the effective address in the P-register. The operation causes the line to be fetched if it is not currently in the data cache and if the address is cacheable (that is, if bit CPLB_L1_CHBL = 1). If the line is already in the cache or if the cache is already fetching a line, the prefetch instruction performs no action, like a NOP.

This instruction does not cause address exception violations. If a protection violation associated with the address occurs, the instruction acts as a NOP and does not cause a protection violation exception.

Options

The instruction can post-increment the line pointer by the cache line size.
Instruction Overview

Flags Affected
None

Required Mode
User & Supervisor

Parallel Issue
This instruction cannot be issued in parallel with other instructions.

Example

```c
prefetch [ p2 ];
prefetch [ p0 ++ ];
```

Also See
None

Special Applications
None
Cache Control

**FLUSH**

**General Form**

FLUSH

**Syntax**

```c
FLUSH [ Preg ] ; /* indexed (a) */
FLUSH [ Preg ++ ] ; /* indexed, post increment (a) */
```

**Syntax Terminology**

Preg: P5-0, SP, FP

**Instruction Length**

In the syntax, comment (a) identifies 16-bit instruction length.

**Functional Description**

The Data Cache Flush instruction causes the data cache to synchronize the specified cache line with higher levels of memory. This instruction selects the cache line corresponding to the effective address contained in the P-register. If the cached data line is dirty, the instruction writes the line out and marks the line clean in the data cache. If the specified data cache line is already clean or the cache does not contain the address in the P-register, this instruction performs no action, like a _NOP_.

This instruction does not cause address exception violations. If a protection violation associated with the address occurs, the instruction acts as a _NOP_ and does not cause a protection violation exception.

**Options**

The instruction can post-increment the line pointer by the cache line size.
Instruction Overview

Flags Affected

None

Required Mode

User & Supervisor

Parallel Issue

The instruction cannot be issued in parallel with other instructions.

Example

```
flush [ p2 ] ;
flush [ p0 ++ ] ;
```

Also See

None

Special Applications

None
Cache Control

FLUSHINV

General Form

FLUSHINV

Syntax

FLUSHINV [ Preg ] ;  /* indexed (a) */
FLUSHINV [ Preg ++ ] ;  /* indexed, post increment (a) */

Syntax Terminology

Preg: P5-0, SP, FP

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The Data Cache Line Invalidate instruction causes the data cache to invalidate a specific line in the cache. The contents of the P-register specify the line to invalidate. If the line is in the cache and dirty, the cache line is written out to the next level of memory in the hierarchy. If the line is not in the cache, the instruction performs no action, like a NOP.

This instruction does not cause address exception violations. If a protection violation associated with the address occurs, the instruction acts as a NOP and does not cause a protection violation exception.

Options

The instruction can post-increment the line pointer by the cache line size.
Instruction Overview

Flags Affected

None

Required Mode

User & Supervisor

Parallel Issue

The Data Cache Line Invalidate instruction cannot be issued in parallel with other instructions.

Example

`flushinv [ p2 ] ;`
`flushinv [ p0 ++ ] ;`

Also See

None

Special Applications

None
Cache Control

**IFLUSH**

**General Form**

```
IFLUSH
```

**Syntax**

```
IFLUSH [ Preg ] ;  /* indexed (a) */
IFLUSH [ Preg ++ ] ;  /* indexed, post increment (a) */
```

**Syntax Terminology**

`Preg`: P5–0, SP, FP

**Instruction Length**

In the syntax, comment (a) identifies 16-bit instruction length.

**Functional Description**

The Instruction Cache Flush instruction causes the instruction cache to invalidate a specific line in the cache. The contents of the P-register specify the line to invalidate. The instruction cache contains no dirty bit. Consequently, the contents of the instruction cache are never flushed to higher levels.

This instruction does not cause address exception violations. If a protection violation associated with the address occurs, the instruction acts as a **NOP** and does not cause a protection violation exception.

**Options**

The instruction can post-increment the line pointer by the cache line size.
Instruction Overview

Flags Affected

None

Required Mode

User & Supervisor

Parallel Issue

This instruction cannot be issued in parallel with other instructions.

Example

iflush [ p2 ] ;
iflush [ p0 + + ] ;

Also See

None

Special Applications

None
18 VIDEO PIXEL OPERATIONS

Instruction Summary

- “ALIGN8, ALIGN16, ALIGN24” on page 18-3
- “DISALGNEXCPT” on page 18-6
- “BYTEOP3P (Dual 16-Bit Add / Clip)” on page 18-8
- “Dual 16-Bit Accumulator Extraction with Addition” on page 18-13
- “BYTEOP16P (Quad 8-Bit Add)” on page 18-15
- “BYTEOP1P (Quad 8-Bit Average – Byte)” on page 18-19
- “BYTEOP2P (Quad 8-Bit Average – Half-Word)” on page 18-24
- “BYTEPACK (Quad 8-Bit Pack)” on page 18-30
- “BYTEOP16M (Quad 8-Bit Subtract)” on page 18-32
- “SAA (Quad 8-Bit Subtract-Absolute-Accumulate)” on page 18-36
- “BYTEUNPACK (Quad 8-Bit Unpack)” on page 18-41
Instruction Overview

This chapter discusses the instructions that manipulate video pixels. Users can take advantage of these instructions to align bytes, disable exceptions that result from misaligned 32-bit memory accesses, and perform dual and quad 8- and 16-bit add, subtract, and averaging operations.
ALIGN8, ALIGN16, ALIGN24

General Form

\[
\text{dest\_reg} = \text{ALIGN8} (\text{src\_reg}_1, \text{src\_reg}_0) \\
\text{dest\_reg} = \text{ALIGN16} (\text{src\_reg}_1, \text{src\_reg}_0) \\
\text{dest\_reg} = \text{ALIGN24} (\text{src\_reg}_1, \text{src\_reg}_0)
\]

Syntax

\[
\text{Dreg} = \text{ALIGN8} (\text{Dreg}, \text{Dreg}) ; /* overlay 1 byte (b) */ \\
\text{Dreg} = \text{ALIGN16} (\text{Dreg}, \text{Dreg}) ; /* overlay 2 bytes (b) */ \\
\text{Dreg} = \text{ALIGN24} (\text{Dreg}, \text{Dreg}) ; /* overlay 3 bytes (b) */
\]

Syntax Terminology

\text{Dreg}: R7-0

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Byte Align instruction copies a contiguous four-byte unaligned word from a combination of two data registers. The instruction version determines the bytes that are copied; in other words, the byte alignment of the copied word. Alignment options are shown in Table 18-1.

The ALIGN16 version performs the same operation as the Vector Pack instruction using the \text{dest\_reg} = \text{PACK} (Dreg\_lo, Dreg\_hi) syntax.

Use the Byte Align instruction to align data bytes for subsequent single-instruction, multiple-data (SIMD) instructions.
Instruction Overview

Table 18-1. Byte Alignment Options

<table>
<thead>
<tr>
<th>src_reg_1</th>
<th>src_reg_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte7</td>
<td>byte6</td>
</tr>
<tr>
<td>byte5</td>
<td>byte4</td>
</tr>
<tr>
<td>byte2</td>
<td>byte1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dest_reg for ALIGN8:</th>
<th>byte4</th>
<th>byte3</th>
<th>byte2</th>
<th>byte1</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest_reg for ALIGN16:</td>
<td>byte5</td>
<td>byte4</td>
<td>byte3</td>
<td>byte2</td>
</tr>
<tr>
<td>dest_reg for ALIGN24:</td>
<td>byte6</td>
<td>byte5</td>
<td>byte4</td>
<td>byte3</td>
</tr>
</tbody>
</table>

The input values are not implicitly modified by this instruction. The destination register can be the same D-register as one of the source registers. Doing this explicitly modifies that source register.

Flags Affected

None

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.
Video Pixel Operations

Example

    // If r3 = 0xABCD 1234 and r4 = 0xBEEF DEAD, then . . .
    r0 = align8 (r3, r4);  /* produces r0 = 0x34BE EFDE, */
    r0 = align16 (r3, r4); /* produces r0 = 0x1234 BEEF, and */
    r0 = align24 (r3, r4); /* produces r0 = 0xCD12 34BE. */

Also See

    Vector PACK

Special Applications

    None
**Instruction Overview**

**DISALGNEXCPT**

General Form

```
DISALGNEXCPT
```

Syntax

```
DISALGNEXCPT ; /* (b) */
```

Syntax Terminology

None

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Disable Alignment Exception for Load (DISALGNEXCPT) instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel. This instruction only affects misaligned 32-bit load instructions that use I-register indirect addressing.

In order to force address alignment to a 32-bit boundary, the two LSBs of the address are cleared before being sent to the memory system. The I-register is not modified by the DISALGNEXCPT instruction. Also, any modifications performed to the I-register by a parallel instruction are not affected by the DISALGNEXCPT instruction.

Flags Affected

None
Video Pixel Operations

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

\begin{verbatim}
disalgnexcpt || r1 = [i0++] || r3 = [i1++] ; /* three instructions in parallel */
disalgnexcpt || [p0 ++ p1] = r5 || r3 = [i1++] ; /* alignment exception is prevented only for the load */
disalgnexcpt || r0 = [p2++] || r3 = [i1++] ; /* alignment exception is prevented only for the I-reg load */
\end{verbatim}

Also See

Any Quad 8-Bit instructions, ALIGN8, ALIGN16, ALIGN24

Special Applications

Use the DISALGNEXCPT instruction when priming data registers for Quad 8-Bit single-instruction, multiple-data (SIMD) instructions.

Quad 8-Bit SIMD instructions require as many as sixteen 8-bit operands, four D-registers worth, to be preloaded with operand data. The operand data is 8 bits and not necessarily word aligned in memory. Thus, use DISALGNEXCPT to prevent spurious exceptions for these potentially misaligned accesses.

During execution, when Quad 8-Bit SIMD instructions perform 8-bit boundary accesses, they automatically prevent exceptions for misaligned accesses. No user intervention is required.
Instruction Overview

BYTEOP3P (Dual 16-Bit Add / Clip)

General Form

\[
\text{dest}_\text{reg} = \text{BYTEOP3P} \left( \text{src}_\text{reg}_0, \text{src}_\text{reg}_1 \right) \ (\text{LO}) \\
\text{dest}_\text{reg} = \text{BYTEOP3P} \left( \text{src}_\text{reg}_0, \text{src}_\text{reg}_1 \right) \ (\text{HI}) \\
\text{dest}_\text{reg} = \text{BYTEOP3P} \left( \text{src}_\text{reg}_0, \text{src}_\text{reg}_1 \right) \ (\text{LO}, \ R) \\
\text{dest}_\text{reg} = \text{BYTEOP3P} \left( \text{src}_\text{reg}_0, \text{src}_\text{reg}_1 \right) \ (\text{HI}, \ R)
\]

Syntax

/* forward byte order operands */
\[
\text{Dreg} = \text{BYTEOP3P} \left( \text{Dreg}_\text{pair}, \text{Dreg}_\text{pair} \right) \ (\text{LO}) ; \ /* \ text{sum} \ into \ low \\
\text{bytes} \ (b) */
\]

/* reverse byte order operands */
\[
\text{Dreg} = \text{BYTEOP3P} \left( \text{Dreg}_\text{pair}, \text{Dreg}_\text{pair} \right) \ (\text{LO}, \ R) ; \ /* \ sum \ into \\
\text{low} \ bytes \ (b) */
\]

Syntax Terminology

\[
\text{Dreg}: \ R7-0 \\
\text{Dreg}_\text{pair}: \ R1:0, \ R3:2, \ only
\]

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Dual 16-Bit Add / Clip instruction adds two 8-bit unsigned values to two 16-bit signed values, then limits (or “clips”) the result to the 8-bit unsigned range 0 through 255, inclusive. The instruction loads the results...
as bytes on half-word boundaries in one 32-bit destination register. Some syntax options load the upper byte in the half-word and others load the lower byte, as shown in Table 18-2, Table 18-4, and Table 18-4.

Table 18-2. Assuming the source registers contain:

```
<table>
<thead>
<tr>
<th>aligned_src_reg_0:</th>
<th>31...........24</th>
<th>23...........16</th>
<th>15...........8</th>
<th>7...............0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>y1</td>
<td>y0</td>
<td></td>
</tr>
<tr>
<td>aligned_src_reg_1:</td>
<td>z3</td>
<td>z2</td>
<td>z1</td>
<td>z0</td>
</tr>
</tbody>
</table>
```

Table 18-3. The versions that load the result into the lower byte—“(LO)”—produce:

```
<table>
<thead>
<tr>
<th>dest_reg:</th>
<th>31...........24</th>
<th>23...........16</th>
<th>15...........8</th>
<th>7...............0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 . . . . 0</td>
<td>y1 + z3 clipped to 8 bits</td>
<td>0 . . . . 0</td>
<td>y0 + z1 clipped to 8 bits</td>
</tr>
</tbody>
</table>
```

Table 18-4. And the versions that load the result into the higher byte—“(HI)”—produce:

```
<table>
<thead>
<tr>
<th>dest_reg:</th>
<th>31...........24</th>
<th>23...........16</th>
<th>15...........8</th>
<th>7...............0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y1 + z2 clipped to 8 bits</td>
<td>0 . . . . 0</td>
<td>y0 + z0 clipped to 8 bits</td>
<td>0 . . . . 0</td>
</tr>
</tbody>
</table>
```

In either case, the unused bytes in the destination register are filled with 0x00.

The 8-bit and 16-bit addition is performed as a signed operation. The 16-bit operand is sign-extended to 32 bits before adding.

The only valid input source register pairs are \( R1:0 \) and \( R3:2 \).
The Dual 16-Bit Add / Clip instruction provides byte alignment directly in the source register pairs `src_reg_0` and `src_reg_1` based on index registers `I0` and `I1`.

- The two LSBs of the `I0` register determine the byte alignment for source register pair `src_reg_0` (typically `R1:0`).
- The two LSBs of the `I1` register determine the byte alignment for source register pair `src_reg_1` (typically `R3:2`).

The relationship between the I-register bits and the byte alignment is illustrated in Table 18-5.

In the default source order case (for example, not the (−, R) syntax), assuming a source register pair contains the following.

### Table 18-5. I-register Bits and the Byte Alignment

<table>
<thead>
<tr>
<th>Two LSB's of I0 or I1</th>
<th>src_reg_pair_HI</th>
<th>src_reg_pair_LO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>byte7</td>
<td>byte6</td>
</tr>
<tr>
<td>00b:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01b:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10b:</td>
<td></td>
<td>byte5</td>
</tr>
<tr>
<td>11b:</td>
<td>byte6</td>
<td>byte5</td>
</tr>
</tbody>
</table>

This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

### Options

The (−, R) syntax reverses the order of the source registers within each register pair. Typical high performance applications cannot afford the overhead of reloading both register pair operands to maintain byte order for every calculation. Instead, they alternate and load only one register pair operand each time and alternate between the forward and reverse byte
Video Pixel Operations

order versions of this instruction. By default, the low order bytes come from the low register in the register pair. The ( – , R) option causes the low order bytes to come from the high register.

In the optional reverse source order case (for example, using the ( – , R) syntax), the only difference is the source registers swap places within the register pair in their byte ordering. Assume a source register pair contains the data shown in Table 18-6.

Table 18-6. I-register Bits and the Byte Alignment

<table>
<thead>
<tr>
<th>The bytes selected are</th>
<th>src_reg_pair_LO</th>
<th>src_reg_pair_HI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two LSB’s of I0 or I1</td>
<td>byte7 byte6 byte5 byte4</td>
<td>byte3 byte2 byte1 byte0</td>
</tr>
<tr>
<td></td>
<td>byte4 byte3 byte2 byte1</td>
<td>byte5 byte4 byte3 byte2</td>
</tr>
<tr>
<td>00b:</td>
<td>byte3 byte2 byte1 byte0</td>
<td></td>
</tr>
<tr>
<td>01b:</td>
<td>byte4 byte3 byte2 byte1</td>
<td></td>
</tr>
<tr>
<td>10b:</td>
<td>byte5 byte4 byte3 byte2</td>
<td></td>
</tr>
<tr>
<td>11b:</td>
<td>byte6 byte5 byte4 byte3</td>
<td></td>
</tr>
</tbody>
</table>

Flags Affected
None

Required Mode
User & Supervisor

Parallel Issue
This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

\[
\begin{align*}
\text{r3} & = \text{byteop3p} \ (r1:0, r3:2) \ (\text{lo}) ; \\
\text{r3} & = \text{byteop3p} \ (r1:0, r3:2) \ (\text{hi}) ; \\
\end{align*}
\]
Instruction Overview

\[ r3 = \text{byteop3p} (r1:0, r3:2) (lo, r) ; \]
\[ r3 = \text{byteop3p} (r1:0, r3:2) (hi, r) ; \]

Also See

BYTEOP16P (Quad 8-Bit Add)

Special Applications

This instruction is primarily intended for video motion compensation algorithms. The instruction supports the addition of the residual to a video pixel value, followed by unsigned byte saturation.
Video Pixel Operations

Dual 16-Bit Accumulator Extraction with Addition

General Form

\[
\text{dest\_reg\_1} = A1.L + A1.H, \text{dest\_reg\_0} = A0.L + A0.H
\]

Syntax

\[
D\text{reg} = A1.L + A1.H, D\text{reg} = A0.L + A0.H \quad ; \quad /* (b) */
\]

Syntax Terminology

\[D\text{reg}: R7-0\]

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Dual 16-Bit Accumulator Extraction with Addition instruction adds together the upper half-words (bits 31 through 16) and lower half-words (bits 15 through 0) of each Accumulator and loads each result into a 32-bit destination register.

Each 16-bit half-word in each Accumulator is sign extended before being added together.

Flags Affected

None

Required Mode

User & Supervisor
Instruction Overview

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

\[ r4 = a1.1 + a1.h, \ r7 = a0.1 + a0.h ; \]

Also See

SAA (Quad 8-Bit Subtract-Absolute-Accumulate)

Special Applications

Use the Dual 16-Bit Accumulator Extraction with Addition instruction for motion estimation algorithms in conjunction with the Quad 8-Bit Subtract-Absolute-Accumulate instruction.
Video Pixel Operations

BYTEOP16P (Quad 8-Bit Add)

General Form

\[(dest\_reg\_1, dest\_reg\_0) = BYTEOP16P (src\_reg\_0, src\_reg\_1)\]
\[(dest\_reg\_1, dest\_reg\_0) = BYTEOP16P (src\_reg\_0, src\_reg\_1) \ (R)\]

Syntax

/* forward byte order operands */
(Dreg, Dreg) = BYTEOP16P (Dreg\_pair, Dreg\_pair) ; /* (b) */
/* reverse byte order operands */
(Dreg, Dreg) = BYTEOP16P (Dreg\_pair, Dreg\_pair) (R)
; /* (b) */

Syntax Terminology

\[Dreg: R7-0\]
\[Dreg\_pair: R1:0, R3:2, only\]

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Quad 8-Bit Add instruction adds two unsigned quad byte number sets byte-wise, adjusting for byte alignment. It then loads the byte-wise results as 16-bit, zero-extended, half-words in two destination registers, as shown in Table 18-7 and Table 18-8.

The only valid input source register pairs are R1:0 and R3:2.
The Quad 8-Bit Add instruction provides byte alignment directly in the source register pairs \( \text{src\_reg\_0} \) and \( \text{src\_reg\_1} \) based on index registers \( I0 \) and \( I1 \).

- The two LSBs of the \( I0 \) register determine the byte alignment for source register pair \( \text{src\_reg\_0} \) (typically \( R1:0 \)).
- The two LSBs of the \( I1 \) register determine the byte alignment for source register pair \( \text{src\_reg\_1} \) (typically \( R3:2 \)).

The relationship between the I-register bits and the byte alignment is illustrated below.

In the default source order case (for example, not the (R) syntax), assume that a source register pair contains the data shown in Table 18-9.

This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

## Options

The (R) syntax reverses the order of the source registers within each register pair. Typical high performance applications cannot afford the overhead of reloading both register pair operands to maintain byte order.
for every calculation. Instead, they alternate and load only one register
pair operand each time and alternate between the forward and reverse byte
order versions of this instruction. By default, the low order bytes come
from the low register in the register pair. The (R) option causes the low
order bytes to come from the high register.

In the optional reverse source order case (for example, using the (R) syn-
tax), the only difference is the source registers swap places within the
register pair in their byte ordering. Assume a source register pair contains
the data shown in Table 18-10.

Table 18-9. I-register Bits and the Byte Alignment

<table>
<thead>
<tr>
<th>Two LSB’s of I0 or I1</th>
<th>src_reg_pair_HI</th>
<th>src_reg_pair_LO</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte7</td>
<td>byte6</td>
<td>byte5</td>
</tr>
<tr>
<td>byte4</td>
<td>byte3</td>
<td>byte2</td>
</tr>
<tr>
<td>byte1</td>
<td>byte0</td>
<td></td>
</tr>
<tr>
<td>00b:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01b:</td>
<td>byte4</td>
<td>byte3</td>
</tr>
<tr>
<td>10b:</td>
<td>byte5</td>
<td>byte4</td>
</tr>
<tr>
<td>11b:</td>
<td>byte6</td>
<td>byte5</td>
</tr>
</tbody>
</table>

Table 18-10. I-register Bits and the Byte Alignment

<table>
<thead>
<tr>
<th>Two LSB’s of I0 or I1</th>
<th>src_reg_pair_LO</th>
<th>src_reg_pair_HI</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte7</td>
<td>byte6</td>
<td>byte5</td>
</tr>
<tr>
<td>byte4</td>
<td>byte3</td>
<td>byte2</td>
</tr>
<tr>
<td>byte1</td>
<td>byte0</td>
<td></td>
</tr>
<tr>
<td>00b:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01b:</td>
<td>byte4</td>
<td>byte3</td>
</tr>
<tr>
<td>10b:</td>
<td>byte5</td>
<td>byte4</td>
</tr>
<tr>
<td>11b:</td>
<td>byte6</td>
<td>byte5</td>
</tr>
</tbody>
</table>

The mnemonic derives its name from the fact that the operands are bytes,
the result is 16 bits, and the arithmetic operation is “plus” for addition.
Instruction Overview

Flags Affected

None

The ADSP-BF535 processor has fewer ASAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

\[(r1,r2) = \text{byteop16p} (r3:2,r1:0) ;\]
\[(r1,r2) = \text{byteop16p} (r3:2,r1:0) (r) ;\]

Also See

BYTEOP16M (Quad 8-Bit Subtract)

Special Applications

This instruction provides packed data arithmetic typical of video and image processing applications.
**Video Pixel Operations**

**BYTEOP1P (Quad 8-Bit Average – Byte)**

**General Form**

\[
dest\_reg = \text{BYTEOP1P ( src\_reg\_0, src\_reg\_1 )}
\]

\[
dest\_reg = \text{BYTEOP1P ( src\_reg\_0, src\_reg\_1 ) (T)}
\]

\[
dest\_reg = \text{BYTEOP1P ( src\_reg\_0, src\_reg\_1 ) (R)}
\]

\[
dest\_reg = \text{BYTEOP1P ( src\_reg\_0, src\_reg\_1 ) (T, R)}
\]

**Syntax**

/* forward byte order operands */

\[
\text{Dreg} = \text{BYTEOP1P (Dreg\_pair, Dreg\_pair) ; /* (b) */}
\]

/* reverse byte order operands */

\[
\text{Dreg} = \text{BYTEOP1P (Dreg\_pair, Dreg\_pair) (R) ; /* (b) */}
\]

**Syntax Terminology**

\[\text{Dreg: R7-0}\]

\[\text{Dreg\_pair: R1:0, R3:2, only}\]

**Instruction Length**

In the syntax, comment (b) identifies 32-bit instruction length.

**Functional Description**

The Quad 8-Bit Average – Byte instruction computes the arithmetic average of two unsigned quad byte number sets byte wise, adjusting for byte alignment. This instruction loads the byte-wise results as concatenated bytes in one 32-bit destination register, as shown in Table 18-11 and Table 18-12.
In the ADSP-BF53x/BF56x Blackfin Processor, arithmetic average (or mean) is calculated by summing the two operands, then shifting right one place to divide by two. The user has two options to bias the result—truncation or rounding up. By default, the architecture rounds up the mean when the sum is odd. However, the syntax supports optional truncation.

See “Rounding and Truncating” on page 1-19 for a description of biased rounding and truncating behavior.

The **RND_MOD** bit in the **ASTAT** register has no bearing on the rounding behavior of this instruction.

The only valid input source register pairs are **R1:0** and **R3:2**.

The Quad 8-Bit Average – Byte instruction provides byte alignment directly in the source register pairs **src_reg_0** and **src_reg_1** based on index registers **I0** and **I1**.

- The two LSBs of the **I0** register determine the byte alignment for source register pair **src_reg_0** (typically **R1:0**).
- The two LSBs of the **I1** register determine the byte alignment for source register pair **src_reg_1** (typically **R3:2**).

### Table 18-11. Source Registers Contain

<table>
<thead>
<tr>
<th>31...........24</th>
<th>23...........16</th>
<th>15...........8</th>
<th>7...........0</th>
</tr>
</thead>
<tbody>
<tr>
<td>y3</td>
<td>y2</td>
<td>y1</td>
<td>y0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31...........24</th>
<th>23...........16</th>
<th>15...........8</th>
<th>7...........0</th>
</tr>
</thead>
<tbody>
<tr>
<td>z3</td>
<td>z2</td>
<td>z1</td>
<td>z0</td>
</tr>
</tbody>
</table>

### Table 18-12. Destination Registers Receive

<table>
<thead>
<tr>
<th>31...........24</th>
<th>23...........16</th>
<th>15...........8</th>
<th>7...........0</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg(y3, z3)</td>
<td>avg(y2, z2)</td>
<td>avg(y1, z1)</td>
<td>avg(y0, z0)</td>
</tr>
</tbody>
</table>

Arithmetic average (or mean) is calculated by summing the two operands, then shifting right one place to divide by two.

The user has two options to bias the result—truncation or rounding up. By default, the architecture rounds up the mean when the sum is odd. However, the syntax supports optional truncation.

See “Rounding and Truncating” on page 1-19 for a description of biased rounding and truncating behavior.

The **RND_MOD** bit in the **ASTAT** register has no bearing on the rounding behavior of this instruction.

The only valid input source register pairs are **R1:0** and **R3:2**.

The Quad 8-Bit Average – Byte instruction provides byte alignment directly in the source register pairs **src_reg_0** and **src_reg_1** based on index registers **I0** and **I1**.

- The two LSBs of the **I0** register determine the byte alignment for source register pair **src_reg_0** (typically **R1:0**).
- The two LSBs of the **I1** register determine the byte alignment for source register pair **src_reg_1** (typically **R3:2**).
Video Pixel Operations

The relationship between the I-register bits and the byte alignment is illustrated below.

In the default source order case (for example, not the \((R)\) syntax), assume a source register pair contains the data shown in Table 18-13.

Table 18-13. I-register Bits and the Byte Alignment

<table>
<thead>
<tr>
<th>The bytes selected are</th>
<th>src_reg_pair_HI</th>
<th>src_reg_pair_LO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two LSB's of I0 or I1</td>
<td>byte7 byte6 byte5 byte4</td>
<td>byte3 byte2 byte1 byte0</td>
</tr>
<tr>
<td>00b:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01b:</td>
<td>byte4</td>
<td>byte3 byte2 byte1</td>
</tr>
<tr>
<td>10b:</td>
<td>byte5</td>
<td>byte4 byte3 byte2</td>
</tr>
<tr>
<td>11b:</td>
<td>byte6</td>
<td>byte5 byte4 byte3</td>
</tr>
</tbody>
</table>

This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

Options

The Quad 8-Bit Average – Byte instruction supports the following options.

Table 18-14. Options for Quad 8-Bit Average – Byte

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>Rounds up the arithmetic mean.</td>
</tr>
<tr>
<td>(T)</td>
<td>Truncates the arithmetic mean.</td>
</tr>
</tbody>
</table>
In the optional reverse source order case (for example, using the (R) syntax), the only difference is the source registers swap places within the register pair in their byte ordering. Assume a source register pair contains the data shown in Table 18-15.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R)</td>
<td>Reverses the order of the source registers within each register pair. Typical high performance applications cannot afford the overhead of reloading both register pair operands to maintain byte order for every calculation. Instead, they alternate and load only one register pair operand each time and alternate between the forward and reverse byte order versions of this instruction. By default, the low order bytes come from the low register in the register pair. The (R) option causes the low order bytes to come from the high register.</td>
</tr>
<tr>
<td>(T, R)</td>
<td>Combines both of the above options.</td>
</tr>
</tbody>
</table>

Table 18-15. I-register Bits and the Byte Alignment

<table>
<thead>
<tr>
<th>The bytes selected are</th>
<th>src_reg_pair_LO</th>
<th>src_reg_pair_HI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two LSB’s of I0 or I1</td>
<td>byte7 byte6 byte5 byte4</td>
<td>byte3 byte2 byte1 byte0</td>
</tr>
<tr>
<td>00b:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01b:</td>
<td>byte4 byte3 byte2 byte1</td>
<td></td>
</tr>
<tr>
<td>10b:</td>
<td>byte5 byte4 byte3 byte2</td>
<td></td>
</tr>
<tr>
<td>11b:</td>
<td>byte6 byte5 byte4 byte3</td>
<td></td>
</tr>
</tbody>
</table>

The mnemonic derives its name from the fact that the operands are bytes, the result is one word, and the basic arithmetic operation is “plus” for addition. The single destination register indicates that averaging is performed.
Video Pixel Operations

Flags Affected

None

⚠️ The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

```
    r3 = byteop1p (r1:0, r3:2) ;
    r3 = byteop1p (r1:0, r3:2) (r) ;
    r3 = byteop1p (r1:0, r3:2) (t) ;
    r3 = byteop1p (r1:0, r3:2) (t,r) ;
```

Also See

BYTEOP16P (Quad 8-Bit Add)

Special Applications

This instruction supports binary interpolation used in fractional motion search and motion compensation algorithms.
Instruction Overview

BYTEOP2P (Quad 8-Bit Average – Half-Word)

General Form

```
dest_reg = BYTEOP2P ( src_reg_0, src_reg_1 ) (RNDL)
dest_reg = BYTEOP2P ( src_reg_0, src_reg_1 ) (RNDH)
dest_reg = BYTEOP2P ( src_reg_0, src_reg_1 ) (TL)
dest_reg = BYTEOP2P ( src_reg_0, src_reg_1 ) (TH)
dest_reg = BYTEOP2P ( src_reg_0, src_reg_1 ) (RNDL, R)
dest_reg = BYTEOP2P ( src_reg_0, src_reg_1 ) (RNDH, R)
dest_reg = BYTEOP2P ( src_reg_0, src_reg_1 ) (TL, R)
dest_reg = BYTEOP2P ( src_reg_0, src_reg_1 ) (TH, R)
```

Syntax

```
/* forward byte order operands */
Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (RNDL);  
/* round into low bytes (b) */
Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (RNDH);  
/* round into high bytes (b) */
Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (TL);    
/* truncate into low bytes (b) */
Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (TH);    
/* truncate into high bytes (b) */
/* reverse byte order operands */
Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (RNDL, R);  
/* round into low bytes (b) */
Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (RNDH, R);  
/* round into high bytes (b) */
Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (TL, R);  
/* truncate into low bytes (b) */
Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (TH, R);  
/* truncate into high bytes (b) */
```
Video Pixel Operations

Syntax Terminology

\( Dreg: R7-0 \)

\( Dreg\_pair: R1:0, R3:2, \text{ only} \)

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Quad 8-Bit Average – Half-Word instruction finds the arithmetic average of two unsigned quad byte number sets byte wise, adjusting for byte alignment. This instruction averages four bytes together. The instruction loads the results as bytes on half-word boundaries in one 32-bit destination register. Some syntax options load the upper byte in the half-word and others load the lower byte, as shown in Table 18-16, Table 18-17, and Table 18-18.

Table 18-16. Source Registers Contain

<table>
<thead>
<tr>
<th>31............24</th>
<th>23............16</th>
<th>15............8</th>
<th>7............0</th>
</tr>
</thead>
<tbody>
<tr>
<td>y3</td>
<td>y2</td>
<td>y1</td>
<td>y0</td>
</tr>
<tr>
<td>z3</td>
<td>z2</td>
<td>z1</td>
<td>z0</td>
</tr>
</tbody>
</table>

Table 18-17. The versions that load the result into the lower byte – RNDL and TL – produce:

<table>
<thead>
<tr>
<th>31............24</th>
<th>23............16</th>
<th>15............8</th>
<th>7............0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 . . . . . .</td>
<td>avg(y3, y2, z3, z2)</td>
<td>0 . . . . . .</td>
<td>avg(y1, y0, z1, z0)</td>
</tr>
</tbody>
</table>

In either case, the unused bytes in the destination register are filled with 0x00.
Arithmetic average (or mean) is calculated by summing the four byte operands, then shifting right two places to divide by four.

When the intermediate sum is not evenly divisible by 4, precision may be lost.

The user has two options to bias the result—truncation or biased rounding.

See “Rounding and Truncating” on page 1-19 for a description of unbiased rounding and truncating behavior.

The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction.

The only valid input source register pairs are R1:0 and R3:2.

The Quad 8-Bit Average – Half-Word instruction provides byte alignment directly in the source register pairs src_reg_0 (typically R1:0) and src_reg_1 (typically R3:2) based only on the I0 register. The byte alignment in both source registers must be identical since only one register specifies the byte alignment for them both.

The relationship between the I-register bits and the byte alignment is illustrated in Table 18-19.

In the default source order case (for example, not the (R) syntax), assume a source register pair contains the data shown in Table 18-19.

This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.
Video Pixel Operations

Table 18-19. I-register Bits and the Byte Alignment

<table>
<thead>
<tr>
<th>The bytes selected are</th>
<th>src_reg_pair_HI</th>
<th>src_reg_pair_LO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two LSB’s of I0 or I1</td>
<td>byte7 byte6 byte5 byte4</td>
<td>byte3 byte2 byte1 byte0</td>
</tr>
<tr>
<td>00b:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01b:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10b:</td>
<td>byte5</td>
<td>byte4</td>
</tr>
<tr>
<td>11b:</td>
<td>byte6</td>
<td>byte5</td>
</tr>
</tbody>
</table>

Options

The Quad 8-Bit Average – Half-Word instruction supports the following options.

Table 18-20. Options for Quad 8-Bit Average – Half-Word

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(RND—)</td>
<td>Rounds up the arithmetic mean.</td>
</tr>
<tr>
<td>(T—)</td>
<td>Truncates the arithmetic mean.</td>
</tr>
<tr>
<td>(—L)</td>
<td>Loads the results into the lower byte of each destination half-word.</td>
</tr>
<tr>
<td>(—H)</td>
<td>Loads the results into the higher byte of each destination half-word.</td>
</tr>
<tr>
<td>(,R)</td>
<td>Reverses the order of the source registers within each register pair. Typical high performance applications cannot afford the overhead of reloading both register pair operands to maintain byte order for every calculation. Instead, they alternate and load only one register pair operand each time and alternate between the forward and reverse byte order versions of this instruction. By default, the low order bytes come from the low register in the register pair. The (R) option causes the low order bytes to come from the high register.</td>
</tr>
</tbody>
</table>

When used together, the order of the options in the syntax makes no difference.
Instruction Overview

In the optional reverse source order case (for example, using the (R) syntax), the only difference is the source registers swap places within the register pair in their byte ordering. Assume a source register pair contains the data shown in Table 18-21.

Table 18-21. I-register Bits and the Byte Alignment

<table>
<thead>
<tr>
<th>The bytes selected are 00b:</th>
<th>src_reg_pair_LO</th>
<th>src_reg_pair_HI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two LSB’s of I0 or I1 byte7</td>
<td>byte3</td>
<td>byte6</td>
</tr>
<tr>
<td>00b:</td>
<td>byte3</td>
<td>byte6</td>
</tr>
<tr>
<td>01b:</td>
<td>byte4</td>
<td>byte3</td>
</tr>
<tr>
<td>10b:</td>
<td>byte5</td>
<td>byte4</td>
</tr>
<tr>
<td>11b:</td>
<td>byte6</td>
<td>byte5</td>
</tr>
</tbody>
</table>

The mnemonic derives its name from the fact that the operands are bytes, the result is two half-words, and the basic arithmetic operation is “plus” for addition. The single destination register indicates that averaging is performed.

Flags Affected

None

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor
Video Pixel Operations

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

\[
\begin{align*}
    r3 &= \text{byteop2p} (r1:0, r3:2) (\text{rndl}) ; \\
    r3 &= \text{byteop2p} (r1:0, r3:2) (\text{rndh}) ; \\
    r3 &= \text{byteop2p} (r1:0, r3:2) (\text{tl}) ; \\
    r3 &= \text{byteop2p} (r1:0, r3:2) (\text{th}) ; \\
    r3 &= \text{byteop2p} (r1:0, r3:2) (\text{rndl}, r) ; \\
    r3 &= \text{byteop2p} (r1:0, r3:2) (\text{rndh}, r) ; \\
    r3 &= \text{byteop2p} (r1:0, r3:2) (\text{tl}, r) ; \\
    r3 &= \text{byteop2p} (r1:0, r3:2) (\text{th}, r) ;
\end{align*}
\]

Also See

BYTEOP1P (Quad 8-Bit Average – Byte)

Special Applications

This instruction supports binary interpolation used in fractional motion search and motion compensation algorithms.
Instruction Overview

BYTEPACK (Quad 8-Bit Pack)

General Form

\[ \text{dest\_reg} = \text{BYTEPACK} ( \text{src\_reg\_0, src\_reg\_1} ) \]

Syntax

\[ Dreg = \text{BYTEPACK} ( Dreg, Dreg ) ; /* (b) */ \]

Syntax Terminology

\[ Dreg: R7-0 \]

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Quad 8-Bit Pack instruction packs four 8-bit values, half-word aligned, contained in two source registers into one register, byte aligned as shown in Table 18-22 and Table 18-23.

Table 18-22. Source Registers Contain

<table>
<thead>
<tr>
<th>31..............24</th>
<th>23..............16</th>
<th>15..............8</th>
<th>7..............0</th>
</tr>
</thead>
<tbody>
<tr>
<td>src_reg_0:</td>
<td></td>
<td>byte1</td>
<td>byte0</td>
</tr>
<tr>
<td>src_reg_1:</td>
<td></td>
<td>byte3</td>
<td>byte2</td>
</tr>
</tbody>
</table>

Table 18-23. Destination Register Receives

<table>
<thead>
<tr>
<th>byte3</th>
<th>byte2</th>
<th>byte1</th>
<th>byte0</th>
</tr>
</thead>
</table>

This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.
Video Pixel Operations

Flags Affected

None

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

\[ r_2 = \text{bytepack}(r_4, r_5); \]

- Assuming:
  - \( r_4 = 0x\text{FEED FACE} \)
  - \( r_5 = 0xBEEF BADD \)

then this instruction returns:

- \( r_2 = 0xEFDD EDCE \)

Also See

BYTEUNPACK (Quad 8-Bit Unpack)

Special Applications

None
BYTEOP16M (Quad 8-Bit Subtract)

General Form

\[
\begin{align*}
(dest\_reg\_1, dest\_reg\_0) &= \text{BYTEOP16M} (src\_reg\_0, src\_reg\_1) \\
(dest\_reg\_1, dest\_reg\_0) &= \text{BYTEOP16M} (src\_reg\_0, src\_reg\_1) \ (R)
\end{align*}
\]

Syntax

/* forward byte order operands */
\[
(Dreg, Dreg) = \text{BYTEOP16M} (Dreg\_pair, Dreg\_pair) ; /* (b */)
\]
/* reverse byte order operands */
\[
(Dreg, Dreg) = \text{BYTEOP16M} (Dreg\_pair, Dreg\_pair) \ (R) ; /* (b) */
\]

Syntax Terminology

\[Dreg: R7-0\]
\[Dreg\_pair: R1:0, R3:2, only\]

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Quad 8-Bit Subtract instruction subtracts two unsigned quad byte number sets byte wise, adjusting for byte alignment. The instruction loads the byte-wise results as sign-extended half-words in two destination registers, as shown in Table 18-24 and Table 18-25.

Table 18-24. Source Registers Contain

<table>
<thead>
<tr>
<th></th>
<th>31.............24</th>
<th>23.............16</th>
<th>15.............8</th>
<th>7.............0</th>
</tr>
</thead>
<tbody>
<tr>
<td>aligned_src_reg_0:</td>
<td>y3</td>
<td>y2</td>
<td>y1</td>
<td>y0</td>
</tr>
<tr>
<td>aligned_src_reg_1:</td>
<td>z3</td>
<td>z2</td>
<td>z1</td>
<td>z0</td>
</tr>
</tbody>
</table>
Video Pixel Operations

The only valid input source register pairs are R1:0 and R3:2.

The Quad 8-Bit Subtract instruction provides byte alignment directly in the source register pairs src_reg_0 and src_reg_1 based on index registers I0 and I1.

- The two LSBs of the I0 register determine the byte alignment for source register pair src_reg_0 (typically R1:0).
- The two LSBs of the I1 register determine the byte alignment for source register pair src_reg_1 (typically R3:2).

The relationship between the I-register bits and the byte alignment is illustrated shown in Table 18-26.

In the default source order case (for example, not the (R) syntax), assume a source register pair contains the data shown in Table 18-26.

Table 18-25. Destination Registers Receive

<table>
<thead>
<tr>
<th>dest_reg_0:</th>
<th>dest_reg_1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>y1 - z1</td>
<td>y0 - z0</td>
</tr>
<tr>
<td>y3 - z3</td>
<td>y2 - z2</td>
</tr>
</tbody>
</table>

Table 18-26. I-register Bits and the Byte Alignment

<table>
<thead>
<tr>
<th>The bytes selected are</th>
<th>src_reg_pair_HI</th>
<th>src_reg_pair_LO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two LSB's of I0 or I1</td>
<td>byte7 byte6 byte5 byte4</td>
<td>byte3 byte2 byte1 byte0</td>
</tr>
<tr>
<td>00b:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01b:</td>
<td>byte4</td>
<td>byte3 byte2 byte1</td>
</tr>
<tr>
<td>10b:</td>
<td>byte5</td>
<td>byte4 byte3 byte2</td>
</tr>
<tr>
<td>11b:</td>
<td></td>
<td>byte3</td>
</tr>
</tbody>
</table>

This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.
Instruction Overview

Options

The (R) syntax reverses the order of the source registers within each register pair. Typical high performance applications cannot afford the overhead of reloading both register pair operands to maintain byte order for every calculation. Instead, they alternate and load only one register pair operand each time and alternate between the forward and reverse byte order versions of this instruction. By default, the low order bytes come from the low register in the register pair. The (R) option causes the low order bytes to come from the high register.

In the optional reverse source order case (for example, using the (R) syntax), the only difference is the source registers swap places within the register pair in their byte ordering. Assume that a source register pair contains the data shown in Table 18-27.

Table 18-27. I-register Bits and the Byte Alignment

<table>
<thead>
<tr>
<th>Two LSB’s of I0 or I1</th>
<th>src_reg_pair_LO</th>
<th>src_reg_pair_HI</th>
</tr>
</thead>
<tbody>
<tr>
<td>00b:</td>
<td>byte7 byte6 byte5 byte4 byte3 byte2 byte1 byte0</td>
<td>byte3 byte2 byte1 byte0</td>
</tr>
<tr>
<td>01b:</td>
<td>byte7 byte6 byte5 byte4 byte3 byte2 byte1 byte0</td>
<td>byte4 byte3 byte2 byte1</td>
</tr>
<tr>
<td>10b:</td>
<td>byte7 byte6 byte5 byte4 byte3 byte2 byte1 byte0</td>
<td>byte5 byte4 byte3 byte2</td>
</tr>
<tr>
<td>11b:</td>
<td>byte7 byte6 byte5 byte4 byte3 byte2 byte1 byte0</td>
<td>byte6 byte5 byte4 byte3</td>
</tr>
</tbody>
</table>

The mnemonic derives its name from the fact that the operands are bytes, the result is 16 bits, and the arithmetic operation is “minus” for subtraction.
Video Pixel Operations

Flags Affected

None

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

(r1,r2)= byteop16m (r3:2,r1:0) ;
(r1,r2)= byteop16m (r3:2,r1:0) (r) ;

Also See

BYTEOP16P (Quad 8-Bit Add)

Special Applications

This instruction provides packed data arithmetic typical of video and image processing applications.
Instruction Overview

SAA (Quad 8-Bit Subtract-Absolute-Accumulate)

General Form

SAA ( src_reg_0, src_reg_1 )
SAA ( src_reg_0, src_reg_1 ) (R)

Syntax

SAA (Dreg_pair, Dreg_pair) ; /* forward byte order operands (b) */
SAA (Dreg_pair, Dreg_pair) (R) ; /* reverse byte order operands (b) */

Syntax Terminology

Dreg_pair: R1:0, R3:2 (This instruction only supports register pairs R1:0 and R3:2.)

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Quad 8-Bit Subtract-Absolute-Accumulate instruction subtracts four pairs of values, takes the absolute value of each difference, and accumulates each result into a 16-bit Accumulator half. The results are placed in the upper- and lower-half Accumulators A0.H, A0.L, A1.H, and A1.L.

Saturation is performed if an operation overflows a 16-bit Accumulator half.

Only register pairs R1:0 and R3:2 are valid sources for this instruction.

This instruction supports the following byte-wise Sum of Absolute Difference (SAD) calculations.
Video Pixel Operations

\[ SAD = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} |a(i,j) - b(i,j)| \]

Figure 18-1. Absolute Difference (SAD) Calculations

Typical values for N are 8 and 16, corresponding to the video block size of 8x8 and 16x16 pixels, respectively. The 16-bit Accumulator registers limit the pixel region or block size to 32x32 pixels.

The SAA instruction behavior is shown below.

Table 18-28. SAA Instruction Behavior

<table>
<thead>
<tr>
<th>src_reg_0</th>
<th>src_reg_1</th>
<th>A1.H</th>
<th>A1.L</th>
<th>A0.H</th>
<th>A0.L</th>
</tr>
</thead>
<tbody>
<tr>
<td>a(i, j+3)</td>
<td>b(i, j+3)</td>
<td>+=</td>
<td>a(i, j+3) - b(i, j+3)</td>
<td>+=</td>
<td>a(i, j+2) - b(i, j+2)</td>
</tr>
<tr>
<td>a(i, j+2)</td>
<td>b(i, j+2)</td>
<td>A1.L</td>
<td></td>
<td>A0.H</td>
<td></td>
</tr>
<tr>
<td>a(i, j+1)</td>
<td>b(i, j+1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a(i, j)</td>
<td>b(i, j)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Quad 8-Bit Subtract-Absolute-Accumulate instruction provides byte alignment directly in the source register pairs `src_reg_0` and `src_reg_1` based on index registers `I0` and `I1`.

- The two LSBs of the `I0` register determine the byte alignment for source register pair `src_reg_0` (typically R1:0).
- The two LSBs of the `I1` register determine the byte alignment for source register pair `src_reg_1` (typically R3:2).

The relationship between the I-register bits and the byte alignment is illustrated in Table 18-29.
In the default source order case (for example, not the (R) syntax), assume a source register pair contain the data shown in Table 18-29.

Table 18-29. I-register Bits and the Byte Alignment

<table>
<thead>
<tr>
<th>Two LSB’s of I0 or I1</th>
<th>src_reg_pair_HI</th>
<th>src_reg_pair_LO</th>
</tr>
</thead>
<tbody>
<tr>
<td>00b:</td>
<td>byte7 byte6 byte5 byte4</td>
<td>byte3 byte2 byte1 byte0</td>
</tr>
<tr>
<td>01b:</td>
<td>byte5 byte4</td>
<td>byte3 byte2 byte1</td>
</tr>
<tr>
<td>10b:</td>
<td>byte6 byte5 byte4</td>
<td>byte3</td>
</tr>
<tr>
<td>11b:</td>
<td>byte6</td>
<td>byte5 byte4 byte3</td>
</tr>
</tbody>
</table>

This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

Options

The (R) syntax reverses the order of the source registers within each pair. Typical high performance applications cannot afford the overhead of reloading both register pair operands to maintain byte order for every calculation. Instead, they alternate and load only one register pair operand each time and alternate between the forward and reverse byte order versions of this instruction. By default, the low order bytes come from the low register in the register pair. The (R) option causes the low order bytes to come from the high register.

When reversing source order by using the (R) syntax, the source registers swap places within the register pair in their byte ordering. If a source register pair contains the data shown in Table 18-30, then the SAA instruction computes 12 pixel operations simultaneously—the three-operation subtract-absolute-accumulate on four pairs of operand bytes in parallel.
Video Pixel Operations

Table 18-30. I-register Bits and the Byte Alignment

<table>
<thead>
<tr>
<th>The bytes selected are</th>
<th>src_reg_pair_LO</th>
<th>src_reg_pair_HI</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Two LSB’s of I0 or I1</strong></td>
<td>byte7 byte6 byte5 byte4</td>
<td>byte3 byte2 byte1 byte0</td>
</tr>
<tr>
<td>00b:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01b:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10b:</td>
<td></td>
<td>byte5 byte4 byte3 byte2</td>
</tr>
<tr>
<td>11b:</td>
<td>byte6 byte5 byte4 byte3</td>
<td></td>
</tr>
</tbody>
</table>

Flags Affected

None

The ADSP-BF535 processor has fewer `ASTAT` flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

```
  saa (r1:0, r3:2) || r0 = [i0++] || r2 = [i1++] ; /* parallel fill instructions */
  saa (r1:0, r3:2) (R) || r1 = [i0++] || r3 = [i1++] ; /* reverse, parallel fill instructions */
  saa (r1:0, r3:2) ; /* last SAA in a loop, no more fill required */
```
Instruction Overview

Also See

DISALGNEXCPT, Load Data Register

Special Applications

Use the Quad 8-Bit Subtract-Absolute-Accumulate instruction for block-based video motion estimation algorithms using block Sum of Absolute Difference (SAD) calculations to measure distortion.
Video Pixel Operations

BYTEUNPACK (Quad 8-Bit Unpack)

General Form

\[
(\text{dest}_\text{reg}_1, \text{dest}_\text{reg}_0) = \text{BYTEUNPACK} \text{ src}_\text{reg}_\text{pair}
\]

\[
(\text{dest}_\text{reg}_1, \text{dest}_\text{reg}_0) = \text{BYTEUNPACK} \text{ src}_\text{reg}_\text{pair} (R)
\]

Syntax

\[
(\text{Dreg}_1, \text{Dreg}_0) = \text{BYTEUNPACK} \text{ Dreg}_\text{pair} ; /* (b) */
\]

\[
(\text{Dreg}_1, \text{Dreg}_0) = \text{BYTEUNPACK} \text{ Dreg}_\text{pair} (R) ; /* reverse source order (b) */
\]

Syntax Terminology

\text{Dreg}: R7-0

\text{Dreg}_\text{pair}: \text{R1:0, R3:2, only}

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Quad 8-Bit Unpack instruction copies four contiguous bytes from a pair of source registers, adjusting for byte alignment. The instruction loads the selected bytes into two arbitrary data registers on half-word alignment.

The two LSBs of the I0 register determine the source byte alignment, as illustrated in Table 18-31.

In the default source order case (for example, not the (R) syntax), assume the source register pair contains the data shown in Table 18-31.

This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.
Instruction Overview

Table 18-31. I-register Bits and the Byte Alignment

<table>
<thead>
<tr>
<th>Two LSB's of I0 or I1</th>
<th>src_reg_pair_HI</th>
<th>src_reg_pair_LO</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte7</td>
<td>byte6</td>
<td>byte5</td>
</tr>
<tr>
<td>00b:</td>
<td>byte3</td>
<td>byte2</td>
</tr>
<tr>
<td>01b:</td>
<td>byte4</td>
<td>byte3</td>
</tr>
<tr>
<td>10b:</td>
<td>byte5</td>
<td>byte4</td>
</tr>
<tr>
<td>11b:</td>
<td>byte6</td>
<td>byte5</td>
</tr>
</tbody>
</table>

Options

The (R) syntax reverses the order of the source registers within the pair. Typical high performance applications cannot afford the overhead of reloading both register pair operands to maintain byte order for every calculation. Instead, they alternate and load only one register pair operand each time and alternate between the forward and reverse byte order versions of this instruction. By default, the low order bytes come from the low register in the register pair. The (R) option causes the low order bytes to come from the high register.

In the optional reverse source order case (for example, using the (R) syntax), the only difference is the source registers swap places in their byte ordering. Assume the source register pair contains the data shown in Table 18-32.

Table 18-32. I-register Bits and the Byte Alignment

<table>
<thead>
<tr>
<th>Two LSB's of I0 or I1</th>
<th>src_reg_pair_LO</th>
<th>src_reg_pair_HI</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte7</td>
<td>byte6</td>
<td>byte5</td>
</tr>
<tr>
<td>00b:</td>
<td>byte3</td>
<td>byte2</td>
</tr>
<tr>
<td>01b:</td>
<td>byte4</td>
<td>byte3</td>
</tr>
<tr>
<td>10b:</td>
<td>byte5</td>
<td>byte4</td>
</tr>
<tr>
<td>11b:</td>
<td>byte6</td>
<td>byte5</td>
</tr>
</tbody>
</table>
The four bytes, now byte aligned, are copied into the destination registers on half-word alignment, as shown in Table 18-33 and Table 18-34.

Table 18-33. Source Register Contains

<table>
<thead>
<tr>
<th>31.................24</th>
<th>23.................16</th>
<th>15.................8</th>
<th>7.................0</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte_D</td>
<td>byte_C</td>
<td>byte_B</td>
<td>byte_A</td>
</tr>
</tbody>
</table>

Aligned bytes: byte_D byte_C byte_B byte_A

Table 18-34. Destination Registers Receive

<table>
<thead>
<tr>
<th>31.................24</th>
<th>23.................16</th>
<th>15.................8</th>
<th>7.................0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>byte_B</td>
<td></td>
<td>byte_A</td>
</tr>
<tr>
<td>dest_reg_0:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dest_reg_1:</td>
<td>byte_D</td>
<td>byte_C</td>
<td></td>
</tr>
</tbody>
</table>

Only register pairs R1:0 and R3:2 are valid sources for this instruction.

Misaligned access exceptions are disabled during this instruction.

Flags Affected

None

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.
Instruction Overview

Example

\[(r6,r5) = \text{byteunpack } r1:0 : \quad /* \text{non-reversing sources} */\]

- Assuming:
  - register 10’s two LSBs = 00b,
  - \(R1 = 0xFEED\) FACE
  - \(R0 = 0xBEEF\) BADD

then this instruction returns:
  - \(R6 = 0x00BE\) 00EF
  - \(R5 = 0x00BA\) 00DD

- Assuming:
  - register 10’s two LSBs = 01b,
  - \(R1 = 0xFEED\) FACE
  - \(R0 = 0xBEEF\) BADD

then this instruction returns:
  - \(R6 = 0x00CE\) 00BE
  - \(R5 = 0x00EF\) 00BA
Video Pixel Operations

- Assuming:
  - register R0’s two LSBs = 10b,
  - R1 = 0xFEED FACE
  - R0 = 0xBEEF BADD
  then this instruction returns:
  - R6 = 0x00FA 00CE
  - R5 = 0x00BE 00EF

- Assuming:
  - register R0’s two LSBs = 11b,
  - R1 = 0xFEED FACE
  - R0 = 0xBEEF BADD
  then this instruction returns:
  - R6 = 0x00ED 00FA
  - R5 = 0x00CE 00BE

(r6,r5) = byteunpack r1:0 (R); /* reversing sources case */
Instruction Overview

• Assuming:
  • register 10’s two LSBs = 00b,
  • R1 = 0xFEED FACE
  • R0 = 0xBEEF BADD

then this instruction returns:
  • R6 = 0x00FE 00ED
  • R5 = 0x00FA 00CE

• Assuming:
  • register 10’s two LSBs = 01b,
  • R1 = 0xFEED FACE
  • R0 = 0xBEEF BADD

then this instruction returns:
  • R6 = 0x00DD 00FE
  • R5 = 0x00ED 00FA

• Assuming:
  • register 10’s two LSBs = 10b,
  • R1 = 0xFEED FACE
  • R0 = 0xBEEF BADD

then this instruction returns:
  • R6 = 0x00BA 00DD
  • R5 = 0x00FE 00ED
Video Pixel Operations

- Assuming:
  - register \( I0 \)'s two LSBs = 11b,
  - \( R1 = 0xFEED \text{ FACE} \)
  - \( R0 = 0xBEEF \text{ BADD} \)

then this instruction returns:
  - \( R6 = 0x00EF \text{ 00BA} \)
  - \( R5 = 0x00DD \text{ 00FE} \)

Also See

BYTEPACK (Quad 8-Bit Pack)

Special Applications

None
Instruction Overview
19 VECTOR OPERATIONS

Instruction Summary

- “Add on Sign” on page 19-3
- “VIT_MAX (Compare-Select)” on page 19-8
- “Vector ABS” on page 19-15
- “Vector Add / Subtract” on page 19-18
- “Vector Arithmetic Shift” on page 19-23
- “Vector Logical Shift” on page 19-28
- “Vector MAX” on page 19-32
- “Vector MIN” on page 19-35
- “Vector Multiply” on page 19-38
- “Vector Multiply and Multiply-Accumulate” on page 19-41
- “Vector Negate (Two’s-Complement)” on page 19-46
- “Vector PACK” on page 19-48
- “Vector SEARCH” on page 19-50
Instruction Overview

Instruction Overview

This chapter discusses the instructions that control vector operations. Users can take advantage of these instructions to perform simultaneous operations on multiple 16-bit values, including add, subtract, multiply, shift, negate, pack, and search. Compare-Select and Add-On-Sign are also included in this chapter.
Vector Operations

Add on Sign

General Form

\[
\text{dest}_\text{hi} = \text{dest}_\text{lo} = \text{SIGN} (\text{src}_0\text{hi}) \times \text{src}_1\text{hi} \\
+ \text{SIGN} (\text{src}_0\text{lo}) \times \text{src}_1\text{lo}
\]

Syntax

\[
\text{Dreg}_\text{hi} = \text{Dreg}_\text{lo} = \text{SIGN} (\text{Dreg}_\text{hi}) \times \text{Dreg}_\text{hi} \\
+ \text{SIGN} (\text{Dreg}_\text{lo}) \times \text{Dreg}_\text{lo};
\]

/* (b) */

Register Consistency

The destination registers \( \text{dest}_\text{hi} \) and \( \text{dest}_\text{lo} \) must be halves of the same data register. Similarly, \( \text{src}_0\text{hi} \) and \( \text{src}_0\text{lo} \) must be halves of the same register and \( \text{src}_1\text{hi} \) and \( \text{src}_1\text{lo} \) must be halves of the same register.

Syntax Terminology

\( \text{Dreg}_\text{hi} : \text{R7-0.H} \)

\( \text{Dreg}_\text{lo} : \text{R7-0.L} \)

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.
Instruction Overview

Functional Description

The Add on Sign instruction performs a two step function, as follows.

1. Multiply the arithmetic sign of a 16-bit half-word number in \( \text{src0} \) by the corresponding half-word number in \( \text{src1} \). The arithmetic sign of \( \text{src0} \) is either (+1) or (–1), depending on the sign bit of \( \text{src0} \). The instruction performs this operation on the upper and lower half-words of the same data registers.

The results of this step obey the signed multiplication rules summarized in Table 19-1. \( Y \) is the number in \( \text{src0} \), and \( Z \) is the number in \( \text{src1} \). The numbers in \( \text{src0} \) and \( \text{src1} \) may be positive or negative.

Table 19-1. Signed Multiplication Rules

<table>
<thead>
<tr>
<th>SRC0</th>
<th>SRC1</th>
<th>Sign-Adjusted SRC1</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Y</td>
<td>+Z</td>
<td>+Z</td>
</tr>
<tr>
<td>+Y</td>
<td>–Z</td>
<td>–Z</td>
</tr>
<tr>
<td>–Y</td>
<td>+Z</td>
<td>–Z</td>
</tr>
<tr>
<td>–Y</td>
<td>–Z</td>
<td>+Z</td>
</tr>
</tbody>
</table>

Note the result always bears the magnitude of \( Z \) with only the sign affected.

2. Then, add the sign-adjusted \( \text{src1} \) upper and lower half-word results together and store the same 16-bit sum in the upper and lower halves of the destination register, as shown in Table 19-2 and Table 19-3.

The sum is not saturated if the addition exceeds 16 bits.
Vector Operations

Table 19-2. Source Registers Contain

<table>
<thead>
<tr>
<th>src0:</th>
<th>31...........24</th>
<th>23...........16</th>
<th>15...........8</th>
<th>7...........0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>a1</td>
<td>a0</td>
<td></td>
</tr>
<tr>
<td>src1:</td>
<td></td>
<td>b1</td>
<td>b0</td>
<td></td>
</tr>
</tbody>
</table>

Table 19-3. Destination Register Receives

<table>
<thead>
<tr>
<th>dest:</th>
<th>31...........24</th>
<th>23...........16</th>
<th>15...........8</th>
<th>7...........0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(sign_adjusted_b1) + (sign_adjusted_b0)</td>
<td>(sign_adjusted_b1) + (sign_adjusted_b0)</td>
<td></td>
</tr>
</tbody>
</table>

Flags Affected

None

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.
Instruction Overview

Example

\[
r7.h = r7.l = \text{sign}(r2.h) \times r3.h + \text{sign}(r2.l) \times r3.l
\]

- If
  - \( R2.H = 2 \)
  - \( R3.H = 23 \)
  - \( R2.L = 2001 \)
  - \( R3.L = 1234 \)

  then
  - \( R7.H = 1257 \) (or \( 1234 + 23 \))
  - \( R7.L = 1257 \)

- If
  - \( R2.H = -2 \)
  - \( R3.H = 23 \)
  - \( R2.L = 2001 \)
  - \( R3.L = 1234 \)

  then
  - \( R7.H = 1211 \) (or \( 1234 - 23 \))
  - \( R7.L = 1211 \)
Vector Operations

- If
  - $R2.H = 2$
  - $R3.H = 23$
  - $R2.L = -2001$
  - $R3.L = 1234$
  then
    - $R7.H = -1211$ (or $(-1234) + 23$)
    - $R7.L = -1211$
  
- If
  - $R2.H = -2$
  - $R3.H = 23$
  - $R2.L = -2001$
  - $R3.L = 1234$
  then
    - $R7.H = -1257$ (or $(-1234) - 23$)
    - $R7.L = -1257$

Also See

None

Special Applications

Use the Sum on Sign instruction to compute the branch metric used by each Viterbi Butterfly.
Instruction Overview

VIT_MAX (Compare-Select)

General Form

\[
\begin{align*}
\text{dest}_\text{reg} &= \text{VIT_MAX} (\text{src}_\text{reg}_0, \text{src}_\text{reg}_1) \ (\text{ASL}) \\
\text{dest}_\text{reg} &= \text{VIT_MAX} (\text{src}_\text{reg}_0, \text{src}_\text{reg}_1) \ (\text{ASR}) \\
\text{dest}_\text{reg}_\text{lo} &= \text{VIT_MAX} (\text{src}_\text{reg}) \ (\text{ASL}) \\
\text{dest}_\text{reg}_\text{lo} &= \text{VIT_MAX} (\text{src}_\text{reg}) \ (\text{ASR})
\end{align*}
\]

Syntax

Dual 16-Bit Operation

\[
\begin{align*}
\text{Dreg} &= \text{VIT_MAX} (\text{Dreg} , \text{Dreg}) \ (\text{ASL}) ; \quad /* \text{shift history bits left (b) */} \\
\text{Dreg} &= \text{VIT_MAX} (\text{Dreg} , \text{Dreg}) \ (\text{ASR}) ; \quad /* \text{shift history bits right (b) */}
\end{align*}
\]

Single 16-Bit Operation

\[
\begin{align*}
\text{Dreg}_\text{lo} &= \text{VIT_MAX} (\text{Dreg}) \ (\text{ASL}) ; \quad /* \text{shift history bits left (b) */} \\
\text{Dreg}_\text{lo} &= \text{VIT_MAX} (\text{Dreg}) \ (\text{ASR}) ; \quad /* \text{shift history bits right (b) */}
\end{align*}
\]

Syntax Terminology

\[
\begin{align*}
\text{Dreg}: \text{R7-0} \\
\text{Dreg}_\text{lo}: \text{R7-0.L}
\end{align*}
\]

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.
Vector Operations

Functional Description

The Compare-Select (VIT_MAX) instruction selects the maximum values of pairs of 16-bit operands, returns the largest values to the destination register, and serially records in A0.W the source of the maximum. This operation performs signed operations. The operands are compared as two’s-complements.

Versions are available for dual and single 16-bit operations. Whereas the dual versions compare four operands to return two maxima, the single versions compare only two operands to return one maximum.

The Accumulator extension bits (bits 39–32) must be cleared before executing this instruction.

This operation is illustrated in Table 19-4 and Table 19-5.

Table 19-4. Source Registers Contain

<table>
<thead>
<tr>
<th>src_reg_0</th>
<th>31..............24</th>
<th>23..............16</th>
<th>15..............8</th>
<th>7..............0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>src_reg_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| src_reg_0 | z1               |                  |                 |                 |
| src_reg_1 |                  |                  |                 |                 |

Table 19-5. Destination Register Contains

<table>
<thead>
<tr>
<th>dest_reg</th>
<th>31..............24</th>
<th>23..............16</th>
<th>15..............8</th>
<th>7..............0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Maximum, y1 or y0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Maximum, z1 or z0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dual 16-Bit Operand Behavior

The ASL version shifts A0 left two bit positions and appends two LSBs to indicate the source of each maximum as shown in Table 19-6 and Table 19-7.
Conversely, the ASR version shifts $A_0$ right two bit positions and appends two MSBs to indicate the source of each maximum as shown in Table 19-8 and Table 19-9.

Table 19-8. ASR Version Shifts

$$
\begin{array}{|c|c|}
\hline
A_0.X & A_0.W \\
\hline
A0 & 00000000 \quad BBXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX \\
\hline
\end{array}
$$

Table 19-9. Where

<table>
<thead>
<tr>
<th>BB</th>
<th>Indicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>$y_0$ and $z_0$ are maxima</td>
</tr>
<tr>
<td>01</td>
<td>$y_0$ and $z_1$ are maxima</td>
</tr>
<tr>
<td>10</td>
<td>$y_1$ and $z_0$ are maxima</td>
</tr>
<tr>
<td>11</td>
<td>$y_1$ and $z_1$ are maxima</td>
</tr>
</tbody>
</table>
Vector Operations

Notice that the history bit code depends on the \( A0 \) shift direction. The bit for \( src_{\_reg\_1} \) is always shifted onto \( A0 \) first, followed by the bit for \( src_{\_reg\_0} \).

The single operand versions behave similarly.

**Single 16-Bit Operand Behavior**

If the dual source register contains the data shown in Table 19-10 the destination register receives the data shown in Table 19-11.

Table 19-10. Source Registers Contain

<table>
<thead>
<tr>
<th>src_reg</th>
<th>31...........24 23............16 15............8 7............0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y1</td>
</tr>
<tr>
<td></td>
<td>y0</td>
</tr>
</tbody>
</table>

Table 19-11. Destination Register Contains

<table>
<thead>
<tr>
<th>dest_reg_lo</th>
<th>31...........24 23............16 15............8 7............0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Maximum, y1 or y0</td>
</tr>
</tbody>
</table>

The ASL version shifts \( A0 \) left one bit position and appends an LSB to indicate the source of the maximum.

Table 19-12. ASL Version Shifts

<table>
<thead>
<tr>
<th>A0 ( .X )</th>
<th>A0 ( .W )</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXBB</td>
</tr>
</tbody>
</table>

Conversely, the ASR version shifts \( A0 \) right one bit position and appends an MSB to indicate the source of the maximum.
Instruction Overview

Table 19-13. ASR Version Shifts

<table>
<thead>
<tr>
<th>A0.X</th>
<th>A0.W</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Table 19-14. Where

<table>
<thead>
<tr>
<th>B</th>
<th>Indicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>y0 is the maximum</td>
</tr>
<tr>
<td>1</td>
<td>y1 is the maximum</td>
</tr>
</tbody>
</table>

The path metrics are allowed to overflow, and maximum comparison is done on the two’s-complement circle. Such comparison gives a better indication of the relative magnitude of two large numbers when a small number is added/subtracted to both.

Flags Affected

None

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.
Vector Operations

Example

\[ r_5 = \text{vit}\_\text{max}(r_3, r_2)(\text{asl}) ; /* \text{shift left, dual operation} */ \]

- Assume:
  - \( R_3 = 0xFFFF\ 0000 \)
  - \( R_2 = 0x0000\ FFFF \)
  - \( A_0 = 0x00\ 0000\ 0000 \)

This example produces:
  - \( R_5 = 0x0000\ 0000 \)
  - \( A_0 = 0x00\ 0000\ 0002 \)

\[ r_7 = \text{vit}\_\text{max}(r_1, r_0)(\text{asr}) ; /* \text{shift right, dual operation} */ \]

- Assume:
  - \( R_1 = 0xFEED\ BEEF \)
  - \( R_0 = 0xDEAF\ 0000 \)
  - \( A_0 = 0x00\ 0000\ 0000 \)

This example produces:
  - \( R_7 = 0xFEED\ 0000 \)
  - \( A_0 = 0x00\ 8000\ 0000 \)
Instruction Overview

r3.l = vit_max (r1)(asl) ; /* shift left, single operation */

- Assume:
  - R1 = 0xFFFF 0000
  - A0 = 0x00 0000 0000

  This example produces:
  - R3.L = 0x0000
  - A0 = 0x00 0000 0000

r3.l = vit_max (r1)(asr) ;  /* shift right, single operation */

- Assume:
  - R1 = 0x1234 FADE
  - A0 = 0x00 FFFF FFFF

  This example produces:
  - R3.L = 0x1234
  - A0 = 0x00 7FFF FFFF

Also See

MAX

Special Applications

The Compare-Select (VIT_MAX) instruction is a key element of the 
Add-Compare-Select (ACS) function for Viterbi decoders. Combine it 
with a Vector Add instruction to calculate a trellis butterfly used in ACS 
functions.
Vector Operations

Vector ABS

General Form

\[ \text{dest} = \text{ABS source} \]

Syntax

\[ Dreg = \text{ABS Dreg} \] ; /* (b) */

Syntax Terminology

\( Dreg: R7-0 \)

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Vector Absolute Value instruction calculates the individual absolute values of the upper and lower halves of a single 32-bit data register. The results are placed into a 32-bit \( \text{dest} \), using the following rules.

- If the input value is positive or zero, copy it unmodified to the destination.
- If the input value is negative, subtract it from zero and store the result in the destination.

For example, if the source register contains the data shown in Table 19-15 the destination register receives the data shown in Table 19-16.

Table 19-15. Source Registers Contain

<table>
<thead>
<tr>
<th>src_reg: 31..............24</th>
<th>23..............16</th>
<th>15..............8</th>
<th>7..............0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.h</td>
<td>x.l</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 19-16. Destination Registers Contain

<table>
<thead>
<tr>
<th>dest_reg: 31..............24</th>
<th>23..............16</th>
<th>15..............8</th>
<th>7..............0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.h</td>
<td>x.l</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Instruction Overview

Table 19-16. Destination Register Contains

<table>
<thead>
<tr>
<th>31............24</th>
<th>23............16</th>
<th>15............8</th>
<th>7............0</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest_reg:</td>
<td>x.h</td>
<td>x.l</td>
<td></td>
</tr>
</tbody>
</table>

This instruction saturates the result.

Flags Affected

This instruction affects flags as follows.

- \(AZ\) is set if either or both result is zero; cleared if both are nonzero.
- \(AN\) is cleared.
- \(V\) is set if either or both result saturates; cleared if both are no saturation.
- \(VS\) is set if \(V\) is set; unaffected otherwise.
- All other flags are unaffected.

⚠️ The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.
Vector Operations

Example

/* If r1 = 0xFFFF 7FFF, then . . . */
r3 = abs r1 (v);
/* . . . produces 0x0001 7FFF */

Also See

ABS

Special Applications

None
Instruction Overview

Vector Add / Subtract

General Form

\[ \text{dest} = \text{src}_\text{reg}_0 +|+ \text{src}_\text{reg}_1 \]
\[ \text{dest} = \text{src}_\text{reg}_0 -|+ \text{src}_\text{reg}_1 \]
\[ \text{dest} = \text{src}_\text{reg}_0 +|– \text{src}_\text{reg}_1 \]
\[ \text{dest} = \text{src}_\text{reg}_0 –|– \text{src}_\text{reg}_1 \]
\[ \text{dest}_0 = \text{src}_\text{reg}_0 +|+ \text{src}_\text{reg}_1, \]
\[ \text{dest}_1 = \text{src}_\text{reg}_0 –|– \text{src}_\text{reg}_1 \]
\[ \text{dest}_0 = \text{src}_\text{reg}_0 +|– \text{src}_\text{reg}_1, \]
\[ \text{dest}_1 = \text{src}_\text{reg}_0 –|+ \text{src}_\text{reg}_1 \]
\[ \text{dest}_0 = \text{src}_\text{reg}_0 + \text{src}_\text{reg}_1, \]
\[ \text{dest}_1 = \text{src}_\text{reg}_0 – \text{src}_\text{reg}_1 \]
\[ \text{dest}_0 = \text{A}_1 + \text{A}_0, \text{dest}_1 = \text{A}_1 – \text{A}_0 \]
\[ \text{dest}_0 = \text{A}_0 + \text{A}_1, \text{dest}_1 = \text{A}_0 – \text{A}_1 \]

Syntax

Dual 16-Bit Operations

\[ \text{Dreg} = \text{Dreg} +|+ \text{Dreg} \text{ (opt\_mode\_0)} ; /* \text{add} | \text{add (b)} */ \]
\[ \text{Dreg} = \text{Dreg} -|+ \text{Dreg} \text{ (opt\_mode\_0)} ; /* \text{subtract} | \text{add (b)} */ \]
\[ \text{Dreg} = \text{Dreg} +|– \text{Dreg} \text{ (opt\_mode\_0)} ; /* \text{add} | \text{subtract (b)} */ \]
\[ \text{Dreg} = \text{Dreg} –|+ \text{Dreg} \text{ (opt\_mode\_0)} ; /* \text{subtract} | \text{subtract (b)} */ \]

Quad 16-Bit Operations

\[ \text{Dreg} = \text{Dreg} +|+ \text{Dreg}, \text{Dreg} = \text{Dreg} –|– \text{Dreg} \text{ (opt\_mode\_0, opt\_mode\_2)} ; /* \text{add} | \text{add, subtract} | \text{subtract}; the set of source registers must be the same for each operation (b) */ \]
\[ \text{Dreg} = \text{Dreg} +|– \text{Dreg}, \text{Dreg} = \text{Dreg} –|+ \text{Dreg} \text{ (opt\_mode\_0, opt\_mode\_2)} ; /* \text{add} | \text{subtract, subtract} | \text{add}; the set of source registers must be the same for each operation (b) */ \]
Vector Operations

Dual 32-Bit Operations

\[ \text{Dreg} = \text{Dreg} + \text{Dreg}, \quad \text{Dreg} = \text{Dreg} - \text{Dreg} \quad (\text{opt\_mode\_1}) ; \]
/* add, subtract; the set of source registers must be the same for each operation (b) */

Dual 40-Bit Accumulator Operations

\[ \text{Dreg} = \text{A1} + \text{A0}, \quad \text{Dreg} = \text{A1} - \text{A0} \quad (\text{opt\_mode\_1}) ; \]
/* add, subtract Accumulators; subtract A0 from A1 (b) */
\[ \text{Dreg} = \text{A0} + \text{A1}, \quad \text{Dreg} = \text{A0} - \text{A1} \quad (\text{opt\_mode\_1}) ; \]
/* add, subtract Accumulators; subtract A1 from A0 (b) */

Syntax Terminology

\text{Dreg}: \text{R7-0}

\text{opt\_mode\_0}: \text{optional (S), (CO), or (SCO)}

\text{opt\_mode\_1}: \text{optional (S)}

\text{opt\_mode\_2}: \text{optional (ASR), or (ASL)}

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Vector Add / Subtract instruction simultaneously adds and/or subtracts two pairs of registered numbers. It then stores the results of each operation into a separate 32-bit data register or 16-bit half register, according to the syntax used. The destination register for each of the quad or dual versions must be unique.
Options

The Vector Add / Subtract instruction provides three option modes.

- `opt_mode_0` supports the Dual and Quad 16-Bit Operations versions of this instruction.
- `opt_mode_1` supports the Dual 32-bit and 40-bit operations.
- `opt_mode_2` supports the Quad 16-Bit Operations versions of this instruction.

Table 19-17 describes the options that the three `opt_modes` support.

Table 19-17. Options for Opt_Mode 0

<table>
<thead>
<tr>
<th>Mode</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>opt_mode_0</code></td>
<td>S</td>
<td>Saturate the results at 16 bits.</td>
</tr>
<tr>
<td></td>
<td>CO</td>
<td>Cross option. Swap the order of the results in the destination register.</td>
</tr>
<tr>
<td></td>
<td>SCO</td>
<td>Saturate and cross option. Combination of (S) and (CO) options.</td>
</tr>
<tr>
<td><code>opt_mode_1</code></td>
<td>S</td>
<td>Saturate the results at 16 or 32 bits, depending on the operand size.</td>
</tr>
<tr>
<td><code>opt_mode_2</code></td>
<td>ASR</td>
<td>Arithmetic shift right. Halve the result (divide by 2) before storing in the destination register. If specified with the S (saturation) flag in Quad 16-Bit Operand versions of this instruction, the scaling is performed before saturation for the ADSP-BF533 processor, and the scaling is performed after saturation for the ADSP-BF535 processor.</td>
</tr>
<tr>
<td></td>
<td>ASL</td>
<td>Arithmetic shift left. Double the result (multiply by 2, truncated) before storing in the destination register. If specified with the S (saturation) flag in Quad 16-Bit Operand versions of this instruction, the scaling is performed before saturation for the ADSP-BF533 processor, and the scaling is performed after saturation for the ADSP-BF535 processor.</td>
</tr>
</tbody>
</table>

The options shown for `opt_mode_2` are scaling options.
Vector Operations

Flags Affected

This instruction affects the following flags.

- **AZ** is set if any results are zero; cleared if all are nonzero.
- **AN** is set if any results are negative; cleared if all non-negative.
- **AC0** is set if the right-hand side of a dual operation generates a carry; cleared if no carry; unaffected if a quad operation.
- **AC1** is set if the left-hand side of a dual operation generates a carry; cleared if no carry; unaffected if a quad operation.
- **V** is set if any results overflow; cleared if none overflows.
- **VS** is set if **V** is set; unaffected otherwise.
- All other flags are unaffected.

- The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

```
r5 = r3 ++ r4 ; /* dual 16-bit operations, add|add */
r6 = r0 -- r1(s) ; /* same as above, subtract|add with saturation */
```
Instruction Overview

r0=r2 +|- r1(co) ; /* add|subtract with half-word results
crossed over in the destination register */
r7=r3 -|- r6(sco) ; /* subtract|subtract with saturation and
half-word results crossed over in the destination register */
r5=r3 +|+ r4, r7=r3-|-r4 ; /* quad 16-bit operations, add|add,
subtract|subtract */
r5=r3 +|- r4, r7=r3 -|+ r4 ; /* quad 16-bit operations,
add|subtract, subtract|add */
r5=r3 +| r4, r7=r3 -| r4(asr) ; /* quad 16-bit operations,
add|subtract, subtract|add, with all results divided by 2 (right
shifted 1 place) before storing into destination register */
r5=r3 +| r4, r7=r3 -| r4(asl) ; /* quad 16-bit operations,
add|subtract, subtract|add, with all results multiplied by 2
(left shifted 1 place) before storing into destination register
dual */
r2=r0+r1, r3=r0-r1 ; /* 32-bit operations */
r2=r0+r1, r3=r0-r1(s) ; /* dual 32-bit operations with
saturation */
r4=a1+a0, r6=a1-a0 ; /* dual 40-bit Accumulator operations, A0
subtracted from A1 */
r4=a0+a1, r6=a0-a1(s) ; /* dual 40-bit Accumulator operations
with saturation, A1 subtracted from A0 */

Also See

Add, Subtract

Special Applications

FFT butterfly routines in which each of the registers is considered a single
complex number often use the Vector Add / Subtract instruction.

/* If r1 = 0x0003 0004 and r2 = 0x0001 0002, then . . . */
r0 = r2 +|- r1(co) ;
/* . . . produces r0 = 0xFFFE 0004 */
Vector Operations

Vector Arithmetic Shift

General Form

\[
\text{dest\_reg} = \text{src\_reg} \gg \text{shift\_magnitude\ (V)} \\
\text{dest\_reg} = \text{ASHIFT src\_reg BY shift\_magnitude\ (V)}
\]

Syntax

Constant Shift Magnitude

\[
\text{Dreg} = \text{Dreg} \gg \text{uimm4\ (V)} ; /* \text{arithmetic shift right, immediate (b)} */ \\
\text{Dreg} = \text{Dreg} \ll \	ext{uimm4\ (V,S)} ; /* \text{arithmetic shift left, immediate with saturation (b)} */
\]

Registered Shift Magnitude

\[
\text{Dreg} = \text{ASHIFT Dreg BY Dreg\_lo\ (V)} ; /* \text{arithmetic shift (b)} */ \\
\text{Dreg} = \text{ASHIFT Dreg BY Dreg\_lo\ (V,S)} ; /* \text{arithmetic shift with saturation (b)} */
\]

Arithmetic Left Shift Immediate

There is no syntax specific to a vector arithmetic left shift immediate instruction. Use the Vector Logical Shift syntax for vector left shifting, which accomplishes the same function for sign-extended numbers in number-normalizing routines. See “\gg” and “\ll” Syntax notes for caveats.

Syntax Terminology

\[
\text{Dreg: R7-0} \\
\text{Dreg\_lo: R7-0.L} \\
\text{uimm4: unsigned 4-bit field, with a range of 0 through 15}
\]
Instruction Overview

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Vector Arithmetic Shift instruction arithmetically shifts a pair of half-word registered numbers a specified distance and direction. Though the two half-word registers are shifted at the same time, the two numbers are kept separate.

Arithmetic right shifts preserve the sign of the preshifted value. The sign bit value backfills the left-most bit position vacated by the arithmetic right shift. For positive numbers, this behavior is equivalent to the logical right shift for unsigned numbers.

Only arithmetic right shifts are supported. Left shifts are performed as logical left shifts that may not preserve the sign of the original number. In the default case—without the optional saturation option—numbers can be left shifted so far that all the sign bits overflow and are lost. However, when the saturation option is enabled, a left shift that would otherwise shift nonsign bits off the left side saturates to the maximum positive or negative value instead. So, with saturation enabled, the result always keeps the same sign as the original number.

See “Saturation” on page 1-17 for a description of saturation behavior.

“>>>” and “<<” Syntax

The two half-word registers in \texttt{dest\_reg} are right shifted by the number of places specified by \texttt{shift\_magnitude}, and the result stored into \texttt{dest\_reg}. The data is always a pair of 16-bit half-registers. Valid \texttt{shift\_magnitude} values are 0 through 15.
“ASHIFT” Syntax

Both half-word registers in src_reg are shifted by the number of places prescribed in shift_magnitude, and the result stored into dest_reg.

The sign of the shift magnitude determines the direction of the shift for the ASHIFT versions.

- Positive shift magnitudes without the saturation flag (−, S) produce Logical Left shifts.
- Positive shift magnitudes with the saturation flag (−, S) produce Arithmetic Left shifts.
- Negative shift magnitudes produce Arithmetic Right shifts.

In essence, the magnitude is the power of 2 multiplied by the src_reg number. Positive magnitudes cause multiplication (N x 2^n), whereas negative magnitudes produce division (N x 2^n or N / 2^n).

The dest_reg and src_reg are both pairs of 16-bit half registers. Saturation of the result is optional.

Valid shift magnitudes for 16-bit src_reg are −16 through +15, zero included. If a number larger than these is supplied, the instruction masks and ignores the more significant bits.

This instruction does not implicitly modify the src_reg values. Optionally, dest_reg can be the same D-register as src_reg. Using the same D-register for the dest_reg and the src_reg explicitly modifies the source register.

Options

The ASHIFT instruction supports the (−, S) option, which saturates the result.
Instruction Overview

Flags Affected

This instruction affects flags as follows.

- $AZ$ is set if either result is zero; cleared if both are nonzero.
- $AN$ is set if either result is negative; cleared if both are non-negative.
- $V$ is set if either result overflows; cleared if neither overflows.
- $VS$ is set if $V$ is set; unaffected otherwise.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.
Vector Operations

Example

r4=r5>>>3 (v) ; /* arithmetic right shift immediate R5.H and R5.L by 3 bits (divide each half-word by 8) If r5 = 0x8004 000F then the result is r4 = 0xF000 0001 */
r4=r5>>>3 (v, s) ; /* same as above, but saturate the result */
r2=ashift r7 by r5.l (v) ; /* arithmetic shift (right or left, depending on sign of r5.l) R7.H and R7.L by magnitude of R5.L */
r2=ashift r7 by r5.l (v, s) ; /* same as above, but saturate the result */
r2=r5<<7 (v,s) ; /* logical left shift immediate R5.H and R5.L by 7 bits. saturated */

Also See

Vector Logical Shift, Arithmetic Shift, Logical Shift

Special Applications

None
Instruction Overview

Vector Logical Shift

General Form

\[
\begin{align*}
\text{dest}_\text{reg} &= \text{src}_\text{reg} >> \text{shift}_\text{magnitude} (V) \\
\text{dest}_\text{reg} &= \text{src}_\text{reg} \ll \text{shift}_\text{magnitude} (V) \\
\text{dest}_\text{reg} &= \text{LSHIFT src}_\text{reg} \text{ BY shift}_\text{magnitude} (V)
\end{align*}
\]

Syntax

Constant Shift Magnitude

\[
\begin{align*}
\text{Dreg} &= \text{Dreg} >> \text{uimm4} (V) ; /* \text{logical shift right, immediate (b) */} \\
\text{Dreg} &= \text{Dreg} \ll \text{uimm4} (V) ; /* \text{logical shift left, immediate (b) */}
\end{align*}
\]

Registered Shift Magnitude

\[
\text{Dreg} = \text{LSHIFT Dreg BY Dreg}_\text{lo} (V) ; /* \text{logical shift (b) */}
\]

Syntax Terminology

\[
\begin{align*}
\text{Dreg} &: R7-0 \\
\text{Dreg}_\text{lo} &: R7-0.L \\
\text{uimm4} &: \text{unsigned 4-bit field, with a range of 0 through 15}
\end{align*}
\]

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Vector Logical Shift logically shifts a pair of half-word registered numbers a specified distance and direction. Though the two half-word registers are shifted at the same time, the two numbers are kept separate.
Vector Operations

Logical shifts discard any bits shifted out of the register and backfill vacated bits with zeros.

“>>” AND “<<” Syntax

The two half-word registers in dest_reg are shifted by the number of places specified by shift_magnitude and the result stored into dest_reg. The data is always a pair of 16-bit half-registers. Valid shift_magnitude values are 0 through 15.

“LSHIFT” Syntax

Both half-word registers in src_reg are shifted by the number of places prescribed in shift_magnitude, and the result is stored into dest_reg.

For the LSHIFT versions, the sign of the shift magnitude determines the direction of the shift.

- Positive shift magnitudes produce left shifts.
- Negative shift magnitudes produce right shifts.

The dest_reg and src_reg are both pairs of 16-bit half-registers.

Valid shift magnitudes for 16-bit src_reg are –16 through +15, zero included. If a number larger than these is supplied, the instruction masks and ignores the more significant bits.

This instruction does not implicitly modify the src_reg values. Optionally, dest_reg can be the same D-register as src_reg. Using the same D-register for the dest_reg and the src_reg explicitly modifies the source register at your discretion.
Instruction Overview

Flags Affected

This instruction affects flags as follows.

- $AZ$ is set if either result is zero; cleared if both are nonzero.
- $AN$ is set if either result is negative; cleared if both are non-negative.
- $V$ is cleared.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer $ASTAT$ flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.
Vector Operations

Example

```
r4=r5>>3 (v) ;
/* logical right shift immediate R5.H and R5.L by 3 bits */
r4=r5<<3 (v) ;
/* logical left shift immediate R5.H and R5.L by 3 bits */
r2=lsr7 by r5.l (v) ;
/* logically shift (right or left, depending on sign of r5.l) R7.H and R7.L by magnitude of R5.L */
```

Also See

- Vector Arithmetic Shift
- Arithmetic Shift
- Logical Shift

Special Applications

None
Instruction Overview

Vector MAX

General Form

\[ \text{dest\_reg} = \text{MAX} \ ( \text{src\_reg\_0}, \text{src\_reg\_1} ) \ (V) \]

Syntax

\[ Dreg = \text{MAX} \ ( \ Dreg \ , \ Dreg \ ) \ (V) ; \quad /* \text{dual 16-bit operations} \]
\[ (b) */ \]

Syntax Terminology

\[ Dreg : R7-0 \]

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Vector Maximum instruction returns the maximum value (meaning the largest positive value, nearest to 0x7FFF) of the 16-bit half-word source registers to the \text{dest\_reg}.

The instruction compares the upper half-words of \text{src\_reg\_0} and \text{src\_reg\_1} and returns that maximum to the upper half-word of \text{dest\_reg}. It also compares the lower half-words of \text{src\_reg\_0} and \text{src\_reg\_1} and returns that maximum to the lower half-word of \text{dest\_reg}. The result is a concatenation of the two 16-bit maximum values.

The Vector Maximum instruction does not implicitly modify input values. The \text{dest\_reg} can be the same D-register as one of the source registers. Doing this explicitly modifies that source register.
Vector Operations

Flags Affected

This instruction affects flags as follows.

- $AZ$ is set if either or both result is zero; cleared if both are nonzero.
- $AN$ is set if either or both result is negative; cleared if both are non-negative.
- $V$ is cleared.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer $ASTAT$ flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

$r7 = \text{max}(r1, r0) (v);$

- Assume $R1 = 0x0007\ 0000$ and $R0 = 0x0000\ 000F$, then $R7 = 0x0007\ 000F$.
- Assume $R1 = 0xFFF7\ 8000$ and $R0 = 0x000A\ 7FFF$, then $R7 = 0x000A\ 7FFF$.
- Assume $R1 = 0x1234\ 5678$ and $R0 = 0x0000\ 000F$, then $R7 = 0x1234\ 5678$. 
Instruction Overview

Also See

Vector SEARCH, Vector MIN, MAX, MIN

Special Applications

None
Vector Operations

Vector MIN

General Form

\[ \text{dest} = \text{MIN} ( \text{src}_0, \text{src}_1 ) (V) \]

Syntax

\[ Dreg = \text{MIN} ( Dreg, Dreg ) (V); /* dual 16-bit operation (b) */ \]

Syntax Terminology

Dreg: R7-0

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Vector Minimum instruction returns the minimum value (the most negative value or the value closest to 0x8000) of the 16-bit half-word source registers to the dest_reg.

This instruction compares the upper half-words of src_reg_0 and src_reg_1 and returns that minimum to the upper half-word of dest_reg. It also compares the lower half-words of src_reg_0 and src_reg_1 and returns that minimum to the lower half-word of dest_reg. The result is a concatenation of the two 16-bit minimum values.

The input values are not implicitly modified by this instruction. The dest_reg can be the same D-register as one of the source registers. Doing this explicitly modifies that source register.
Instruction Overview

Flags Affected

This instruction affects flags as follows.

- \( AZ \) is set if either or both result is zero; cleared if both are nonzero.
- \( AN \) is set if either or both result is negative; cleared if both are non-negative.
- \( V \) is cleared.
- All other flags are unaffected.

\( \ast \) The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

\[ r7 = \text{min} \ (r1, r0) \ (v) ; \]

- Assume \( R1 = 0x0007 \ 0000 \) and \( R0 = 0x0000 \ 000F \), then \( R7 = 0x0000 \ 0000 \).
- Assume \( R1 = 0xFF7 \ 8000 \) and \( R0 = 0x000A \ 7FFF \), then \( R7 = 0xFF7 \ 8000 \).
- Assume \( R1 = 0x1234 \ 5678 \) and \( R0 = 0x0000 \ 000F \), then \( R7 = 0x0000 \ 000F \).
Vector Operations

Also See

Vector SEARCH, Vector MAX, MAX, MIN

Special Applications

None
Vector Multiply

Simultaneous Issue and Execution

A pair of compatible, scalar (individual) Multiply 16-Bit Operands instructions from “Multiply 16-Bit Operands” on page 15-43 can be combined into a single Vector Multiply instruction. The vector instruction executes the two scalar operations simultaneously and saves the results as a vector couplet.

See the Arithmetic Operations “Multiply 16-Bit Operands” on page 15-43 for the scalar instruction details.

Any MAC0 scalar Multiply 16-Bit Operands instruction can be combined with a compatible MAC1 scalar Multiply 16-Bit Operands instruction under the following conditions.

- Both scalar instructions must share the same mode option (for example, default, 1s, 1u, 1). Exception: the MAC1 instruction can optionally employ the mixed mode (M) that does not apply to MAC0.
- Both scalar instructions must share the same pair of source registers, but can reference different halves of those registers.
- Both scalar operations (if they are writes) must write to the same sized destination registers, either 16 or 32 bits.
- The destination registers for both scalar operations must form a vector couplet, as described below.
  - 16-bit: store results in the upper- and lower-halves of the same 32-bit Dreg. MAC0 writes to the lower half and MAC1 writes to the upper half.
  - 32-bit: store results in valid Dreg pairs. MAC0 writes to the pair’s lower (even-numbered) Dreg and MAC1 writes to the upper (odd-numbered) Dreg.
Vector Operations

Valid Dreg pairs are R7:6, R5:4, R3:2, and R1:0.

Syntax

Separate the two compatible scalar instructions with a comma to produce a vector instruction. Add a semicolon to the end of the combined instruction, as usual. The order of the MAC operations on the command line is arbitrary.

Instruction Length

This instruction is 32 bits long.

Flags Affected

This instruction affects the following flags.

- \( V \) is set if any result saturates; cleared if none saturates.
- \( VS \) is set if \( V \) is set; unaffected otherwise.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Example

```c
r2.h=r7.l*r6.h, r2.l=r7.h*r6.h ;
/* simultaneous MAC0 and MAC1 execution, 16-bit results. Both results are signed fractions. */
r4.l=r1.l*r0.l, r4.h=r1.h*r0.h ;
/* same as above. MAC order is arbitrary. */
r0.h=r3.h*r2.l (m), r0.l=r3.l*r2.l ;
```
/* MAC1 multiplies a signed fraction by an unsigned fraction. MACO multiplies two signed fractions. */
  r5.h=r3.h*r2.h (m),  r5.l=r3.l*r2.l (fu) ;
/* MAC1 multiplies signed fraction by unsigned fraction. MACO multiplies two unsigned fractions. */
  r0.h=r3.h*r2.h,  r0.l=r3.l*r2.l (is) ;
/* both MACs perform signed integer multiplication. */
  r3.h=r0.h*r1.h,  r3.l=r0.l*r1.l (s2rnd) ;
/* MAC1 and MACO multiply signed fractions. Both scale the result on the way to the destination register. */
  r0.l=r7.l*r6.l,  r0.h=r7.h*r6.h (iss2) ;
/* both MACs process signed integer operands and scale and round the result on the way to the destination half-registers. */
  r7=r2.l*r5.l,  r6=r2.h*r5.h ;
/* both operations produce 32-bit results and save in a Dreg pair. */
  r0=r4.l*r7.l,  r1=r4.h*r7.h (s2rnd) ;
/* same as above, but with signed fraction scaling mode. Order of the MAC instructions makes no difference. */
Vector Operations

Vector Multiply and Multiply-Accumulate

Simultaneous Issue and Execution

A pair of compatible, scalar (individual) instructions from

- “Multiply and Multiply-Accumulate to Accumulator” on page 15-53
- “Multiply and Multiply-Accumulate to Half-Register” on page 15-58
- “Multiply and Multiply-Accumulate to Data Register” on page 15-67

can be combined into a single vector instruction. The vector instruction executes the two scalar operations simultaneously and saves the results as a vector couplet.

See the Arithmetic Operations sections listed above for the scalar instruction details.

Any MAC0 scalar instruction from the list above can be combined with a compatible MAC1 scalar instruction under the following conditions.

- Both scalar instructions must share the same mode option (for example, default, IS, IU, T). Exception: the MAC1 instruction can optionally employ the mixed mode (M) that does not apply to MAC0.

- Both scalar instructions must share the same pair of source registers, but can reference different halves of those registers.

- If both scalar operations write to destination D-registers, they must write to the same sized destination D-registers, either 16 or 32 bits.
Instruction Overview

- The destination D-registers (if applicable) for both scalar operations must form a vector couplet, as described below.
  - 16-bit: store the results in the upper- and lower-halves of the same 32-bit Dreg. MAC0 writes to the lower half, and MAC1 writes to the upper half.
  - 32-bit: store the results in valid Dreg pairs. MAC0 writes to the pair’s lower (even-numbered) Dreg, and MAC1 writes to the upper (odd-numbered) Dreg.

Valid Dreg pairs are R7:6, R5:4, R3:2, and R1:0.

Syntax

Separate the two compatible scalar instructions with a comma to produce a vector instruction. Add a semicolon to the end of the combined instruction, as usual. The order of the MAC operations on the command line is arbitrary.

Instruction Length

This instruction is 32 bits long.

Flags Affected

The flags reflect the results of the two scalar operations. This instruction affects flags as follows.

- V is set if any result extracted to a Dreg saturates; cleared if no Dregs saturate.
- VS is set if V is set; unaffected otherwise.
- AV0 is set if result in Accumulator A0 (MAC0 operation) saturates; cleared if A0 result does not saturate.
- AV0S is set if AV0 is set; unaffected otherwise.
Vector Operations

- $AV1$ is set if result in Accumulator $A1$ (MAC1 operation) saturates; cleared if $A1$ result does not saturate.
- $AV1S$ is set if $AV1$ is set; unaffected otherwise.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Example

Result is 40-bit Accumulator

```c
a1=r2.1*r3.h, a0=r2.h*r3.h; /* both multiply signed fractions into separate Accumulators */
a0=r1.1*r0.1, a1+=r1.h*r0.h; /* same as above, but sum result into A1. MAC order is arbitrary. */
a1+=r3.h*r3.1, a0-=r3.h*r3.h; /* sum product into A1, subtract product from A0 */
a1=r3.h*r2.1 (m), a0+=r3.1*r2.1; /* MAC1 multiplies a signed fraction in r3.h by an unsigned fraction in r2.1. MAC0 multiplies two signed fractions. */
a1=r7.h*r4.h (m), a0+=r7.1*r4.1 (fu); /* MAC1 multiplies signed fraction by unsigned fraction. MAC0 multiplies and accumulates two unsigned fractions. */
a1+=r3.h*r2.1, a0=r3.1*r2.1 (is); /* both MACs perform signed integer multiplication */
a1=r6.h*r7.h, a0+=r6.1*r7.1 (w32); /* both MACs multiply signed fractions, sign extended, and saturate both Accumulators at bit 31 */
```
Instruction Overview

Result is 16-bit half D-register

\[ r2.h = (a1 = r7.1 * r6.h), \quad r2.l = (a0 = r7.h * r6.h) ; \quad /* \text{simultaneous MAC0} \]
\[ \text{and MAC1 execution, both are signed fractions, both products load into the Accumulators, MAC1 into half-word registers.} */ \]
\[ r4.l = (a0 = r1.1 * r0.1), \quad r4.h = (a1 = r1.h * r0.h) ; \quad /* \text{same as above, but sum result into A1. MAC order is arbitrary.} */ \]
\[ r7.h = (a1 = r6.h * r5.1), \quad r7.l = (a0 = r6.h * r5.h) ; \quad /* \text{sum into A1, subtract into A0} */ \]
\[ r0.h = (a1 = r7.h * r4.1) \text{ (m)}, \quad r0.l = (a0 = r7.1 * r4.1) ; \quad /* MAC1 multiplies a signed fraction by an unsigned fraction. MAC0 multiplies two signed fractions. */ \]
\[ r5.h = (a1 = r3.h * r2.1) \text{ (m)}, \quad r5.l = (a0 = r3.1 * r2.1) \text{ (fu)} ; \quad /* MAC1 multiplies signed fraction by unsigned fraction. MAC0 multiplies two unsigned fractions. */ \]
\[ r0.h = (a1 = r3.h * r2.h), \quad r0.l = (a0 = r3.1 * r2.1) \text{ (is)} ; \quad /* \text{both MACs perform signed integer multiplication.} */ \]
\[ r5.h = (a1 = r3.h * r1.h), \quad a0 = r2.1 * r1.1 ; \quad /* \text{both MACs multiply signed fractions. MAC0 does not copy the accum result.} */ \]
\[ r3.h = (a1 = r2.h * r1.h) \text{ (m)}, \quad a0 = r2.1 * r1.1 ; \quad /* MAC1 multiplies signed fraction by unsigned fraction and uses all 40 bits of A1. MAC0 multiplies two signed fractions. */ \]
\[ r3.h = a1, \quad r3.1 = (a0 = r0.1 * r1.1) \text{ (s2rnd)} ; \quad /* MAC1 copies Accumulator to register half. MAC0 multiplies signed fractions. Both scale the result and round on the way to the destination register. */ \]
\[ r0.l = (a0 = r7.1 * r6.1), \quad r0.h = (a1 = r7.h * r6.h) \text{ (iss2)} ; \quad /* \text{both MACs process signed integer the way to the destination half-registers.} */ \]

19-44 ADSP-BF53x/BF56x Blackfin Processor Programming Reference
Vector Operations

Result is 32-bit D-register

\[ r_3 = (a_1 = r_6.h \times r_7.h), \quad r_2 = (a_0 = r_6.1 \times r_7.1) \quad / \text{simultaneous MAC0 and MAC1 execution, both are signed fractions, both products load into the Accumulators} */ \]
\[ r_4 = (a_0 = r_6.1 \times r_7.1), \quad r_5 = (a_1 = r_6.h \times r_7.h) \quad / \text{same as above, but sum result into A1. MAC order is arbitrary} */ \]
\[ r_7 = (a_1 = r_3.h \times r_5.h), \quad r_6 = (a_0 = r_3.1 \times r_5.1) \quad / \text{sum into A1, subtract into A0} */ \]
\[ r_1 = (a_1 = r_7.1 \times r_4.1) \quad (m), \quad r_0 = (a_0 = r_7.h \times r_4.h) \quad / \text{MAC1 multiplies a signed fraction by an unsigned fraction. MAC0 multiplies two signed fractions} */ \]
\[ r_5 = (a_1 = r_3.h \times r_7.h) \quad (m), \quad r_4 = (a_0 = r_3.1 \times r_7.1) \quad (fu) \quad / \text{MAC1 multiplies signed fraction by unsigned fraction. MAC0 multiplies two unsigned fractions} */ \]
\[ r_1 = (a_1 = r_3.h \times r_2.h), \quad r_0 = (a_0 = r_3.1 \times r_2.1) \quad (is) \quad / \text{both MACs perform signed integer multiplication} */ \]
\[ r_5 = (a_1 = r_6.h \times r_7.h), \quad a_0 = r_6.1 \times r_7.1 \quad / \text{both MACs multiply signed fractions. MAC0 does not copy the accum result} */ \]
\[ r_3 = (a_1 = r_6.h \times r_7.h) \quad (m), \quad a_0 = r_6.1 \times r_7.1 \quad / \text{MAC1 multiplies signed fraction by unsigned fraction and uses all 40 bits of A1. MAC0 multiplies two signed fractions} */ \]
\[ r_3 = a_1, \quad r_2 = (a_0 = r_0.1 \times r_1.1) \quad (s2rnd) \quad / \text{MAC1 moves Accumulator to register. MAC0 multiplies signed fractions. Both scale the result and round on the way to the destination register} */ \]
\[ r_0 = (a_0 = r_7.1 \times r_6.1), \quad r_1 = (a_1 = r_7.h \times r_6.h) \quad (iss2) \quad / \text{both MACs process signed integer operands and scale the result on the way to the destination registers} */ \]
Instruction Overview

Vector Negate (Two's-Complement)

General Form

\[
\text{dest\_reg} = - \text{source\_reg (V)}
\]

Syntax

\[
\text{Dreg} = - \text{Dreg (V)} ; /* dual 16-bit operation (b) */
\]

Syntax Terminology

\[
\text{Dreg: R7-0}
\]

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Vector Negate instruction returns the same magnitude with the opposite arithmetic sign, saturated for each 16-bit half-word in the source. The instruction calculates by subtracting the source from zero.

See “Saturation” on page 1-17 for a description of saturation behavior.

Flags Affected

This instruction affects flags as follows.

- \( AZ \) is set if either or both results are zero; cleared if both are nonzero.
- \( AN \) is set if either or both results are negative; cleared if both are non-negative.
- \( V \) is set if either or both results saturate; cleared if neither saturates.

19-46 ADSP-BF53x/BF56x Blackfin Processor Programming Reference
Vector Operations

- $V_S$ is set if $V$ is set; unaffected otherwise.
- $AC_0$ is set if carry occurs from either or both results; cleared if neither produces a carry.
- All other flags are unaffected.

The ADSP-BF535 processor has fewer $A$STAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

```plaintext
r5 = -r3 (v) ; /* R5.H becomes the negative of R3.H and R5.L becomes the negative of R3.L If r3 = 0x0004 7FFF the result is r5 = 0xFFFF 8001 */
```

Also See

Negate (Two’s-Complement)

Special Applications

None
Instruction Overview

Vector PACK

General Form

\[
\text{Dest\_reg} = \text{PACK} ( \text{src\_half\_0}, \text{src\_half\_1} )
\]

Syntax

\[
\text{Dreg} = \text{PACK} ( \text{Dreg\_lo\_hi}, \text{Dreg\_lo\_hi} ) ; \quad /* (b) */
\]

Syntax Terminology

\[
\text{Dreg}: \text{R7-0}
\]

\[
\text{Dreg\_lo\_hi}: \text{R7-0.L}, \text{R7-0.H}
\]

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

The Vector Pack instruction packs two 16-bit half-word numbers into the halves of a 32-bit data register as shown in Table 19-18 and Table 19-19.

Table 19-18. Source Registers Contain

<table>
<thead>
<tr>
<th>src_half_0</th>
<th>15............8 7..............0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>half_word_0</td>
</tr>
<tr>
<td>src_half_1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>half_word_1</td>
</tr>
</tbody>
</table>

Table 19-19. Destination Register Contains

<table>
<thead>
<tr>
<th>dest_reg:</th>
<th>31............24 23............16 15............8 7..............0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>half_word_0</td>
</tr>
<tr>
<td></td>
<td>half_word_1</td>
</tr>
</tbody>
</table>
Vector Operations

Flags Affected

None

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with specific other 16-bit instructions. For details, see “Issuing Parallel Instructions” on page 20-1.

Example

\[
\begin{align*}
r_3 &= \text{pack}(r_4.l, r_5.l) ; \\
r_1 &= \text{pack}(r_6.l, r_4.h) ; \\
r_0 &= \text{pack}(r_2.h, r_4.l) ; \\
r_5 &= \text{pack}(r_7.h, r_2.h) ;
\end{align*}
\]

Also See

BYTEPACK (Quad 8-Bit Pack)

Special Applications

\[
\begin{align*}
&\text{/* If } r_4.l = \text{0xDEAD} \text{ and } r_5.l = \text{0xBEEF}, \text{ then } \\
&\quad \text{r3 = pack}(r_4.l, r_5.l) ; \\
&\text{/* . . . produces r3 = 0xDEAD BEEF */}
\end{align*}
\]
Instruction Overview

Vector SEARCH

General Form

\[(\text{dest\_pointer\_hi, dest\_pointer\_lo}) = \text{SEARCH \text{src\_reg (searchmode)}}\]

Syntax

\[\text{(Dreg, Dreg)} = \text{SEARCH Dreg (searchmode)} ; \quad /* (b) */\]

Syntax Terminology

\[\text{Dreg: R7-0}
\]

\[\text{searchmode: (GT), (GE), (LE), or (LT)}\]

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

Functional Description

This instruction is used in a loop to locate a maximum or minimum element in an array of 16-bit packed data. Two values are tested at a time.

The Vector Search instruction compares two 16-bit, signed half-words to values stored in the Accumulators. Then, it conditionally updates each Accumulator and destination pointer based on the comparison.

Pointer register \(P0\) is always the implied array pointer for the elements being searched.

More specifically, the signed high half-word of \(\text{src\_reg}\) is compared in magnitude with the 16 low-order bits in \(A1\). If \(\text{src\_reg\_hi}\) meets the comparison criterion, then \(A1\) is updated with \(\text{src\_reg\_hi}\), and the value in pointer register \(P0\) is stored in \(\text{dest\_pointer\_hi}\). The same operation is performed for \(\text{src\_reg\_low and A0}\).
Vector Operations

Based on the search mode specified in the syntax, the instruction tests for maximum or minimum signed values.

Values are sign extended when copied into the Accumulator(s).

See “Example” for one way to implement the search loop. After the vector search loop concludes, A1 and A0 hold the two surviving elements, and dest_pointer_hi and dest_pointer_lo contain their respective addresses. The next step is to select the final value from these two surviving elements.

Modes

The four supported compare modes are specified by the mandatory searchmode flag.

Table 19-20. Compare Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(GT)</td>
<td>Greater than. Find the location of the first maximum number in an array.</td>
</tr>
<tr>
<td>(GE)</td>
<td>Greater than or equal. Find the location of the last maximum number in an array.</td>
</tr>
<tr>
<td>(LT)</td>
<td>Less than. Find the location of the first minimum number in an array.</td>
</tr>
<tr>
<td>(LE)</td>
<td>Less than or equal. Find the location of the last minimum number in an array.</td>
</tr>
</tbody>
</table>

Summary

Assumed Pointer P0

src_reg_hi Compared to least significant 16 bits of A1. If compare condition is met, overwrites lower 16 bits of A1 and copies P0 into dest_pointer_hi.

src_reg_lo Compared to least significant 16 bits of A0. If compare condition is met, overwrites lower 16 bits of A0 and copies P0 into dest_pointer_lo.
Instruction Overview

Flags Affected

None

The ADSP-BF535 processor has fewer ASTAT flags and some flags operate differently than subsequent Blackfin family products. For more information on the ADSP-BF535 status flags, see Table A-1 on page A-3.

Required Mode

User & Supervisor

Parallel Issue

This instruction can be issued in parallel with the combination of one 16-bit length load instruction to the P0 register and one 16-bit NOP. No other instructions can be issued in parallel with the Vector Search instruction. Note the following legal and illegal forms.

(r1, r0) = search r2 (LT) || r2 = [p0++p3]; /* ILLEGAL */

(r1, r0) = search r2 (LT) || r2 = [p0++]; /* LEGAL */

(r1, r0) = search r2 (LT) || r2 = [p0+]; /* LEGAL */

Example

/* Initialize Accumulators with appropriate value for the type of search. */
   r0.l=0x7fff ;
   r0.h=0 ;
   a0=r0 ;  /* max positive 16-bit value */
   a1=r0 ;  /* max positive 16-bit value */
/* Initialize R2. */
   r2=[p0++] ;
/* Assume P1 is initialized to the size of the vector length. */
Vector Operations

LSETUP (loop_, loop_) LC0=P1>>1 ; /* set up the loop */
loop_: (r1,r0) = SEARCH R2 (LE) || R2=[P0++];
    /* search for the last minimum in all but the
    last element of the array */
    (r1,r0) = SEARCH R2 (LE);
    /* finally, search the last element */
    /* The lower 16 bits of A1 and A0 contain the last minimums of the
array. R1 contains the value of P0 corresponding to the value in
A1. R0 contains the value of P0 corresponding to the value in A0.
Next, compare A1 and A0 together and R1 and R0 together to find
the single, last minimum in the array.
Note: In this example, the resulting pointers are past the actual
surviving array element due to the post-increment operation. */
c = a0 <= a1 ;
r0 += -4 ;
r1 += -2 ;
if !cc r0 = r1 ; /* the pointer to the survivor is in r0 */

Also See

Vector MAX, Vector MIN, MAX, MIN

Special Applications

This instruction is used in a loop to locate an element in a vector accord-
ing to the element’s value.
This chapter discusses the instructions that can be issued in parallel. It identifies supported combinations for parallel issue, parallel issue syntax, 32-bit ALU/MAC instructions, 16-bit instructions, and examples.

The Blackfin processor is not superscalar; it does not execute multiple instructions at once. However, it does permit up to three instructions to be issued in parallel with some limitations. A multi-issue instruction is 64-bits in length and consists of one 32-bit instruction and two 16-bit instructions. All three instructions execute in the same amount of time as the slowest of the three.

Sections in this chapter

- “Supported Parallel Combinations” on page 20-1
- “Parallel Issue Syntax” on page 20-2
- “32-Bit ALU/MAC Instructions” on page 20-3
- “16-Bit Instructions” on page 20-6
- “Examples” on page 20-8

Supported Parallel Combinations

The diagram in Table 20-1 illustrates the combinations for parallel issue that the Blackfin processor supports.
Parallel Issue Syntax

The syntax of a parallel issue instruction is as follows.

- **A 32-bit ALU/MAC instruction || A 16-bit instruction || A 16-bit instruction ;**

  The vertical bar (||) indicates the following instruction is to be issued in parallel with the previous instruction. Note the terminating semicolon appears only at the end of the parallel issue instruction.

  It is possible to issue a 32-bit ALU/MAC instruction in parallel with only one 16-bit instruction using the following syntax. The result is still a 64-bit instruction with a 16-bit NOP automatically inserted into the unused 16-bit slot.

- **A 32-bit ALU/MAC instruction || A 16-bit instruction ;**

  Alternately, it is also possible to issue two 16-bit instructions in parallel with one another without an active 32-bit ALU/MAC instruction by using the MNOP instruction, shown below. Again, the result is still a 64-bit instruction.

- **MNOP || A 16-bit instruction || A 16-bit instruction ;**

  See the MNOP (32-bit NOP) instruction description in “No Op” on page 16-25. The MNOP instruction does not have to be explicitly included by the programmer; the software tools prepend it automatically. The MNOP instruction will appear in disassembled parallel 16-bit instructions.

Table 20-1. Parallel Issue Combinations

<table>
<thead>
<tr>
<th>32-bit ALU/MAC instruction</th>
<th>16-bit Instruction</th>
<th>16-bit Instruction</th>
</tr>
</thead>
</table>
32-Bit ALU/MAC Instructions

The list of 32-bit instructions that can be in a parallel instruction are shown in Table 20-2.

Table 20-2. 32-Bit DSP Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic Operations</strong></td>
<td></td>
</tr>
<tr>
<td>ABS (Absolute Value)</td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>Only the versions that support optional saturation.</td>
</tr>
<tr>
<td>Add/Subtract – Prescale Up</td>
<td></td>
</tr>
<tr>
<td>Add/Subtract – Prescale Down</td>
<td></td>
</tr>
<tr>
<td>EXPADJ (Exponent Detection)</td>
<td></td>
</tr>
<tr>
<td>MAX (Maximum)</td>
<td></td>
</tr>
<tr>
<td>MIN (Minimum)</td>
<td></td>
</tr>
<tr>
<td>Modify – Decrement (for Accumulators, only)</td>
<td></td>
</tr>
<tr>
<td>Modify – Increment (for Accumulators, only)</td>
<td>Accumulator versions only.</td>
</tr>
<tr>
<td>Negate (Two’s-Complement)</td>
<td>Accumulator versions only.</td>
</tr>
<tr>
<td>RND (Round to Half-Word)</td>
<td></td>
</tr>
<tr>
<td>Saturate</td>
<td></td>
</tr>
<tr>
<td>SIGNBITS</td>
<td></td>
</tr>
<tr>
<td>Subtract</td>
<td>Saturating versions only.</td>
</tr>
<tr>
<td><strong>Load Store</strong></td>
<td></td>
</tr>
<tr>
<td>Load Immediate</td>
<td>Accumulator versions only.</td>
</tr>
</tbody>
</table>
### 32-Bit ALU/MAC Instructions

Table 20-2. 32-Bit DSP Instructions  (Cont’d)

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bit Operations</strong></td>
<td></td>
</tr>
<tr>
<td>DEPOSIT (Bit Field Deposit)</td>
<td></td>
</tr>
<tr>
<td>EXTRACT (Bit Field Extract)</td>
<td></td>
</tr>
<tr>
<td>BITMUX (Bit Multiplex)</td>
<td></td>
</tr>
<tr>
<td>ONES (One’s-Population Count)</td>
<td></td>
</tr>
<tr>
<td><strong>Logical Operations</strong></td>
<td></td>
</tr>
<tr>
<td>^ (Exclusive-OR) (Bit-Wise XOR)</td>
<td></td>
</tr>
<tr>
<td><strong>Move</strong></td>
<td></td>
</tr>
<tr>
<td>Move Register</td>
<td>40-bit Accumulator versions only.</td>
</tr>
<tr>
<td>Move Register Half</td>
<td></td>
</tr>
<tr>
<td><strong>Shift / Rotate Operations</strong></td>
<td></td>
</tr>
<tr>
<td>Arithmetic Shift</td>
<td>Saturating and Accumulator versions only.</td>
</tr>
<tr>
<td>Logical Shift</td>
<td>32-bit instruction size versions only.</td>
</tr>
<tr>
<td>ROT (Rotate)</td>
<td></td>
</tr>
<tr>
<td>External Event Management</td>
<td></td>
</tr>
<tr>
<td>No Op</td>
<td>32-bit MNOP only</td>
</tr>
</tbody>
</table>
### Issuing Parallel Instructions

Table 20-2. 32-Bit DSP Instructions (Cont’d)

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vector Operations</strong></td>
<td></td>
</tr>
<tr>
<td>VIT_MAX (Compare-Select)</td>
<td></td>
</tr>
<tr>
<td>Add on Sign</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half-Register</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td></td>
</tr>
<tr>
<td>Vector ABS (Vector Absolute Value)</td>
<td></td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td></td>
</tr>
<tr>
<td>Vector Arithmetic Shift</td>
<td></td>
</tr>
<tr>
<td>Vector Logical Shift</td>
<td></td>
</tr>
<tr>
<td>Vector MAX (Vector Maximum)</td>
<td></td>
</tr>
<tr>
<td>Vector MIN (Vector Minimum)</td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td></td>
</tr>
<tr>
<td>Vector Negate (Two’s-Complement)</td>
<td></td>
</tr>
<tr>
<td>Vector PACK</td>
<td></td>
</tr>
<tr>
<td>Vector SEARCH</td>
<td></td>
</tr>
</tbody>
</table>
16-Bit Instructions

The two 16-bit instructions in a multi-issue instruction must each be from Group 1 and Group 2 instructions shown in Table 20-3 and Table 20-4.

The following additional restrictions also apply to the 16-bit instructions of the multi-issue instruction.

- Only one of the 16-bit instructions can be a store instruction.
- If the two 16-bit instructions are memory access instructions, then both cannot use P-registers as address registers. In this case, at least one memory access instruction must be an I-register version.

Table 20-2. 32-Bit DSP Instructions (Cont’d)

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Video Pixel Operations</strong></td>
<td></td>
</tr>
<tr>
<td>ALIGN8, ALIGN16, ALIGN24 (Byte Align)</td>
<td></td>
</tr>
<tr>
<td>DISALGNEXCPT (Disable Alignment Exception for Load)</td>
<td></td>
</tr>
<tr>
<td>SAA (Quad 8-Bit Subtract-Absolute-Accumulate)</td>
<td></td>
</tr>
<tr>
<td>Dual 16-Bit Accumulator Extraction with Addition</td>
<td></td>
</tr>
<tr>
<td>BYTEOP16P (Quad 8-Bit Add)</td>
<td></td>
</tr>
<tr>
<td>BYTEOP16M (Quad 8-Bit Subtract)</td>
<td></td>
</tr>
<tr>
<td>BYTEOP1P (Quad 8-Bit Average – Byte)</td>
<td></td>
</tr>
<tr>
<td>BYTEOP2P (Quad 8-Bit Average – Half-Word)</td>
<td></td>
</tr>
<tr>
<td>BYTEOP3P (Dual 16-Bit Add / Clip)</td>
<td></td>
</tr>
<tr>
<td>BYTEPACK (Quad 8-Bit Pack)</td>
<td></td>
</tr>
<tr>
<td>BYTEUNPACK (Quad 8-Bit Unpack)</td>
<td></td>
</tr>
</tbody>
</table>
### Issuing Parallel Instructions

Table 20-3. Group1 Compatible 16-Bit Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic Operations</strong></td>
<td></td>
</tr>
<tr>
<td>Add Immediate</td>
<td>Ireg versions only.</td>
</tr>
<tr>
<td>Modify – Decrement</td>
<td>Ireg versions only.</td>
</tr>
<tr>
<td>Modify – Increment</td>
<td>Ireg versions only.</td>
</tr>
<tr>
<td>Subtract Immediate</td>
<td>Ireg versions only.</td>
</tr>
<tr>
<td><strong>Load / Store</strong></td>
<td></td>
</tr>
<tr>
<td>Load Pointer Register</td>
<td></td>
</tr>
<tr>
<td>Load Data Register</td>
<td></td>
</tr>
<tr>
<td>Load Half-Word – Zero-Extended</td>
<td></td>
</tr>
<tr>
<td>Load Half-Word – Sign-Extended</td>
<td></td>
</tr>
<tr>
<td>Load High Data Register Half</td>
<td></td>
</tr>
<tr>
<td>Load Low Data Register Half</td>
<td></td>
</tr>
<tr>
<td>Load Byte – Zero-Extended</td>
<td></td>
</tr>
<tr>
<td>Load Byte – Sign-Extended</td>
<td></td>
</tr>
<tr>
<td>Store Pointer Register</td>
<td></td>
</tr>
<tr>
<td>Store Data Register</td>
<td></td>
</tr>
<tr>
<td>Store High Data Register Half</td>
<td></td>
</tr>
<tr>
<td>Store Low Data Register Half</td>
<td></td>
</tr>
<tr>
<td>Store Byte</td>
<td></td>
</tr>
</tbody>
</table>
Examples

Two Parallel Memory Access Instructions

/* Subtract-Absolute-Accumulate issued in parallel with the memory access instructions that fetch the data for the next SAA instruction. This sequence is executed in a loop to flip-flop back and forth between the data in R1 and R3, then the data in R0 and R2. */
saa (r1:0, r3:2) || r0=[i0++] || r2=[i1++] ;
saa (r1:0, r3:2)(r) || r1=[i0++] || r3=[i1++] ;
mnop || r1 = [i0++] || r3 = [i1++] ;

Table 20-4. Group2 Compatible 16-Bit Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Load / Store</strong></td>
<td></td>
</tr>
<tr>
<td>Load Data Register</td>
<td>Ireg versions only.</td>
</tr>
<tr>
<td>Load High Data Register Half</td>
<td>Ireg versions only.</td>
</tr>
<tr>
<td>Load Low Data Register Half</td>
<td>Ireg versions only.</td>
</tr>
<tr>
<td>Store Data Register</td>
<td>Ireg versions only.</td>
</tr>
<tr>
<td>Store High Data Register Half</td>
<td>Ireg versions only.</td>
</tr>
<tr>
<td>Store Low Data Register Half</td>
<td>Ireg versions only.</td>
</tr>
<tr>
<td><strong>External Event Management</strong></td>
<td></td>
</tr>
<tr>
<td>No Op</td>
<td>16-bit NOP only.</td>
</tr>
</tbody>
</table>
Issuing Parallel Instructions

One Ireg and One Memory Access Instruction in Parallel

/* Add on Sign while incrementing an Ireg and loading a data register based on the previous value of the Ireg. */
r7.h=r7.l=sign(r2.h)*r3.h + sign(r2.l)*r3.l || i0+=m3 || r0=[i0] ;
/* Add/subtract two vector values while incrementing an Ireg and loading a data register. */
R2 = R2 +|+ R4, R4 = R2 |-| R4 (ASR) || I0 += M0 (BREV) || R1 = [I0] ;
/* Multiply and accumulate to Accumulator while loading a data register and storing a data register using an Ireg pointer. */
/* Multiply and accumulate while loading two data registers. One load uses an Ireg pointer. */
A1+=R0.L*R2.H,A0+=R0.L*R2.L || R2.L=W[I2++] || R0=[I1--] ;
/* Pack two vector values while storing a data register using an Ireg pointer and loading another data register. */
R1=PACK(R1.H,R0.H) || [I0++]\=R0 || R2.L=W[I2++] ;

One Ireg Instruction in Parallel

/* Multiply-Accumulate to a Data register while incrementing an Ireg. */
r6=(a0+=r3.h*r2.h)(fu) || i2\=-m0 ;
/* which the assembler expands into: */
r6=(a0+=r3.h*r2.h)(fu) || i2\=-m0 || nop ; */
Examples
21 DEBUG

The Blackfin processor’s debug functionality is used for software debugging. It also complements some services often found in an operating system (OS) kernel. The functionality is implemented in the processor hardware and is grouped into multiple levels.

A summary of available debug features is shown in Table 21-1.

Table 21-1. Blackfin Debug Features

<table>
<thead>
<tr>
<th>Debug Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Watchpoints</td>
<td>Specify address ranges and conditions that halt the processor when satisfied.</td>
</tr>
<tr>
<td>Trace History</td>
<td>Stores the last 16 discontinuous values of the Program Counter in an on-chip trace buffer.</td>
</tr>
<tr>
<td>Cycle Count</td>
<td>Provides functionality for all code profiling functions.</td>
</tr>
<tr>
<td>Performance Monitoring</td>
<td>Allows internal resources to be monitored and measured non-intrusively.</td>
</tr>
</tbody>
</table>

Watchpoint Unit

By monitoring the addresses on both the instruction bus and the data bus, the Watchpoint Unit provides several mechanisms for examining program behavior. After counting the number of times a particular address is matched, the unit schedules an event based on this count.
Watchpoint Unit

In addition, information that the Watchpoint Unit provides helps in the optimization of code. The unit also makes it easier to maintain executables through code patching.

The Watchpoint Unit contains these memory-mapped registers (MMRs), which are accessible in Supervisor and Emulator modes:

- The Watchpoint Status register (WPSTAT)
- Six Instruction Watchpoint Address registers (WPIA[5:0])
- Six Instruction Watchpoint Address Count registers (WPIACNT[5:0])
- The Instruction Watchpoint Address Control register (WPIACTL)
- Two Data Watchpoint Address registers (WPDA[1:0])
- Two Data Watchpoint Address Count registers (WPDACNT[1:0])
- The Data Watchpoint Address Control register (WPDACTL)

Two operations implement instruction watchpoints:

- The values in the six Instruction Watchpoint Address registers, WPIA[5:0], are compared to the address on the instruction bus.

- Corresponding count values in the Instruction Watchpoint Address Count registers, WPIACNT[5:0], are decremented on each match.

The six Instruction Watchpoint Address registers may be further grouped into three ranges of instruction-address-range watchpoints. The ranges are identified by the addresses in WPIA0 to WPIA1, WPIA2 to WPIA3, and WPIA4 to WPIA5.
The address ranges stored in WPIA0, WPIA1, WPIA2, WPIA3, WPIA4, and WPIA5 must satisfy these conditions:

\[
WPIA0 \leq WPIA1 \\
WPIA2 \leq WPIA3 \\
WPIA4 \leq WPIA5
\]

Two operations implement data watchpoints:

- The values in the two Data Watchpoint Address registers, WPDA[1:0], are compared to the address on the data buses.
- Corresponding count values in the Data Watchpoint Address Count registers, WPDACNT[1:0], are decremented on each match.

The two Data Watchpoint Address registers may be further grouped together into one data-address-range watchpoint, WPDA[1:0].

The instruction and data count value registers must be loaded with the number of times the watchpoint must match minus one. After the count value reaches zero, the subsequent watchpoint match results in an exception or emulation event.

Note count values must be reinitialized after the event has occurred.

An event can also be triggered on a combination of the instruction and data watchpoints. If the WPAND bit in the WPIACTL register is set, then an event is triggered only when both an instruction address watchpoint matches and a data address watchpoint matches. If the WPAND bit is 0, then an event is triggered when any of the enabled watchpoints or watchpoint ranges match.
Watchpoint Unit

To enable the Watchpoint Unit, the WPPWR bit in the WPIACTL register must be set. If WPPWR = 1, then the individual watchpoints and watchpoint ranges may be enabled using the specific enable bits in the WPIACTL and WPDACTL MMRs. If WPPWR = 0, then all watchpoint activity is disabled.

Instruction Watchpoints

Each instruction watchpoint is controlled by three bits in the WPIACTL register, as shown in Table 21-2.

Table 21-2. WPIACTL Control Bits

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMUSWx</td>
<td>Determines whether an instruction-address match causes either an emulation event or an exception event.</td>
</tr>
<tr>
<td>WPICNTENx</td>
<td>Enables the 16-bit counter that counts the number of address matches. If the counter is disabled, then every match causes an event.</td>
</tr>
<tr>
<td>WPIAENx</td>
<td>Enables the address watchpoint activity.</td>
</tr>
</tbody>
</table>

When two watchpoints are associated to form a range, two additional bits are used, as shown in Table 21-3.

Table 21-3. WPIACTL Watchpoint Range Control Bits

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WPIRENxy</td>
<td>Indicates the two watchpoints that are to be associated to form a range.</td>
</tr>
<tr>
<td>WPIRINVxy</td>
<td>Determines whether an event is caused by an address within the range identified or outside of the range identified.</td>
</tr>
</tbody>
</table>
Debug

Code patching allows software to replace sections of existing code with new code. The watchpoint registers are used to trigger an exception at the start addresses of the earlier code. The exception routine then vectors to the location in memory that contains the new code.

On the processor, code patching can be achieved by writing the start address of the earlier code to one of the WPIAn registers and setting the corresponding EMUSWx bit to trigger an exception. In the exception service routine, the WPSTAT register is read to determine which watchpoint triggered the exception. Next, the code writes the start address of the new code in the RETX register, and then returns from the exception to the new code. Because the exception mechanism is used for code patching, event service routines of the same or higher priority (exception, NMI, and reset routines) cannot be patched.

A write to the WPSTAT MMR clears all the sticky status bits. The data value written is ignored.

WPIAn Registers

When the Watchpoint Unit is enabled, the values in the Instruction Watchpoint Address registers (WPIAn) are compared to the address on the instruction bus. Corresponding count values in the Instruction Watchpoint Address Count registers (WPIACNTn) are decremented on each match.

Figure 21-1 shows the Instruction Watchpoint Address registers, WPIA[5:0].
Watchpoint Unit

Instruction Watchpoint Address Registers (WPIAn)

For Memory-mapped addresses, see Table 21-4.

WPIA (Instruction Address)[30:15]

WPIA (Instruction Address)[14:0]

Figure 21-1. Instruction Watchpoint Address Registers

Table 21-4. Instruction Watchpoint Register Memory-mapped Addresses

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>WPIA0</td>
<td>0xFFE0 7040</td>
</tr>
<tr>
<td>WPIA1</td>
<td>0xFFE0 7044</td>
</tr>
<tr>
<td>WPIA2</td>
<td>0xFFE0 7048</td>
</tr>
<tr>
<td>WPIA3</td>
<td>0xFFE0 704C</td>
</tr>
<tr>
<td>WPIA4</td>
<td>0xFFE0 7050</td>
</tr>
<tr>
<td>WPIA5</td>
<td>0xFFE0 7054</td>
</tr>
</tbody>
</table>

WPIACNTn Registers

When the Watchpoint Unit is enabled, the count values in the Instruction Watchpoint Address Count registers (WPIACNT[5:0]) are decremented each time the address or the address bus matches a value in the WPIAn registers. Load the WPIACNTn register with a value that is one less than the number of times the watchpoint must match before triggering an event (see Figure 21-2). The WPIACNTn register will decrement to 0x0000 when the programmed count expires.
Instruction Watchpoint Address Count Registers (WPIACNTn)

For Memory-mapped addresses, see Table 21-5.

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>WPIACNT0</td>
<td>0xFFF0 7080</td>
</tr>
<tr>
<td>WPIACNT1</td>
<td>0xFFF0 7084</td>
</tr>
<tr>
<td>WPIACNT2</td>
<td>0xFFF0 7088</td>
</tr>
<tr>
<td>WPIACNT3</td>
<td>0xFFF0 708C</td>
</tr>
<tr>
<td>WPIACNT4</td>
<td>0xFFF0 7090</td>
</tr>
<tr>
<td>WPIACNT5</td>
<td>0xFFF0 7094</td>
</tr>
</tbody>
</table>

WPIACNT (Count Value)[15:0]

Figure 21-2. Instruction Watchpoint Address Count Registers

Table 21-5. Instruction Watchpoint Address Count Register Memory-mapped Addresses

WPIACTL Register

Three bits in the Instruction Watchpoint Address Control register (WPIACTL) control each instruction watchpoint. Figure 21-3 describes the upper half of the register. Figure 21-4 on page 21-9 describes the lower half of the register. For more information about the bits in this register, see “Instruction Watchpoints” on page 21-4.

The bits in the WPIACTL register have no effect unless the WPPWR bit is set.

The bits in the WPIACTL register have no effect unless the WPPWR bit is set.
Watchpoint Unit

Instruction Watchpoint Address Control Register (WPIACTL)
In range comparisons, IA = instruction address

![Diagram of Instruction Watchpoint Address Control Register (WPIACTL)]

- **RPAND**
  - 0 - Any enabled watchpoint triggers an exception or emulation event
  - 1 - Any enabled instruction address watchpoint AND any enabled data address watchpoint trigger an exception or emulation event

- **EMUSW3**
  - 0 - Match on WPIA3 causes an exception event
  - 1 - Match on WPIA3 causes an emulation event

- **WPIREN45**
  - 0 - Disable range comparison
  - 1 - Enable range comparison:
    - (Start address = WPIA4, End address = WPIA5)

- **WPIRINV45**
  - Valid when WPIREN45 = 1
  - 0 - Inclusive range comparison:
    - WPIA4 < IA <= WPIA5
  - 1 - Exclusive range comparison:
    - IA <= WPIA4 || IA > WPIA5

- **WPIAEN4**
  - 0 - Disable instruction address watchpoint, WPIA4
  - 1 - Enable instruction address watchpoint, WPIA4

- **WPICNTE4**
  - Valid when WPIREN45 = 0
  - 0 - Disable watchpoint instruction address counter 5
  - 1 - Enable watchpoint instruction address counter 5

- **EMUSW5**
  - 0 - Match on WPIA5 causes an exception event
  - 1 - Match on WPIA5 causes an emulation event

- **WPICNTE5**
  - 0 - Disable watchpoint instruction address counter 5
  - 1 - Enable watchpoint instruction address counter 5

- **WPICNTE4**
  - If range comparison is enabled, this bit enables the counter for range 45
  - 0 - Disable watchpoint instruction address counter 4
  - 1 - Enable watchpoint instruction address counter 4

Figure 21-3. Instruction Watchpoint Address Control Register (WPIACTL)[31:16]
Figure 21-4. Instruction Watchpoint Address Control Register (WPIACTL)[15:0]
Watchpoint Unit

Data Address Watchpoints

Each data watchpoint is controlled by four bits in the WPDACTL register, as shown in Table 21-6.

Table 21-6. Data Address Watchpoints

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WPDAACCn</td>
<td>Determines whether the match should be on a read or write access.</td>
</tr>
<tr>
<td>WPDSRCn</td>
<td>Determines which DAG the unit should monitor.</td>
</tr>
<tr>
<td>WPDCNTENn</td>
<td>Enables the counter that counts the number of address matches. If the counter is disabled, then every match causes an event.</td>
</tr>
<tr>
<td>WPDAENn</td>
<td>Enables the data watchpoint activity.</td>
</tr>
</tbody>
</table>

When the two watchpoints are associated to form a range, two additional bits are used. See Table 21-7.

Table 21-7. WPDACTL Watchpoint Control Bits

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WPDREN01</td>
<td>Indicates the two watchpoints associated to form a range.</td>
</tr>
<tr>
<td>WPDRINV01</td>
<td>Determines whether an event is caused by an address within the range identified or outside the range.</td>
</tr>
</tbody>
</table>

Note data address watchpoints always trigger emulation events.

WPDA\text{An} Registers

When the Watchpoint Unit is enabled, the values in the Data Watchpoint Address registers (WPDA\text{An}) are compared to the address on the data buses. Corresponding count values in the Data Watchpoint Address Count registers (WPDACNT\text{An}) are decremented on each match.
Figure 21-5 shows the Data Watchpoint Address registers, WPDA[1:0].

Data Watchpoint Address Registers (WPDA\textsubscript{n})

<table>
<thead>
<tr>
<th>WPDA0: 0xFFE0 7140</th>
<th>WPDA1: 0xFFE0 7144</th>
</tr>
</thead>
<tbody>
<tr>
<td>X X X X X X X X X X X X X X</td>
<td>Reset = Undefined</td>
</tr>
</tbody>
</table>

WPDA (Data Address)[31:16]

<table>
<thead>
<tr>
<th>WPDA (Data Address)[15:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>X X X X X X X X X X X X X X</td>
</tr>
</tbody>
</table>

WPDA0: 0xFFE0 7140
WPDA1: 0xFFE0 7144

Figure 21-5. Data Watchpoint Address Registers

**WPDACNT\textsubscript{n} Registers**

When the Watchpoint Unit is enabled, the count values in the Data Watchpoint Address Count Value registers (WPDACNT\textsubscript{n}) are decremented each time the address or the address bus matches a value in the WPDA\textsubscript{n} registers. Load this WPDACNT\textsubscript{n} register with a value that is one less than the number of times the watchpoint must match before triggering an event.
Watchpoint Unit

The WPDACNTn register will decrement to 0x0000 when the programmed count expires. Figure 21-6 shows the Data Watchpoint Address Count Value registers, WPDACNT[1:0].

Data Watchpoint Address Count Value Registers (WPDACNTn)

<table>
<thead>
<tr>
<th>WPDACNT0: 0xFFE0 7180</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16</td>
</tr>
<tr>
<td>X X X X X X X X X X X X X X X X</td>
</tr>
<tr>
<td>Reset = Undefined</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WPDACNT1: 0xFFE0 7184</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>X X X X X X X X X X X X X X X X</td>
</tr>
</tbody>
</table>

WPDACTL Register

For more information about the bits in the Data Watchpoint Address Control register (WPDACTL), see “Data Address Watchpoints” on page 21-10.
Figure 21-7. Data Watchpoint Address Control Register
Watchpoint Status register \((WPSTAT)\) monitors the status of the watchpoints. It may be read and written in Supervisor or Emulator modes only. When a watchpoint or watchpoint range matches, this register reflects the source of the watchpoint. The status bits in the \(WPSTAT\) register are sticky, and all of them are cleared when any write, regardless of the value, is performed to the register.

**Figure 21-8** shows the Watchpoint Status register.

**Figure 21-8. Watchpoint Status Register**
Trace Unit

The Trace Unit stores a history of the last 16 changes in program flow taken by the program sequencer. The history allows the user to recreate the program sequencer’s recent path.

The trace buffer can be enabled to cause an exception when full. The exception service routine associated with the exception saves trace buffer entries to memory. Thus, the complete path of the program sequencer since the trace buffer was enabled can be recreated.

Changes in program flow because of zero-overhead loops are not stored in the trace buffer. For debugging code that is halted within a zero-overhead loop, the iteration count is available in the Loop Count registers, LC0 and LC1.

The trace buffer can be configured to omit the recording of changes in program flow that match either the last entry or one of the last two entries. Omitting one of these entries from the record prevents the trace buffer from overflowing because of loops in the program. Because zero-overhead loops are not recorded in the trace buffer, this feature can be used to prevent trace overflow from loops that are nested four deep.

When read, the Trace Buffer register (TBUF) returns the top value from the Trace Unit stack, which contains as many as 16 entries. Each entry contains a pair of branch source and branch target addresses. A read of TBUF returns the newest entry first, starting with the branch destination. The next read provides the branch source address.
Trace Unit

The number of valid entries in \texttt{TBUF} is held in the \texttt{TBUFCNT} field of the \texttt{TBUFSTAT} register. On every second read, \texttt{TBUFCNT} is decremented. Because each entry corresponds to two pieces of data, a total of \(2 \times \texttt{TBUFCNT}\) reads empties the \texttt{TBUF} register.

- Discontinuities that are the same as either of the last two entries in the trace buffer are not recorded.

Because reading the trace buffer is a destructive operation, it is recommended that \texttt{TBUF} be read in a non-interruptible section of code.

Note, if single-level compression has occurred, the least significant bit (LSB) of the branch target address is set. If two-level compression has occurred, the LSB of the branch source address is set.

TBUFCTL Register

The Trace Unit is enabled by two control bits in the Trace Buffer Control register (\texttt{TBUFCTL}) register. First, the Trace Unit must be activated by setting the \texttt{TBUFPWR} bit. If \texttt{TBUFPWR} = 1, then setting \texttt{TBUFEN} to 1 enables the Trace Unit.

Figure 21-9 describes the Trace Buffer Control register (\texttt{TBUFCTL}). If \texttt{TBUFOVF} = 1, then the Trace Unit does not record discontinuities in the exception, NMI, and reset routines.
Figure 21-10 shows the Trace Buffer Status register (TBUFSTAT). Two reads from TBUF decrements TBUFCNT by one.

**TBUFSTAT Register**

Two reads from TBUF decrements TBUFCNT by one.
Trace Unit

TBUF Register

Figure 21-11 shows the Trace Buffer register (TBUF). The first read returns the latest branch target address. The second read returns the latest branch source address.

The Trace Unit does not record changes in program flow in:

- Emulator mode
- The exception or higher priority service routines (if TBUFOVF = 1)

In the exception service routine, the program flow discontinuities may be read from TBUF and stored in memory by the code shown in Listing 21-1.

While TBUF is being read, be sure to disable the trace buffer from recording new discontinuities.

Code to Recreate the Execution Trace in Memory

Listing 21-1 provides code that recreates the entire execution trace in memory.

21-18  ADSP-BF53x/BF56x Blackfin Processor Programming Reference
Listing 21-1. Recreating the Execution Trace in Memory

```c
[-sp] = (r7:7, p5:2); /* save registers used in this routine */
p5 = 32; /* 32 reads are needed to empty TBUF */
p2.l = buf; /* pointer to the header (first location) of the
software trace buffer */
p2.h = buf; /* the header stores the first available empty buf
location for subsequent trace dumps */

p4 = [p2++]; /* get the first available empty buf location from
the buf header */
p3.l = TBUF & 0xffff; /* low 16 bits of TBUF */
p3.h = TBUF >> 16; /* high 16 bits of TBUF */

lsetup(loop1_start, loop1_end) lc0 = p5;
loop1_start: r7 = [p3]; /* read from TBUF */
loop1_end: [p4++] = r7; /* write to memory and increment */
[p2] = p4; /* pointer to the next available buf location is
saved in the header of buf */
(r7:7, p5:3) = [sp++]; /* restore saved registers */
```

Performance Monitoring Unit

Two 32-bit counters, the Performance Monitor Counter registers (PFCNTR[1:0]) and the Performance Control register (PFCTL), count the number of occurrences of an event from within a processor core unit during a performance monitoring period. These registers provide feedback indicating the measure of load balancing between the various resources on the chip so that expected and actual usage can be compared and analyzed. In addition, events such as mispredictions and hold cycles can also be monitored.
Performance Monitoring Unit

PFCNTRn Registers

Figure 21-12 shows the Performance Monitor Counter registers, PFCNTR[1:0]. The PFCNTR0 register contains the count value of performance counter 0. The PFCNTR1 register contains the count value of performance counter 1.

Performance Monitor Counter Registers (PFCNTRn)

To enable the Performance Monitoring Unit, set the PFPWR bit in the Performance Monitor Control register (PFCTL), shown in Figure 21-13. Once the unit is enabled, individual count-enable bits (PFCENn) take effect. Use the PFCENx bits to enable or disable the performance monitors in User mode, Supervisor mode, or both. Use the PEMUSwx bits to select the type of event triggered.
**Event Monitor Table**

Table 21-8 identifies events that cause the Performance Monitor Counter registers (PFMON0 or PFMON1) to increment.

---

ADSP-BF53x/BF56x Blackfin Processor Programming Reference 21-21
### Table 21-8. Event Monitor Table

<table>
<thead>
<tr>
<th>PFMONx Fields</th>
<th>Events That Cause the Count Value to Increment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Loop 0 iterations</td>
</tr>
<tr>
<td>0x01</td>
<td>Loop 1 iterations</td>
</tr>
<tr>
<td>0x02</td>
<td>Loop buffer 0 not optimized</td>
</tr>
<tr>
<td>0x03</td>
<td>Loop buffer 1 not optimized</td>
</tr>
<tr>
<td>0x04</td>
<td>PC invariant branches (requires trace buffer to be enabled, see “TBUFCTL Register” on page 21-16)</td>
</tr>
<tr>
<td>0x06</td>
<td>Conditional branches</td>
</tr>
<tr>
<td>0x09</td>
<td>Total branches including calls, returns, branches, but not interrupts (requires trace buffer to be enabled, see “TBUFCTL Register” on page 21-16)</td>
</tr>
<tr>
<td>0x0A</td>
<td>Stalls due to CSYNC, SSYNC</td>
</tr>
<tr>
<td>0x0B</td>
<td>EXCPT instructions</td>
</tr>
<tr>
<td>0x0C</td>
<td>CSYNC, SSYNC instructions</td>
</tr>
<tr>
<td>0x0D</td>
<td>Committed instructions</td>
</tr>
<tr>
<td>0x0E</td>
<td>Interrupts taken</td>
</tr>
<tr>
<td>0x0F</td>
<td>Misaligned address violation exceptions</td>
</tr>
<tr>
<td>0x10</td>
<td>Stall cycles due to read after write hazards on DAG registers</td>
</tr>
<tr>
<td>0x13</td>
<td>Stall cycles due to RAW data hazards in computes</td>
</tr>
<tr>
<td>0x80</td>
<td>Code memory fetches postponed due to DMA collisions (minimum count of two per event)</td>
</tr>
<tr>
<td>0x81</td>
<td>Code memory TAG stalls (cache misses, or Flushl operations, count of 3 per Flushl). Note code memory stall results in a processor stall only if instruction assembly unit FIFO empties.</td>
</tr>
<tr>
<td>0x82</td>
<td>Code memory fill stalls (cacheable or non-cacheable). Note code memory stall results in a processor stall only if instruction assembly unit FIFO empties.</td>
</tr>
<tr>
<td>0x83</td>
<td>Code memory 64-bit words delivered to processor instruction assembly unit</td>
</tr>
</tbody>
</table>
The cycle counter counts \( \text{CCLK} \) cycles while the program is executing. All cycles, including execution, wait state, interrupts, and events, are counted while the processor is in User or Supervisor mode, but the cycle counter stops counting in Emulator mode.
Cycle Counter

The cycle counter is 64 bits and increments every cycle. The count value is stored in two 32-bit registers, CYCLES and CYCLES2. The least significant 32 bits (LSBs) are stored in CYCLES. The most significant 32 bits (MSBs) are stored in CYCLES2.

To ensure read coherency, a read of CYCLES stores the current CYCLES2 value in a shadow register, and all subsequent reads of CYCLES2 come from the shadow register. The shadow register is only updated on another read from CYCLES.

In User mode, these two registers may be read, but not written. In Supervisor and Emulator modes, they are read/write registers.

To enable the cycle counters, set the CCEN bit in the SYSCFG register. The following example shows how to use the cycle counter:

```c
R2 = 0;
CYCLES = R2;
CYCLES2 = R2;
R2 = SYSCFG;
BITSET(R2, 1);
SYSCFG = R2;
/* Insert code to be benchmarked here. */
R2 = SYSCFG;
BITCLR(R2, 1);
SYSCFG = R2;
```

CYCLES and CYCLES2 Registers

The Execution Cycle Count registers (CYCLES and CYCLES2) are shown in Figure 21-14. This 64-bit counter increments every CCLK cycle. The CYCLES register contains the least significant 32 bits of the cycle counter’s 64-bit count value. The most significant 32 bits are contained by CYCLES2.
Note when single-stepping through instructions in a debug environment, the CYCLES register increases in non-unity increments due to the interaction of the debugger over JTAG.

The CYCLES and CYCLES2 registers are not system MMRs, but are instead system registers.

### Execution Cycle Count Registers (CYCLES and CYCLES2)

RO in User mode, RW in Supervisor and Emulator modes

![Execution Cycle Count Registers](image-url)

---

**Figure 21-14. Execution Cycle Count Registers**
Cycle Counter

SYSCFG Register

The System Configuration register (SYSCFG) controls the configuration of the processor. This register is accessible only from the Supervisor mode.

System Configuration Register (SYSCFG)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</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>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Reset = 0x0000 0030

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</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>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[\text{SNEN (Self-Nesting Interrupt Enable)}
0 - Disable self-nesting of core interrupts
1 - Enable self-nesting of core interrupts
\]

\[\text{CCEN (Cycle Counter Enable)}
0 - Disable 64-bit, free-running cycle counter
1 - Enable 64-bit, free-running cycle counter
\]

\[\text{SSSTEP (Supervisor Single Step)}
When set, a Supervisor exception is taken after each instruction is executed. It applies only to User mode, or when processing interrupts in Supervisor mode. It is ignored if the core is processing an exception or higher priority event. If precise exception timing is required, CSYNC must be used after setting this bit.
\]

Figure 21-15. System Configuration Register
Product Identification Register

The 32-bit DSP Device ID register (DSPID) is a core MMR that contains core identification and revision fields for the core.

DSPID Register

The DSP Device ID register (DSPID), shown in Figure 21-16, is a read-only register and is part of the core.

DSP Device ID Register (DSPID)

RO

0xFFF0 5000

Reset = E504 0000

Analog Devices, Inc.[7:0]

Major Architectural Change[7:0]

Implementation[15:0]

Figure 21-16. DSP Device ID Register
Product Identification Register
A  ADSP-BF535 CONSIDERATIONS

The ADSP-BF535 processor operates differently from other Blackfin processors in some areas. This chapter describes these differences.

ADSP-BF535 Operating Modes and States

In the “Operating Modes and States” chapter, several of the descriptions do not apply to the ADSP-BF535 processor. These are:

- In Table 3-3 on page 3-4, IDLE is also a protected instruction and is not accessible in User mode.

- In “Idle State” on page 3-9, an IDLE instruction must be followed by an SSYNC instruction on the ADSP-BF535 processor for the IDLE instruction to halt the processor.

- Table 3-5 on page 3-11 (Processor State Upon Reset) does not apply to the ADSP-BF535. Please consult the ADSP-BF535 Blackfin Processor Hardware Reference for the reset values.
ADSP-BF535 Flags

Table A-1 lists the Blackfin processor instruction set and the affect on flags when these instructions execute on an ADSP-BF535 processor. The symbol definitions for the flag bits in the table are as follows.

- – indicates that the flag is NOT AFFECTED by execution of the instruction
- * indicates that the flag is SET OR CLEARED depending on execution of the instruction
- ** indicates that the flag is CLEARED by execution of the instruction
- U indicates that the flag state is UNDEFINED following execution of the instruction; if the value of this bit is needed for program execution, the program needs to check the bit prior to executing the instruction with a U in a bit field.

The flags with undefined (U) results on the ADSP-BF535 have defined results on subsequent Blackfin processors.

Because the AC0, AC1, V, AV0, AV1, AV0S, and AV1S flags do not exist on the ADSP-BF535, these flags do not appear in Table A-1.
ADSP-BF535 Considerations

Table A-1. ASTAT Flag Behavior for the ADSP-BF535

<table>
<thead>
<tr>
<th>Instruction</th>
<th>CC</th>
<th>AZ</th>
<th>AN</th>
<th>AC0 COPY</th>
<th>V COPY</th>
<th>AQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IF CC JUMP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTS, RTI, RTX, RTN, RTE (Return)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSETUP, LOOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Pointer Register</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Data Register</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Half-Word – Zero-Extended</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Half-Word – Sign-Extended</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load High Data Register Half</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Low Data Register Half</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Byte – Zero-Extended</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Byte – Sign-Extended</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store Pointer Register</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store Data Register</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store High Data Register Half</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store Low Data Register Half</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store Byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move Register (except acc to dreg)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- indicates that the flag is NOT AFFECTED by execution of the instruction
* indicates that the flag is SET OR CLEARED depending on execution of the instruction
** indicates that the flag is CLEARED by execution of the instruction
U indicates that the flag state is UNDEFINED following execution of the instruction; if the value of this bit is needed for program execution, the program needs to check the bit prior to executing the instruction with a U in a bit field.
### ADSP-BF535 Flags

Table A-1. ASTAT Flag Behavior for the ADSP-BF535 (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>CC</th>
<th>AZ</th>
<th>AN</th>
<th>AC0 COPY</th>
<th>V COPY</th>
<th>AQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move Register (acc to dreg)</td>
<td></td>
<td>U</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move Conditional</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move Half to Full Word – Zero-Extended</td>
<td></td>
<td>*</td>
<td>**</td>
<td>**</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move Half to Full Word – Sign-Extended</td>
<td></td>
<td>*</td>
<td>*</td>
<td>**</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move Register Half (except acc to half dreg)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move Register Half (acc to half dreg)</td>
<td></td>
<td>U</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move Byte – Zero-Extended</td>
<td></td>
<td>*</td>
<td>*</td>
<td>**</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move Byte – Sign-Extended</td>
<td></td>
<td>*</td>
<td>*</td>
<td>**</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--SP (Push)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--SP (Push Multiple)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SP++ (Pop)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SP++ (Pop Multiple)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINK, UNLINK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compare Data Register</td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td>U</td>
</tr>
<tr>
<td>Compare Pointer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compare Accumulator</td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td>U</td>
</tr>
<tr>
<td>Move CC</td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Negate CC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp; (AND)</td>
<td></td>
<td>*</td>
<td>*</td>
<td>**</td>
<td>**</td>
<td></td>
</tr>
<tr>
<td>– (NOT One’s-Complement)</td>
<td></td>
<td>*</td>
<td>*</td>
<td>**</td>
<td>**</td>
<td></td>
</tr>
</tbody>
</table>

– indicates that the flag is NOT AFFECTED by execution of the instruction

* indicates that the flag is SET OR CLEARED depending on execution of the instruction

** indicates that the flag is CLEARED by execution of the instruction

U indicates that the flag state is UNDEFINED following execution of the instruction; if the value of this bit is needed for program execution, the program needs to check the bit prior to executing the instruction with a U in a bit field.
### Table A-1. ASTAT Flag Behavior for the ADSP-BF535 (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>CC</th>
<th>AZ</th>
<th>AN</th>
<th>( AC_0 )</th>
<th>( V_0 )</th>
<th>AQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>) (OR)</td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>(^\wedge) (Exclusive-OR)</td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>BXORSHIFT, BXOR</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BITCLR</td>
<td></td>
<td>*</td>
<td>*</td>
<td>U</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>BITSET</td>
<td></td>
<td></td>
<td>U</td>
<td>U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BITTGL</td>
<td></td>
<td>*</td>
<td>*</td>
<td>U</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>BITTST</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEPOSIT</td>
<td></td>
<td>*</td>
<td>*</td>
<td>U</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>EXTRACT</td>
<td></td>
<td>*</td>
<td>*</td>
<td>U</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>BITMUX</td>
<td></td>
<td>U</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ONES (One’s-Population Count)</td>
<td></td>
<td>U</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add with Shift (preg version)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add with Shift (dreg version)</td>
<td>*</td>
<td></td>
<td>U</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift with Add</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arithmetic Shift (to dreg)</td>
<td>*</td>
<td></td>
<td>*</td>
<td>U</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>Arithmetic Shift (to A0)</td>
<td>*</td>
<td></td>
<td>*</td>
<td>U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arithmetic Shift (to A1)</td>
<td>*</td>
<td></td>
<td>*</td>
<td>U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical Shift (to preg)</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical Shift (to dreg)</td>
<td>*</td>
<td></td>
<td>*</td>
<td>U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical Shift (to A0)</td>
<td>*</td>
<td></td>
<td>*</td>
<td>U</td>
<td>U</td>
<td></td>
</tr>
</tbody>
</table>

- indicates that the flag is NOT AFFECTED by execution of the instruction
* indicates that the flag is SET OR CLEARED depending on execution of the instruction
** indicates that the flag is CLEARED by execution of the instruction
U indicates that the flag state is UNDEFINED following execution of the instruction; if the value of this bit is needed for program execution, the program needs to check the bit prior to executing the instruction with a U in a bit field.
## ADSP-BF535 Flags

Table A-1. ASTAT Flag Behavior for the ADSP-BF535 (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>CC</th>
<th>AZ</th>
<th>AN</th>
<th>AC0 COPY</th>
<th>V_ COPY</th>
<th>AQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical Shift (to A1)</td>
<td>–</td>
<td>*</td>
<td>*</td>
<td>U</td>
<td>U</td>
<td>–</td>
</tr>
<tr>
<td>ROT (Rotate)</td>
<td>*</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>ABS (to dreg)</td>
<td>–</td>
<td>*</td>
<td>**</td>
<td>U</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>ABS (to A0)</td>
<td>–</td>
<td>*</td>
<td>**</td>
<td>U</td>
<td>U</td>
<td>–</td>
</tr>
<tr>
<td>ABS (to A1)</td>
<td>–</td>
<td>*</td>
<td>**</td>
<td>U</td>
<td>U</td>
<td>–</td>
</tr>
<tr>
<td>Add (preg version)</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Add (dreg version)</td>
<td>–</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>–</td>
</tr>
<tr>
<td>Add/Subtract – Prescale Down</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Add/Subtract – Prescale Up</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Add Immediate (to preg or ireg)</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Add Immediate (to dreg)</td>
<td>–</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>DIVS, DIVQ (Divide Primitive)</td>
<td>–</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>*</td>
</tr>
<tr>
<td>EXPADJ</td>
<td>–</td>
<td>U</td>
<td>U</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>MAX</td>
<td>–</td>
<td>*</td>
<td>*</td>
<td>U</td>
<td>U</td>
<td>–</td>
</tr>
<tr>
<td>MIN</td>
<td>–</td>
<td>*</td>
<td>*</td>
<td>U</td>
<td>U</td>
<td>–</td>
</tr>
<tr>
<td>Modify – Decrement (to preg or ireg)</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Modify – Decrement (to acc)</td>
<td>–</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>–</td>
</tr>
<tr>
<td>Modify – Increment (to preg or ireg)</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Modify – Increment (extracted to dreg)</td>
<td>–</td>
<td>*</td>
<td>*</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Modify – Increment (to acc)</td>
<td>–</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>–</td>
</tr>
</tbody>
</table>

- indicates that the flag is NOT AFFECTED by execution of the instruction
* indicates that the flag is SET OR CLEARED depending on execution of the instruction
** indicates that the flag is CLEARED by execution of the instruction
U indicates that the flag state is UNDEFINED following execution of the instruction; if the value of this bit is needed for program execution, the program needs to check the bit prior to executing the instruction with a U in a bit field.
## ADSP-BF535 Considerations

Table A-1. ASTAT Flag Behavior for the ADSP-BF535 (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>CC</th>
<th>AZ</th>
<th>AN</th>
<th>AC0 COPY</th>
<th>V COPY</th>
<th>AQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>U</td>
<td>–</td>
</tr>
<tr>
<td>Multiply 32-Bit Operands</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>U</td>
<td>–</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half-Register</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>U</td>
<td>–</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>U</td>
<td>–</td>
</tr>
<tr>
<td>Negate (Two’s-Complement) (to dreg)</td>
<td>–</td>
<td>*</td>
<td>*</td>
<td>U</td>
<td>*</td>
<td>–</td>
</tr>
<tr>
<td>Negate (Two’s-Complement) (to A0)</td>
<td>–</td>
<td>*</td>
<td>*</td>
<td>U</td>
<td>U</td>
<td>–</td>
</tr>
<tr>
<td>Negate (Two’s-Complement) (to A1)</td>
<td>–</td>
<td>*</td>
<td>*</td>
<td>U</td>
<td>U</td>
<td>–</td>
</tr>
<tr>
<td>RND (Round to Half-Word)</td>
<td>–</td>
<td>*</td>
<td>*</td>
<td>U</td>
<td>*</td>
<td>–</td>
</tr>
<tr>
<td>Saturate</td>
<td>–</td>
<td>*</td>
<td>*</td>
<td>U</td>
<td>U</td>
<td>–</td>
</tr>
<tr>
<td>SIGNBITS</td>
<td>–</td>
<td>U</td>
<td>U</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Subtract</td>
<td>–</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>–</td>
</tr>
<tr>
<td>Subtract Immediate (to ireg)</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Idle</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Core Synchronize</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>System Synchronize</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>EMUEXCEPT (Force Emulation)</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Disable Interrupts</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Enable Interrupts</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>RAISE (Force Interrupt / Reset)</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

– indicates that the flag is NOT AFFECTED by execution of the instruction
* indicates that the flag is SET OR CLEARED depending on execution of the instruction
** indicates that the flag is CLEARED by execution of the instruction
U indicates that the flag state is UNDEFINED following execution of the instruction; if the value of this bit is needed for program execution, the program needs to check the bit prior to executing the instruction with a U in a bit field.
## ADSP-BF535 Flags

Table A-1. ASTAT Flag Behavior for the ADSP-BF535 (Cont'd)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>CC</th>
<th>AZ</th>
<th>AN</th>
<th>AC0_COPY</th>
<th>V_COPY</th>
<th>AQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXCPT (Force Exception)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test and Set Byte (Atomic)</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No Op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PREFETCH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLUSH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLUSHINV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFLUSH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALIGN8, ALIGN16, ALIGN24</td>
<td></td>
<td>U</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DISALGNEXCPT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BYTEOP3P (Dual 16-Bit Add / Clip)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual 16-Bit Accumulator Extraction with Addition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BYTEOP16P (Quad 8-Bit Add)</td>
<td></td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>BYTEOP1P (Quad 8-Bit Average – Byte)</td>
<td></td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>BYTEOP2P (Quad 8-Bit Average – Half-Word)</td>
<td></td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>BYTEPACK (Quad 8-Bit Pack)</td>
<td></td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>BYTEOP16M (Quad 8-Bit Subtract)</td>
<td></td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>SAA (Quad 8-Bit Subtract-Absolute-Accumulate)</td>
<td></td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>BYTEUNPACK (Quad 8-Bit Unpack)</td>
<td></td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>Add on Sign</td>
<td></td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>VIT_MAX (Compare-Select)</td>
<td></td>
<td>U</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- indicates that the flag is NOT AFFECTED by execution of the instruction
* indicates that the flag is SET OR CLEARED depending on execution of the instruction
** indicates that the flag is CLEARED by execution of the instruction
U indicates that the flag state is UNDEFINED following execution of the instruction; if the value of this bit is needed for program execution, the program needs to check the bit prior to executing the instruction with a U in a bit field.
## ADSP-BF535 Considerations

Table A-1. ASTAT Flag Behavior for the ADSP-BF535 (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>CC</th>
<th>AZ</th>
<th>AN</th>
<th>AC0_COPY</th>
<th>V_COPY</th>
<th>AQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector ABS</td>
<td></td>
<td>*</td>
<td>**</td>
<td>U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td></td>
<td>*</td>
<td>**</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Arithmetic Shift</td>
<td></td>
<td>*</td>
<td>*</td>
<td>U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Logical Shift</td>
<td></td>
<td>*</td>
<td>*</td>
<td>U</td>
<td>**</td>
<td></td>
</tr>
<tr>
<td>Vector MAX</td>
<td></td>
<td>*</td>
<td>*</td>
<td>U</td>
<td>**</td>
<td></td>
</tr>
<tr>
<td>Vector MIN</td>
<td></td>
<td>*</td>
<td>*</td>
<td>U</td>
<td>**</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply</td>
<td></td>
<td></td>
<td></td>
<td>U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td></td>
<td></td>
<td></td>
<td>U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Negate (Two’s-Complement)</td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector PACK</td>
<td></td>
<td>U</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector SEARCH</td>
<td></td>
<td>U</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- indicates that the flag is NOT AFFECTED by execution of the instruction
* indicates that the flag is SET OR CLEARED depending on execution of the instruction
** indicates that the flag is CLEARED by execution of the instruction
U indicates that the flag state is UNDEFINED following execution of the instruction; if the value of this bit is needed for program execution, the program needs to check the bit prior to executing the instruction with a U in a bit field.
ADSP-BF535 Flags
The Blackfin processor’s memory-mapped registers (MMRs) are in the address range 0xFFE0 0000 – 0xFFFF FFFF.

All core MMRs must be accessed with a 32-bit read or write access.

This appendix lists core MMR addresses and register names. To find more information about an MMR, refer to the page shown in the “See Section” column. When viewing the PDF version of this document, click a reference in the “See Section” column to jump to additional information about the MMR.

L1 Data Memory Controller Registers

L1 Data Memory Controller registers (0xFFE0 0000 – 0xFFE0 0404)

Table B-1. L1 Data Memory Controller Registers

<table>
<thead>
<tr>
<th>Memory-mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 0004</td>
<td>DMEM_CONTROL</td>
<td>“DMEM_CONTROL Register” on page 6-24</td>
</tr>
<tr>
<td>0xFFE0 0008</td>
<td>DCPLB_STATUS</td>
<td>“DCPLB_STATUS and ICPLB_STATUS Registers” on page 6-61</td>
</tr>
<tr>
<td>0xFFE0 000C</td>
<td>DCPLB_FAULT_ADDR</td>
<td>“DCPLB_FAULT_ADDR and ICPLB_FAULT_ADDR Registers” on page 6-63</td>
</tr>
<tr>
<td>0xFFE0 0100</td>
<td>DCPLB_ADDR0</td>
<td>“DCPLB_ADDRs Registers” on page 6-59</td>
</tr>
</tbody>
</table>
L1 Data Memory Controller Registers

Table B-1. L1 Data Memory Controller Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 0104</td>
<td>DCPLB_ADDR1</td>
<td>“DCPLB_ADDRx Registers” on page 6-59</td>
</tr>
<tr>
<td>0xFFE0 0108</td>
<td>DCPLB_ADDR2</td>
<td>“DCPLB_ADDRx Registers” on page 6-59</td>
</tr>
<tr>
<td>0xFFE0 010C</td>
<td>DCPLB_ADDR3</td>
<td>“DCPLB_ADDRx Registers” on page 6-59</td>
</tr>
<tr>
<td>0xFFE0 0110</td>
<td>DCPLB_ADDR4</td>
<td>“DCPLB_ADDRx Registers” on page 6-59</td>
</tr>
<tr>
<td>0xFFE0 0114</td>
<td>DCPLB_ADDR5</td>
<td>“DCPLB_ADDRx Registers” on page 6-59</td>
</tr>
<tr>
<td>0xFFE0 0118</td>
<td>DCPLB_ADDR6</td>
<td>“DCPLB_ADDRx Registers” on page 6-59</td>
</tr>
<tr>
<td>0xFFE0 011C</td>
<td>DCPLB_ADDR7</td>
<td>“DCPLB_ADDRx Registers” on page 6-59</td>
</tr>
<tr>
<td>0xFFE0 0120</td>
<td>DCPLB_ADDR8</td>
<td>“DCPLB_ADDRx Registers” on page 6-59</td>
</tr>
<tr>
<td>0xFFE0 0124</td>
<td>DCPLB_ADDR9</td>
<td>“DCPLB_ADDRx Registers” on page 6-59</td>
</tr>
<tr>
<td>0xFFE0 0128</td>
<td>DCPLB_ADDR10</td>
<td>“DCPLB_ADDRx Registers” on page 6-59</td>
</tr>
<tr>
<td>0xFFE0 012C</td>
<td>DCPLB_ADDR11</td>
<td>“DCPLB_ADDRx Registers” on page 6-59</td>
</tr>
<tr>
<td>0xFFE0 0130</td>
<td>DCPLB_ADDR12</td>
<td>“DCPLB_ADDRx Registers” on page 6-59</td>
</tr>
<tr>
<td>0xFFE0 0134</td>
<td>DCPLB_ADDR13</td>
<td>“DCPLB_ADDRx Registers” on page 6-59</td>
</tr>
<tr>
<td>0xFFE0 0138</td>
<td>DCPLB_ADDR14</td>
<td>“DCPLB_ADDRx Registers” on page 6-59</td>
</tr>
<tr>
<td>0xFFE0 013C</td>
<td>DCPLB_ADDR15</td>
<td>“DCPLB_ADDRx Registers” on page 6-59</td>
</tr>
<tr>
<td>0xFFE0 0200</td>
<td>DCPLB_DATA0</td>
<td>“DCPLB_DATAx Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 0204</td>
<td>DCPLB_DATA1</td>
<td>“DCPLB_DATAx Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 0208</td>
<td>DCPLB_DATA2</td>
<td>“DCPLB_DATAx Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 020C</td>
<td>DCPLB_DATA3</td>
<td>“DCPLB_DATAx Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 0210</td>
<td>DCPLB_DATA4</td>
<td>“DCPLB_DATAx Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 0214</td>
<td>DCPLB_DATA5</td>
<td>“DCPLB_DATAx Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 0218</td>
<td>DCPLB_DATA6</td>
<td>“DCPLB_DATAx Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 021C</td>
<td>DCPLB_DATA7</td>
<td>“DCPLB_DATAx Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 0220</td>
<td>DCPLB_DATA8</td>
<td>“DCPLB_DATAx Registers” on page 6-57</td>
</tr>
</tbody>
</table>
Core MMR Assignments

Table B-1. L1 Data Memory Controller Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 0224</td>
<td>DCPLB_DATA9</td>
<td>“DCPLB_DATAx Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 0228</td>
<td>DCPLB_DATA10</td>
<td>“DCPLB_DATAx Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 022C</td>
<td>DCPLB_DATA11</td>
<td>“DCPLB_DATAx Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 0230</td>
<td>DCPLB_DATA12</td>
<td>“DCPLB_DATAx Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 0234</td>
<td>DCPLB_DATA13</td>
<td>“DCPLB_DATAx Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 0238</td>
<td>DCPLB_DATA14</td>
<td>“DCPLB_DATAx Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 023C</td>
<td>DCPLB_DATA15</td>
<td>“DCPLB_DATAx Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 0300</td>
<td>DTEST_COMMAND</td>
<td>“DTEST_COMMAND Register” on page 6-39</td>
</tr>
<tr>
<td>0xFFE0 0400</td>
<td>DTEST_DATA0</td>
<td>“DTEST_DATA0 Register” on page 6-42</td>
</tr>
<tr>
<td>0xFFE0 0404</td>
<td>DTEST_DATA1</td>
<td>“DTEST_DATA1 Register” on page 6-41</td>
</tr>
</tbody>
</table>
L1 Instruction Memory Controller Registers

L1 Instruction Memory Controller registers (0xFFE0 1004 –0xFFE0 1404)

Table B-2. L1 Instruction Memory Controller Registers

<table>
<thead>
<tr>
<th>Memory-mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 1004</td>
<td>IMEM_CONTROL</td>
<td>“IMEM_CONTROL Register” on page 6-5</td>
</tr>
<tr>
<td>0xFFE0 1008</td>
<td>ICPLB_STATUS</td>
<td>“DCPLB_STATUS and ICPLB_STATUS Registers” on page 6-61</td>
</tr>
<tr>
<td>0xFFE0 100C</td>
<td>ICPLB_FAULT_ADDR</td>
<td>“DCPLB_FAULT_ADDR and ICPLB_FAULT_ADDR Registers” on page 6-63</td>
</tr>
<tr>
<td>0xFFE0 1100</td>
<td>ICPLB_ADDR0</td>
<td>“ICPLB_ADDRx Registers” on page 6-60</td>
</tr>
<tr>
<td>0xFFE0 1104</td>
<td>ICPLB_ADDR1</td>
<td>“ICPLB_ADDRx Registers” on page 6-60</td>
</tr>
<tr>
<td>0xFFE0 1108</td>
<td>ICPLB_ADDR2</td>
<td>“ICPLB_ADDRx Registers” on page 6-60</td>
</tr>
<tr>
<td>0xFFE0 110C</td>
<td>ICPLB_ADDR3</td>
<td>“ICPLB_ADDRx Registers” on page 6-60</td>
</tr>
<tr>
<td>0xFFE0 1110</td>
<td>ICPLB_ADDR4</td>
<td>“ICPLB_ADDRx Registers” on page 6-60</td>
</tr>
<tr>
<td>0xFFE0 1114</td>
<td>ICPLB_ADDR5</td>
<td>“ICPLB_ADDRx Registers” on page 6-60</td>
</tr>
<tr>
<td>0xFFE0 1118</td>
<td>ICPLB_ADDR6</td>
<td>“ICPLB_ADDRx Registers” on page 6-60</td>
</tr>
<tr>
<td>0xFFE0 111C</td>
<td>ICPLB_ADDR7</td>
<td>“ICPLB_ADDRx Registers” on page 6-60</td>
</tr>
<tr>
<td>0xFFE0 1120</td>
<td>ICPLB_ADDR8</td>
<td>“ICPLB_ADDRx Registers” on page 6-60</td>
</tr>
<tr>
<td>0xFFE0 1124</td>
<td>ICPLB_ADDR9</td>
<td>“ICPLB_ADDRx Registers” on page 6-60</td>
</tr>
<tr>
<td>0xFFE0 1128</td>
<td>ICPLB_ADDR10</td>
<td>“ICPLB_ADDRx Registers” on page 6-60</td>
</tr>
<tr>
<td>0xFFE0 112C</td>
<td>ICPLB_ADDR11</td>
<td>“ICPLB_ADDRx Registers” on page 6-60</td>
</tr>
<tr>
<td>0xFFE0 1130</td>
<td>ICPLB_ADDR12</td>
<td>“ICPLB_ADDRx Registers” on page 6-60</td>
</tr>
</tbody>
</table>
Core MMR Assignments

Table B-2. L1 Instruction Memory Controller Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFF0 1134</td>
<td>ICPLB_ADDR13</td>
<td>“ICPLB_ADDRx Registers” on page 6-60</td>
</tr>
<tr>
<td>0xFFF0 1138</td>
<td>ICPLB_ADDR14</td>
<td>“ICPLB_ADDRx Registers” on page 6-60</td>
</tr>
<tr>
<td>0xFFF0 113C</td>
<td>ICPLB_ADDR15</td>
<td>“ICPLB_ADDRx Registers” on page 6-60</td>
</tr>
<tr>
<td>0xFFF0 1200</td>
<td>ICPLB_DATA0</td>
<td>“ICPLB_DATAx Registers” on page 6-55</td>
</tr>
<tr>
<td>0xFFF0 1204</td>
<td>ICPLB_DATA1</td>
<td>“ICPLB_DATAx Registers” on page 6-55</td>
</tr>
<tr>
<td>0xFFF0 1208</td>
<td>ICPLB_DATA2</td>
<td>“ICPLB_DATAx Registers” on page 6-55</td>
</tr>
<tr>
<td>0xFFF0 120C</td>
<td>ICPLB_DATA3</td>
<td>“ICPLB_DATAx Registers” on page 6-55</td>
</tr>
<tr>
<td>0xFFF0 1210</td>
<td>ICPLB_DATA4</td>
<td>“ICPLB_DATAx Registers” on page 6-55</td>
</tr>
<tr>
<td>0xFFF0 1214</td>
<td>ICPLB_DATA5</td>
<td>“ICPLB_DATAx Registers” on page 6-55</td>
</tr>
<tr>
<td>0xFFF0 1218</td>
<td>ICPLB_DATA6</td>
<td>“ICPLB_DATAx Registers” on page 6-55</td>
</tr>
<tr>
<td>0xFFF0 121C</td>
<td>ICPLB_DATA7</td>
<td>“ICPLB_DATAx Registers” on page 6-55</td>
</tr>
<tr>
<td>0xFFF0 1220</td>
<td>ICPLB_DATA8</td>
<td>“ICPLB_DATAx Registers” on page 6-55</td>
</tr>
<tr>
<td>0xFFF0 1224</td>
<td>ICPLB_DATA9</td>
<td>“ICPLB_DATAx Registers” on page 6-55</td>
</tr>
<tr>
<td>0xFFF0 1228</td>
<td>ICPLB_DATA10</td>
<td>“ICPLB_DATAx Registers” on page 6-55</td>
</tr>
<tr>
<td>0xFFF0 122C</td>
<td>ICPLB_DATA11</td>
<td>“ICPLB_DATAx Registers” on page 6-55</td>
</tr>
<tr>
<td>0xFFF0 1230</td>
<td>ICPLB_DATA12</td>
<td>“ICPLB_DATAx Registers” on page 6-55</td>
</tr>
<tr>
<td>0xFFF0 1234</td>
<td>ICPLB_DATA13</td>
<td>“ICPLB_DATAx Registers” on page 6-55</td>
</tr>
<tr>
<td>0xFFF0 1238</td>
<td>ICPLB_DATA14</td>
<td>“ICPLB_DATAx Registers” on page 6-55</td>
</tr>
<tr>
<td>0xFFF0 123C</td>
<td>ICPLB_DATA15</td>
<td>“ICPLB_DATAx Registers” on page 6-55</td>
</tr>
<tr>
<td>0xFFF0 1300</td>
<td>ITEST_COMMAND</td>
<td>“ITEST_COMMAND Register” on page 6-21</td>
</tr>
<tr>
<td>0xFFF0 1400</td>
<td>ITEST_DATA0</td>
<td>“ITEST_DATA0 Register” on page 6-23</td>
</tr>
<tr>
<td>0xFFF0 1404</td>
<td>ITEST_DATA1</td>
<td>“ITEST_DATA1 Register” on page 6-22</td>
</tr>
</tbody>
</table>
## Interrupt Controller Registers

Interrupt Controller registers (0xFFE0 2000 – 0xFFE0 2110)

Table B-3. Interrupt Controller Registers

<table>
<thead>
<tr>
<th>Memory-mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 2000</td>
<td>EVT0 (EMU)</td>
<td>“Core Event Vector Table” on page 4-42</td>
</tr>
<tr>
<td>0xFFE0 2004</td>
<td>EVT1 (RST)</td>
<td>“Core Event Vector Table” on page 4-42</td>
</tr>
<tr>
<td>0xFFE0 2008</td>
<td>EVT2 (NMI)</td>
<td>“Core Event Vector Table” on page 4-42</td>
</tr>
<tr>
<td>0xFFE0 200C</td>
<td>EVT3 (EVX)</td>
<td>“Core Event Vector Table” on page 4-42</td>
</tr>
<tr>
<td>0xFFE0 2010</td>
<td>EVT4</td>
<td>“Core Event Vector Table” on page 4-42</td>
</tr>
<tr>
<td>0xFFE0 2014</td>
<td>EVT5 (IVHW)</td>
<td>“Core Event Vector Table” on page 4-42</td>
</tr>
<tr>
<td>0xFFE0 2018</td>
<td>EVT6 (TMR)</td>
<td>“Core Event Vector Table” on page 4-42</td>
</tr>
<tr>
<td>0xFFE0 201C</td>
<td>EVT7 (IVG7)</td>
<td>“Core Event Vector Table” on page 4-42</td>
</tr>
<tr>
<td>0xFFE0 2020</td>
<td>EVT8 (IVG8)</td>
<td>“Core Event Vector Table” on page 4-42</td>
</tr>
<tr>
<td>0xFFE0 2024</td>
<td>EVT9 (IVG9)</td>
<td>“Core Event Vector Table” on page 4-42</td>
</tr>
<tr>
<td>0xFFE0 2028</td>
<td>EVT10 (IVG10)</td>
<td>“Core Event Vector Table” on page 4-42</td>
</tr>
<tr>
<td>0xFFE0 202C</td>
<td>EVT11 (IVG11)</td>
<td>“Core Event Vector Table” on page 4-42</td>
</tr>
<tr>
<td>0xFFE0 2030</td>
<td>EVT12 (IVG12)</td>
<td>“Core Event Vector Table” on page 4-42</td>
</tr>
</tbody>
</table>
Core MMR Assignments

Table B-3. Interrupt Controller Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 2034</td>
<td>EVT13 (IVG13)</td>
<td>“Core Event Vector Table” on page 4-42</td>
</tr>
<tr>
<td>0xFFE0 2038</td>
<td>EVT14 (IVG14)</td>
<td>“Core Event Vector Table” on page 4-42</td>
</tr>
<tr>
<td>0xFFE0 203C</td>
<td>EVT15 (IVG15)</td>
<td>“Core Event Vector Table” on page 4-42</td>
</tr>
<tr>
<td>0xFFE0 2104</td>
<td>IMASK</td>
<td>“IMASK Register” on page 4-38</td>
</tr>
<tr>
<td>0xFFE0 2108</td>
<td>IPEND</td>
<td>“IPEND Register” on page 4-40</td>
</tr>
<tr>
<td>0xFFE0 210C</td>
<td>ILAT</td>
<td>“ILAT Register” on page 4-39</td>
</tr>
<tr>
<td>0xFFE0 2110</td>
<td>IPRIO</td>
<td>“IPRIO Register and Write Buffer Depth” on page 6-35</td>
</tr>
</tbody>
</table>

Debug, MP, and Emulation Unit Registers

Debug, MP, and Emulation Unit registers (0xFFE0 5000 – 0xFFE0 5008)

Table B-4. Debug and Emulation Unit Registers

<table>
<thead>
<tr>
<th>Memory-mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 5000</td>
<td>DSPID</td>
<td>“DSPID Register” on page 21-27</td>
</tr>
</tbody>
</table>
Trace Unit Registers

Trace Unit registers (0xFFE0 6000 – 0xFFE0 6100)

Table B-5. Trace Unit Registers

<table>
<thead>
<tr>
<th>Memory-mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 6000</td>
<td>TBUFCTL</td>
<td>“TBUFCTL Register” on page 21-16</td>
</tr>
<tr>
<td>0xFFE0 6004</td>
<td>TBUFSTAT</td>
<td>“TBUFSTAT Register” on page 21-17</td>
</tr>
<tr>
<td>0xFFE0 6100</td>
<td>TBUF</td>
<td>“TBUF Register” on page 21-18</td>
</tr>
</tbody>
</table>

Watchpoint and Patch Registers

Watchpoint and Patch registers (0xFFE0 7000 – 0xFFE0 7200)

Table B-6. Watchpoint and Patch Registers

<table>
<thead>
<tr>
<th>Memory-mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 7000</td>
<td>WPIACTL</td>
<td>“WPIACTL Register” on page 21-7</td>
</tr>
<tr>
<td>0xFFE0 7040</td>
<td>WPIA0</td>
<td>“WPIAn Registers” on page 21-5</td>
</tr>
<tr>
<td>0xFFE0 7044</td>
<td>WPIA1</td>
<td>“WPIAn Registers” on page 21-5</td>
</tr>
<tr>
<td>0xFFE0 7048</td>
<td>WPIA2</td>
<td>“WPIAn Registers” on page 21-5</td>
</tr>
<tr>
<td>0xFFE0 704C</td>
<td>WPIA3</td>
<td>“WPIAn Registers” on page 21-5</td>
</tr>
<tr>
<td>0xFFE0 7050</td>
<td>WPIA4</td>
<td>“WPIAn Registers” on page 21-5</td>
</tr>
<tr>
<td>0xFFE0 7054</td>
<td>WPIA5</td>
<td>“WPIAn Registers” on page 21-5</td>
</tr>
<tr>
<td>0xFFE0 7080</td>
<td>WPIACNT0</td>
<td>“WPIACNTn Registers” on page 21-6</td>
</tr>
<tr>
<td>0xFFE0 7084</td>
<td>WPIACNT1</td>
<td>“WPIACNTn Registers” on page 21-6</td>
</tr>
<tr>
<td>0xFFE0 7088</td>
<td>WPIACNT2</td>
<td>“WPIACNTn Registers” on page 21-6</td>
</tr>
</tbody>
</table>
Core MMR Assignments

Table B-6. Performance Monitor Registers

<table>
<thead>
<tr>
<th>Memory-mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 708C</td>
<td>WPIACNT3</td>
<td>“WPIACNTn Registers” on page 21-6</td>
</tr>
<tr>
<td>0xFFE0 7090</td>
<td>WPIACNT4</td>
<td>“WPIACNTn Registers” on page 21-6</td>
</tr>
<tr>
<td>0xFFE0 7094</td>
<td>WPIACNT5</td>
<td>“WPIACNTn Registers” on page 21-6</td>
</tr>
<tr>
<td>0xFFE0 7100</td>
<td>WPDACTL</td>
<td>“WPDACTL Register” on page 21-12</td>
</tr>
<tr>
<td>0xFFE0 7140</td>
<td>WPDA0</td>
<td>“WPDAan Registers” on page 21-10</td>
</tr>
<tr>
<td>0xFFE0 7144</td>
<td>WPDA1</td>
<td>“WPDAan Registers” on page 21-10</td>
</tr>
<tr>
<td>0xFFE0 7180</td>
<td>WPDACNT0</td>
<td>“WPDACNTn Registers” on page 21-11</td>
</tr>
<tr>
<td>0xFFE0 7184</td>
<td>WPDACNT1</td>
<td>“WPDACNTn Registers” on page 21-11</td>
</tr>
<tr>
<td>0xFFE0 7200</td>
<td>WPSTAT</td>
<td>“WPSTAT Register” on page 21-14</td>
</tr>
</tbody>
</table>

Performance Monitor Registers

Performance Monitor registers (0xFFE0 8000 – 0xFFE0 8104)

Table B-7. Performance Monitor Registers

<table>
<thead>
<tr>
<th>Memory-mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 8000</td>
<td>PFCTL</td>
<td>“PFCTL Register” on page 21-20</td>
</tr>
<tr>
<td>0xFFE0 8100</td>
<td>PFCNTR0</td>
<td>“PFCNTRn Registers” on page 21-20</td>
</tr>
<tr>
<td>0xFFE0 8104</td>
<td>PFCNTR1</td>
<td>“PFCNTRn Registers” on page 21-20</td>
</tr>
</tbody>
</table>
Performance Monitor Registers
This appendix describes the operation codes (or, “opcodes”) for each Blackfin instruction. The purpose is to specify the instruction codes for Blackfin software and tools developers.

Introduction

This format separates instructions as much as practical for maximum clarity. Users are better served by clear, distinct opcode descriptions instead of confusing tables of convoluted algorithms to construct each opcode. The format minimizes the number of variables the reader must master to represent or recognize bit fields within the opcodes. This more explicit format expands the listings to more pages, but is easier and quicker to reference. The success of this document is measured by how little time it takes for you to find the information you want.

However, some instructions (such as Multiply-and-Accumulate and Vector Multiply-and-Accumulate) support so many options and variations that individual listings for each version are simply not manageable. In those cases, bit fields are defined and used.

Appendix Organization

This appendix lists each instruction with its corresponding opcode. Instructions are grouped according to function.

The instructions also appear in order of their corresponding opcodes in “Instructions Listed By Operation Code” on page C-139.
Introduction

Glossary

The following terms appear throughout this document. Without trying to explain the Blackfin architecture, here are the terms used with their definitions. See chapters 1 through 6 for more details on the architecture.

Register Names

The architecture includes the following registers.

Table C-1. Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accumulators</td>
<td>The set of 40-bit registers A1 and A0 that normally contain data that is being</td>
</tr>
<tr>
<td></td>
<td>manipulated. Each Accumulator can be accessed in five ways—as one 40-bit</td>
</tr>
<tr>
<td></td>
<td>register, as one 32-bit register (designated as A1.W or A0.W), as two 16-bit</td>
</tr>
<tr>
<td></td>
<td>registers similar to Data registers (designated as A1.H, A1.L, A0.H, or A0.L)</td>
</tr>
<tr>
<td></td>
<td>and as one 8-bit register (designated A1.X or A0.X) for the bits that extend</td>
</tr>
<tr>
<td></td>
<td>beyond bit 31.</td>
</tr>
<tr>
<td>Data Registers</td>
<td>The set of 32-bit registers R0, R1, ..., R6, R7 that normally contain data for</td>
</tr>
<tr>
<td></td>
<td>manipulation. Abbreviated D-register or Dreg. Data registers can be accessed</td>
</tr>
<tr>
<td></td>
<td>as 32-bit registers, or optionally as two independent 16-bit registers. The least</td>
</tr>
<tr>
<td></td>
<td>significant 16 bits of each register is called the &quot;low&quot; half and is designated</td>
</tr>
<tr>
<td></td>
<td>with &quot;.L&quot; following the register name. The most significant 16-bit is called the</td>
</tr>
<tr>
<td></td>
<td>&quot;high&quot; half and is designated with &quot;.H&quot; following the name. Example: R7.L,</td>
</tr>
<tr>
<td></td>
<td>r2.h, r4.L, R0.h.</td>
</tr>
<tr>
<td>Pointer Registers</td>
<td>The set of 32-bit registers P0, P1, ..., P4, P5, including SP and FP that</td>
</tr>
<tr>
<td></td>
<td>normally contain byte addresses of data structures. Accessed only as a 32-bit</td>
</tr>
<tr>
<td></td>
<td>register. Abbreviated P-register or Preg. Example: p2, p5, fp, sp.</td>
</tr>
<tr>
<td>Stack Pointer</td>
<td>SP; contains the 32-bit address of the last occupied byte location in the stack.</td>
</tr>
<tr>
<td></td>
<td>The stack grows by decrementing the Stack Pointer. A subset of the Pointer</td>
</tr>
<tr>
<td></td>
<td>Registers.</td>
</tr>
<tr>
<td>Frame Pointer</td>
<td>FP; contains the 32-bit address of the previous Frame Pointer in the stack,</td>
</tr>
<tr>
<td></td>
<td>located at the top of a frame. A subset of the Pointer Registers.</td>
</tr>
<tr>
<td>Loop Top</td>
<td>LT0 and LT1; contains 32-bit address of the top of a zero overhead loop.</td>
</tr>
<tr>
<td>Loop Count</td>
<td>LC0 and LC1; contains 32-bit counter of the zero overhead loop executions.</td>
</tr>
<tr>
<td>Loop Bottom</td>
<td>LB0 and LB1; contains 32-bit address of the bottom of a zero overhead loop.</td>
</tr>
</tbody>
</table>
Instruction Opcodes

Table C-1. Registers (Cont’d)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index Register</td>
<td>The set of 32-bit registers I0, I1, I2, I3 that normally contain byte addresses of data structures. Abbreviated I-register or Ireg.</td>
</tr>
<tr>
<td>Modify Registers</td>
<td>The set of 32-bit registers M0, M1, M2, M3 that normally contain offset values that are added or subtracted to one of the Index registers. Abbreviated as Mreg.</td>
</tr>
<tr>
<td>Length Registers</td>
<td>The set of 32-bit registers L0, L1, L2, L3 that normally contain the length (in bytes) of the circular buffer. Abbreviated as Lreg. Clear Lreg to disable circular addressing for the corresponding Ireg. Example: Clear L3 to disable circular addressing for I3.</td>
</tr>
<tr>
<td>Base Registers</td>
<td>The set of 32-bit registers B0, B1, B2, B3 that normally contain the base address (in bytes) of the circular buffer. Abbreviated as Breg.</td>
</tr>
</tbody>
</table>

Functional Units

The architecture includes three processor sections.

Table C-2. Processor Sections

<table>
<thead>
<tr>
<th>Processor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Address Generator (DAG)</td>
<td>Calculates the effective address for indirect and indexed memory accesses. Operates on the Pointer, Index, Modify, Length, and Base Registers. Consists of two units—DAG0 and DAG1.</td>
</tr>
<tr>
<td>Multiply and Accumulate Unit (MAC)</td>
<td>Performs multiply computations and accumulations on data. Operates on the Data Registers and Accumulators. Consists of two units (MAC0 and MAC1), each associated with an Accumulator (A0 and A1, respectively). Each MAC operates in conjunction with an Arithmetic Logical Unit.</td>
</tr>
<tr>
<td>Arithmetic Logical Unit (ALU)</td>
<td>Performs arithmetic computations and binary shifts on data. Operates on the Data Registers and Accumulators. Consists of two units (ALU0 and ALU1), each associated with an Accumulator (A0 and A1, respectively). Each ALU operates in conjunction with a Multiply and Accumulate Unit.</td>
</tr>
</tbody>
</table>
Introduction

Notation Conventions

This appendix uses the following conventions:

- Register names are alphabetic, followed by a number in cases where there are more than one register in a logical group. Thus, examples include ASTAT, FP, R3, and M2.

  Register names are reserved and may not be used as program identifiers.

- Some operations require a register pair. Register pairs are always Data Registers and are denoted using a colon, for example, R3:2. The larger number must is written first. Note: The hardware supports only odd-even pairs, for example, R7:6, R5:4, R3:2, and R1:0.

- Some instructions require a group of adjacent registers. Adjacent registers are denoted by the range enclosed in brackets, e.g., R[7:3]. Again, the larger number appears first.

- Portions of a particular register may be individually specified. This is written with a dot ("." ) following the register name, then a letter denoting the desired portion. For 32-bit registers, ".H" denotes the most significant ("High") portion, ".L" denotes the least significant portion. The subdivisions of the 40-bit registers are described later.

  Register names are reserved and may not be used as program identifiers.
Instruction Opcodes

This appendix uses the following conventions to describe options in the assembler syntax:

- When there is a choice of any one register within a register group, this appendix shows the register set using a single dash to indicate the range of possible register numbers. The register numbers always decrement from high to low. For example, “R7–0” means that any one of the eight Data Registers can be used.

- A range of sequential registers or bits, considered as a group, are denoted using a colon “:”. The register or bit numbers appear highest first, followed by the lowest. For example, the group of Data Registers R3, R2, R1, and R0 are abbreviated R3:0. This nomenclature is similar to that used for valid Data Register pairs, but here, more than a single pair can be represented. Another example is the least significant eight bits of a register are denoted 7:0. In the case of bits, there is no convention to include the register name with the bit range; the register must be clear by context.

- Immediate values are designated as “imm” with the following modifiers:
  - “imm” indicates a signed value; for example, imm7.
  - the “u” prefix indicates an unsigned value; for example, uimm4.
  - the decimal number indicates how many bits the value can include; for example, imm5 is a 5-bit value.
  - any alignment requirements are designated by an optional “m” suffix followed by a number; for example, uimm16m2 is an unsigned, 16-bit integer that must be an even number, and imm7m4 is a signed, 7-bit integer that must be a multiple of 4.
Introduction

- PC-relative, signed values are designated as “pcrel” with the following modifiers:
  - the decimal number indicates how many bits the value can include; for example, pcrel5 is a 5-bit value.
  - any alignment requirements are designated by an optional “m” suffix followed by a number; for example, pcrel13m2 is a 13-bit integer that must be an even number.
- Loop PC-relative, signed values are designated as “lppcrel” with the following modifiers:
  - the decimal number indicates how many bits the value can include; for example, lppcrel5 is a 5-bit value.
  - any alignment requirements are designated by an optional “m” suffix followed by a number; for example, lppcrel13m2 is a 13-bit integer that must be an even number.

Arithmetic Status Flags

The Blackfin architecture includes 12 arithmetic status flags that indicate specific results of a prior operation. These flags reside in the Arithmetic Status (ASTAT) Register. A summary of the flags appears below. All flags are active high. Instructions regarding P-registers, I-registers, L-registers, M-registers, or B-registers do not affect flags.
See Chapter 2, Computational Units, for more details.

Table C-3. Arithmetic Status Flag Summary

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC0</td>
<td>Carry (ALU0)</td>
</tr>
<tr>
<td>AC1</td>
<td>Carry (ALU1)</td>
</tr>
<tr>
<td>AN</td>
<td>Negative</td>
</tr>
<tr>
<td>AQ</td>
<td>Quotient</td>
</tr>
<tr>
<td>AV0</td>
<td>Accumulator 0 Overflow</td>
</tr>
<tr>
<td>AVS0</td>
<td>Accumulator 0 Sticky Overflow; set when AV0 is set, but remains set until explicitly cleared by user code</td>
</tr>
<tr>
<td>AV1</td>
<td>Accumulator 1 Overflow</td>
</tr>
<tr>
<td>AVS1</td>
<td>Accumulator 1 Sticky Overflow; set when AV1 is set, but remains set until explicitly cleared by user code</td>
</tr>
<tr>
<td>AZ</td>
<td>Zero</td>
</tr>
<tr>
<td>CC</td>
<td>Control Code bit; multipurpose flag set, cleared and tested by specific instructions</td>
</tr>
<tr>
<td>V</td>
<td>Overflow for Data Register results</td>
</tr>
<tr>
<td>VS</td>
<td>Sticky Overflow for Data Register results; set when V is set, but remains set until explicitly cleared by user code</td>
</tr>
</tbody>
</table>
Introduction

Core Register Encoding Map

Instruction opcodes can address any core register by Register Group and Register Number using the following encoding.

Table C-4. Core Register Encoding Map

<table>
<thead>
<tr>
<th>REGISTER GROUP</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R0</td>
<td>R1</td>
<td>R2</td>
<td>R3</td>
<td>R4</td>
<td>R5</td>
<td>R6</td>
<td>R7</td>
</tr>
<tr>
<td>1</td>
<td>P0</td>
<td>P1</td>
<td>P2</td>
<td>P3</td>
<td>P4</td>
<td>P5</td>
<td>SP</td>
<td>FP</td>
</tr>
<tr>
<td>2</td>
<td>I0</td>
<td>I1</td>
<td>I2</td>
<td>I3</td>
<td>M0</td>
<td>M1</td>
<td>M2</td>
<td>M3</td>
</tr>
<tr>
<td>3</td>
<td>B0</td>
<td>B1</td>
<td>B2</td>
<td>B3</td>
<td>L0</td>
<td>L1</td>
<td>L2</td>
<td>L3</td>
</tr>
<tr>
<td>4</td>
<td>A0.x</td>
<td>A0.w</td>
<td>A1.x</td>
<td>A1.w</td>
<td>&lt;res.&gt;</td>
<td>&lt;res.&gt;</td>
<td>&lt;res.&gt;</td>
<td>ASTAT</td>
</tr>
<tr>
<td>6</td>
<td>LC0</td>
<td>LT0</td>
<td>LB0</td>
<td>LC1</td>
<td>LT1</td>
<td>LB1</td>
<td>CYCLES</td>
<td>CYCLES2</td>
</tr>
<tr>
<td>7</td>
<td>USP</td>
<td>SEQSTAT</td>
<td>SYSCFG</td>
<td>RETI</td>
<td>RETX</td>
<td>RETN</td>
<td>RETE</td>
<td>EMUDAT</td>
</tr>
</tbody>
</table>

Opcode Representation

The Blackfin architecture accepts 16- and 32-bit opcodes. This document represents the opcodes as hexadecimal values or ranges of values and as binary bit fields.

Some instructions have no variable arguments, and therefore produce only one hex value. The value appears in the “min” Hex Opcode Range column. Instructions that support variable arguments (such as a choice of source or destination registers, optional modes, or constants) span a range of hex values. The minimum and maximum allowable hex values are shown in that case. As explained in “Holes In Opcode Ranges” on page C-10, the instruction may not produce all possible hex values within the range.
A single 16-bit field represents 16-bit opcodes, and two stacked 16-bit fields represent 32-bit opcodes. When stacked, the upper 16 bits show the most significant bits; the lower 16 bits, the least significant bits. See the example table, below.

The hex values of 32-bit instructions are shown stacked in the same order as the bit fields—most significant over least significant.

See “Opcode Representation In Listings, Memory Dumps” on page C-11 for parsing instructions when comparing hex opcodes in debugging software to this reference.

Table C-5. Sample Opcode Representation

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instruction</strong></td>
<td>Single Hex Value</td>
<td>bit bit bit bit bit bit bit bit bit bit bit bit bit bit bit</td>
</tr>
<tr>
<td><strong>Name</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Syntax without variable arguments (16-bit Instruction)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instruction</strong></td>
<td>Min. Value—Max. Value</td>
<td>bit bit bit bit bit bit bit bit bit bit bit bit bit bit bit</td>
</tr>
<tr>
<td><strong>Name</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Syntax with variable arguments (16-bit Instruction)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instruction</strong></td>
<td>Single Hex Value</td>
<td>bit bit bit bit bit Most significant bits bit bit bit bit bit bit bit bit bit bit bit bit</td>
</tr>
<tr>
<td><strong>Name</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Syntax without variable arguments (32-bit Instruction)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instruction</strong></td>
<td>Min. Value—Max. Value</td>
<td>bit bit bit bit bit Most significant bits bit bit bit bit bit bit bit bit bit bit bit</td>
</tr>
<tr>
<td><strong>Name</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Syntax with variable arguments (32-bit Instruction)</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Introduction

Opcode Bit Terminology

The following conventions describe the instruction opcode bit states.

Table C-6.

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Binary zero bit, logical &quot;low&quot;</td>
</tr>
<tr>
<td>1</td>
<td>Binary one bit, logical &quot;high&quot;</td>
</tr>
<tr>
<td>x</td>
<td>&quot;don't care&quot; bit</td>
</tr>
</tbody>
</table>

Undefined Opcodes

Any and all undefined instruction opcode bit patterns are reserved, potentially for future use.

Holes In Opcode Ranges

Holes may exist in the range of operation codes shown for some instructions. For example, one version of the Zero Overhead Loop Setup instruction spans the opcode range E080 0000 through E08F 03FF, as shown in the excerpt, below. However, not all values within that range are valid opcodes; some bit field values are fixed, leaving gaps or “holes” in the sequence of valid opcodes. These undefined opcode holes are reserved for potential future use.
Instruction Opcodes

Table C-7. Sample Opcode Holes Representation

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero Overhead Loop Setup</td>
<td>0xE080 0000—0xE08F 03FF</td>
<td>1 1 1 1 0 0 0 0 1 0 0 0 pcrl5m2 divided by 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 0 0 0 0 pcrl11m2 divided by 2</td>
</tr>
</tbody>
</table>

LOOP loop_name LC0
LOOP_BEGIN loop_name
LOOP_END loop_name
... is mapped to...
LSETUP (pcrl5m2, pcrl11m2) LC0
... where the address of LOOP_BEGIN determines pcrl5m2, and the address of LOOP_END determines pcrl11m2.

Opcode Representation In Listings, Memory Dumps

The Blackfin assembler produces opcodes in little endian format for memory storage. Little endian format is efficient for instruction fetching, but not especially convenient for user readability. Each 16 bits of opcode are stored in memory with the least significant byte first followed by the most significant byte in the next higher address.

32-bit opcodes appear in memory as the most significant 16 bits first, followed by the least significant 16 bits at the next higher address. The reason is that the instruction length is encoded in the most significant 16 bits of the opcode. By storing this information in the lower addresses, the Program Sequencer can determine in one fetch whether it can begin processing the current instruction right away or must wait to fetch the remainder of the instruction first.
For example, a 32-bit opcode 0xFEED FACE is stored in memory locations as shown in Table C-8, below.

Table C-8. Example Memory Contents

<table>
<thead>
<tr>
<th>Relative Byte Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xED</td>
</tr>
<tr>
<td>1</td>
<td>0xFE</td>
</tr>
<tr>
<td>2</td>
<td>0xCE</td>
</tr>
<tr>
<td>3</td>
<td>0xFA</td>
</tr>
</tbody>
</table>

This byte sequence is displayed in ascending address order as...

0xED
0xFE
0xCE
0xFA

... or in 16-bit format as...

0xEDFE
0xCEFA

Or in 32-bit format as...

0xFEED FACE

This reference appendix lists the opcodes in this final format since it matches the opcode bit patterns as recognized by the processor.
# Program Flow Control Instructions

Table C-9. Program Flow Control Instructions (Sheet 1 of 3)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump</td>
<td>0x0050—0x0057</td>
<td>0 0 0 0 0 0 0 0 1 0 0 1</td>
<td>Preg #</td>
</tr>
<tr>
<td>JUMP (Preg)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jump</td>
<td>0x0080—0x0087</td>
<td>0 0 0 0 0 0 0 0 1 0 0 0</td>
<td>Preg #</td>
</tr>
<tr>
<td>JUMP (PC+Preg)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jump</td>
<td>0x2000—0xFFF</td>
<td>0 0 1 0</td>
<td>pcrel13m2 divided by 2</td>
</tr>
<tr>
<td>JUMPS pcrel13m2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jump</td>
<td>0xE200 0000—0xE2FF</td>
<td>1 1 1 0 0 0 0 1 0</td>
<td>Most significant bits of pcrel25m2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Least significant bits of pcrel25m2 divided by 2</td>
</tr>
<tr>
<td>Conditional Jump</td>
<td>0x1800—0x17FF</td>
<td>0 0 0 1 x x</td>
<td>pcrel11m2 divided by 2</td>
</tr>
<tr>
<td>IF CC JUMP pcrel11m2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conditional Jump</td>
<td>0x1C00—0x1FF</td>
<td>0 0 0 1 1 1</td>
<td>pcrel11m2 divided by 2</td>
</tr>
<tr>
<td>IF CC JUMP pcrel11m2 (bp)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conditional Jump</td>
<td>0x1000—0x13FF</td>
<td>0 0 0 1 0 0</td>
<td>pcrel11m2 divided by 2</td>
</tr>
<tr>
<td>IF !CC JUMP pcrel11m2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conditional Jump</td>
<td>0x1400—0x1BFF</td>
<td>0 0 0 1 0 1</td>
<td>pcrel11m2 divided by 2</td>
</tr>
<tr>
<td>IF !CC JUMP pcrel11m2 (bp)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call</td>
<td>0x0060—0x0067</td>
<td>0 0 0 0 0 0 0 0 1 1 0</td>
<td>Preg #</td>
</tr>
<tr>
<td>CALL (Preg)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Program Flow Control Instructions

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call</td>
<td>0x0070—0x0077</td>
<td>00000000000111 Preg #</td>
</tr>
<tr>
<td>CALL (PC+Preg)</td>
<td>0xE300 0000—0xE3FF FFFF</td>
<td>11100011 Most significant bits of pcrel25m2</td>
</tr>
<tr>
<td>CALL pcrel25m2</td>
<td></td>
<td>Least significant bits of pcrel25m2 divided by 2</td>
</tr>
<tr>
<td>Return</td>
<td>0x0010—</td>
<td>0000000000010000</td>
</tr>
<tr>
<td>Return</td>
<td>0x0011—</td>
<td>0000000000010001</td>
</tr>
<tr>
<td>Return</td>
<td>0x0012—</td>
<td>0000000000010010</td>
</tr>
<tr>
<td>Return</td>
<td>0x0013—</td>
<td>0000000000010011</td>
</tr>
<tr>
<td>Return</td>
<td>0x0014—</td>
<td>0000000000010100</td>
</tr>
<tr>
<td>RTS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZERO OVERHEAD LOOP SETUP</td>
<td>0xE080 0000—0xE08F 03FF</td>
<td>111000001000 pcrel5m2 divided by 2</td>
</tr>
<tr>
<td>LOOP loop_name LC0 LOOP_BEGIN loop_name LOOP_END loop_name... is mapped to...LSETUP (pcrel5m2, pcrel11m2) LC0... where the address of LOOP_BEGIN determines pcrel5m2, and the address of LOOP_END determines pcrel11m2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZERO OVERHEAD LOOP SETUP</td>
<td>0xE0A0 0000—0xE0AF F3FF</td>
<td>111000001010 pcrel5m2 divided by 2</td>
</tr>
<tr>
<td>LOOP loop_name LC0 = Preg LOOP_BEGIN loop_name LOOP_END loop_name ... is mapped to...LSETUP (pcrel5m2, pcrel11m2) LC0 = Preg... where the address of LOOP_BEGIN determines pcrel5m2, and the address of LOOP_END determines pcrel11m2.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table C-9. Program Flow Control Instructions (Sheet 3 of 3)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero Overhead Loop Setup</td>
<td>0xE0E0 0000—</td>
<td>1 1 1 1 0 0 0 0 1 1 1 0 pcrel5m2 divided by 2</td>
</tr>
<tr>
<td></td>
<td>0xE0AF F3FF</td>
<td>Preg # x x pcrel11m2 divided by 2</td>
</tr>
<tr>
<td>Zero Overhead Loop Setup</td>
<td>0xE090 0000—</td>
<td>1 1 1 0 0 0 0 0 1 0 0 1 pcrel5m2 divided by 2</td>
</tr>
<tr>
<td></td>
<td>0xE09F 03FF</td>
<td>0 0 0 0 x x pcrel11m2 divided by 2</td>
</tr>
<tr>
<td>Zero Overhead Loop Setup</td>
<td>0xE0B0 0000—</td>
<td>1 1 1 0 0 0 0 1 0 1 1 pcrel5m2 divided by 2</td>
</tr>
<tr>
<td></td>
<td>0xE0BF F3FF</td>
<td>Preg # x x pcrel11m2 divided by 2</td>
</tr>
<tr>
<td>Zero Overhead Loop Setup</td>
<td>0xE0F0 0000—</td>
<td>1 1 1 0 0 0 0 1 1 1 1 pcrel5m2 divided by 2</td>
</tr>
<tr>
<td></td>
<td>0xE0FF F3FF</td>
<td>Preg # x x pcrel11m2 divided by 2</td>
</tr>
</tbody>
</table>

LOOP loop_name LC0 = Preg >> 1 LOOP_BEGIN loop_name LOOP_END loop_name ... is mapped to... LSETUP (pcrel5m2, pcrel11m2) LC0 = Preg >> 1 ... where the address of LOOP_BEGIN determines pcrel5m2, and the address of LOOP_END determines pcrel11m2.

LOOP loop_name LC1 LOOP_BEGIN loop_name LOOP_END loop_name ... is mapped to... LSETUP (pcrel5m2, pcrel11m2) LC1 = Preg ... where the address of LOOP_BEGIN determines pcrel5m2, and the address of LOOP_END determines pcrel11m2.

LOOP loop_name LC1 = Preg LOOP_BEGIN loop_name LOOP_END loop_name ... is mapped to... LSETUP (pcrel5m2, pcrel11m2) LC1 = Preg ... where the address of LOOP_BEGIN determines pcrel5m2, and the address of LOOP_END determines pcrel11m2.

LOOP loop_name LC1 = Preg >> 1 LOOP_BEGIN loop_name LOOP_END loop_name ... is mapped to... LSETUP (pcrel5m2, pcrel11m2) LC1 = Preg >> 1 ... where the address of LOOP_BEGIN determines pcrel5m2, and the address of LOOP_END determines pcrel11m2.
## Load / Store Instructions

### Table C-10. Load / Store Instructions (Sheet 1 of 12)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
<th>Reg grp. #</th>
<th>Reg #</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Load Immediate</strong></td>
<td>0xE100 0000—0xE11F FFFF</td>
<td>1 1 1 0 0 0 1 0 0 0</td>
<td>uimm16</td>
<td></td>
</tr>
<tr>
<td>reg_lo = uimm16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Load Immediate</strong></td>
<td>0xE140 0000—0xE15F FFFF</td>
<td>1 1 1 0 0 0 0 1 1 1 0</td>
<td>uimm16</td>
<td></td>
</tr>
<tr>
<td>reg_hi = uimm16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Load Immediate</strong></td>
<td>0xE180 0000—0xE19F FFFF</td>
<td>1 1 1 0 0 0 1 1 0 0 0</td>
<td>uimm16</td>
<td></td>
</tr>
<tr>
<td>reg = uimm16 (Z)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Load Immediate</strong></td>
<td>0xC408 003F 1100010 xxx001000</td>
<td>1 1 0 0 0 1 0 x x 0 0 1 0 0 0</td>
<td>0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1</td>
<td></td>
</tr>
<tr>
<td>A0 = 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Load Immediate</strong></td>
<td>0xC408 403F 1100010000001000</td>
<td>1 1 0 0 0 1 0 0 0 0 0 1 0 0 0</td>
<td>0 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1</td>
<td></td>
</tr>
<tr>
<td>A1 = 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Load Immediate</strong></td>
<td>0xC408 803F 1100010 xxx001000</td>
<td>1 1 0 0 0 1 0 x x 0 0 1 0 0 0</td>
<td>1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1</td>
<td></td>
</tr>
<tr>
<td>A1 = A0 = 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Load Immediate</strong></td>
<td>0x6000—0x63FF 011000imm7 Dreg #</td>
<td>0 1 1 0 0 0 immm7</td>
<td>Dreg #</td>
<td></td>
</tr>
<tr>
<td>Dreg = immm7 (X)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Load Immediate</strong></td>
<td>0x6800—0x6BFF 011010imm7 Preg #</td>
<td>0 1 1 0 1 0 immm7</td>
<td>Preg #</td>
<td></td>
</tr>
<tr>
<td>Preg = immm7 (X)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Instruction Opcodes

## Table C-10. Load / Store Instructions (Sheet 2 of 12)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Immediate</td>
<td>0xE120 0000— 0xE13F FFFF</td>
<td>1 1 1 0 0 0 1 0 1 0 1 0 1 0 0 0</td>
<td>Reg grp. #</td>
</tr>
<tr>
<td>reg = imm16 (X)</td>
<td></td>
<td>imm16</td>
<td></td>
</tr>
<tr>
<td>Load Pointer Register</td>
<td>0x9140—</td>
<td>1 0 0 1 0 0 0 1 0 1 0 1 0 0 1 0 1</td>
<td>Source Preg #</td>
</tr>
<tr>
<td>Preg = [ Preg ]</td>
<td>0x917F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Pointer Register</td>
<td>0x9040—</td>
<td>1 0 0 1 0 0 0 0 0 0 1 0 1 0 1 0 1</td>
<td>Source Preg #</td>
</tr>
<tr>
<td>Preg = [ Preg ++ ]</td>
<td>0x907F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Pointer Register</td>
<td>0x90C0—</td>
<td>1 0 0 1 0 0 0 0 0 0 1 0 1 1 0 1 1</td>
<td>Source Preg #</td>
</tr>
<tr>
<td>Preg = [ Preg -- ]</td>
<td>0x90FF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Pointer Register</td>
<td>0xACC0—</td>
<td>1 0 1 0 0 0 0 0 0 1 1 0 1 0 1 1 1</td>
<td>Source Preg #</td>
</tr>
<tr>
<td>Preg = [ Preg + uimm6m4 ]</td>
<td>0xAFFFF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Pointer Register</td>
<td>0xE500 0000— 0xE53F 7FFF</td>
<td>1 1 1 0 0 1 0 1 0 1 0 1 0 0 0 0</td>
<td>Source Preg #</td>
</tr>
<tr>
<td>Preg = [ Preg + uimm17m4 ]</td>
<td>0xE53F FFFF</td>
<td>uimm17m4 divided by 4</td>
<td></td>
</tr>
<tr>
<td>Load Pointer Register</td>
<td>0xE500 8000— 0xE53F FFFF</td>
<td>1 1 1 0 0 1 0 1 0 1 0 1 0 0 0 0</td>
<td>Source Preg #</td>
</tr>
<tr>
<td>Preg = [ Preg – uimm17m4 ]</td>
<td>0xE53F FFFF</td>
<td>uimm17m4 divided by 4</td>
<td></td>
</tr>
<tr>
<td>Load Pointer Register</td>
<td>0xB808—</td>
<td>1 0 1 1 1 0 0 1 0 1 1 0 0 0 1 0 0</td>
<td>uimm7m4 divided by 4</td>
</tr>
<tr>
<td>Preg = [ FP – uimm7m4 ]</td>
<td>0xB9FF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Data Register</td>
<td>0x9100—</td>
<td>1 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0</td>
<td>Preg #</td>
</tr>
<tr>
<td>Dreg = [ Preg ]</td>
<td>0x913F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Load / Store Instructions

## Table C-10. Load / Store Instructions (Sheet 3 of 12)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Data Register</td>
<td>0x9000—0x903F</td>
<td>1 0 0 1 0 1 0 0 0 0 0 0 Preg # Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg = [ Preg ++ ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Data Register</td>
<td>0x9080—0x90BF</td>
<td>1 0 0 1 0 0 0 0 1 0 0 Preg # Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg = [ Preg -- ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Data Register</td>
<td>0xA000—0xA3FF</td>
<td>1 0 1 0 0 0 uimm6m4 divided by 4 Preg # Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg = [ Preg + uimm6m4 ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Data Register</td>
<td>0xE400 0000—0xE4EF 7FFF</td>
<td>1 1 1 0 0 1 0 0 0 0 0 Preg # Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg = [ Preg + uimm17m4 ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Data Register</td>
<td>0xE400 8000—0xE43F FFFF</td>
<td>1 1 1 0 0 0 0 0 0 0 0 Preg # Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg = [ Preg -- uimm17m4 ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Data Register</td>
<td>0x8000—0x81FF</td>
<td>1 0 0 0 0 0 0 0 0 0 0 0 Dest. Dreg # Index Preg # Pointer Preg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg = [ Preg ++ Preg ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOTE: Pointer Preg number cannot be the same as Index Preg number. If so, this opcode represents a non-post-modify instruction version.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Data Register</td>
<td>0xB800—0xB9F7</td>
<td>1 0 0 1 1 1 0 uimm7m4 divided by 4 Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg = [ FP – uimm7m4 ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Data Register</td>
<td>0x9D00—0x9D1F</td>
<td>1 0 0 0 1 1 0 0 0 0 0 Ireg # Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg = [ Ireg ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Data Register</td>
<td>0x9C00—0x9C1F</td>
<td>1 0 0 1 1 1 0 0 0 0 0 Ireg # Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg = [ Ireg ++ ]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Instruction Opcodes

#### Table C-10. Load / Store Instructions (Sheet 4 of 12)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Data Register</td>
<td>0x9C80—0x9C9F</td>
<td>1 0 0 1 1 1 0 0 1 0 0</td>
</tr>
<tr>
<td>Dreg = [ Ireg -- ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Data Register</td>
<td>0x9D80—0x9DF</td>
<td>1 0 0 1 1 1 0 1 1</td>
</tr>
<tr>
<td>Dreg = [ Ireg ++ Mreg ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Half Word, Zero Extended</td>
<td>0x9500—0x953F</td>
<td>1 0 0 1 0 1 0 1 1</td>
</tr>
<tr>
<td>Dreg = W [ Preg ] (Z)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Half Word, Zero Extended</td>
<td>0x9400—0x943F</td>
<td>1 0 0 1 0 1 0 0 0</td>
</tr>
<tr>
<td>Dreg = W [ Preg ++ ] (Z)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Half Word, Zero Extended</td>
<td>0x9480—0x94BF</td>
<td>1 0 0 1 1 0 1 0 0</td>
</tr>
<tr>
<td>Dreg = W [ Preg -- ] (Z)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Half Word, Zero Extended</td>
<td>0xA400—0xA47F</td>
<td>1 0 1 0 0 1 0 1 1</td>
</tr>
<tr>
<td>Dreg = W [ Preg + uimm5m2 ] (Z)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Half Word, Zero Extended</td>
<td>0xE440 0000—0xE47F 8FFF</td>
<td>1 1 1 0 0 1 0 1 0</td>
</tr>
<tr>
<td>Dreg = W [ Preg + uimm16m2 ] (Z)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Half Word, Zero Extended</td>
<td>0xE440 8000—0xE47F FFFF</td>
<td>1 1 1 0 0 1 0 0 0</td>
</tr>
<tr>
<td>Dreg = W [ Preg -- uimm16m2 ] (Z)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Half Word, Zero Extended</td>
<td>0x8601—0x87FE</td>
<td>1 0 0 0 1 0 1 1</td>
</tr>
<tr>
<td>Dreg = W [ Preg ++ Preg ] (Z)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: Pointer Preg number cannot be the same as Index Preg number. If so, this opcode represents a non-post-modify instruction version.
## Load / Store Instructions

### Table C-10. Load / Store Instructions (Sheet 5 of 12)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Half Word, Sign Extended</td>
<td>0x9540—0x957F</td>
<td>0b10010101 Preg # Dreg #</td>
</tr>
<tr>
<td>Dreg = W [ Preg ] (X)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Half Word, Sign Extended</td>
<td>0x9440—0x947F</td>
<td>0b10010100 Preg # Dreg #</td>
</tr>
<tr>
<td>Dreg = W [ Preg ++ ] (X)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Half Word, Sign Extended</td>
<td>0x94C0—0x94FF</td>
<td>0b10010100 Preg # Dreg #</td>
</tr>
<tr>
<td>Dreg = W [ Preg — ] (X)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Half Word, Sign Extended</td>
<td>0xA800—0xABFF</td>
<td>0b101010 uimm5m2 divided by 2 Preg # Dreg #</td>
</tr>
<tr>
<td>Dreg = W [ Preg + uimm5m2 ] (X)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Half Word, Sign Extended</td>
<td>0xE540 0000—0xE57F 8FFF</td>
<td>0b11100101 uimm16m2 divided by 2 Preg # Dreg #</td>
</tr>
<tr>
<td>Dreg = W [ Preg + uimm16m2 ] (X)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Half Word, Sign Extended</td>
<td>0xE540 8000—0xE57F FFFF</td>
<td>0b11100100 uimm16m2 divided by 2 Preg # Dreg #</td>
</tr>
<tr>
<td>Dreg = W [ Preg — uimm16m2 ] (X)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Half Word, Sign Extended</td>
<td>0x8E00—0x8FFF</td>
<td>0b1000111 Dest. Dreg # Index Preg # Pointer Preg #</td>
</tr>
<tr>
<td>Dreg = W [ Preg ++ Preg ] (X)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: Pointer Preg number cannot be the same as Index Preg number. If so, this opcode represents a non-post-modify instruction version.

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load High Data Register Half</td>
<td>0x9D40—0x9D5F</td>
<td>0b10011101 Ireg # Dreg #</td>
</tr>
<tr>
<td>Dreg_hi = W [ Ireg ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load High Data Register Half</td>
<td>0x9C40—0x9C5F</td>
<td>0b10011100 Ireg # Dreg #</td>
</tr>
<tr>
<td>Dreg_hi = W [ Ireg ++ ]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Instruction Opcodes

#### Table C-10. Load / Store Instructions (Sheet 6 of 12)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load High Data Register Half</td>
<td>0x9CC0—0x9CDF</td>
<td><img src="#" alt="Bin" /></td>
</tr>
<tr>
<td>Dreg_his = W [ Ireg — ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load High Data Register Half</td>
<td>0x8400—0x85FF</td>
<td><img src="#" alt="Bin" /></td>
</tr>
<tr>
<td>Dreg_his = W [ Preg ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOTE: The two least significant bit fields must refer to the same Preg number. Otherwise, this opcode represents a post-modify version of this instruction.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load High Data Register Half</td>
<td>0x8401—0x85FE</td>
<td><img src="#" alt="Bin" /></td>
</tr>
<tr>
<td>Dreg_his = W [ Preg ++ Preg ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOTE: Pointer Preg number cannot be the same as Index Preg number. If so, this opcode represents a non-post-modify instruction version.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Low Data Register Half</td>
<td>0x9D20—0x9D3F</td>
<td><img src="#" alt="Bin" /></td>
</tr>
<tr>
<td>Dreg _lo = W [ Ireg ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Low Data Register Half</td>
<td>0x9C20—0x9C3F</td>
<td><img src="#" alt="Bin" /></td>
</tr>
<tr>
<td>Dreg _lo = W [ Ireg ++ ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Low Data Register Half</td>
<td>0x9CA0—0x9CBF</td>
<td><img src="#" alt="Bin" /></td>
</tr>
<tr>
<td>Dreg _lo = W [ Ireg — — ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Low Data Register Half</td>
<td>0x8200—0x83FF</td>
<td><img src="#" alt="Bin" /></td>
</tr>
<tr>
<td>Dreg _lo = W [ Preg ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOTE: Both Pointer Preg # fields must refer to the same Preg number. Otherwise, this opcode represents a post-modify version of this instruction.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Low Data Register Half</td>
<td>0x8201—0x83FE</td>
<td><img src="#" alt="Bin" /></td>
</tr>
<tr>
<td>Dreg _lo = W [ Preg ++ Preg ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOTE: Pointer Preg number cannot be the same as Index Preg number. If so, this opcode represents a non-post-modify instruction version.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Load / Store Instructions

### Table C-10. Load / Store Instructions (Sheet 7 of 12)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Byte, Zero Extended</td>
<td>0x9900—0x993F</td>
<td>1 0 0 1 1 0 0 1 0 0</td>
</tr>
<tr>
<td>Dreg = B [ Preg ] (Z)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Byte, Zero Extended</td>
<td>0x9800—0x983F</td>
<td>1 0 0 1 1 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg = B [ Preg ++ ] (Z)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Byte, Zero Extended</td>
<td>0x9880—0x98BF</td>
<td>1 0 0 1 1 0 0 0 0 1 0</td>
</tr>
<tr>
<td>Dreg = B [ Preg — ] (Z)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Byte, Zero Extended</td>
<td>0xE480 0000—0xE4BF 7FFF</td>
<td>1 1 1 0 1 0 0 0 0 1 0</td>
</tr>
<tr>
<td>Dreg = B [ Preg + uimm15 ] (Z)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Byte, Zero Extended</td>
<td>0xE480 8000—0xE4BF FFFF</td>
<td>1 1 1 0 1 0 0 0 0 1 0</td>
</tr>
<tr>
<td>Dreg = B [ Preg — uimm15] (Z)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Byte, Sign Extended</td>
<td>0x9940—0x997F</td>
<td>1 0 0 1 1 0 0 1 0 0</td>
</tr>
<tr>
<td>Dreg = B [ Preg ] (X)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Byte, Sign Extended</td>
<td>0x9840—0x987F</td>
<td>1 0 0 1 1 0 0 0 0 1 0</td>
</tr>
<tr>
<td>Dreg = B [ Preg ++ ] (X)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Byte, Sign Extended</td>
<td>0x98C0—0x98FF</td>
<td>1 0 0 1 1 0 0 0 0 1 1</td>
</tr>
<tr>
<td>Dreg = B [ Preg — ] (X)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Byte, Sign Extended</td>
<td>0xE580 0000—0xE5BF 7FFF</td>
<td>1 1 1 0 1 0 0 0 1 1 0</td>
</tr>
<tr>
<td>Dreg = B [ Preg + uimm15 ] (X)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Instruction Opcodes

Table C-10. Load / Store Instructions (Sheet 8 of 12)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Byte, Sign Extended</td>
<td>0xE580 8000—0xE5BF FFFF</td>
<td>1 1 1 1 0 0 1 0 1 1 0 Preg # Dreg #</td>
</tr>
</tbody>
</table>

\[ \text{Dreg} = B \text{ [ Preg – uimm15]} \text{ (X)} \]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store Pointer Register</td>
<td>0x9340—0x937F</td>
<td>1 0 0 1 0 0 1 0 1 0 1 Dest. Pointer Preg # Source Preg #</td>
</tr>
</tbody>
</table>

\[ \text{[ Preg ]} = \text{Preg} \]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store Pointer Register</td>
<td>0x9240—0x927F</td>
<td>1 0 0 1 0 0 1 0 0 0 1 Dest. Pointer Preg # Source Preg #</td>
</tr>
</tbody>
</table>

\[ \text{[ Preg ++ ]} = \text{Preg} \]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store Pointer Register</td>
<td>0x92C0—0x92FF</td>
<td>1 0 0 1 0 0 1 0 0 1 1 Dest. Pointer Preg # Source Preg #</td>
</tr>
</tbody>
</table>

\[ \text{[ Preg – – ]} = \text{Preg} \]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store Pointer Register</td>
<td>0xBC00—0xBFFF</td>
<td>1 0 1 1 1 1 uimm6m4 divided by 4 Dest. Pointer Preg # Source Preg #</td>
</tr>
</tbody>
</table>

\[ \text{[ Preg + uimm6m4 ]} = \text{Preg} \]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store Pointer Register</td>
<td>0xE700 0000—0xE7EF FFFF</td>
<td>1 1 1 1 0 0 1 1 1 0 0 Dest. Pointer Preg # Source Preg #</td>
</tr>
</tbody>
</table>

\[ \text{uimm17m4 divided by 4} \]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store Pointer Register</td>
<td>0xE700 8000—0xE73F FFFF</td>
<td>1 1 1 0 0 1 1 1 0 0 Dest. Pointer Preg # Source Preg #</td>
</tr>
</tbody>
</table>

\[ \text{uimm17m4 divided by 4} \]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store Pointer Register</td>
<td>0xE700 8000—0xE73F FFFF</td>
<td>1 1 1 0 0 1 1 1 0 0 Dest. Pointer Preg # Source Preg #</td>
</tr>
</tbody>
</table>

\[ \text{uimm17m4 divided by 4} \]
### Table C-10. Load / Store Instructions (Sheet 9 of 12)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Store Pointer Register</strong></td>
<td>0xBA08—0xBBFF</td>
<td>1 0 1 1 0 1 1 0 1 0imm7m4 divided by 4 Preg #</td>
</tr>
<tr>
<td>![FP – uimm7m4 ] = Preg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Store Data Register</strong></td>
<td>0x9300—0x933F</td>
<td>1 0 0 1 0 0 1 1 0 0 Preg # Dreg #</td>
</tr>
<tr>
<td>![Preg ] = Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Store Data Register</strong></td>
<td>0x9200—0x923F</td>
<td>1 0 0 1 0 0 1 0 0 0 Preg # Dreg #</td>
</tr>
<tr>
<td>![Preg ++ ] = Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Store Data Register</strong></td>
<td>0x9280—0x92BF</td>
<td>1 0 0 1 0 0 1 0 1 0 Preg # Dreg #</td>
</tr>
<tr>
<td>![Preg – – ] = Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Store Data Register</strong></td>
<td>0xB000—0xB3FF</td>
<td>1 0 1 1 0 0 Preg # Dreg #</td>
</tr>
<tr>
<td>![Preg + uimm6m4 ] = Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Store Data Register</strong></td>
<td>0xE600 0000—0xE63F 7FFF</td>
<td>1 1 1 1 0 1 1 0 0 0 Preg # Dreg #</td>
</tr>
<tr>
<td>![Preg + uimm17m4 ] = Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Store Data Register</strong></td>
<td>0xE600 8000—0xE63F FFFF</td>
<td>1 1 1 1 0 1 1 0 0 0 Preg # Dreg #</td>
</tr>
<tr>
<td>![Preg – uimm17m4 ] = Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Store Data Register</strong></td>
<td>0x8800—0x89FF</td>
<td>1 0 0 0 1 0 1 0 Preg # Dreg #</td>
</tr>
<tr>
<td>![Preg ++ Preg ] = Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Store Data Register</strong></td>
<td>0xBA00—0xBBF7</td>
<td>1 0 1 1 0 1 0imm7m4 divided by 4 Dreg #</td>
</tr>
<tr>
<td>![FP – uimm7m4 ] = Dreg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:** Pointer Preg number cannot be the same as Index Preg number. If so, this opcode represents a non-post-modify instruction version.
### Instruction Opcodes

Table C-10. Load / Store Instructions (Sheet 10 of 12)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Store Data Register</strong></td>
<td>0x9F00—0x9F1F</td>
<td>10</td>
</tr>
<tr>
<td>[Ireg] = Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Store Data Register</strong></td>
<td>0x9E00—0x9E1F</td>
<td>10</td>
</tr>
<tr>
<td>[Ireg ++] = Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Store Data Register</strong></td>
<td>0x9E80—0x9E9F</td>
<td>10</td>
</tr>
<tr>
<td>[Ireg --] = Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Store Data Register</strong></td>
<td>0x9F80—0x9FFF</td>
<td>1M</td>
</tr>
<tr>
<td>[Ireg ++ Mreg] = Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Store High Data Register Half</strong></td>
<td>0x9F40—0x9F5F</td>
<td>110</td>
</tr>
<tr>
<td>W[Ireg] = Dreg_hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Store High Data Register Half</strong></td>
<td>0x9E40—0x9E5F</td>
<td>110</td>
</tr>
<tr>
<td>W[Ireg ++] = Dreg_hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Store High Data Register Half</strong></td>
<td>0xEC0—0x9EDF</td>
<td>110</td>
</tr>
<tr>
<td>W[Ireg --] = Dreg_hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Store High Data Register Half</strong></td>
<td>0x8C00—0x8DFF</td>
<td>110</td>
</tr>
<tr>
<td>W[Preg] = Dreg_hi</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: Both Pointer Preg # fields must refer to the same Preg number. Otherwise, this opcode represents a post-modify version of this instruction.

| **Store High Data Register Half** | 0x8C01—0x8DFE | 110 |
| W[Preg ++ Preg] = Dreg_hi       |                      |     |

NOTE: Pointer Preg number cannot be the same as Index Preg number. If so, this opcode represents a non-post-modify instruction version.
Table C-10. Load / Store Instructions (Sheet 11 of 12)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store Low Data Register Half</td>
<td>0x9F20—0x9F3F</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Ireg # Dreg #</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W [ Ireg ] = Dreg_lo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>Store Low Data Register Half</td>
<td>0x9E20—0x9E3F</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Ireg # Dreg #</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W [ Ireg ++ ] = Dreg_lo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>Store Low Data Register Half</td>
<td>0x9EA0—0x9EBF</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Ireg # Dreg #</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W [ Ireg -- ] = Dreg_lo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>Store Low Data Register Half</td>
<td>0x8A00—0x8BFF</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Source Dreg # Pointer Preg # Pointer Preg #</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W [ Preg ] = Dreg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>NOTE: Both Pointer Preg # fields must refer to the same Preg number. Otherwise, this opcode represents a post-modify version of this instruction.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>Store Low Data Register Half</td>
<td>0x9700—0x973F</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Preg # Dreg #</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W [ Preg ] = Dreg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>Store Low Data Register Half</td>
<td>0x9600—0x963F</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Preg # Dreg #</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W [ Preg ++ ] = Dreg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>Store Low Data Register Half</td>
<td>0x9680—0x96BF</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Preg # Dreg #</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W [ Preg -- ] = Dreg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>Store Low Data Register Half</td>
<td>0xB400—0xB7FF</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>uimm5m2 divided by 2</td>
<td>Preg # Dreg #</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W [ Preg + uimm5m2 ] = Dreg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>Store Low Data Register Half</td>
<td>0xE640 0000—0xE67F 7FFF</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Preg # Dreg #</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W [ Preg + uimm16m2 ] = Dreg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>

C-26 ADSP-BF53x/BF56x Blackfin Processor Programming Reference
### Instruction Opcodes

#### Table C-10. Load / Store Instructions (Sheet 12 of 12)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Store Low Data Register Half</strong></td>
<td>0xE640 8000—0xE67F FFFF</td>
<td>uimm16m2 divided by 2</td>
</tr>
<tr>
<td>W [ Preg – uimm16m2 ] = Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Store Low Data Register Half</strong></td>
<td>0x8A01—0x8BFE</td>
<td>Source Dreg #</td>
</tr>
<tr>
<td>W [ Preg ++ Preg ] = Dreg_lo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOTE: Pointer Preg number cannot be the same as Index Preg number. If so, this opcode represents a non-post-modify instruction version.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Store Byte</strong></td>
<td>0x9B00—0x9B3F</td>
<td></td>
</tr>
<tr>
<td>B [ Preg ] = Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Store Byte</strong></td>
<td>0x9A00—0x9A3F</td>
<td></td>
</tr>
<tr>
<td>B [ Preg ++ ] = Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Store Byte</strong></td>
<td>0x9A80—0x9ABF</td>
<td></td>
</tr>
<tr>
<td><strong>Store Byte</strong></td>
<td>0xE680 0000—0xE6BF 7FFF</td>
<td>uimm15</td>
</tr>
<tr>
<td><strong>Store Byte</strong></td>
<td>0xE680 8000—0xE6BF FFFF</td>
<td>uimm15</td>
</tr>
</tbody>
</table>
### Move Instructions

Table C-11. Move Instructions (Sheet 1 of 9)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Move Register</strong></td>
<td>0x3000—0x3FFF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x3000—0x3FFF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x3000—0x3FFF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x3000—0x3FFF</td>
<td></td>
</tr>
<tr>
<td>genreg = genreg</td>
<td>genreg = dagreg</td>
<td></td>
</tr>
<tr>
<td>dagreg = genreg</td>
<td>dagreg = dagreg</td>
<td></td>
</tr>
<tr>
<td>genreg = USP</td>
<td>USP = genreg</td>
<td></td>
</tr>
<tr>
<td>Dreg = sysreg</td>
<td>sysreg = Dreg</td>
<td></td>
</tr>
<tr>
<td>sysreg = Preg</td>
<td>sysreg = USP</td>
<td></td>
</tr>
<tr>
<td>A0 = A1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1 = A0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A0 = Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1 = Dreg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:**
- genreg represents general-purpose registers.
- dagreg is the Dual Accumulator Group.
- USP is the User Stack Pointer.
- SysReg represents system registers.
- Source# and Dest# represent source and destination register numbers.

C-28 ADSP-BF53x/BF56x Blackfin Processor Programming Reference
### Instruction Opcodes

**Table C-11. Move Instructions (Sheet 2 of 9)**

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Move Register</strong></td>
<td>0xC00B 3800—0xC00B 39C0</td>
<td>1110000000001011 0011100 Dreg_even # 000000</td>
</tr>
<tr>
<td><strong>Move Register</strong></td>
<td>0xC08B 3800—0xC08B 39C0</td>
<td>1110000010001011 0011100 Dreg_even # 000000</td>
</tr>
<tr>
<td><strong>Move Register</strong></td>
<td>0xC12B 3800—0xC12B 39C0</td>
<td>1110000100101011 0011100 Dreg_even # 000000</td>
</tr>
<tr>
<td><strong>Move Register</strong></td>
<td>0xC00F 1800—0xC00F 19C0</td>
<td>1110000000001111 0001100 Dreg_odd # of the pair containing Dreg_odd</td>
</tr>
<tr>
<td><strong>Move Register</strong></td>
<td>0xC08F 1800—0xC08F 19C0</td>
<td>1110000000001111 0001100 Dreg_odd # of the pair containing Dreg_odd</td>
</tr>
</tbody>
</table>
### Move Instructions

Table C-11. Move Instructions (Sheet 3 of 9)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move Register</td>
<td>0xC12F 1800—- 0xC12F 19C0</td>
<td>1 1 0 0 0 0 0 1 0 0 1 0 1 1 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 0 1 1 0 0 Dreg_even # of the pair containing Dreg_odd</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_odd = A1 (ISS2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move Register</td>
<td>0xC00F 3800—- 0xC00F 39C0</td>
<td>1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1 1 1 0 0 Dreg_even # of the register pair</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_even = A0, Dreg_odd = A1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move Register</td>
<td>0xC08F 3800—- 0xC08F 39C0</td>
<td>1 1 0 0 0 0 0 0 1 0 0 0 1 1 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1 1 1 0 0 Dreg_even # of the register pair</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_even = A0, Dreg_odd = A1 (FU)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move Register</td>
<td>0xC12F 3800—- 0xC12F 39C0</td>
<td>1 1 0 0 0 0 0 1 0 0 1 0 1 1 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1 1 1 0 0 Dreg_even # of the register pair</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_even = A0, Dreg_odd = A1 (ISS2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

C-30 ADSP-BF53x/BF56x Blackfin Processor Programming Reference
# Instruction Opcodes

Table C-11. Move Instructions (Sheet 4 of 9)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Move Conditional</strong></td>
<td>0x0700—0x073F</td>
<td>0000011100 Dreg #</td>
</tr>
<tr>
<td>IF CC Dreg=Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Move Conditional</strong></td>
<td>0x0740—0x077F</td>
<td>0000011101 Dreg #</td>
</tr>
<tr>
<td>IF CC Dreg=Preg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Move Conditional</strong></td>
<td>0x0780—0x07BF</td>
<td>0000011110 Preg #</td>
</tr>
<tr>
<td>IF CC Preg=Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Move Conditional</strong></td>
<td>0x07C0—0x07FF</td>
<td>0000011111 Preg #</td>
</tr>
<tr>
<td>IF !CC Preg=Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Move Conditional</strong></td>
<td>0x0600—0x063F</td>
<td>0000011000 Dreg #</td>
</tr>
<tr>
<td>IF !CC Dreg=Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Move Conditional</strong></td>
<td>0x0640—0x067F</td>
<td>0000011001 Dreg #</td>
</tr>
<tr>
<td>IF !CC Dreg=Preg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Move Conditional</strong></td>
<td>0x0680—0x06BF</td>
<td>0000011010 Preg #</td>
</tr>
<tr>
<td>IF !CC Preg=Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Move Conditional</strong></td>
<td>0x06C0—0x06FF</td>
<td>0000011011 Preg #</td>
</tr>
<tr>
<td>IF !CC Preg=Preg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Move Half to Full Word, Zero Extended</strong></td>
<td>0x42C0—0x42FF</td>
<td>0100001011 Dreg #</td>
</tr>
<tr>
<td>Dreg = Dreg_lo (Z)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Move Half to Full Word, Sign Extended</strong></td>
<td>0x4280—0x42BF</td>
<td>0100001010 Dreg #</td>
</tr>
<tr>
<td>Dreg = Dreg_lo (X)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Move Instructions

Table C-11. Move Instructions (Sheet 5 of 9)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Move Register Half</strong></td>
<td>0xC409 4000—0xC409 4038</td>
<td>1100010xxx001001</td>
</tr>
<tr>
<td>Dreg #</td>
<td>000</td>
<td></td>
</tr>
<tr>
<td>Source Dreg #</td>
<td>000</td>
<td>00</td>
</tr>
<tr>
<td>A0.X = Dreg_lo</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Move Register Half</strong></td>
<td>0xC409 C000—0xC409 C038</td>
<td>1100010xxx001001</td>
</tr>
<tr>
<td>Dreg #</td>
<td>000</td>
<td>00</td>
</tr>
<tr>
<td>Source Dreg #</td>
<td>000</td>
<td>00</td>
</tr>
<tr>
<td>A1.X = Dreg_lo</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Move Register Half</strong></td>
<td>0xC40A 0000—0xC40A 0E00</td>
<td>1100010xxx001010</td>
</tr>
<tr>
<td>Dreg #</td>
<td>000111111</td>
<td></td>
</tr>
<tr>
<td>Dest Dreg #</td>
<td>000011111111</td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = A0.X</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Move Register Half</strong></td>
<td>0xC40A 4000—0xC40A 4E00</td>
<td>1100010xxx001010</td>
</tr>
<tr>
<td>Dreg #</td>
<td>000111111</td>
<td></td>
</tr>
<tr>
<td>Dest Dreg #</td>
<td>000011111111</td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = A1.X</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Move Register Half</strong></td>
<td>0xC409 0000—0xC409 0038</td>
<td>1100010xxx001010</td>
</tr>
<tr>
<td>Dreg #</td>
<td>000111111</td>
<td></td>
</tr>
<tr>
<td>Source 0 Dreg #</td>
<td>000000000000</td>
<td>00</td>
</tr>
<tr>
<td>A0.L = Dreg_lo</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Move Register Half</strong></td>
<td>0xC409 8000—0xC409 8038</td>
<td>1100010xxx001010</td>
</tr>
<tr>
<td>Dreg #</td>
<td>000111111</td>
<td></td>
</tr>
<tr>
<td>Source 0 Dreg #</td>
<td>000000000000</td>
<td>00</td>
</tr>
<tr>
<td>A1.L = Dreg_lo</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table C-11. Move Instructions (Sheet 6 of 9)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Move Register Half</strong></td>
<td>0xC429 0000—0xC429 0038</td>
<td>11100001001x x 10101001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00000000000 Source 0</td>
</tr>
<tr>
<td><strong>A0.H = Dreg_hi</strong></td>
<td>0xC429 8000—0xC429 8038</td>
<td>11100001001x x 10101001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10000000000 Source 0</td>
</tr>
<tr>
<td><strong>A1.H = Dreg_hi</strong></td>
<td>0xC003 3800—0xC003 39C0</td>
<td>111000000001110011000011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000111100 Dreg #</td>
</tr>
<tr>
<td><strong>Dreg_lo = A0</strong></td>
<td>0xC083 3800—0xC083 39C0</td>
<td>11100000000100000000111</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000111100 Dreg #</td>
</tr>
<tr>
<td><strong>Dreg_lo = A0 (FU)</strong></td>
<td>0xC103 3800—0xC103 39C0</td>
<td>11100000000100000000111</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000111100 Dreg #</td>
</tr>
<tr>
<td><strong>Dreg_lo = A0 (IS)</strong></td>
<td>0xC183 3800—0xC183 39C0</td>
<td>11100000000111000000111</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000111100 Dreg #</td>
</tr>
<tr>
<td><strong>Dreg_lo = A0 (IU)</strong></td>
<td>0xC043 3800—0xC043 39C0</td>
<td>11100000000100000000111</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000111100 Dreg #</td>
</tr>
<tr>
<td><strong>Dreg_lo = A0 (T)</strong></td>
<td>0xC429 0000—0xC429 0038</td>
<td>11100001001x x 10101001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00000000000 Source 0</td>
</tr>
</tbody>
</table>
# Move Instructions

Table C-11. Move Instructions (Sheet 7 of 9)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move Register Half</td>
<td>0xC023 3800—</td>
<td></td>
<td>0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td>0xC023 39C0</td>
<td></td>
<td>0 0 1 1 1 0 0 Dreg # 0 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_lo = A0 (S2RND)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move Register Half</td>
<td>0xC123 3800—</td>
<td></td>
<td>0 1 0 0 0 0 0 1 0 0 1 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td>0xC123 39C0</td>
<td></td>
<td>0 0 1 1 1 0 0 Dreg # 0 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_lo = A0 (ISS2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move Register Half</td>
<td>0xC163 3800—</td>
<td></td>
<td>0 1 0 0 0 0 0 1 0 1 1 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td>0xC163 39C0</td>
<td></td>
<td>0 0 1 1 1 0 0 Dreg # 0 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_lo = A0 (IH)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move Register Half</td>
<td>0xC007 1800—</td>
<td></td>
<td>0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1</td>
</tr>
<tr>
<td></td>
<td>0xC007 19C0</td>
<td></td>
<td>0 0 0 1 1 0 0 Dreg # 0 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_hi = A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move Register Half</td>
<td>0xC107 1800—</td>
<td></td>
<td>0 1 0 0 0 0 0 1 0 0 0 0 0 1 1 1</td>
</tr>
<tr>
<td></td>
<td>0xC107 19C0</td>
<td></td>
<td>0 0 0 1 1 0 0 Dreg # 0 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_hi = A1 (IS)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move Register Half</td>
<td>0xC087 1800—</td>
<td></td>
<td>0 1 0 0 0 0 0 1 0 0 0 0 0 1 1 1</td>
</tr>
<tr>
<td></td>
<td>0xC087 19C0</td>
<td></td>
<td>0 0 0 1 1 0 0 Dreg # 0 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_hi = A1 (FU)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move Register Half</td>
<td>0xC187 1800—</td>
<td></td>
<td>0 1 0 0 0 0 0 1 1 0 0 0 0 1 1 1</td>
</tr>
<tr>
<td></td>
<td>0xC187 19C0</td>
<td></td>
<td>0 0 0 1 1 0 0 Dreg # 0 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_hi = A1 (IU)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Instruction Opcodes

#### Table C-11. Move Instructions (Sheet 8 of 9)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Move Register Half</strong></td>
<td>0xC047 1800—0xC047 19C0</td>
<td>1100000000000111</td>
</tr>
<tr>
<td><strong>Dreg_hi = A1 (T)</strong></td>
<td></td>
<td>00011110 Dreg # 00000000</td>
</tr>
</tbody>
</table>

| **Move Register Half**  | 0xC027 1800—0xC027 19C0 | 1100000000100111 |
| **Dreg_hi = A1 (S2RND)** |              | 00011110 Dreg # 00000000 |

| **Move Register Half**  | 0xC127 1800—0xC127 19C0 | 1100000100100111 |
| **Dreg_hi = A1 (ISS2)** |              | 00011110 Dreg # 00000000 |

| **Move Register Half**  | 0xC167 1800—0xC167 19C0 | 1100000101100111 |
| **Dreg_hi = A1 (IH)**   |              | 00011110 Dreg # 00000000 |

| **Move Register Half**  | 0xC007 3800—0xC007 39C0 | 1100000010000111 |
| **Dreg_lo = A0, Dreg_hi = A1 (FU)** |              | 00111110 Dreg # 00000000 |

| **Move Register Half**  | 0xC087 3800—0xC087 39C0 | 1100000010000111 |
| **Dreg_lo = A0, Dreg_hi = A1 (FU)** |              | 00111110 Dreg # 00000000 |

| **Move Register Half**  | 0xC107 3800—0xC107 39C0 | 1100000010000111 |
| **Dreg_lo = A0, Dreg_hi = A1 (IS)** |              | 00111110 Dreg # 00000000 |
### Move Instructions

Table C-11. Move Instructions (Sheet 9 of 9)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
<th>Source</th>
<th>Dest.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move Register Half</td>
<td>0xC187 3800—0xC187 39C0</td>
<td>1 1 0 0 0 0 0 1 1 0 0 0 0 1 1 1</td>
<td>0 0 1 1 1 0 0 Dreg # 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xC047 3800—0xC047 39C0</td>
<td>1 1 0 0 0 0 0 0 1 0 0 0 1 1 1</td>
<td>0 0 1 1 1 0 0 Dreg # 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xC027 3800—0xC027 39C0</td>
<td>1 1 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1</td>
<td>0 0 1 1 1 0 0 Dreg # 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xC127 3800—0xC127 39C0</td>
<td>1 1 0 0 0 0 0 1 0 0 1 0 0 1 1 1</td>
<td>0 0 1 1 1 0 0 Dreg # 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xC167 3800—0xC167 39C0</td>
<td>1 1 0 0 0 0 1 0 1 1 0 0 1 1 1</td>
<td>0 0 1 1 1 0 0 Dreg # 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>Move Byte, Zero Extended</td>
<td>0x4340—0x437F</td>
<td>0 1 0 0 0 0 1 1 0 1</td>
<td>Source Dreg # Dest. Dreg #</td>
<td></td>
</tr>
<tr>
<td>Dreg = Dreg_byte (Z)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move Byte, Sign Extended</td>
<td>0x4300—0x433F</td>
<td>0 1 0 0 0 0 1 1 0 0</td>
<td>Source Dreg # Dest. Dreg #</td>
<td></td>
</tr>
<tr>
<td>Dreg = Dreg_byte (X)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Instruction Opcodes

## Stack Control Instructions

Table C-12. Stack Control Instructions (Sheet 1 of 2)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Push</strong></td>
<td>0x0140—0x017F</td>
<td>0 0 0 0 0 0 0 0 1 0 1 1 1 Reg. group</td>
</tr>
<tr>
<td>[−−SP]=allreg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Push Multiple**

<table>
<thead>
<tr>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x05C0—0x05FD</td>
<td>0 0 0 0 0 0 0 1 0 1 1 1 Dreg #</td>
</tr>
</tbody>
</table>

**Push Multiple**

<table>
<thead>
<tr>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0540—0x0578</td>
<td>0 0 0 0 0 0 1 0 1 0 1 0 1 Preg #</td>
</tr>
</tbody>
</table>

**Push Multiple**

<table>
<thead>
<tr>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x04C0—0x04C5</td>
<td>0 0 0 0 0 0 1 0 1 1 0 0 0 Preg #</td>
</tr>
</tbody>
</table>

**Pop**

<table>
<thead>
<tr>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0100—0x013F</td>
<td>0 0 0 0 0 0 0 0 1 0 0 1 1 Reg. group</td>
</tr>
</tbody>
</table>

**Pop Multiple**

<table>
<thead>
<tr>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0580—0x05BD</td>
<td>0 0 0 0 0 0 1 0 1 1 0 Dreg #</td>
</tr>
</tbody>
</table>

**Pop Multiple**

<table>
<thead>
<tr>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R7:Dreglim, P5:Preglim)=SP++</td>
<td></td>
</tr>
</tbody>
</table>

**Pop**

<table>
<thead>
<tr>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>mostreg=SP++</td>
<td></td>
</tr>
</tbody>
</table>

**Pop Multiple**

<table>
<thead>
<tr>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R7:Dreglim, P5:Preglim)=SP++</td>
<td></td>
</tr>
</tbody>
</table>

NOTE: See two above notes on interpretation of the register number fields.

---

ADSP-BF53x/BF56x Blackfin Processor Programming Reference  C-37
## Stack Control Instructions

Table C-12. Stack Control Instructions (Sheet 2 of 2)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pop Multiple</strong></td>
<td>0x0500—0x0538</td>
<td><img src="image" alt="Bin" /></td>
</tr>
<tr>
<td><strong>Pop Multiple</strong></td>
<td>0x0480—0x0485</td>
<td><img src="image" alt="Bin" /></td>
</tr>
</tbody>
</table>

NOTE: The embedded register number represents the lowest register in the range to be used. Example: "100b" in that field means R7 through R4 are used.

(R7:Dreglim)=SP++

```
(R7:Dreglim)=SP++
```

NOTE: The embedded register number represents the lowest register in the range to be used. Example: "010b" in that field means P5 through P2 are used. The highest useful value allowed is P4.

(P5:Preglim)=SP++

```
(P5:Preglim)=SP++
```

**Linkage**

<table>
<thead>
<tr>
<th>0xE800 0000—0xE800 FFFF</th>
<th><img src="image" alt="Bin" /></th>
</tr>
</thead>
</table>

    uimm18m4 divided by 4

<table>
<thead>
<tr>
<th>0xE801 0000</th>
<th><img src="image" alt="Bin" /></th>
</tr>
</thead>
</table>

UNITLINK
## Control Code Bit Management Instructions

Table C-13. Control Code Bit Management Instructions (Sheet 1 of 4)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Compare Data Register</strong></td>
<td>0x0800—0x083F</td>
<td>00000010001000000</td>
</tr>
<tr>
<td>CC = Dreg == Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Compare Data Register</strong></td>
<td>0x0C00—0x0C3F</td>
<td>00000011001000000</td>
</tr>
<tr>
<td>CC = Dreg == imm3</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Compare Data Register</strong></td>
<td>0x0880—0x08BF</td>
<td>00000010001000001</td>
</tr>
<tr>
<td>CC = Dreg &lt; Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Compare Data Register</strong></td>
<td>0x0C80—0x0CBF</td>
<td>00000011001000100</td>
</tr>
<tr>
<td>CC = Dreg &lt; imm3</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Compare Data Register</strong></td>
<td>0x0900—0x093F</td>
<td>00000010001000100</td>
</tr>
<tr>
<td>CC = Dreg &lt;= Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Compare Data Register</strong></td>
<td>0x0D00—0x0D3F</td>
<td>00000011001001010</td>
</tr>
<tr>
<td>CC = Dreg &lt;= imm3</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Compare Data Register</strong></td>
<td>0x0980—0x09BF</td>
<td>00000010001011110</td>
</tr>
<tr>
<td>CC = Dreg &lt; Dreg (IU)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Compare Data Register</strong></td>
<td>0x0D80—0x0DBF</td>
<td>00000011001010010</td>
</tr>
<tr>
<td>CC = Dreg &lt; uimm3 (IU)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Compare Data Register</strong></td>
<td>0x0A00—0x0A3F</td>
<td>00000011001000000</td>
</tr>
<tr>
<td>CC = Dreg &lt;= Dreg (IU)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Control Code Bit Management Instructions

### Table C-13. Control Code Bit Management Instructions (Sheet 2 of 4)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Compare Data Register</strong></td>
<td>0x0E00—0x0E3F</td>
<td>0 0 0 0 1 1 1 1 0 0 0 uimm3 Dest reg #</td>
</tr>
<tr>
<td><strong>Compare Pointer Register</strong></td>
<td>0x0840—0x087F</td>
<td>0 0 0 0 1 0 0 0 1 Source reg # Dest reg #</td>
</tr>
<tr>
<td><strong>Compare Pointer Register</strong></td>
<td>0x0C40—0x0C7F</td>
<td>0 0 0 0 1 1 0 0 1 imm3 Dest reg #</td>
</tr>
<tr>
<td><strong>Compare Pointer Register</strong></td>
<td>0x08C0—0x08FF</td>
<td>0 0 0 1 0 0 0 1 1 Source reg # Dest reg #</td>
</tr>
<tr>
<td><strong>Compare Pointer Register</strong></td>
<td>0x0CC0—0x0CFF</td>
<td>0 0 0 1 1 0 0 1 1 imm3 Dest reg #</td>
</tr>
<tr>
<td><strong>Compare Pointer Register</strong></td>
<td>0x0940—0x097F</td>
<td>0 0 0 0 1 0 0 1 1 Source reg # Dest reg #</td>
</tr>
<tr>
<td><strong>Compare Pointer Register</strong></td>
<td>0x0D40—0x0D7F</td>
<td>0 0 0 1 1 1 1 1 imm3 Dest reg #</td>
</tr>
<tr>
<td><strong>Compare Pointer Register</strong></td>
<td>0x09C0—0x09FF</td>
<td>0 0 0 1 0 0 0 1 1 Source reg # Dest reg #</td>
</tr>
<tr>
<td><strong>Compare Pointer Register</strong></td>
<td>0x0DC0—0x0DFF</td>
<td>0 0 0 1 0 1 0 0 1 uimm3 Dest reg #</td>
</tr>
<tr>
<td><strong>Compare Pointer Register</strong></td>
<td>0x0A40—0x0A7F</td>
<td>0 0 0 1 0 1 0 0 1 Source reg # Dest reg #</td>
</tr>
</tbody>
</table>
### Instruction Opcodes

Table C-13. Control Code Bit Management Instructions (Sheet 3 of 4)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Binary</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Compare Pointer Register</strong></td>
<td>0x0E40—0x0E7F</td>
<td>Dreg = CC</td>
<td></td>
</tr>
<tr>
<td><strong>Compare Accumulator</strong></td>
<td>0xA8</td>
<td>CC = A0 == A1</td>
<td></td>
</tr>
<tr>
<td><strong>Compare Accumulator</strong></td>
<td>0xB0</td>
<td>CC = A0 &lt; A1</td>
<td></td>
</tr>
<tr>
<td><strong>Compare Accumulator</strong></td>
<td>0xB8</td>
<td>CC = A0 &lt;= A1</td>
<td></td>
</tr>
<tr>
<td><strong>Move CC</strong></td>
<td>0x0200—0x0207</td>
<td>statbit = CC</td>
<td></td>
</tr>
<tr>
<td><strong>Move CC</strong></td>
<td>0x0380—0x039F</td>
<td>statbit</td>
<td>= CC</td>
</tr>
<tr>
<td><strong>Move CC</strong></td>
<td>0x03A0—0x03BF</td>
<td>statbit &amp;= CC</td>
<td></td>
</tr>
<tr>
<td><strong>Move CC</strong></td>
<td>0x03C0—0x03DF</td>
<td>statbit ^= CC</td>
<td></td>
</tr>
<tr>
<td><strong>Move CC</strong></td>
<td>0x0208—0x020F</td>
<td>CC = Dreg</td>
<td></td>
</tr>
<tr>
<td><strong>Move CC</strong></td>
<td>0x0300—0x031F</td>
<td>CC = statbit</td>
<td></td>
</tr>
</tbody>
</table>
# Control Code Bit Management Instructions

Table C-13. Control Code Bit Management Instructions (Sheet 4 of 4)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move CC</td>
<td>0x0320—0x033F</td>
<td><img src="image" alt="Binary Representation" /></td>
</tr>
<tr>
<td>CC</td>
<td>= statbit</td>
<td></td>
</tr>
<tr>
<td>Move CC</td>
<td>0x0340—0x035F</td>
<td><img src="image" alt="Binary Representation" /></td>
</tr>
<tr>
<td>CC &amp;= statbit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move CC</td>
<td>0x0360—0x037F</td>
<td><img src="image" alt="Binary Representation" /></td>
</tr>
<tr>
<td>CC ^= statbit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negate CC</td>
<td>0x0218</td>
<td><img src="image" alt="Binary Representation" /></td>
</tr>
<tr>
<td>CC = !CC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C-42 ADSP-BF53x/BF56x Blackfin Processor Programming Reference
## Logical Operations Instructions

Table C-14. Logical Operations Instructions

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AND</strong></td>
<td>0x5400—0x55FF</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</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>Dreg = Dreg &amp; Dreg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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><strong>NOT (One's-Complement)</strong></td>
<td>0x43C0—0x43FF</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Dreg = ~ Dreg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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><strong>OR</strong></td>
<td>0x5600—0x57FF</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</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>Dreg = Dreg</td>
<td>Dreg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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><strong>Exclusive OR</strong></td>
<td>0x5800—0x59FF</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</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>Dreg = Dreg</td>
<td>Dreg = Dreg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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><strong>Bit Wise Exclusive OR</strong></td>
<td>0xC60B 0000—0xC60B 0E38</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dreg_lo = CC = BXORSHIFT (A0, Dreg)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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><strong>Bit Wise Exclusive OR</strong></td>
<td>0xC60B 4000—0xC60B 4E38</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dreg_lo = CC = BXOR (A0, Dreg)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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><strong>Bit Wise Exclusive OR</strong></td>
<td>C60C 4000—C60C 4E00</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Dreg_lo = CC = BXOR (A0, A1, CC)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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><strong>Bit Wise Exclusive OR</strong></td>
<td>C60C 0000</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A0 = BXORSHIFT (A0, A1, CC)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>

ADSP-BF53x/BF56x Blackfin Processor Programming Reference C-43
### Bit Operations Instructions

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bit Clear</strong></td>
<td>0x4C00—0x4CFF</td>
<td></td>
</tr>
<tr>
<td><strong>BITCLR (Dreg, uimm5)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bit Set</strong></td>
<td>0x4A00—0x4AFF</td>
<td></td>
</tr>
<tr>
<td><strong>BITSET (Dreg, uimm5)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bit Toggle</strong></td>
<td>0x4B00—0x4BFF</td>
<td></td>
</tr>
<tr>
<td><strong>BITTGL (Dreg, uimm5)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bit Test</strong></td>
<td>0x4900—0x49FF</td>
<td></td>
</tr>
<tr>
<td><strong>CC = BITTST (Dreg, uimm5)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bit Test</strong></td>
<td>0x4800—0x48FF</td>
<td></td>
</tr>
<tr>
<td><strong>CC = !BITTST (Dreg, uimm5)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bit Field Deposit</strong></td>
<td>0xC60A 8000—0xC60A 8E3F</td>
<td></td>
</tr>
<tr>
<td><strong>Dreg = DEPOSIT (Dreg, Dreg)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bit Field Deposit</strong></td>
<td>0xC60A C000—0xC60A CE3F</td>
<td></td>
</tr>
<tr>
<td><strong>Dreg = DEPOSIT (Dreg, Dreg) (X)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bit Field Extraction</strong></td>
<td>0xC60A 0000—0xC60A 0E3F</td>
<td></td>
</tr>
<tr>
<td><strong>Dreg = EXTRACT (Dreg, Dreg_lo) (Z)</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Instruction Opcodes

#### Table C-15. Bit Operations Instructions (Sheet 2 of 2)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Field Extraction</td>
<td>0xC60A 4000—0xC60A 4E3F</td>
<td>110001100xx01010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Dreg = EXTRACT (Dreg, Dreg_lo) (X)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit Multiplex</td>
<td>0xC608 0000—0xC608 003F</td>
<td>110001100xx01000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>BITMUX (Dreg, Dreg, A0) (ASR)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit Multiplex</td>
<td>0xC608 4000—0xC608 403F</td>
<td>110001100xx01000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>BITMUX (Dreg, Dreg, A0) (ASL)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>One's-Population Count</td>
<td>0xC606 C000—0xC606 CE07</td>
<td>110001100xx01000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Dreg_lo = ONES Dreg
## Shift / Rotate Operations Instructions

### Table C-16. Shift / Rotate Operations Instructions (Sheet 1 of 9)

| Instruction and Version | Opcode Range | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-------------------------|--------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Add with Shift          | 0x4580—0x45BF| 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  |    |    |    |    |    |
|                         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|                         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Add with Shift          | 0x45C0—0x45FF| 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 1  |    |    |    |    |    |    |
|                         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|                         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Preg = (Preg + Preg) << 1 |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|                        |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Add with Shift          | 0x4100—0x413F| 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  |    |    |    |    |    |    |
|                         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|                         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Dreg = (Dreg + Dreg) << 1 |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|                         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Add with Shift          | 0x4140—0x417F| 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  |    |    |    |    |    |
|                         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|                         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Dreg = (Dreg + Dreg) << 2 |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|                        |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Shift with Add          | 0x5C00—0x5DFF| 0  | 1  | 0  | 1  | 1  | 1  | 0  |    |    |    |    |    |    |    |    |    |
|                         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|                         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Preg = Preg + (Preg <<1) |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|                        |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Shift with Add          | 0x5E00—0x5FFFFF | 0  | 1  | 0  | 1  | 1  | 1  | 0  |    |    |    |    |    |    |    |    |    |
|                         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|                         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Preg = Preg + (Preg <<2) |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|                        |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Arithmetic Shift        | 0x4D00—0x4DFF | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  |
|                         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|                         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Dreg >>>= uimm5         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|                        |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Arithmetic Shift        | 0xC680 0180—0xC680 0FFFF | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
|                         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|                         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Dreg_lo = Dreg_lo >>> uimm4 |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|                        |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Arithmetic Shift        | 0xC680 1180—0xC680 1FFFF | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
|                         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|                         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Dreg_lo = Dreg_hi >>> uimm4 |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|                        |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Arithmetic Shift        | 0xC680 0180—0xC680 0FFFF | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
|                         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|                         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Dreg_lo = Dreg_lo >>> uimm4 |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|                        |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Arithmetic Shift        | 0xC680 1180—0xC680 1FFFF | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
|                         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|                         |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Dreg_lo = Dreg_hi >>> uimm4 |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|                        |              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

C-46 ADSP-BF53x/BF56x Blackfin Processor Programming Reference
### Instruction Opcodes

#### Table C-16. Shift / Rotate Operations Instructions (Sheet 2 of 9)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0xC680 2180—0xC680 2FFF</td>
<td>1 1 0 0 0 1 1 0 1 x x 0 0 0 0 0</td>
</tr>
<tr>
<td>Arithmetic Shift (S)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xC680 3180—0xC680 3FFF</td>
<td>1 1 0 0 0 1 1 0 1 x x 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo &gt;&gt;&gt; uimm4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arithmetic Shift (S)</td>
<td>0xC680 4000—0xC680 4E7F</td>
<td>1 1 0 0 0 1 1 0 1 x x 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo &lt;&lt; uimm4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arithmetic Shift (S)</td>
<td>0xC680 5000—0xC680 5E7F</td>
<td>1 1 0 0 0 1 1 0 1 x x 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo &lt;&lt; uimm4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arithmetic Shift (S)</td>
<td>0xC680 6000—0xC680 6E7F</td>
<td>1 1 0 0 0 1 1 0 1 x x 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo &lt;&lt; uimm4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arithmetic Shift (S)</td>
<td>0xC680 7000—0xC680 7E7F</td>
<td>1 1 0 0 0 1 1 0 1 x x 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_hi &lt;&lt; uimm4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arithmetic Shift (S)</td>
<td>0xC682 0100—0xC682 0FFF</td>
<td>1 1 0 0 0 1 1 0 1 x x 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg = Dreg &gt;&gt;&gt; uimm5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

ADSP-BF53x/BF56x Blackfin Processor Programming Reference  C-47
### Shift / Rotate Operations Instructions

Table C-16. Shift / Rotate Operations Instructions (Sheet 3 of 9)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td>0xC682 4000—0xC680 4EFF</td>
<td>1 1 0 0 0 0 1 1 0 1 x x 0 0 0 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1 0 0 Dest. Dreg # uimm5 Source Dreg #</td>
</tr>
<tr>
<td>Dreg = Dreg &lt;&lt; uimm5 (S)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td>0xC683 0100—0xC683 01F8</td>
<td>1 1 0 0 0 0 1 1 0 1 x x 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 0 0 0 0 0 2's complement of uimm5 0 0 0</td>
</tr>
<tr>
<td>A0 = A0 &gt;&gt;&gt; uimm5</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td>0xC683 1100—0xC683 11F8</td>
<td>1 1 0 0 0 0 1 1 0 1 x x 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 0 1 0 0 0 2's complement of uimm5 0 0 0</td>
</tr>
<tr>
<td>A1 = A1 &gt;&gt;&gt; uimm5</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td>0x4000—0x403F</td>
<td>0 1 0 0 0 0 0 0 0 0 0 Source Dreg # Dest. Dreg #</td>
</tr>
<tr>
<td>Dreg &gt;&gt;&gt;= Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td>0xC600 0000—0xC600 0E3F</td>
<td>1 1 0 0 0 1 1 0 0 x x 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 0 0 Dest. Dreg # x x Source Dreg # sh_mag Dreg #</td>
</tr>
<tr>
<td>Dreg_lo = ASHIFT Dreg_lo BY Dreg_lo</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td>0xC600 1000—0xC600 1E3F</td>
<td>1 1 0 0 0 1 1 0 0 x x 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 0 1 Dest. Dreg # x x Source Dreg # sh_mag Dreg #</td>
</tr>
<tr>
<td>Dreg_lo = ASHIFT Dreg_hi BY Dreg_lo</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td>0xC600 2000—0xC600 2E3F</td>
<td>1 1 0 0 0 1 1 0 0 x x 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1 0 Dest. Dreg # x x Source Dreg # sh_mag Dreg #</td>
</tr>
<tr>
<td>Dreg_hi = ASHIFT Dreg_lo BY Dreg_lo</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td>0xC600 3000—0xC600 3E3F</td>
<td>1 1 0 0 0 1 1 0 0 x x 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1 1 Dest. Dreg # x x Source Dreg # sh_mag Dreg #</td>
</tr>
<tr>
<td>Dreg_hi = ASHIFT Dreg_hi BY Dreg_lo</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Instruction Opcodes

#### Table C-16. Shift / Rotate Operations Instructions (Sheet 4 of 9)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td>0xC600 4000—0xC600 4E3F</td>
<td>110001100xx00000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000</td>
</tr>
<tr>
<td>Dreg_lo = ASHIFT Dreg_lo BY Dreg_lo (S)</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td>0xC600 5000—0xC600 5E3F</td>
<td>110001100xx00000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0110</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000</td>
</tr>
<tr>
<td>Dreg_lo = ASHIFT Dreg_hi BY Dreg_lo (S)</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td>0xC600 6000—0xC600 6E3F</td>
<td>110001100xx00000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0111</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000</td>
</tr>
<tr>
<td>Dreg_hi = ASHIFT Dreg_lo BY Dreg_lo (S)</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td>0xC602 0000—0xC602 0E3F</td>
<td>110001100xx00000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000</td>
</tr>
<tr>
<td>Dreg = ASHIFT Dreg BY Dreg_lo</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td>0xC602 4000—0xC602 4E3F</td>
<td>110001100xx00000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000</td>
</tr>
<tr>
<td>Dreg = ASHIFT Dreg BY Dreg_lo (S)</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td>0xC603 0000—0xC603 0038</td>
<td>110001100xx00000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000</td>
</tr>
<tr>
<td>A0 = ASHIFT A0 BY Dreg_lo</td>
<td></td>
<td>11</td>
</tr>
</tbody>
</table>
### Shift / Rotate Operations Instructions

#### Table C-16. Shift / Rotate Operations Instructions (Sheet 5 of 9)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td>0xC603 1000 — 0xC603 1038</td>
<td>1 1 0 0 0 1 1 0 0 x x 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 0 1 0 0 0 x x Source Dreg #</td>
</tr>
<tr>
<td>A1 = ASHIFT A1 BY Dreg_lo</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Logical Shift | 0x4500 — 0x453F | 0 1 0 0 0 1 0 1 0 0 Source Preg # | Dest. Preg # |
| Preg = Preg >> 1 |

| Logical Shift | 0x44C0 — 0x44FF | 0 1 0 0 0 1 0 0 1 1 Source Preg # | Dest. Preg # |
| Preg = Preg >> 2 |

| Logical Shift | 0x5A00 — 0x5BFF | 0 1 0 1 1 0 0 1 1 Source Preg # | Dest. Preg # | Dest. Preg # |
| NOTE: Both Destination Preg # fields must refer to the same Preg number. Otherwise, this opcode represents an Add with Shift instruction. |
| NOTE: This Preg = Preg <<1 instruction produces the same opcode as the special case of the Preg = Preg + Preg Add instruction, where both input operands are the same Preg (e.g., p3 = p0+p0) that accomplishes the same function. Both syntaxes double the input operand value, then place the result in a Preg. |
| Preg = Preg << 1 |

| Logical Shift | 0x4440 — 0x447F | 0 1 0 0 0 1 0 0 1 0 0 1 Source Preg # | Dest. Preg # |
| Preg = Preg << 2 |

| Logical Shift | 0x4E00 — 0x4EFF | 0 1 0 0 1 1 1 uimm5 | Dest. Dreg # |
| Dreg >>= uimm5 |

| Logical Shift | 0x4F00 — 0x4FFF | 0 1 0 0 1 1 uimm5 | Dest. Dreg # |
| Dreg <<= uimm5 |

| Logical Shift | 0xC680 8180 — 0xC680 8FFF | 1 1 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 |
| Dreg_lo = Dreg_lo >> uimm4 |

| 2's comp. of uimm4 | Source Dreg # |

C-50 ADSP-BF53x/BF56x Blackfin Processor Programming Reference
### Instruction Opcodes

Table C-16. Shift / Rotate Operations Instructions (Sheet 6 of 9)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical Shift</td>
<td>0xC680 9180—0xC680 9FFF</td>
<td>1 1 0 0 0 1 1 0 1 x x 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 0 1 Dest. Dreg #</td>
</tr>
<tr>
<td>Logical Shift</td>
<td>0xC680 A180—0xC680 AFFF</td>
<td>1 1 0 0 0 1 1 0 1 x x 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 1 0 Dest. Dreg #</td>
</tr>
<tr>
<td>Logical Shift</td>
<td>0xC680 B180—0xC680 BFFF</td>
<td>1 1 0 0 0 1 1 0 1 x x 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 1 1 Dest. Dreg #</td>
</tr>
<tr>
<td>Logical Shift</td>
<td>0xC680 8000—0xC680 8E7F</td>
<td>1 1 0 0 0 1 1 0 1 x x 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 0 0 Dest. Dreg #</td>
</tr>
<tr>
<td>Logical Shift</td>
<td>0xC680 9000—0xC680 9E7F</td>
<td>1 1 0 0 0 1 1 0 1 x x 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 0 1 Dest. Dreg #</td>
</tr>
<tr>
<td>Logical Shift</td>
<td>0xC680 A000—0xC680 AE7F</td>
<td>1 1 0 0 0 1 1 0 1 x x 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 1 0 Dest. Dreg #</td>
</tr>
<tr>
<td>Logical Shift</td>
<td>0xC680 B000—0xC680 BE7F</td>
<td>1 1 0 0 0 1 1 0 1 x x 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 1 1 Dest. Dreg #</td>
</tr>
</tbody>
</table>
## Shift / Rotate Operations Instructions

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Logical Shift</strong></td>
<td>0xC682 8100—0xC682 8FFF</td>
<td></td>
<td>1 1 0 0 0 1 1 0 1 x 0 0 0 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg = Dreg &gt;&gt; uimm5</td>
<td></td>
<td>0100 0000 0 2’s comp. of uimm5</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td>0xC682 8000—0xC682 8EFF</td>
<td></td>
<td>1 1 0 0 0 1 1 0 1 x 0 0 0 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg = Dreg &lt;&lt; uimm5</td>
<td></td>
<td>0100 0000 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td>0xC683 4100—0xC683 41F8</td>
<td></td>
<td>1 1 0 0 0 1 1 0 1 x 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1 1 0 0 0 0 0 0 0 2’s comp. of uimm5</td>
</tr>
<tr>
<td>A0 = A0 &gt;&gt; uimm5</td>
<td></td>
<td>0100 0000 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td>0xC683 4000—0xC683 40F8</td>
<td></td>
<td>1 1 0 0 0 1 1 0 1 x 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>A0 = A0 &lt;&lt; uimm5</td>
<td></td>
<td>0100 0000 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td>0xC683 5100—0xC683 51F8</td>
<td></td>
<td>1 1 0 0 0 1 1 0 1 x 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1 1 0 1 0 0 0 0 0 2’s comp. of uimm5</td>
</tr>
<tr>
<td>A1 = A1 &gt;&gt; uimm5</td>
<td></td>
<td>0100 0000 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td>0xC683 5000—0xC683 50F8</td>
<td></td>
<td>1 1 0 0 0 1 1 0 1 x 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>A1 = A1 &lt;&lt; uimm5</td>
<td></td>
<td>0100 0000 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td>0x4080—0x40BF</td>
<td></td>
<td>0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg &lt;&lt;= Dreg</td>
<td></td>
<td>0100 0000 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td>0x4040—0x407F</td>
<td></td>
<td>0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg &gt;&gt;= Dreg</td>
<td></td>
<td>0100 0000 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

Table C-16. Shift / Rotate Operations Instructions (Sheet 7 of 9)
### Logical Shift

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Logical Shift</strong></td>
<td>0xC600 8000—</td>
<td>0xC600 8E3F</td>
</tr>
<tr>
<td></td>
<td>1 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 0 0 Dest. Dreg # xxx Source Dreg # sh_mag Dreg #</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dreg_lo = LSHIFT Dreg_lo BY Dreg_lo</td>
<td></td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td>0xC600 9000—</td>
<td>0xC600 9E3F</td>
</tr>
<tr>
<td></td>
<td>1 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 0 1 Dest. Dreg # xxx Source Dreg # sh_mag Dreg #</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dreg_lo = LSHIFT Dreg_hi BY Dreg_lo</td>
<td></td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td>0xC600 A000—</td>
<td>0xC600 AE3F</td>
</tr>
<tr>
<td></td>
<td>1 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 1 0 Dest. Dreg # xxx Source Dreg # sh_mag Dreg #</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dreg_hi = LSHIFT Dreg_lo BY Dreg_lo</td>
<td></td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td>0xC600 B000—</td>
<td>0xC600 BE3F</td>
</tr>
<tr>
<td></td>
<td>1 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 1 1 Dest. Dreg # xxx Source Dreg # sh_mag Dreg #</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dreg_hi = LSHIFT Dreg_hi BY Dreg_lo</td>
<td></td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td>0xC602 8000—</td>
<td>0xC602 8E3F</td>
</tr>
<tr>
<td></td>
<td>1 1 0 0 0 1 1 0 0 0 0 0 0 0 0 1 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 0 0 Dest. Dreg # xxx Source Dreg # sh_mag Dreg #</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dreg = LSHIFT Dreg BY Dreg_lo</td>
<td></td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td>0xC603 4000—</td>
<td>0xC603 4038</td>
</tr>
<tr>
<td></td>
<td>1 1 0 0 0 1 1 0 0 0 0 0 1 1 1 1 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 1 0 0 0 0 0 0 Source Dreg # 0 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A0 = LSHIFT A0 BY Dreg_lo</td>
<td></td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td>0xC603 5000—</td>
<td>0xC603 5038</td>
</tr>
<tr>
<td></td>
<td>1 1 0 0 0 1 1 0 0 0 0 0 1 1 1 1 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 1 0 1 0 0 0 Source Dreg # 0 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A1 = LSHIFT A1 BY Dreg_lo</td>
<td></td>
</tr>
</tbody>
</table>
# Shift / Rotate Operations Instructions

Table C-16. Shift / Rotate Operations Instructions (Sheet 9 of 9)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rotate</td>
<td>0xC682 C000—</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xC682 CFFF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>111001101xx000100 imm6 Source Dreg #</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11000 Dest. Dreg #</td>
<td></td>
</tr>
<tr>
<td>Dreg = ROT Dreg BY imm6</td>
<td>0xC683 8000—</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xC683 81F8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>111001101xx000110000 immm6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11000 Dest. Dreg #</td>
<td></td>
</tr>
<tr>
<td>A0 = ROT A0 BY imm6</td>
<td>0xC683 9000—</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xC683 91F8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>111001101xx000110000 immm6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11000 Dest. Dreg #</td>
<td></td>
</tr>
<tr>
<td>A1 = ROT A1 BY imm6</td>
<td>0xC602 C000—</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xC602 CFFFF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>111001100xx000100 imm6 Source Dreg #</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11000 Dest. Dreg #</td>
<td></td>
</tr>
<tr>
<td>Dreg = ROT Dreg BY Dreg_lo</td>
<td>0xC603 8000—</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xC603 8038</td>
<td></td>
</tr>
<tr>
<td></td>
<td>111001100xx000110000 immm6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11000 Dest. Dreg #</td>
<td></td>
</tr>
<tr>
<td>A0 = ROT A0 BY Dreg_lo</td>
<td>0xC603 9000—</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xC603 9038</td>
<td></td>
</tr>
<tr>
<td></td>
<td>111001100xx000110000 immm6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11000 Dest. Dreg #</td>
<td></td>
</tr>
<tr>
<td>A1 = ROT A1 BY Dreg_lo</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table C-17. Arithmetic Operations Instructions (Sheet 1 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute Value</td>
<td>0xC410 403F</td>
<td>11 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

A0 = ABS A1

| Absolute Value | 0xC430 003F | 11 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 |

A1 = ABS A0

| Absolute Value | 0xC430 403F | 11 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 |

A1 = ABS A1

| Absolute Value | 0xC410 C03F | 11 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 |

A1 = ABS A1, A0 = ABS A0

| Absolute Value | 0xC407 8000—0xC407 8E38 | 1 1 0 0 0 0 0 1 1 0 0 0 0 1 1 1 1 |

Dreg = ABS Dreg

| Add | 0x5A00—0x5BFF | 0 1 1 0 1 0 0 1 0 0 1 1 1 1 1 1 1 1 1 1 |

NOTE: The special case of Preg = Preg + Preg, where both input operands are the same Preg (e.g., p3 = p0+p0), produces the same opcode as the Logical Shift instruction Preg = Preg << 1 that accomplishes the same function. Both syntaxes double the input operand value, then place the result in a Preg.

Preg = Preg + Preg

| Add | 0x5000—0x51FF | 0 1 1 0 1 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 |

Dreg = Dreg + Dreg
## Arithmetic Operations Instructions

### Table C-17. Arithmetic Operations Instructions (Sheet 2 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>0xC404 0000— 0xC404 0E3F</td>
<td>1100 0010 001 0 x x 0 0 0 1 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg = Dreg + Dreg (NS)</td>
<td></td>
<td>0 0 0 0 Dest. Dreg # 0 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Add</td>
<td>0xC404 2000— 0xC404 2E3F</td>
<td>1100 0010 001 0 x x 0 0 0 1 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg = Dreg + Dreg (S)</td>
<td></td>
<td>0 0 1 0 Dest. Dreg # 0 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Add</td>
<td>0xC402 0000— 0xC402 0E3F</td>
<td>1100 0010 001 0 x x 0 0 0 0 0 1 0 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo + Dreg_lo (NS)</td>
<td></td>
<td>0 0 0 0 Dest. Dreg # 0 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Add</td>
<td>0xC402 4000— 0xC402 4E3F</td>
<td>1100 0010 001 0 x x 0 0 0 0 0 0 1 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo + Dreg_hi (NS)</td>
<td></td>
<td>0 0 0 0 Dest. Dreg # 0 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Add</td>
<td>0xC402 8000— 0xC402 8E3F</td>
<td>1100 0010 001 0 x x 0 0 0 0 0 0 0 1 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_hi + Dreg_lo (NS)</td>
<td></td>
<td>1 0 0 0 Dest. Dreg # 0 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Add</td>
<td>0xC402 C000— 0xC402 CE3F</td>
<td>1100 0010 001 0 x x 0 0 0 0 0 0 0 1 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_hi + Dreg_hi (NS)</td>
<td></td>
<td>1 1 0 0 Dest. Dreg # 0 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Add</td>
<td>0xC422 0000— 0xC422 0E3F</td>
<td>1100 0010 001 0 x x 1 0 0 0 0 1 0 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo + Dreg_lo (NS)</td>
<td></td>
<td>0 0 0 0 Dest. Dreg # 0 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
</tbody>
</table>
Table C-17. Arithmetic Operations Instructions (Sheet 3 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add 0xC422 4000—</td>
<td>1 1 0 0 0 1 0 x x x 1 0 0 0 1 0</td>
<td>0 1 0 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Add 0xC422 4E3F</td>
<td>1 1 0 0 0 1 0 x x x 1 0 0 0 1 0</td>
<td>0 1 0 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo + Dreg_hi (NS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add 0xC422 8000—</td>
<td>1 1 0 0 0 1 0 x x x 1 0 0 0 1 0</td>
<td>1 0 0 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Add 0xC422 8E3F</td>
<td>1 1 0 0 0 1 0 x x x 1 0 0 0 1 0</td>
<td>1 0 0 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_hi + Dreg_lo (NS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add 0xC422 C000—</td>
<td>1 1 0 0 0 1 0 x x x 1 0 0 0 1 0</td>
<td>1 1 0 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Add 0xC422 CE3F</td>
<td>1 1 0 0 0 1 0 x x x 1 0 0 0 1 0</td>
<td>1 1 0 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_hi + Dreg_hi (NS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add 0xC402 2000—</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 0 1 0</td>
<td>0 0 1 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Add 0xC402 2E3F</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 0 1 0</td>
<td>0 0 1 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo + Dreg_lo (S)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add 0xC402 6000—</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 0 1 0</td>
<td>0 1 1 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Add 0xC402 6E3F</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 0 1 0</td>
<td>0 1 1 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo + Dreg_hi (S)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add 0xC402 A000—</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 0 1 0</td>
<td>1 0 1 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Add 0xC402 AE3F</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 0 1 0</td>
<td>1 0 1 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_hi + Dreg_lo (S)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add 0xC402 E000—</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 0 1 0</td>
<td>1 1 1 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Add 0xC402 EE3F</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 0 1 0</td>
<td>1 1 1 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_hi + Dreg_hi (S)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Arithmetic Operations Instructions

## Table C-17. Arithmetic Operations Instructions (Sheet 4 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Add</strong> 0xC422 2000— 0xC422 2E3F</td>
<td>1 1 0 0 0 1 0 x x x 1 0 0 0 1 0</td>
<td>0 0 1 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td><strong>Add</strong> 0xC422 6000— 0xC422 6E3F</td>
<td>1 1 0 0 0 1 0 x x x 1 0 0 0 1 0</td>
<td>0 1 1 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td><strong>Add</strong> 0xC422 A000— 0xC422 AE3F</td>
<td>1 1 0 0 0 1 0 x x x 1 0 0 0 1 0</td>
<td>1 0 1 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td><strong>Add</strong> 0xC422 E000— 0xC422 EE3F</td>
<td>1 1 0 0 0 1 0 x x x 1 0 0 0 1 0</td>
<td>1 1 1 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td><strong>Add/Subtract, Prescale Down</strong> 0xC405 9000— 0xC405 9E3F</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 1 0 1</td>
<td>1 0 0 1 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td><strong>Add/Subtract, Prescale Down</strong> 0xC425 9000— 0xC425 9E3F</td>
<td>1 1 0 0 0 1 0 x x x 1 0 0 1 0 1</td>
<td>1 0 0 1 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td><strong>Add/Subtract, Prescale Down</strong> 0xC405 D000— 0xC405 DE3F</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 1 0 1</td>
<td>1 1 0 1 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
</tbody>
</table>

Dreg_hi = Dreg_lo + Dreg_lo (S)
Dreg_hi = Dreg_lo + Dreg_hi (S)
Dreg_hi = Dreg_hi + Dreg_lo (S)
Dreg_hi = Dreg_hi + Dreg_hi (S)
Dreg_lo = Dreg + Dreg (RND20)
Dreg_hi = Dreg + Dreg (RND20)
Dreg_lo = Dreg – Dreg (RND20)
### Table C-17. Arithmetic Operations Instructions (Sheet 5 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add/Subtract, Prescale Down</td>
<td>0xC425 D000—0xC425 DE3F</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Dest.</td>
<td>Dreg #</td>
<td>0</td>
<td>0</td>
<td>Source 0</td>
<td>Dreg #</td>
<td>Source 1</td>
<td>Dreg #</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg_hi = Dreg – Dreg (RND20)</td>
<td></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>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Add/Subtract, Prescale Up</td>
<td>0xC405 0000—0xC405 0E3F</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></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>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Dreg_lo = Dreg + Dreg (RND12)</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Add/Subtract, Prescale Up</td>
<td>0xC425 0000—0xC425 0E3F</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></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>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Dreg_hi = Dreg + Dreg (RND12)</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Add/Subtract, Prescale Up</td>
<td>0xC405 4000—0xC405 4E3F</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></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>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Dreg_lo = Dreg – Dreg (RND12)</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Add/Subtract, Prescale Up</td>
<td>0xC425 4000—0xC425 4E3F</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></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>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Dreg_hi = Dreg – Dreg (RND12)</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Add Immediate</td>
<td>0x6400—0x6700</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>imm7</td>
<td>Dreg #</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</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>Dreg += imm7</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>imm7</td>
<td>Preg #</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>imm7</td>
<td>Preg #</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Add Immediate</td>
<td>0x9F60—0x9F63</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Ireg #</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
## Arithmetic Operations Instructions

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Immediate</td>
<td>0x9F68—0x9F6B</td>
<td>1</td>
<td>0 0 1 1 1 1 1 1 0 1 1 0 1 0 Ireg #</td>
</tr>
<tr>
<td>Ireg += 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Divide Primitive</td>
<td>0x4240—0x427F</td>
<td>0 1 0 0 0 1 0 0 1 0 0 1 Source Dreg #</td>
<td>Dest. Dreg #</td>
</tr>
<tr>
<td>DIVS (Dreg, Dreg)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Divide Primitive</td>
<td>0x4200—0x423F</td>
<td>0 1 0 0 0 0 1 0 0 0 0 Source Dreg #</td>
<td>Dest. Dreg #</td>
</tr>
<tr>
<td>DIVQ (Dreg, Dreg)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exponent Detection</td>
<td>0xC607 0000—</td>
<td>1 1 0 0 0 1 1 1 0 0 x x 0 0 1 1 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xC607 0E3F</td>
<td>0 0 0 0 Dest. Dreg # x x Source 0 Dreg #</td>
<td>Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg_lo = EXPADJ (Dreg, Dreg_lo)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exponent Detection</td>
<td>0xC607 8000—</td>
<td>1 1 0 0 0 1 1 0 0 x x 0 0 1 1 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xC607 8E3F</td>
<td>1 0 0 0 Dest. Dreg # x x Source 0 Dreg #</td>
<td>Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg_lo = EXPADJ (Dreg_lo, Dreg_lo)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exponent Detection</td>
<td>0xC607 C000—</td>
<td>1 1 0 0 0 1 1 0 0 x x 0 0 1 1 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xC607 CE3F</td>
<td>1 1 0 0 Dest. Dreg # x x Source 0 Dreg #</td>
<td>Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg_lo = EXPADJ (Dreg_hi, Dreg_lo)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exponent Detection</td>
<td>0xC607 4000—</td>
<td>1 1 0 0 0 1 1 0 0 x x 0 0 1 1 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xC607 4E3F</td>
<td>0 0 0 0 Dest. Dreg # x x Source 0 Dreg #</td>
<td>Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg_lo = EXPADJ (Dreg_lo) (V)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum</td>
<td>0xC407 0000—</td>
<td>1 1 0 0 0 1 0 0 0 x x 0 0 1 1 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xC407 0E3F</td>
<td>0 0 0 0 Dest. Dreg # x x Source 0 Dreg #</td>
<td>Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg = MAX (Dreg, Dreg)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Instruction Opcodes

#### Table C-17. Arithmetic Operations Instructions (Sheet 7 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>0xC407 4000—0xC407 4E3F</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Modify, Decrement</td>
<td>0xC40B C03F</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A0 – = A1</td>
<td>0x4400—0x443F</td>
<td>1</td>
<td>1</td>
<td>1</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Modify, Decrement</td>
<td>0xC40B E03F</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A0 – = A1 (W32)</td>
<td>0x9E70—0x9E7F</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Modify, Decrement</td>
<td>0x9E70—0x9E7F</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Modify, Increment</td>
<td>0xC40B 803F</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A0 += A1</td>
<td>0xC40B A03F</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Modify, Increment</td>
<td>0xC40B A03F</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</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>Modify, Increment</td>
<td>0x4540—0x457F</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Modify, Increment</td>
<td>0x9E60—0x9E6F</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Ireg += Mreg</td>
<td>0x9E60—0x9E6F</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Dreg = MIN (Dreg, Dreg)

Preg = Preg

Ireg = Mreg

A0 += A1

A0 += A1 (W32)

Preg += Preg (BREV)

Ireg += Mreg
## Arithmetic Operations Instructions

Table C-17. Arithmetic Operations Instructions (Sheet 8 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modify, Increment</td>
<td>0x9EE0—</td>
<td>1 0 0 1 1 1 1 0 1 1 1 0</td>
</tr>
<tr>
<td></td>
<td>0x9EEF</td>
<td></td>
</tr>
<tr>
<td>Ireg += Mreg (brev)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modify, Increment</td>
<td>0xC40B 003F—</td>
<td>1 1 0 0 0 1 0 x x x 0 0 1 0 1 1</td>
</tr>
<tr>
<td></td>
<td>0xC40B 0E00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 0 0</td>
<td>Dest Dreg #</td>
</tr>
<tr>
<td></td>
<td>0 0 0 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 1 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 1</td>
<td></td>
</tr>
<tr>
<td>Dreg = (A0 += A1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modify, Increment</td>
<td>0xC42B 403F—</td>
<td>1 1 0 0 0 1 0 x x x 1 0 1 0 1 1</td>
</tr>
<tr>
<td></td>
<td>0xC42B 4E00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 1 0 0</td>
<td>Dest Dreg #</td>
</tr>
<tr>
<td></td>
<td>0 0 0 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 1 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 1</td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = (A0 += A1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modify, Increment</td>
<td>0xC200 2000—</td>
<td>1 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>0xC200 27FF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 1 0</td>
<td>Dreg half Dreg #</td>
</tr>
<tr>
<td></td>
<td>Dest. Dreg #</td>
<td></td>
</tr>
<tr>
<td></td>
<td>src_reg_0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>src_reg_1</td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modify, Increment</td>
<td>0xC280 2000—</td>
<td>1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>0xC280 27FF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 1 0</td>
<td>Dreg half Dreg #</td>
</tr>
<tr>
<td></td>
<td>Dest. Dreg #</td>
<td></td>
</tr>
<tr>
<td></td>
<td>src_reg_0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>src_reg_1</td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (FU)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modify, Increment</td>
<td>C300 2000—</td>
<td>1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>0xC300 27FF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 1 0</td>
<td>Dreg half Dreg #</td>
</tr>
<tr>
<td></td>
<td>Dest. Dreg #</td>
<td></td>
</tr>
<tr>
<td></td>
<td>src_reg_0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>src_reg_1</td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (IS)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: When issuing compatible load/store instructions in parallel with a Multiply 16-Bit Operands instruction, add 0x0800 0000 to the Multiply 16-Bit Operands opcode.

Dreg_hi = (A0 += A1)

Dreg_lo = Dreg_lo_hi * Dreg_lo_hi

Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (FU)

Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (IS)
## Instruction Opcodes

### Multiply 16-Bit Operands

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC380 2000—0xC380 27FF</td>
<td>1 1 0 0 0 0 1 1 1 0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1 0 0 Dreg half Dest. Dreg # src_reg_ src_reg_ src_reg_ 1 Dreg #</td>
</tr>
</tbody>
</table>

\[
\text{Dreg} \_\text{lo} = \text{Dreg} \_\text{lo} \_\text{hi} \times \text{Dreg} \_\text{lo} \_\text{hi} \ (\text{IU})
\]

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC240 2000—0xC240 27FF</td>
<td>1 1 0 0 0 0 1 0 0 1 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1 0 0 Dreg half Dest. Dreg # src_reg_ src_reg_ src_reg_ 1 Dreg #</td>
</tr>
</tbody>
</table>

\[
\text{Dreg} \_\text{lo} = \text{Dreg} \_\text{lo} \_\text{hi} \times \text{Dreg} \_\text{lo} \_\text{hi} \ (\text{T})
\]

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC2C0 2000—0xC2C0 27FF</td>
<td>1 1 0 0 0 0 0 1 1 1 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1 0 0 Dreg half Dest. Dreg # src_reg_ src_reg_ src_reg_ 1 Dreg #</td>
</tr>
</tbody>
</table>

\[
\text{Dreg} \_\text{lo} = \text{Dreg} \_\text{lo} \_\text{hi} \times \text{Dreg} \_\text{lo} \_\text{hi} \ (\text{TFU})
\]

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC220 2000—0xC220 27FF</td>
<td>1 1 0 0 0 0 0 0 1 0 0 1 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1 0 0 Dreg half Dest. Dreg # src_reg_ src_reg_ src_reg_ 1 Dreg #</td>
</tr>
</tbody>
</table>

\[
\text{Dreg} \_\text{lo} = \text{Dreg} \_\text{lo} \_\text{hi} \times \text{Dreg} \_\text{lo} \_\text{hi} \ (\text{S2RND})
\]

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC320 2000—0xC320 27FF</td>
<td>1 1 0 0 0 0 0 1 1 0 0 1 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1 0 0 Dreg half Dest. Dreg # src_reg_ src_reg_ src_reg_ 1 Dreg #</td>
</tr>
</tbody>
</table>

\[
\text{Dreg} \_\text{lo} = \text{Dreg} \_\text{lo} \_\text{hi} \times \text{Dreg} \_\text{lo} \_\text{hi} \ (\text{ISS2})
\]

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC360 2000—0xC360 27FF</td>
<td>1 1 0 0 0 0 0 1 1 0 1 1 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1 0 0 Dreg half Dest. Dreg # src_reg_ src_reg_ src_reg_ 1 Dreg #</td>
</tr>
</tbody>
</table>

\[
\text{Dreg} \_\text{lo} = \text{Dreg} \_\text{lo} \_\text{hi} \times \text{Dreg} \_\text{lo} \_\text{hi} \ (\text{IH})
\]

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC208 2000—0xC208 27FF</td>
<td>1 1 0 0 0 0 0 0 1 0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1 0 0 Dreg half Dest. Dreg # src_reg_ src_reg_ src_reg_ 1 Dreg #</td>
</tr>
</tbody>
</table>

\[
\text{Dreg} \_\text{even} = \text{Dreg} \_\text{lo} \_\text{hi} \times \text{Dreg} \_\text{lo} \_\text{hi}
\]

### NOTE

When issuing compatible load/store instructions in parallel with a Multiply 16-Bit Operands instruction, add 0x0800 0000 to the Multiply 16-Bit Operands opcode.
### Table C-17. Arithmetic Operations Instructions (Sheet 10 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC288 2000—0xC288 27FF</td>
<td>11 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi (FU)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC308 2000—0xC308 27FF</td>
<td>11 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi (IS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC228 2000—0xC228 27FF</td>
<td>11 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi (S2RND)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC328 2000—0xC328 27FF</td>
<td>11 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0</td>
</tr>
<tr>
<td>NOTE: When issuing compatible load/store instructions in parallel with a Multiply 16-Bit Operands instruction, add 0x0800 0000 to the Multiply 16-Bit Operands opcode.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC204 0000—0xC204 C1FF</td>
<td>11 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC284 0000—0xC284 C1FF</td>
<td>11 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (FU)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC304 0000—0xC304 C1FF</td>
<td>11 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IS)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Instruction Opcodes

### Table C-17. Arithmetic Operations Instructions (Sheet 11 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC384 0000—0xC384 C1FF</td>
<td>1 1 1 0 0 0 1 1 1 0 0 0 1 0 0 1 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half 0 0 0 0 0 0 Dest. src_reg_0 Dreg # src_reg_1 Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IU)</td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC244 0000—0xC244 C1FF</td>
<td>1 1 1 0 0 0 1 0 0 1 0 0 1 0 0 1 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half 0 0 0 0 0 0 Dest. src_reg_0 Dreg # src_reg_1 Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (T)</td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC2C4 0000—0xC2C4 C1FF</td>
<td>1 1 1 0 0 0 1 0 1 1 0 0 1 0 0 1 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half 0 0 0 0 0 0 Dest. src_reg_0 Dreg # src_reg_1 Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (TFU)</td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC224 0000—0xC224 C1FF</td>
<td>1 1 1 0 0 0 1 0 0 0 1 0 0 1 0 0 1 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half 0 0 0 0 0 0 Dest. src_reg_0 Dreg # src_reg_1 Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (S2RND)</td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC324 0000—0xC324 C1FF</td>
<td>1 1 1 0 0 0 1 1 0 1 0 1 0 0 1 0 1 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half 0 0 0 0 0 0 Dest. src_reg_0 Dreg # src_reg_1 Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (ISS2)</td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC364 0000—0xC364 C1FF</td>
<td>1 1 1 0 0 0 1 1 0 1 1 0 0 1 0 1 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half 0 0 0 0 0 0 Dest. src_reg_0 Dreg # src_reg_1 Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IH)</td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC214 0000—0xC214 C1FF</td>
<td>1 1 1 0 0 0 1 0 0 0 1 0 1 0 1 0 1 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half 0 0 0 0 0 0 Dest. src_reg_0 Dreg # src_reg_1 Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (M)</td>
</tr>
</tbody>
</table>
## Arithmetic Operations Instructions

### Table C-17. Arithmetic Operations Instructions (Sheet 12 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC294 0000—0xC294 C1FF</td>
<td>1 1 0 0 0 0 1 0 1 0 0 1 0 0 1 0 0 0</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (FU, M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC314 0000—0xC314 C1FF</td>
<td>1 1 0 0 0 0 1 0 1 0 1 0 0 1 0 0 0</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IS, M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC394 0000—0xC394 C1FF</td>
<td>1 1 0 0 0 0 1 1 0 0 1 0 1 0 1 0 0 0</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IU, M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC254 0000—0xC254 C1FF</td>
<td>1 1 0 0 0 0 1 0 1 0 1 0 1 0 1 0 0 0</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (T, M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC2D4 0000—0xC2D4 C1FF</td>
<td>1 1 0 0 0 0 1 0 1 1 0 1 0 1 0 1 0 0 0</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (TFU, M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC234 0000—0xC234 C1FF</td>
<td>1 1 0 0 0 0 1 0 0 0 1 1 0 1 0 1 0 0 0</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (S2RND, M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC334 0000—0xC334 C1FF</td>
<td>1 1 0 0 0 0 1 1 0 0 1 1 0 1 0 1 0 0 0</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (ISS2, M)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

C-66 ADSP-BF53x/BF56x Blackfin Processor Programming Reference
### Instruction Opcodes

Table C-17. Arithmetic Operations Instructions (Sheet 13 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC374 0000—</td>
<td>1 1 0 0 0 0 1 1 0 1 1 1 1 0 1 0 0 0 Dreg half 0 0 0 0 0 0 Dest. src_reg_0 src_reg_1 Dreg # 0 Dreg # 1 Dreg #</td>
</tr>
<tr>
<td></td>
<td>0xC374 C1FF</td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:** When issuing compatible load/store instructions in parallel with a Multiply 16-Bit Operands instruction, add 0x0800 0000 to the Multiply 16-Bit Operands opcode.

Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IH, M)

<table>
<thead>
<tr>
<th>Multiply 16-Bit Operands</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0xC20C 0000—</td>
<td>1 1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 1 1 0 0 0 Dreg half 0 0 0 0 0 0 Dest. src_reg_0 src_reg_1 Dreg # 0 Dreg # 1 Dreg #</td>
</tr>
<tr>
<td></td>
<td>0xC20C C1FF</td>
<td></td>
</tr>
</tbody>
</table>

Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (FU)

<table>
<thead>
<tr>
<th>Multiply 16-Bit Operands</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0xC28C 0000—</td>
<td>1 1 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 1 1 1 0 0 0 Dreg half 0 0 0 0 0 0 Dest. src_reg_0 src_reg_1 Dreg # 0 Dreg # 1 Dreg #</td>
</tr>
<tr>
<td></td>
<td>0xC28C C1FF</td>
<td></td>
</tr>
</tbody>
</table>

Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (IS)

<table>
<thead>
<tr>
<th>Multiply 16-Bit Operands</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0xC30C 0000—</td>
<td>1 1 0 0 0 0 1 1 0 0 0 0 1 0 0 0 1 1 1 0 0 0 Dreg half 0 0 0 0 0 0 Dest. src_reg_0 src_reg_1 Dreg # 0 Dreg # 1 Dreg #</td>
</tr>
<tr>
<td></td>
<td>0xC30C C1FF</td>
<td></td>
</tr>
</tbody>
</table>

Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (S2RND)

<table>
<thead>
<tr>
<th>Multiply 16-Bit Operands</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0xC22C 0000—</td>
<td>1 1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 1 1 0 0 0 Dreg half 0 0 0 0 0 0 Dest. src_reg_0 src_reg_1 Dreg # 0 Dreg # 1 Dreg #</td>
</tr>
<tr>
<td></td>
<td>0xC22C C1FF</td>
<td></td>
</tr>
</tbody>
</table>

Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (ISS2)

<table>
<thead>
<tr>
<th>Multiply 16-Bit Operands</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0xC32C 0000—</td>
<td>1 1 0 0 0 0 1 1 0 0 1 0 0 0 1 1 1 0 0 0 Dreg half 0 0 0 0 0 0 Dest. src_reg_0 src_reg_1 Dreg # 0 Dreg # 1 Dreg #</td>
</tr>
<tr>
<td></td>
<td>0xC32C C1FF</td>
<td></td>
</tr>
</tbody>
</table>

Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (M)
### Arithmetic Operations Instructions

Table C-17. Arithmetic Operations Instructions (Sheet 14 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiply 16-Bit Operands</strong></td>
<td>0xC29C 0000—0xC29C C1FF</td>
<td>1 1 0 0 0 0 1 0 1 0 1 1 1 1 0 0</td>
</tr>
<tr>
<td>Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (FU, M)</td>
<td>Dreg half 0 0 0 0 0 0 Dest. Dreg # src_reg_0 src_reg_1 Dreg #</td>
<td></td>
</tr>
<tr>
<td><strong>Multiply 16-Bit Operands</strong></td>
<td>0xC31C 0000—0xC31C C1FF</td>
<td>1 1 0 0 0 0 1 0 1 0 0 1 1 1 1 0 0</td>
</tr>
<tr>
<td>Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (IS, M)</td>
<td>Dreg half 0 0 0 0 0 0 Dest. Dreg # src_reg_0 src_reg_1 Dreg #</td>
<td></td>
</tr>
<tr>
<td><strong>Multiply 16-Bit Operands</strong></td>
<td>0xC239 0000—0xC239 C1FF</td>
<td>1 1 0 0 0 0 1 0 1 0 1 1 1 1 0 0</td>
</tr>
<tr>
<td>Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (S2RND, M)</td>
<td>Dreg half 0 0 0 0 0 0 Dest. Dreg # src_reg_0 src_reg_1 Dreg #</td>
<td></td>
</tr>
<tr>
<td><strong>Multiply 16-Bit Operands</strong></td>
<td>0xC33C 0000—0xC33C C1FF</td>
<td>1 1 0 0 0 0 1 1 0 0 1 1 1 1 0 0</td>
</tr>
<tr>
<td>Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (ISS2, M)</td>
<td>Dreg half 0 0 0 0 0 0 Dest. Dreg # src_reg_0 src_reg_1 Dreg #</td>
<td></td>
</tr>
</tbody>
</table>

NOTE: When issuing compatible load/store instructions in parallel with a Multiply 16-Bit Operands instruction, add 0x0800 0000 to the Multiply 16-Bit Operands opcode.

Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (ISS2, M)

| **Multiply 32-Bit Operands** | 0x40C0—0x40FF | 0 1 0 0 0 0 0 1 1 Source Dreg # Dest. Dreg # |

Dreg * = Dreg
Table C-17. Arithmetic Operations Instructions (Sheet 15 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC003 0000—0xC003 063F</td>
<td>1110000000100001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00000D_reg_half 000src_reg_0 Dreg # 000src_reg_1 Dreg #</td>
</tr>
<tr>
<td>A0 = Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC083 0000—0xC083 063F</td>
<td>1110000010000011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00000D_reg_half 000src_reg_0 Dreg # 000src_reg_1 Dreg #</td>
</tr>
<tr>
<td>A0 = Dreg_lo_hi * Dreg_lo_hi (FU)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC103 0000—0xC103 063F</td>
<td>1110000100000011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00000D_reg_half 000src_reg_0 Dreg # 000src_reg_1 Dreg #</td>
</tr>
<tr>
<td>A0 = Dreg_lo_hi * Dreg_lo_hi (IS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC063 0000—0xC063 063F</td>
<td>1110000001100011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00000D_reg_half 000src_reg_0 Dreg # 000src_reg_1 Dreg #</td>
</tr>
<tr>
<td>A0 = Dreg_lo_hi * Dreg_lo_hi (W32)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: When issuing compatible load/store instructions in parallel with a Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Multiply and Multiply-Accumulate opcode.
A0 = Dreg_lo_hi * Dreg_lo_hi (W32)
### Table C-17. Arithmetic Operations Instructions (Sheet 16 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Accumulator</strong></td>
<td>0xC003 0800— 0xC003 0E3F</td>
<td>1100000000000011 00001D reg half 000src_reg_0 Dreg # 0 Dreg #</td>
</tr>
<tr>
<td>A0 += Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Accumulator</strong></td>
<td>0xC083 0800— 0xC083 0E3F</td>
<td>1100000010000011 00001D reg half 000src_reg_0 Dreg # 0 Dreg #</td>
</tr>
<tr>
<td>A0 += Dreg_lo_hi * Dreg_lo_hi (FU)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Accumulator</strong></td>
<td>0xC103 0800— 0xC103 0E3F</td>
<td>1100000100000011 00001D reg half 000src_reg_0 Dreg # 0 Dreg #</td>
</tr>
<tr>
<td>A0 += Dreg_lo_hi * Dreg_lo_hi (IS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Accumulator</strong></td>
<td>0xC063 0800— 0xC063 0E3F</td>
<td>1100000001100011 00010D reg half 000src_reg_0 Dreg # 0 Dreg #</td>
</tr>
<tr>
<td>A0 – = Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Accumulator</strong></td>
<td>0xC003 1000— 0xC003 163F</td>
<td>1100000000000011 00001D reg half 000src_reg_0 Dreg # 0 Dreg #</td>
</tr>
<tr>
<td>A0 – = Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Accumulator</strong></td>
<td>0xC083 1000— 0xC083 163F</td>
<td>1100000010000011 00001D reg half 000src_reg_0 Dreg # 0 Dreg #</td>
</tr>
<tr>
<td>A0 – = Dreg_lo_hi * Dreg_lo_hi (FU)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Accumulator</strong></td>
<td>0xC103 1000— 0xC103 163F</td>
<td>1100000100000011 00001D reg half 000src_reg_0 Dreg # 0 Dreg #</td>
</tr>
<tr>
<td>A0 – = Dreg_lo_hi * Dreg_lo_hi (IS)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:** When issuing compatible load/store instructions in parallel with a Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Multiply and Multiply-Accumulate opcode.

A0 += Dreg_lo_hi * Dreg_lo_hi (W32)

A0 – = Dreg_lo_hi * Dreg_lo_hi

A0 – = Dreg_lo_hi * Dreg_lo_hi (FU)

A0 – = Dreg_lo_hi * Dreg_lo_hi (IS)
Table C-17. Arithmetic Operations Instructions (Sheet 17 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC063 1000—0xC063 163F</td>
<td><img src="#" alt="Bin" /></td>
</tr>
</tbody>
</table>

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------------------------|----------------|-----|
| A0 = Dreg_lo_hi * Dreg_lo_hi (W32) | ![Bin](#) |

A1 = Dreg_lo_hi * Dreg_lo_hi

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC000 1800—0xC000 D83F</td>
<td><img src="#" alt="Bin" /></td>
</tr>
</tbody>
</table>

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------------------------|----------------|-----|
| A1 = Dreg_lo_hi * Dreg_lo_hi (FU) | ![Bin](#) |

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC080 1800—0xC080 D83F</td>
<td><img src="#" alt="Bin" /></td>
</tr>
</tbody>
</table>

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------------------------|----------------|-----|
| A1 = Dreg_lo_hi * Dreg_lo_hi (IS) | ![Bin](#) |

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC060 1800—0xC060 D83F</td>
<td><img src="#" alt="Bin" /></td>
</tr>
</tbody>
</table>

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------------------------|----------------|-----|
| A1 = Dreg_lo_hi * Dreg_lo_hi (W32) | ![Bin](#) |

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC010 1800—0xC010 D83F</td>
<td><img src="#" alt="Bin" /></td>
</tr>
</tbody>
</table>

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------------------------|----------------|-----|
| A1 = Dreg_lo_hi * Dreg_lo_hi (M) | ![Bin](#) |

NOTE: When issuing compatible load/store instructions in parallel with a Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Multiply and Multiply-Accumulate opcode.
### Arithmetic Operations Instructions

#### Table C-17. Arithmetic Operations Instructions (Sheet 18 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Multiply and Multiply-Accumulate to Accumulator</em></td>
<td>0xC070 1800—0xC070 D83F</td>
<td>1100000001110000 11 Dreg half 01100000 src_reg_0 0 src_reg_1 Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOTE: When issuing compatible load/store instructions in parallel with a Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Multiply and Multiply-Accumulate opcode.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 = Dreg_lo_hi * Dreg_lo_hi (W32, M)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Multiply and Multiply-Accumulate to Accumulator</em></td>
<td>0xC001 1800—0xC001 D83F</td>
<td>1100000000000001 11 Dreg half 01100000 src_reg_0 0 src_reg_1 Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 += Dreg_lo_hi * Dreg_lo_hi (FU)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Multiply and Multiply-Accumulate to Accumulator</em></td>
<td>0xC081 1800—0xC081 D83F</td>
<td>1100000100000001 11 Dreg half 01100000 src_reg_0 0 src_reg_1 Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 += Dreg_lo_hi * Dreg_lo_hi (IS)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Multiply and Multiply-Accumulate to Accumulator</em></td>
<td>0xC061 1800—0xC061 D83F</td>
<td>1100000001100001 11 Dreg half 01100000 src_reg_0 0 src_reg_1 Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 += Dreg_lo_hi * Dreg_lo_hi (W32)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Multiply and Multiply-Accumulate to Accumulator</em></td>
<td>0xC011 1800—0xC011 D83F</td>
<td>1100000000010001 11 Dreg half 01100000 src_reg_0 0 src_reg_1 Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 += Dreg_lo_hi * Dreg_lo_hi (M)
### Instruction Opcodes

#### Multiply and Multiply-Accumulate to Accumulator

- **Opcode**: 0xC071 1800—0xC071 D83F
- **Bin**: 1100000000000010
- **Description**: Dreg
- **Register**: half

**NOTE**: When issuing compatible load/store instructions in parallel with a Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Multiply and Multiply-Accumulate opcode.

#### Multiply and Multiply-Accumulate to Accumulator

- **Opcode**: 0xC002 1800—0xC002 D83F
- **Bin**: 1100000000000010
- **Description**: Dreg
- **Register**: half

#### Multiply and Multiply-Accumulate to Accumulator

- **Opcode**: 0xC082 1800—0xC082 D83F
- **Bin**: 1100000010000010
- **Description**: Dreg
- **Register**: half

#### Multiply and Multiply-Accumulate to Accumulator

- **Opcode**: 0xC102 1800—0xC102 D83F
- **Bin**: 1100000100000010
- **Description**: Dreg
- **Register**: half

#### Multiply and Multiply-Accumulate to Accumulator

- **Opcode**: 0xC062 1800—0xC062 D83F
- **Bin**: 1100000001100010
- **Description**: Dreg
- **Register**: half

#### Multiply and Multiply-Accumulate to Accumulator

- **Opcode**: 0xC022 1800—0xC022 D83F
- **Bin**: 1100000000010010
- **Description**: Dreg
- **Register**: half

---

**Table C-17. Arithmetic Operations Instructions (Sheet 19 of 44)**

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC071 1800—0xC071 D83F</td>
<td>1100000000000010</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC002 1800—0xC002 D83F</td>
<td>1100000000000010</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC082 1800—0xC082 D83F</td>
<td>1100000010000010</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC102 1800—0xC102 D83F</td>
<td>1100000100000010</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC062 1800—0xC062 D83F</td>
<td>1100000001100010</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC022 1800—0xC022 D83F</td>
<td>1100000000010010</td>
</tr>
</tbody>
</table>
### Table C-17. Arithmetic Operations Instructions (Sheet 20 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC072 1800—0xC072 D83F</td>
<td>1 1 1 0 0 0 0 0 0 1 1 1 0 0 0 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half: 0 1 1 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: When issuing compatible load/store instructions in parallel with a Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Multiply and Multiply-Accumulate opcode.

A1 = Dreg_lo_hi * Dreg_lo_hi (W32, M)

### Multiply and Multiply-Accumulate to Half Register

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC003 2000—0xC003 27FF</td>
<td>1 1 0 0 0 0 0 0 0 1 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half: 0 1 1 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi)

### Multiply and Multiply-Accumulate to Half Register

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC083 2000—0xC083 27FF</td>
<td>1 1 0 0 0 0 0 0 1 0 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half: 0 1 1 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (FU)

### Multiply and Multiply-Accumulate to Half Register

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC103 2000—0xC103 27FF</td>
<td>1 1 0 0 0 0 0 1 0 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half: 0 1 1 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (IS)
### Instruction Opcodes

#### Multiply and Multiply-Accumulate to Half Register

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC043 2000—0xC043 27FF</td>
<td>1100000001000011 00100D r e g half Dest. Dreg # src_reg_0 src_reg_1 Dreg #</td>
</tr>
<tr>
<td>Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (T)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC0C3 2000—0xC0C3 27FF</td>
<td>1100000011000011 00100D r e g half Dest. Dreg # src_reg_0 src_reg_1 Dreg #</td>
</tr>
<tr>
<td>Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (TFU)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC023 2000—0xC023 27FF</td>
<td>1100000000100011 00100D r e g half Dest. Dreg # src_reg_0 src_reg_1 Dreg #</td>
</tr>
<tr>
<td>Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (S2RND)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC123 2000—0xC123 27FF</td>
<td>1100000100100011 00100D r e g half Dest. Dreg # src_reg_0 src_reg_1 Dreg #</td>
</tr>
<tr>
<td>Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (ISS2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC163 2000—0xC163 27FF</td>
<td>1100000101100011 00100D r e g half Dest. Dreg # src_reg_0 src_reg_1 Dreg #</td>
</tr>
<tr>
<td>Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (IH)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:** When issuing compatible load/store instructions in parallel with a Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Multiply and Multiply-Accumulate opcode.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC003 2800—0xC003 2FFF</td>
<td>1100000000000011 00101D r e g half Dest. Dreg # src_reg_0 src_reg_1 Dreg #</td>
</tr>
<tr>
<td>Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C-17. Arithmetic Operations Instructions (Sheet 21 of 44)
## Arithmetic Operations Instructions

Table C-17. Arithmetic Operations Instructions (Sheet 22 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td>0xC083 2800—0xC083 2FFF</td>
<td>1 1 1 0 0 0 0 0 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 1 1</td>
</tr>
<tr>
<td>Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td>0xC103 2800—0xC103 2FFF</td>
<td>1 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td>0xC183 2800—0xC183 2FFF</td>
<td>1 1 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td>0xC043 2800—0xC043 2FFF</td>
<td>1 1 1 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td>0xC0C3 2800—0xC0C3 2FFF</td>
<td>1 1 1 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td>0xC023 2800—0xC023 2FFF</td>
<td>1 1 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td>0xC123 2800—0xC123 2FFF</td>
<td>1 1 1 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1</td>
</tr>
</tbody>
</table>
### Multiply and Multiply-Accumulate to Half Register

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td>0xC163 2800—0xC163 2FFF</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (IH)</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</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>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**NOTE:** When issuing compatible load/store instructions in parallel with a Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Multiply and Multiply-Accumulate opcode.

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td>0xC003 3000—0xC003 37FF</td>
<td>1</td>
<td>1</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>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Dreg_lo = (A0 – = Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</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>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td>0xC103 3000—0xC103 37FF</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Dreg_lo = (A0 – = Dreg_lo_hi * Dreg_lo_hi) (IS)</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</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>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td>0xC183 3000—0xC183 37FF</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Dreg_lo = (A0 – = Dreg_lo_hi * Dreg_lo_hi) (IU)</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</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>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td>0xC0C3 3000—0xC0C3 37FF</td>
<td>1</td>
<td>1</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>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</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>Dreg_lo = (A0 – = Dreg_lo_hi * Dreg_lo_hi) (TFU)</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</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>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</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>
</tbody>
</table>
### Table C-17. Arithmetic Operations Instructions (Sheet 24 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC023 3000—0xC023 37FF</td>
<td>0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1</td>
</tr>
<tr>
<td>Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (S2RND)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC023 3000—0xC023 37FF</td>
<td>0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1</td>
</tr>
<tr>
<td>Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (ISS2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC004 1800—0xC004 D9FF</td>
<td>0 1 1 0 0 0 0 0 0 0 0 0 0 0 1 0</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IH)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC084 1800—0xC084 D9FF</td>
<td>0 1 1 0 0 0 0 0 0 0 1 0 0 0 1 0</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC104 1800—0xC104 D9FF</td>
<td>0 1 1 0 0 0 0 0 0 0 1 0 0 0 1 0</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC184 1800—0xC184 D9FF</td>
<td>0 1 1 0 0 0 0 0 0 0 1 0 0 1 0 0</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IU)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: When issuing compatible load/store instructions in parallel with a Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Multiply and Multiply-Accumulate opcode.
### Instruction Opcodes

#### Table C-17. Arithmetic Operations Instructions (Sheet 25 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC044 1800—0xC044 D9FF</td>
<td>1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

\[ \text{Dreg}_\text{hi} = (A1 = \text{Dreg}_\text{lo}_\text{hi} \times \text{Dreg}_\text{lo}_\text{hi}) \text{ (T)} \]

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC0C4 1800—0xC0C4 D9FF</td>
<td>1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

\[ \text{Dreg}_\text{hi} = (A1 = \text{Dreg}_\text{lo}_\text{hi} \times \text{Dreg}_\text{lo}_\text{hi}) \text{ (TFU)} \]

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC024 1800—0xC024 D9FF</td>
<td>1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

\[ \text{Dreg}_\text{hi} = (A1 = \text{Dreg}_\text{lo}_\text{hi} \times \text{Dreg}_\text{lo}_\text{hi}) \text{ (S2RND)} \]

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC124 1800—0xC124 D9FF</td>
<td>1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

\[ \text{Dreg}_\text{hi} = (A1 = \text{Dreg}_\text{lo}_\text{hi} \times \text{Dreg}_\text{lo}_\text{hi}) \text{ (ISS2)} \]

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC164 1800—0xC164 D9FF</td>
<td>1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

\[ \text{Dreg}_\text{hi} = (A1 = \text{Dreg}_\text{lo}_\text{hi} \times \text{Dreg}_\text{lo}_\text{hi}) \text{ (IH)} \]

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC014 1800—0xC014 D9FF</td>
<td>1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

\[ \text{Dreg}_\text{hi} = (A1 = \text{Dreg}_\text{lo}_\text{hi} \times \text{Dreg}_\text{lo}_\text{hi}) \text{ (M)} \]

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC094 1800—0xC094 D9FF</td>
<td>1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

\[ \text{Dreg}_\text{hi} = (A1 = \text{Dreg}_\text{lo}_\text{hi} \times \text{Dreg}_\text{lo}_\text{hi}) \text{ (FU, M)} \]
### Table C-17. Arithmetic Operations Instructions (Sheet 26 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC114 1800—0xC114 D9FF</td>
<td>111000000000000011000100100</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi)</td>
<td>(IS, M)</td>
<td>Dest. Dreg # src_reg_0 Dreg # src_reg_1 Dreg #</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC194 1800—0xC194 D9FF</td>
<td>111000000000000011000000100</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi)</td>
<td>(IU, M)</td>
<td>Dest. Dreg # src_reg_0 Dreg # src_reg_1 Dreg #</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC054 1800—0xC054 D9FF</td>
<td>111000000000000011000000100</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi)</td>
<td>(T, M)</td>
<td>Dest. Dreg # src_reg_0 Dreg # src_reg_1 Dreg #</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC0D4 1800—0xC0D4 D9FF</td>
<td>111000000000000011000000100</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi)</td>
<td>(TFU, M)</td>
<td>Dest. Dreg # src_reg_0 Dreg # src_reg_1 Dreg #</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC034 1800—0xC034 D9FF</td>
<td>111000000000000011000000100</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi)</td>
<td>(S2RND, M)</td>
<td>Dest. Dreg # src_reg_0 Dreg # src_reg_1 Dreg #</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC134 1800—0xC134 D9FF</td>
<td>111000000000000011000000100</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi)</td>
<td>(ISS2, M)</td>
<td>Dest. Dreg # src_reg_0 Dreg # src_reg_1 Dreg #</td>
</tr>
</tbody>
</table>

**NOTE:** When issuing compatible load/store instructions in parallel with a Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Multiply and Multiply-Accumulate opcode.

Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IH, M)
### Instruction Opcodes

#### Multiply and Multiply-Accumulate to Half Register

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC005 1800—0xC005 D9FF</td>
<td>01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D</td>
</tr>
<tr>
<td>Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC085 1800—0xC085 D9FF</td>
<td>01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D</td>
</tr>
<tr>
<td>Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC105 1800—0xC105 D9FF</td>
<td>01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D</td>
</tr>
<tr>
<td>Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IS)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC185 1800—0xC185 D9FF</td>
<td>01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D</td>
</tr>
<tr>
<td>Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IU)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC045 1800—0xC045 D9FF</td>
<td>01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D</td>
</tr>
<tr>
<td>Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (T)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC0C5 1800—0xC0C5 D9FF</td>
<td>01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D</td>
</tr>
<tr>
<td>Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (TFU)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC025 1800—0xC025 D9FF</td>
<td>01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D 01100D</td>
</tr>
<tr>
<td>Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (S2RND)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table C-17. Arithmetic Operations Instructions (Sheet 28 of 44)

| Instruction and Version | Opcode Range   | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------------------------|----------------|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| Multiply and Multiply-Accumulate to Half Register | 0xC125 1800—0xC125 D9FF | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | |
| | | Dreg half | 0 | 1 | 1 | 0 | 0 | Dest. Dreg # | src_reg_0 | src_reg_1 | Dreg # |
| Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (ISS2) | | |

| Multiply and Multiply-Accumulate to Half Register | 0xC165 1800—0xC165 D9FF | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | |
| | | Dreg half | 0 | 1 | 1 | 0 | 0 | Dest. Dreg # | src_reg_0 | src_reg_1 | Dreg # |
| Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IH) | | |

| Multiply and Multiply-Accumulate to Half Register | 0xC015 1800—0xC015 D9FF | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | |
| | | Dreg half | 0 | 1 | 1 | 0 | 0 | Dest. Dreg # | src_reg_0 | src_reg_1 | Dreg # |
| Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (M) | | |

| Multiply and Multiply-Accumulate to Half Register | 0xC095 1800—0xC095 D9FF | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | |
| | | Dreg half | 0 | 1 | 1 | 0 | 0 | Dest. Dreg # | src_reg_0 | src_reg_1 | Dreg # |
| Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (FU, M) | | |

| Multiply and Multiply-Accumulate to Half Register | 0xC115 1800—0xC115 D9FF | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | |
| | | Dreg half | 0 | 1 | 1 | 0 | 0 | Dest. Dreg # | src_reg_0 | src_reg_1 | Dreg # |
| Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IS, M) | | |

| Multiply and Multiply-Accumulate to Half Register | 0xC195 1800—0xC195 D9FF | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | |
| | | Dreg half | 0 | 1 | 1 | 0 | 0 | Dest. Dreg # | src_reg_0 | src_reg_1 | Dreg # |
| Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IU, M) | | |

| Multiply and Multiply-Accumulate to Half Register | 0xC055 1800—0xC055 D9FF | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | |
| | | Dreg half | 0 | 1 | 1 | 0 | 0 | Dest. Dreg # | src_reg_0 | src_reg_1 | Dreg # |
| Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (T, M) | | |
### Instruction Opcodes

#### Table C-17. Arithmetic Operations Instructions (Sheet 29 of 44)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC0D5 1800—0xC0D5 D9FF</td>
<td>0111000100100001011001000011001001110010010010010100101100</td>
</tr>
<tr>
<td></td>
<td>Dreg_hi = (A1 + Dreg_lo_hi * Dreg_lo_hi) (TFU, M)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC035 1800—0xC035 D9FF</td>
<td>011100000011001101110010010010010100101100</td>
</tr>
<tr>
<td></td>
<td>Dreg_hi = (A1 + Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC135 1800—0xC135 D9FF</td>
<td>011100010011011001110010010010010100101100</td>
</tr>
<tr>
<td></td>
<td>Dreg_hi = (A1 + Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC175 1800—0xC175 D9FF</td>
<td>011100010111011001110010010010010100101100</td>
</tr>
<tr>
<td></td>
<td>Dreg_hi = (A1 + Dreg_lo_hi * Dreg_lo_hi) (IH, M)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC006 1800—0xC006 D9FF</td>
<td>0111000000001100100000110010010010100101100</td>
</tr>
<tr>
<td></td>
<td>Dreg_hi = (A1 + Dreg_lo_hi * Dreg_lo_hi) (IH, M)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC086 1800—0xC086 D9FF</td>
<td>0111000010001100100000110010010010100101100</td>
</tr>
<tr>
<td></td>
<td>Dreg_hi = (A1 + Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC106 1800—0xC106 D9FF</td>
<td>0111000100001100100000110010010010100101100</td>
</tr>
<tr>
<td></td>
<td>Dreg_hi = (A1 + Dreg_lo_hi * Dreg_lo_hi) (IS)</td>
<td></td>
</tr>
</tbody>
</table>

NOTE: When issuing compatible load/store instructions in parallel with a Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Multiply and Multiply-Accumulate opcode.

Dreg_hi = (A1 + Dreg_lo_hi * Dreg_lo_hi) (IH, M)
### Table C-17. Arithmetic Operations Instructions (Sheet 30 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC186 1800—0xC186 D9FF</td>
<td>1110000110000110</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IU)</td>
<td></td>
<td>01100Dest. Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 src_reg_ Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 src_reg_</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 src_reg_</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC046 1800—0xC046 D9FF</td>
<td>1110000001000110</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (T)</td>
<td></td>
<td>01100Dest. Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 src_reg_ Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 src_reg_</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 src_reg_</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC0C6 1800—0xC0C6 D9FF</td>
<td>1110000011000110</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (TFU)</td>
<td></td>
<td>01100Dest. Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 src_reg_ Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 src_reg_</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 src_reg_</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC026 1800—0xC026 D9FF</td>
<td>1110000000100110</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (S2RND)</td>
<td></td>
<td>01100Dest. Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 src_reg_ Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 src_reg_</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 src_reg_</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC126 1800—0xC126 D9FF</td>
<td>1110000100100110</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (ISS2)</td>
<td></td>
<td>01100Dest. Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 src_reg_ Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 src_reg_</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 src_reg_</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC166 1800—0xC166 D9FF</td>
<td>1110000101100110</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IH)</td>
<td></td>
<td>01100Dest. Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 src_reg_ Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 src_reg_</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 src_reg_</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC016 1800—0xC016 D9FF</td>
<td>1110000000010110</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (M)</td>
<td></td>
<td>01100Dest. Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 src_reg_ Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 src_reg_</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 src_reg_</td>
</tr>
</tbody>
</table>
### Instruction Opcodes

Table C-17. Arithmetic Operations Instructions (Sheet 31 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC096 1800—0xC096 D9FF</td>
<td>1 1 0 0 0 0 0 0 0 1 0 0 1 0 1 1 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half 0 1 1 0 0 Dest. Dreg # src_reg_0 src_reg_1 src_reg_2 src_reg_3</td>
</tr>
<tr>
<td>Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (FU, M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC116 1800—0xC116 D9FF</td>
<td>1 1 0 0 0 0 0 1 1 0 0 0 1 0 1 1 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half 0 1 1 0 0 Dest. Dreg # src_reg_0 src_reg_1 src_reg_2 src_reg_3</td>
</tr>
<tr>
<td>Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (IS, M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC196 1800—0xC196 D9FF</td>
<td>1 1 0 0 0 0 1 1 0 0 0 1 0 1 1 1 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half 0 1 1 0 0 Dest. Dreg # src_reg_0 src_reg_1 src_reg_2 src_reg_3</td>
</tr>
<tr>
<td>Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (IU, M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC056 1800—0xC056 D9FF</td>
<td>1 1 0 0 0 0 0 0 0 1 1 0 0 1 1 1 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half 0 1 1 0 0 Dest. Dreg # src_reg_0 src_reg_1 src_reg_2 src_reg_3</td>
</tr>
<tr>
<td>Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (T, M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC0D6 1800—0xC0D6 D9FF</td>
<td>1 1 0 0 0 0 0 1 1 0 0 0 1 0 1 1 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half 0 1 1 0 0 Dest. Dreg # src_reg_0 src_reg_1 src_reg_2 src_reg_3</td>
</tr>
<tr>
<td>Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (TFU, M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC036 1800—0xC036 D9FF</td>
<td>1 1 0 0 0 0 0 0 0 1 1 0 1 0 1 1 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half 0 1 1 0 0 Dest. Dreg # src_reg_0 src_reg_1 src_reg_2 src_reg_3</td>
</tr>
<tr>
<td>Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC136 1800—0xC136 D9FF</td>
<td>1 1 0 0 0 0 0 1 1 0 0 1 1 0 1 1 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half 0 1 1 0 0 Dest. Dreg # src_reg_0 src_reg_1 src_reg_2 src_reg_3</td>
</tr>
<tr>
<td>Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table C-17. Arithmetic Operations Instructions (Sheet 32 of 44)

### Multiply and Multiply-Accumulate to Half Register

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC176 1800—0xC176 D9FF</td>
<td>1 1 0 0 0 0 0 1 0 1 1 1 0 1 1 1 0</td>
</tr>
<tr>
<td>Dreg half</td>
<td>0 1 1 0 0 0 1 0 Dest. src_reg_0 Dreg # src_reg_1 Dreg #</td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:** When issuing compatible load/store instructions in parallel with a Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Multiply and Multiply-Accumulate opcode.

\[
Dreg_{hi} = (A1 = Dreg_{lo} \times Dreg_{hi}) \text{ (IH, M)}
\]

**LEGEND:**
- Dreg half determines which halves of the input operand registers to use.
- Dreg_lo * Dreg_lo
- Dreg_lo * Dreg_hi
- Dreg_hi * Dreg_lo
- Dreg_hi * Dreg_hi

Dest. Dreg # encodes the destination Data Register.

\[
src\_reg\_0 \text{ Dreg # encodes the input operand register to the left of the "*" operand.}
\]

\[
src\_reg\_1 \text{ Dreg # encodes the input operand register to the right of the "*" operand.}
\]

**NOTE:** When issuing compatible load/store instructions in parallel with a Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Multiply and Multiply-Accumulate opcode.

### Multiply and Multiply-Accumulate to Data Register

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC00D 0000—0xC00D 07FF</td>
<td>1 1 0 0 0 0 0 0 0 0 1 0 1 1 1</td>
</tr>
<tr>
<td>Dreg even</td>
<td>0 0 0 0 0 Dreg half Dest. src_reg_0 Dreg # src_reg_1 Dreg #</td>
<td></td>
</tr>
</tbody>
</table>

\[
Dreg_{even} = (A0 = Dreg_{lo} \times Dreg_{lo})
\]

### Multiply and Multiply-Accumulate to Data Register

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC08D 0000—0xC08D 07FF</td>
<td>1 1 0 0 0 0 0 1 0 0 0 1 0 1 1 1</td>
</tr>
<tr>
<td>Dreg even</td>
<td>0 0 0 0 0 Dreg half Dest. src_reg_0 Dreg # src_reg_1 Dreg #</td>
<td></td>
</tr>
</tbody>
</table>

\[
Dreg_{even} = (A0 = Dreg_{lo} \times Dreg_{lo}) \text{ (FU)}
\]
Instruction Opcodes

### Multiply and Multiply-Accumulate to Data Register

| Instruction and Version | Opcode Range | OPCODE | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------------------------|--------------|--------|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| Multiply and Multiply-Accumulate to Data Register | 0xC10D 0000—0xC10D 07FF | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| | | 0 | 0 | 0 | 0 | 0 | 0 | Dreg half | Dest. | src_reg_ | src_reg_ | 0 | Dreg # | 1 | Dreg # |

Dreg_even = \((A_0 = Dreg_{lo\_hi} \times Dreg_{lo\_hi})\) (IS)

| Multiply and Multiply-Accumulate to Data Register | 0xC02D 0000—0xC02D 07FF | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| | | 0 | 0 | 0 | 0 | 0 | 0 | Dreg half | Dest. | src_reg_ | src_reg_ | 0 | Dreg # | 1 | Dreg # |

Dreg_even = \((A_0 = Dreg_{lo\_hi} \times Dreg_{lo\_hi})\) (S2RND)

| Multiply and Multiply-Accumulate to Data Register | 0xC12D 0000—0xC12D 07FF | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| | | 0 | 0 | 0 | 0 | 0 | 0 | Dreg half | Dest. | src_reg_ | src_reg_ | 0 | Dreg # | 1 | Dreg # |

NOTE: When issuing compatible load/store instructions in parallel with a Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Multiply and Multiply-Accumulate opcode.

Dreg_even = \((A_0 = Dreg_{lo\_hi} \times Dreg_{lo\_hi})\) (ISS2)

| Multiply and Multiply-Accumulate to Data Register | 0xC00D 0800—0xC00D 0FFF | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| | | 0 | 0 | 0 | 0 | 0 | 1 | Dreg half | Dest. | src_reg_ | src_reg_ | 0 | Dreg # | 1 | Dreg # |

| Multiply and Multiply-Accumulate to Data Register | 0xC08D 0800—0xC08D 0FFF | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| | | 0 | 0 | 0 | 0 | 1 | Dreg half | Dest. | src_reg_ | src_reg_ | 0 | Dreg # | 1 | Dreg # |

Dreg_even = \((A_0 += Dreg_{lo\_hi} \times Dreg_{lo\_hi})\) (FU)

| Multiply and Multiply-Accumulate to Data Register | 0xC10D 0800—0xC10D 0FFF | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| | | 0 | 0 | 0 | 0 | 1 | Dreg half | Dest. | src_reg_ | src_reg_ | 0 | Dreg # | 1 | Dreg # |

Dreg_even = \((A_0 += Dreg_{lo\_hi} \times Dreg_{lo\_hi})\) (IS)

| Multiply and Multiply-Accumulate to Data Register | 0xC02D 0800—0xC02D 0FFF | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| | | 0 | 0 | 0 | 0 | 1 | Dreg half | Dest. | src_reg_ | src_reg_ | 0 | Dreg # | 1 | Dreg # |

Dreg_even = \((A_0 += Dreg_{lo\_hi} \times Dreg_{lo\_hi})\) (S2RND)

---

ADSP-BF53x/BF56x Blackfin Processor Programming Reference C-87
### Table C-17. Arithmetic Operations Instructions (Sheet 34 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC12D 0800—0xC12D 0FFF</td>
<td>1 1 0 0 0 0 0 1 0 1 0 1 0 1 0 1 1</td>
</tr>
<tr>
<td>Dreg_even = (A0 += Dreg_lo_hi * Dreg_lo_hi) (ISS2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC00D 1000—0xC00D 17FF</td>
<td>1 1 0 0 0 0 0 0 0 1 0 1 0 1 1</td>
</tr>
<tr>
<td>Dreg_even = (A0 – = Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC08D 1000—0xC08D 17FF</td>
<td>1 1 0 0 0 0 0 0 1 0 0 0 1 0 1 1</td>
</tr>
<tr>
<td>Dreg_even = (A0 – = Dreg_lo_hi * Dreg_lo_hi) (IS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC10D 1000—0xC10D 17FF</td>
<td>1 1 0 0 0 0 0 1 0 1 0 0 1 0 1 1</td>
</tr>
<tr>
<td>Dreg_even = (A0 – = Dreg_lo_hi * Dreg_lo_hi) (S2RND)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC12D 1000—0xC12D 17FF</td>
<td>1 1 0 0 0 0 0 1 0 1 0 1 0 1 1</td>
</tr>
<tr>
<td>Dreg_even = (A0 – = Dreg_lo_hi * Dreg_lo_hi) (ISS2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Instruction Opcodes

Table C-17. Arithmetic Operations Instructions (Sheet 35 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC008 1800—0xC008 D9FF</td>
<td>1 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC088 1800—0xC088 D9FF</td>
<td>1 1 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC108 1800—0xC108 D9FF</td>
<td>1 1 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC028 1800—0xC028 D9FF</td>
<td>1 1 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (S2RND)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC128 1800—0xC128 D9FF</td>
<td>1 1 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (ISS2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC018 1800—0xC018 D9FF</td>
<td>1 1 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC098 1800—0xC098 D9FF</td>
<td>1 1 0 0 0 0 0 1 0 0 0 1 1 0 0 0 0 0</td>
</tr>
<tr>
<td>Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (FU, M)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Arithmetic Operations Instructions

#### Table C-17. Arithmetic Operations Instructions (Sheet 36 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC118 1800—0xC118 D9FF</td>
<td>1110000100011000 01100D src_reg_0 srcreg_1 Dreg # 0 Dreg #</td>
</tr>
<tr>
<td>Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IS, M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC038 1800—0xC038 D9FF</td>
<td>1100000000111000 01100D src_reg_0 srcreg_1 Dreg # 0 Dreg #</td>
</tr>
<tr>
<td>Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC138 1800—0xC138 D9FF</td>
<td>1100000100111000 01100D src_reg_0 srcreg_1 Dreg # 0 Dreg #</td>
</tr>
<tr>
<td>Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC009 1800—0xC009 D9FF</td>
<td>1110000000001001 01100D src_reg_0 srcreg_1 Dreg # 0 Dreg #</td>
</tr>
<tr>
<td>Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC089 1800—0xC089 D9FF</td>
<td>1110000010001001 01100D src_reg_0 srcreg_1 Dreg # 0 Dreg #</td>
</tr>
<tr>
<td>Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC109 1800—0xC109 D9FF</td>
<td>1110000100001001 01100D src_reg_0 srcreg_1 Dreg # 0 Dreg #</td>
</tr>
<tr>
<td>Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC029 1800—0xC029 D9FF</td>
<td>1110000000101001 01100D src_reg_0 srcreg_1 Dreg # 0 Dreg #</td>
</tr>
<tr>
<td>Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (S2RND)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: When issuing compatible load/store instructions in parallel with a Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Multiply and Multiply-Accumulate opcode.
### Instruction Opcodes

Table C-17. Arithmetic Operations Instructions (Sheet 37 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td>0xC129 1800—0xC129 D9FF</td>
<td>Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (ISS2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td>0xC019 1800—0xC019 D9FF</td>
<td>Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (M)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td>0xC099 1800—0xC099 D9FF</td>
<td>Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (FU, M)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td>0xC119 1800—0xC119 D9FF</td>
<td>Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IS, M)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td>0xC039 1800—0xC039 D9FF</td>
<td>Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td>0xC139 1800—0xC139 D9FF</td>
<td>Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td>0xC00A 1800—0xC00A D9FF</td>
<td>Dreg_odd = (A1 –= Dreg_lo_hi * Dreg_lo_hi)</td>
</tr>
</tbody>
</table>

**NOTE:** When issuing compatible load/store instructions in parallel with a Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Multiply and Multiply-Accumulate opcode.

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td>0xC00A 1800—0xC00A D9FF</td>
<td>Dreg_odd = (A1 –= Dreg_lo_hi * Dreg_lo_hi)</td>
</tr>
</tbody>
</table>

---

ADSP-BF53x/BF56x Blackfin Processor Programming Reference C-91
### Arithmetic Operations Instructions

#### Table C-17. Arithmetic Operations Instructions (Sheet 38 of 44)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Range</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td>0xC08A 1800—0xC08A D9FF</td>
<td>![1100000010001010]</td>
</tr>
<tr>
<td>Dreg_odd = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td></td>
<td>![Dreg half 0 1 1 0 0 Dest. Dreg # src_reg_0 Dreg # src_reg_1 Dreg #]</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td>0xC10A 1800—0xC10A D9FF</td>
<td>![1100000100001010]</td>
</tr>
<tr>
<td>Dreg_odd = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (IS)</td>
<td></td>
<td>![Dreg half 0 1 1 0 0 Dest. Dreg # src_reg_0 Dreg # src_reg_1 Dreg #]</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td>0xC02A 1800—0xC02A D9FF</td>
<td>![1100000000101010]</td>
</tr>
<tr>
<td>Dreg_odd = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (S2RND)</td>
<td></td>
<td>![Dreg half 0 1 1 0 0 Dest. Dreg # src_reg_0 Dreg # src_reg_1 Dreg #]</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td>0xC12A 1800—0xC12A D9FF</td>
<td>![1100000100101010]</td>
</tr>
<tr>
<td>Dreg_odd = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (ISS2)</td>
<td></td>
<td>![Dreg half 0 1 1 0 0 Dest. Dreg # src_reg_0 Dreg # src_reg_1 Dreg #]</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td>0xC01A 1800—0xC01A D9FF</td>
<td>![1100000000011010]</td>
</tr>
<tr>
<td>Dreg_odd = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (M)</td>
<td></td>
<td>![Dreg half 0 1 1 0 0 Dest. Dreg # src_reg_0 Dreg # src_reg_1 Dreg #]</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td>0xC09A 1800—0xC09A D9FF</td>
<td>![1100000010011010]</td>
</tr>
<tr>
<td>Dreg_odd = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (FU, M)</td>
<td></td>
<td>![Dreg half 0 1 1 0 0 Dest. Dreg # src_reg_0 Dreg # src_reg_1 Dreg #]</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td>0xC11A 1800—0xC11A D9FF</td>
<td>![1100000100011010]</td>
</tr>
<tr>
<td>Dreg_odd = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (IS, M)</td>
<td></td>
<td>![Dreg half 0 1 1 0 0 Dest. Dreg # src_reg_0 Dreg # src_reg_1 Dreg #]</td>
</tr>
</tbody>
</table>
### Multiply and Multiply-Accumulate to Data Register

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC03A 1800—0xC03A D9FF</td>
<td>1100000000111010</td>
</tr>
</tbody>
</table>

\[
\text{Dreg\_odd} = (A_1 - D\text{reg\_lo}_hi \times D\text{reg\_lo}_hi) \text{ (S2RND, M)}
\]

Multiply and Multiply-Accumulate to Data Register

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC13A 1800—0xC13A D9FF</td>
<td>1100000100111010</td>
</tr>
</tbody>
</table>

\[
\text{Dreg\_odd} = (A_1 - D\text{reg\_lo}_hi \times D\text{reg\_lo}_hi) \text{ (ISS2, M)}
\]

**NOTE:** When issuing compatible load/store instructions in parallel with a Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Multiply and Multiply-Accumulate opcode.

\[
\text{Dreg\_odd} = (A_1 - D\text{reg\_lo}_hi \times D\text{reg\_lo}_hi)
\]

### LEGEND:
- **Dreg half** determines which halves of the input operand registers to use.
- **Dest. Dreg #** encodes the destination Data Register.
- **src_reg_0 Dreg #** encodes the input operand register to the left of the “*” operand.
- **src_reg_1 Dreg #** encodes the input operand register to the right of the “*” operand.

### Negate (Two’s-Complement)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negate (Two’s-Complement)</td>
<td>0xC407 C000—0xC407 CFC0</td>
<td>1100010xxx000111</td>
</tr>
</tbody>
</table>

\[
\text{Dreg} = - \text{Dreg (NS)}
\]
## Arithmetic Operations Instructions

Table C-17. Arithmetic Operations Instructions (Sheet 40 of 44)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0xC407 E000—0xC407 EFC0</td>
<td>1 1 1 1 0 0 0 1 0 x x x 0 1 1 1 1</td>
</tr>
<tr>
<td></td>
<td>Dreg = – Dreg (S)</td>
<td></td>
</tr>
<tr>
<td>Negate (Two's-Complement)</td>
<td>0xC40E 003F</td>
<td>1 1 1 0 0 0 1 0 x x x 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td></td>
<td>A0 = – A0</td>
<td></td>
</tr>
<tr>
<td>Negate (Two's-Complement)</td>
<td>0xC40E 403F</td>
<td>1 1 0 0 0 1 0 x x x 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td></td>
<td>A0 = – A1</td>
<td></td>
</tr>
<tr>
<td>Negate (Two's-Complement)</td>
<td>0xC42E 003F</td>
<td>1 1 0 0 0 1 0 x x x 1 0 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td></td>
<td>A1 = – A0</td>
<td></td>
</tr>
<tr>
<td>Negate (Two's-Complement)</td>
<td>0xC42E 403F</td>
<td>1 1 0 0 0 1 0 x x x 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td></td>
<td>A1 = – A1</td>
<td></td>
</tr>
<tr>
<td>Negate (Two's-Complement)</td>
<td>0xC40E C03F</td>
<td>1 1 1 0 0 0 1 0 x x x 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1</td>
</tr>
<tr>
<td></td>
<td>A1 = – A1, A0 = – A0</td>
<td></td>
</tr>
<tr>
<td>Round to Half Word</td>
<td>0xC40C C000—0xC40C CE38</td>
<td>1 1 0 0 0 1 0 x x x 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td></td>
<td>Dreg_lo = Dreg (RND)</td>
<td></td>
</tr>
<tr>
<td>Round to Half Word</td>
<td>0xC42C C000—0xC42C CE38</td>
<td>1 1 0 0 0 1 0 x x x 1 0 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td></td>
<td>Dreg_hi = Dreg (RND)</td>
<td></td>
</tr>
</tbody>
</table>
### Instruction Opcodes

Table C-17. Arithmetic Operations Instructions (Sheet 41 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saturate</td>
<td>0xC408 203F</td>
<td>1 1 0 0 0 1 0 x x 0 0 1 0 0 0</td>
</tr>
<tr>
<td>Saturate</td>
<td>0xC408 603F</td>
<td>1 1 1 0 0 1 0 x x 0 0 1 0 0 0</td>
</tr>
<tr>
<td>Saturate</td>
<td>0xC408 A03F</td>
<td>1 1 0 0 0 1 0 x x 0 0 1 0 0 0</td>
</tr>
<tr>
<td>A0 = A0 (S)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1 = A1 (S)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sign Bit</td>
<td>0xC605 0000—</td>
<td>1 1 0 0 0 1 1 0 x x 0 0 1 0 0 1</td>
</tr>
<tr>
<td></td>
<td>0xC605 0E07</td>
<td>0 0 0 0 1 0 x x 0 0 Source Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = SIGNBITS Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sign Bit</td>
<td>0xC605 4000—</td>
<td>1 1 0 0 0 1 1 0 x x 0 0 1 0 0 1</td>
</tr>
<tr>
<td></td>
<td>0xC605 4E07</td>
<td>0 1 0 0 1 0 x x 0 0 Source Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = SIGNBITS Dreg_lo</td>
<td></td>
<td>1 1 0 0 0 1 1 0 x x 0 0 1 0 0 1</td>
</tr>
<tr>
<td>Dreg_lo = SIGNBITS Dreg_lo</td>
<td></td>
<td>0 0 x x x 0 Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = SIGNBITS Dreg_lo</td>
<td></td>
<td>1 1 0 0 0 1 1 0 x x 0 0 1 0 0 1</td>
</tr>
<tr>
<td>Dreg_lo = SIGNBITS Dreg_lo</td>
<td></td>
<td>0 0 x x x 0 Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = SIGNBITS Dreg_lo</td>
<td></td>
<td>1 1 0 0 0 1 1 0 x x 0 0 1 0 0 1</td>
</tr>
<tr>
<td>Dreg_lo = SIGNBITS Dreg_lo</td>
<td></td>
<td>0 0 x x x 0 Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = SIGNBITS A0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sign Bit</td>
<td>0xC606 0000—</td>
<td>1 1 0 0 0 1 1 0 x x 0 0 1 0 0 1</td>
</tr>
<tr>
<td></td>
<td>0xC606 0E00</td>
<td>0 0 x x x 0 Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = SIGNBITS A0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sign Bit</td>
<td>0xC606 4000—</td>
<td>1 1 0 0 0 1 1 0 x x 0 0 1 0 0 1</td>
</tr>
<tr>
<td></td>
<td>0xC606 4E00</td>
<td>0 0 x x x 0 Dreg #</td>
</tr>
</tbody>
</table>
### Arithmetic Operations Instructions

Table C-17. Arithmetic Operations Instructions (Sheet 42 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtract</td>
<td>0x5200—0x53FF</td>
<td>0 1 0 1 0 0 1 0 1 0 0 1 0 0 1 0 0 1 0 0</td>
</tr>
<tr>
<td>Subtract</td>
<td>0xC404 4000—0xC404 4E3F</td>
<td>1 1 0 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0</td>
</tr>
<tr>
<td>Subtract</td>
<td>0xC404 6000—0xC404 6E3F</td>
<td>1 1 0 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0</td>
</tr>
<tr>
<td>Subtract</td>
<td>0xC403 0000—0xC403 0E3F</td>
<td>1 1 0 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0</td>
</tr>
<tr>
<td>Subtract</td>
<td>0xC403 4000—0xC403 4E3F</td>
<td>1 1 0 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0</td>
</tr>
<tr>
<td>Subtract</td>
<td>0xC403 8000—0xC403 8E3F</td>
<td>1 1 0 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0</td>
</tr>
<tr>
<td>Subtract</td>
<td>0xC403 C000—0xC403 CE3F</td>
<td>1 1 0 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0</td>
</tr>
<tr>
<td>Subtract</td>
<td>0xC423 0000—0xC423 0E3F</td>
<td>1 1 0 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0</td>
</tr>
</tbody>
</table>

Dreg = Dreg – Dreg

Dreg = Dreg – Dreg (NS)

Dreg = Dreg – Dreg (S)

Dreg_lo = Dreg_lo – Dreg_lo (NS)

Dreg_lo = Dreg_lo – Dreg_hi (NS)

Dreg_lo = Dreg_hi – Dreg_lo (NS)

Dreg_lo = Dreg_hi – Dreg_hi (NS)

Dreg_hi = Dreg_lo – Dreg_lo (NS)
### Instruction Opcodes

Table C-17. Arithmetic Operations Instructions (Sheet 43 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtract</td>
<td>0xC423 0000—</td>
<td>1 1 0 0 0 0 1 0 x x 1 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td>0xC423 4E3F</td>
<td></td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo − Dreg_hi (NS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtract</td>
<td>0xC423 8000—</td>
<td>1 1 0 0 0 0 1 0 x x 1 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td>0xC423 8E3F</td>
<td></td>
</tr>
<tr>
<td>Dreg_hi = Dreg_hi − Dreg_lo (NS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtract</td>
<td>0xC423 C000—</td>
<td>1 1 0 0 0 0 1 0 x x 1 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td>0xC423 CE3F</td>
<td></td>
</tr>
<tr>
<td>Dreg_hi = Dreg_hi − Dreg_hi (NS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtract</td>
<td>0xC403 2000—</td>
<td>1 1 0 0 0 0 1 0 x x 0 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td>0xC403 2E3F</td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo − Dreg_lo (S)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtract</td>
<td>0xC403 6000—</td>
<td>1 1 0 0 0 0 1 0 x x 0 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td>0xC403 6E3F</td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo − Dreg_hi (S)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtract</td>
<td>0xC403 A000—</td>
<td>1 1 0 0 0 0 1 0 x x 0 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td>0xC403 AE3F</td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = Dreg_hi − Dreg_lo (S)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtract</td>
<td>0xC403 E000—</td>
<td>1 1 0 0 0 0 1 0 x x 0 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td>0xC403 EE3F</td>
<td></td>
</tr>
</tbody>
</table>
## Arithmetic Operations Instructions

Table C-17. Arithmetic Operations Instructions (Sheet 44 of 44)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Subtract</strong></td>
<td>0xC423 2000—0xC423 2E3F</td>
<td>1 1 0 0 0 1 0 x x 1 0 0 0 1 1</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo – Dreg_lo (S)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Subtract</strong></td>
<td>0xC423 6000—0xC423 6E3F</td>
<td>1 1 0 0 0 1 0 x x 1 0 0 0 1 1</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo – Dreg_hi (S)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Subtract</strong></td>
<td>0xC423 A000—0xC423 AE3F</td>
<td>1 1 0 0 0 1 0 x x 1 0 0 0 1 1</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_hi – Dreg_lo (S)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Subtract</strong></td>
<td>0xC423 E000—0xC423 EE3F</td>
<td>1 1 0 0 0 1 0 x x 1 0 0 0 1 1</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_hi – Dreg_hi (S)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Subtract Immediate</strong></td>
<td>0x9F64—0x9F67</td>
<td>1 0 0 1 1 1 1 1 0 1 1 0 0 1 Ireg #</td>
</tr>
<tr>
<td>Ireg – = 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Subtract Immediate</strong></td>
<td>0x9F6C—0x9F6F</td>
<td>1 0 0 1 1 1 1 1 0 1 1 0 1 1 Ireg #</td>
</tr>
<tr>
<td>Ireg – = 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## External Event Management Instructions

Table C-18. External Event Management Instructions (Sheet 1 of 2)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Idle</em></td>
<td>0x0020</td>
<td>0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>IDLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>Core Synchronize</em></td>
<td>0x0023</td>
<td>0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>CSYNC</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>System Synchronize</em></td>
<td>0x0024</td>
<td>0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>SSYNC</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>Force Emulation</em></td>
<td>0x0025</td>
<td>0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 1 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>EMUXEXCPT</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>Disable Interrupts</em></td>
<td>0x0030—0x0037</td>
<td>0 0 0 0 0 0 0 0 0 0 0 1 1 Dreg #</td>
</tr>
<tr>
<td>CLI Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>Enable Interrupts</em></td>
<td>0x0040—0x0047</td>
<td>0 0 0 0 0 0 0 0 0 0 0 1 0 0 Dreg #</td>
</tr>
<tr>
<td>STI Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>Force Interrupt / Reset</em></td>
<td>0x0090—0x009F</td>
<td>0 0 0 0 0 0 0 0 1 0 0 1 1 uimm4</td>
</tr>
<tr>
<td>RAISE uimm4</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>Force Exception</em></td>
<td>0x00A0—0x00AF</td>
<td>0 0 0 0 0 0 0 0 1 0 0 1 0 uimm4</td>
</tr>
<tr>
<td>EXCPT uimm4</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>Test and Set Byte (Atomic)</em></td>
<td>0x00B0—0x00B5</td>
<td>0 0 0 0 0 0 0 0 1 0 1 1 0 Preg #</td>
</tr>
<tr>
<td>TESTSET (Preg)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>No Op</em></td>
<td>0x0000</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>NOP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Table C-18. External Event Management Instructions (Sheet 2 of 2)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>No Op</strong></td>
<td>0xC003 1800</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1</td>
<td></td>
</tr>
<tr>
<td><strong>MNOP</strong></td>
<td>0xC803 1800</td>
<td></td>
</tr>
<tr>
<td><strong>No Op</strong></td>
<td>1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 1 1</td>
<td></td>
</tr>
<tr>
<td><strong>MNOP</strong> when issued in parallel with two compatible load/store instructions</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Abort</strong></td>
<td>0x002F</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 0 0 0 0 0 0 1 0 1 1 1</td>
<td></td>
</tr>
</tbody>
</table>

NOTE: Abort is only valid on a simulator.

ABORT
## Cache Control Instructions

Table C-19. Cache Control Instructions

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Cache Prefetch</strong></td>
<td>0x0240—0x0247</td>
<td>0 0 0 0 0 0 1 0 0 1 0 0 0 Preg #</td>
</tr>
<tr>
<td><strong>PREFETCH [Preg]</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Data Cache Prefetch</strong></td>
<td>0x0260—0x0267</td>
<td>0 0 0 0 0 0 1 0 0 1 1 0 0 Preg #</td>
</tr>
<tr>
<td><strong>PREFETCH [Preg++]</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Data Cache Flush</strong></td>
<td>0x0250—0x0257</td>
<td>0 0 0 0 0 0 1 0 0 1 0 1 0 Preg #</td>
</tr>
<tr>
<td><strong>FLUSH [Preg]</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Data Cache Line Invalidate</strong></td>
<td>0x0248—0x024F</td>
<td>0 0 0 0 0 1 0 0 1 0 0 1 Preg #</td>
</tr>
<tr>
<td><strong>FLUSHINV [Preg]</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instruction Cache Flush</strong></td>
<td>0x0258—0x025F</td>
<td>0 0 0 0 0 1 0 0 1 0 1 1 Preg #</td>
</tr>
<tr>
<td><strong>IFLUSH [Preg]</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Video Pixel Operations Instructions

### Table C-20. Video Pixel Operations Instructions (Sheet 1 of 5)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Byte Align</strong></td>
<td>0xC60D 0000—0xC60D 0E3F</td>
<td>1 1 0 0 0 1 1 0 0 x x 0 1 1 0 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 0 0 Dest. Dreg # x x Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td><strong>Dreg = ALIGN8 (Dreg, Dreg)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Byte Align</strong></td>
<td>0xC60D 4000—0xC60D 4E3F</td>
<td>1 1 0 0 0 1 1 0 0 x x 0 1 1 0 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1 0 0 Dest. Dreg # x x Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td><strong>Dreg = ALIGN16 (Dreg, Dreg)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Byte Align</strong></td>
<td>0xC60D 800—0xC60D 8E3F</td>
<td>1 1 0 0 0 1 1 0 0 x x 0 1 1 0 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 0 0 Dest. Dreg # x x Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td><strong>Dreg = ALIGN24 (Dreg, Dreg)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Disable Alignment Exception for Load</strong></td>
<td>0xC412 C000</td>
<td>1 1 0 0 0 1 0 x x 0 1 0 0 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 1 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td><strong>NOTE:</strong> When issuing compatible load/store instructions in parallel with a Disable Alignment Exception for Load instruction, add 0x0800 0000 to the Disable Alignment Exception for Load opcode. <strong>DISALGNEXCPT</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dual 16-Bit Add / Clip</strong></td>
<td>0xC417 0000—0xC417 0E3F</td>
<td>1 1 0 0 0 1 1 0 0 x x 0 1 1 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 0 0 Dest. 0 Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td><strong>Dreg = BYTEOP3P (Dreg_pair, Dreg_pair) (LO)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dual 16-Bit Add / Clip</strong></td>
<td>0xC437 0000—0xC437 0E3F</td>
<td>1 1 0 0 0 1 1 0 0 x x 1 1 0 1 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 0 0 Dest. 0 Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td><strong>Dreg = BYTEOP3P (Dreg_pair, Dreg_pair) (HI)</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Instruction Opcodes

Table C-20. Video Pixel Operations Instructions (Sheet 2 of 5)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dual 16-Bit Add / Clip</strong></td>
<td>0xC417 2000—0xC417 1E3F</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 0 0 1 0 x x 0 1 0 1 1 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 1 0 Dest. 0 Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
<td></td>
</tr>
<tr>
<td>Dreg = BYTEOP3P (Dreg_pair, Dreg_pair) (LO, R)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dual 16-Bit Add / Clip</strong></td>
<td>0xC437 2000—0xC437 1E3F</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 0 0 1 0 x x 1 1 0 1 1 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 1 0 Dest. 0 Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
<td></td>
</tr>
<tr>
<td>NOTE: When issuing compatible load/store instructions in parallel with a Dual 16-Bit Add / Clip instruction, add 0x0800 0000 to the Dual 16-Bit Add / Clip opcode.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg = BYTEOP3P (Dreg_pair, Dreg_pair) (HI, R)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dual 16-Bit Accumulator Extraction with Addition</strong></td>
<td>0xC40C 403F—0xC40C 4FC0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 0 0 1 0 x x 0 1 1 1 1 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 1 0 0 Dest. of A1 Op Dreg # Dreg of A0 Op Dreg # 1 1 1 1 1 1</td>
<td></td>
</tr>
<tr>
<td>NOTE: When issuing compatible load/store instructions in parallel with a Dual 16-Bit Accumulator Extraction with Addition instruction, add 0x0800 0000 to the Dual 16-Bit Accumulator Extraction with Addition opcode.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Quad 8-Bit Add</strong></td>
<td>0xC415 0000—0xC415 0FFF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 0 0 1 0 x x 0 1 0 1 0 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 Dest. 1 Dreg # Dest. 0 Dreg # Source 0 Dreg # Source 1 Dreg #</td>
<td></td>
</tr>
<tr>
<td>(Dreg, Dreg) = BYTEOP16P (Dreg_pair, Dreg_pair)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Quad 8-Bit Add</strong></td>
<td>0xC415 2000—0xC415 2FFF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 0 0 1 0 x x 0 1 0 1 0 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 1 0 Dest. 1 Dreg # Dest. 0 Dreg # Source 0 Dreg # Source 1 Dreg #</td>
<td></td>
</tr>
<tr>
<td>NOTE: When issuing compatible load/store instructions in parallel with a Quad 8-Bit Add instruction, add 0x0800 0000 to the Quad 8-Bit Add opcode.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Dreg, Dreg) = BYTEOP16P (Dreg_pair, Dreg_pair) (R)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table C-20. Video Pixel Operations Instructions (Sheet 3 of 5)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quad 8-Bit Average-Byte</td>
<td>0xC414 0000—</td>
<td>1 1 0 0 0 1 1 0 x x 0 1 0 1 0 0</td>
</tr>
<tr>
<td></td>
<td>0xC414 0E3F</td>
<td>0 0 0 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Quad 8-Bit Average-Byte</td>
<td>0xC414 4000—</td>
<td>1 1 0 0 0 1 1 0 x x 0 1 0 1 0 0</td>
</tr>
<tr>
<td></td>
<td>0xC414 4E3F</td>
<td>0 1 0 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Quad 8-Bit Average-Byte</td>
<td>0xC414 2000—</td>
<td>1 1 0 0 0 1 1 0 x x 0 1 0 1 0 0</td>
</tr>
<tr>
<td></td>
<td>0xC414 2E3F</td>
<td>0 0 1 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Quad 8-Bit Average-Byte</td>
<td>0xC414 6000—</td>
<td>1 1 0 0 0 1 1 0 x x 0 1 0 1 0 0</td>
</tr>
<tr>
<td></td>
<td>0xC414 6E3F</td>
<td>0 1 1 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
</tbody>
</table>

NOTE: When issuing compatible load/store instructions in parallel with a Quad 8-Bit Average-Byte instruction, add 0x0800 0000 to the Quad 8-Bit Average-Byte opcode.

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quad 8-Bit Average-Half Word</td>
<td>0xC416 0000—</td>
<td>1 1 0 0 0 1 1 0 x x 0 1 0 1 1 1</td>
</tr>
<tr>
<td></td>
<td>0xC416 0E3F</td>
<td>0 0 0 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Quad 8-Bit Average-Half Word</td>
<td>0xC436 0000—</td>
<td>1 1 0 0 0 1 1 0 x x 1 1 0 1 1 1</td>
</tr>
<tr>
<td></td>
<td>0xC436 0E3F</td>
<td>0 0 0 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Quad 8-Bit Average-Half Word</td>
<td>0xC416 4000—</td>
<td>1 1 0 0 0 1 1 0 x x 0 1 0 1 1 1</td>
</tr>
<tr>
<td></td>
<td>0xC416 6E3F</td>
<td>0 1 0 0 Dest. Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
</tbody>
</table>

C-104 ADSP-BF53x/BF56x Blackfin Processor Programming Reference
### Instruction Opcodes

#### Table C-20. Video Pixel Operations Instructions (Sheet 4 of 5)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Quad 8-Bit Average-Half Word</strong></td>
<td>0xC436 4000—0xC436 6E3F</td>
<td>11 10 0 0 1 0 x x 1 1 0 1 1 0</td>
</tr>
<tr>
<td>Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (TH)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Quad 8-Bit Average-Half Word</strong></td>
<td>0xC416 2000—0xC416 2E3F</td>
<td>11 10 0 0 1 0 x x x 0 1 0 1 1 0</td>
</tr>
<tr>
<td>Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (RNDL, R)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Quad 8-Bit Average-Half Word</strong></td>
<td>0xC436 2000—0xC436 2E3F</td>
<td>11 10 0 0 1 0 x x 1 1 0 1 1 0</td>
</tr>
<tr>
<td>Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (RNDH, R)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Quad 8-Bit Average-Half Word</strong></td>
<td>0xC416 6000—0xC416 7E3F</td>
<td>11 10 0 0 1 0 x x 0 1 0 1 1 0</td>
</tr>
<tr>
<td>Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (TL, R)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: When issuing compatible load/store instructions in parallel with a Quad 8-Bit Average-Half Word instruction, add 0x0800 0000 to the Quad 8-Bit Average-Half Word opcode.

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Quad 8-Bit Pack</strong></td>
<td>0xC418 0000—0xC418 0E3F</td>
<td>11 10 0 0 1 0 x x 0 1 1 0 0 0</td>
</tr>
<tr>
<td>Dreg = BYTEPACK (Dreg, Dreg)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: When issuing compatible load/store instructions in parallel with a Quad 8-Bit Pack instruction, add 0x0800 0000 to the Quad 8-Bit Pack opcode.
## Video Pixel Operations Instructions

### Table C-20. Video Pixel Operations Instructions (Sheet 5 of 5)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Quad 8-Bit Subtract</strong></td>
<td>0xC415 4000—0xC415 4FFF</td>
<td>1 1 0 0 1 0 0 1 0 x x 0 1 0 1 0 1</td>
</tr>
<tr>
<td></td>
<td>0xC415 6000—0xC415 6FFF</td>
<td>0 1 1 0 0 0 1 0 1 0 0 1 0 0 1 0 1</td>
</tr>
</tbody>
</table>

(Dreg, Dreg) = BYTEOP16M (Dreg_pair, Dreg_pair)

NOTE: When issuing compatible load/store instructions in parallel with a Quad 8-Bit Subtract instruction, add 0x0800 0000 to the Quad 8-Bit Subtract opcode.

### Quad 8-Bit Subtract-Absolute-Accumulate

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Quad 8-Bit Subtract-Absolute-Accumulate</strong></td>
<td>0xC412 0000—0xC412 003F</td>
<td>1 1 0 0 0 1 0 x x 0 1 0 0 0 1 0 1 0</td>
</tr>
</tbody>
</table>

SAA (Dreg_pair, Dreg_pair)

NOTE: When issuing compatible load/store instructions in parallel with a Quad 8-Bit Subtract-Absolute-Accumulate instruction, add 0x0800 0000 to the Quad 8-Bit Subtract-Absolute-Accumulate opcode.

SAA (Dreg_pair, Dreg_pair) (R)

### Quad 8-Bit Unpack

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Quad 8-Bit Unpack</strong></td>
<td>0xC418 4000—0xC418 4FFF</td>
<td>1 1 0 0 1 0 0 1 0 x x 0 1 1 0 0 0</td>
</tr>
</tbody>
</table>

(Dreg, Dreg) = BYTEUNPACK Dreg_pair

NOTE: When issuing compatible load/store instructions in parallel with a Quad 8-Bit Unpack instruction, add 0x0800 0000 to the Quad 8-Bit Unpack opcode.

(Dreg, Dreg) = BYTEUNPACK Dreg_pair (R)
## Instruction Opcodes

### Vector Operations Instructions

Table C-21. Vector Operations Instructions (Sheet 1 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
<th>Source 1</th>
<th>Dreg #</th>
<th>Source 0</th>
<th>Dreg #</th>
<th>Dest.</th>
<th>Dreg #</th>
<th>Source 0</th>
<th>Dreg #</th>
<th>Source 1</th>
<th>Dreg #</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Add on Sign</strong></td>
<td>0xC40C 0000—0xC40C 0E38</td>
<td>1 1 0 0 0 1 0 x x x 0 0 1 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 Dest. Dreg # 0 0 0</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><strong>Dreg_hi = Dreg_lo = SIGN (Dreg_hi) * Dreg_hi + SIGN (Dreg_lo) * Dreg_lo</strong></td>
<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><strong>Compare Select (VIT_MAX)</strong></td>
<td>0xC609 C000—0xC609 CE07</td>
<td>1 1 0 0 0 1 1 0 0 x x x 0 1 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 0 0 Dest. Dreg # x x x</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><strong>Dreg = VIT_MAX (Dreg, Dreg) (ASR)</strong></td>
<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><strong>Compare Select (VIT_MAX)</strong></td>
<td>0xC609 8000—0xC609 8E07</td>
<td>1 1 0 0 0 1 1 0 0 x x x 0 1 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 0 0 Dest. Dreg # x x x</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><strong>Dreg = VIT_MAX (Dreg, Dreg) (ASL)</strong></td>
<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><strong>Compare Select (VIT_MAX)</strong></td>
<td>0xC609 4000—0xC609 4E07</td>
<td>1 1 0 0 0 1 1 0 0 x x x 0 1 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 1 0 0 Dest. Dreg # x x x 0 0 0</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><strong>Dreg_lo = VIT_MAX (Dreg) (ASR)</strong></td>
<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><strong>Compare Select (VIT_MAX)</strong></td>
<td>0xC609 0000—0xC609 0E07</td>
<td>1 1 0 0 0 1 1 0 0 x x x 0 1 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 Dest. Dreg # x x x 0 0 0</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><strong>Dreg_lo = VIT_MAX (Dreg) (ASL)</strong></td>
<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><strong>Vector Absolute Value</strong></td>
<td>0xC406 8000—0xC406 8E38</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 1 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 0 0 Dest. Dreg # 0 0 0</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><strong>Dreg = ABS Dreg (V)</strong></td>
<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><strong>Vector Add / Subtract</strong></td>
<td>0xC400 0000—0xC400 0E3F</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 Dest. Dreg # 0 0 0</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>**Dreg = Dreg +</td>
<td>+ Dreg**</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>
### Vector Operations Instructions

#### Table C-21. Vector Operations Instructions (Sheet 2 of 33)

*Instruction and Version* | *Opcode Range* | *Bin* |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Add / Subtract 0xC400 2000—0xC400 2E3F</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 0 0 0</td>
<td>0 0 1 0 Dest Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Vector Add / Subtract 0xC400 1000—0xC400 1E3F</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 0 0 0</td>
<td>0 0 0 1 Dest Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Vector Add / Subtract 0xC400 3000—0xC400 3E3F</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 0 0 0</td>
<td>0 0 1 1 Dest Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Vector Add / Subtract 0xC400 8000—0xC400 8E3F</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 0 0 0</td>
<td>0 0 1 1 Dest Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Vector Add / Subtract 0xC400 A000—0xC400 AE3F</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 0 0 0</td>
<td>1 0 1 1 Dest Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Vector Add / Subtract 0xC400 9000—0xC400 9E3F</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 0 0 0</td>
<td>1 0 1 1 Dest Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Vector Add / Subtract 0xC400 B000—0xC400 BE3F</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 0 0 0</td>
<td>1 0 1 1 Dest Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
</tbody>
</table>
### Instruction Opcodes

Table C-21. Vector Operations Instructions (Sheet 3 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC400 4000—0xC400 4E3F</td>
<td>1 1 0 0 0 1 0 x x 0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>0xC400 4E3F</td>
<td>0 1 0 0 Dest Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td></td>
<td>Dreg = Dreg +</td>
<td>– Dreg</td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC400 6000—0xC400 6E3F</td>
<td>1 1 0 0 0 1 0 x x 0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>0xC400 6E3F</td>
<td>0 1 1 0 Dest Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td></td>
<td>Dreg = Dreg +</td>
<td>– Dreg (S)</td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC400 5000—0xC400 5E3F</td>
<td>1 1 0 0 0 1 0 x x 0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>0xC400 5E3F</td>
<td>0 1 0 1 Dest Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td></td>
<td>Dreg = Dreg +</td>
<td>– Dreg (CO)</td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC400 7000—0xC400 7E3F</td>
<td>1 1 0 0 0 1 0 x x 0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>0xC400 7E3F</td>
<td>0 1 1 1 Dest Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td></td>
<td>Dreg = Dreg +</td>
<td>– Dreg (SC0)</td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC400 C000—0xC400 CE3F</td>
<td>1 1 0 0 0 1 0 x x 0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>0xC400 CE3F</td>
<td>1 1 0 0 Dest Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td></td>
<td>Dreg = Dreg —</td>
<td>– Dreg</td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC400 E000—0xC400 EE3F</td>
<td>1 1 0 0 0 1 0 x x 0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>0xC400 EE3F</td>
<td>1 1 1 0 Dest Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td></td>
<td>Dreg = Dreg —</td>
<td>– Dreg (S)</td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC400 D000—0xC400 DE3F</td>
<td>1 1 0 0 0 1 0 x x 0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>0xC400 DE3F</td>
<td>1 1 0 1 Dest Dreg # 0 0 0 Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td></td>
<td>Dreg = Dreg —</td>
<td>– Dreg (CO)</td>
</tr>
</tbody>
</table>
## Vector Operations Instructions

### Vector Add / Subtract

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Opcode Bin</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC400 F000—0xC400 FE3F</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 0 0 0</td>
<td>Dreg = Dreg –</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 0000—0xC401 0FFF</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 0 0 1</td>
<td>Dreg = Dreg +</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 8000—0xC401 8FFF</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 0 0 1</td>
<td>Dreg = Dreg +</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 C000—0xC401 CFFF</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 0 0 1</td>
<td>Dreg = Dreg +</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 2000—0xC401 2FFF</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 0 0 1</td>
<td>Dreg = Dreg +</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 A000—0xC401 AFFF</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 0 0 1</td>
<td>Dreg = Dreg +</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 E000—0xC401 EFF</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 0 0 1</td>
<td>Dreg = Dreg +</td>
</tr>
</tbody>
</table>
## Instruction Opcodes

Table C-21. Vector Operations Instructions (Sheet 5 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 1000—0xC401 1FFF</td>
<td>11100010010xx000001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0001 Dest 1 Dreg # Dest 0 Dreg # Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg, Dreg = Dreg --</td>
<td>-- Dreg (CO)</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 9000—0xC401 9FFF</td>
<td>11100010010xx000001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1001 Dest 1 Dreg # Dest 0 Dreg # Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg, Dreg = Dreg --</td>
<td>-- Dreg (CO, ASR)</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 D000—0xC401 DFFF</td>
<td>11100010010xx000001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1101 Dest 1 Dreg # Dest 0 Dreg # Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg, Dreg = Dreg --</td>
<td>-- Dreg (CO, ASL)</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 3000—0xC401 3FFF</td>
<td>11100010010xx000001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0011 Dest 1 Dreg # Dest 0 Dreg # Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg, Dreg = Dreg --</td>
<td>-- Dreg (SCO)</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 B000—0xC401 BFFF</td>
<td>11100010010xx000001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1011 Dest 1 Dreg # Dest 0 Dreg # Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg, Dreg = Dreg --</td>
<td>-- Dreg (SCO, ASR)</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 F000—0xC401 FFFF</td>
<td>11100010010xx000001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1111 Dest 1 Dreg # Dest 0 Dreg # Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg, Dreg = Dreg --</td>
<td>-- Dreg (SCO, ASL)</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC421 0000—0xC421 0FFF</td>
<td>11100010010xx100001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000 Dest 1 Dreg # Dest 0 Dreg # Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg, Dreg = Dreg --</td>
<td>+ Dreg</td>
</tr>
</tbody>
</table>
### Table C-21. Vector Operations Instructions (Sheet 6 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC421 8000—</td>
<td>1 1 0 0 0 1 0 x x x 1 0 0 0 0 1</td>
<td>Dest 1</td>
<td>Dest 0</td>
<td>Source 0</td>
<td>Source 1</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC421 C000—</td>
<td>1 1 0 0 0 1 0 x x x 1 0 0 0 0 1</td>
<td>Dest 1</td>
<td>Dest 0</td>
<td>Source 0</td>
<td>Source 1</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC421 2000—</td>
<td>1 1 0 0 0 1 0 x x x 1 0 0 0 0 1</td>
<td>Dest 1</td>
<td>Dest 0</td>
<td>Source 0</td>
<td>Source 1</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC421 A000—</td>
<td>1 1 0 0 0 1 0 x x x 1 0 0 0 0 1</td>
<td>Dest 1</td>
<td>Dest 0</td>
<td>Source 0</td>
<td>Source 1</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC421 E000—</td>
<td>1 1 0 0 0 1 0 x x x 1 0 0 0 0 1</td>
<td>Dest 1</td>
<td>Dest 0</td>
<td>Source 0</td>
<td>Source 1</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC421 1000—</td>
<td>1 1 0 0 0 1 0 x x x 1 0 0 0 0 1</td>
<td>Dest 1</td>
<td>Dest 0</td>
<td>Source 0</td>
<td>Source 1</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC421 9000—</td>
<td>1 1 0 0 0 1 0 x x x 1 0 0 0 0 1</td>
<td>Dest 1</td>
<td>Dest 0</td>
<td>Source 0</td>
<td>Source 1</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td>Dreg #</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Instruction Opcodes

#### Table C-21. Vector Operations Instructions (Sheet 7 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC421 D000—</td>
<td>1 1 0 0 0 1 0 x x x 1 0 0 0 0 1</td>
</tr>
<tr>
<td>0xC421 DFFF</td>
<td></td>
<td>1 1 1 0 Dest 1 Dreg # Dest 0 Dreg # Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>– Dreg, Dreg = Dreg —+ Dreg (CO, ASL)</td>
<td></td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC421 3000—</td>
<td>1 1 0 0 0 1 0 x x x 1 0 0 0 0 1</td>
</tr>
<tr>
<td>0xC421 3FFF</td>
<td></td>
<td>0 0 1 1 Dest 1 Dreg # Dest 0 Dreg # Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>– Dreg, Dreg = Dreg —+ Dreg (SCO)</td>
<td></td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC421 B000—</td>
<td>1 1 0 0 0 1 0 x x x 1 0 0 0 0 1</td>
</tr>
<tr>
<td>0xC421 BFFF</td>
<td></td>
<td>1 0 1 1 Dest 1 Dreg # Dest 0 Dreg # Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>– Dreg, Dreg = Dreg —+ Dreg (SCO, ASR)</td>
<td></td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC421 F000—</td>
<td>1 1 0 0 0 1 0 x x x 1 0 0 0 0 1</td>
</tr>
<tr>
<td>0xC421 FFFF</td>
<td></td>
<td>1 1 1 1 Dest 1 Dreg # Dest 0 Dreg # Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>– Dreg, Dreg = Dreg —+ Dreg (SCO, ASL)</td>
<td></td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC404 8000—</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 1 0 0</td>
</tr>
<tr>
<td>0xC404 8FFF</td>
<td></td>
<td>1 0 0 0 Dest 1 Dreg # Dest 0 Dreg # Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg = Dreg + Dreg, Dreg = Dreg — Dreg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC404 A000—</td>
<td>1 1 0 0 0 1 0 x x x 0 0 0 1 0 0</td>
</tr>
<tr>
<td>0xC404 AFFF</td>
<td></td>
<td>1 0 1 0 Dest 1 Dreg # Dest 0 Dreg # Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td>Dreg = Dreg + Dreg, Dreg = Dreg — Dreg (S)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC411 003F—</td>
<td>1 1 0 0 0 1 0 x x x 0 1 0 0 0 1</td>
</tr>
<tr>
<td>0xC411 0FC0</td>
<td></td>
<td>0 0 0 0 Dest 1 Dreg # Dest 0 Dreg # 1 1 1 1 1 1</td>
</tr>
<tr>
<td>Dreg = A1 + A0, Dreg = A1 — A0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Vector Operations Instructions

Table C-21. Vector Operations Instructions (Sheet 8 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Opcode</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC411 203F—0xC411 2FC0</td>
<td>1 1 0 0 0 1 0 x x x 0 1 0 0 0 1</td>
<td>Dest 1 Dreg # Dest 0 Dreg # 1 1 1 1 1 1</td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC411 403F—0xC411 4FC0</td>
<td>1 1 0 0 0 1 0 x x x 0 1 0 0 0 1</td>
<td>Dest 1 Dreg # Dest 0 Dreg # 1 1 1 1 1 1</td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC411 603F—0xC411 6FC0</td>
<td>1 1 0 0 0 1 0 x x x 0 1 0 0 0 1</td>
<td>Dest 1 Dreg # Dest 0 Dreg # 1 1 1 1 1 1</td>
</tr>
<tr>
<td><strong>Vector Arithmetic Shift</strong></td>
<td>0xC681 0100—0xC681 0FFF</td>
<td>1 1 0 0 0 1 1 1 0 1 x x 0 0 0 0 1</td>
<td>Dest. Dreg # 2's complement of uimm5 Source Dreg #</td>
</tr>
<tr>
<td><strong>Vector Arithmetic Shift</strong></td>
<td>0xC681 4000—0xC681 4EFF</td>
<td>1 1 0 0 0 1 1 1 0 1 x x 0 0 0 0 1</td>
<td>Dest. Dreg # uimm5 Source Dreg #</td>
</tr>
<tr>
<td><strong>Vector Arithmetic Shift</strong></td>
<td>0xC601 0000—0xC601 0E3F</td>
<td>1 1 0 0 0 1 1 0 0 x x 0 0 0 0 1</td>
<td>Dest. Dreg # Source 0 Dreg # Source 1 Dreg #</td>
</tr>
<tr>
<td><strong>Vector Arithmetic Shift</strong></td>
<td>0xC601 4000—0xC601 4E3F</td>
<td>1 1 0 0 0 1 1 0 0 x x 0 0 0 0 1</td>
<td>Dest. Dreg # Source 0 Dreg # Source 1 Dreg #</td>
</tr>
</tbody>
</table>
### Instruction Opcodes

Table C-21. Vector Operations Instructions (Sheet 9 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Logical Shift</td>
<td>0xC681 8180—0xC681 8FFF</td>
<td>110001101xx00001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00000000000001</td>
</tr>
<tr>
<td></td>
<td>Source</td>
<td>Dest. Dreg #</td>
</tr>
<tr>
<td></td>
<td>Source</td>
<td>2’s comp of uimm4</td>
</tr>
<tr>
<td>Dreg = Dreg &gt;&gt; uimm4 (V)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Logical Shift</td>
<td>0xC681 8000—0xC681 8E7F</td>
<td>110001101xx00001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00000000000001</td>
</tr>
<tr>
<td></td>
<td>Source</td>
<td>Dest. Dreg #</td>
</tr>
<tr>
<td></td>
<td>Source</td>
<td>uimm4</td>
</tr>
<tr>
<td>Dreg = Dreg &lt;&lt; uimm4 (V)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Logical Shift</td>
<td>0xC601 8000—0xC601 8E3F</td>
<td>110001100xx00001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00000000000001</td>
</tr>
<tr>
<td></td>
<td>Source 0</td>
<td>Dest. Dreg #</td>
</tr>
<tr>
<td></td>
<td>Source 1</td>
<td>Dest. Dreg #</td>
</tr>
<tr>
<td>Dreg = LSHIFT Dreg BY Dreg_lo (V)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Maximum</td>
<td>0xC406 0000—0xC406 0E3F</td>
<td>1100010xxx000110</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00000000000001</td>
</tr>
<tr>
<td></td>
<td>Source 0</td>
<td>Dest. Dreg #</td>
</tr>
<tr>
<td></td>
<td>Source 1</td>
<td>Dest. Dreg #</td>
</tr>
<tr>
<td>Dreg = MAX (Dreg, Dreg) (V)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Minimum</td>
<td>0xC406 4000—0xC406 4E3F</td>
<td>1100010xxx000110</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00000000000001</td>
</tr>
<tr>
<td></td>
<td>Source 0</td>
<td>Dest. Dreg #</td>
</tr>
<tr>
<td></td>
<td>Source 1</td>
<td>Dest. Dreg #</td>
</tr>
<tr>
<td>Dreg = MIN (Dreg, Dreg) (V)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC204 2000—0xC204 E7FF</td>
<td>1100001000000100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00000000000001</td>
</tr>
<tr>
<td></td>
<td>Dreg_lo_half 1</td>
<td>Dreg_lo_half 0</td>
</tr>
<tr>
<td></td>
<td>Dest. Dreg #</td>
<td>src_reg_hi</td>
</tr>
<tr>
<td></td>
<td>Dreg_lo_lo</td>
<td>src_reg_lo</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_lo, Dreg_hi = Dreg_lo_hi * Dreg_lo_lo (FU)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC284 2000—0xC284 E7FF</td>
<td>1100001000000100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00000000000001</td>
</tr>
<tr>
<td></td>
<td>Dreg_lo_half 1</td>
<td>Dreg_lo_half 0</td>
</tr>
<tr>
<td></td>
<td>Dest. Dreg #</td>
<td>src_reg_hi</td>
</tr>
<tr>
<td></td>
<td>Dreg_lo_lo</td>
<td>src_reg_lo</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_lo, Dreg_hi = Dreg_lo_hi * Dreg_lo_lo (FU)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Vector Operations Instructions

Table C-21. Vector Operations Instructions (Sheet 10 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode</th>
<th>Opcode Range</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vector Multiply</strong></td>
<td>0xC304</td>
<td>E7FF</td>
<td>1 1 0 0 0 0 1 1 0 0 0 0 1 0 0</td>
</tr>
<tr>
<td></td>
<td>0xC304</td>
<td></td>
<td>Dreg half 1 0 0 Dest. src_reg_0 src_reg_1 Dreg #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Dreg half 0 Dest. src_reg_0 src_reg_1 Dreg #</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IS)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| **Vector Multiply**     | 0xC384 | E7FF         | 1 1 0 0 0 0 1 1 1 0 0 0 1 0 0       |
| | 0xC384 |            | Dreg half 1 0 0 Dest. src_reg_0 src_reg_1 Dreg # |
| |        |            | Dreg half 0 Dest. src_reg_0 src_reg_1 Dreg # |
| Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IU) |        |            |

| **Vector Multiply**     | 0xC244 | E7FF         | 1 1 0 0 0 0 1 0 0 0 0 1 0 0       |
| | 0xC244 |            | Dreg half 1 0 0 Dest. src_reg_0 src_reg_1 Dreg # |
| |        |            | Dreg half 0 Dest. src_reg_0 src_reg_1 Dreg # |
| Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (T) |        |            |

| **Vector Multiply**     | 0xC2C4 | E7FF         | 1 1 0 0 0 0 1 0 1 1 0 0 0 1 0       |
| | 0xC2C4 |            | Dreg half 1 0 0 Dest. src_reg_0 src_reg_1 Dreg # |
| |        |            | Dreg half 0 Dest. src_reg_0 src_reg_1 Dreg # |
| Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (TFU) |        |            |

| **Vector Multiply**     | 0xC224 | E7FF         | 1 1 0 0 0 0 1 0 0 0 1 0 0 1 0       |
| | 0xC224 |            | Dreg half 1 0 0 Dest. src_reg_0 src_reg_1 Dreg # |
| |        |            | Dreg half 0 Dest. src_reg_0 src_reg_1 Dreg # |
| Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (S2RND) |        |            |

| **Vector Multiply**     | 0xC324 | E7FF         | 1 1 0 0 0 0 1 1 0 0 1 0 0 1 0       |
| | 0xC324 |            | Dreg half 1 0 0 Dest. src_reg_0 src_reg_1 Dreg # |
| |        |            | Dreg half 0 Dest. src_reg_0 src_reg_1 Dreg # |
| Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (ISS2) |        |            |
Table C-21. Vector Operations Instructions (Sheet 11 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Multiply</td>
<td>0xC364 2000—</td>
<td>111000011011001001100100</td>
</tr>
<tr>
<td></td>
<td>0xC364 E7FF</td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IH)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC214 2000—</td>
<td>1110000100010100010010100</td>
</tr>
<tr>
<td></td>
<td>0xC214 E7FF</td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC294 2000—</td>
<td>111000010100101001010100</td>
</tr>
<tr>
<td></td>
<td>0xC294 E7FF</td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (FU, M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC314 2000—</td>
<td>111000011001010010101000</td>
</tr>
<tr>
<td></td>
<td>0xC314 E7FF</td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IS, M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC394 2000—</td>
<td>111000011001010010101000</td>
</tr>
<tr>
<td></td>
<td>0xC394 E7FF</td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IU, M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC254 2000—</td>
<td>111000010010100010101000</td>
</tr>
<tr>
<td></td>
<td>0xC254 E7FF</td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (T, M)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Vector Operations Instructions

Table C-21. Vector Operations Instructions (Sheet 12 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
<th>Dreg</th>
<th>src_reg_</th>
<th>Dest.</th>
<th>Dreg #</th>
<th>Dreg #</th>
<th>Dreg #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Multiply</td>
<td>0xC2D4 2000—</td>
<td>1 1 0 0 0 0 0 1 1 0 1 1 0 1 0 1 0</td>
<td>Dreg half 1</td>
<td>src_reg_</td>
<td>Dest.</td>
<td>Dreg #</td>
<td>src_reg_</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xC2D4 E7FF</td>
<td></td>
<td>1 1 0 0 0 0 1 0 0 0 1 1 0 1 0 1 0</td>
<td>half 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dreg_lo = Dreg_lo_hi * Dreg_lo_hi,
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (TFU, M)

| Vector Multiply         | 0xC234 2000— | 1 1 0 0 0 0 1 1 0 1 0 1 0 0 | Dreg half 1 | src_reg_ | Dest. | Dreg # | src_reg_ |
|                         | 0xC234 E7FF   |     | 1 1 0 0 0 0 1 0 0 0 1 1 0 1 0 1 0 | half 0 |   |   | |

Dreg_lo = Dreg_lo_hi * Dreg_lo_hi,
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (S2RND, M)

| Vector Multiply         | 0xC334 2000— | 1 1 0 0 0 0 1 0 0 0 1 1 0 1 0 1 0 | Dreg half 1 | src_reg_ | Dest. | Dreg # | src_reg_ |
|                         | 0xC334 E7FF   |     | 1 1 0 0 0 0 1 0 0 0 1 1 0 1 0 1 0 | half 0 |   |   | |

Dreg_lo = Dreg_lo_hi * Dreg_lo_hi,
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (ISS2, M)

| Vector Multiply         | 0xC374 2000— | 1 1 0 0 0 0 1 1 0 1 0 1 0 | Dreg half 1 | src_reg_ | Dest. | Dreg # | src_reg_ |
|                         | 0xC374 E7FF   |     | 1 1 0 0 0 0 1 0 0 0 1 1 0 1 0 1 0 | half 0 |   |   | |

Dreg_lo = Dreg_lo_hi * Dreg_lo_hi,
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IH, M)

| Vector Multiply         | 0xC20C 2000— | 1 1 0 0 0 0 1 1 0 1 1 0 0 | Dreg half 1 | src_reg_ | Dest. | Dreg # | src_reg_ |
|                         | 0xC20C E7FF   |     | 1 1 0 0 0 0 1 0 0 0 1 1 0 1 0 1 0 | half 0 |   |   | |

Dreg_even = Dreg_lo_hi * Dreg_lo_hi,
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi

NOTE: When issuing compatible load/store instructions in parallel with a Vector Multiply instruction, add 0x0800 0000 to the Vector Multiply opcode.

NOTE: The ranges of these vector opcodes naturally overlap with the component scalar Multiply 16-Bit Operands opcodes. In fact, each vector opcode is the logical "OR" of the two component scalar opcodes.
## Instruction Opcodes

Table C-21. Vector Operations Instructions (Sheet 13 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vector Multiply</strong></td>
<td>0xC28C 2000—0xC28C E7FF</td>
<td>1100001010001100</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi, Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (FU)</td>
<td>Dreg half 1 1 0 0</td>
<td>Dest. src_reg_ src_reg_</td>
</tr>
<tr>
<td></td>
<td>Dreg half 0 0</td>
<td>Dreg #</td>
</tr>
<tr>
<td><strong>Vector Multiply</strong></td>
<td>0xC30C 2000—0xC30C E7FF</td>
<td>1100001100001100</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi, Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (IS)</td>
<td>Dreg half 1 1 0 0</td>
<td>Dest. src_reg_ src_reg_</td>
</tr>
<tr>
<td></td>
<td>Dreg half 0 0</td>
<td>Dreg #</td>
</tr>
<tr>
<td><strong>Vector Multiply</strong></td>
<td>0xC22C 2000—0xC22C E7FF</td>
<td>1100001000101100</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi, Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (S2RND)</td>
<td>Dreg half 1 1 0 0</td>
<td>Dest. src_reg_ src_reg_</td>
</tr>
<tr>
<td></td>
<td>Dreg half 0 0</td>
<td>Dreg #</td>
</tr>
<tr>
<td><strong>Vector Multiply</strong></td>
<td>0xC32C 2000—0xC32C E7FF</td>
<td>1100001100101100</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi, Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (ISS2)</td>
<td>Dreg half 1 1 0 0</td>
<td>Dest. src_reg_ src_reg_</td>
</tr>
<tr>
<td></td>
<td>Dreg half 0 0</td>
<td>Dreg #</td>
</tr>
<tr>
<td><strong>Vector Multiply</strong></td>
<td>0xC21C 2000—0xC21C E7FF</td>
<td>1100001000011100</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi, Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (M)</td>
<td>Dreg half 1 1 0 0</td>
<td>Dest. src_reg_ src_reg_</td>
</tr>
<tr>
<td></td>
<td>Dreg half 0 0</td>
<td>Dreg #</td>
</tr>
<tr>
<td><strong>Vector Multiply</strong></td>
<td>0xC29C 2000—0xC29C E7FF</td>
<td>1100001010011100</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi, Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (FU, M)</td>
<td>Dreg half 1 1 0 0</td>
<td>Dest. src_reg_ src_reg_</td>
</tr>
<tr>
<td></td>
<td>Dreg half 0 0</td>
<td>Dreg #</td>
</tr>
</tbody>
</table>
## Vector Operations Instructions

Table C-21. Vector Operations Instructions (Sheet 14 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vector Multiply</strong></td>
<td>0xC31C 2000—</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xC31C E7FF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>111000110011</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0011110000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0001110001</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000111100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1111100001</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000110001</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000110011</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000110100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000110111</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000111000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000111001</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000111010</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000111011</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000111100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000111101</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000111110</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000111111</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000100000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000100001</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000100010</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000100011</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000100100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000100101</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000100110</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000100111</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000101000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000101001</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000101010</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000101011</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000101100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000101101</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000101110</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000101111</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000110000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000110001</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000110010</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000110011</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000110100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000110101</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000110110</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000110111</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000111000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000111001</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000111010</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000111011</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000111100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000111101</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000111110</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110000111111</td>
<td></td>
</tr>
</tbody>
</table>

Dreg_even = Dreg_lo_hi * Dreg_lo_hi
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (IS, M)

Dreg_even = Dreg_lo_hi * Dreg_lo_hi
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (S2RND, M)

NOTE: When issuing compatible load/store instructions in parallel with a Vector Multiply instruction, add 0x0800 0000 to the Vector Multiply opcode.

NOTE: The ranges of these vector opcodes naturally overlaps with the component scalar Multiply 16-Bit Operands opcodes. In fact, each vector opcode is the logical "OR" of the two component scalar opcodes.

Dreg_even = Dreg_lo_hi * Dreg_lo_hi
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (ISS2, M)
### Instruction Opcodes

**Table C-21. Vector Operations Instructions (Sheet 15 of 33)**

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vector Multiply and Multiply-Accumulate</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEGEND:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg half 0 and Dreg half 1 determine which halves of the input operand registers to use.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg half 0 controls MAC0 operating on Dreg_lo and Dreg_even, and Dreg half 1 controls MAC1 operating on Dreg_hi and Dreg_odd.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo * Dreg_lo</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>Dreg_lo * Dreg_hi</td>
<td>0 1</td>
<td></td>
</tr>
<tr>
<td>Dreg_hi * Dreg_lo</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>Dreg_hi * Dreg_hi</td>
<td>1 1</td>
<td></td>
</tr>
</tbody>
</table>

Dest. Dreg # encodes the destination Data Register.
src_reg_0 Dreg # encodes the input operand register to the left of the "*" operand.
src_reg_1 Dreg # encodes the input operand register to the right of the "*" operand.

**Vector Multiply and Multiply-Accumulate**

NOTE: When issuing compatible load/store instructions in parallel with a Vector Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Vector Multiply and Multiply-Accumulate opcode.

Multiply and Multiply-Accumulate to Accumulator with Multiply and Multiply-Accumulate to Accumulator

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vector Multiply and Multiply-Accumulate</strong></td>
<td>0xC000 0000—0xC003 DE3F</td>
</tr>
</tbody>
</table>

A0 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi,
A1 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi
### Vector Operations Instructions

Table C-21. Vector Operations Instructions (Sheet 16 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC080 0000—0xC083 DE3F</td>
<td>1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 op1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A0 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC100 0000—0xC103 DE3F</td>
<td>1 1 0 0 0 0 0 1 0 0 0 0 0 0 0 op1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A0 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC060 0000—0xC063 DE3F</td>
<td>1 1 0 0 0 0 0 0 0 1 1 0 0 0 0 op1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A0 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC010 0000—0xC013 DE3F</td>
<td>1 1 0 0 0 0 0 0 0 0 1 0 0 0 0 op1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A0 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC070 0000—0xC073 DE3F</td>
<td>1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 op1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A0 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: When issuing compatible load/store instructions in parallel with a Vector Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Vector Multiply and Multiply-Accumulate opcode.

NOTE: The ranges of these vector opcodes naturally overlaps with the component scalar Multiply and Multiply-Accumulate opcodes. In fact, each vector opcode is the logical “OR” of the two component scalar opcodes.

A0 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi | |
A1 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi (W32, M)
### Instruction Opcodes

#### Table C-21. Vector Operations Instructions (Sheet 17 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>LEGEND:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>op0 and op1 specify the arithmetic operation for each MAC.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>op0 controls MAC0 operating on Accumulator A0 and op1 controls MAC1 operating on A1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>“=”</td>
<td>op0 = 0</td>
<td></td>
</tr>
<tr>
<td>“+=”</td>
<td>op0 = 1</td>
<td></td>
</tr>
<tr>
<td>“–=”</td>
<td>op1 = 1</td>
<td></td>
</tr>
<tr>
<td>Dreg_half_0 and Dreg_half_1 determine which halves of the input operand registers to use.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg_half_0 controls MAC0 operating on Accumulator A0 and Dreg_half_1 controls MAC1 operating on A1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo * Dreg_lo</td>
<td>Dreg_half_0 = 0</td>
<td></td>
</tr>
<tr>
<td>Dreg_lo * Dreg_hi</td>
<td>Dreg_half_0 = 1</td>
<td></td>
</tr>
<tr>
<td>Dreg_hi * Dreg_lo</td>
<td>Dreg_half_1 = 1</td>
<td></td>
</tr>
<tr>
<td>Dreg_hi * Dreg_hi</td>
<td>Dreg_half_1 = 0</td>
<td></td>
</tr>
<tr>
<td>src_reg_0 Dreg # encodes the input operand register to the left of the “*” operand.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src_reg_1 Dreg # encodes the input operand register to the right of the “*” operand.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Vector Multiply and Multiply-Accumulate

**NOTE:** When issuing compatible load/store instructions in parallel with a Vector Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Vector Multiply and Multiply-Accumulate opcode.

Multiply and Multiply-Accumulate to Half Register with Multiply and Multiply-Accumulate to Half Register

---

ADSP-BF53x/BF56x Blackfin Processor Programming Reference  C-123
### Vector Operations Instructions

Table C-21. Vector Operations Instructions (Sheet 18 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC004 2000—0xC007 FFFF</td>
<td>1 1 0 0 0 0 0 0 0 0 0 0 1 op1</td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC084 2000—0xC087 FFFF</td>
<td>1 1 0 0 0 0 0 0 1 0 0 0 0 1 op1</td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC104 2000—0xC107 FFFF</td>
<td>1 1 0 0 0 0 0 0 1 0 0 0 0 1 op1</td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC184 2000—0xC187 FFFF</td>
<td>1 1 0 0 0 0 0 0 1 1 0 0 0 0 1 op1</td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC044 2000—0xC047 FFFF</td>
<td>1 1 0 0 0 0 0 0 0 1 0 0 0 1 op1</td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC0C4 2000—0xC0C7 FFFF</td>
<td>1 1 0 0 0 0 0 0 0 1 1 0 0 0 1 op1</td>
</tr>
</tbody>
</table>
### Instruction Opcodes

#### Table C-21. Vector Operations Instructions (Sheet 19 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Multiply and</td>
<td>0xC024 2000—0xC027</td>
<td>Dreg_lo = (A0 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) (S2RND)</td>
</tr>
<tr>
<td>Multiply-Accumulate</td>
<td>FFFF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 op1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dreg half 0 1 1 0 0 Dest. src_reg_ src_reg_ 0 Dreg # 1 Dreg #</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply and</td>
<td>0xC124 2000—0xC127</td>
<td>Dreg_lo = (A0 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) (ISS2)</td>
</tr>
<tr>
<td>Multiply-Accumulate</td>
<td>FFFF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 0 0 0 0 0 1 0 0 1 0 0 1 0 1 op1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dreg half 1 1 op0 Dreg half 0 Dest. src_reg_ src_reg_ 0 Dreg # 1 Dreg #</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply and</td>
<td>0xC164 2000—0xC167</td>
<td>Dreg_lo = (A0 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) (IH)</td>
</tr>
<tr>
<td>Multiply-Accumulate</td>
<td>FFFF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 0 0 0 0 0 1 0 0 1 0 0 1 0 1 op1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dreg half 1 1 op0 Dreg half 0 Dest. src_reg_ src_reg_ 0 Dreg # 1 Dreg #</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply and</td>
<td>0xC014 2000—0xC017</td>
<td>Dreg_lo = (A0 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) (M)</td>
</tr>
<tr>
<td>Multiply-Accumulate</td>
<td>FFFF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 0 0 0 0 0 1 0 0 1 0 1 op1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dreg half 1 1 op0 Dreg half 0 Dest. src_reg_ src_reg_ 0 Dreg # 1 Dreg #</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply and</td>
<td>0xC094 2000—0xC097</td>
<td>Dreg_lo = (A0 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) (FU, M)</td>
</tr>
<tr>
<td>Multiply-Accumulate</td>
<td>FFFF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 0 0 0 0 0 1 0 0 1 0 1 op1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dreg half 1 1 op0 Dreg half 0 Dest. src_reg_ src_reg_ 0 Dreg # 1 Dreg #</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply and</td>
<td>0xC114 2000—0xC117</td>
<td>Dreg_lo = (A0 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) (IS, M)</td>
</tr>
<tr>
<td>Multiply-Accumulate</td>
<td>FFFF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 0 0 0 0 0 1 0 0 1 0 1 op1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dreg half 1 1 op0 Dreg half 0 Dest. src_reg_ src_reg_ 0 Dreg # 1 Dreg #</td>
<td></td>
</tr>
</tbody>
</table>
## Vector Operations Instructions

Table C-21. Vector Operations Instructions (Sheet 20 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC194 2000—0xC197 FFFF</td>
<td>1 1 1 0 0 1 0 0 1 1 0 1 0 1 0 1 1 op1</td>
</tr>
</tbody>
</table>

Dreg_lo = (A0 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi) ,
Dreg_hi = (A1 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi) (IU, M)

Dreg_lo = (A0 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi) ,
Dreg_hi = (A1 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi) (T, M)

Dreg_lo = (A0 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi) ,
Dreg_hi = (A1 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)

Dreg_lo = (A0 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi) ,
Dreg_hi = (A1 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)
Instruction Opcodes

Table C-21. Vector Operations Instructions (Sheet 21 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Multiply and</td>
<td>0xC174 2000—</td>
<td></td>
</tr>
<tr>
<td>Multiply-Accumulate</td>
<td>0xC177 FFFF</td>
<td></td>
</tr>
</tbody>
</table>

NOTE: When issuing compatible load/store instructions in parallel with a Vector Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Vector Multiply and Multiply-Accumulate opcode.

NOTE: The ranges of these vector opcodes naturally overlaps with the component scalar Multiply and Multiply-Accumulate opcodes. In fact, each vector opcode is the logical "OR" of the two component scalar opcodes.

Dreg_lo = (A0 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi),
Dreg_hi = (A1 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi) (IH, M)
Vector Operations Instructions

Table C-21. Vector Operations Instructions (Sheet 22 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEGEND:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>op0 and op1 specify the arithmetic operation for each MAC.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>op0 controls MAC0 operating on Accumulator A0 and op1 controls MAC1 operating on A1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>“=”</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>“+”</td>
<td>0 1</td>
<td></td>
</tr>
<tr>
<td>“-”</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>Dreg half 0 and Dreg half 1 determine which halves of the input operand registers to use.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg half 0 controls MAC0 operating on Accumulator A0 and</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg half 1 controls MAC1 operating on A1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo * Dreg_lo</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>Dreg_lo * Dreg_hi</td>
<td>0 1</td>
<td></td>
</tr>
<tr>
<td>Dreg_hi * Dreg_lo</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>Dreg_hi * Dreg_hi</td>
<td>1 1</td>
<td></td>
</tr>
<tr>
<td>Dest. Dreg # encodes the destination Data Register.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src_reg_0 Dreg # encodes the input operand register to the left of the “*” operand.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src_reg_1 Dreg # encodes the input operand register to the right of the “*” operand.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Vector Multiply and Multiply-Accumulate

NOTE: When issuing compatible load/store instructions in parallel with a Vector Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Vector Multiply and Multiply-Accumulate opcode.

Multiply and Multiply-Accumulate to Data Register with Multiply and Multiply-Accumulate to Data Register

C-128 ADSP-BF53x/BF56x Blackfin Processor Programming Reference
### Table C-21. Vector Operations Instructions (Sheet 23 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vector Multiply and Multiply-Accumulate</strong></td>
<td>0xC00C 2000—0xC00F FFFF</td>
<td>1 1 0 0 0 0 0 0 0 0 1 1 op1</td>
</tr>
<tr>
<td>Dreg_even = (A0 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi), Dreg_odd = (A1 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi)</td>
<td>Dreg half 1 1 op0</td>
<td>Dest. src_reg_ src_reg_</td>
</tr>
<tr>
<td>Dreg_odd = (A1 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td>Dreg half 0</td>
<td>src_reg_ Dreg #</td>
</tr>
<tr>
<td>Dreg_even = (A0 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi), Dreg_odd = (A1 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) (IS)</td>
<td>Dreg half 0</td>
<td>src_reg_ Dreg #</td>
</tr>
<tr>
<td>Dreg_even = (A0 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi), Dreg_odd = (A1 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) (S2RND)</td>
<td>Dreg half 0</td>
<td>src_reg_ Dreg #</td>
</tr>
<tr>
<td>Dreg_even = (A0 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi), Dreg_odd = (A1 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) (ISS2)</td>
<td>Dreg half 0</td>
<td>src_reg_ Dreg #</td>
</tr>
<tr>
<td>Dreg_even = (A0 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi), Dreg_odd = (A1 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) (M)</td>
<td>Dreg half 0</td>
<td>src_reg_ Dreg #</td>
</tr>
</tbody>
</table>
## Vector Operations Instructions

Table C-21. Vector Operations Instructions (Sheet 24 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Multiply and</td>
<td>0xC09C 2000—</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>op1</td>
<td></td>
</tr>
<tr>
<td>Multiply-Accumulate</td>
<td>0xC09F FFFF</td>
<td>Dreg</td>
<td>half</td>
<td>1</td>
<td>op0</td>
<td>Dreg</td>
<td>half</td>
<td>0</td>
<td>Dest.</td>
<td>Dreg</td>
<td>#</td>
<td>src_reg_</td>
<td>src_reg_</td>
<td>src_reg_</td>
<td>src_reg_</td>
<td>1 Dreg</td>
<td>#</td>
</tr>
</tbody>
</table>

Dreg_even = (A0 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi),
Dreg_odd = (A1 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi) (FU, M)

| Vector Multiply and     | 0xC11C 2000— | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | op1|
| Multiply-Accumulate     | 0xC11F FFFF  | Dreg | half | 1  | op0 | Dreg | half | 0  | Dest.| Dreg | #  | src_reg_ | src_reg_ | src_reg_ | src_reg_ | 1 Dreg | # |

Dreg_even = (A0 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi),
Dreg_odd = (A1 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi) (IS, M)

| Vector Multiply and     | 0xC03C 2000— | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | op1|
| Multiply-Accumulate     | 0xC03F FFFF  | Dreg | half | 1  | op0 | Dreg | half | 0  | Dest.| Dreg | #  | src_reg_ | src_reg_ | src_reg_ | src_reg_ | 1 Dreg | # |

Dreg_even = (A0 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi),
Dreg_odd = (A1 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)

| Vector Multiply and     | 0xC13C 2000— | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | op1|
| Multiply-Accumulate     | 0xC13F FFFF  | Dreg | half | 1  | op0 | Dreg | half | 0  | Dest.| Dreg | #  | src_reg_ | src_reg_ | src_reg_ | src_reg_ | 1 Dreg | # |

NOTE: When issuing compatible load/store instructions in parallel with a Vector Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Vector Multiply and Multiply-Accumulate opcode.

NOTE: The ranges of these vector opcodes naturally overlaps with the component scalar Multiply and Multiply-Accumulate opcodes. In fact, each vector opcode is the logical “OR” of the two component scalar opcodes.

Dreg_even = (A0 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi),
Dreg_odd = (A1 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)
### Instruction Opcodes

#### Table C-21. Vector Operations Instructions (Sheet 25 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vector Multiply and Multiply-Accumulate</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEGEND: op0 and op1 specify the arithmetic operation for each MAC.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>op0 controls MAC0 operating on Accumulator A0 and op1 controls MAC1 operating on A1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>“=”</td>
<td>op0 = 0</td>
<td>0 0</td>
</tr>
<tr>
<td>“+=”</td>
<td>op1 = 1</td>
<td>0 1</td>
</tr>
<tr>
<td>“−=”</td>
<td>op0 = 1</td>
<td>1 0</td>
</tr>
<tr>
<td>Dreg half 0 and Dreg half 1 determine which halves of the input operand registers to use.</td>
<td>Dreg</td>
<td></td>
</tr>
<tr>
<td>Dreg half 0 controls MAC0 operating on Accumulator A0 and Dreg half 1 controls MAC1 operating on A1.</td>
<td>half 0</td>
<td></td>
</tr>
<tr>
<td>Dreg half 0</td>
<td>and</td>
<td>Dreg half 1</td>
</tr>
<tr>
<td>Dreg_lo * Dreg_lo</td>
<td>Dreg_lo</td>
<td></td>
</tr>
<tr>
<td>Dreg_lo * Dreg_hi</td>
<td>Dreg_lo</td>
<td></td>
</tr>
<tr>
<td>Dreg_hi * Dreg_lo</td>
<td>Dreg_lo</td>
<td></td>
</tr>
<tr>
<td>Dreg_hi * Dreg_hi</td>
<td>Dreg_lo</td>
<td></td>
</tr>
<tr>
<td>Dest. Dreg # encodes the destination Data Register.</td>
<td>Dreg</td>
<td></td>
</tr>
<tr>
<td>src_reg_0 Dreg # encodes the input operand register to the left of the “*” operand.</td>
<td>src_reg_0</td>
<td></td>
</tr>
<tr>
<td>src_reg_1 Dreg # encodes the input operand register to the right of the “*” operand.</td>
<td>src_reg_1</td>
<td></td>
</tr>
</tbody>
</table>

**Vector Multiply and Multiply-Accumulate**

NOTE: When issuing compatible load/store instructions in parallel with a Vector Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Vector Multiply and Multiply-Accumulate opcode.

Multiply and Multiply-Accumulate to Accumulator with Multiply and Multiply-Accumulate to Half Register
### Vector Operations Instructions

**Table C-21. Vector Operations Instructions (Sheet 26 of 33)**

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC004 0000—0xC007 DFFF</td>
<td>11000000000001 op1</td>
</tr>
<tr>
<td>A0 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = (A1 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC084 0000—0xC087 DFFF</td>
<td>11000000100001 op1</td>
</tr>
<tr>
<td>A0 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = (A1 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC104 0000—0xC107 DFFF</td>
<td>11000001000001 op1</td>
</tr>
<tr>
<td>A0 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = (A1 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) (IS)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC000 2000—0xC003 FFFF</td>
<td>11000000000000 op1</td>
</tr>
<tr>
<td>Dreg_lo = (A0 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) , A1 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC080 2000—0xC083 FFFF</td>
<td>11000000000000 op1</td>
</tr>
<tr>
<td>Dreg_lo = (A0 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) , A1 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi (FU)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Instruction Opcodes

### Table C-21. Vector Operations Instructions (Sheet 27 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode</th>
<th>Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC100 2000—0xC103 FFFF</td>
<td>1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 op1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dreg half 1</td>
<td>Dreg half 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dest. src_reg_</td>
<td>src_reg_</td>
</tr>
<tr>
<td></td>
<td></td>
<td>src_reg_</td>
<td>src_reg_</td>
</tr>
</tbody>
</table>

NOTE: When issuing compatible load/store instructions in parallel with a Vector Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Vector Multiply and Multiply-Accumulate opcode.

NOTE: The ranges of these vector opcodes naturally overlaps with the component scalar Multiply and Multiply-Accumulate opcodes. In fact, each vector opcode is the logical "OR" of the two component scalar opcodes.

Dreg_lo = \( (A0 \text{ [=, +=, or –=]} Dreg_{lo_{hi}} \times Dreg_{lo_{hi}}) \),

A1 \( [=, +=, \text{or –=} \text{ Dreg}_{lo_{hi}} \times Dreg_{lo_{hi}} \) (IS)
Vector Operations Instructions

Table C-21. Vector Operations Instructions (Sheet 28 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LEGEND:
- op0 and op1 specify the arithmetic operation for each MAC.
- op0 controls MAC0 operating on Accumulator A0 and op1 controls MAC1 operating on A1.
  - "="
  - "+="
  - "–=

- Dreg half 0 and Dreg half 1 determine which halves of the input operand registers to use.
  - Dreg half 0 controls MAC0 operating on Accumulator A0 and
  - Dreg half 1 controls MAC1 operating on A1.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dreg_lo * Dreg_lo</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>Dreg_lo * Dreg_hi</td>
<td>0 1</td>
<td></td>
</tr>
<tr>
<td>Dreg_hi * Dreg_lo</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>Dreg_hi * Dreg_hi</td>
<td>1 1</td>
<td></td>
</tr>
</tbody>
</table>

Dest. Dreg # encodes the destination Data Register.
src_reg_0 Dreg # encodes the input operand register to the left of the "*" operand.
src_reg_1 Dreg # encodes the input operand register to the right of the "*" operand.

Vector Multiply and Multiply-Accumulate

NOTE: When issuing compatible load/store instructions in parallel with a Vector Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Vector Multiply and Multiply-Accumulate opcode.

Multiply and Multiply-Accumulate to Accumulator with Multiply and Multiply-Accumulate to Data Register

C-134 ADSP-BF53x/BF56x Blackfin Processor Programming Reference
### Instruction Opcodes

Table C-21. Vector Operations Instructions (Sheet 29 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC00C 0000— 0xC00F DFFF</td>
<td>1 1 1 0 0 0 0 0 0 1 0 1 1 op1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dest. Dreg half 1 0</td>
</tr>
<tr>
<td></td>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC08C 0000— 0xC08F DFFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dest. Dreg half 1 0</td>
</tr>
<tr>
<td></td>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC10C 0000— 0xC10F DFFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dest. Dreg half 1 0</td>
</tr>
<tr>
<td></td>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC008 2000— 0xC00B FFFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dest. Dreg half 1 1</td>
</tr>
<tr>
<td></td>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC088 2000— 0xC08B FFFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dest. Dreg half 1 1</td>
</tr>
</tbody>
</table>

A0 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi,
Dreg_odd = (A1 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi)

Dreg_even = (A0 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi),
A1 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi

Dreg_even = (A0 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi),
A1 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi (FU)
## Vector Operations Instructions

### Table C-21. Vector Operations Instructions (Sheet 30 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode</th>
<th>Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vector Multiply and Multiply-Accumulate</strong></td>
<td>0xC108 2000—</td>
<td>0xC10B FFFF</td>
<td>1 1 0 0 0 0 0 1 0 0 0 0 1 0 op1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Dest. half 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>src_reg_</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>src_reg_</td>
</tr>
</tbody>
</table>

Note: When issuing compatible load/store instructions in parallel with a Vector Multiply and Multiply-Accumulate instruction, add 0x0800 0000 to the Vector Multiply and Multiply-Accumulate opcode.

Note: The ranges of these vector opcodes naturally overlaps with the component scalar Multiply and Multiply-Accumulate opcodes. In fact, each vector opcode is the logical "OR" of the two component scalar opcodes.

Dreg_even = (A0 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) ,
A1 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi (IS)
### Instruction Opcodes

**Table C-21. Vector Operations Instructions (Sheet 31 of 33)**

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>op0 and op1</td>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>

**LEGEND:**
- op0 and op1 specify the arithmetic operation for each MAC.
- op0 controls MAC0 operating on Accumulator A0 and op1 controls MAC1 operating on A1.

<table>
<thead>
<tr>
<th>Multiply and Multiply-Accumulate</th>
<th>Dest. Dreg #</th>
<th>src_reg_0</th>
<th>src_reg_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dreg_lo * Dreg_lo</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>Dreg_lo * Dreg_hi</td>
<td>0 1</td>
<td>0 1</td>
<td></td>
</tr>
<tr>
<td>Dreg_hi * Dreg_lo</td>
<td>1 0</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>Dreg_hi * Dreg_hi</td>
<td>1 1</td>
<td>1 1</td>
<td></td>
</tr>
</tbody>
</table>

Dest. Dreg # encodes the destination Data Register.
src_reg_0 Dreg # encodes the input operand register to the left of the “*” operand.
src_reg_1 Dreg # encodes the input operand register to the right of the “*” operand.
Vector Operations Instructions

Table C-21. Vector Operations Instructions (Sheet 32 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Negate (Two’s-Complement)</td>
<td>0xC40F C000—0xC40F CE38</td>
<td>11 1 0 0 0 1 0 x x 0 0 1 1 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11 1 0 0 Dest. Dreg # 0 0 0 Source Dreg # 0 0 0</td>
</tr>
<tr>
<td></td>
<td>Dest.</td>
<td>Dreg # 000</td>
</tr>
<tr>
<td></td>
<td>Dreg = – Dreg (V)</td>
<td></td>
</tr>
<tr>
<td>Vector Pack</td>
<td>0xC604 0000—0xC604 0E3F</td>
<td>11 1 0 0 0 1 1 0 0 x x 0 0 1 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 0 0 Dest. Dreg # x x Source 0 Dreg # 0 1 0 0 Source 1 Dreg #</td>
</tr>
<tr>
<td></td>
<td>Dest.</td>
<td>Dreg # xxx</td>
</tr>
<tr>
<td></td>
<td>Dreg = PACK (Dreg_lo, Dreg_lo)</td>
<td></td>
</tr>
<tr>
<td>Vector Pack</td>
<td>0xC604 4000—0xC604 4E3F</td>
<td>11 1 0 0 0 1 1 0 0 x x 0 0 1 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1 0 0 Dest. Dreg # x x Source 0 Dreg # 0 1 0 0 Source 1 Dreg #</td>
</tr>
<tr>
<td></td>
<td>Dest.</td>
<td>Dreg # xxx</td>
</tr>
<tr>
<td></td>
<td>Dreg = PACK (Dreg_lo, Dreg_hi)</td>
<td></td>
</tr>
<tr>
<td>Vector Pack</td>
<td>0xC604 8000—0xC604 8E3F</td>
<td>11 1 0 0 0 1 1 0 0 x x 0 0 1 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 0 0 Dest. Dreg # x x Source 0 Dreg # 0 1 0 0 Source 1 Dreg #</td>
</tr>
<tr>
<td></td>
<td>Dest.</td>
<td>Dreg # xxx</td>
</tr>
<tr>
<td></td>
<td>Dreg = PACK (Dreg_hi, Dreg_lo)</td>
<td></td>
</tr>
<tr>
<td>Vector Pack</td>
<td>0xC604 C000—0xC604 CE3F</td>
<td>11 1 0 0 0 1 1 0 0 x x 0 0 1 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 1 0 0 Dest. Dreg # x x Source 0 Dreg # 0 1 0 0 Source 1 Dreg #</td>
</tr>
<tr>
<td></td>
<td>Dest.</td>
<td>Dreg # xxx</td>
</tr>
<tr>
<td></td>
<td>Dreg = PACK (Dreg_hi, Dreg_hi)</td>
<td></td>
</tr>
<tr>
<td>Vector Search</td>
<td>0xC40D 0000—0xC40D 2FFF</td>
<td>11 1 0 0 0 1 0 x x 0 0 1 1 0 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 0 0 Dest. 1 Dreg # 0 0 0 0 Dest. 0 Dreg # 0 0 0 0 Source Dreg # 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>Dest.</td>
<td>Dreg # 000</td>
</tr>
<tr>
<td></td>
<td>(Dreg, Dreg) = SEARCH Dreg (GT)</td>
<td></td>
</tr>
<tr>
<td>Vector Search</td>
<td>0xC40D 4000—0xC40D 6FFF</td>
<td>11 1 0 0 0 1 0 x x 0 0 1 1 0 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1 0 0 Dest. 1 Dreg # 0 0 0 0 Dest. 0 Dreg # 0 0 0 0 Source Dreg # 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>Dest.</td>
<td>Dreg # 000</td>
</tr>
<tr>
<td></td>
<td>(Dreg, Dreg) = SEARCH Dreg (GE)</td>
<td></td>
</tr>
</tbody>
</table>
Instruction Opcodes

Table C-21. Vector Operations Instructions (Sheet 33 of 33)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
<th>OPCODE</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Search</td>
<td>0xC40D 8000—0xC40D AFF8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 1 0 0 0 1 0 x x x 0 0 1 1 0 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 0 0</td>
<td>Dest. 1 Dreg #</td>
</tr>
<tr>
<td>(Dreg, Dreg) = SEARCH Dreg (LT)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Vector Search           | 0xC40D C000—0xC40D EFF8 | | |
| | | 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| | | 1 1 0 0 0 1 0 x x x 0 0 1 1 0 1 |
| | | 1 1 0 0 | Dest. 1 Dreg # | Dest. 0 Dreg # | Source Dreg # | 0 0 0 |

NOTE: When issuing compatible load/store instructions in parallel with a Vector Search instruction, add 0x0800 0000 to the Vector Search opcode.

(Dreg, Dreg) = SEARCH Dreg (LE)

Instructions Listed By Operation Code

Table C-22 and Table C-23 list the 16- and 32-bit operation codes.
## 16-Bit Opcode Instructions

Table C-22 lists the instructions that are represented by 16-bit opcodes.

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Op NOP</td>
<td>0x0000—</td>
</tr>
<tr>
<td>Return RTS</td>
<td>0x0010—</td>
</tr>
<tr>
<td>Return RTI</td>
<td>0x0011—</td>
</tr>
<tr>
<td>Return RTX</td>
<td>0x0012—</td>
</tr>
<tr>
<td>Return RTN</td>
<td>0x0013—</td>
</tr>
<tr>
<td>Return RTE</td>
<td>0x0014—</td>
</tr>
<tr>
<td>Idle IDLE</td>
<td>0x0020—</td>
</tr>
<tr>
<td>Core Synchronize CSYNC</td>
<td>0x0023—</td>
</tr>
<tr>
<td>System Synchronize SSYNC</td>
<td>0x0024—</td>
</tr>
<tr>
<td>Force Emulation EMUEXCPT</td>
<td>0x0025—</td>
</tr>
<tr>
<td>Abort ABORT</td>
<td>0x002F—</td>
</tr>
<tr>
<td>Disable Interrupts CLI Dreg</td>
<td>0x0030—</td>
</tr>
<tr>
<td></td>
<td>0x0037</td>
</tr>
<tr>
<td>Enable Interrupts STI Dreg</td>
<td>0x0040—</td>
</tr>
<tr>
<td></td>
<td>0x0047</td>
</tr>
<tr>
<td>Jump JUMP (Preg)</td>
<td>0x0050—</td>
</tr>
<tr>
<td></td>
<td>0x0057</td>
</tr>
</tbody>
</table>
### Instruction Opcodes

Table C-22. 16-Bit Opcode Instructions (Sheet 2 of 14)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Call</strong></td>
<td>0x0060—</td>
</tr>
<tr>
<td>CALL (Preg)</td>
<td>0x0067</td>
</tr>
<tr>
<td><strong>Call</strong></td>
<td>0x0070—</td>
</tr>
<tr>
<td>CALL (PC+Preg)</td>
<td>0x0077</td>
</tr>
<tr>
<td><strong>Jump</strong></td>
<td>0x0080—</td>
</tr>
<tr>
<td>JUMP (PC+Preg)</td>
<td>0x0087</td>
</tr>
<tr>
<td><strong>Force Interrupts / Reset</strong></td>
<td>0x0090—</td>
</tr>
<tr>
<td>RAISE uimm4</td>
<td>0x009F</td>
</tr>
<tr>
<td><strong>Force Exception</strong></td>
<td>0x00A0—</td>
</tr>
<tr>
<td>EXCPT uimm4</td>
<td>0x00AF</td>
</tr>
<tr>
<td><strong>Test and Set Byte (Atomic)</strong></td>
<td>0x00B0—</td>
</tr>
<tr>
<td>TESTSET (Preg)</td>
<td>0x00B5</td>
</tr>
<tr>
<td><strong>Pop</strong></td>
<td>0x0100—</td>
</tr>
<tr>
<td>mostreg=[SP++]</td>
<td>0x013F</td>
</tr>
<tr>
<td><strong>Push</strong></td>
<td>0x0140—</td>
</tr>
<tr>
<td>[– –SP]=allreg</td>
<td>0x017F</td>
</tr>
<tr>
<td><strong>Move CC</strong></td>
<td>0x0200—</td>
</tr>
<tr>
<td>Dreg = CC</td>
<td>0x0207</td>
</tr>
<tr>
<td><strong>Move CC</strong></td>
<td>0x0208—</td>
</tr>
<tr>
<td>CC = Dreg</td>
<td>0x020F</td>
</tr>
<tr>
<td><strong>Negate CC</strong></td>
<td>0x0218—</td>
</tr>
<tr>
<td>CC = !CC</td>
<td></td>
</tr>
<tr>
<td><strong>Data Cache Prefetch</strong></td>
<td>0x0240—</td>
</tr>
<tr>
<td>PREFETCH [Preg]</td>
<td>0x0247</td>
</tr>
<tr>
<td><strong>Data Cache Line Invalidate</strong></td>
<td>0x0248—</td>
</tr>
<tr>
<td>FLUSHINV [Preg]</td>
<td>0x024F</td>
</tr>
<tr>
<td><strong>Data Cache Flush</strong></td>
<td>0x0250—</td>
</tr>
<tr>
<td>FLUSH [Preg]</td>
<td>0x0257</td>
</tr>
<tr>
<td><strong>Instruction Cache Flush</strong></td>
<td>0x0258—</td>
</tr>
<tr>
<td>IFLUSH [Preg]</td>
<td>0x025F</td>
</tr>
<tr>
<td><strong>Data Cache Prefetch</strong></td>
<td>0x0260—</td>
</tr>
<tr>
<td>PREFETCH [Preg++]</td>
<td>0x0267</td>
</tr>
</tbody>
</table>
Table C-22. 16-Bit Opcode Instructions (Sheet 3 of 14)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Cache Line Invalidate</td>
<td>0x0268—0x026F</td>
</tr>
<tr>
<td>FLUSHINV [Preg++]</td>
<td></td>
</tr>
<tr>
<td>Data Cache Flush</td>
<td>0x0270—0x0277</td>
</tr>
<tr>
<td>FLUSH [Preg++]</td>
<td></td>
</tr>
<tr>
<td>Instruction Cache Flush</td>
<td>0x0278—0x027F</td>
</tr>
<tr>
<td>IFLUSH [Preg++]</td>
<td></td>
</tr>
<tr>
<td>Move CC</td>
<td>0x0300—0x030F</td>
</tr>
<tr>
<td>CC = statbit</td>
<td></td>
</tr>
<tr>
<td>Move CC</td>
<td>0x0320—0x032F</td>
</tr>
<tr>
<td>CC</td>
<td>= statbit</td>
</tr>
<tr>
<td>Move CC</td>
<td>0x0340—0x034F</td>
</tr>
<tr>
<td>CC &amp;= statbit</td>
<td></td>
</tr>
<tr>
<td>Move CC</td>
<td>0x0360—0x036F</td>
</tr>
<tr>
<td>CC ^= statbit</td>
<td></td>
</tr>
<tr>
<td>Move CC</td>
<td>0x0380—0x038F</td>
</tr>
<tr>
<td>statbit = CC</td>
<td></td>
</tr>
<tr>
<td>Move CC</td>
<td>0x03A0—0x03A0</td>
</tr>
<tr>
<td>statbit</td>
<td>= CC</td>
</tr>
<tr>
<td>Move CC</td>
<td>0x03C0—0x03D0</td>
</tr>
<tr>
<td>statbit &amp;= CC</td>
<td></td>
</tr>
<tr>
<td>Pop Multiple (P5:Preglim)=[SP++]</td>
<td>0x0480—0x0485</td>
</tr>
<tr>
<td>Push Multiple [−−SP]=(P5:Preglim)</td>
<td>0x04C0—0x04C5</td>
</tr>
<tr>
<td>Pop Multiple (R7:Dreglim)=[SP++]</td>
<td>0x0500—0x0538</td>
</tr>
<tr>
<td>Push Multiple [−−SP]=(R7:Dreglim)</td>
<td>0x0540—0x0578</td>
</tr>
<tr>
<td>Pop Multiple (R7:Dreglim, P5:Preglim)=[SP++]</td>
<td>0x0580—0x05BD</td>
</tr>
</tbody>
</table>
### Instruction Opcodes

#### Table C-22. 16-Bit Opcode Instructions (Sheet 4 of 14)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Push Multiple</strong></td>
<td>0x05C0—0x05FD</td>
</tr>
<tr>
<td>[-~SP]=&lt;(R7:Dreglim, P5:Preglim)</td>
<td></td>
</tr>
<tr>
<td><strong>Move Conditional</strong></td>
<td>0x0600—0x063F</td>
</tr>
<tr>
<td>IF !CC Dreg=Dreg</td>
<td></td>
</tr>
<tr>
<td><strong>Move Conditional</strong></td>
<td>0x0640—0x067F</td>
</tr>
<tr>
<td>IF !CC Dreg=Preg</td>
<td></td>
</tr>
<tr>
<td><strong>Move Conditional</strong></td>
<td>0x0680—0x06BF</td>
</tr>
<tr>
<td>IF !CC Preg=Dreg</td>
<td></td>
</tr>
<tr>
<td><strong>Move Conditional</strong></td>
<td>0x06C0—0x06FF</td>
</tr>
<tr>
<td>IF !CC Preg=Preg</td>
<td></td>
</tr>
<tr>
<td><strong>Move Conditional</strong></td>
<td>0x0700—0x073F</td>
</tr>
<tr>
<td>IF CC Dreg=Dreg</td>
<td></td>
</tr>
<tr>
<td><strong>Move Conditional</strong></td>
<td>0x0740—0x077F</td>
</tr>
<tr>
<td>IF CC Dreg=Preg</td>
<td></td>
</tr>
<tr>
<td><strong>Move Conditional</strong></td>
<td>0x07C0—0x07FF</td>
</tr>
<tr>
<td>IF CC Preg=Preg</td>
<td></td>
</tr>
<tr>
<td><strong>Compare Data Register</strong></td>
<td>0x0800—0x083F</td>
</tr>
<tr>
<td>CC = Dreg == Dreg</td>
<td></td>
</tr>
<tr>
<td><strong>Compare Pointer Register</strong></td>
<td>0x0840—0x087F</td>
</tr>
<tr>
<td>CC = Preg == Preg</td>
<td></td>
</tr>
<tr>
<td><strong>Compare Data Register</strong></td>
<td>0x0880—0x08BF</td>
</tr>
<tr>
<td>CC = Dreg &lt; Dreg</td>
<td></td>
</tr>
<tr>
<td><strong>Compare Pointer Register</strong></td>
<td>0x08C0—0x08FF</td>
</tr>
<tr>
<td>CC = Preg &lt; Preg</td>
<td></td>
</tr>
<tr>
<td><strong>Compare Data Register</strong></td>
<td>0x0900—0x093F</td>
</tr>
<tr>
<td>CC = Dreg &lt;= Dreg</td>
<td></td>
</tr>
<tr>
<td><strong>Compare Pointer Register</strong></td>
<td>0x0940—0x097F</td>
</tr>
<tr>
<td>CC = Preg &lt;= Preg</td>
<td></td>
</tr>
<tr>
<td><strong>Compare Data Register</strong></td>
<td>0x0980—0x09BF</td>
</tr>
<tr>
<td>CC = Dreg &lt; Dreg (IU)</td>
<td></td>
</tr>
</tbody>
</table>
### Instructions Listed By Operation Code

Table C-22. 16-Bit Opcode Instructions (Sheet 5 of 14)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare Pointer Register 0x09C0—0x09FF</td>
<td></td>
</tr>
<tr>
<td>CC = Preg &lt; Preg (IU)</td>
<td></td>
</tr>
<tr>
<td>Compare Data Register 0x0A00—0x0A3F</td>
<td></td>
</tr>
<tr>
<td>CC = Dreg &lt;= Dreg (IU)</td>
<td></td>
</tr>
<tr>
<td>Compare Pointer Register 0x0A40—0x0A7F</td>
<td></td>
</tr>
<tr>
<td>CC = Preg &lt;= Preg (IU)</td>
<td></td>
</tr>
<tr>
<td>Compare Accumulator 0x0A80—</td>
<td></td>
</tr>
<tr>
<td>CC = A0 == A1</td>
<td></td>
</tr>
<tr>
<td>Compare Accumulator 0x0B00—</td>
<td></td>
</tr>
<tr>
<td>CC = A0 &lt; A1</td>
<td></td>
</tr>
<tr>
<td>Compare Data Register 0x0C00—0x0C3F</td>
<td></td>
</tr>
<tr>
<td>CC = Dreg == imm3</td>
<td></td>
</tr>
<tr>
<td>Compare Pointer Register 0x0C40—0x0C7F</td>
<td></td>
</tr>
<tr>
<td>CC = Preg == imm3</td>
<td></td>
</tr>
<tr>
<td>Compare Data Register 0x0C80—0x0CBF</td>
<td></td>
</tr>
<tr>
<td>CC = Dreg &lt; imm3</td>
<td></td>
</tr>
<tr>
<td>Compare Pointer Register 0x0CC0—0x0CFF</td>
<td></td>
</tr>
<tr>
<td>CC = Preg &lt; imm3</td>
<td></td>
</tr>
<tr>
<td>Compare Data Register 0x0D00—0x0D3F</td>
<td></td>
</tr>
<tr>
<td>CC = Dreg &lt;= imm3</td>
<td></td>
</tr>
<tr>
<td>Compare Pointer Register 0x0D40—0x0D7F</td>
<td></td>
</tr>
<tr>
<td>CC = Preg &lt;= imm3</td>
<td></td>
</tr>
<tr>
<td>Compare Data Register 0x0D80—0x0DBF</td>
<td></td>
</tr>
<tr>
<td>CC = Dreg &lt; uimm3 (IU)</td>
<td></td>
</tr>
<tr>
<td>Compare Pointer Register 0x0D00—0x0DFF</td>
<td></td>
</tr>
<tr>
<td>CC = Preg &lt; uimm3 (IU)</td>
<td></td>
</tr>
<tr>
<td>Compare Data Register 0x0E00—0x0E3F</td>
<td></td>
</tr>
<tr>
<td>CC = Dreg &lt;= uimm3 (IU)</td>
<td></td>
</tr>
<tr>
<td>Compare Pointer Register 0x0E40—0x0E7F</td>
<td></td>
</tr>
<tr>
<td>CC = Preg &lt;= uimm3 (IU)</td>
<td></td>
</tr>
</tbody>
</table>
## Instruction Opcodes

Table C-22. 16-Bit Opcode Instructions (Sheet 6 of 14)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Conditional Jump</strong></td>
<td>0x1000—0x13FF</td>
</tr>
<tr>
<td>IF !CC JUMP pcrel11m2</td>
<td></td>
</tr>
<tr>
<td><strong>Conditional Jump</strong></td>
<td>0x1800—0x17FF</td>
</tr>
<tr>
<td>IF CC JUMP pcrel11m2</td>
<td></td>
</tr>
<tr>
<td><strong>Conditional Jump</strong></td>
<td>0x1400—0x1BFF</td>
</tr>
<tr>
<td>IF !CC JUMP pcrel11m2 (bp)</td>
<td></td>
</tr>
<tr>
<td><strong>Conditional Jump</strong></td>
<td>0x1C00—0x1FFF</td>
</tr>
<tr>
<td>IF CC JUMP pcrel11m2 (bp)</td>
<td></td>
</tr>
<tr>
<td><strong>Jump</strong></td>
<td>0x2000—0x2FFF</td>
</tr>
<tr>
<td>JUMPS pcrel13m2</td>
<td></td>
</tr>
<tr>
<td><strong>Move Register</strong></td>
<td>0x3000—0x3FFF</td>
</tr>
<tr>
<td>genreg = genreg</td>
<td></td>
</tr>
<tr>
<td>genreg = dagreg</td>
<td></td>
</tr>
<tr>
<td>dagreg = genreg</td>
<td></td>
</tr>
<tr>
<td>dagreg = dagreg</td>
<td></td>
</tr>
<tr>
<td>genreg = USP</td>
<td></td>
</tr>
<tr>
<td>USP = genreg</td>
<td></td>
</tr>
<tr>
<td>Dreg = sysreg</td>
<td></td>
</tr>
<tr>
<td>sysreg = Dreg</td>
<td></td>
</tr>
<tr>
<td>sysreg = Preg</td>
<td></td>
</tr>
<tr>
<td>sysreg = USP</td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td>0x4000—0x403F</td>
</tr>
<tr>
<td>Dreg &gt;&gt;&gt;= Dreg</td>
<td></td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td>0x4040—0x407F</td>
</tr>
<tr>
<td>Dreg &gt;&gt;= Dreg</td>
<td></td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td>0x4080—0x40BF</td>
</tr>
<tr>
<td>Dreg &lt;&lt;= Dreg</td>
<td></td>
</tr>
<tr>
<td><strong>Multiply 32-Bit Operands</strong></td>
<td>0x40C0—0x40FF</td>
</tr>
<tr>
<td>Dreg *= Dreg</td>
<td></td>
</tr>
<tr>
<td><strong>Add with Shift</strong></td>
<td>0x4100—0x413F</td>
</tr>
<tr>
<td>Dreg = (Dreg + Dreg) &lt;&lt; 1</td>
<td></td>
</tr>
<tr>
<td><strong>Add with Shift</strong></td>
<td>0x4140—0x417F</td>
</tr>
<tr>
<td>Dreg = (Dreg + Dreg) &lt;&lt; 2</td>
<td></td>
</tr>
<tr>
<td><strong>Divide Primitive</strong></td>
<td>0x4200—0x423F</td>
</tr>
<tr>
<td>DIVQ (Dreg, Dreg)</td>
<td></td>
</tr>
</tbody>
</table>
## Instructions Listed By Operation Code

### Table C-22. 16-Bit Opcode Instructions (Sheet 7 of 14)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Divide Primitive</strong></td>
<td>0x4240—0x427F</td>
</tr>
<tr>
<td>DIVS (Dreg, Dreg)</td>
<td></td>
</tr>
<tr>
<td><strong>Move Half to Full Word, Sign Extended</strong></td>
<td>0x4280—0x42BF</td>
</tr>
<tr>
<td>Dreg = Dreg_lo (X)</td>
<td></td>
</tr>
<tr>
<td><strong>Move Half to Full Word – Zero Extended</strong></td>
<td>0x42C0—0x42FF</td>
</tr>
<tr>
<td>Dreg = Dreg_lo (Z)</td>
<td></td>
</tr>
<tr>
<td><strong>Move Byte, Sign Extended</strong></td>
<td>0x4300—0x433F</td>
</tr>
<tr>
<td>Dreg = Dreg_byte (X)</td>
<td></td>
</tr>
<tr>
<td><strong>Move Byte, Zero Extended</strong></td>
<td>0x4340—0x437F</td>
</tr>
<tr>
<td>Dreg = Dreg_byte (Z)</td>
<td></td>
</tr>
<tr>
<td><strong>Negate (Two’s-Complement)</strong></td>
<td>0x4380—0x43BF</td>
</tr>
<tr>
<td>Dreg = − Dreg</td>
<td></td>
</tr>
<tr>
<td><strong>NOT (One’s-Complement)</strong></td>
<td>0x43C0—0x43FF</td>
</tr>
<tr>
<td>Dreg = ~ Dreg</td>
<td></td>
</tr>
<tr>
<td><strong>Modify-Decrement</strong></td>
<td>0x4400—0x443F</td>
</tr>
<tr>
<td>Preg −= Preg</td>
<td></td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td>0x4440—0x447F</td>
</tr>
<tr>
<td>Preg = Preg &lt;&lt; 2</td>
<td></td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td>0x44C0—0x44FF</td>
</tr>
<tr>
<td>Preg = Preg &gt;&gt; 2</td>
<td></td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td>0x4500—0x453F</td>
</tr>
<tr>
<td>Preg = Preg &gt;&gt; 1</td>
<td></td>
</tr>
<tr>
<td><strong>Modify-Increment</strong></td>
<td>0x4540—0x457F</td>
</tr>
<tr>
<td>Preg += Preg (BREV)</td>
<td></td>
</tr>
<tr>
<td><strong>Add with Shift</strong></td>
<td>0x4580—0x45BF</td>
</tr>
<tr>
<td>Preg = (Preg + Preg) &lt;&lt; 1</td>
<td></td>
</tr>
<tr>
<td><strong>Add with Shift</strong></td>
<td>0x45C0—0x45FF</td>
</tr>
<tr>
<td>Preg = (Preg + Preg) &lt;&lt; 2</td>
<td></td>
</tr>
<tr>
<td><strong>Bit Test</strong></td>
<td>0x4800—0x48FF</td>
</tr>
<tr>
<td>CC = ! BITTST (Dreg, uimm5)</td>
<td></td>
</tr>
<tr>
<td><strong>Bit Test</strong></td>
<td>0x4900—0x49FF</td>
</tr>
<tr>
<td>CC = BITTST (Dreg, uimm5)</td>
<td></td>
</tr>
</tbody>
</table>
### Instruction Opcodes

#### Table C-22. 16-Bit Opcode Instructions (Sheet 8 of 14)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bit Set</strong></td>
<td>0x4A00—0x4AFF</td>
</tr>
<tr>
<td>BITSET (Dreg, uimm5)</td>
<td>0x4A00—0x4AFF</td>
</tr>
<tr>
<td><strong>Bit Toggle</strong></td>
<td>0x4B00—0x4BFF</td>
</tr>
<tr>
<td>BITTGL (Dreg, uimm5)</td>
<td>0x4B00—0x4BFF</td>
</tr>
<tr>
<td><strong>Bit Clear</strong></td>
<td>0x4C00—0x4CFF</td>
</tr>
<tr>
<td>BITCLR (Dreg, uimm5)</td>
<td>0x4C00—0x4CFF</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td>0x4D00—0x4DFF</td>
</tr>
<tr>
<td>Dreg &gt;&gt;&gt;= uimm5</td>
<td>0x4D00—0x4DFF</td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td>0x4E00—0x4EFF</td>
</tr>
<tr>
<td>Dreg &gt;&gt;= uimm5</td>
<td>0x4E00—0x4EFF</td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td>0x4F00—0x4FFF</td>
</tr>
<tr>
<td>Dreg &lt;&lt;= uimm5</td>
<td>0x4F00—0x4FFF</td>
</tr>
<tr>
<td><strong>Add</strong></td>
<td>0x5000—0x51FF</td>
</tr>
<tr>
<td>Dreg = Dreg + Dreg</td>
<td>0x5000—0x51FF</td>
</tr>
<tr>
<td><strong>Subtract</strong></td>
<td>0x5200—0x53FF</td>
</tr>
<tr>
<td>Dreg = Dreg – Dreg</td>
<td>0x5200—0x53FF</td>
</tr>
<tr>
<td><strong>AND</strong></td>
<td>0x5400—0x55FF</td>
</tr>
<tr>
<td>Dreg = Dreg &amp; Dreg</td>
<td>0x5400—0x55FF</td>
</tr>
<tr>
<td><strong>OR</strong></td>
<td>0x5600—0x57FF</td>
</tr>
<tr>
<td>Dreg = Dreg</td>
<td>0x5600—0x57FF</td>
</tr>
<tr>
<td><strong>Exclusive-OR</strong></td>
<td>0x5800—0x59FF</td>
</tr>
<tr>
<td>Dreg = Dreg ^ Dreg</td>
<td>0x5800—0x59FF</td>
</tr>
<tr>
<td><strong>Add</strong></td>
<td>0x5A00—0x5BFF</td>
</tr>
<tr>
<td>Preg = Preg + Preg</td>
<td>0x5A00—0x5BFF</td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td>0x5A00—0x5BFF</td>
</tr>
<tr>
<td>Preg = Preg &lt;&lt; 1</td>
<td>0x5A00—0x5BFF</td>
</tr>
<tr>
<td><strong>Shift with Add</strong></td>
<td>0x5C00—0x5DFF</td>
</tr>
<tr>
<td>Preg = Preg + (Preg &lt;&lt;1)</td>
<td>0x5C00—0x5DFF</td>
</tr>
<tr>
<td><strong>Shift with Add</strong></td>
<td>0x5E00—0x5FFF</td>
</tr>
<tr>
<td>Preg = Preg + (Preg &lt;&lt;2)</td>
<td>0x5E00—0x5FFF</td>
</tr>
</tbody>
</table>

**NOTE:** The special case of the Preg = Preg + Preg Add instruction, where both input operands are the same Preg (e.g., p3 = p0+p0), produces the same opcode as the Logical Shift instruction Preg = Preg << 1 that accomplishes the same function. Both syntaxes double the input operand value, then place the result in a Preg.
### Instructions Listed By Operation Code

Table C-22. 16-Bit Opcode Instructions (Sheet 9 of 14)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Load Immediate</strong></td>
<td>0x6000—</td>
</tr>
<tr>
<td>Dreg = imm7 (X)</td>
<td>0x63FF</td>
</tr>
<tr>
<td><strong>Add Immediate</strong></td>
<td>0x6400—</td>
</tr>
<tr>
<td>Dreg += imm7</td>
<td>0x6700</td>
</tr>
<tr>
<td><strong>Load Immediate</strong></td>
<td>0x6800—</td>
</tr>
<tr>
<td>Preg = imm7 (X)</td>
<td>0x6BFF</td>
</tr>
<tr>
<td><strong>Add Immediate</strong></td>
<td>0x6C00—</td>
</tr>
<tr>
<td>Preg += imm7</td>
<td>0x6FFF—</td>
</tr>
<tr>
<td><strong>Load Data Register</strong></td>
<td>0x8000—</td>
</tr>
<tr>
<td>Dreg = [ Preg ++ Preg ]</td>
<td>0x81FF—</td>
</tr>
<tr>
<td><strong>Load Low Data Register Half</strong></td>
<td>0x8200—</td>
</tr>
<tr>
<td>Dreg_lo = W [ Preg ]</td>
<td>0x83FF—</td>
</tr>
<tr>
<td><strong>Load Low Data Register Half</strong></td>
<td>0x8201—</td>
</tr>
<tr>
<td>Dreg_lo = W [ Preg ++ Preg ]</td>
<td>0x83FE—</td>
</tr>
<tr>
<td><strong>Load High Data Register Half</strong></td>
<td>0x8400—</td>
</tr>
<tr>
<td>Dreg_hi = W [ Preg ]</td>
<td>0x85FF—</td>
</tr>
<tr>
<td><strong>Load High Data Register Half</strong></td>
<td>0x8401—</td>
</tr>
<tr>
<td>Dreg_hi = W [ Preg ++ Preg ]</td>
<td>0x85FE—</td>
</tr>
<tr>
<td><strong>Load Half Word, Zero Extended</strong></td>
<td>0x8601—</td>
</tr>
<tr>
<td>Dreg = W [ Preg ++ Preg ] (Z)</td>
<td>0x87FE—</td>
</tr>
<tr>
<td><strong>Load Half Word, Sign Extended</strong></td>
<td>0x8E00—</td>
</tr>
<tr>
<td>Dreg = W [ Preg ++ Preg ] (X)</td>
<td>0x8FFF—</td>
</tr>
<tr>
<td><strong>Store Data Register</strong></td>
<td>0x8800—</td>
</tr>
<tr>
<td>[ Preg ++ Preg ] = Dreg</td>
<td>0x89FF</td>
</tr>
<tr>
<td><strong>Store Low Data Register Half</strong></td>
<td>0x8A00—</td>
</tr>
<tr>
<td>W [ Preg ] = Dreg_lo</td>
<td>0x8BFF</td>
</tr>
<tr>
<td><strong>Store Low Data Register Half</strong></td>
<td>0x8A01—</td>
</tr>
<tr>
<td>W [ Preg ++ Preg ] = Dreg_lo</td>
<td>0x8BFE</td>
</tr>
<tr>
<td><strong>Store High Data Register Half</strong></td>
<td>0x8C00—</td>
</tr>
<tr>
<td>W [ Preg ] = Dreg_hi</td>
<td>0x8DFF</td>
</tr>
<tr>
<td><strong>Store High Data Register Half</strong></td>
<td>0x8C01—</td>
</tr>
<tr>
<td>W [ Preg ++ Preg ] = Dreg_hi</td>
<td>0x8DFE</td>
</tr>
</tbody>
</table>

C-148 ADSP-BF53x/BF56x Blackfin Processor Programming Reference
## Instruction Opcodes

Table C-22. 16-Bit Opcode Instructions (Sheet 10 of 14)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Load Data Register</strong></td>
<td>0x9000—0x903F</td>
</tr>
<tr>
<td>Dreg = [ Preg ++ ]</td>
<td></td>
</tr>
<tr>
<td><strong>Load Pointer Register</strong></td>
<td>0x9040—0x907F</td>
</tr>
<tr>
<td>Preg = [ Preg ++ ]</td>
<td></td>
</tr>
<tr>
<td><strong>Load Data Register</strong></td>
<td>0x9080—0x90BF</td>
</tr>
<tr>
<td>Dreg = [ Preg -- ]</td>
<td></td>
</tr>
<tr>
<td><strong>Load Pointer Register</strong></td>
<td>0x90C0—0x90FF</td>
</tr>
<tr>
<td>Preg = [ Preg -- ]</td>
<td></td>
</tr>
<tr>
<td><strong>Load Data Register</strong></td>
<td>0x9100—0x913F</td>
</tr>
<tr>
<td>Dreg = [ Preg ]</td>
<td></td>
</tr>
<tr>
<td><strong>Load Pointer Register</strong></td>
<td>0x9140—0x917F</td>
</tr>
<tr>
<td>Preg = [ Preg ]</td>
<td></td>
</tr>
<tr>
<td><strong>Store Data Register</strong></td>
<td>0x9200—0x923F</td>
</tr>
<tr>
<td>[ Preg ++ ] = Dreg</td>
<td></td>
</tr>
<tr>
<td><strong>Store Pointer Register</strong></td>
<td>0x9240—0x927F</td>
</tr>
<tr>
<td>[ Preg ++ ] = Preg</td>
<td></td>
</tr>
<tr>
<td><strong>Store Data Register</strong></td>
<td>0x9280—0x92BF</td>
</tr>
<tr>
<td>[ Preg -- ] = Dreg</td>
<td></td>
</tr>
<tr>
<td><strong>Store Pointer Register</strong></td>
<td>0x92C0—0x92FF</td>
</tr>
<tr>
<td>[ Preg -- ] = Preg</td>
<td></td>
</tr>
<tr>
<td><strong>Store Data Register</strong></td>
<td>0x9300—0x933F</td>
</tr>
<tr>
<td>[ Preg ] = Dreg</td>
<td></td>
</tr>
<tr>
<td><strong>Store Pointer Register</strong></td>
<td>0x9340—0x937F</td>
</tr>
<tr>
<td>[ Preg ] = Preg</td>
<td></td>
</tr>
<tr>
<td><strong>Load Half Word, Zero Extended</strong></td>
<td>0x9400—0x943F</td>
</tr>
<tr>
<td>Dreg = W [ Preg ++ ] (Z)</td>
<td></td>
</tr>
<tr>
<td><strong>Load Half Word, Sign Extended</strong></td>
<td>0x9440—0x947F</td>
</tr>
<tr>
<td>Dreg = W [ Preg ++ ] (X)</td>
<td></td>
</tr>
<tr>
<td><strong>Load Half Word, Zero Extended</strong></td>
<td>0x9480—0x94BF</td>
</tr>
<tr>
<td>Dreg = W [ Preg -- ] (Z)</td>
<td></td>
</tr>
<tr>
<td><strong>Load Half Word, Sign Extended</strong></td>
<td>0x94C0—0x94FF</td>
</tr>
<tr>
<td>Dreg = W [ Preg -- ] (X)</td>
<td></td>
</tr>
</tbody>
</table>
### Instructions Listed By Operation Code

Table C-22. 16-Bit Opcode Instructions (Sheet 11 of 14)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Load Half Word, Zero Extended</strong></td>
<td>0x9500—0x953F</td>
</tr>
<tr>
<td>Dreg = W [ Preg ] (Z)</td>
<td>0x9500—0x953F</td>
</tr>
<tr>
<td><strong>Load Half Word, Sign Extended</strong></td>
<td>0x9540—0x957F</td>
</tr>
<tr>
<td>Dreg = W [ Preg ] (X)</td>
<td>0x9540—0x957F</td>
</tr>
<tr>
<td><strong>Store Low Data Register Half</strong></td>
<td>0x9600—0x963F</td>
</tr>
<tr>
<td>W [ Preg ++ ] = Dreg</td>
<td>0x9600—0x963F</td>
</tr>
<tr>
<td><strong>Store Low Data Register Half</strong></td>
<td>0x9680—0x96BF</td>
</tr>
<tr>
<td>W [ Preg -- ] = Dreg</td>
<td>0x9680—0x96BF</td>
</tr>
<tr>
<td><strong>Store Low Data Register Half</strong></td>
<td>0x9700—0x973F</td>
</tr>
<tr>
<td>W [ Preg ] = Dreg</td>
<td>0x9700—0x973F</td>
</tr>
<tr>
<td><strong>Load Byte, Zero Extended</strong></td>
<td>0x9800—0x983F</td>
</tr>
<tr>
<td>Dreg = B [ Preg ++ ] (Z)</td>
<td>0x9800—0x983F</td>
</tr>
<tr>
<td><strong>Load Byte, Sign Extended</strong></td>
<td>0x9840—0x987F</td>
</tr>
<tr>
<td>Dreg = B [ Preg ++ ] (X)</td>
<td>0x9840—0x987F</td>
</tr>
<tr>
<td><strong>Load Byte, Zero Extended</strong></td>
<td>0x9880—0x98BF</td>
</tr>
<tr>
<td>Dreg = B [ Preg -- ] (Z)</td>
<td>0x9880—0x98BF</td>
</tr>
<tr>
<td><strong>Load Byte, Sign Extended</strong></td>
<td>0x98C0—0x98FF</td>
</tr>
<tr>
<td>Dreg = B [ Preg -- ] (X)</td>
<td>0x98C0—0x98FF</td>
</tr>
<tr>
<td><strong>Load Byte, Zero Extended</strong></td>
<td>0x9900—0x993F</td>
</tr>
<tr>
<td>Dreg = B [ Preg ] (Z)</td>
<td>0x9900—0x993F</td>
</tr>
<tr>
<td><strong>Load Byte, Sign Extended</strong></td>
<td>0x9940—0x997F</td>
</tr>
<tr>
<td>Dreg = B [ Preg ] (X)</td>
<td>0x9940—0x997F</td>
</tr>
<tr>
<td><strong>Store Byte</strong></td>
<td>0x9A00—0x9A3F</td>
</tr>
<tr>
<td>B [ Preg ++ ] = Dreg</td>
<td>0x9A00—0x9A3F</td>
</tr>
<tr>
<td><strong>Store Byte</strong></td>
<td>0x9A80—0x9ABF</td>
</tr>
<tr>
<td>B [ Preg -- ] = Dreg</td>
<td>0x9A80—0x9ABF</td>
</tr>
<tr>
<td><strong>Store Byte</strong></td>
<td>0x9B00—0x9B3F</td>
</tr>
<tr>
<td>B [ Preg ] = Dreg</td>
<td>0x9B00—0x9B3F</td>
</tr>
<tr>
<td><strong>Load Data Register</strong></td>
<td>0x9C00—0x9C1F</td>
</tr>
<tr>
<td>Dreg = [ Ireg ++ ]</td>
<td>0x9C00—0x9C1F</td>
</tr>
<tr>
<td><strong>Load Low Data Register Half</strong></td>
<td>0x9C20—0x9C3F</td>
</tr>
<tr>
<td>Dreg_lo = W [ Ireg ++ ]</td>
<td>0x9C20—0x9C3F</td>
</tr>
</tbody>
</table>
## Instruction Opcodes

Table C-22. 16-Bit Opcode Instructions (Sheet 12 of 14)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Load High Data Register Half</strong></td>
<td>0x9C40—0x9C5F</td>
</tr>
<tr>
<td>Dreg_hi = W [ Ireg ++ ]</td>
<td></td>
</tr>
<tr>
<td><strong>Load Data Register</strong></td>
<td>0x9C80—0x9C9F</td>
</tr>
<tr>
<td>Dreg = [ Ireg -- ]</td>
<td></td>
</tr>
<tr>
<td><strong>Load Low Data Register Half</strong></td>
<td>0x9CA0—0x9CBF</td>
</tr>
<tr>
<td>Dreg_lo = W [ Ireg -- ]</td>
<td></td>
</tr>
<tr>
<td><strong>Load High Data Register Half</strong></td>
<td>0x9CC0—0x9CDF</td>
</tr>
<tr>
<td>Dreg_hi = W [ Ireg -- ]</td>
<td></td>
</tr>
<tr>
<td><strong>Load Data Register</strong></td>
<td>0x9D00—0x9D1F</td>
</tr>
<tr>
<td>Dreg = [ Ireg ]</td>
<td></td>
</tr>
<tr>
<td><strong>Load Low Data Register Half</strong></td>
<td>0x9D20—0x9D3F</td>
</tr>
<tr>
<td>Dreg_lo = W [ Ireg ]</td>
<td></td>
</tr>
<tr>
<td><strong>Load High Data Register Half</strong></td>
<td>0x9D40—0x9D5F</td>
</tr>
<tr>
<td>Dreg_hi = W [ Ireg ]</td>
<td></td>
</tr>
<tr>
<td><strong>Load Data Register</strong></td>
<td>0x9D80—0x9DFF</td>
</tr>
<tr>
<td>Dreg = [ Ireg + Mreg ]</td>
<td></td>
</tr>
<tr>
<td><strong>Store Data Register</strong></td>
<td>0x9E00—0x9E1F</td>
</tr>
<tr>
<td>[ Ireg ++ ] = Dreg</td>
<td></td>
</tr>
<tr>
<td><strong>Store Low Data Register Half</strong></td>
<td>0x9E20—0x9E3F</td>
</tr>
<tr>
<td>W [ Ireg ++ ] = Dreg_lo</td>
<td></td>
</tr>
<tr>
<td><strong>Store High Data Register Half</strong></td>
<td>0x9E40—0x9E5F</td>
</tr>
<tr>
<td>W [ Ireg ++ ] = Dreg_hi</td>
<td></td>
</tr>
<tr>
<td><strong>Modify-Increment</strong></td>
<td>0x9E60—0x9E6F</td>
</tr>
<tr>
<td>Ireg += Mreg</td>
<td></td>
</tr>
<tr>
<td><strong>Modify-Decrement</strong></td>
<td>0x9E70—0x9E7F</td>
</tr>
<tr>
<td>Ireg -= Mreg</td>
<td></td>
</tr>
<tr>
<td><strong>Store Data Register</strong></td>
<td>0x9E80—0x9E9F</td>
</tr>
<tr>
<td>[ Ireg -- ] = Dreg</td>
<td></td>
</tr>
<tr>
<td><strong>Store Low Data Register Half</strong></td>
<td>0x9EA0—0x9EBF</td>
</tr>
<tr>
<td>W [ Ireg -- ] = Dreg_lo</td>
<td></td>
</tr>
<tr>
<td><strong>Store High Data Register Half</strong></td>
<td>0x9EC0—0x9EDF</td>
</tr>
<tr>
<td>W [ Ireg -- ] = Dreg_hi</td>
<td></td>
</tr>
</tbody>
</table>
## Instructions Listed By Operation Code

### Table C-22. 16-Bit Opcode Instructions (Sheet 13 of 14)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modify-Increment</td>
<td>0x9EE0—0x9EEF</td>
</tr>
<tr>
<td>Ireg += Mreg (brev)</td>
<td></td>
</tr>
<tr>
<td>Store Data Register</td>
<td>0x9F00—0x9F1F</td>
</tr>
<tr>
<td>[ Ireg ] = Dreg</td>
<td></td>
</tr>
<tr>
<td>Store Low Data Register Half</td>
<td>0x9F20—0x9F3F</td>
</tr>
<tr>
<td>W [ Ireg ] = Dreg_lo</td>
<td></td>
</tr>
<tr>
<td>Store High Data Register Half</td>
<td>0x9F40—0x9F5F</td>
</tr>
<tr>
<td>W [ Ireg ] = Dreg_hi</td>
<td></td>
</tr>
<tr>
<td>Add Immediate</td>
<td>0x9F60—0x9F63</td>
</tr>
<tr>
<td>Ireg += 2</td>
<td></td>
</tr>
<tr>
<td>Subtract Immediate</td>
<td>0x9F64—0x9F67</td>
</tr>
<tr>
<td>Ireg -= 2</td>
<td></td>
</tr>
<tr>
<td>Add Immediate</td>
<td>0x9F68—0x9F6B</td>
</tr>
<tr>
<td>Ireg += 4</td>
<td></td>
</tr>
<tr>
<td>Subtract Immediate</td>
<td>0x9F6C—0x9F6F</td>
</tr>
<tr>
<td>Ireg -= 4</td>
<td></td>
</tr>
<tr>
<td>Store Data Register</td>
<td>0x9F80—0x9FFF</td>
</tr>
<tr>
<td>[ Ireg ++ Mreg ] = Dreg</td>
<td></td>
</tr>
<tr>
<td>Load Data Register</td>
<td>0xA000—0xA3FF</td>
</tr>
<tr>
<td>Dreg = [ Preg + uimm6m4 ]</td>
<td></td>
</tr>
<tr>
<td>Load Half Word, Zero Extended</td>
<td>0xA400—0xA7FF</td>
</tr>
<tr>
<td>Dreg = W [ Preg + uimm5m2 ] (Z)</td>
<td></td>
</tr>
<tr>
<td>Load Half Word, Sign Extended</td>
<td>0xA800—0xABFF</td>
</tr>
<tr>
<td>Dreg = W [ Preg + uimm5m2 ] (X)</td>
<td></td>
</tr>
<tr>
<td>Load Pointer Register</td>
<td>0xAC00—0xAFFF</td>
</tr>
<tr>
<td>Preg = [ Preg + uimm6m4 ]</td>
<td></td>
</tr>
<tr>
<td>Store Data Register</td>
<td>0xB000—0xB3FF</td>
</tr>
<tr>
<td>[ Preg + uimm6m4 ] = Dreg</td>
<td></td>
</tr>
</tbody>
</table>
## Instruction Opcodes

Table C-22. 16-Bit Opcode Instructions (Sheet 14 of 14)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store Low Data Register Half</td>
<td>0xB400—0xB7FF</td>
</tr>
<tr>
<td>W [ Preg + uimm5m2 ] = Dreg</td>
<td>0xB400—0xB7FF</td>
</tr>
<tr>
<td>Load Data Register</td>
<td>0xB800—0xB9F7</td>
</tr>
<tr>
<td>Dreg = [ FP – uimm7m4 ]</td>
<td>0xB800—0xB9F7</td>
</tr>
<tr>
<td>Load Pointer Register</td>
<td>0xB808—0xB9FF</td>
</tr>
<tr>
<td>Preg = [ FP – uimm7m4 ]</td>
<td>0xB808—0xB9FF</td>
</tr>
<tr>
<td>Store Data Register</td>
<td>0xBA00—0xBBF7</td>
</tr>
<tr>
<td>[ FP – uimm7m4 ] = Dreg</td>
<td>0xBA00—0xBBF7</td>
</tr>
<tr>
<td>Store Pointer Register</td>
<td>0xBA08—0xBBFF</td>
</tr>
<tr>
<td>[ FP – uimm7m4 ] = Preg</td>
<td>0xBA08—0xBBFF</td>
</tr>
<tr>
<td>Store Pointer Register</td>
<td>0xBC00—0xBFFF</td>
</tr>
<tr>
<td>[ Preg + uimm6m4 ] = Preg</td>
<td>0xBC00—0xBFFF</td>
</tr>
</tbody>
</table>
### 32-Bit Opcode Instructions

Table C-23 lists the instructions that are represented by 32-bit opcodes.

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vector Multiply and Multiply-Accumulate</strong></td>
<td></td>
</tr>
<tr>
<td>A0 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi ,</td>
<td>0xC000 0000—</td>
</tr>
<tr>
<td>A1 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi</td>
<td>0xC003 DE3F</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Accumulator</strong></td>
<td></td>
</tr>
<tr>
<td>A1 = Dreg_lo_hi * Dreg_lo_hi</td>
<td>0xC000 1800—</td>
</tr>
<tr>
<td><strong>Vector Multiply and Multiply-Accumulate</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = (A0 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi) ,</td>
<td>0xC000 2000—</td>
</tr>
<tr>
<td>A1 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi</td>
<td>0xC003 FFFF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Accumulator</strong></td>
<td></td>
</tr>
<tr>
<td>A0 = Dreg_lo_hi * Dreg_lo_hi</td>
<td>0xC001 1800—</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Accumulator</strong></td>
<td></td>
</tr>
<tr>
<td>A0 += Dreg_lo_hi * Dreg_lo_hi</td>
<td>0xC002 1800—</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Accumulator</strong></td>
<td></td>
</tr>
<tr>
<td>A0 –= Dreg_lo_hi * Dreg_lo_hi</td>
<td>0xC003 0000—</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Accumulator</strong></td>
<td></td>
</tr>
<tr>
<td>A0 += Dreg_lo_hi * Dreg_lo_hi</td>
<td>0xC003 0800—</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Accumulator</strong></td>
<td></td>
</tr>
<tr>
<td>A0 –= Dreg_lo_hi * Dreg_lo_hi</td>
<td>0xC003 1000—</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi)</td>
<td>0xC003 2000—</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi)</td>
<td>0xC003 2800—</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = (A0 –= Dreg_lo_hi * Dreg_lo_hi)</td>
<td>0xC003 3000—</td>
</tr>
<tr>
<td><strong>Move Register Half</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = A0</td>
<td>0xC003 3800—</td>
</tr>
</tbody>
</table>

---
### Instruction Opcodes

#### Table C-23. 32-Bit Opcode Instructions (Sheet 2 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vector Multiply and Multiply-Accumulate</strong></td>
<td></td>
</tr>
<tr>
<td>A0 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = (A1 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi)</td>
<td>0xC004 0000—0xC007 DFFF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi)</td>
<td>0xC004 1800—0xC004 D9FF</td>
</tr>
<tr>
<td><strong>Vector Multiply and Multiply-Accumulate</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = (A0 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi)</td>
<td>0xC004 2000—0xC007 FFFF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi)</td>
<td>0xC005 1800—0xC005 D9FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_hi = (A1 –= Dreg_lo_hi * Dreg_lo_hi)</td>
<td>0xC006 1800—0xC006 D9FF</td>
</tr>
<tr>
<td><strong>Move Register Half</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_hi = A1</td>
<td>0xC007 1800—0xC007 19C0</td>
</tr>
<tr>
<td><strong>Move Register Half</strong></td>
<td></td>
</tr>
</tbody>
</table>
| Dreg_lo = A0, Dreg_hi = A1  
Dreg_hi = A1, Dreg_lo = A0 | 0xC007 3800—0xC007 39C0 |
| **Multiply and Multiply-Accumulate to Data Register**  |
| Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) | 0xC008 1800—0xC008 D9FF |
| **Vector Multiply and Multiply-Accumulate**  |
| Dreg_even = (A0 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi) , A1 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi | 0xC008 2000—0xC00B FFFF |
| **Multiply and Multiply-Accumulate to Data Register**  |
| Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) | 0xC009 1800—0xC009 D9FF |
| **Multiply and Multiply-Accumulate to Data Register**  |
| Dreg_odd = (A1 –= Dreg_lo_hi * Dreg_lo_hi) | 0xC00A 1800—0xC00A D9FF |
| **Vector Multiply and Multiply-Accumulate**  |
| A0 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi , Dreg_odd = (A1 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi) | 0xC00C 0000—0xC00F DFFF |
| **Vector Multiply and Multiply-Accumulate**  |
| Dreg_even = (A0 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi) , Dreg_odd = (A1 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi) | 0xC00C 2000—0xC00F FFFF |
| **Multiply and Multiply-Accumulate to Data Register**  |
| Dreg_even = (A0 = Dreg_lo_hi * Dreg_lo_hi) | 0xC00D 0000—0xC00D 07FF |
Instructions Listed By Operation Code

Table C-23. 32-Bit Opcode Instructions (Sheet 3 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC00D 0800—0xC00D 0FFF</td>
</tr>
<tr>
<td>Dreg_even = (A0 += Dreg_lo_hi * Dreg_lo_hi)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC00D 1000—0xC00D 17FF</td>
</tr>
<tr>
<td>Dreg_even = (A0 -= Dreg_lo_hi * Dreg_lo_hi)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC010 0000—0xC013 DE3F</td>
</tr>
<tr>
<td>A0 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
</tr>
<tr>
<td>A1 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi (M)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC010 1800—0xC010 D83F</td>
</tr>
<tr>
<td>A1 = Dreg_lo_hi * Dreg_lo_hi (M)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC011 1800—0xC011 D83F</td>
</tr>
<tr>
<td>A1 += Dreg_lo_hi * Dreg_lo_hi (M)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC014 1800—0xC014 D9FF</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (M)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC014 2000—0xC017 FFFF</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC015 1800—0xC015 D9FF</td>
</tr>
<tr>
<td>Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (M)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC016 1800—0xC016 D9FF</td>
</tr>
<tr>
<td>Dreg_hi = (A1 -= Dreg_lo_hi * Dreg_lo_hi) (M)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC018 1800—0xC018 D9FF</td>
</tr>
<tr>
<td>Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (M)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC019 1800—0xC019 D9FF</td>
</tr>
<tr>
<td>Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (M)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC01A 1800—0xC01A D9FF</td>
</tr>
<tr>
<td>Dreg_odd = (A1 -= Dreg_lo_hi * Dreg_lo_hi) (M)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC01C 2000—0xC01F FFFF</td>
</tr>
<tr>
<td>Dreg_even = (A0 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi) , Dreg_odd = (A1 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi) (M)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC022 1800—0xC022 D83F</td>
</tr>
<tr>
<td>A1 -= Dreg_lo_hi * Dreg_lo_hi (M)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC023 2000—0xC023 27FF</td>
</tr>
<tr>
<td>Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (S2RND)</td>
<td></td>
</tr>
</tbody>
</table>
### Instruction Opcodes

**Table C-23. 32-Bit Opcode Instructions (Sheet 4 of 40)**

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC023 2800—0xC023 2FFF</td>
</tr>
<tr>
<td>Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (S2RND)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC023 3000—0xC023 37FF</td>
</tr>
<tr>
<td>Dreg_lo = (A0 -= Dreg_lo_hi * Dreg_lo_hi) (S2RND)</td>
<td></td>
</tr>
<tr>
<td>Move Register Half</td>
<td>0xC023 3800—0xC023 39C0</td>
</tr>
<tr>
<td>Dreg_lo = A0 (S2RND)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC024 1800—0xC024 D9FF</td>
</tr>
<tr>
<td>Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (S2RND)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC024 2000—0xC027 FFFF</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC025 1800—0xC025 D9FF</td>
</tr>
<tr>
<td>Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (S2RND)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC026 1800—0xC026 D9FF</td>
</tr>
<tr>
<td>Dreg_hi = (A1 -= Dreg_lo_hi * Dreg_lo_hi) (S2RND)</td>
<td></td>
</tr>
<tr>
<td>Move Register Half</td>
<td>0xC027 1800—0xC027 19C0</td>
</tr>
<tr>
<td>Dreg_hi = A1 (S2RND)</td>
<td></td>
</tr>
<tr>
<td>Move Register Half</td>
<td>0xC027 3800—0xC027 39C0</td>
</tr>
<tr>
<td>Dreg_lo = A0, Dreg_hi = A1 (S2RND)</td>
<td></td>
</tr>
<tr>
<td>Move Register Half</td>
<td>0xC028 0000—0xC028 39C0</td>
</tr>
<tr>
<td>Dreg_hi = A1, Dreg_lo = A0 (S2RND)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC029 1800—0xC029 D9FF</td>
</tr>
<tr>
<td>Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (S2RND)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC029 1800—0xC029 D9FF</td>
</tr>
<tr>
<td>Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (S2RND)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC02A 1800—0xC02A D9FF</td>
</tr>
<tr>
<td>Dreg_odd = (A1 -= Dreg_lo_hi * Dreg_lo_hi) (S2RND)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC02C 2000—0xC02F FFFF</td>
</tr>
<tr>
<td>Dreg_even = (A0 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi) , Dreg_odd = (A1 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi) (S2RND)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC02D 0000—0xC02D 07FF</td>
</tr>
<tr>
<td>Dreg_even = (A0 += Dreg_lo_hi * Dreg_lo_hi) (S2RND)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC02D 0800—0xC02D 0FFF</td>
</tr>
<tr>
<td>Dreg_even = (A0 += Dreg_lo_hi * Dreg_lo_hi) (S2RND)</td>
<td></td>
</tr>
</tbody>
</table>
Table C-23. 32-Bit Opcode Instructions (Sheet 5 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_even = (A0 ←= Dreg_lo_hi * Dreg_lo_hi) (S2RND)</td>
<td>0xC02D 1000—0xC02D 17FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td>0xC034 1800—0xC034 D9FF</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)</td>
<td>0xC034 2000—0xC037 FFFF</td>
</tr>
<tr>
<td><strong>Vector Multiply and Multiply-Accumulate</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = (A0 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi), Dreg_hi = (A1 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)</td>
<td></td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td>0xC035 1800—0xC035 D9FF</td>
</tr>
<tr>
<td>Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)</td>
<td>0xC036 1800—0xC036 D9FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td>0xC038 1800—0xC038 D9FF</td>
</tr>
<tr>
<td>Dreg_odd = (A1 ←= Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)</td>
<td>0xC039 1800—0xC039 D9FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td>0xC03A 1800—0xC03A D9FF</td>
</tr>
<tr>
<td>Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)</td>
<td>0xC03C 2000—0xC03F FFFF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td>0xC043 1800—0xC043 27FF</td>
</tr>
<tr>
<td>Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (T)</td>
<td>0xC043 2800—0xC043 2FFF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td>0xC044 1800—0xC044 37FF</td>
</tr>
<tr>
<td>Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (T)</td>
<td>0xC044 2000—0xC044 99C0</td>
</tr>
<tr>
<td><strong>Move Register Half</strong></td>
<td>0xC044 2800—0xC044 D9FF</td>
</tr>
<tr>
<td>Dreg_lo = A0 (T)</td>
<td>0xC047 FFFF</td>
</tr>
<tr>
<td><strong>Vector Multiply and Multiply-Accumulate</strong></td>
<td>0xC047 2000—0xC047 37FF</td>
</tr>
<tr>
<td>Dreg_lo = (A0 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi), Dreg_hi = (A1 [=, +=, or −=] Dreg_lo_hi * Dreg_lo_hi) (T)</td>
<td>0xC047 2800—0xC047 FFFF</td>
</tr>
</tbody>
</table>
### Instruction Opcodes

Table C-23. 32-Bit Opcode Instructions (Sheet 6 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
</table>
| Multiply and Multiply-Accumulate to Half Register  
Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (T)  | 0xC045 1800—0xC045 D9FF |
| Multiply and Multiply-Accumulate to Half Register  
Dreg_hi = (A1 -= Dreg_lo_hi * Dreg_lo_hi) (T)  | 0xC046 1800—0xC046 D9FF |
| Move Register Half  
Dreg_hi = A1 (T)  | 0xC047 1800—0xC047 19C0 |
| Move Register Half  
Dreg_lo = A0, Dreg_hi = A1 (T)  | 0xC047 3800—0xC047 39C0 |
| Multiply and Multiply-Accumulate to Half Register  
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (T, M)  | 0xC054 1800—0xC054 D9FF |
| Vector Multiply and Multiply-Accumulate  
Dreg_lo = (A0 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi)  
Dreg_hi = (A1 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi) (T, M)  | 0xC054 2000—0xC057 FFFF |
| Multiply and Multiply-Accumulate to Half Register  
Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (T, M)  | 0xC055 1800—0xC055 D9FF |
| Multiply and Multiply-Accumulate to Half Register  
Dreg_hi = (A1 -= Dreg_lo_hi * Dreg_lo_hi) (T, M)  | 0xC056 1800—0xC056 D9FF |
| Vector Multiply and Multiply-Accumulate  
A0 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi  
A1 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi (W32)  | 0xC060 0000—0xC063 DE3F |
| Multiply and Multiply-Accumulate to Accumulator  
A1 = Dreg_lo_hi * Dreg_lo_hi (W32)  | 0xC060 1800—0xC060 D83F |
| Multiply and Multiply-Accumulate to Accumulator  
A1 += Dreg_lo_hi * Dreg_lo_hi (W32)  | 0xC061 1800—0xC061 D83F |
| Multiply and Multiply-Accumulate to Accumulator  
A1 -= Dreg_lo_hi * Dreg_lo_hi (W32)  | 0xC062 1800—0xC062 D83F |
| Multiply and Multiply-Accumulate to Accumulator  
A0 = Dreg_lo_hi * Dreg_lo_hi (W32)  | 0xC063 0000—0xC063 063F |
| Multiply and Multiply-Accumulate to Accumulator  
A0 += Dreg_lo_hi * Dreg_lo_hi (W32)  | 0xC063 0800—0xC063 E3F |
| Multiply and Multiply-Accumulate to Accumulator  
A0 -= Dreg_lo_hi * Dreg_lo_hi (W32)  | 0xC063 1000—0xC063 163F |
Table C-23. 32-Bit Opcode Instructions (Sheet 7 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC070 0000—0xC073 DE3F</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC070 1800—0xC070 D83F</td>
</tr>
<tr>
<td>A1 = Dreg_lo_hi * Dreg_lo_hi (W32, M)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC071 1800—0xC071 D83F</td>
</tr>
<tr>
<td>A1 += Dreg_lo_hi * Dreg_lo_hi (W32, M)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC072 1800—0xC072 D83F</td>
</tr>
<tr>
<td>A1 –= Dreg_lo_hi * Dreg_lo_hi (W32, M)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC080 0000—0xC083 DE3F</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC080 1800—0xC080 D83F</td>
</tr>
<tr>
<td>A1 = Dreg_lo_hi * Dreg_lo_hi (FU)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC081 1800—0xC081 D83F</td>
</tr>
<tr>
<td>A1 += Dreg_lo_hi * Dreg_lo_hi (FU)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC082 1800—0xC082 D83F</td>
</tr>
<tr>
<td>A1 –= Dreg_lo_hi * Dreg_lo_hi (FU)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC083 0000—0xC083 063F</td>
</tr>
<tr>
<td>A0 = Dreg_lo_hi * Dreg_lo_hi (FU)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC083 0800—0xC083 0E3F</td>
</tr>
<tr>
<td>A0 += Dreg_lo_hi * Dreg_lo_hi (FU)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Accumulator</td>
<td>0xC083 1000—0xC083 163F</td>
</tr>
<tr>
<td>A0 –= Dreg_lo_hi * Dreg_lo_hi (FU)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC083 2000—0xC083 27FF</td>
</tr>
<tr>
<td>Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC083 2800—0xC083 3FF</td>
</tr>
<tr>
<td>Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC083 3000—0xC083 37FF</td>
</tr>
<tr>
<td>Dreg_lo = (A0 –= Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td></td>
</tr>
</tbody>
</table>
### Instruction Opcodes

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move Register Half</td>
<td>0xC083 3800—</td>
</tr>
<tr>
<td>Dreg_lo = A0 (FU)</td>
<td>0xC083 39C0</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC084 1800—</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td>0xC084 D9FF</td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC084 0000—</td>
</tr>
<tr>
<td>A0 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi, Dreg_hi = (A1 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td>0xC087 DFFF</td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC084 2000—</td>
</tr>
<tr>
<td>Dreg_lo = (A0 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi), Dreg_hi = (A1 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td>0xC087 FFFF</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC085 1800—</td>
</tr>
<tr>
<td>Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td>0xC085 D9FF</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC086 1800—</td>
</tr>
<tr>
<td>Dreg_hi = (A1 -= Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td>0xC086 D9FF</td>
</tr>
<tr>
<td>Move Register Half</td>
<td>0xC087 1800—</td>
</tr>
<tr>
<td>Dreg_hi = A1 (FU)</td>
<td>0xC087 19C0</td>
</tr>
<tr>
<td>Move Register Half</td>
<td>0xC087 3800—</td>
</tr>
<tr>
<td>Dreg_lo = A0, Dreg_hi = A1 (FU)</td>
<td>0xC087 39C0</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC088 1800—</td>
</tr>
<tr>
<td>Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td>0xC088 D9FF</td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC088 2000—</td>
</tr>
<tr>
<td>Dreg_even = (A0 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi), A1 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi (FU)</td>
<td>0xC08B FFFF</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC089 1800—</td>
</tr>
<tr>
<td>Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td>0xC089 D9FF</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC08A 1800—</td>
</tr>
<tr>
<td>Dreg_odd = (A1 -= Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td>0xC08A D9FF</td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC08C 0000—</td>
</tr>
<tr>
<td>A0 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi, Dreg_odd = (A1 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td>0xC08F DFFF</td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC08C 2000—</td>
</tr>
<tr>
<td>Dreg_even = (A0 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi), Dreg_odd = (A1 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td>0xC08F FFFF</td>
</tr>
</tbody>
</table>
### Instructions Listed By Operation Code

Table C-23. 32-Bit Opcode Instructions (Sheet 9 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC08D 0000—0xC08D 07FF</td>
</tr>
<tr>
<td>Multiply even = (A0 = Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td>0xC08D 0000—0xC08D 07FF</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC08D 0800—0xC08D 0FFF</td>
</tr>
<tr>
<td>Multiply even = (A0 += Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td>0xC08D 0800—0xC08D 0FFF</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC08D 1000—0xC08D 17FF</td>
</tr>
<tr>
<td>Multiply even = (A0 -= Dreg_lo_hi * Dreg_lo_hi) (FU)</td>
<td>0xC08D 1000—0xC08D 17FF</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC094 1800—0xC094 D9FF</td>
</tr>
<tr>
<td>Multiply hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (FU, M)</td>
<td>0xC094 1800—0xC094 D9FF</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC095 1800—0xC095 D9FF</td>
</tr>
<tr>
<td>Multiply even = (A1 += Dreg_lo_hi * Dreg_lo_hi) (FU, M)</td>
<td>0xC095 1800—0xC095 D9FF</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC096 1800—0xC096 D9FF</td>
</tr>
<tr>
<td>Multiply hi = (A1 -= Dreg_lo_hi * Dreg_lo_hi) (FU, M)</td>
<td>0xC096 1800—0xC096 D9FF</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC098 1800—0xC098 D9FF</td>
</tr>
<tr>
<td>Multiply odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (FU, M)</td>
<td>0xC098 1800—0xC098 D9FF</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC099 1800—0xC099 D9FF</td>
</tr>
<tr>
<td>Multiply odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (FU, M)</td>
<td>0xC099 1800—0xC099 D9FF</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC09A 1800—0xC09A D9FF</td>
</tr>
<tr>
<td>Multiply even = (A0 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi)</td>
<td>0xC09A 1800—0xC09A D9FF</td>
</tr>
<tr>
<td>Multiply odd = (A1 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi) (FU, M)</td>
<td>0xC09A 1800—0xC09A D9FF</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC0C3 2000—0xC0C3 27FF</td>
</tr>
<tr>
<td>Multiply even = (A0 = Dreg_lo_hi * Dreg_lo_hi) (TFU)</td>
<td>0xC0C3 2000—0xC0C3 27FF</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC0C3 2800—0xC0C3 2FFF</td>
</tr>
<tr>
<td>Multiply even = (A0 += Dreg_lo_hi * Dreg_lo_hi) (TFU)</td>
<td>0xC0C3 2800—0xC0C3 2FFF</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC0C3 3000—0xC0C3 37FF</td>
</tr>
<tr>
<td>Multiply even = (A0 -= Dreg_lo_hi * Dreg_lo_hi) (TFU)</td>
<td>0xC0C3 3000—0xC0C3 37FF</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC0C4 1800—0xC0C4 D9FF</td>
</tr>
<tr>
<td>Multiply hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (TFU)</td>
<td>0xC0C4 1800—0xC0C4 D9FF</td>
</tr>
</tbody>
</table>
## Instruction Opcodes

### Table C-23. 32-Bit Opcode Instructions (Sheet 10 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vector Multiply and Multiply-Accumulate</strong></td>
<td></td>
</tr>
<tr>
<td>( D_{\text{reg lo}} = (A0 \mid=, , +=, , or , -=] \ D_{\text{reg hi}} \times D_{\text{reg hi}} , )</td>
<td>0xC0C4 2000—0xC0C7 FFFF</td>
</tr>
<tr>
<td>( D_{\text{reg hi}} = (A1 \mid=, , +=, , or , -=] \ D_{\text{reg hi}} \times D_{\text{reg hi}} ) (TFU)</td>
<td></td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td></td>
</tr>
<tr>
<td>( D_{\text{reg hi}} = (A1 += D_{\text{reg lo}} \times D_{\text{reg lo}}) ) (TFU)</td>
<td>0xC0C5 1800—0xC0C5 D9FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td></td>
</tr>
<tr>
<td>( D_{\text{reg hi}} = (A1 -= D_{\text{reg lo}} \times D_{\text{reg lo}}) ) (TFU)</td>
<td>0xC0C6 1800—0xC0C6 D9FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td></td>
</tr>
<tr>
<td>( D_{\text{reg hi}} = (A1 = D_{\text{reg lo}} \times D_{\text{reg lo}}) ) (TFU, M)</td>
<td>0xC0D4 1800—0xC0D4 D9FF</td>
</tr>
<tr>
<td><strong>Vector Multiply and Multiply-Accumulate</strong></td>
<td></td>
</tr>
<tr>
<td>( D_{\text{reg lo}} = (A0 \mid=, , +=, , or , -=] \ D_{\text{reg hi}} \times D_{\text{reg hi}} , )</td>
<td>0xC0D4 2000—0xC0D7 FFFF</td>
</tr>
<tr>
<td>( D_{\text{reg hi}} = (A1 \mid=, , +=, , or , -=] \ D_{\text{reg hi}} \times D_{\text{reg hi}} ) (TFU, M)</td>
<td></td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td></td>
</tr>
<tr>
<td>( D_{\text{reg hi}} = (A1 += D_{\text{reg lo}} \times D_{\text{reg lo}}) ) (TFU, M)</td>
<td>0xC0D5 1800—0xC0D5 D9FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td></td>
</tr>
<tr>
<td>( D_{\text{reg hi}} = (A1 -= D_{\text{reg lo}} \times D_{\text{reg lo}}) ) (TFU, M)</td>
<td>0xC0D6 1800—0xC0D6 D9FF</td>
</tr>
<tr>
<td><strong>Vector Multiply and Multiply-Accumulate</strong></td>
<td></td>
</tr>
<tr>
<td>( A0 \mid=, , +=, , or , -=] \ D_{\text{reg lo}} \times D_{\text{reg lo}} ) , ( A1 \mid=, , +=, , or , -=] \ D_{\text{reg hi}} \times D_{\text{reg hi}} ) (IS)</td>
<td>0xC100 0000—0xC103 DE3F</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Accumulator</strong></td>
<td></td>
</tr>
<tr>
<td>( A1 = D_{\text{reg lo}} \times D_{\text{reg lo}} ) (IS)</td>
<td>0xC100 1800—0xC100 D83F</td>
</tr>
<tr>
<td><strong>Vector Multiply and Multiply-Accumulate</strong></td>
<td></td>
</tr>
<tr>
<td>( D_{\text{reg lo}} = (A0 \mid=, , +=, , or , -=] \ D_{\text{reg lo}} \times D_{\text{reg lo}} , )</td>
<td>0xC100 2000—0xC103 FFFF</td>
</tr>
<tr>
<td>( A1 \mid=, , +=, , or , -=] \ D_{\text{reg hi}} \times D_{\text{reg hi}} ) (IS)</td>
<td></td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Accumulator</strong></td>
<td></td>
</tr>
<tr>
<td>( A1 += D_{\text{reg lo}} \times D_{\text{reg lo}} ) (IS)</td>
<td>0xC101 1800—0xC101 D83F</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Accumulator</strong></td>
<td></td>
</tr>
<tr>
<td>( A1 -= D_{\text{reg lo}} \times D_{\text{reg lo}} ) (IS)</td>
<td>0xC102 1800—0xC102 D83F</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Accumulator</strong></td>
<td></td>
</tr>
<tr>
<td>( A0 = D_{\text{reg lo}} \times D_{\text{reg lo}} ) (IS)</td>
<td>0xC103 0000—0xC103 063F</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Accumulator</strong></td>
<td></td>
</tr>
<tr>
<td>( A0 += D_{\text{reg lo}} \times D_{\text{reg lo}} ) (IS)</td>
<td>0xC103 0800—0xC103 0E3F</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Accumulator</strong></td>
<td></td>
</tr>
<tr>
<td>( A0 -= D_{\text{reg lo}} \times D_{\text{reg lo}} ) (IS)</td>
<td>0xC103 1000—0xC103 163F</td>
</tr>
</tbody>
</table>
### Instruction List by Operation Code

#### Table C-23. 32-Bit Opcode Instructions (Sheet 11 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong>&lt;br&gt;Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (IS)</td>
<td>0xC103 2000—0xC103 27FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong>&lt;br&gt;Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (IS)</td>
<td>0xC103 2800—0xC103 2FFF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong>&lt;br&gt;Dreg_lo = (A0 -= Dreg_lo_hi * Dreg_lo_hi) (IS)</td>
<td>0xC103 3000—0xC103 37FF</td>
</tr>
<tr>
<td><strong>Move Register Half</strong>&lt;br&gt;Dreg_lo = A0 (IS)</td>
<td>0xC103 3800—0xC103 39C0</td>
</tr>
<tr>
<td><strong>Vector Multiply and Multiply-Accumulate</strong>&lt;br&gt;A0 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi, Dreg_hi = (A1 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi) (IS)</td>
<td>0xC104 0000—0xC107 DFFF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong>&lt;br&gt;Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IS)</td>
<td>0xC104 1800—0xC104 D9FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong>&lt;br&gt;Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IS)</td>
<td>0xC105 1800—0xC105 D9FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong>&lt;br&gt;Dreg_hi = (A1 -= Dreg_lo_hi * Dreg_lo_hi) (IS)</td>
<td>0xC106 1800—0xC106 D9FF</td>
</tr>
<tr>
<td><strong>Move Register Half</strong>&lt;br&gt;Dreg_hi = A1 (IS)</td>
<td>0xC107 1800—0xC107 19C0</td>
</tr>
<tr>
<td><strong>Move Register Half</strong>&lt;br&gt;Dreg_lo = A0, Dreg_hi = A1 (IS)</td>
<td>0xC107 3800—0xC107 39C0</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong>&lt;br&gt;Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IS)</td>
<td>0xC108 1800—0xC108 D9FF</td>
</tr>
<tr>
<td><strong>Vector Multiply and Multiply-Accumulate</strong>&lt;br&gt;Dreg_even = (A0 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi), A1 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi (IS)</td>
<td>0xC109 2000—0xC10B FFFF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong>&lt;br&gt;Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IS)</td>
<td>0xC109 1800—0xC109 D9FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong>&lt;br&gt;Dreg_odd = (A1 -= Dreg_lo_hi * Dreg_lo_hi) (IS)</td>
<td>0xC10A 1800—0xC10A D9FF</td>
</tr>
</tbody>
</table>
### Instruction Opcodes

#### Table C-23. 32-Bit Opcode Instructions (Sheet 12 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vector Multiply and Multiply-Accumulate</strong></td>
<td>0xC10C 0000—0xC10F DFFF</td>
</tr>
<tr>
<td>Dreg_lo = (A0 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi), Dreg_odd = (A1 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi) (IS)</td>
<td>0xC10C 0000—0xC10F DFFF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td>0xC10D 0000—0xC10D 07FF</td>
</tr>
<tr>
<td>Dreg_even = (A0 = Dreg_lo_hi * Dreg_lo_hi) (IS)</td>
<td>0xC10D 0000—0xC10D 07FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td>0xC114 1800—0xC114 D9FF</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IS, M)</td>
<td>0xC114 1800—0xC114 D9FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td>0xC11A 1800—0xC11A 27FF</td>
</tr>
<tr>
<td>Dreg_odd = (A1 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi), Dreg_lo = (A0 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi) (IS, M)</td>
<td>0xC11A 1800—0xC11A 27FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td>0xC11C 2000—0xC11C 27FF</td>
</tr>
<tr>
<td>Dreg_even = (A0 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi), Dreg_odd = (A1 [=, +=, or –=] Dreg_lo_hi * Dreg_lo_hi) (IS, M)</td>
<td>0xC11C 2000—0xC11C 27FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td>0xC123 2000—0xC123 27FF</td>
</tr>
<tr>
<td>Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (ISS2)</td>
<td>0xC123 2000—0xC123 27FF</td>
</tr>
</tbody>
</table>
### Instructions Listed By Operation Code

Table C-23. 32-Bit Opcode Instructions (Sheet 13 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (ISS2)</td>
<td>0xC123 3000—0xC123 37FF</td>
</tr>
<tr>
<td><strong>Move Register Half</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = A0 (ISS2)</td>
<td>0xC123 3800—0xC123 39C0</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (ISS2)</td>
<td>0xC124 1800—0xC124 D9FF</td>
</tr>
<tr>
<td><strong>Vector Multiply and Multiply-Accumulate</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = (A0 +=, +=, or -=) Dreg_lo_hi * Dreg_lo_hi, Dreg_hi = (A1 +=, +=, or -=) Dreg_lo_hi * Dreg_lo_hi (ISS2)</td>
<td>0xC124 2000—0xC127 FFFF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (ISS2)</td>
<td>0xC125 1800—0xC125 D9FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (ISS2)</td>
<td>0xC126 1800—0xC126 D9FF</td>
</tr>
<tr>
<td><strong>Move Register Half</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_hi = A1 (ISS2)</td>
<td>0xC127 1800—0xC127 19C0</td>
</tr>
<tr>
<td><strong>Move Register Half</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = A0, Dreg_hi = A1 (ISS2) Dreg_hi = A1, Dreg_lo = A0 (ISS2)</td>
<td>0xC127 3800—0xC127 39C0</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (ISS2)</td>
<td>0xC128 1800—0xC128 D9FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (ISS2)</td>
<td>0xC129 1800—0xC129 D9FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (ISS2)</td>
<td>0xC12A 1800—0xC12A D9FF</td>
</tr>
<tr>
<td><strong>Vector Multiply and Multiply-Accumulate</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_even = (A0 +=, +=, or -=) Dreg_lo_hi * Dreg_lo_hi, Dreg_odd = (A1 +=, +=, or -=) Dreg_lo_hi * Dreg_lo_hi (ISS2)</td>
<td>0xC12C 2000—0xC12F FFFF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_even = (A0 += Dreg_lo_hi * Dreg_lo_hi) (ISS2)</td>
<td>0xC12D 0000—0xC12D 07FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_even = (A0 += Dreg_lo_hi * Dreg_lo_hi) (ISS2)</td>
<td>0xC12D 0800—0xC12D 0FFF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Data Register</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_even = (A0 += Dreg_lo_hi * Dreg_lo_hi) (ISS2)</td>
<td>0xC12D 1000—0xC12D 17FF</td>
</tr>
</tbody>
</table>
### Table C-23. 32-Bit Opcode Instructions (Sheet 14 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC134 1800—0xC134 D9FF</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC134 2000—0xC137 FFFF</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC135 1800—0xC135 D9FF</td>
</tr>
<tr>
<td>Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC136 1800—0xC136 D9FF</td>
</tr>
<tr>
<td>Dreg_hi = (A1 -= Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC138 1800—0xC138 D9FF</td>
</tr>
<tr>
<td>Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC139 1800—0xC139 D9FF</td>
</tr>
<tr>
<td>Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Data Register</td>
<td>0xC13A 1800—0xC13A D9FF</td>
</tr>
<tr>
<td>Dreg_odd = (A1 -= Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC13C 2000—0xC13F FFFF</td>
</tr>
<tr>
<td>Dreg_even = (A0 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi) , Dreg_odd = (A1 [=, +=, or -=] Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC163 2000—0xC163 27FF</td>
</tr>
<tr>
<td>Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (IH)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC163 2800—0xC163 2FF</td>
</tr>
<tr>
<td>Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (IH)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC163 3000—0xC163 37FF</td>
</tr>
<tr>
<td>Dreg_lo = (A0 -= Dreg_lo_hi * Dreg_lo_hi) (IH)</td>
<td></td>
</tr>
<tr>
<td>Move Register Half</td>
<td>0xC163 3800—0xC163 39C0</td>
</tr>
<tr>
<td>Dreg_lo = A0 (IH)</td>
<td></td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC164 1800—0xC164 D9FF</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IH)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply and Multiply-Accumulate</td>
<td>0xC164 2000—0xC167 FFFF</td>
</tr>
<tr>
<td>Multiply and Multiply-Accumulate to Half Register</td>
<td>0xC165 1800—0xC165 D9FF</td>
</tr>
<tr>
<td>Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IH)</td>
<td></td>
</tr>
</tbody>
</table>
## Instructions Listed By Operation Code

### Table C-23. 32-Bit Opcode Instructions (Sheet 15 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong>  &lt;br&gt; Dreg_hi = (A1 -= Dreg_lo_hi * Dreg_lo_hi) (IH)</td>
<td>0xC166 1800—0xC166 D9FF</td>
</tr>
<tr>
<td><strong>Move Register Half</strong>  &lt;br&gt; Dreg_hi = A1 (IH)</td>
<td>0xC167 1800—0xC167 19C0</td>
</tr>
<tr>
<td><strong>Move Register Half</strong>  &lt;br&gt; Dreg_lo = A0, Dreg_hi = A1 (IH)  &lt;br&gt; Dreg_hi = A1, Dreg_lo = A0 (IH)</td>
<td>0xC167 3800—0xC167 39C0</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong>  &lt;br&gt; Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IH, M)</td>
<td>0xC174 1800—0xC174 D9FF</td>
</tr>
<tr>
<td><strong>Vector Multiply and Multiply-Accumulate</strong>  &lt;br&gt; Dreg_lo = (A0 {=, +=, or -=} Dreg_lo_hi * Dreg_lo_hi),  &lt;br&gt; Dreg_hi = (A1 {=, +=, or -=} Dreg_lo_hi * Dreg_lo_hi) (IH, M)</td>
<td>0xC174 2000—0xC177 FFFF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong>  &lt;br&gt; Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IU)</td>
<td>0xC175 1800—0xC175 D9FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong>  &lt;br&gt; Dreg_hi = (A1 -= Dreg_lo_hi * Dreg_lo_hi) (IU)</td>
<td>0xC176 1800—0xC176 D9FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong>  &lt;br&gt; Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (IU)</td>
<td>0xC184 1800—0xC184 19C0</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong>  &lt;br&gt; Dreg_lo = (A0 -= Dreg_lo_hi * Dreg_lo_hi) (IU)</td>
<td>0xC184 2000—0xC184 D9FF</td>
</tr>
<tr>
<td><strong>Move Register Half</strong>  &lt;br&gt; Dreg_lo = A0 (IU)</td>
<td>0xC183 2000—0xC183 21FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong>  &lt;br&gt; Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (IU)</td>
<td>0xC183 2800—0xC183 29FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong>  &lt;br&gt; Dreg_lo = (A0 -= Dreg_lo_hi * Dreg_lo_hi) (IU)</td>
<td>0xC183 3000—0xC183 31FF</td>
</tr>
<tr>
<td><strong>Move Register Half</strong>  &lt;br&gt; Dreg_lo = A0 (IU)</td>
<td>0xC183 3800—0xC183 39C0</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong>  &lt;br&gt; Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (IU)</td>
<td>0xC184 1800—0xC184 19C0</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong>  &lt;br&gt; Dreg_lo = (A0 -= Dreg_lo_hi * Dreg_lo_hi) (IU)</td>
<td>0xC184 2000—0xC184 D9FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong>  &lt;br&gt; Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (IU)</td>
<td>0xC185 1800—0xC185 19C0</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong>  &lt;br&gt; Dreg_lo = (A0 -= Dreg_lo_hi * Dreg_lo_hi) (IU)</td>
<td>0xC186 1800—0xC186 19C0</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong>  &lt;br&gt; Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (IU)</td>
<td>0xC186 2000—0xC186 21FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong>  &lt;br&gt; Dreg_lo = (A0 -= Dreg_lo_hi * Dreg_lo_hi) (IU)</td>
<td>0xC186 2800—0xC186 29FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong>  &lt;br&gt; Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (IU)</td>
<td>0xC186 3000—0xC186 31FF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong>  &lt;br&gt; Dreg_lo = (A0 -= Dreg_lo_hi * Dreg_lo_hi) (IU)</td>
<td>0xC186 3800—0xC186 39C0</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong>  &lt;br&gt; Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (IU)</td>
<td>0xC186 4000—0xC186 D9FF</td>
</tr>
</tbody>
</table>

---

C-168 ADSP-BF53x/BF56x Blackfin Processor Programming Reference
### Instruction Opcodes

#### Table C-23. 32-Bit Opcode Instructions (Sheet 16 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Move Register Half</strong></td>
<td>0xC187 1800—0xC187 19C0</td>
</tr>
<tr>
<td>Dreg_hi = A1 (IU)</td>
<td></td>
</tr>
<tr>
<td><strong>Move Register Half</strong></td>
<td>0xC187 3800—0xC187 39C0</td>
</tr>
<tr>
<td>Dreg_lo = A0, Dreg_hi = A1 (IU)</td>
<td></td>
</tr>
<tr>
<td>Dreg_hi = A1, Dreg_lo = A0 (IU)</td>
<td></td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td>0xC194 1800—0xC194 D9FF</td>
</tr>
<tr>
<td>Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IU, M)</td>
<td></td>
</tr>
<tr>
<td><strong>Vector Multiply and Multiply-Accumulate</strong></td>
<td>0xC194 2000—0xC197 FFFF</td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td>0xC195 1800—0xC195 D9FF</td>
</tr>
<tr>
<td>Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IU, M)</td>
<td></td>
</tr>
<tr>
<td><strong>Multiply and Multiply-Accumulate to Half Register</strong></td>
<td>0xC196 1800—0xC196 D9FF</td>
</tr>
<tr>
<td>Dreg_hi = (A1 -= Dreg_lo_hi * Dreg_lo_hi) (IU, M)</td>
<td></td>
</tr>
<tr>
<td><strong>Multiply 16-Bit Operands</strong></td>
<td>0xC200 2000—0xC200 27FF</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
</tr>
<tr>
<td><strong>Multiply 16-Bit Operands</strong></td>
<td>0xC204 0000—0xC204 C1FF</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
</tr>
<tr>
<td><strong>Vector Multiply</strong></td>
<td>0xC204 2000—0xC204 E7FF</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi, Dreg_hi = Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
</tr>
<tr>
<td><strong>Multiply 16-Bit Operands</strong></td>
<td>0xC208 2000—0xC208 27FF</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
</tr>
<tr>
<td><strong>Multiply 16-Bit Operands</strong></td>
<td>0xC20C 0000—0xC20C C1FF</td>
</tr>
<tr>
<td>Dreg_odd = Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
</tr>
<tr>
<td><strong>Vector Multiply</strong></td>
<td>0xC20C 2000—0xC20C E7FF</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi, Dreg_odd = Dreg_lo_hi * Dreg_lo_hi</td>
<td></td>
</tr>
<tr>
<td><strong>Multiply 16-Bit Operands</strong></td>
<td>0xC214 0000—0xC214 C1FF</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (M)</td>
<td></td>
</tr>
<tr>
<td><strong>Vector Multiply</strong></td>
<td>0xC214 2000—0xC214 E7FF</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi, Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (M)</td>
<td></td>
</tr>
</tbody>
</table>
Table C-23. 32-Bit Opcode Instructions (Sheet 17 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC21C 0000—0xC21C C1FF</td>
</tr>
<tr>
<td>Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (M)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC21C 2000—0xC21C E7FF</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi ,</td>
<td></td>
</tr>
<tr>
<td>Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (M)</td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC220 2000—0xC220 27FF</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (S2RND)</td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC224 0000—0xC224 C1FF</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (S2RND)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC224 2000—0xC224 E7FF</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ,</td>
<td></td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (S2RND)</td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC228 2000—0xC228 27FF</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi (S2RND)</td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC22C 0000—0xC22C C1FF</td>
</tr>
<tr>
<td>Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (S2RND)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC22C 2000—0xC22C E7FF</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi ,</td>
<td></td>
</tr>
<tr>
<td>Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (S2RND)</td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC234 0000—0xC234 C1FF</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (S2RND, M)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC234 2000—0xC234 E7FF</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ,</td>
<td></td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (S2RND, M)</td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC239 0000—0xC239 C1FF</td>
</tr>
<tr>
<td>Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (S2RND, M)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC239 2000—0xC239 E7FF</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi ,</td>
<td></td>
</tr>
<tr>
<td>Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (S2RND, M)</td>
<td></td>
</tr>
</tbody>
</table>
Table C-23. 32-Bit Opcode Instructions (Sheet 18 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Multiply</td>
<td>0xC244 2000—0xC244 E7FF</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (T)</td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC254 0000—0xC254 C1FF</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (T, M)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC254 2000—0xC254 E7FF</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (T, M)</td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC280 2000—0xC280 27FF</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (FU)</td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC284 0000—0xC284 C1FF</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (FU)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC284 2000—0xC284 E7FF</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (FU)</td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC288 2000—0xC288 27FF</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi (FU)</td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC28C 0000—0xC28C C1FF</td>
</tr>
<tr>
<td>Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (FU)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC28C 2000—0xC28C E7FF</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi , Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (FU)</td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC294 0000—0xC294 C1FF</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (FU, M)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC294 2000—0xC294 E7FF</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (FU, M)</td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC29C 0000—0xC29C C1FF</td>
</tr>
<tr>
<td>Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (FU, M)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC29C 2000—0xC29C E7FF</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi , Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (FU, M)</td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC2C0 2000—0xC2C0 27FF</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (TFU)</td>
<td></td>
</tr>
</tbody>
</table>
### Table C-23. 32-Bit Opcode Instructions (Sheet 19 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC2C4 0000—0xC2C4 C1FF</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (TFU)</td>
<td>0xC2C4 0000—0xC2C4 C1FF</td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC2C4 2000—0xC2C4 E7FF</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (TFU)</td>
<td>0xC2C4 2000—0xC2C4 E7FF</td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC2D4 0000—0xC2D4 C1FF</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (TFU, M)</td>
<td>0xC2D4 0000—0xC2D4 C1FF</td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC2D4 2000—0xC2D4 E7FF</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (TFU, M)</td>
<td>0xC2D4 2000—0xC2D4 E7FF</td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC300 2000—0xC300 27FF</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (IS)</td>
<td>0xC300 2000—0xC300 27FF</td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC304 0000—0xC304 27FF</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IS)</td>
<td>0xC304 0000—0xC304 27FF</td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC304 2000—0xC304 E7FF</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IS)</td>
<td>0xC304 2000—0xC304 E7FF</td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC308 2000—0xC308 27FF</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi (IS)</td>
<td>0xC308 2000—0xC308 27FF</td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC30C 0000—0xC30C 27FF</td>
</tr>
<tr>
<td>Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (IS)</td>
<td>0xC30C 0000—0xC30C 27FF</td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC30C 2000—0xC30C E7FF</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi , Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (IS)</td>
<td>0xC30C 2000—0xC30C E7FF</td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC314 0000—0xC314 27FF</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IS, M)</td>
<td>0xC314 0000—0xC314 27FF</td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC314 2000—0xC314 E7FF</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IS, M)</td>
<td>0xC314 2000—0xC314 E7FF</td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC318 0000—0xC318 27FF</td>
</tr>
<tr>
<td>Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (IS, M)</td>
<td>0xC318 0000—0xC318 27FF</td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC318 2000—0xC318 E7FF</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi , Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (IS, M)</td>
<td>0xC318 2000—0xC318 E7FF</td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC31C 0000—0xC31C 27FF</td>
</tr>
<tr>
<td>Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (IS, M)</td>
<td>0xC31C 0000—0xC31C 27FF</td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC31C 2000—0xC31C E7FF</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi , Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (IS, M)</td>
<td>0xC31C 2000—0xC31C E7FF</td>
</tr>
</tbody>
</table>
### Instruction Opcodes

#### Table C-23. 32-Bit Opcode Instructions (Sheet 20 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC320 2000—0xC320 27FF</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (ISS2)</td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC324 0000—0xC324 C1FF</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (ISS2)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC324 2000—0xC324 E7FF</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (ISS2)</td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC328 2000—0xC328 27FF</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi (ISS2)</td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC32C 0000—0xC32C C1FF</td>
</tr>
<tr>
<td>Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (ISS2)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC32C 2000—0xC32C E7FF</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi , Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (ISS2)</td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC334 0000—0xC334 C1FF</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (ISS2, M)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC334 2000—0xC334 E7FF</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (ISS2, M)</td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC33C 0000—0xC33C C1FF</td>
</tr>
<tr>
<td>Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (ISS2, M)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC33C 2000—0xC33C E7FF</td>
</tr>
<tr>
<td>Dreg_even = Dreg_lo_hi * Dreg_lo_hi , Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (ISS2, M)</td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC360 2000—0xC360 27FF</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (IH)</td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC364 0000—0xC364 C1FF</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IH)</td>
<td></td>
</tr>
<tr>
<td>Vector Multiply</td>
<td>0xC364 2000—0xC364 E7FF</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IH)</td>
<td></td>
</tr>
<tr>
<td>Multiply 16-Bit Operands</td>
<td>0xC374 0000—0xC374 C1FF</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IH, M)</td>
<td></td>
</tr>
</tbody>
</table>
### Instructions Listed By Operation Code

Table C-23. 32-Bit Opcode Instructions (Sheet 21 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vector Multiply</strong></td>
<td>0xC374 2000—</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IH, M)</td>
<td></td>
</tr>
<tr>
<td><strong>Multiply 16-Bit Operands</strong></td>
<td>0xC380 2000—</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (IU)</td>
<td></td>
</tr>
<tr>
<td><strong>Multiply 16-Bit Operands</strong></td>
<td>0xC384 0000—</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IU)</td>
<td></td>
</tr>
<tr>
<td><strong>Vector Multiply</strong></td>
<td>0xC384 2000—</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IU)</td>
<td></td>
</tr>
<tr>
<td><strong>Multiply 16-Bit Operands</strong></td>
<td>0xC394 0000—</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IU, M)</td>
<td></td>
</tr>
<tr>
<td><strong>Vector Multiply</strong></td>
<td>0xC394 2000—</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IU, M)</td>
<td></td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC400 0000—</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg</td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC400 1000—</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg (CO)</td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC400 2000—</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg (S)</td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC400 3000—</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg (SC0)</td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC400 4000—</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>– Dreg</td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC400 5000—</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>– Dreg (CO)</td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC400 6000—</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>– Dreg (S)</td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC400 7000—</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>– Dreg (SC0)</td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC400 8000—</td>
</tr>
<tr>
<td>Dreg = Dreg –</td>
<td>+ Dreg</td>
</tr>
</tbody>
</table>
### Instruction Opcodes

#### Table C-23. 32-Bit Opcode Instructions (Sheet 22 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC400 9000—</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg (CO)</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC400 A000—</td>
</tr>
<tr>
<td>Dreg = Dreg –</td>
<td>– Dreg (S)</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC400 B000—</td>
</tr>
<tr>
<td>Dreg = Dreg –</td>
<td>– Dreg (SC0)</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC400 C000—</td>
</tr>
<tr>
<td>Dreg = Dreg –</td>
<td>– Dreg (CO)</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC400 D000—</td>
</tr>
<tr>
<td>Dreg = Dreg –</td>
<td>– Dreg (S)</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC400 E000—</td>
</tr>
<tr>
<td>Dreg = Dreg –</td>
<td>– Dreg (SC0)</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 0000—</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg, Dreg = Dreg –</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 1000—</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg, Dreg = Dreg –</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 2000—</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg, Dreg = Dreg –</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 3000—</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg, Dreg = Dreg –</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 4000—</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg, Dreg = Dreg –</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 5000—</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg, Dreg = Dreg –</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 6000—</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg, Dreg = Dreg –</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 7000—</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg, Dreg = Dreg –</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 8000—</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg, Dreg = Dreg –</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 9000—</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg, Dreg = Dreg –</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 A000—</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg, Dreg = Dreg –</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 B000—</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg, Dreg = Dreg –</td>
</tr>
<tr>
<td>Vector Add / Subtract</td>
<td>0xC401 C000—</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>+ Dreg, Dreg = Dreg –</td>
</tr>
</tbody>
</table>
### Instructions Listed By Operation Code

#### Table C-23. 32-Bit Opcode Instructions (Sheet 23 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = Dreg + Dreg, Dreg = Dreg – Dreg (CO, ASL)</td>
<td>0xC401 D000—0xC401 DFFFF</td>
</tr>
<tr>
<td>Dreg = Dreg + Dreg, Dreg = Dreg – Dreg (S, ASL)</td>
<td>0xC401 E000—0xC401 EFFFF</td>
</tr>
<tr>
<td>Dreg = Dreg + Dreg, Dreg = Dreg – Dreg (SCO, ASL)</td>
<td>0xC401 F000—0xC401 FFFFF</td>
</tr>
<tr>
<td><strong>Add</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo + Dreg_lo (NS)</td>
<td>0xC402 0000—0xC402 0E3F</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo + Dreg_lo (S)</td>
<td>0xC402 2000—0xC402 2E3F</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo + Dreg_hi (NS)</td>
<td>0xC402 4000—0xC402 4E3F</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo + Dreg_hi (S)</td>
<td>0xC402 6000—0xC402 6E3F</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_hi + Dreg_lo (NS)</td>
<td>0xC402 8000—0xC402 8E3F</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_hi + Dreg_lo (S)</td>
<td>0xC402 A000—0xC402 AE3F</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_hi + Dreg_hi (NS)</td>
<td>0xC402 C000—0xC402 CE3F</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_hi + Dreg_hi (S)</td>
<td>0xC402 E000—0xC402 EE3F</td>
</tr>
<tr>
<td><strong>Subtract</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo – Dreg_lo (NS)</td>
<td>0xC403 0000—0xC403 0E3F</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo – Dreg_lo (S)</td>
<td>0xC403 2000—0xC403 2E3F</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo – Dreg_hi (NS)</td>
<td>0xC403 4000—0xC403 4E3F</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo – Dreg_hi (S)</td>
<td>0xC403 6000—0xC403 6E3F</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_hi – Dreg_lo (NS)</td>
<td>0xC403 8000—0xC403 8E3F</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_hi – Dreg_hi (NS)</td>
<td>0xC403 E000—0xC403 EFFFF</td>
</tr>
</tbody>
</table>
### Instruction Opcodes

Table C-23. 32-Bit Opcode Instructions (Sheet 24 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtract Dreg_lo = Dreg_hi – Dreg_lo (S)</td>
<td>0xC403 A000—0xC403 AE3F</td>
</tr>
<tr>
<td>Subtract Dreg_lo = Dreg_hi – Dreg_hi (NS)</td>
<td>0xC403 C000—0xC403 CE3F</td>
</tr>
<tr>
<td>Subtract Dreg_lo = Dreg_hi – Dreg_hi (S)</td>
<td>0xC403 E000—0xC403 EE3F</td>
</tr>
<tr>
<td>Add Dreg = Dreg + Dreg (NS)</td>
<td>0xC404 0000—0xC404 0E3F</td>
</tr>
<tr>
<td>Add Dreg = Dreg + Dreg (S)</td>
<td>0xC404 2000—0xC404 2E3F</td>
</tr>
<tr>
<td>Subtract Dreg = Dreg – Dreg (NS)</td>
<td>0xC404 4000—0xC404 4E3F</td>
</tr>
<tr>
<td>Subtract Dreg = Dreg – Dreg (S)</td>
<td>0xC404 6000—0xC404 6E3F</td>
</tr>
<tr>
<td>Vector Add / Subtract Dreg = Dreg + Dreg, Dreg = Dreg – Dreg</td>
<td>0xC404 8000—0xC404 8FFF</td>
</tr>
<tr>
<td>Vector Add / Subtract Dreg = Dreg + Dreg, Dreg = Dreg – Dreg (S)</td>
<td>0xC404 A000—0xC404 AFFF</td>
</tr>
<tr>
<td>Add/Subtract-Prescale Up Dreg_lo = Dreg + Dreg (RND12)</td>
<td>0xC405 0000—0xC405 0E3F</td>
</tr>
<tr>
<td>Add/Subtract-Prescale Up Dreg_lo = Dreg – Dreg (RND12)</td>
<td>0xC405 4000—0xC405 4E3F</td>
</tr>
<tr>
<td>Add/Subtract-Prescale Down Dreg_lo = Dreg + Dreg (RND20)</td>
<td>0xC405 9000—0xC405 9E3F</td>
</tr>
<tr>
<td>Add/Subtract-Prescale Down Dreg_lo = Dreg – Dreg (RND20)</td>
<td>0xC405 D000—0xC405 DE3F</td>
</tr>
<tr>
<td>Vector Maximum Dreg = MAX (Dreg, Dreg) (V)</td>
<td>0xC406 0000—0xC406 0E3F</td>
</tr>
<tr>
<td>Vector Minimum Dreg = MIN (Dreg, Dreg) (V)</td>
<td>0xC406 4000—0xC406 4E3F</td>
</tr>
<tr>
<td>Vector Absolute Value Dreg = ABS Dreg (V)</td>
<td>0xC406 8000—0xC406 8E38</td>
</tr>
</tbody>
</table>
### Instructions Listed By Operation Code

#### Table C-23. 32-Bit Opcode Instructions (Sheet 25 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Maximum</strong> Dreg = MAX (Dreg, Dreg)</td>
<td>0xC407 0000—0xC407 0E3F</td>
</tr>
<tr>
<td><strong>Minimum</strong> Dreg = MIN (Dreg, Dreg)</td>
<td>0xC407 4000—0xC407 4E3F</td>
</tr>
<tr>
<td><strong>Absolute Value</strong> Dreg = ABS Dreg</td>
<td>0xC407 8000—0xC407 8E3F</td>
</tr>
<tr>
<td><strong>Negate (Two's-Complement)</strong> Dreg = – Dreg (NS)</td>
<td>0xC407 C000—0xC407 CFC0</td>
</tr>
<tr>
<td><strong>Negate (Two's-Complement)</strong> Dreg = – Dreg (S)</td>
<td>0xC407 E000—0xC407 EFC0</td>
</tr>
<tr>
<td><strong>Load Immediate</strong> A0 = 0</td>
<td>0xC408 003F</td>
</tr>
<tr>
<td><strong>Saturate</strong> A0 = A0 (S)</td>
<td>0xC408 203F</td>
</tr>
<tr>
<td><strong>Load Immediate</strong> A1 = 0</td>
<td>0xC408 403F</td>
</tr>
<tr>
<td><strong>Saturate</strong> A1 = A1 (S)</td>
<td>0xC408 603F</td>
</tr>
<tr>
<td><strong>Load Immediate</strong> A1 = A0 = 0</td>
<td>0xC408 803F</td>
</tr>
<tr>
<td><strong>Saturate</strong> A1 = A1 (S), A0 = A0 (S)</td>
<td>0xC408 A03F</td>
</tr>
<tr>
<td><strong>Move Register Half</strong> A0.L = Dreg_lo</td>
<td>0xC409 0000—0xC409 0038</td>
</tr>
<tr>
<td><strong>Move Register Half</strong> A0.X = Dreg_lo</td>
<td>0xC409 4000—0xC409 4038</td>
</tr>
<tr>
<td><strong>Move Register Half</strong> A1.L = Dreg_lo</td>
<td>0xC409 8000—0xC409 8038</td>
</tr>
<tr>
<td><strong>Move Register Half</strong> A1.X = Dreg_lo</td>
<td>0xC409 C000—0xC409 C038</td>
</tr>
<tr>
<td><strong>Move Register Half</strong> Dreg_lo = A0.X</td>
<td>0xC40A 003F—0xC40A 0E00</td>
</tr>
</tbody>
</table>
### Instruction Opcodes

Table C-23. 32-Bit Opcode Instructions (Sheet 26 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move Register Half</td>
<td>0xC40A 403F—0xC40A 4E00</td>
</tr>
<tr>
<td>Dreg_lo = A1.X</td>
<td></td>
</tr>
<tr>
<td>Modify-Increment</td>
<td>0xC40B 003F—0xC40B 0E00</td>
</tr>
<tr>
<td>Dreg = (A0 += A1)</td>
<td></td>
</tr>
<tr>
<td>Modify-Increment</td>
<td>0xC40B 403F—0xC40B 4E00</td>
</tr>
<tr>
<td>Dreg_lo = (A0 += A1)</td>
<td></td>
</tr>
<tr>
<td>Modify-Increment</td>
<td>0xC40B 803F</td>
</tr>
<tr>
<td>A0 += A1</td>
<td></td>
</tr>
<tr>
<td>Modify-Increment</td>
<td>0xC40B A03F</td>
</tr>
<tr>
<td>A0 += A1 (W32)</td>
<td></td>
</tr>
<tr>
<td>Modify-Decrement</td>
<td>0xC40B C03F</td>
</tr>
<tr>
<td>A0 -= A1</td>
<td></td>
</tr>
<tr>
<td>Modify-Decrement</td>
<td>0xC40B E03F</td>
</tr>
<tr>
<td>A0 -= A1 (W32)</td>
<td></td>
</tr>
<tr>
<td>Add on Sign</td>
<td>0xC40C 0000—0xC40C 0E38</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo = SIGN(Dreg_hi) * Dreg_hi + SIGN(Dreg_lo) * Dreg_lo</td>
<td></td>
</tr>
<tr>
<td>Dual 16-Bit Accumulator Extraction with Addition</td>
<td>0xC40C 403F—0xC40C 4FC0</td>
</tr>
<tr>
<td>Round to Half Word</td>
<td>0xC40C C000—0xC40C CE38</td>
</tr>
<tr>
<td>Dreg_lo = Dreg (RND)</td>
<td></td>
</tr>
<tr>
<td>Vector Search</td>
<td>0xC40D 0000—0xC40D 2FFF</td>
</tr>
<tr>
<td>(Dreg, Dreg) = SEARCH Dreg (GT)</td>
<td></td>
</tr>
<tr>
<td>Vector Search</td>
<td>0xC40D 4000—0xC40D 6FFF</td>
</tr>
<tr>
<td>(Dreg, Dreg) = SEARCH Dreg (GE)</td>
<td></td>
</tr>
<tr>
<td>Vector Search</td>
<td>0xC40D 8000—0xC40D AFF8</td>
</tr>
<tr>
<td>(Dreg, Dreg) = SEARCH Dreg (LT)</td>
<td></td>
</tr>
<tr>
<td>Vector Search</td>
<td>0xC40D C000—0xC40D EFF8</td>
</tr>
<tr>
<td>(Dreg, Dreg) = SEARCH Dreg (LE)</td>
<td></td>
</tr>
<tr>
<td>Negate (Two's-Complement)</td>
<td>0xC40E 003F</td>
</tr>
<tr>
<td>A0 = − A0</td>
<td></td>
</tr>
<tr>
<td>Negate (Two's-Complement)</td>
<td>0xC40E 403F</td>
</tr>
<tr>
<td>A0 = − A1</td>
<td></td>
</tr>
</tbody>
</table>
### Instructions Listed By Operation Code

#### Table C-23. 32-Bit Opcode Instructions (Sheet 27 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Negate (Two's-Complement)</strong>&lt;br&gt;A1 = – A1, A0 = – A0</td>
<td>0xC40E C03F</td>
</tr>
<tr>
<td><strong>Vector Negate (Two's-Complement)</strong>&lt;br&gt;Dreg = – Dreg (V)</td>
<td>0xC40F 0000—0xC40F CE38</td>
</tr>
<tr>
<td><strong>Absolute Value</strong>&lt;br&gt;A0 = ABS A0</td>
<td>0xC410 0000</td>
</tr>
<tr>
<td><strong>Absolute Value</strong>&lt;br&gt;A0 = ABS A1</td>
<td>0xC410 403F</td>
</tr>
<tr>
<td><strong>Absolute Value</strong>&lt;br&gt;A1 = ABS A1, A0 = ABS A0</td>
<td>0xC410 C03F</td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong>&lt;br&gt;Dreg = A1 + A0, Dreg = A1 – A0</td>
<td>0xC411 003F—0xC411 0FC0</td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong>&lt;br&gt;Dreg = A1 + A0, Dreg = A1 – A0 (S)</td>
<td>0xC411 203F—0xC411 2FC0</td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong>&lt;br&gt;Dreg = A0 + A1, Dreg = A0 – A1</td>
<td>0xC411 403F—0xC411 4FC0</td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong>&lt;br&gt;Dreg = A0 + A1, Dreg = A0 – A1 (S)</td>
<td>0xC411 603F—0xC411 6FC0</td>
</tr>
<tr>
<td><strong>Quad 8-Bit Subtract-Absolute-Accumulate</strong>&lt;br&gt;SAA (Dreg_pair, Dreg_pair)</td>
<td>0xC412 0000—0xC412 003F</td>
</tr>
<tr>
<td><strong>Quad 8-Bit Subtract-Absolute-Accumulate</strong>&lt;br&gt;SAA (Dreg_pair, Dreg_pair) (R)</td>
<td>0xC412 2000—0xC412 203F</td>
</tr>
<tr>
<td><strong>Quad 8-Bit Subtract-Absolute-Accumulate</strong>&lt;br&gt;SAA (Dreg_pair, Dreg_pair) (T)</td>
<td>0xC412 4000—0xC412 4E3F</td>
</tr>
<tr>
<td><strong>Quad 8-Bit Subtract-Absolute-Accumulate</strong>&lt;br&gt;SAA (Dreg_pair, Dreg_pair) (T, R)</td>
<td>0xC412 6000—0xC412 6E3F</td>
</tr>
<tr>
<td><strong>Disable Alignment Exception for Load</strong>&lt;br&gt;DISALGNEXCPT</td>
<td>0xC412 C000</td>
</tr>
<tr>
<td><strong>Quad 8-Bit Average-Byte</strong>&lt;br&gt;Dreg = BYTEOP1P (Dreg_pair, Dreg_pair)</td>
<td>0xC414 0000—0xC414 0E3F</td>
</tr>
<tr>
<td><strong>Quad 8-Bit Average-Byte</strong>&lt;br&gt;Dreg = BYTEOP1P (Dreg_pair, Dreg_pair) (R)</td>
<td>0xC414 2000—0xC414 2E3F</td>
</tr>
<tr>
<td><strong>Quad 8-Bit Average-Byte</strong>&lt;br&gt;Dreg = BYTEOP1P (Dreg_pair, Dreg_pair) (T)</td>
<td>0xC414 4000—0xC414 4E3F</td>
</tr>
<tr>
<td><strong>Quad 8-Bit Average-Byte</strong>&lt;br&gt;Dreg = BYTEOP1P (Dreg_pair, Dreg_pair) (T, R)</td>
<td>0xC414 6000—0xC414 6E3F</td>
</tr>
</tbody>
</table>
## Instruction Opcodes

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Quad 8-Bit Add</strong></td>
<td>0xC415 0000—0xC415 0FFF</td>
</tr>
<tr>
<td>(Dreg, Dreg) = BYTEOP16P (Dreg_pair, Dreg_pair)</td>
<td></td>
</tr>
<tr>
<td><strong>Quad 8-Bit Add</strong></td>
<td>0xC415 2000—0xC415 2FFF</td>
</tr>
<tr>
<td>(Dreg, Dreg) = BYTEOP16P (Dreg_pair, Dreg_pair) (R)</td>
<td></td>
</tr>
<tr>
<td><strong>Quad 8-Bit Subtract</strong></td>
<td>0xC415 4000—0xC415 4FFF</td>
</tr>
<tr>
<td>(Dreg, Dreg) = BYTEOP16M (Dreg_pair, Dreg_pair)</td>
<td></td>
</tr>
<tr>
<td><strong>Quad 8-Bit Subtract</strong></td>
<td>0xC415 6000—0xC415 6FFF</td>
</tr>
<tr>
<td>(Dreg, Dreg) = BYTEOP16M (Dreg_pair, Dreg_pair) (R)</td>
<td></td>
</tr>
<tr>
<td><strong>Quad 8-Bit Average-Half Word</strong></td>
<td>0xC416 0000—0xC416 0E3F</td>
</tr>
<tr>
<td>Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (RNDL)</td>
<td></td>
</tr>
<tr>
<td><strong>Quad 8-Bit Average-Half Word</strong></td>
<td>0xC416 2000—0xC416 2E3F</td>
</tr>
<tr>
<td>Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (RNDL, R)</td>
<td></td>
</tr>
<tr>
<td><strong>Quad 8-Bit Average-Half Word</strong></td>
<td>0xC416 4000—0xC416 6E3F</td>
</tr>
<tr>
<td>Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (TL)</td>
<td></td>
</tr>
<tr>
<td><strong>Quad 8-Bit Average-Half Word</strong></td>
<td>0xC416 6000—0xC416 7E3F</td>
</tr>
<tr>
<td>Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (TL, R)</td>
<td></td>
</tr>
<tr>
<td><strong>Dual 16-Bit Add / Clip</strong></td>
<td>0xC417 0000—0xC417 1E3F</td>
</tr>
<tr>
<td>Dreg = BYTEOP3P (Dreg_pair, Dreg_pair) (LO)</td>
<td></td>
</tr>
<tr>
<td><strong>Dual 16-Bit Add / Clip</strong></td>
<td>0xC417 2000—0xC417 3E3F</td>
</tr>
<tr>
<td>Dreg = BYTEOP3P (Dreg_pair, Dreg_pair) (LO, R)</td>
<td></td>
</tr>
<tr>
<td><strong>Quad 8-Bit Pack</strong></td>
<td>0xC418 0000—0xC418 0E3F</td>
</tr>
<tr>
<td>Dreg = BYTEPACK (Dreg, Dreg)</td>
<td></td>
</tr>
<tr>
<td><strong>Quad 8-Bit Unpack</strong></td>
<td>0xC418 4000—0xC418 4FFF8</td>
</tr>
<tr>
<td>(Dreg, Dreg) = BYTEUNPACK Dreg_pair</td>
<td></td>
</tr>
<tr>
<td><strong>Quad 8-Bit Unpack</strong></td>
<td>0xC418 6000—0xC418 6FF8</td>
</tr>
<tr>
<td>(Dreg, Dreg) = BYTEUNPACK Dreg_pair (R)</td>
<td></td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC421 0000—0xC421 0FFF</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>– Dreg, Dreg = Dreg –</td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC421 1000—0xC421 1FFF</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>– Dreg, Dreg = Dreg –</td>
</tr>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td>0xC421 2000—0xC421 2FFF</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>– Dreg, Dreg = Dreg –</td>
</tr>
</tbody>
</table>
Instructions Listed By Operation Code

Table C-23. 32-Bit Opcode Instructions (Sheet 29 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vector Add / Subtract</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>– Dreg, Dreg = Dreg –</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>– Dreg, Dreg = Dreg –</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>– Dreg, Dreg = Dreg –</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>– Dreg, Dreg = Dreg –</td>
</tr>
<tr>
<td>Dreg = Dreg +</td>
<td>– Dreg, Dreg = Dreg –</td>
</tr>
<tr>
<td><strong>Add</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo + Dreg_lo (NS)</td>
<td>0xC422 0000—0xC422 0E3F</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo + Dreg_lo (S)</td>
<td>0xC422 2000—0xC422 2E3F</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo + Dreg_hi (NS)</td>
<td>0xC422 4000—0xC422 4E3F</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo + Dreg_hi (S)</td>
<td>0xC422 6000—0xC422 6E3F</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_hi + Dreg_lo (NS)</td>
<td>0xC422 8000—0xC422 8E3F</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_hi + Dreg_lo (S)</td>
<td>0xC422 A000—0xC422 AE3F</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_hi + Dreg_hi (NS)</td>
<td>0xC422 C000—0xC422 CE3F</td>
</tr>
</tbody>
</table>
### Instruction Opcodes

#### Table C-23. 32-Bit Opcode Instructions (Sheet 30 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>0xC422 E000—0xC422 EE3F</td>
</tr>
<tr>
<td>Subtract</td>
<td>0xC423 0000—0xC423 0E3F</td>
</tr>
<tr>
<td>Subtract</td>
<td>0xC423 2000—0xC423 2E3F</td>
</tr>
<tr>
<td>Subtract</td>
<td>0xC423 4000—0xC423 4E3F</td>
</tr>
<tr>
<td>Subtract</td>
<td>0xC423 6000—0xC423 6E3F</td>
</tr>
<tr>
<td>Subtract</td>
<td>0xC423 8000—0xC423 8E3F</td>
</tr>
<tr>
<td>Subtract</td>
<td>0xC423 A000—0xC423 AE3F</td>
</tr>
<tr>
<td>Subtract</td>
<td>0xC423 C000—0xC423 CE3F</td>
</tr>
<tr>
<td>Subtract</td>
<td>0xC423 E000—0xC423 EE3F</td>
</tr>
<tr>
<td>Add/Subtract-Prescale Up</td>
<td>0xC425 0000—0xC425 0E3F</td>
</tr>
<tr>
<td>Add/Subtract-Prescale Up</td>
<td>0xC425 4000—0xC425 4E3F</td>
</tr>
<tr>
<td>Add/Subtract-Prescale Down</td>
<td>0xC425 9000—0xC425 9E3F</td>
</tr>
<tr>
<td>Add/Subtract-Prescale Down</td>
<td>0xC425 D000—0xC425 DE3F</td>
</tr>
<tr>
<td>Move Register Half</td>
<td>0xC429 0000—0xC429 0038</td>
</tr>
<tr>
<td>Move Register Half</td>
<td>0xC429 8000—0xC429 8038</td>
</tr>
<tr>
<td>Modify-Increment</td>
<td>0xC42B 403F—0xC42B 4E00</td>
</tr>
</tbody>
</table>
**Table C-23. 32-Bit Opcode Instructions (Sheet 31 of 40)**

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Round to Half Word</strong></td>
<td>0xC42C C000—0xC42C CE38</td>
</tr>
<tr>
<td>Dreg_hi = Dreg (RND)</td>
<td></td>
</tr>
<tr>
<td><strong>Negate (Two’s-Complement)</strong></td>
<td>0xC42E 003F</td>
</tr>
<tr>
<td>A1 = – A0</td>
<td></td>
</tr>
<tr>
<td><strong>Negate (Two’s-Complement)</strong></td>
<td>0xC42E 403F</td>
</tr>
<tr>
<td>A1 = – A1</td>
<td></td>
</tr>
<tr>
<td><strong>Absolute Value</strong></td>
<td>0xC430 003F</td>
</tr>
<tr>
<td>A1 = ABS A0</td>
<td></td>
</tr>
<tr>
<td><strong>Absolute Value</strong></td>
<td>0xC430 403F</td>
</tr>
<tr>
<td>A1 = ABS A1</td>
<td></td>
</tr>
<tr>
<td><strong>Quad 8-Bit Average-Half Word</strong></td>
<td>0xC43C 0000—0xC43C 0E3F</td>
</tr>
<tr>
<td>Dreg = BYTEOP2P(Dreg_pair, Dreg_pair) (RNDH)</td>
<td></td>
</tr>
<tr>
<td><strong>Quad 8-Bit Average-Half Word</strong></td>
<td>0xC43C 2000—0xC43C 2E3F</td>
</tr>
<tr>
<td>Dreg = BYTEOP2P(Dreg_pair, Dreg_pair) (RND, R)</td>
<td></td>
</tr>
<tr>
<td><strong>Quad 8-Bit Average-Half Word</strong></td>
<td>0xC43C 4000—0xC43C 6E3F</td>
</tr>
<tr>
<td>Dreg = BYTEOP2P(Dreg_pair, Dreg_pair) (TH)</td>
<td></td>
</tr>
<tr>
<td><strong>Quad 8-Bit Average-Half Word</strong></td>
<td>0xC43C 6000—0xC43C 7E3F</td>
</tr>
<tr>
<td>Dreg = BYTEOP2P(Dreg_pair, Dreg_pair) (TH, R)</td>
<td></td>
</tr>
<tr>
<td><strong>Dual 16-Bit Add / Clip</strong></td>
<td>0xC43D 0000—0xC43D 0E3F</td>
</tr>
<tr>
<td>Dreg = BYTEOP3P(Dreg_pair, Dreg_pair) (HI)</td>
<td></td>
</tr>
<tr>
<td><strong>Dual 16-Bit Add / Clip</strong></td>
<td>0xC43D 2000—0xC43D 1E3F</td>
</tr>
<tr>
<td>Dreg = BYTEOP3P(Dreg_pair, Dreg_pair) (HI, R)</td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td>0xC600 0000—0xC600 0E3F</td>
</tr>
<tr>
<td>Dreg_lo = ASHIFT Dreg_lo BY Dreg_lo</td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td>0xC600 1000—0xC600 1E3F</td>
</tr>
<tr>
<td>Dreg_lo = ASHIFT Dreg_hi BY Dreg_lo</td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td>0xC600 2000—0xC600 2E3F</td>
</tr>
<tr>
<td>Dreg_hi = ASHIFT Dreg_lo BY Dreg_lo</td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td>0xC600 3000—0xC600 3E3F</td>
</tr>
<tr>
<td>Dreg_hi = ASHIFT Dreg_hi BY Dreg_lo</td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td>0xC600 4000—0xC600 4E3F</td>
</tr>
<tr>
<td>Dreg_lo = ASHIFT Dreg_lo BY Dreg_lo (S)</td>
<td></td>
</tr>
</tbody>
</table>
### Instruction Opcodes

#### Table C-23. 32-Bit Opcode Instructions (Sheet 32 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = ASHIFT Dreg_hi BY Dreg_lo (S)</td>
<td>0xC600 5000—  0xC600 5E3F</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_hi = ASHIFT Dreg_lo BY Dreg_lo (S)</td>
<td>0xC600 6000—  0xC600 6E3F</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_hi = ASHIFT Dreg_hi BY Dreg_lo (S)</td>
<td>0xC600 7000—  0xC600 7E3F</td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = LSHIFT Dreg_lo BY Dreg_lo</td>
<td>0xC600 8000—  0xC600 8E3F</td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = LSHIFT Dreg_hi BY Dreg_lo</td>
<td>0xC600 9000—  0xC600 9E3F</td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_hi = LSHIFT Dreg_hi BY Dreg_lo</td>
<td>0xC600 A000—  0xC600 AE3F</td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_hi = LSHIFT Dreg_lo BY Dreg_lo</td>
<td>0xC600 B000—  0xC600 BE3F</td>
</tr>
<tr>
<td><strong>Vector Arithmetic Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = ASHIFT Dreg BY Dreg_lo (V)</td>
<td>0xC601 0000—  0xC601 0E3F</td>
</tr>
<tr>
<td><strong>Vector Arithmetic Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = ASHIFT Dreg BY Dreg_lo (V, S)</td>
<td>0xC601 4000—  0xC601 4E3F</td>
</tr>
<tr>
<td><strong>Vector Logical Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = LSHIFT Dreg BY Dreg_lo (V)</td>
<td>0xC601 8000—  0xC601 8E3F</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = ASHIFT Dreg BY Dreg_lo</td>
<td>0xC602 0000—  0xC602 0E3F</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = ASHIFT Dreg_by Dreg_lo (S)</td>
<td>0xC602 4000—  0xC602 4E3F</td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = LSHIFT Dreg_by Dreg_lo (S)</td>
<td>0xC602 8000—  0xC602 8E3F</td>
</tr>
<tr>
<td><strong>Rotate</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = ROT Dreg_by Dreg_lo</td>
<td>0xC602 C000—  0xC602 CE3F</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td></td>
</tr>
<tr>
<td>A0 = ASHIFT A0 BY Dreg_lo</td>
<td>0xC603 0000—  0xC603 0038</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td></td>
</tr>
<tr>
<td>A1 = ASHIFT A1 BY Dreg_lo</td>
<td>0xC603 1000—  0xC603 1038</td>
</tr>
</tbody>
</table>

ADSP-BF53x/BF56x Blackfin Processor Programming Reference C-185
### Instructions Listed By Operation Code

#### Table C-23. 32-Bit Opcode Instructions (Sheet 33 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Logical Shift</em></td>
<td>0xC603 4000—0xC603 4038</td>
</tr>
<tr>
<td>A0 = LSHIFT A0 BY Dreg_lo</td>
<td>0xC603 4038</td>
</tr>
<tr>
<td><em>Logical Shift</em></td>
<td>0xC603 5000—0xC603 5038</td>
</tr>
<tr>
<td>A1 = LSHIFT A1 BY Dreg_lo</td>
<td>0xC603 5038</td>
</tr>
<tr>
<td><em>Rotate</em></td>
<td>0xC603 8000—0xC603 8038</td>
</tr>
<tr>
<td>A0 = ROT A0 BY Dreg_lo</td>
<td>0xC603 8038</td>
</tr>
<tr>
<td><em>Rotate</em></td>
<td>0xC603 9000—0xC603 9038</td>
</tr>
<tr>
<td>A1 = ROT A1 BY Dreg_lo</td>
<td>0xC603 9038</td>
</tr>
<tr>
<td><em>Vector Pack</em></td>
<td>0xC604 0000—0xC604 0E3F</td>
</tr>
<tr>
<td>Dreg = PACK (Dreg_lo, Dreg_lo)</td>
<td>0xC604 0E3F</td>
</tr>
<tr>
<td><em>Vector Pack</em></td>
<td>0xC604 4000—0xC604 4E3F</td>
</tr>
<tr>
<td>Dreg = PACK (Dreg_lo, Dreg_hi)</td>
<td>0xC604 4E3F</td>
</tr>
<tr>
<td><em>Vector Pack</em></td>
<td>0xC604 8000—0xC604 8E3F</td>
</tr>
<tr>
<td>Dreg = PACK (Dreg_hi, Dreg_lo)</td>
<td>0xC604 8E3F</td>
</tr>
<tr>
<td><em>Vector Pack</em></td>
<td>0xC604 C000—0xC604 CE3F</td>
</tr>
<tr>
<td>Dreg = PACK (Dreg_hi, Dreg_hi)</td>
<td>0xC604 CE3F</td>
</tr>
<tr>
<td><em>Sign Bit</em></td>
<td>0xC605 0000—0xC605 0E07</td>
</tr>
<tr>
<td>Dreg_lo = SIGNBITS Dreg</td>
<td>0xC605 0E07</td>
</tr>
<tr>
<td><em>Sign Bit</em></td>
<td>0xC605 4000—0xC605 4E07</td>
</tr>
<tr>
<td>Dreg_lo = SIGNBITS Dreg_lo</td>
<td>0xC605 4E07</td>
</tr>
<tr>
<td><em>Sign Bit</em></td>
<td>0xC605 8000—0xC605 8E07</td>
</tr>
<tr>
<td>Dreg_lo = SIGNBITS Dreg_hi</td>
<td>0xC605 8E07</td>
</tr>
<tr>
<td><em>Sign Bit</em></td>
<td>0xC606 0000—0xC606 0E00</td>
</tr>
<tr>
<td>Dreg_lo = SIGNBITS A0</td>
<td>0xC606 0E00</td>
</tr>
<tr>
<td><em>Sign Bit</em></td>
<td>0xC606 4000—0xC606 4E00</td>
</tr>
<tr>
<td>Dreg_lo = SIGNBITS A1</td>
<td>0xC606 4E00</td>
</tr>
<tr>
<td><em>Ones-Population Count</em></td>
<td>0xC606 C000—0xC606 CE07</td>
</tr>
<tr>
<td>Dreg_lo = ONES Dreg</td>
<td>0xC606 CE07</td>
</tr>
<tr>
<td><em>Exponent Detection</em></td>
<td>0xC607 0000—0xC607 0E3F</td>
</tr>
<tr>
<td>Dreg_lo = EXPADJ (Dreg, Dreg_lo)</td>
<td>0xC607 0E3F</td>
</tr>
<tr>
<td><em>Exponent Detection</em></td>
<td>0xC607 4000—0xC607 4E3F</td>
</tr>
<tr>
<td>Dreg_lo = EXPADJ (Dreg, Dreg_lo) (V)</td>
<td>0xC607 4E3F</td>
</tr>
</tbody>
</table>

---

C-186 ADSP-BF53x/BF56x Blackfin Processor Programming Reference
### Instruction Opcodes

#### Table C-23. 32-Bit Opcode Instructions (Sheet 34 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exponent Detection</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = EXPADJ (Dreg_lo, Dreg_lo)</td>
<td>0xC607 8000—0xC607 8E3F</td>
</tr>
<tr>
<td>Dreg_lo = EXPADJ (Dreg_hi, Dreg_lo)</td>
<td>0xC607 C000—0xC607 CE3F</td>
</tr>
<tr>
<td><strong>Bit Multiplex</strong></td>
<td></td>
</tr>
<tr>
<td>BITMUX (Dreg, Dreg, A0) (ASR)</td>
<td>0xC608 0000—0xC608 003F</td>
</tr>
<tr>
<td>BITMUX (Dreg, Dreg, A0) (ASL)</td>
<td>0xC608 4000—0xC608 403F</td>
</tr>
<tr>
<td><strong>Compare-Select (VIT_MAX)</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = VIT_MAX (Dreg) (ASL)</td>
<td>0xC609 0000—0xC609 0E07</td>
</tr>
<tr>
<td>Dreg_lo = VIT_MAX (Dreg) (ASR)</td>
<td>0xC609 4000—0xC609 4E07</td>
</tr>
<tr>
<td><strong>Compare-Select (VIT_MAX)</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = VIT_MAX (Dreg, Dreg) (ASL)</td>
<td>0xC609 8000—0xC609 8E07</td>
</tr>
<tr>
<td>Dreg = VIT_MAX (Dreg, Dreg) (ASR)</td>
<td>0xC609 C000—0xC609 CE07</td>
</tr>
<tr>
<td><strong>Bit Field Extraction</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = EXTRACT (Dreg, Dreg_lo) (Z)</td>
<td>0xC60A 0000—0xC60A 0E3F</td>
</tr>
<tr>
<td>Dreg = EXTRACT (Dreg, Dreg_lo) (X)</td>
<td>0xC60A 4000—0xC60A 4E3F</td>
</tr>
<tr>
<td><strong>Bit Field Deposit</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = DEPOSIT (Dreg, Dreg)</td>
<td>0xC60A 8000—0xC60A 8E3F</td>
</tr>
<tr>
<td>Dreg = DEPOSIT (Dreg, Dreg) (X)</td>
<td>0xC60A C000—0xC60A CE3F</td>
</tr>
<tr>
<td><strong>Bit Wise Exclusive OR</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_lo = CC = BXORSHIFT (A0, Dreg)</td>
<td>0xC60B 0000—0xC60B 0E38</td>
</tr>
<tr>
<td>Dreg_lo = CC = BXOR (A0, Dreg)</td>
<td>0xC60B 4000—0xC60B 4E38</td>
</tr>
<tr>
<td><strong>Bit Wise Exclusive OR</strong></td>
<td></td>
</tr>
<tr>
<td>A0 = BXORSHIFT (A0, A1, CC)</td>
<td>0xC60C 0000</td>
</tr>
<tr>
<td>Dreg_lo = CC = BXOR (A0, A1, CC)</td>
<td>0xC60C 4000—0xC60C 4E00</td>
</tr>
</tbody>
</table>
## Instructions Listed By Operation Code

Table C-23. 32-Bit Opcode Instructions (Sheet 35 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Byte Align</strong>&lt;br&gt;Dreg = ALIGN8 (Dreg, Dreg)</td>
<td>0xC60D 0000—0xC60D 0E3F</td>
</tr>
<tr>
<td><strong>Byte Align</strong>&lt;br&gt;Dreg = ALIGN16 (Dreg, Dreg)</td>
<td>0xC60D 4000—0xC60D 4E3F</td>
</tr>
<tr>
<td><strong>Byte Align</strong>&lt;br&gt;Dreg = ALIGN24 (Dreg, Dreg)</td>
<td>0xC60D 8000—0xC60D 8E3F</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong>&lt;br&gt;Dreg_lo = Dreg_lo &gt;&gt;&gt; uimm4</td>
<td>0xC680 0180—0xC680 0FFF</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong>&lt;br&gt;Dreg_lo = Dreg_hi &gt;&gt;&gt; uimm4</td>
<td>0xC680 1180—0xC680 1FFF</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong>&lt;br&gt;Dreg_hi = Dreg_lo &gt;&gt;&gt; uimm4</td>
<td>0xC680 2180—0xC680 2FFF</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong>&lt;br&gt;Dreg_hi = Dreg_hi &gt;&gt;&gt; uimm4</td>
<td>0xC680 3180—0xC680 3FFF</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong>&lt;br&gt;Dreg_hi = Dreg_lo &lt;&lt;&lt; uimm4 (S)</td>
<td>0xC680 4000—0xC680 4E7F</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong>&lt;br&gt;Dreg_hi = Dreg_hi &lt;&lt;&lt; uimm4 (S)</td>
<td>0xC680 5000—0xC680 5E7F</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong>&lt;br&gt;Dreg_hi = Dreg_lo &lt;&lt;&lt; uimm4 (S)</td>
<td>0xC680 6000—0xC680 6E7F</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong>&lt;br&gt;Dreg_hi = Dreg_hi &lt;&lt;&lt; uimm4 (S)</td>
<td>0xC680 7000—0xC680 7E7F</td>
</tr>
<tr>
<td><strong>Logical Shift</strong>&lt;br&gt;Dreg_lo = Dreg_lo &lt;&lt; uimm4</td>
<td>0xC680 8000—0xC680 8E7F</td>
</tr>
<tr>
<td><strong>Logical Shift</strong>&lt;br&gt;Dreg_lo = Dreg_lo &gt;&gt; uimm4</td>
<td>0xC680 9000—0xC680 9E7F</td>
</tr>
<tr>
<td><strong>Logical Shift</strong>&lt;br&gt;Dreg_lo = Dreg_hi &lt;&lt; uimm4</td>
<td>0xC680 A000—0xC680 AE7F</td>
</tr>
</tbody>
</table>
### Instruction Opcodes

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Logical Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo &gt;&gt; uimm4</td>
<td>0xC680 A180—0xC680 AFFF</td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_hi = Dreg_hi &lt;&lt; uimm4</td>
<td>0xC680 B000—0xC680 BE7F</td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg_hi = Dreg_hi &gt;&gt; uimm4</td>
<td>0xC680 B180—0xC680 BFFF</td>
</tr>
<tr>
<td><strong>Vector Arithmetic Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = Dreg &gt;&gt; uimm5 (V)</td>
<td>0xC681 0100—0xC681 0FFF</td>
</tr>
<tr>
<td><strong>Vector Arithmetic Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = Dreg &lt;&lt; uimm5 (V, S)</td>
<td>0xC681 4000—0xC681 4EFF</td>
</tr>
<tr>
<td><strong>Vector Logical Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = Dreg &lt;&lt; uimm4 (V)</td>
<td>0xC681 8000—0xC681 8EFF</td>
</tr>
<tr>
<td><strong>Vector Logical Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = Dreg &gt;&gt; uimm4 (V)</td>
<td>0xC681 8180—0xC681 8FFF</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = Dreg &gt;&gt; uimm5</td>
<td>0xC682 0100—0xC682 0FFF</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = Dreg &lt;&lt; uimm5 (S)</td>
<td>0xC682 4000—0xC682 4EFF</td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = Dreg &lt;&lt; uimm5</td>
<td>0xC682 8000—0xC682 8EFF</td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = Dreg &gt;&gt; uimm5</td>
<td>0xC682 8100—0xC682 8FFF</td>
</tr>
<tr>
<td><strong>Rotate</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = ROT Dreg BY imm6</td>
<td>0xC682 C000—0xC682 CFFF</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td></td>
</tr>
<tr>
<td>A0 = A0 &gt;&gt;&gt; uimm5</td>
<td>0xC683 0100—0xC683 01F8</td>
</tr>
<tr>
<td><strong>Arithmetic Shift</strong></td>
<td></td>
</tr>
<tr>
<td>A1 = A1 &gt;&gt;&gt; uimm5</td>
<td>0xC683 1100—0xC683 11F8</td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td></td>
</tr>
<tr>
<td>A0 = A0 &lt;&lt;&lt; uimm5</td>
<td>0xC683 4000—0xC683 40F8</td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td></td>
</tr>
<tr>
<td>A0 = A0 &gt;&gt;&gt; uimm5</td>
<td>0xC683 4100—0xC683 41F8</td>
</tr>
</tbody>
</table>

Table C-23. 32-Bit Opcode Instructions (Sheet 36 of 40)
Table C-23. 32-Bit Opcode Instructions (Sheet 37 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Logical Shift</strong></td>
<td></td>
</tr>
<tr>
<td>$A1 = A1 \ll \text{uimm5}$</td>
<td>0xC683 5000—0xC683 50F8</td>
</tr>
<tr>
<td><strong>Logical Shift</strong></td>
<td></td>
</tr>
<tr>
<td>$A1 = A1 \gg \text{uimm5}$</td>
<td>0xC683 5100—0xC683 51F8</td>
</tr>
<tr>
<td><strong>Rotate</strong></td>
<td></td>
</tr>
<tr>
<td>$A0 = \text{ROT A0 BY imm6}$</td>
<td>0xC683 8000—0xC683 81F8</td>
</tr>
<tr>
<td><strong>Rotate</strong></td>
<td></td>
</tr>
<tr>
<td>$A1 = \text{ROT A1 BY imm6}$</td>
<td>0xC683 9000—0xC683 91F8</td>
</tr>
<tr>
<td><strong>No Op</strong></td>
<td></td>
</tr>
<tr>
<td>MNOP when issued in parallel with two compatible load/store instructions</td>
<td>0xC803 1800</td>
</tr>
</tbody>
</table>

**Zero Overhead Loop Setup**

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP loop_name LC0</td>
<td>0xE080 0000—0xE08F 03FF</td>
</tr>
<tr>
<td>LOOP_BEGIN loop_name</td>
<td></td>
</tr>
<tr>
<td>LOOP_END loop_name</td>
<td></td>
</tr>
<tr>
<td>LSETUP (pcrel5m2, pcrel11m2) LC0</td>
<td></td>
</tr>
<tr>
<td>... where the address of LOOP_BEGIN determines pcrel5m2, and the address of LOOP_END determines pcrel11m2.</td>
<td></td>
</tr>
</tbody>
</table>

**Zero Overhead Loop Setup**

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP loop_name LC1</td>
<td>0xE090 0000—0xE09F 03FF</td>
</tr>
<tr>
<td>LOOP_BEGIN loop_name</td>
<td></td>
</tr>
<tr>
<td>LOOP_END loop_name</td>
<td></td>
</tr>
<tr>
<td>LSETUP (pcrel5m2, pcrel11m2) LC1</td>
<td></td>
</tr>
<tr>
<td>... where the address of LOOP_BEGIN determines pcrel5m2, and the address of LOOP_END determines pcrel11m2.</td>
<td></td>
</tr>
</tbody>
</table>

**Zero Overhead Loop Setup**

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP loop_name LC0 = Preg</td>
<td>0xE0A0 0000—0xE0AF F3FF</td>
</tr>
<tr>
<td>LOOP_BEGIN loop_name</td>
<td></td>
</tr>
<tr>
<td>LOOP_END loop_name</td>
<td></td>
</tr>
<tr>
<td>LSETUP (pcrel5m2, pcrel11m2) LC0 = Preg</td>
<td></td>
</tr>
<tr>
<td>... where the address of LOOP_BEGIN determines pcrel5m2, and the address of LOOP_END determines pcrel11m2.</td>
<td></td>
</tr>
</tbody>
</table>
### Table C-23. 32-Bit Opcode Instructions (Sheet 38 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero Overhead Loop Setup</td>
<td>0xE0B0 0000—0xE0BF F3FF</td>
</tr>
<tr>
<td>Loop loop_name LC1 = Preg</td>
<td>0xE0E0 0000—0xE0AF F3FF</td>
</tr>
<tr>
<td>Loop loop_name</td>
<td>0xE0F0 0000—0xE0FF F3FF</td>
</tr>
<tr>
<td>Load Immediate</td>
<td>0xE100 0000—0xE11F FFFF</td>
</tr>
<tr>
<td>Load Immediate</td>
<td>0xE120 0000—0xE13F FFFF</td>
</tr>
<tr>
<td>Load Immediate</td>
<td>0xE140 0000—0xE15F FFFF</td>
</tr>
<tr>
<td>Load Immediate</td>
<td>0xE180 0000—0xE19F FFFF</td>
</tr>
<tr>
<td>Jump</td>
<td>0xE200 0000—0xE2FF FFFF</td>
</tr>
<tr>
<td>Call</td>
<td>0xE300 0000—0xE3FF FFFF</td>
</tr>
</tbody>
</table>
### Instructions Listed By Operation Code

Table C-23. 32-Bit Opcode Instructions (Sheet 39 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Load Data Register</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = [ Preg + uimm17m4 ]</td>
<td>0xE400 0000—0xE4EF 7FFF</td>
</tr>
<tr>
<td><strong>Load Data Register</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = [ Preg – uimm17m4 ]</td>
<td>0xE400 8000—0xE43F FFFF</td>
</tr>
<tr>
<td><strong>Load Half Word, Zero Extended</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = W [ Preg + uimm16m2 ] (Z)</td>
<td>0xE440 0000—0xE47F FFFF</td>
</tr>
<tr>
<td><strong>Load Half Word, Zero Extended</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = W [ Preg – uimm16m2 ] (Z)</td>
<td>0xE440 8000—0xE47F FFFF</td>
</tr>
<tr>
<td><strong>Load Byte, Zero Extended</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = B [ Preg + uimm15 ] (Z)</td>
<td>0xE480 0000—0xE4BF FFFF</td>
</tr>
<tr>
<td><strong>Load Byte, Zero Extended</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = B [ Preg – uimm15 ] (Z)</td>
<td>0xE480 8000—0xE4BF FFFF</td>
</tr>
<tr>
<td><strong>Load Pointer Register</strong></td>
<td></td>
</tr>
<tr>
<td>Preg = [ Preg + uimm17m4 ]</td>
<td>0xE500 0000—0xE53F FFFF</td>
</tr>
<tr>
<td><strong>Load Pointer Register</strong></td>
<td></td>
</tr>
<tr>
<td>Preg = [ Preg – uimm17m4 ]</td>
<td>0xE500 8000—0xE53F FFFF</td>
</tr>
<tr>
<td><strong>Load Half Word, Sign Extended</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = W [ Preg + uimm16m2 ] (X)</td>
<td>0xE540 0000—0xE57F FFFF</td>
</tr>
<tr>
<td><strong>Load Half Word, Sign Extended</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = W [ Preg – uimm16m2 ] (X)</td>
<td>0xE540 8000—0xE57F FFFF</td>
</tr>
<tr>
<td><strong>Load Byte, Sign Extended</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = B [ Preg + uimm15 ] (X)</td>
<td>0xE580 0000—0xE5BF FFFF</td>
</tr>
<tr>
<td><strong>Load Byte, Sign Extended</strong></td>
<td></td>
</tr>
<tr>
<td>Dreg = B [ Preg – uimm15 ] (X)</td>
<td>0xE580 8000—0xE5BF FFFF</td>
</tr>
<tr>
<td><strong>Store Data Register</strong></td>
<td></td>
</tr>
<tr>
<td>[ Preg + uimm17m4 ] = Dreg</td>
<td>0xE600 0000—0xE63F FFFF</td>
</tr>
<tr>
<td><strong>Store Data Register</strong></td>
<td></td>
</tr>
<tr>
<td>[ Preg – uimm17m4 ] = Dreg</td>
<td>0xE600 8000—0xE63F FFFF</td>
</tr>
<tr>
<td><strong>Store Low Data Register Half</strong></td>
<td></td>
</tr>
<tr>
<td>W [ Preg + uimm16m2 ] = Dreg</td>
<td>0xE640 0000—0xE67F FFFF</td>
</tr>
<tr>
<td><strong>Store Low Data Register Half</strong></td>
<td></td>
</tr>
<tr>
<td>W [ Preg – uimm16m2 ] = Dreg</td>
<td>0xE640 8000—0xE67F FFFF</td>
</tr>
</tbody>
</table>
## Instruction Opcodes

Table C-23. 32-Bit Opcode Instructions (Sheet 40 of 40)

<table>
<thead>
<tr>
<th>Instruction and Version</th>
<th>Opcode Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Store Byte</strong></td>
<td></td>
</tr>
<tr>
<td>B [ Preg + uimm15 ] = Dreg</td>
<td>0xE680 0000—0xE6BF 7FFF</td>
</tr>
<tr>
<td><strong>Store Byte</strong></td>
<td></td>
</tr>
<tr>
<td>B [ Preg – uimm15] = Dreg</td>
<td>0xE680 8000—0xE6BF FFFF</td>
</tr>
<tr>
<td><strong>Store Pointer Register</strong></td>
<td></td>
</tr>
<tr>
<td>[ Preg + uimm17m4 ] = Preg</td>
<td>0xE700 0000—0xE7EF 8FFF</td>
</tr>
<tr>
<td><strong>Store Pointer Register</strong></td>
<td></td>
</tr>
<tr>
<td>[ Preg – uimm17m4 ] = Preg</td>
<td>0xE700 8000—0xE73F FFFF</td>
</tr>
<tr>
<td><strong>Linkage</strong></td>
<td></td>
</tr>
<tr>
<td>LINK uimm18m4</td>
<td>0xE800 0000—0xE800 FFFF</td>
</tr>
<tr>
<td><strong>Linkage</strong></td>
<td></td>
</tr>
<tr>
<td>UNLINK</td>
<td>0xE801 0000</td>
</tr>
</tbody>
</table>
Instructions Listed By Operation Code
D  NUMERIC FORMATS

ADSP-BF53x/BF56x Blackfin family processors support 8-, 16-, 32-, and 40-bit fixed-point data in hardware. Special features in the computation units allow support of other formats in software. This appendix describes various aspects of these data formats. It also describes how to implement a block floating-point format in software.

Unsigned or Signed: Two’s-complement Format

Unsigned integer numbers are positive, and no sign information is contained in the bits. Therefore, the value of an unsigned integer is interpreted in the usual binary sense. The least significant words of multiple-precision numbers are treated as unsigned numbers.

Signed numbers supported by the ADSP-BF53x/BF56x Blackfin family are in two’s-complement format. Signed-magnitude, one’s-complement, binary-coded decimal (BCD) or excess-n formats are not supported.

Integer or Fractional Data Formats

The ADSP-BF53x/BF56x Blackfin family supports both fractional and integer data formats. In an integer, the radix point is assumed to lie to the right of the least significant bit (LSB), so that all magnitude bits have a weight of 1 or greater. This format is shown in Figure D-1. Note in two’s-complement format, the sign bit has a negative weight.
Integer or Fractional Data Formats

In a fractional format, the assumed radix point lies within the number, so that some or all of the magnitude bits have a weight of less than 1. In the format shown in Figure D-2, the assumed radix point lies to the left of the three LSBs, and the bits have the weights indicated.

The native formats for the Blackfin processor family are a signed fractional 1.M format and an unsigned fractional 0.N format, where N is the number of bits in the data word and M = N – 1.

The notation used to describe a format consists of two numbers separated by a period (,); the first number is the number of bits to the left of the radix point, the second is the number of bits to the right of the radix point. For example, 16.0 format is an integer format; all bits lie to the left of the radix point. The format in Figure D-2 is 13.3.

---

Figure D-1. Integer Format

In a fractional format, the assumed radix point lies within the number, so that some or all of the magnitude bits have a weight of less than 1. In the format shown in Figure D-2, the assumed radix point lies to the left of the three LSBs, and the bits have the weights indicated.

The native formats for the Blackfin processor family are a signed fractional 1.M format and an unsigned fractional 0.N format, where N is the number of bits in the data word and M = N – 1.

The notation used to describe a format consists of two numbers separated by a period (,); the first number is the number of bits to the left of the radix point, the second is the number of bits to the right of the radix point. For example, 16.0 format is an integer format; all bits lie to the left of the radix point. The format in Figure D-2 is 13.3.
Figure D-2. Example of Fractional Format
Integer or Fractional Data Formats

Table D-1 shows the ranges of signed numbers representable in the fractional formats that are possible with 16 bits.

Table D-1. Fractional Formats and Their Ranges

<table>
<thead>
<tr>
<th>Format</th>
<th># of Integer Bits</th>
<th># of Fractional Bits</th>
<th>Max Positive Value (0x7FFF) In Decimal</th>
<th>Max Negative Value (0x8000) In Decimal</th>
<th>Value of 1 LSB (0x0001) In Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.15</td>
<td>1</td>
<td>15</td>
<td>0.999969482421875</td>
<td>-1.0</td>
<td>0.000030517578125</td>
</tr>
<tr>
<td>2.14</td>
<td>2</td>
<td>14</td>
<td>1.999938964843750</td>
<td>-2.0</td>
<td>0.000061035156250</td>
</tr>
<tr>
<td>3.13</td>
<td>3</td>
<td>13</td>
<td>3.999877929687500</td>
<td>-4.0</td>
<td>0.000122070312500</td>
</tr>
<tr>
<td>4.12</td>
<td>4</td>
<td>12</td>
<td>7.999755859375000</td>
<td>-8.0</td>
<td>0.000244140625000</td>
</tr>
<tr>
<td>5.11</td>
<td>5</td>
<td>11</td>
<td>15.999511718750000</td>
<td>-16.0</td>
<td>0.0004882812500000</td>
</tr>
<tr>
<td>6.10</td>
<td>6</td>
<td>10</td>
<td>31.999023437500000</td>
<td>-32.0</td>
<td>0.0009765625000000</td>
</tr>
<tr>
<td>7.9</td>
<td>7</td>
<td>9</td>
<td>63.998046875000000</td>
<td>-64.0</td>
<td>0.0019531250000000</td>
</tr>
<tr>
<td>8.8</td>
<td>8</td>
<td>8</td>
<td>127.996093750000000</td>
<td>-128.0</td>
<td>0.0039062500000000</td>
</tr>
<tr>
<td>9.7</td>
<td>9</td>
<td>7</td>
<td>255.992187500000000</td>
<td>-256.0</td>
<td>0.0078125000000000</td>
</tr>
<tr>
<td>10.6</td>
<td>10</td>
<td>6</td>
<td>511.984375000000000</td>
<td>-512.0</td>
<td>0.0156250000000000</td>
</tr>
<tr>
<td>11.5</td>
<td>11</td>
<td>5</td>
<td>1023.968750000000000</td>
<td>-1024.0</td>
<td>0.0312500000000000</td>
</tr>
<tr>
<td>12.4</td>
<td>12</td>
<td>4</td>
<td>2047.937500000000000</td>
<td>-2048.0</td>
<td>0.06250000000000000</td>
</tr>
<tr>
<td>13.3</td>
<td>13</td>
<td>3</td>
<td>4095.875000000000000</td>
<td>-4096.0</td>
<td>0.12500000000000000</td>
</tr>
<tr>
<td>14.2</td>
<td>14</td>
<td>2</td>
<td>8191.750000000000000</td>
<td>-8192.0</td>
<td>0.250000000000000000</td>
</tr>
<tr>
<td>15.1</td>
<td>15</td>
<td>1</td>
<td>16383.500000000000000</td>
<td>-16384.0</td>
<td>0.5000000000000000000</td>
</tr>
<tr>
<td>16.0</td>
<td>16</td>
<td>0</td>
<td>32767.000000000000000</td>
<td>-32768.0</td>
<td>1.0000000000000000000</td>
</tr>
</tbody>
</table>

---

D-4 ADSP-BF53x/BF56x Blackfin Processor Programming Reference
**Binary Multiplication**

In addition and subtraction, both operands must be in the same format (signed or unsigned, radix point in the same location), and the result format is the same as the input format. Addition and subtraction are performed the same way whether the inputs are signed or unsigned.

In multiplication, however, the inputs can have different formats, and the result depends on their formats. The ADSP-BF53x/BF56x Blackfin family assembly language allows you to specify whether the inputs are both signed, both unsigned, or one of each (mixed-mode). The location of the radix point in the result can be derived from its location in each of the inputs. This is shown in Figure D-3. The product of two 16-bit numbers is a 32-bit number. If the inputs’ formats are M.N and P.Q, the product has the format (M + P).(N + Q). For example, the product of two 13.3 numbers is a 26.6 number. The product of two 1.15 numbers is a 2.30 number.

![Figure D-3. Format of Multiplier Result](image)

<table>
<thead>
<tr>
<th>General Rule</th>
<th>4-bit Example</th>
<th>16-bit Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>M.N</td>
<td>1.111 (1.3 Format)</td>
<td>5.3 x 5.3</td>
</tr>
<tr>
<td>x P.Q</td>
<td>x 11.11 (2.2 Format)</td>
<td>x 10.6</td>
</tr>
<tr>
<td>(M + P).(N + Q)</td>
<td>1111</td>
<td>1.15 x 1.15</td>
</tr>
<tr>
<td></td>
<td>1111</td>
<td>2.30</td>
</tr>
<tr>
<td></td>
<td>1111</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1111</td>
<td></td>
</tr>
<tr>
<td></td>
<td>111.00001 (3.5 Format = (1 + 2).(2 + 3) )</td>
<td>2.30</td>
</tr>
</tbody>
</table>

Figure D-3. Format of Multiplier Result
Fractional Mode And Integer Mode

A product of 2 two’s-complement numbers has two sign bits. Since one of these bits is redundant, you can shift the entire result left one bit. Additionally, if one of the inputs was a 1.15 number, the left shift causes the result to have the same format as the other input (with 16 bits of additional precision). For example, multiplying a 1.15 number by a 5.11 number yields a 6.26 number. When shifted left one bit, the result is a 5.27 number, or a 5.11 number plus 16 LSBs.

The ADSP-BF53x/BF56x Blackfin family provides a means (a signed fractional mode) by which the multiplier result is always shifted left one bit before being written to the result register. This left shift eliminates the extra sign bit when both operands are signed, yielding a result that is correctly formatted.

When both operands are in 1.15 format, the result is 2.30 (30 fractional bits). A left shift causes the multiplier result to be 1.31 which can be rounded to 1.15. Thus, if you use a signed fractional data format, it is most convenient to use the 1.15 format.

Block Floating-Point Format

A block floating-point format enables a fixed-point processor to gain some of the increased dynamic range of a floating-point format without the overhead needed to do floating-point arithmetic. However, some additional programming is required to maintain a block floating-point format.

A floating-point number has an exponent that indicates the position of the radix point in the actual value. In block floating-point format, a set (block) of data values share a common exponent. A block of fixed-point values can be converted to block floating-point format by shifting each value left by the same amount and storing the shift value as the block exponent.
Typically, block floating-point format allows you to shift out non-significant MSBs (most significant bits), increasing the precision available in each value. Block floating-point format can also be used to eliminate the possibility of a data value overflowing. See Figure D-4. Each of the three data samples shown has at least two non-significant, redundant sign bits. Each data value can grow by these two bits (two orders of magnitude) before overflowing. These bits are called guard bits.

Figure D-4. Data With Guard Bits

If it is known that a process will not cause any value to grow by more than the two guard bits, then the process can be run without loss of data. Later, however, the block must be adjusted to replace the guard bits before the next process.

Figure D-5 shows the data after processing but before adjustment. The block floating-point adjustment is performed as follows.

- Assume the output of the `SIGNBITS` instruction is \( SB \) and \( SB \) is used as an argument in the `EXPADJ` instruction. Initially, the value of \( SB \) is +2, corresponding to the two guard bits. During processing, each resulting data value is inspected by the `EXPADJ` instruction, which counts the number of redundant sign bits and adjusts \( SB \) if the
Block Floating-Point Format

The number of redundant sign bits is less than two. In this example, \( SB = +1 \) after processing, indicating the block of data must be shifted right one bit to maintain the two guard bits.

- If \( SB \) were 0 after processing, the block would have to be shifted two bits right. In either case, the block exponent is updated to reflect the shift.

1. Check for bit growth

<table>
<thead>
<tr>
<th>One Guard Bit</th>
<th>EXPADJ instruction checks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponent, adjusts SB</td>
<td></td>
</tr>
</tbody>
</table>
   | \begin{align*}
   \text{Exponent} = +2, SB = +2
   \end{align*} |                                   |
   | \begin{align*}
   \text{Exponent} = +1, SB = +1
   \end{align*} |                                   |
   | \begin{align*}
   \text{Exponent} = +4, SB = +1
   \end{align*} |                                   |

   0x1FFF = 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
   0x3FFF = 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
   0x07FF = 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1

   Sign Bit

2. Shift right to restore guard bits

<table>
<thead>
<tr>
<th>Two Guard Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponent, adjusts SB</td>
</tr>
</tbody>
</table>
   | \begin{align*}
   \text{Exponent} = +4, SB = +1
   \end{align*} |                                   |

   0x0FFF = 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
   0x1FFF = 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
   0x03FF = 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1

   Sign Bit

Figure D-5. Block Floating-point Adjustment

---

D-8 ADSP-BF53x/BF56x Blackfin Processor Programming Reference
I  INDEX

Numerics
16-bit accumulator extraction with x2 scaling, 16-bit saturation and rounding (S2RND)
multiply 16-bit operands instruction, 15-43
multiply and multiply-accumulate to data register instruction, 15-67
use with multiply instructions, 15-43, 15-58, 15-67
16-bit instructions, parallel instructions, 20-6
16-bit operations, defined with example, 2-27
32-bit accumulator extraction with x2 scaling and 32-bit saturation (ISS2)
multiply 16-bit operands instruction, 15-43
multiply and multiply-accumulate to data register instruction, 15-67
use with multiply instructions, 15-43, 15-58, 15-67
32-bit ALU/MAC instructions, parallel, 20-3
32-bit operations, 2-29

A
A0 accumulator register, 2-2
A0 overflow (AV0) bit, 2-25
A1 accumulator register, 2-2
A1 overflow (AV1) bit, 2-25
AAU (address arithmetic unit), 5-1 to 5-24
addressing examples, 5-5
architecture, 5-2
components, 5-3
figure, 5-4
instruction summary, 5-19
multi-issue operations, 5-24
abort instruction, C-100
ABS (vector absolute value) instruction, 15-3, 19-15, C-107
AC0 carry bit, 2-14, 2-25
AC1 carry bit, 2-14, 2-25
AC (address calculation), 4-7
access way / instruction address bit 11, 6-40
access way (WAYSEL[1:0]) field, 6-21
accumulator instructions
accumulator to D-register move, 9-4
accumulator to half D-register move, 9-16, 9-19
compare accumulator, 11-9
dual 16-bit accumulator extraction with addition, 18-13
accumulator registers
32-bit integer MAC instruction, 2-46
A0, 2-2
A1, 2-2
accumulators
corresponding to MACs, 1-14, C-3
description, 1-13
description of, C-2
Index

accumulators (continued)
  extension registers A0.x and A1.x, 9-15
  initializing, 8-4
  overflow arithmetic status flags, 1-15, C-7
  result registers A[1:0], 2-8, 2-36, 2-43
  saturation quantity, 1-12
  sticky overflow arithmetic status flag, 1-15

accumulator to D-register move
  instruction, 9-2
  option flags, 9-4
accumulator to half D-register move
  instruction, 9-2
  option flags, 9-4

add immediate instruction, 15-16, C-59
add instructions, 15-6, C-59
  add, C-55
  add immediate, 15-6, C-59
  add on sign, 19-3, C-107
  add with shift, 14-2, C-46
dual 16-bit add / clip, C-102
quad 8-bit add, C-103
vector add / subtract, 19-18, C-107
add on (SIGN) instruction, 19-3, C-107

address arithmetic unit. See AAU
address calculation (AC), 4-7
address collision, SRAM, 6-27

addressing (continued)
  post-modify, 5-11
  pre-modify stack pointer, 5-11
  transfers supported (table), 5-17
address-modify operation, 5-15
address pointer registers. See pointer registers (P[5:0])
address-tag compare operation, 6-13
+– (vector add / subtract) operator, 19-18
add/subtract - prescale down instruction, 15-10, C-58
add/subtract - prescale up instruction, 15-13, C-59
add with shift instruction, 14-2, C-46
ADSP-BF535 processor
  flags, A-2
  MMRs, 6-73
  special considerations, A-1
ADSP-BF53x processor arithmetic, 2-5
ALIGN16 (byte align) instruction, 18-3
ALIGN24 (byte align) instruction, 18-3
ALIGN8 (byte align) instruction, 18-3
alignment
  ALIGN16 (byte align) instruction, 18-3
  ALIGN24 (byte align) instruction, 18-3
  ALIGN8 (byte align) instruction, 18-3
  exceptions, 6-71
  memory operations, 6-71
alignment exceptions
  disabling, 5-16
  when triggered, 5-10
allocating system stack, 4-56
allreg syntax convention, 10-2
ALU, video, 2-1
ALU0 carry (AC0) bit, 2-25
ALU1 carry (AC1) bit, 2-25
ALU (arithmetic logic unit), 2-26 to 2-35
  arithmetic, 2-14
  arithmetic formats, 2-16
  data types, 2-14

I-2 ADSP-BF53x/BF56x Blackfin Processor Programming Reference
ALU (arithmetic logic unit) (continued)
defined, 1-14
functions listed, 2-26
inputs and outputs, 2-26
instruction effects on flags, 2-30
instructions, 2-30
operations, 2-26 to 2-30
status, 2-24
summary, C-3
video instructions, 2-35
Analog Devices, Inc.[7:0] field, 21-27
AND instruction, 12-2, C-43
AND operator, 2-26
AN (negative flag) bit, 2-14, 2-25
AQ bit, 2-25
architecture
Blackfin, 20-1
level 1 (L1) data memory, 6-27
level 2 (L2) data memory, 6-43
processor core, 2-2
arithmetic formats summary, 2-16 to 2-17
arithmetic logic unit. See ALU
arithmetic operations, 2-26, 15-1, 15-2
arithmetic operations instructions, C-55
>>> (arithmetic right shift) operator, 19-23
arithmetic shift (ASHIFT) instruction,
  14-7, C-46
arithmetic shifts, 2-1, 2-15, 2-48, 14-10,
  19-24
arithmetic status flags, C-6, C-7
AC0, carry (ALU0), C-7
AC1, carry (ALU1), C-7
AN, negative, C-7
AQ, divide primitive quotient, C-7
AV0, overflow (A0), C-7
AV1, overflow (A1), C-7
AVS0, sticky overflow (A0), C-7
AVS1, sticky overflow (A1), C-7
AZ, zero, C-7
CC, control code bit, C-7
arithmetic status flags (continued)
on ADSP-BF535, A-2
summarized, 1-15
V, overflow (D-register), 1-17, C-7
VS, sticky overflow (D-register), C-7
arithmetic status register (ASTAT). See
  ASTAT
array access (TAGSELB) bit, 6-21, 6-40
ASHIFT (arithmetic shift) instruction,
  14-7
ASHIFT (arithmetic shift) instruction,
summarized, 2-48
ASHIFT...BY instruction, 14-7, 19-23
= (assign) operator, 8-3
= assign (representative sample, only)
  operator, 18-3, 19-3
assembly language, 2-1
= (assign) operator, 9-2, 10-2
ASTAT (arithmetic status) register, 1-15,
  2-25, 11-1, C-6, C-7
arithmetic status flags, 1-15, 1-17
AZ status flag, 1-15
CC (control code) bit, 1-15
RND_MOD bit, 1-20
V (overflow for data register results
  copy), 1-15
VS (sticky overflow for data register
  results), 1-15
asynchronous memory controller, device
  bank control, 1-6
atomic operations, 6-72
atomic read-modify-write instruction (test
  and set byte), C-99
-- (autodecrement) operator, 10-2, 10-5
auto-decrement addressing, example, 5-10
++ (autoincrement) operator, 10-8, 10-12,
  17-7, 17-9
auto-increment addressing, example, 5-10
automatic address translation, 6-52
AV0 bit, 2-14, 2-25, 2-38

Index
Index

AVOS bit, 2-25
AV1 bit, 2-25, 2-38
AV1 overflow flag, 2-14
AV1S bit, 2-25
average, quad 8-bit average – byte instruction, 18-23, C-104
AZ bit, 2-25
AZ status flag, 1-15

B
backgnd_reg bit field deposit instruction, 13-10
bank access, L2 memory, 6-43
barrel-shifter. See shifter
base registers (B[3:0]), 5-8
AAU, 5-3
DAG, 5-13
defined, 1-14
description, 1-22, C-3
function in circular addressing, 1-22
biased rounding, 1-19, 2-19
binal point, decimal point placement, 1-16
binary multiplication, D-5
binary numbers, 2-4
bit clear (BITCLR) instruction. See BITCLR
BITCLR (bit clear) instruction, C-44
bit field deposit (DEPOSIT) instruction, 13-10, C-44
backgnd_reg, 13-10
foregnd_reg, 13-10
bit field extraction (EXTRACT) instruction, 13-16, C-44
pattern_reg, 13-16
scene_reg, 13-16
bit manipulation
bit clear, example, 2-52
bit set, example, 2-52
bit test, example, 2-52
bit toggle, example, 2-52
bit multiplex (BITMUX) instruction. See BITMUX
BITMUX (bit multiplex) instruction, C-45
bit operations instructions, 13-1, C-44
bit reverse (BREV) option
modify – increment instruction, 15-37
bit-reversed addressing, 5-15
bit-reversed carry addressing, 5-2
bits
overflow, 2-14
range of sequential, notation convention, 1-11, C-5
BITSET (bit set) instruction, 13-4, C-44
BITTGL (bit toggle) instruction, 13-6, C-44
bit toggle (BITTGL) instruction. See BITTGL
BITTST (bit test) instruction, 13-8, C-44
bit-wise exclusive-OR (BOXR) instruction, 12-10, C-43
Blackfin computational units, data size supported, 1-1
Blackfin processor, memory structure of, 1-4
Blackfin processor family
debug facilities, 21-1
instruction set introduced, 1-4
I/O memory space, 1-6
memory architecture, 1-4
native formats, D-2
parallel instructions, 20-1
block floating-point format, D-6
BMODE
bits, 3-14
state, 3-13
BOXR (bit-wise exclusive-OR) instruction, 12-10
branch
conditional, 4-19
latency, conditional branches, 4-21
branch (continued)
  latency, reducing, 4-10
  latency, unconditional, 4-21
  occurring, 4-10
  prediction, static, 4-20
  target, 4-13
  target address for unconditional branches, 4-21
  types, 4-11
  branching, 4-10, 7-5 to 7-7
  branch instructions, loops, 7-18
  B-registers (base), 5-3, 5-8, 5-13
  Breg. See base registers (B[3:0])
  buffer base address, 5-13
  buffers
    cacheability protection lookaside buffers
      (CPLBs), 6-10, 6-46, 6-47
    flushing core buffer, 16-5
    line fill, 6-34
    victim, 6-34
  BXOR instruction, 12-10
  BXORSHIFT instruction, 12-10
  byte align instruction, 18-3, C-102
  BYTEOP16M (quad 8-bit subtract)
    instruction, 18-32
  BYTEOP16P (quad 8-bit add) instruction,
    18-15
  BYTEOP1P (quad 8-bit average - byte)
    instruction, 18-19
  BYTEOP2P (quad 8-bit average - half-word) instruction, 18-24
  BYTEOP3P (dual 16-bit add/clip)
    instruction, 18-8
  BYTEPACK (quad 8-bit pack) instruction, 18-30
  BYTEUNPACK (quad 8-bit unpack)
    instruction, 18-41

C
  cache
    address collision, 6-29
    block, definition, 6-74
    coherency support, 6-71
    control instructions, C-101
    controller, 6-33
    controlling, 17-1
    data cache access, 6-33
    data cache control instructions, 6-37
    data cache invalidation, 6-38
    enabling, 6-53
    enabling or disabling, 6-6
    flushing, 6-6
    instruction cache management, 6-16
    invalidation, 6-18
    L1 memory, 6-2
    L2 memory, 6-44
    line fill, 6-14
    line fill buffer, 6-15
    locking by line, 6-16
    locking by way, 6-17
    mapping into data banks, 6-30
    modes, 6-29
    replacement policy, 6-5
    validity of cache lines, 6-11
    word fetching order, 6-14
    write memory operations, 6-35
    cacheability protection lookaside buffers.
      See CPLBs
    cache controller, replacement policy, 6-33
    cache hits
      address-tag compare, 6-13
      controller test, 6-33
      definition, 6-13, 6-74
    cache inhibited accesses, 6-72
Index

cache line
  components, 6-10
definition, 6-74
replacement unit, 6-15
states, 6-34

cache memory, programming model, 6-2

cache misses
  cache controller, 6-33
  contrasted with cache hits, 6-13
definition, 6-74
replacement policy, 6-15

cache way lock (ILOC[3:0]) field, 6-5, 6-7, 6-17

CALL instruction
  direct, 4-12
dynamic range, 4-12
indirect, 4-12
opcode range, C-13
range, 4-12
return address, 4-10
subroutines, 4-13
syntax, 7-8
carry bits, 2-14
CC (control code) bit, 1-15
diagram, 2-25
JUMP instruction, 4-10
ways of accessing, 4-18
CCEN (cycle counter enable) bit, 21-26

CEC (core event controller), 1-7, 4-48, 4-55
choice of one register within a group,
notation convention, C-5

CIRCLR (circular addressing)
enabling, 1-22
example instructions, 1-22
initializing buffer registers, 8-11, 8-24, 8-28, 8-42, 8-46, 8-50
load data register, 8-11
load high data register half, 8-24
load low data register half, 8-28
store data register, 8-41
store high data register half, 8-46
store low data register half, 8-50
supporting add immediate instructions, 15-16
supporting modify – decrement instructions, 15-35
supporting modify – increment instructions, 15-39
supporting subtract immediate instructions, 15-90
circular buffer addressing
defined, 5-12
registers, 5-12
wraparound, 5-15
circular buffers, 5-8
addressing, 5-12
contents, 1-21
maximum length of, 1-21
setting addresses for, 5-3
circular data buffers (figure), 5-14
clean, memory term definition, 6-75
CLI (disable interrupts) instruction, 6-74, 16-13
CMPLP[1:0] field, 21-17
code examples
  core MMR programming, 6-74
epilog code for nested ISR, 4-54
exception handler, 4-68
exception routine, 4-70
interrupt enabling and disabling, 6-74
load base of MMRs, 6-74
Index

code examples (continued)
loop, 4-22
prolog code for nested ISR, 4-53
recreating execution trace, 21-18
restoring control register, 6-74
using hardware loops in ISR, 4-28
code patching, 21-5
collision, address, 6-27
collision, cache address, 6-29
compare accumulator instruction, 11-9, C-41
compare data register instruction, 11-2, C-39
compare instructions
compare accumulator, 11-9, C-41
compare data register, 11-2, C-39
compare pointer, 11-6, C-40
compare-select (VIT_MAX), 19-8, C-107
compare pointer instruction, 11-6, C-40
compare-select (VIT_MAX) instruction, 19-8, C-107
computational instructions, 2-1
computational status, using, 2-24
computational units, 2-1 to 2-56
computation stalls, 4-9
compute register file, 1-2
conditional
branches defined, 4-19
branch latency, 4-21
CSYNC instruction, 6-69
instructions, 2-24, 4-3
JUMP instruction, 4-10
register move, 4-20
speculative load execution example, 6-69
condition code (CC) bit, See CC bit
condition code (CC) flag bit, 4-18
constants
imm16, 8-4
imm3, 11-2, 11-6
constants (continued)
imm3 constant, 11-2
imm6, 14-21
imm7, 8-4, 15-16
lppcrel11m2, 7-14
notation convention, 1-11, 1-12, C-5, C-6
pcrel11m2, 7-6
pcrel13m2, 7-2
pcrel25m2, 7-3, 7-8
pcrel5m2, 7-14
pcrelm2, 7-2
uimm15, 8-31, 8-34, 8-54
uimm16, 8-4
uimm16m2, 8-15, 8-19, 8-50
uimm17m4, 8-7, 8-11, 8-37, 8-41
uimm18m4, 10-17
uimm3, 11-2, 11-6
uimm4, 14-8, 14-15, 16-17, 16-20, 19-23, 19-28
uimm5, 13-2, 13-4, 13-6, 13-8, 14-8, 14-15
uimm5m2, 8-15, 8-19, 8-50
uimm6m4, 8-7, 8-11, 8-37, 8-41
uimm7m4, 8-7, 8-11, 8-37, 8-41
user_label, 7-3, 7-6, 7-8
constants, notation convention, C-6
content-addressable memory (CAM), 6-46
control code bit management instructions, C-39
control code (CC) bit
in ASTAT register, 11-1
moving status of, 11-13
controlling program flow, 4-19
control register
data memory, 6-24
instruction memory, 6-5
restoration, 6-74
conventions, xxxvi
convergent rounding, 1-19, 2-20
Index

copyback buffer, 6-35

core
architecture, 1-1 to 1-4, 2-2
double-fault condition, 4-46
double-fault reset, 3-13
waking up from idled state, 4-34

core architecture, contents of, 1-1

core event
emulation, 4-30
exception, 4-30
in EVT, 4-41
MMR location, 4-42
NIM, 4-30
reserved, 4-30
reset, 4-30

core event controller (CEC). See CEC
core event core timer, 4-30
core event vector table (table), 4-42
core interrupt latch (ILAT) register. See ILAT
core interrupt mask (IMASK) register. See IMASK
core interrupts pending (IPEND) register. See IPEND
core MMRs, 6-73, B-1
core-only software reset, 3-13, 3-16
core synchronize (CSYNC) instruction, 16-5
core synchronize instruction, C-99
core timer core event, 4-30
core timer interrupt (IVTMR) bit, 4-47
counter, cycle, 4-6, 21-23
counter register, loops, 7-17
count instructions, one’s population count, 13-26
count value (WPDACNT[15:0]) field, 21-12
count value (WPIACNT[15:0]) field, 21-7
CPLB_DIRTY bit, 6-57
CPLB_L1_AOW bit, 6-57

CPLB_L1_CHBL bit, 6-55, 6-57
CPLB_LOCK bit, 6-55, 6-57
CPLB_LRUPRIO bit, 6-5, 6-16, 6-55

CPLBs
after reset, 6-26
defined, 6-46, 6-47
disabling, 6-6
enabling, 6-6, 6-10
enabling and disabling, 6-26
L1 instruction cache, 6-10
management, 6-50
replacement policy, 6-51
CPLB_SUPV_WR bit, 6-57
CPLB_USER_RD bit, 6-55, 6-57
CPLB_USER_WR bit, 6-57
CPLB_VALID bit, 6-55, 6-57
CPLB_WT bit, 6-57
C-style indexed stack manipulation, 1-3
CSYNC (core synchronize) instruction, 16-5
customer support, xxviii
cycle
counter, 21-23, 21-24
counter register, 4-6
cycle counter enable (CCEN) bit, 21-26
CYCLES/CYCLES2[15:0] field, 21-25
CYCLES/CYCLES2[31:16] field, 21-25
CYCLES2 (execution cycle count) register, 21-24
CYCLES (execution cycle count) register, 21-24

D
DAG0 CPLB miss, 4-66
DAG0 misaligned access, 4-66
DAG0 multiple CPLB hits, 4-66
DAG0 port preference (PORT_PREF0) bit, 6-25, 6-26
DAG0 protection violation, 4-66
DAG1 CPLB miss, 4-66
DAG1 misaligned access, 4-66
DAG1 multiple CPLB hits, 4-66
DAG1 port preference (PORT_PREF1) bit, 6-24, 6-25
DAG1 protection violation, 4-66
DAG (data address generator) register addressing modes, 5-18
as processor section (table), C-3
defined, 1-14, 2-6
description summary, C-3
exceptions, 4-65
performance with reads, 6-26
register modification, 5-15
registers, 5-8
stalls, 4-9
support for branches, 4-3
dagreg, syntax, 9-3
data
flow, 2-1
memory stalls, 4-9
store format, 6-75
transfers, 2-8
watchpoints, 21-3
data, sign extending, 2-12
data, zero extending, 2-12
data[15:0] field, 6-23, 6-42
data[31:16] field, 6-23, 6-42
data[47:32] field, 6-22, 6-41
data[63:48] field, 6-22, 6-41
data address generator (DAG). See DAG
data address registers, initialization, 5-4
data address (WPDA[31:16]) field, 21-11
data arithmetic unit, diagram, 1-2
data bank access bit, 6-40
data banks, configuration, 6-30
data cache
access, 6-33
classification instructions, 6-37
control instructions, 6-37
flush instruction, C-101

Index

data cache
invalidation, 6-38
L1 memory, 6-2, 6-29
data cacheability protection lookaside buffer enable (ENDCPLB) bit, 6-25,
6-26, 6-30, 6-51
data cache line, states, 6-34
data cache line flush and invalidate (FLUSHINV) instruction. See FLUSHINV
data cache line invalidate instruction, C-101
data cache lines, instructions, 17-1
data cache prefetch (PREFETCH) instruction, 6-37, 17-3, C-101
data cache select/address bit 14, 6-40
data fetch 1 (DF1), 4-7
data fetch 2 (DF2), 4-7
data formats, 2-4 to 2-5
binary multiplication, D-5
table, 2-13
data/instruction access bit, 6-40
data memory, L1, 6-24 to 6-38
data memory control
(DMEM_CONTROL) register, 6-24, 6-47
data operations, CPLB, 6-47
data register file, 2-1, 2-2, 2-6, 2-7, 2-8
data registers
description, 1-13, C-2
groups, 2-6
names, 3-4
data SRAM, L1, 6-27
data test command
(DTEST_COMMAND) register, 6-39
data test data (DTEST_DATAx) registers, 6-41, 6-42, 17-2
data test registers, 6-38 to 6-42
data types, 2-11 to 2-23
Index

data watchpoint address control
(WPDACCTL) register, 21-12

DF1 (data fetch 1), 4-7

DF2 (data fetch 2), 4-7

DF2 stage, 4-9
direct branch address, 4-11
direct-mapped, definition, 6-74
dirty, definition, 6-75

DIRTY bit, 6-42
disable alignment exception for load
(DISALGNEXCPT) instruction. See
DISALGNEXCPT
disable interrupts (CLI) instruction, 6-74,
16-13, C-99
disabling interrupts, global, 4-48

DISALGNEXCPT (disable alignment
exception for load) instruction
cautions, 5-17
syntax, 18-6
table, C-102
uses of, 5-16
divide primitive (DIVS, DIVQ)
instructions, 2-14, 2-34, 15-19, C-60
DIVQ (divide primitive) instruction, 2-34,
15-19
DIVS (divide primitive) instruction, 2-34,
15-19
DMA bandwidth, 6-27
DMC[1:0] (L1 data memory configure)
field, 6-25, 6-27, 6-30, 6-38
DMEM_CONTROL (data memory
control) register, 6-24, 6-47
double-fault condition, 4-46
double word index (DW[1:0]) field, 6-21
DPMC (dynamic power management
controller)
Dreg_even
multiply and multiply-accumulate,
15-67
syntax, 9-3
Dreg_hi, 15-43, 15-58, 19-3
syntax, 8-45, 9-16
deferring exception processing, 4-68
deposit (bit field deposit) instruction,
13-10
debug, MP, and emulation unit registers,
B-7
debug control (DBGCTLR) register. See
DBGCTR
debug features, 21-1
dec (instruction decode), 4-7
dec stage, 4-9
deconomy

DEC (instruction decode), 4-7

DEC stage, 4-9
deferring exception processing, 4-68
deposit (bit field deposit) instruction,
Index

syntax, 8-27, 8-49, 9-10, 9-13, 9-16
Dreg_odd
multiply and multiply-accumulate, 15-67
syntax, 9-3
Dreg overflow (V) bit, 2-25
DSPID (product identification) register, 21-27
DTEST_COMMAND (data test command) register, 6-39
DTEST_DATA0 register, 6-42
DTEST_DATA1 register, 6-41
DTEST_DATAx (data test data) registers, 6-41, 6-42, 17-2
dual 16-bit accumulator extraction with addition, 18-13
dual 16-bit accumulator extraction with addition instruction, C-103
dual 16-bit add/clip (BYTEOP3P) instruction, 18-8, C-102
dual 16-bit operations, defined with example, 2-27
dual 32-bit operations, defined with example, 2-29
dual MAC operations example, 2-47
DW[1:0] (double word index) field, 6-21, 6-40
dynamic power management controller (DPMC). See DPMC

E
EAB bus, 6-8, 6-14
EBIU (external bus interface unit), 1-5, 1-6
EMU (emulation) bit, 4-40, 4-41
EMUEXCPT (force emulation) instruction, 3-9, 16-11
emulation
core event, 4-30
events, 1-7, 3-1, 4-45
force emulation instruction, 16-11
return from (RTE instruction), 7-10, 7-11
emulation mode, 1-4, 3-1, 3-9, 4-45, 21-18
EMUSW0 bit, 21-9
EMUSW1 bit, 21-9
EMUSW2 bit, 21-9
EMUSW3 bit, 21-8
EMUSW4 bit, 21-8
EMUSW5 bit, 21-8
enable interrupts (STI) instruction, 6-74, 16-15, C-99
enabling interrupts, global, 4-48
ENDCPLB (data cacheability protection looksaside buffer enable) bit, 6-25, 6-26, 6-30, 6-51
endian format, data and instruction storage, 6-65
ENICPLB (instruction CPLB enable) bit, 6-7, 6-51
EPROM, 1-6
errors
bus parity, 4-60
bus timeout, 4-60
hardware, 4-59
hardware conditions causing, 4-60
internal core, 4-60
misalignment of data, 6-71
multiple hardware, 4-60
peripheral, 4-60
evaluation of loop conditions, 4-22

event controller
  activities managed, 4-29
  MMRs, 4-38
  processor mode, 3-1
  sequencer, 4-3
  tasks, 1-6, 4-3

event handling
  activities managed, 4-29
  nesting, 1-6
  prioritization, 1-6
  types supported, 1-7

event prioritization, 1-6

events
  asynchronous, 1-6
  definition, 4-29
  emulation, 1-7, 3-1, 4-45
  exception, 1-7, 4-61
  incrementing PFMONx, 21-21
  interrupt, 1-7
  latency in servicing, 4-56
  nested, 4-40
  NMI, 1-7
  processing, 4-3
  reset, 1-7
  synchronous, 1-6
  triggering, 21-3
  types of, 1-7

event vector table (EVT). See EVT
EVT (event vector table), 4-31
EVX (exception) bit, 4-40, 4-41
EX1 (execute 1), 4-7
EX1 stage, 4-9
EX2 (execute 2), 4-7
EX2 stage, 4-9, 4-20
EXCAUSE (exception cause) field, 4-59
exception core event, 4-30
exception events, 3-4
exception routine, example code, 4-70

exceptions
  address violations not flagged, 17-3 to 17-9
  alignment, 7-3, 8-8 to 8-50, 10-3, 10-7, 10-10, 10-15, 10-19
  alignment errors prevented, 18-6 to 18-41
  attempting to write protected memory, 6-53
  by descending priority (table), 4-65
  DAG0 CPLB miss, 4-66
  DAG0 misaligned access, 4-66
  DAG0 multiple CPLB hits, 4-66
  DAG0 protection violation, 4-66
  DAG1 CPLB miss, 4-66
  DAG1 misaligned access, 4-66
  DAG1 multiple CPLB hits, 4-66
  DAG1 protection violation, 4-66
deferring, 4-68
defined, 1-7
  emulation, 16-11
  events, 4-63
  events that cause, 4-61
  force exception (EXCPT) instruction, 4-66, 16-20
  graceful instruction abort, 10-6, 10-15, 10-18
  handler, executing, 4-66
  handler routine, 16-20
  handling instructions in pipeline, 4-67
  I-fetch access exception, 4-65
  I-fetch CPLB miss, 4-65
  I-fetch misaligned access, 4-65
  I-fetch multiple CPLB hits, 4-65
  I-fetch protection violation, 4-65
  illegal combination, 4-66
  illegal instruction, 16-11
  illegal use protected resource, 4-66
  MMRs, 6-73
  MMU, 6-53
exceptions (continued)
  multiple, 4-65
  not invoked by force interrupt / reset
  instruction, 16-18
  not masked by disable interrupts
  instruction, 16-13
  prioritization, 6-52
  program flow, 4-2
  protection violation, 7-11, 9-6, 10-3,
  10-10, 16-4, 16-13, 16-15, 16-19
  protection violation, not flagged, 17-3,
  17-5, 17-7, 17-9
  resolved during synchronization, 16-5,
  16-6, 16-9
  resolving before TESTSET operation
  begins, 16-24
  resolving before TESTSET operation
  completes, 16-23
  return from (RTE) instruction, 7-11
  return from (RTX) instruction, 7-10,
  7-11
  single step, 4-66
  trace buffer, 4-66
  undefined instruction, 4-65, 8-8, 10-3
  unrecoverable event, 4-65
  watchpoint match, 4-65
  while exception handler executing, 4-67
  exclusive, definition, 6-75
  exclusive data cache line, 6-34
  exclusive-OR, 12-8
  exclusive-OR (XOR) instruction, 12-8,
  C-43
  EXCPNT (force exception) instruction,
  4-66, 16-20, C-99
  execute 1 (EX1), 4-7
  execute 2 (EX2), 4-7
  execution cycle count (CYCLES and
  CYCLES2) registers, 21-24
  execution trace, code examples, 21-18
  execution unit, components, 4-8
  EXPADJ instruction, 15-26
  exponent detection
    functions, 2-48
    instruction, C-60
  exponent logic, 2-15
  external bus interface unit. See EBIU
  external event management instructions,
  16-1, C-99
  external memory, 1-6, 6-1
  EXTRACT (bit field extraction)
    instruction, 13-16

F
Fast Fourier Transform, 5-15
FAULT[15:0] field, 6-62, 6-63
FAULT_ADDR[15:0] field, 6-64
FAULT_ADDR[31:16] field, 6-64
FAULT_DAG bit, 6-62
FAULT_ILLADDR bit, 6-62, 6-63
FAULT_RW bit, 6-62
FAULT_USERSUPV bit, 6-62, 6-63
fetch address
  described, 4-8
  incrementation, 4-8
  instruction pipeline, 4-3
fetching order, 6-14
FFT (Fast Fourier Transform) calculations,
  5-15
field deposit, 2-52
field extract, 2-52
flags
  ADSP-BF535, A-2
  arithmetic status, summarized, 1-15
  overflow, 2-14
  flash memory, 1-6
  flow control instructions, C-13
  FLUSH (data cache flush) instruction,
  6-37, 17-5, C-101
Index

FLUSHINV (data cache line flush and invalidate) instruction, 6-37, 17-7, C-101
force emulation (EMUEXCPT) instruction, 16-11, C-99
force exception (EXCPT) instruction, 16-20, C-99
force interrupt / reset (RAISE) instruction, 3-11, 16-17, C-99
foregnd_reg bit field deposit instruction, 13-10
FP (frame pointer). See frame pointer (FP) register
fractional
data format, D-1
mode, 2-15, D-6
representation, 2-5
fractional multiplier results format, 2-17
fractions
binal point, 1-16
binary convention, 1-16
multiplication, 2-45
frame pointer (FP) register, 1-13, 4-6, 5-6, 8-8, 8-12, 8-38, 10-17, C-2
fully associative, definition, 6-75
functions, exponent detection, 2-48

G
general-purpose interrupt
RAISE instruction, 4-47
SIC, 4-30
with multiple peripheral interrupts, 4-37
genreg, syntax, 9-3
global enabling and disabling interrupts, 4-48
global interrupt disable bit, 4-41
GSM (global system for mobile communications)
speech-compression routines, 2-23
speech vocoder algorithms, 2-41

H
hardware
loops, 4-21
reset, 3-12, 3-13
hardware error
core event, 4-30
core event mapping, 4-30
interrupt (HWE), 4-59, 4-60
multiple, 4-60
Harvard architecture, 1-4, 6-3
hierarchical memory structure, 1-4
high half-word Accumulator extraction with saturation and rounding (IH)
multiply and multiply-accumulate to half-register instruction, 15-58
high half-word accumulator extraction with saturation and rounding (IH)
multiply 16-bit operands instruction, 15-43
use with instructions, 15-58
HWE (hardware error interrupt), 4-59, 4-60
HWERRCAUSE (hardware error cause) field, 4-59

I
IAR (interrupt assignment register), 1-8
ICACHE instruction, 6-18
ICPLB address (ICPLB_ADDRx) registers, 6-47, 6-60
ICPLB_ADDRx (ICPLB address) registers, 6-47, 6-60
ICPLB data (ICPLB_DATAx) registers, 6-47, 6-55
ICPLB_DATAx (ICPLB data) registers, 6-47, 6-55
ICPLB fault address (ICPLB_FAULT_ADDR) register, 6-63
Index

ICPLB_FAULT_ADDR (ICPLB fault address) register, 6-63
ICPLB_STATUS (ICPLB status) register, 6-61, 6-62
identifying processor mode, 3-2
IDLE (idle) instruction, 3-9, 16-3, 16-14, 16-16, A-1, C-99
idle state
   defined, 3-9
   processor mode, 3-2
   program flow, 4-2
   transition to, 3-10
   wake up core from, 4-34
IF1 (instruction fetch 1), 4-7
IF2 (instruction fetch 2), 4-7
IF3 (instruction fetch 3), 4-7
IF3 stage, 4-9
IF CC instruction, 9-8
IF CC JUMP instruction, 7-5
I-fetch access exception, 4-65
I-fetch CPLB miss, 4-65
I-fetch misaligned access, 4-65
I-fetch multiple CPLB hits, 4-65
I-fetch protection violation, 4-65
IFLUSH (instruction cache flush)
   instruction, 6-6, 6-18, 17-9, C-101
ILAT (core interrupt latch) register
   diagram
   EXCPCT instruction, 16-20
   latched interrupt request, 4-39
   system interrupt processing, 4-31
illegal combination, exception type, 4-66
illegal use protected resource, 4-66
ILOC[3:0] (cache way lock) field, 6-5, 6-7, 6-17
IMASK (core interrupt mask) register, 4-31, 6-74
IMC (L1 instruction memory configuration) bit, 6-6, 6-7, 6-19
IMEM_CONTROL (instruction memory control) register, 6-5, 6-47
imm16 constant, 8-4
imm3 constant, 11-6
imm6 constant, 14-21
imm7 constant, 8-4, 15-16
immediate constant, C-5
immediate shift
   defined, 2-50
   example, 2-49, 2-51
immediate values, designation, 1-11
implementation[15:0] field, 21-27
index, definition, 6-75
indexed addressing
   example, 5-8
   with immediate offset, 5-10
index registers ([I[3:0]])
   add immediate instructions, 15-16
   addresses, 5-8
   defined, 1-14, 5-12
   description, 1-21, C-3
   example, 5-3
   function in circular addressing, 1-21
   load data register, 8-10
   load high data register half, 8-23, 8-27
   modify – decrement instructions, 15-34
   modify – increment instructions, 15-37
   store data register, 8-40
   store high data register half, 8-45
   store low data register half, 8-49
   subtract immediate instruction, 15-90
indirect branch address, 4-11
initialization
   loop registers, 7-15
   of data address registers, 5-4
   of interrupts, 4-34
   of length registers, 5-4
   inner loops, 4-25
   input/output loop performance, 1-21
   inputs and outputs (ALU), 2-26
Index

instruction address, 4-3
instruction-address-range watchpoints, 21-2
instruction address (WPIA[14:0]) field, 21-6
instruction address (WPIA[30:15]) field, 21-6
instruction alignment unit, 4-8
instruction cache
  coherency, 6-16
  invalidating, 17-1
  invalidation, 6-18
  L1 memory, 6-2
  management, 6-16
instruction cache flush (IFLUSH)
  instruction, 6-6, 6-18, 17-9, C-101
instruction CPLB enable (ENICPLB) bit, 6-7, 6-51
instruction decode (DEC), 4-7
instruction fetch, 4-8, 6-47
  time loop, 4-24
instruction fetch 1 (IFin1), 4-7
instruction fetch 2 (IF2), 4-7
instruction fetch 3 (IF3), 4-7
instruction loop buffer, 4-24
instruction memory control
  (IMEM_CONTROL) register, 6-5, 6-47
instruction memory unit, 4-8
instruction opcodes
  abort, C-100
  add, C-55
  add immediate, C-59
  add on sign, C-107
  add/subtract - prescale down, C-58
  add/subtract - prescale up, C-59
  add with shift, C-46
  AND, C-43
  arithmetic shift, C-46
  bit clear, C-44

instruction opcodes (continued)
  bit field deposit, C-44
  bit field extraction, C-44
  bit multiplex, C-45
  bit set, C-44
  bit test, C-44
  bit toggle, C-44
  bit-wise exclusive-OR, C-43
  byte align, C-102
  CALL, C-13
  compare accumulator, C-41
  compare data register, C-39
  compare pointer, C-40
  compare-select (VIT_MAX), C-107
  core synchronize, C-99
  data cache flush, C-101
  data cache line invalidate, C-101
  data cache prefetch, C-101
  disable alignment exception for load, C-102
  disable interrupts, C-99
  divide primitive, C-60
  dual 16-bit accumulator extraction with addition, C-103
  dual 16-bit add / clip, C-102
  enable interrupts, C-99
  exclusive-OR, C-43
  exponent detection, C-60
  force emulation, C-99
  force exception, C-99
  force interrupt / reset, C-99
  idle, C-99
  instruction cache flush, C-101
  JUMP, C-13
  linkage, C-38
  load byte – sign-extended, C-22
  load byte – zero-extended, C-22
  load data register, C-17
  load half-word – sign-extended, C-20
  load half-word – zero-extended, C-19
instruction opcodes (continued)
load high data register half, C-20
load immediate, C-16
load low data register half, C-21
load pointer register, C-17
logical shift, C-50
maximum, C-60
minimum, C-61
modify – decrement, C-61
modify – increment, C-61
move byte – sign-extended, C-36
move byte – zero extended, C-36
move CC, C-41
move conditional, C-31
move half to full word – sign-extended, C-31
move half to full word – zero-extended, C-31
move register, C-28
move register half, C-32
multiply 16-bit operands, C-62
multiply 32-bit operands, C-68
multiply and multiply-accumulate to accumulator, C-69
multiply and multiply-accumulate to data register, C-86
multiply and multiply-accumulate to half-register, C-74
negate CC, C-42
negate (two’s-complement), C-93
no op, C-99
NOT (one’s-complement), C-43
one’s-population count, C-45
OR, C-43
pop, C-37
pop multiple, C-37
push, C-37
push multiple, C-37
quad 8-bit add, C-103

instruction opcodes (continued)
quad 8-bit average – byte, C-104
quad 8-bit average – half-word, C-104
quad 8-bit pack, C-105
quad 8-bit subtract, C-106
quad 8-bit subtract-absolute-accumulate, C-106
quad 8-bit unpack, C-106
return, C-14
rotate, C-54
round to half-word (RND), C-94
saturate, C-95
shift with add, C-46
sign bit, C-95
store byte, C-27
store data register, C-24
store high data register half, C-25
store low data register half, C-26
store pointer register, C-23
subtract, C-96
subtract immediate, C-98
system synchronize, C-99
test and set byte (atomic), C-99
vector absolute value, C-107
vector add / subtract, C-107
vector arithmetic shift, C-114
vector logical shift, C-115
vector maximum, C-115
vector minimum, C-115
vector multiply, C-115
vector multiply and
multiply-accumulate, C-121
vector negate (two’s-complement), C-138
vector pack, C-138
vector search, C-138
zero-overhead loops
setup, C-14
instruction opcodes, constructing, C-1
Index

instruction pipeline
  described, 4-3
  stages, 4-7
  stages (table), 4-7

instructions
  See also specific instruction
  16-bit parallel, 20-6
  32-bit ALU/MAC, 20-3
  ALU, 2-30, 2-32
  byte align, 18-3
  conditional, 2-24, 4-3
  data cache control, 6-37
  data cache lines, 17-1
  FLUSH, C-101
  in pipeline when interrupt occurs, 4-67
  interlocked pipeline, 6-66
  issuing in parallel, 20-1
  JUMP, 7-2
  linkage, 10-17
  load / store, 6-66
  multi-issue instruction, 20-1
  multiplier, 2-38
  program flow, 7-2
  protected, 3-4
  return, 3-5, 7-10
  shifter, 2-53
  store, 20-6
  stored in memory, 6-65
  synchronizing, 6-68
  video pixel operations, 18-1
  width, 4-8

instruction set, optimization of, 1-4
instruction summary, AAU, 5-20
instruction test command
  (ITEST_COMMAND) register, 6-21
instruction test data (ITEST_DATAx)
  registers, 6-22, 17-2
instruction test registers, 6-19 to 6-23
  writing to, 6-20

instruction watchpoint address control
  (WPIACTL) register, 21-7
instruction watchpoint address count
  (WPIACNTn) registers, 21-5, 21-6
instruction watchpoint address (WPIAn)
  registers, 21-5
instruction watchpoints, control bits, 21-4
  integer
    data format, D-1
    mode, 2-15, D-6
    multiplication, 2-45
  integer multiplier results format, 2-17
internal memory
  described, 1-5
  L1 memory, 6-2
interrupt assignment register (IAR), 1-8
interrupt controller registers, B-6
interrupt handling, instructions in pipeline, 4-67
interrupt mask (IMASK) register, 16-15
interrupt priority (IPRIO) register, 6-35
interrupts
  cause, 1-7
  control of system, 4-30
  core, 6-34
  definition, 4-30
  disabling, 6-74, 16-13
  enabling, 16-15
  enabling and disabling, 6-74
  force interrupt / reset (RAISE)
    instruction, 16-17
  for peripheral, 4-33
  general-purpose, 4-30, 4-37, 4-47
  generated by peripheral, 4-31
  global enabling and disabling, 4-48
  hardware error, 4-59
  initialization, 4-34
  IPRIO register, 6-35
  multiple sources, 4-32
  nested, 4-40, 4-51
Interruptions (continued)

- NMI, return from (RTN instruction), 7-10
- Non-nested, 4-51
- Peripheral, 4-30
- Popping RETI from stack, 10-3
- Priority, 16-17
- Priority watermark, 6-35
- Processing, 4-3, 4-31
- Program flow, 4-2
- Return from interrupt (RTI) instruction, 7-10, 7-11
- Servicing, 4-48
- Shared, 4-37
- Sources, peripheral, 4-35
- Supported by CEC, 1-8
- Uninterruptable instructions, 7-11, 10-6, 10-15, 10-18, 16-23
- Vector, 16-17

Interrupt Service Routine, determining source of interrupt, 4-36

Invalidation of instruction cache, 6-18
Invalid data cache line, 6-34, 6-75
I/O memory space, 1-6
IPEND (core interrupts pending) register
IPRIO (interrupt priority) register, 6-35
IPRIO_MARK[0:3] (priority watermark) field, 6-36
I-registers (index), 5-8
Ireg registers. See index registers
ISR (interrupt service routine)
  Multiple interrupt sources, 4-32
  Using hardware loops, 4-28
ITEST_COMMAND (instruction test command) register, 6-21
ITEST_DATAx (instruction test data registers, 6-22, 17-2
IVGn bits, 4-39, 4-40, 4-41
IVHW (hardware error) bit, 4-39, 4-40, 4-41, 4-59

IVTMR (core timer interrupt) bit, 4-39, 4-40, 4-41, 4-47

J

- JTAG port, 3-16
- JUMP.0 (unknown) instruction, 4-11
- JUMP instruction
  Conditional, 4-10
  Conditional jump, 7-5
  Contrasted with CALL, 4-10
  Indirect, 4-12
  Opcodes, C-13
  Range, 4-11
  Syntax, 7-2
- JUMP.L (long jump) instruction, 4-11
- Jumps, program flow, 4-1
- JUMP.S (short jump) instruction, 4-11

L

- L1 data
  Memory, defined, 1-5
  Memory controller registers, B-1
  SRAM, 6-27
  L1 data cache bank select (DCBS) bit, 6-25, 6-26, 6-29, 6-31
  L1 data memory configure (DMC[1:0]) field, 6-25, 6-27, 6-30, 6-38
  L1 instruction memory configuration (IMC) bit, 1-5, 6-6, 6-7, 6-19
  L1 instruction memory controller registers, B-4
  L1 memory, 1-4, 1-5

  See also level 1 (L1) memory; level 1 (L1) data memory; level 1 (L1) instruction memory

  L1 scratchpad RAM, defined, 1-5
  L2 (level 2) memory, defined, 1-5
  L2 memory. See level 2 (L2) memory
  Latched interrupt request, 4-39
Index

latency
  in interrupt processing, 4-34
level 2 (L2) memory, 6-4, 6-44
  servicing events, 4-56
  when servicing interrupts, 4-48
LB (loop bottom) register, 1-14, 4-22, 4-24
LC (loop count) register, 1-14, 4-22, 4-24
leaf functions, 4-14
least recently used (LRU) algorithm
definition, 6-75
length (Lreg) registers, description, C-3
length registers, initialization, 5-4
length registers (L[3:0]), 5-3, 5-8, 5-13
defined, 1-14
description, 1-22
  function in circular addressing, 1-22
level 1 (L1) data memory, 6-24 to 6-38
  architecture, 6-27
  traffic, 6-24
level 1 (L1) instruction memory, 6-5 to 6-19
  bank architecture, 6-8
  configuration, 6-10
  DAG reference exception, 6-7
  enabled as cache, 6-53
  instruction cache, 6-10
  organization, 6-10
  subbank organization, 6-5
level 1 (L1) memory, 1-4
  See also level 1 (L1) data memory; level 1 (L1) instruction memory
about, 6-3
  address alignment, 6-7
  data cache, 6-29
  definition, 6-75
  frequency, 6-4
  overview, 6-2
  scratchpad data SRAM, 6-4
level 2 (L2) memory
  CCLK cycles, 6-1
  defined, 1-5, 6-43
  enabling cache, 6-6
  latency, 6-4, 6-44
  latency with cache off, 6-45
  latency with cache on, 6-44
  non-cacheable, 6-45
  overview, 6-4
line fill
  buffers, 6-15, 6-34
  cache, 6-14
linkage instruction, 10-17, C-38
linkage instructions. See LINK, UNLINK
LINK instruction
code sequence, 4-17
subroutine example, 4-18
syntax, 10-17
little endian, definition, 6-75
little endian byte order, 2-13
load
  operation, 6-66
  ordering, 6-67
load byte – sign-extended instruction, 8-34, C-22
load byte – zero-extended instruction, 8-31, C-22
load data register instruction, 8-10, C-17
load half-word – sign-extended instruction, 8-19, C-20
load half-word – zero-extended instruction, 8-15, C-19
load high data register half instruction, 8-23, C-20
load immediate instruction, 8-3, C-16
load instructions, 8-1
load byte – sign-extended, 8-34, C-22
load byte – zero-extended, 8-31, C-22
load half-word – sign-extended, 8-19, C-20
Index

load instructions  (continued)
  load half-word – zero-extended, 8-15, C-19
  load high data register half, 8-23, C-20
  load immediate, 8-3, C-16
  load low data register half, 8-27, C-21
  load pointer register, 8-7, C-17
  stalled, 6-34
load low data register half instruction, 8-27, C-21
load pointer register instruction, 8-7, C-17
loads, speculative execution, 6-69
load / store instructions, 5-6, C-16
logging nested interrupt, 4-55
logical operations, 2-26, 12-1, C-43
  << (logical left shift) operator, 19-23, 19-28
  >> (logical right shift) operator, 19-28
logical shift (LSHIFT) instruction, 2-48, 14-14, C-50
logical shifts, 2-1, 2-15
long jump (JUMP.L) instruction, 4-11
look-ahead address, 4-7
loopback, 4-22, 4-25
loop bottom (LB0, LB1) registers, 1-14, 4-22, 4-24, C-2
loop counter, modifying, 7-18
loop count (LC0, LC1) registers, 1-14, 4-22, 4-24, C-2
LOOP instruction, 7-13
loop PC-relative constant, 1-12, C-6
loop registers
  initialization, 7-15
  table of, 4-22
  zero-overhead, 4-6
loops
  branch instructions, 7-18
  buffer, 4-24
  conditions, evaluation, 4-22
  counter register, 7-17
  disabling, 4-23

loops  (continued)
  hardware, 4-21
  inner, 4-25
  instruction fetch time, 4-24
  interrupted, 4-27
  last instruction restrictions, 7-17
  loopback, 4-22
  loop bottom register, 9-7
  loop count register, 9-7
  LOOP instruction, 7-13
  loop top register, 9-7
  LSETUP (loop setup) instruction, 7-13
  modifying loop counter, 7-18
  nested, 7-16
  outer, 4-25
  program flow, 4-1
  registers, 4-6, 4-22, 7-15
  restoring, 4-27
  saving and resuming, example, 4-27
  small loop count values, 7-16
  span, 7-15
  termination conditions, 4-3
  top and bottom addresses, 4-23
  two-dimensional, 4-24
  undefined execution, 7-18
  unrolling, example, 4-26
  zero-overhead, 7-15
  zero-overhead and trace buffer, 21-15
  zero-overhead loops
    setup instruction, 7-13
loop setup (LSETUP) instruction, 7-13
loop top (LT0, LT1) registers, 1-13, 4-22, 4-24, C-2
lppcrel11m2, 7-14
L-registers (length), 5-8
Lreg. See length registers (L[3:0])
LRU (least recently used) bit, 6-42
LRU (least recently used) policy, 6-51
LRUPRIO bit, 6-23
Index

LRUPRIORST (LRU priority reset) bit, 6-5, 6-7
LSETUP (loop setup) instruction, 4-22, 7-13
LSHIFT...BY instruction, 14-14, 19-28
LSHIFT (logical shift) instruction, 2-48, 14-14
LT (loop top) register, 1-13, 4-22, 4-24

M
MAC (multiplier-accumulator), 2-35 to 2-48
See also multiply without accumulate
A0, A1, 2-46
capabilities, 1-3
combining MAC0 and MAC1
operations in vector instructions, 19-41
defined, 1-14
description summary, C-3
dual operations, 2-47
dual operations example, 2-47
multicycle 32-bit instruction, 2-46
major architectural change[7:0] field, 21-27
manual, conventions, xxxvi
mapping multiple interrupt sources, 4-55
maximum instruction, C-60
maximum instructions
maximum, 15-30, C-60
vector maximum, 19-32, C-115
MAX (vector maximum) instruction, 15-30, 19-32
media access control, See MAC

memory
See also cache; level 1 (L1) memory; level 1 (L1) data memory; level 1 (L1) instruction memory; level 2 (L2) memory
access limit, 1-5
address alignment, 5-16
architecture, 6-2 to 6-4
architecture of, 1-4
DMA controller, 1-5
external, 1-6
instruction storage, 6-65
internal, 1-5
L1 data, 6-24 to 6-38
level 2 (L2), 6-43
management, 6-45
nonaligned operations, 6-71
off-chip, 1-5, 1-6
page descriptor table, 6-50
pages, 6-48
protected, 3-5
protected regions, 6-54
protection and properties, 6-45 to 6-63
protection between tasks, 6-52
protection in user mode, 6-53
terminology, 6-74 to 6-76
transaction model, 6-65
memory management unit. See MMU
memory-mapped register. See MMR
memory pages, 6-48
minimum instructions, C-61
minimum, 15-32, C-61
vector minimum, C-115
minimum (MIN) instruction, 15-32
vector minimum, 19-35
MIN (vector minimum) instruction, 15-32, 19-35
mixed mode option (M), 15-43, 15-48, 15-55, 15-64, 15-70
mixed-multiply mode, 2-41
MMR (memory-mapped register), 6-72 to 6-74
accessibility, 6-73
ADSP-BF535, 6-73
internal memory space, 6-2
interrupt service routines, 4-3
I/O devices, 1-6
location of core events, 4-42
restriction on reading, 3-1
space configuration, 6-30
MMU (memory management unit), 6-45
application, 6-52
exception, 6-53
exception handler, 6-50, 6-51
on reset, 6-52
purpose, 1-4
MNOP (32-bit no op) instruction, parallel
instruction issues, 20-2
mode control pins, 3-13
modes
ADSP-BF535, A-1
emulation, 1-4, 3-1, 4-45
identifying, 3-2
operation, 1-4
supervisor, 1-4, 3-1
transitional conditions, 3-2
user, 1-4, 3-1
modified, definition, 6-75
modified addressing, 5-5
modified data cache line, 6-34
modify address, 5-2
modify – decrement instruction, 15-34, C-61
modify – increment instruction, 15-37, C-61
modify instructions
modify – decrement, 15-34, C-61
modify – increment, 15-37, C-61
modify registers (M[3:0]), 5-8
DAGs and circular buffering, 5-12
defined, 1-14
description, C-3
explained, 5-3
function in circular addressing, 1-21
load data register, 8-10
modify – decrement instructions, 15-34
modify – increment instructions, 15-37
store data register, 8-40
used to increment Ireg, 1-21
modulo addressing, 1-21
mostreg, syntax, 10-8
move byte – sign-extended instruction, 9-25, C-36
move byte – zero-extended instruction, 9-23, C-36
move conditional (move CC) instruction, 9-8, 11-12, C-31, C-41
move half to full word – sign-extended instruction, 9-10, C-31
move half to full word – zero-extended instruction, 9-13, C-31
move instructions, C-28
move register half instruction, 9-15, C-32
move register instruction, 9-2, C-28
M-registers (modify), 5-8
Mreg. See modify registers (M[3:0])
multi-cycle instructions, 4-9
multi-issue instruction, 4-9, 20-1, 20-6
multi-issue operations, AAU instructions, 5-24
multiple exceptions, for an instruction, 4-65
multiple interrupt sources
mapping, 4-55
sharing single core interrupt, 4-32
multiplier
accumulator result registers A[1: 0], 2-36, 2-37

Index
multiplier

\begin{itemize}
\item arithmetic integer modes formats, 2-16
\item data types, 2-14
\item fractional modes format, 2-16
\item instruction effects on flags, 2-38
\item instruction options, 2-40
\item operands for input, 2-36
\item operations, 2-36
\item results, 2-37
\item results saturation, 2-38
\item results storage, 2-42
\item rounding, 2-37
\item saturation, 2-38
\item status, 2-24
\item status bits, 2-38
\end{itemize}

multiplier accumulators. See MAC

multiplier results rounding, 2-19

* (multiply) operator, 19-3

multiply, vector instruction, 19-38, C-115

multiply 16-bit operands instruction,
\hspace{1cm} 15-43, C-62

multiply 32-bit operands instruction,
\hspace{1cm} 15-51, C-68

multiply-and-accumulate functions, 2-17

multiply and accumulate (MAC) unit. See MAC (multiplier-accumulator)

multiply and multiply-accumulate, vector
\hspace{1cm} instruction, 19-41, C-121

multiply and multiply-accumulate to
\hspace{1cm} accumulator instruction, 15-53, C-69

multiply and multiply-accumulate to data
\hspace{1cm} register instruction, 15-67, C-86

multiply and multiply-accumulate to
\hspace{1cm} half-register instruction, 15-58, C-74

multiply without accumulate, 2-44

fractional, unsigned operand example, 2-45

unsigned integer operand example, 2-45

N

\begin{itemize}
\item \(\neg\) (negate, two’s-complement) operator, 19-46
\item negate CC instruction, 11-15, C-42
\item negate instructions
\hspace{1cm} negate CC, 11-15, C-42
\hspace{1cm} vector negate, 19-46, C-138
\item negate (two’s-complement) instruction,
\hspace{1cm} 15-73, C-93
\item negative result (AN) bit, 2-25
\item nestable loops, registers, 7-15
\item nested interrupt
\hspace{1cm} explained, 4-51
\hspace{1cm} IPEND register, 4-40
\hspace{1cm} logging, 4-55
\item nested interrupt handling (figure), 4-53
\item nested ISR
\hspace{1cm} example Epilog code, 4-54
\hspace{1cm} example Prolog code, 4-53
\item nested loops, 4-25, 7-16
\item nesting of events, 1-6
\item NIM core event, 4-30
\item NMI event, 1-7
\item NMI (nonmaskable interrupt) bit, 1-7, 3-1, 4-40, 4-41, 4-46
\item nonaligned memory operations, 6-71
\item nonmaskable interrupt (NMI). See NMI
\item non-nested interrupt
\hspace{1cm} defined, 4-51
\hspace{1cm} interrupt handling (figure), 4-51
\item non-OS environments, 3-7
\item non-processing states, 3-2
\item nonsequential program
\hspace{1cm} operation, 4-9
\hspace{1cm} structures, 4-1
\item no op (MNOP) instruction, parallel
\hspace{1cm} instruction issues, 20-2
\item no op (NOP) instruction, 16-25, C-99
\item NOP (16-bit no op) instruction, 16-25
\item NOP instruction, C-99
\end{itemize}
Index

notation conventions
  choice of one register within a group, 1-11, C-5
  constants, 1-11, C-5
  fractions, 1-16
  loop PC-relative constants, 1-12, C-6
  multiple data registers in one instruction, 1-11
  PC-relative constants, 1-12, C-6
  range of sequential registers or bits, 1-11, C-5
  register pairs, 1-10, C-4
  register portions, 1-11, C-4
  set of registers in one instruction, C-4
  NOT (one’s-complement) instruction, 12-4, C-43
  NOT operator, 2-26
  numbers
    binary, 2-4
    data formats, 2-13
    fractional representation, 2-5
    signed, 2-5
    two’s-complement, 2-5
    unsigned, 2-4
  numeric formats, D-1 to D-8
    binary multiplication, D-5
    block floating-point, D-6
    integer mode, D-6
    two’s-complement, D-1

O
  off-chip memory, 1-6
  on-chip L2 memory, 6-4
  on-chip memory, 1-5
  one’s-population count (ONES) instruction, 13-26, C-45
  ONES instruction, 13-26
  opcodes. See instruction opcodes
  operating modes, 3-1 to 3-10

operators
  logical OR, 12-6
  logical OR assign, 11-12
  OR, 12-6
  += (vector add/add), 19-18
  +− (vector add/ subtract), 19-18
  −+ (vector subtract/add), 19-18
  −− (vector subtract/ subtract), 19-18
  + (add), 18-13, 19-18
  + add, 14-5, 15-6, 15-10, 15-13
  >>> (arithmetic right shift), 19-23
  >>> arithmetic right shift, 14-7
  >>>= arithmetic right shift assign, 14-7
  = (assign, representative sample, only), 8-3, 9-2, 10-2, 18-3, 19-3
  = assign (representative sample, only), 11-12, 12-10, 13-8, 14-2, 15-3
  =− (autodecrement), 10-2, 10-5
  ++ (autoincrement), 10-8, 10-12, 17-7, 17-9
  =~ bit invert (one’s-complement) assign, 11-15, 13-8
  == compare-equal, 11-2, 11-6, 11-9
  < less-than, 11-2, 11-6, 11-9
  <= less-than or equal, 11-2, 11-6, 11-9
  & logical AND, 12-2
  &= logical AND assign, 11-12
  <<= (logical left shift), 19-23, 19-28
  <<= (logical left shift), 14-2, 14-5, 14-7, 14-14
  <<= (logical left shift assign, 14-14
  >> (logical right shift), 19-28
  >> logical right shift, 14-14
  >>= logical right shift assign, 14-14
  ^ logical XOR, 12-8
  ^= logical XOR assign, 11-12
  =~ multi-bit invert (one’s-complement) assign, 12-4
operators  (continued)
* (multiply), 19-3
= (negate, two’s-complement), 19-46
= (negate (two’s-complement) assign, 15-73
– (subtract), 19-18
– subtract, 15-10, 15-13, 15-86

option flags  (continued)
no saturate (NS), negate (two’s-complement) instruction, 15-73
no saturate (NS), subtract instruction, 15-86
no saturate (NS) add instruction, 15-6
saturate accumulator at 32-bit word boundary (W32), 15-34, 15-37, 15-53
saturate and cross outputs (SCO), 19-18
saturate (S), 19-18
saturate (S), arithmetic shift instruction, 14-7
saturate (S), negate (two’s-complement) instruction, 15-73
saturate (S), subtract instruction, 15-86
saturate (S) add instruction, 15-6
saturate (S) instruction, 15-80
sign extended (X), 8-3, 8-19, 8-34, 9-13, 13-10, 13-16
sign-extended (X), 9-23
truncate, signed fraction operands (T), 15-43, 15-58
truncate, unsigned fraction operands (TFU), 15-43, 15-58
truncate (T), 9-16, 18-19
zero extended (Z), 8-3, 8-15, 8-31, 9-10, 13-16
zero-extended (Z), 9-23

ordering
loads and stores, 6-67
weak and strong, 6-67
OR operation, 12-6, C-43
OR operator, 2-26, 12-6, C-43
outer loops, 4-25

overflow
arithmetic flag, behavior, 1-17
arithmetic status flags, 1-15, C-7
behavior, 1-18
overflow (continued)
bits, 2-14
flags (AV0,AV1), 2-14
impossible in the multiply 32-bit
operands instruction, 15-51
prevention in divide primitive
instruction, 15-22
saturation of multiplier results, 2-38
user-implemented for multiply 32-bit
operands instruction, 15-51

P
PAB (peripheral access bus), 6-73
packing instructions
quad 8-bit unpack, C-106
vector pack, 19-48, C-138
PACK (vector pack) instruction, 19-48,
C-138
page descriptor table, 6-53
PAGE_SIZE[1:0] field, 6-55, 6-57
parallel instructions
issuing, 20-1
syntax, 20-2
parameter passing, 4-15
patch registers, B-8
pattern_reg bit field extraction instruction,
13-16
PC (program counter) register
non-memory mapped, 4-6
pcrel1m2, 7-6
pcrel13m2, 7-2
pcrel25m2, 7-3, 7-8
pcrel5m2, 7-14
PC-relative
constant, 1-12, C-6
offset, 4-11, 4-12
pcrelm2, 7-2
PEMUSWx bits, 21-20, 21-21
pending event requests, coordinating, 4-38
performance monitor control (PFCTL)
register, 21-19, 21-20
performance monitor counter (PFCNTRn)
registers, 21-19, 21-20
performance monitor registers, B-9
performance monitor unit, 21-19
peripheral
interrupt generated by, 4-31
interrupts, 4-30
interrupts, relative priority, 4-37
interrupts, source masking, 4-36
interrupt sources, 4-35
supporting interrupts, 4-33
waking up core, 4-34
peripheral access bus (PAB), 6-73
PFCEN0[1:0] field, 21-21
PFCEN1[1:0] field, 21-21
PFCNT0 bit, 21-21
PFCNT1 bit, 21-21
PFCNTRn (performance monitor counter)
register, 21-20
PFCNTRn (performance monitor counter)
registers, 21-19
PFCTL (performance monitor control)
register, 21-19, 21-20
PFMON0[7:0] field, 21-21
PFMON1[7:0] field, 21-21
PFPWR bit, 21-21
pipeline
figure, 4-8
instruction, 4-3, 4-7
instructions when interrupt occurs, 4-67
interlocked, 6-66
stages, 4-7
stalls, 6-66
pointer register file, 2-6
pointer register modification, 5-15
pointer registers (P[5:0])
defined, 5-6
description, 1-13
Index

pointer registers (P[5:0]) (continued)
description of, C-2
element, 5-3
indirect/post-increment index
addressing, 8-47
load half-word – sign-extended instruction, 8-20
source pointer register, 8-11
user mode, 3-4
pop instruction, 10-8, C-37
pop multiple instruction, 10-12, C-37
popping, manual, 4-5
PORT_PREF0 (DAG0 port preference)
bit, 6-25, 6-26
PORT_PREF1 (DAG1 port preference)
bit, 6-24, 6-25
post-modify addressing
AAU architecture, 5-2
circular buffering, 5-13
defined, 5-5
element, 5-11
post-modify buffer access, 5-14
powerdown warning, as NMI, 4-46
powerup, 3-12
PRCENx bits, 21-20
PREFETCH (data cache prefetch)
instruction, 6-37, 17-3, C-101
Preg. See pointer registers (P[5:0])
pre-modify instruction example, 5-11
pre-modify stack pointer addressing, 5-11
prioritization of events, 1-6
priority watermark (PRIO_MARK[0:3])
field, 6-36
processor, single-core bus architecture, 6-2
processor core architecture, diagram, 1-2, 2-2
processor mode
determination, 3-1
emulation, 3-9
processor mode (continued)
figure, 3-2
identifying, 3-2
IPEND interrogation, 3-1
supervisor, 3-7
user, 3-3
processor state
idle, 3-9
reset, 3-10
upon reset, 3-11
product identification (DSPID) register.
See DSPID (product identification) register
program counter (PC) register
non-memory mapped, 4-6
PC-relative indirect JUMP and CALL, 4-13
PC-relative offset, 4-11
program flow
control instructions, C-13
controlling, 4-19
described, 4-1
instructions, 7-2
sequencer tasks for, 1-3
program sequencer, 4-1 to 4-58
history, 21-15	tasks performed, 1-3
program structures, nonsequential, 4-1
protected instructions, 3-4
memory, 3-5
memory regions, 6-54
resources, 3-4
protected instructions, 3-4
protection violation exceptions, 17-3, 17-5, 17-7, 17-9
pushing, manual, 4-5
push instruction, 10-2, C-37
push multiple instruction, 10-5, C-37
Index

Q
quad 16-bit operations, defined with example, 2-28
quad 8-bit add (BYTEOP16P) instruction, 18-15
quad 8-bit add instruction, C-103
quad 8-bit average - byte (BYTEOP1P) instruction, 18-19, 18-23, C-104
quad 8-bit average - half-word (BYTEOP2P) instruction, 18-24
quad 8-bit average – half-word instruction, C-104
quad 8-bit pack (BYTEPACK) instruction, 18-30, 18-32, C-105
quad 8-bit subtract/absolute-accumulate (SAA) instruction, 18-36, C-106
quad 8-bit subtract/absolute value/accumulate (SAA) operations, 1-3
quad 8-bit unpack (BYTEUNPACK) instruction, 18-41, C-106
quotient (AQ) bit, 2-25

R
RAB (register access bus), 6-72
radix point, D-1, D-2
RAISE1 instruction, 3-14
RAISE (force interrupt / reset) instruction, 3-11, 4-47, 16-17
range
CALL instruction, 4-12
conditional branches, 4-20
JUMP instruction, 4-11
of signed numbers, D-4
reading MMRs, restriction, 3-1
read transfer, address, 6-14
read / write access bit, 6-40
real-time clock (RTC), processor idle state, 3-10
reg, list of registers, 8-4
register access bus (RAB), 6-72
register file, 2-6 to 2-11
compute, 1-2
instructions, conditional branch, 4-10
instructions, list of, 2-9
reading, 4-7
reads, 4-9
stalls, 4-9
writes, 4-9
register move, conditional, 4-20
register pairs, valid pairs defined, C-4
register portions, notation convention, C-4
registers
accessible in user mode, 3-4
choice of one register within a group, notation convention, 1-11, C-5
core, B-1 to B-9
data test, 6-38 to 6-42
memory-mapped, core, B-1 to B-9
product identification, 21-27
range of sequential, notation convention, 1-11, C-5
register set notation, multiple data registers in one instruction, C-4
register shift
defined, 2-50, 2-51
example, 2-50, 2-51
related publications, xxxi
replacement policy
CPLBs, 6-51
definition, 6-75
for cache controller, 6-33
reserved core event, 4-30
reset
core double-fault, 3-13
core event, 4-30
core-only software, 3-13, 3-16
Index

reset
  effect on memory configuration, 6-26
  event, 1-7
  hardware, 3-12, 3-13
  initialization sequence, for interrupts, 4-37
  interrupt (RST), 4-46
  MMU, 6-52
  processor state upon reset, 3-11
  state, 3-2, 3-10
  system and core, 3-16
  system software, 3-12, 3-14
  watchdog timer, 3-12, 3-14
  RESET pin, 4-46
  RESET signal
    de-asserting, 3-7
    reset state, 3-10
  reset values, ADSP-BF535, A-1
  resources, protected, 3-4
  restoring loops, 4-27
  resuming loops, 4-27
  RETI (return from interrupt) register, 10-3
  RETS (return from subroutine) register, 4-12, 7-9, 10-3, 10-17
  return address
    for CALL instruction, 4-10
    registers, 4-6
    storage, 4-3
  return from emulation (RTE) instruction.
    See RTE
  return from exception (RTX) instruction.
    See RTX
  return from interrupt (RETI) register. See RETI
  return from interrupt (RTI) instruction. See RTI
  return from nonmaskable interrupt (RTN) instruction. See RTN
  return from subroutine (RETS) register. See RETS
  return from subroutine (RTS) instruction. See RTS
  return instruction
    defined, 4-10
    invoking user mode, 3-5
    opcodes, C-14
    supervisor mode, 3-7
    syntax, 7-10
  RETx register, 3-5
  RND12 instruction, 15-13
  RND20 instruction, 15-10
  RND_MOD (rounding) bit
    affected instructions, 9-18, 15-38, 15-45, 15-59
    ASTAT register, 1-20
    biased rounding, 2-23
    rounding defined, 2-19
    rounding mode, 2-25
  RND option (multiplier results rounding), 2-19
  RND (round to half-word) instruction, 15-77, C-94
  ROM
    external memory device, 1-6
    internal memory space, 6-2
  rotate operations instructions, C-46
  rotate (ROT) instruction, 14-21, C-54
  ROT...BY instruction, 14-21
  ROT (rotate) instruction, 14-21, C-54
  rounding
    behavior, 1-20
    biased, 1-19, 2-19, 2-22
    convergent, 1-19, 2-20
    defined, 2-19
    example, 2-19
    instructions, 2-19
Index

rounding
  instructions for different bit boundaries, 2-24
  round-to-nearest defined, 1-19
  setting mode, 2-2
  unbiased, 1-19, 2-19
rounding mode (RND_MOD) bit, 2-25
round to half-word (RND) instruction. See RND
round-to-nearest rounding, 1-19, 2-22
RST (reset interrupt) bit, 4-40, 4-41, 4-46
RTC (real-time clock), 3-12
RTE (return from emulation) instruction, 4-10, 7-10
RTI (return from interrupt) instruction, 4-10, 4-45, 7-10
RTN (return from NMI) instruction, 4-10, 7-10
RTS (return from subroutine) instruction, 4-10, 7-10
RTX (return from exception) instruction, 4-10, 7-10
RW (read / write access) bit, 6-21

s
SAA. See quad 8-bit subtract-absolute value-accumulate operations
saturate accumulator at 32-bit word boundary (W32)
multiply and multiply-accumulate to accumulator instruction, 15-53
use with modify instructions, 15-34, 15-37
use with multiply instructions, 15-53
saturate instruction, 15-80, C-95
saturation
  16-bit register range, 1-18
  32-bit register range, 1-18
  40-bit register range, 1-18

saturation
  accumulator, 1-12
  defined, 1-17
saving loops, example, 4-27
SBNK[1:0] (subbank access) field, 6-21
scalar operations, 19-38, 19-41
scene_reg bit field extraction instruction, 13-16
scratchpad data memory (SRAM), 6-4, 6-47
SDRAM (synchronous dynamic random access memory), 1-5, 1-6
SEARCH (vector search) instruction, 19-50, C-138
self-nesting interrupt enable (SNEN) bit, 21-26
SEQSTAT (sequencer status) register, 4-6, 4-59, 16-3, 16-8
sequencer registers, 3-4
sequencer stalls, 4-9
sequencer status (SEQSTAT) register. See SEQSTAT
sequential registers or bits, range of, notation convention, C-5
servicing interrupt, 4-48
set, definition, 6-75
set associative, 6-76
SET index field, 6-21
set index field, 6-21, 6-40
SFTRESET (software reset) bit, 4-59
shared interrupt, 4-37, 4-55
shifter, 2-1, 2-48 to 2-56
  arithmetic formats, 2-17
data types, 2-15
diagram, 1-2
field deposit, 2-52
field extract, 2-52
immediate shifts, defined, 2-49, 2-50
immediate shifts, example, 2-49, 2-51
instruction effects on status flags, 2-53
Index

Shifter
- Instructions listed, 2-53
- Operations, 2-48
- Register shifts, defined, 2-50
- Register shifts, example, 2-50
- Status flags, 2-53
- Tasks performed, 1-3
- Three-operand shifts, 2-50
- Two-operand shifts, defined, 2-49

Shift instructions
- Arithmetic shift, 2-48, 14-7, C-46
  ASHIFT, 2-48
- Logical shift, 2-48, 14-14, C-50
  LSHIFT, 2-48
- ROT, 2-48
- Rotate, 2-48
- Shift with add, 14-5, C-46
- Vector arithmetic shift, 19-23, C-114
- Vector logical shift, 19-28, C-115

Shift / rotate operations instructions, C-46

Short jump (JUMP.S) instruction, 4-11

SIC_IARx (system interrupt assignment) register, 4-31

SIC_IARx (system interrupt assignment) register 0, 4-37

SIC_IARx (system interrupt assignment) registers

SIC_IMASK (system interrupt controller interrupt mask) register, 3-13, 4-31

SIC_ISR (system interrupt status) register, 4-31

SIC_IWR (system interrupt wakeup enable) register, 4-31

SIC (system interrupt controller), 1-7, 1-8, 4-35, 4-55

SIGN (add on) instruction, 19-3

Sign bit (SIGNBITS) instruction. See
  SIGNBITS

SIGNBITS (sign bit) instruction, 15-83, C-95

Signed fraction operands (T)
- Multiply 16-bit operands instruction, 15-43
- Multiply and multiply-accumulate to half-register instruction, 15-58
- Use with multiply instructions, 15-43, 15-58

Signed numbers
- Data format, 2-4
- Ranges, D-4
- Supported by Blackfin, D-1
- Two’s-complement, 2-5

Signed operator (IS)
- Multiply 16-bit operands instruction, 15-43
- Multiply and multiply-accumulate to accumulator instruction, 15-53
- Multiply and multiply-accumulate to data register instruction, 15-67
- Multiply and multiply-accumulate to half-register instruction, 15-58

Sign extended (X) flag, use with bit field instructions, 13-10, 13-16

Sign extending data, 2-12

SIGN instruction, C-107

SIMD video ALU operations, 2-35

Single 16-bit operations, defined with example, 2-27

Single 32-bit operations, defined with example, 2-29

Single step exception, 4-66

Small loop count values, 7-16

SNEN (self-nesting interrupt enable) bit, 21-26

Software interrupt handlers, 4-30

Software loops, zero-overhead, 7-15
software reset method
  RAISE1 instruction, 3-14
system software reset field, 3-14
watchdog timer, 3-14
software reset (SWRST) register, 3-15
speculative load execution, 6-69
SP (kernel stack pointer), 3-7
SPORT, processor idle state, 3-10
SP (stack pointer) register, 1-13, 4-6
SP (supervisor stack pointer) register, 5-6
SRAM
  address collision, 6-27
L1 data, 6-27
L1 instruction access, 6-8
L2, 6-43
  memory architecture, 6-2
scratchpad, 6-4
SRAM ADDR[13:12] field, 6-40
SRAM (static random access memory), 1-5, 1-6
SSSTEP (supervisor single step) bit, 21-26
SSYNC (system synchronize) instruction, 16-8
stack
  effect of linkage instruction, 10-18
  effect of pop instruction, 10-9
  effect of pop multiple instruction, 10-13
  effect of push instruction, 10-2
  effect of push multiple instruction, 10-6
  manipulation, 1-3
  maximum frame size, 10-18
  variables, 4-15
stack control instructions, C-37
stack pointer, 1-13, 10-7 to 10-19, C-2
stack pointer (SP) register, 4-6, 5-6
stalled load instruction, 6-34
stalls
  computation, 4-9
  DAG, 4-9
  data memory, 4-9
  pipeline, 6-66
  register file, 4-9
  sequencer, 4-9
STATDA0 bit, 21-14
STATDA1 bit, 21-14
states
  idle, 3-2
  reset, 3-2
STATIA0 bit, 21-14
STATIA1 bit, 21-14
STATIA2 bit, 21-14
STATIA3 bit, 21-14
STATIA4 bit, 21-14
STATIA5 bit, 21-14
static random access memory. See SRAM
status flags, arithmetic, summarized, 1-15
status registers, 3-4
sticky A0 overflow (AV0S) bit, 2-25
sticky A1 overflow (AV1S) bit, 2-25
sticky dreg overflow (VS) bit, 2-25
sticky overflow arithmetic status flag, 1-15
STI (enable interrupts) instruction, 6-74, 16-15
store
  operation, 6-66
  ordering, 6-67
store byte instruction, 8-54, C-27
store data register instruction, C-24
store high data register half instruction, 8-45, C-25
store instructions, 8-1
  store byte, 8-54, C-27
  store data register, 8-40, C-24
  store high data register half, 8-45, C-25
  store low data register half, 8-49, C-26
  store pointer register, 8-37, C-23
table of, C-16
store low data register half instruction, 8-49, C-26
Index

store pointer register instruction, 8-37,
   C-23
subbank access (SBNK[1:0]) field, 6-21,
   6-40
subroutines
   defined, 4-13
   program flow, 4-1
   return from (RTS) instruction, 7-10,
   7-11
   – (subtract) operator, 19-18
   –|+ (vector subtract/add) operator, 19-18
subtract immediate instruction, 15-90,
   C-98
subtract instructions
   quad 8-bit subtract, C-106
   quad 8-bit subtract-absolute-accumulate, C-106
   subtract, 15-86, C-96
   subtract immediate, 15-90, C-98
   vector add / subtract, 15-86, C-107
   –|– (vector subtract/subtract) operator, 19-18
superscalar architecture, 20-1
supervisor mode, 1-4, 3-1, 3-7
   disable interrupts instructions, 16-13
   enable interrupts instructions, 16-15
   exclusive supervisor instructions, 7-11
   exclusive supervisor registers, 9-6, 10-3,
   10-10
   force interrupt / reset instructions, 16-19
   idle instructions, 16-4
   MMR access, 6-73
   return instructions, 7-11
   supervisor single step (SSSTEP) bit, 21-26
   supervisor stack, preventing exceptions, 4-56
   supervisor stack pointer (SP) register, 5-6,
   5-7
   supply addressing, 5-2
   support, technical or customer, xxviii
   SWRST (software reset register), 3-15
   synchronize, core instruction, 16-5
   synchronous dynamic random access
   memory. See SDRAM
syntax
   allreg, 10-2
   case insensitivity, 1-8
   comment delineator rules, 1-10
   constant notation convention, 1-12, C-5,
   C-6
dagreg, 9-3
Dreg_even, 9-3, 15-67
Dreg_hi, 8-45, 9-16, 15-43, 15-58, 19-3
Dreg_lo, 8-27, 8-49, 9-10, 9-13, 9-16,
   12-10, 13-16, 13-26, 14-8, 14-15,
   15-26, 15-43, 15-58, 15-83, 19-3,
   19-8, 19-23, 19-28
Dreg_lo_hi, 14-8, 14-15, 15-6, 15-10,
   15-13, 15-26, 15-38, 15-43, 15-53,
   15-58, 15-67, 15-77, 15-83, 15-86,
   19-48
Dreg_odd, 9-3, 15-67
   free format rules, 1-9
   genreg, 9-3
   imm3 constant, 11-2, 11-6
   imm6 constant, 14-21
   immediate values, notation convention,
   1-11
   instruction delimiting rules, 1-9
   mostreg, 10-8
   parallel instructions, 20-2
   reg, 8-4
   sysreg, 9-3
   uimm3 constant, 11-2, 11-6
   uimm4 constant, 14-8, 14-15, 16-17,
   16-20, 19-23, 19-28
   uimm5 constant, 13-2, 13-4, 13-6, 14-8,
   14-15
   SYSCFG (system configuration) register,
   21-26
Index

SYSCR (system reset configuration) register, 3-14
sysreg, syntax, 9-3
system and core event mapping (table), 4-30
system and core MMRs, 3-4
system and core reset, 3-16
system configuration (SYSCFG) register, 21-26
system events controlling, 1-7
prioritizing, 1-7
system interrupt assignment (SIC_IARx) register 0. See SIC_IARx
system interrupt assignment (SIC_IARx) registers. See SIC_IARx
system interrupt controller (SIC). See SIC
system interrupt mask (SIC_IMASK) register. See SIC_IMASK
system interrupt processing, 4-31
system interrupts, 4-30
system interrupt status (SIC_ISR) register. See SIC_ISR
system interrupt wakeup enable (SIC_IWR) register. See SIC_IWR
system registers, protecting from unintended access, 1-4
system reset configuration (SYSCR) register. See SYSCR
system software reset, 3-12, 3-14
system stack, recommendation for allocating, 4-56
system synchronize instruction, C-99
system synchronize (SSYNC) instruction, 16-8

T
tag, definition, 6-76
tag[1:0] field, 6-23
ntag[19:4] field, 6-23, 6-42
tag[3:2] field, 6-23, 6-42
tag bit, 6-42
TAGSELB (array access) bit, 6-21
TBUFF[15:0] field, 21-18
TBUFF[31:16] field, 21-18
TBUF[4:0] field, 21-17
TBUFCTL (trace buffer control) register, 21-16
TBUFEN bit, 21-17
TBUFOVF bit, 21-16, 21-17
TBUFPSR bit, 21-16, 21-17
TBUFSTAT (trace buffer status) register, 21-17
TBUF (trace buffer) register, 21-16, 21-18
technical support, xxviii
test and set byte (atomic) instruction, 16-22, C-99
TESTSET (test and set byte) instruction atomic operations, 6-72
syntax, 16-22
three-operand shift, 2-50
throughput achieved by SRAM, 6-2
interlocked pipeline, 6-66
trace buffer configuring, 21-15
exception, 4-66
reading, 21-16
zero-overhead loops, 21-15
trace buffer control (TBUFCTL) register, 21-16
trace buffer status (TBUFSTAT) register, 21-17
trace buffer (TBUF) register, 21-16, 21-18
trace unit, 21-15 to 21-18
emulation mode, 21-18
trace unit registers, B-8
truncation behavior, 1-20
defined, 1-20, 2-23
Index

two's complement format, D-1

two's-complement (vector negate) instruction, 19-46
two-dimensional loops, 4-24
two-operand shift, defined, 2-49

U
ujimm15 constant, 8-31, 8-34, 8-54
ujimm16 constant, 8-4
ujimm16m2 constant, 8-15, 8-19, 8-50
ujimm17m4 constant, 8-7, 8-11, 8-37, 8-41
ujimm18m4 constant, 10-17
ujimm3 constant, 11-2, 11-6
ujimm4 constant, 14-8, 14-15, 16-17, 16-20, 19-23, 19-28
ujimm5 constant, 13-2, 13-4, 13-6, 13-8, 14-8, 14-15
ujimm5m2 constant, 8-15, 8-19, 8-50
ujimm6m4 constant, 8-7, 8-11, 8-37, 8-41
ujimm7m4 constant, 8-7, 8-11, 8-37, 8-41
unbiased rounding, 1-19, 2-19

unconditional branches
branch latency, 4-21
branch target address, 4-21
undefined instruction, 4-65
unknown jump (JUMP.0) instruction, 4-11

UNLINK instruction
code sequence, 4-17
syntax, 10-17
unrecoverable event, 4-65

unrolling loops, example, 4-26
unsigned fraction operands (TFU)
multiply 16-bit operands instruction, 15-43
multiply and multiply-accumulate to half-register instruction, 15-58
use with multiply instructions, 15-43, 15-58

unsigned integer, D-1

unsigned numbers
data formats, 2-13
defined, 2-4

unsigned operator (FU)
multiply 16-bit operands instruction, 15-43
multiply and multiply-accumulate to accumulator instruction, 15-53
multiply and multiply-accumulate to data register instruction, 15-67
multiply and multiply-accumulate to half-register instruction, 15-58

unsigned operator (IU)
multiply 16-bit operands instruction, 15-43
multiply and multiply-accumulate to half-register instruction, 15-58
use with multiply instructions, 15-43, 15-58

upper bits of address for match[21:6] field, 6-59, 6-60
upper bits of address for match[5:0] field, 6-59, 6-60

user_label, 7-3, 7-6, 7-8

user mode
accessible registers, 3-3
accessing MMRs, 6-73
access restriction, 1-4
defined, 3-1
entering, 3-5
leaving, 3-6
memory protection, 6-53
protected instructions, 3-4

user stack pointer (USP) register. See USP

USP (user stack pointer), 3-7, 5-6, 5-7
Index

V
valid, definition, 6-76
VALID bit, 6-42
  cache line replacement, 6-15
clearing, 6-37
figure, 6-23
function, 6-11
  instruction cache invalidation, 6-18
V bit, 2-25, 2-38
V_COPY bit, 2-25
vector absolute value (ABS) instruction, 19-15, C-107
vector add / subtract instruction, 19-18, C-107
vector arithmetic shift instruction, 19-23, C-114
vector couplet, 19-38, 19-41
vector instructions
  vector absolute value (ABS) instruction, 19-15, C-107
  vector add / subtract, 19-18, C-107
  vector arithmetic shift, 19-23, C-114
  vector maximum, 19-32, C-115
  vector minimum, 19-35, C-115
  vector multiply, 19-38, C-115
  vector multiply and multiply-accumulate, 19-41, C-121
  vector pack, 19-48, C-138
  vector search, 19-50, C-138
  vector interrupt, 16-17
  vector logical shift instruction, 19-28, C-115
  vector maximum (MAX) instruction, 19-32, C-115
  vector minimum (MIN) instruction, 19-35, C-115
  vector multiply and multiply-accumulate instruction, 19-41, C-121
  vector multiply instruction, 19-38, C-115
  vector negate (two's-complement) instruction, 19-46, C-138
  vector operations instructions, C-107
  vector pack (PACK) instruction, 19-48, C-138
  vector search (SEARCH) instruction, 19-50, C-138
  victim, definition, 6-76
  victim buffers, 6-34
  video ALU (arithmetic logic unit) operations, 2-35
  video ALUs, 2-1
  video bit field operations, 13-10, 13-16
  video information, processing, 2-35
  video pixel operations instructions, C-102
  video pixels, instructions, 18-1
  VIT_MAX (compare-select) instruction, 19-8, C-107
  Von-Neumann architecture, 6-1
  V (overflow for data register results copy), 1-15
  VS (sticky overflow for data register results), 1-15, 2-25, 2-38

W
WAKEUP signal, 3-10
watchdog timer
  reset source and result, 3-12
  software reset method, 3-14
watchpoint match, 4-65
watchpoint registers, B-8
watchpoints
data, 21-3
  instruction-address-range, 21-2
watchpoint status (WPSTAT) register, 21-14
watchpoint unit, 21-1 to 21-14
code patching, 21-5
data address watchpoints, 21-10
Index

watchpoint unit (continued)
even triggering, 21-3
instruction watchpoints, 21-4
memory-mapped registers, 21-2
WPIACTL watchpoint ranges, 21-4
ways
1-way associative (direct-mapped), 6-74
definition, 6-76
locking, 6-17
priority in cache line replacement, 6-15
WAYSEL[1:0] (access way) field, 6-21
WB (write back) instruction pipeline stage, 4-9
width, instruction, 4-8
word, defined, 2-6
word fetching order, 6-14
WPAND bit, 21-3, 21-8
WPDACC0[1:0] field, 21-13
WPDACC1[1:0] field, 21-13
WPDACNTn (registers data watchpoint address count value) registers, 21-11, 21-12
WPDACCTL (data watchpoint address control) register, 21-12
WPDA (data address)[15:0] field, 21-11
WPDA (data address)[31:16] field, 21-11
WPDAEN0 bit, 21-13
WPDAEN1 bit, 21-13
WPDA (data watchpoint address) registers, 21-10
WPDCNTEN0 bit, 21-13
WPDCNTEN1 bit, 21-13
WPDREN01 bit, 21-13
WPDRINV01 bit, 21-13
WPDRSRO[1:0] field, 21-13
WPDSRC1[1:0] field, 21-13
WPIACNTn (instruction watchpoint address count) registers, 21-5, 21-6, 21-7
WPIACTL (instruction watchpoint address control) register, 21-7
WPIAEN0 bit, 21-9
WPIAEN1 bit, 21-9
WPIAEN2 bit, 21-9
WPIAEN3 bit, 21-9
WPIAEN4 bit, 21-8
WPIAEN5 bit, 21-8
WPIAn (instruction watchpoint address) registers, 21-5, 21-6
WPICNTEN0 bit, 21-9
WPICNTEN1 bit, 21-9
WPICNTEN2 bit, 21-9
WPICNTEN3 bit, 21-9
WPICNTEN4 bit, 21-8
WPICNTEN5 bit, 21-8
WPIREN01 bit, 21-9
WPIREN23 bit, 21-9
WPIREN45 bit, 21-8
WPIRINV01 bit, 21-9
WPIRINV23 bit, 21-9
WPIRINV45 bit, 21-8
WPPWR bit, 21-4, 21-7, 21-9
WPSTAT (watchpoint status) register, 21-14
wraparound buffer, 5-15write-back (WB)
cache mode, 6-29
cache write methods, 6-35
definition, 6-76
instruction pipeline stage, 4-7
write buffer depth, 6-35write-through
cache mode, 6-29
cache write methods, 6-35
definition, 6-76
Index

X
XOR (exclusive-OR) instruction, 2-26, 12-8

Z
zero extended (Z) flag, use with instructions, 13-16
zero extending data, 2-12
zero-overhead, loop registers, 4-6
zero-overhead loops
  registers, 4-22
  setting up, 7-13
  setup instruction, C-14
  trace buffer, 21-15
zero result (AZ) bit, 2-25