ADSP-BF53x/BF56x Blackfin® Processor Programming Reference

Revision 1.2, February 2007

Part Number
82-000556-01

Analog Devices, Inc.
One Technology Way
Norwood, Mass. 02062-9106

ANALOG DEVICES

Copyright Information

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

INTRODUCTION

COMPUTATIONAL UNITS

OPERATING MODES AND STATES

PROGRAM SEQUENCER

ADDRESS ARITHMETIC UNIT

MEMORY

PROGRAM FLOW CONTROL

LOAD / STORE

MOVE

STACK CONTROL

CONTROL CODE BIT MANAGEMENT

LOGICAL OPERATIONS

BIT OPERATIONS

SHIFT/ROTATE OPERATIONS

ARITHMETIC OPERATIONS

EXTERNAL EVENT MANAGEMENT

CACHE CONTROL

VIDEO PIXEL OPERATIONS

VECTOR OPERATIONS

ISSUING PARALLEL INSTRUCTIONS

DEBUG

ADSP-BF535 CONSIDERATIONS

CORE MMR ASSIGNMENTS

INSTRUCTION OPCODES

NUMERIC FORMATS

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.

- Chapter 7–Chapter 19, "Program Flow Control", "Load / Store", "Move", "Stack Control", "Control Code Bit Management", "Logical Operations", "Bit Operations", "Shift/Rotate Operations", "Arithmetic Operations", "External Event Management", "Cache Control", "Video Pixel Operations", and "Vector Operations"
Provide descriptions of assembly language instructions and describe their execution.

- 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).

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.

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*

- *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*
- *VisualDSP++ Kernel (VDK) User's Guide*

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.

File	Description
.CHM	Help system files and manuals in Help format
.HTM or .HTML	Dinkum Abridged C++ library and FlexLM network license manager software documentation. Viewing and printing the .HTML files requires a browser, such as Internet Explorer 4.0 (or higher).
.PDF	VisualDSP++ and processor manuals in Portable Documentation Format (PDF). Viewing and printing the .PDF files requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher).

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 (toolbar 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.

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.

Example	Description
Close command (**File** menu)	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.
`this|that`	Alternative items in syntax descriptions are delimited with a vertical bar; read the example as `this` or `that`. One or the other is required.
`{this | that}`	Optional items in syntax descriptions appear within curly braces; read the example as an optional `this` or `that`.
`[{({S|SU})}]`	Optional items for some lists may appear within parenthesis. If an option is chosen, the parenthesis must be used (for example, `(S)`). If no option is chosen, omit the parenthsis.
`.SECTION`	Commands, directives, keywords, and feature names are in text with `letter gothic` font.
`filename`	Non-keyword placeholders appear in text with italic style format.
SWRST Software Reset register	Register names appear in UPPERCASE and a special typeface. The descriptive names of registers are in mixed case and regular typeface.
TMR0E, $\overline{\text{RESET}}$	Pin names appear in UPPERCASE and a special typeface. Active low signals appear with an $\overline{\text{OVERBAR}}$.
DRx, SIC_IMASKx, I[3:0] $\overline{\text{SMS[3:0]}}$	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; $\overline{\text{SMS[3:0]}}$ indicates $\overline{\text{SMS3}}$, $\overline{\text{SMS2}}$, $\overline{\text{SMS1}}$, and $\overline{\text{SMS0}}$).

Example	Description
`0xFBCD CBA9`	Hexadecimal numbers use the 0x prefix and are typically shown with a space between the upper four and lower four digits.
`b#1010 0101`	Binary numbers use the b# prefix and are typically shown with a space between each four digit group.
ⓘ	**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.

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.

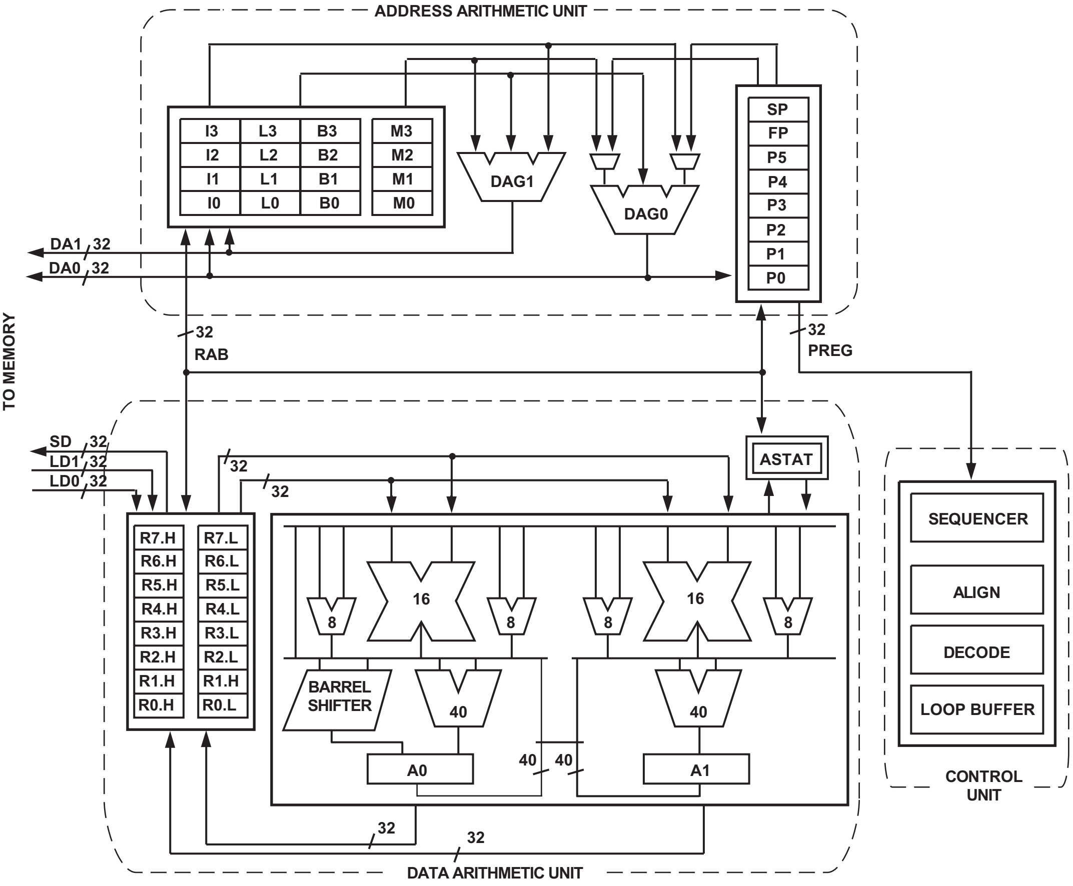


Figure 1-1. Processor Core Architecture

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.

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.

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.

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`.

- 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, 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.

- 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 an 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

Register	Description
Accumulators	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.
Data Registers	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.
Pointer Registers	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.
Stack Pointer	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.
Frame Pointer	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.
Loop Top	LT0 and LT1; contains 32-bit address of the top of a zero overhead loop.

Table 1-1. Registers (Cont'd)

Register	Description
Loop Count	LC0 and LC1; contains 32-bit counter of the zero overhead loop executions.
Loop Bottom	LB0 and LB1; contains 32-bit address of the bottom of a zero overhead loop.
Index Register	The set of 32-bit registers I0, I1, I2, I3 that normally contain byte addresses of data structures. Abbreviated I-register or Ireg.
Modify Registers	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.
Length Registers	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.
Base Registers	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.

Functional Units

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

Table 1-2. Processor Sections

Processor	Description
Data Address Generator (DAG)	Calculates the effective address for indirect and indexed memory accesses. Consists of two sections–DAG0 and DAG1.
Multiply and Accumulate Unit (MAC)	Performs the arithmetic functions on data. Consists of two sections (MAC0 and MAC1)–each associated with an Accumulator (A0 and A1, respectively).
Arithmetic Logical Unit (ALU)	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.

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

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

ⓘ 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 binal point.

In DSP instructions that assume placement of a binal point, for example in computing sign bits for normalization or for alignment purposes, the binal 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

Registers Size	Format	Notation	Sign Bit	Extension Bits	Fractional Bits
40-bit registers	Signed Fractional	9.31	1	8	31
	Unsigned Fractional	8.32	0	8	32
32-bit registers	Signed Fractional	1.31	1	0	31
	Unsigned Fractional	0.32	0	0	32
16-bit registers	Signed Fractional	1.15	1	0	15
	Unsigned Fractional	0.16	0	0	16

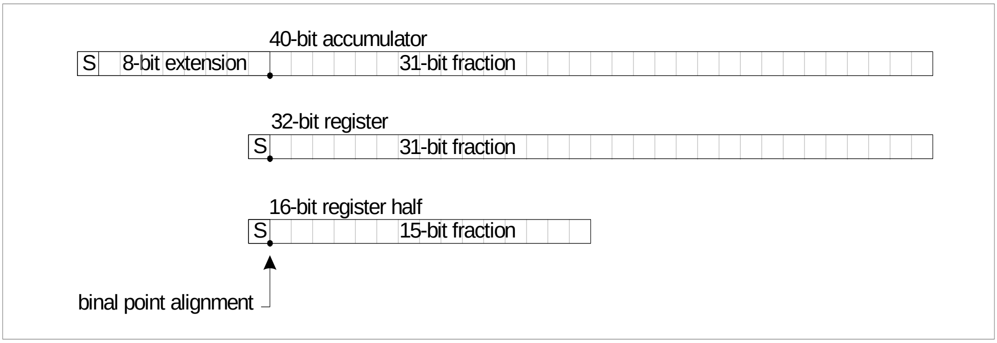


Figure 1-2. Conventional Placement of Binal Point

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 0x7FFF FFFF. 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 0x7F FFFF 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.

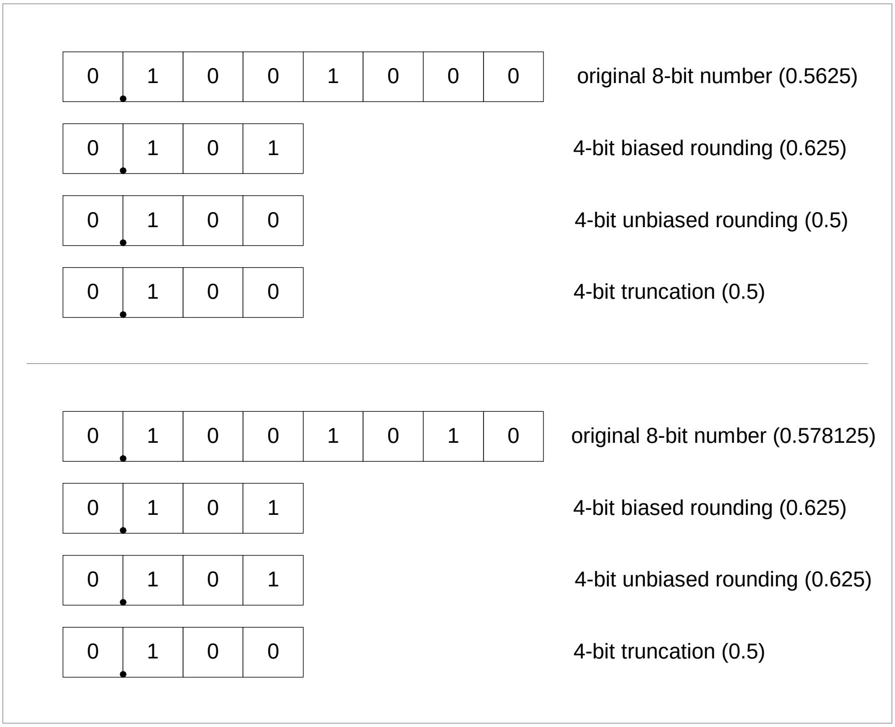

0 1 0 0 1 0 0 0
original 8-bit number (0.5625)
0 1 0 1
4-bit biased rounding (0.625)
0 1 0 0
4-bit unbiased rounding (0.5)
0 1 0 0
4-bit truncation (0.5)
0 1 0 0 1 0 1 0
original 8-bit number (0.578125)
0 1 0 1
4-bit biased rounding (0.625)
0 1 0 1
4-bit unbiased rounding (0.625)
0 1 0 0
4-bit truncation (0.5)

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 (*Ireg*) through a predefined address range, then automatically resets the *Ireg* 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 *Ireg*, *Breg*, and *Lreg* (i.e., I0/B0/L0, I1/B1/L1, I2/B2/L2, and I3/B3/L3)

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

The circular buffer registers define the length (*Lreg*) of the data block in bytes and the base (*Breg*) address to reinitialize the *Ireg*.

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 *Ireg* by clearing the *Lreg* that corresponds to the *Ireg* used in the instruction. For example, clear L2 to disable circular addressing for register I2. Any nonzero value in an *Lreg* 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.

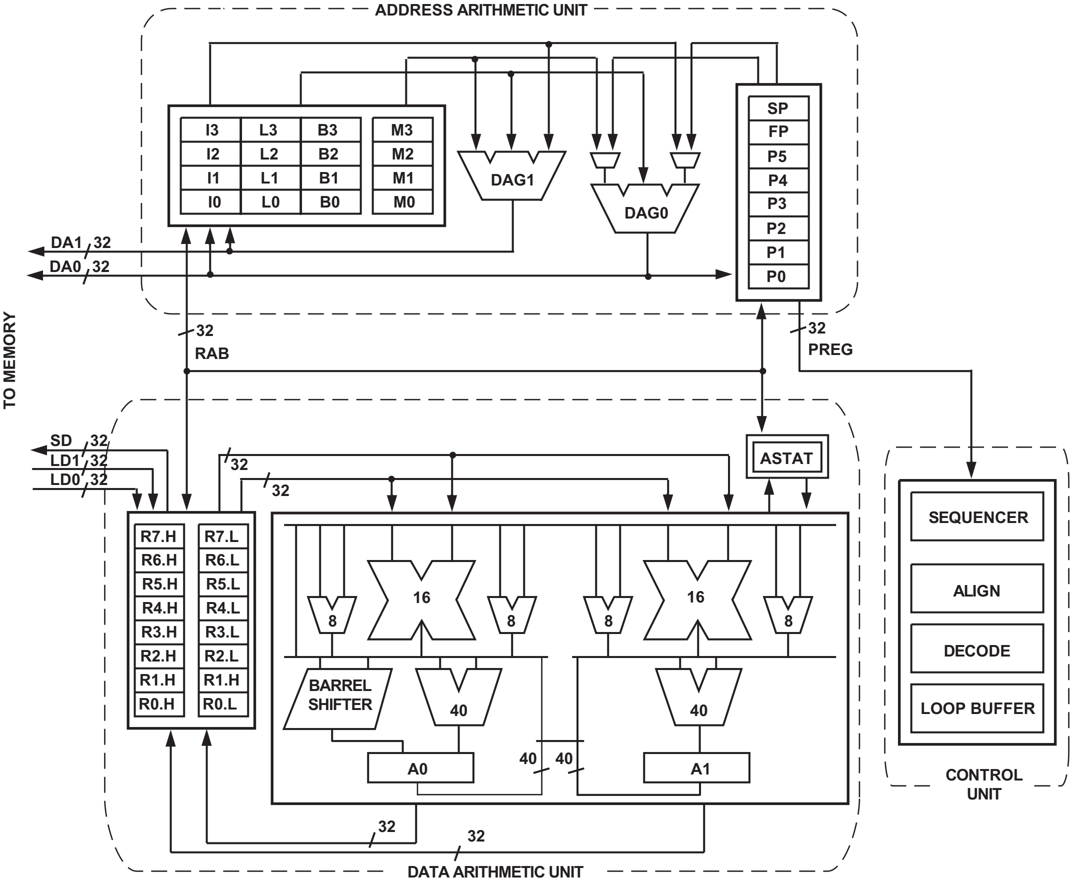


Figure 2-1. Processor Core Architecture

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.

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.

1.15 NUMBER (HEXADECIMAL)	DECIMAL EQUIVALENT
0x0001	0.000031
0x7FFF	0.999969
0xFFFF	–0.000031
0x8000	–1.000000

-2^{0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}

Figure 2-2. Bit Weighting for 1.15 Numbers

Register Files

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, `A0.W` contains the lower 32 bits of the 40-bit `A0` register; `A0.L` contains the lower 16 bits of `A0.W`, and `A0.H` contains the upper 16 bits of `A0.W`.

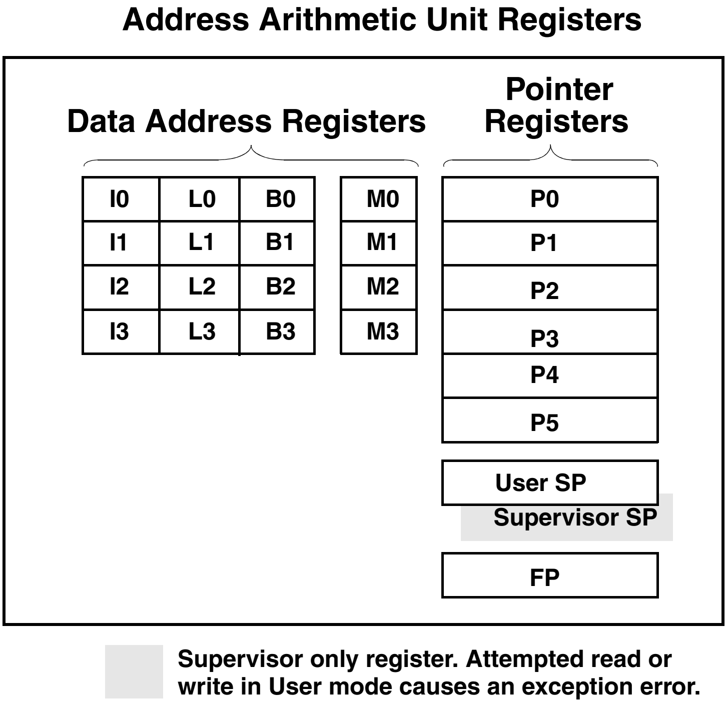

Address Arithmetic Unit Registers
Data Address Registers
Pointer Registers
I0
L0
B0
M0
I1
L1
B1
M1
I2
L2
B2
M2
I3
L3
B3
M3
P0
P1
P2
P3
P4
P5
User SP
Supervisor SP
FP
Supervisor only register. Attempted read or write in User mode causes an exception error.

Figure 2-3. Register Files

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 R0 may be regarded as two independent register halves, R0.L and R0.H.

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

```
R2 = R1 + R2;  /* 32-bit addition */
R2.L = R1.H * R0.L;  /* 16-bit multiplication */
```

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 (An.L) or high half (An.H) plus its 8-bit extension (An.X). Each can also be referred to as a 32-bit register (An.W) consisting of the lower 32 bits, or as a complete 40-bit result register (An).

These examples illustrate this convention:

```
A0 = A1;  /* 40-bit move */
A1.W = R7;  /* 32-bit move */
A0.H = R5.H; /* 16-bit move */
R6.H = A0.X; /* read 8-bit value and sign extend to 16 bits */
```

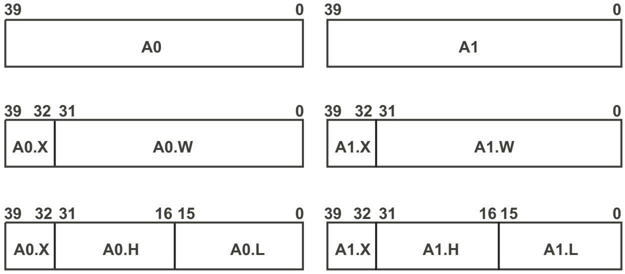


Figure 2-4. 40-Bit Accumulator Registers

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], B[3:0], L[3:0], A0.X, A0.W, A1.X, A1.W, ASTAT, RETS, RETI, RETX, RETN, RETE, LC[1:0], LT[1:0], LB[1:0], USP, SEQSTAT, SYSCFG, CYCLES, and CYCLES2.
- A*n* 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], 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.
- A*n*.L denotes the lower 16 bits of Accumulator A0.W or A1.W.
- A*n*.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.

- 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

Instruction	ASTAT Status Flags						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AVS	AV1 AV1S	CC	V V_COPY VS
allreg = allreg ; [1]	–	–	–	–	–	–	–
A*n* = A*n* ;	–	–	–	–	–	–	–
A*n* = Dreg ;	–	–	–	–	–	–	–
Dreg_even = A0 ;	*	*	–	–	–	–	*
Dreg_odd = A1 ;	*	*	–	–	–	–	*
Dreg_even = A0, Dreg_odd = A1 ;	*	*	–	–	–	–	*
Dreg_odd = A1, Dreg_even = A0 ;	*	*	–	–	–	–	*
IF CC DPreg = DPreg ;	–	–	–	–	–	–	–
IF ! CC DPreg = DPreg ;	–	–	–	–	–	–	–
Dreg = Dreg_lo (Z) ;	*	**	**	–	–	–	**/–
Dreg = Dreg_lo (X) ;	*	*	**	–	–	–	**/–
A*n*.X = Dreg_lo ;	–	–	–	–	–	–	–
Dreg_lo = A*n*.X ;	–	–	–	–	–	–	–
A*n*.L = Dreg_lo ;	–	–	–	–	–	–	–

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

Instruction	ASTAT Status Flags						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AVS	AV1 AV1S	CC	V V_COPY VS
A*n*.H = Dreg_hi ;	–	–	–	–	–	–	–
Dreg_lo = A0 ;	*	*	–	–	–	–	*
Dreg_hi = A1 ;	*	*	–	–	–	–	*
Dreg_hi = A1 ; Dreg_lo = A0 ;	*	*	–	–	–	–	*
Dreg_lo = A0 ; Dreg_hi = A1 ;	*	*	–	–	–	–	*
Dreg = Dreg_byte (Z) ;	*	**	**	–	–	–	**/–
Dreg = Dreg_byte (X) ;	*	*	**	–	–	–	**/–

[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

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.

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.

Table 2-2. Data Formats

Format	Representation in Memory	Representation in 32-bit Register
32.0 Unsigned Word	dddd dddd dddd dddd dddd dddd dddd dddd	dddd dddd dddd dddd dddd dddd dddd dddd
32.0 Signed Word	sddd dddd dddd dddd dddd dddd dddd dddd	sddd dddd dddd dddd dddd dddd dddd dddd
16.0 Unsigned Half Word	dddd dddd dddd dddd	0000 0000 0000 0000 dddd dddd dddd dddd
16.0 Signed Half Word	sddd dddd dddd dddd	ssss ssss ssss ssss sddd dddd dddd dddd
8.0 Unsigned Byte	dddd dddd	0000 0000 0000 0000 0000 0000 dddd dddd
8.0 Signed Byte	sddd dddd	ssss ssss ssss ssss ssss ssss sddd dddd
1.15 Signed Fraction	s.ddd dddd dddd dddd	ssss ssss ssss ssss s.ddd dddd dddd dddd
1.31 Signed Fraction	s.ddd dddd dddd dddd dddd dddd dddd dddd	s.ddd dddd dddd dddd dddd dddd dddd dddd
Packed 8.0 Unsigned Byte	dddd dddd *dddd dddd* dddd dddd *dddd dddd*	dddd dddd *dddd dddd* dddd dddd *dddd dddd*
Packed 1.15 Signed Fraction	s.ddd dddd dddd dddd *s.ddd dddd dddd dddd*	s.ddd dddd dddd dddd *s.ddd dddd dddd dddd*

Endianess

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

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.

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

Operation	Operand Formats	Result Formats
Addition	Signed or unsigned	Interpret flags
Subtraction	Signed or unsigned	Interpret flags
Logical	Binary string	Same as operands
Division	Explicitly signed or unsigned	Same as operands

Table 2-4. Multiplier Fractional Modes Formats

Operation	Operand Formats	Result Formats
Multiplication	1.15 explicitly signed or unsigned	2.30 shifted to 1.31
Multiplication/Addition	1.15 explicitly signed or unsigned	2.30 shifted to 1.31
Multiplication/Subtraction	1.15 explicitly signed or unsigned	2.30 shifted to 1.31

Table 2-5. Multiplier Arithmetic Integer Modes Formats

Operation	Operand Formats	Result Formats
Multiplication	16.0 explicitly signed or unsigned	32.0 not shifted

Table 2-5. Multiplier Arithmetic Integer Modes Formats (Cont'd)

Operation	Operand Formats	Result Formats
Multiplication/Addition	16.0 explicitly signed or unsigned	32.0 not shifted
Multiplication/Subtraction	16.0 explicitly signed or unsigned	32.0 not shifted

Table 2-6. Shifter Arithmetic Formats

Operation	Operand Formats	Result Formats
Logical Shift	Unsigned binary string	Same as operands
Arithmetic Shift	Signed	Same as operands
Exponent Detect	Signed	Same as operands

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.

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

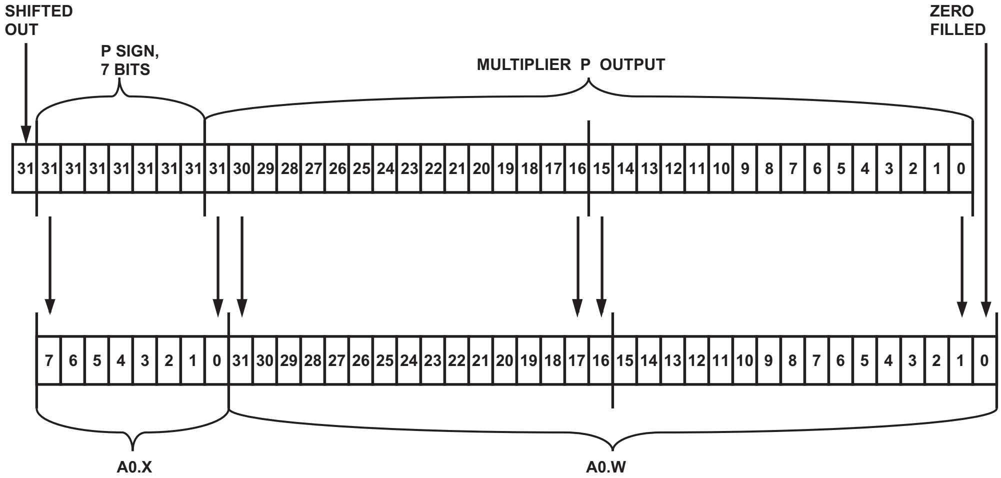


Figure 2-5. Fractional Multiplier Results Format

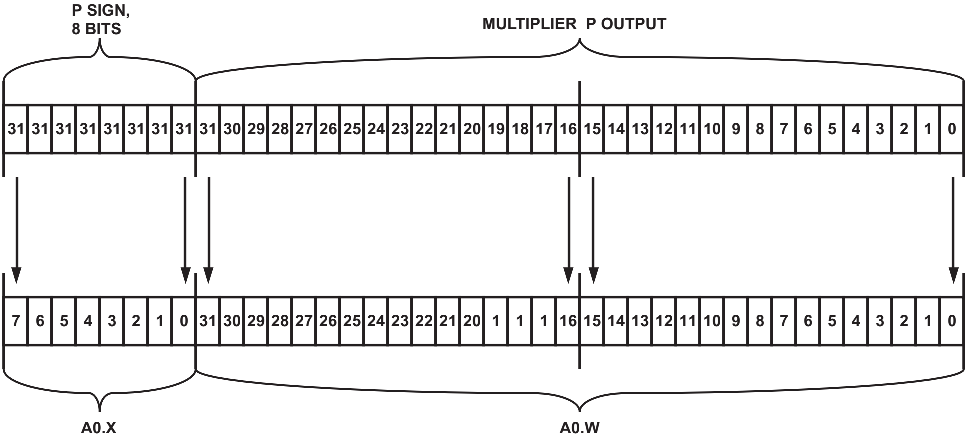


Figure 2-6. Integer Multiplier Results Format

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.

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`.

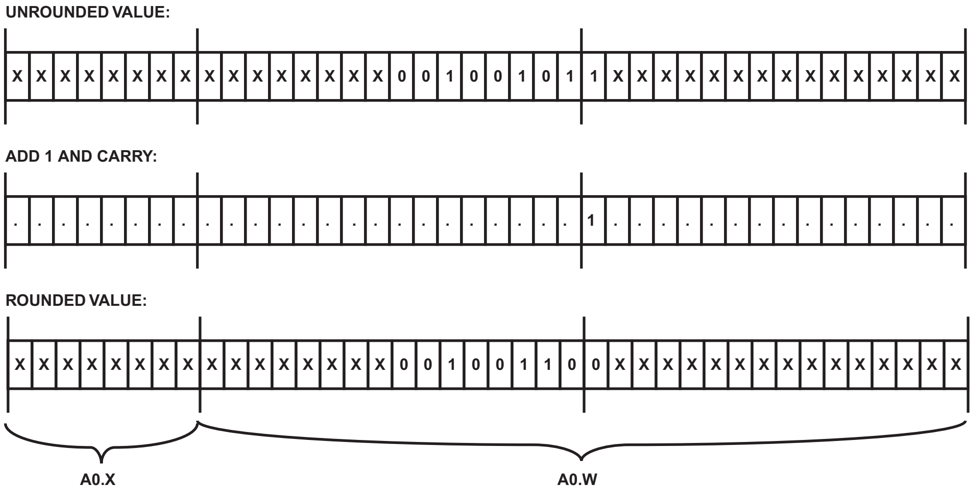

UNROUNDED VALUE:
X X X X X X X X X X X X X X X X 0 0 1 0 0 1 0 1 1 X X X X X X X X X X X X X X X
ADD 1 AND CARRY:
1
ROUNDED VALUE:
X X X X X X X X X X X X X X X X 0 0 1 0 0 1 1 0 0 X X X X X X X X X X X X X X X
A0.X
A0.W

Figure 2-7. Typical Unbiased Multiplier Rounding

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.

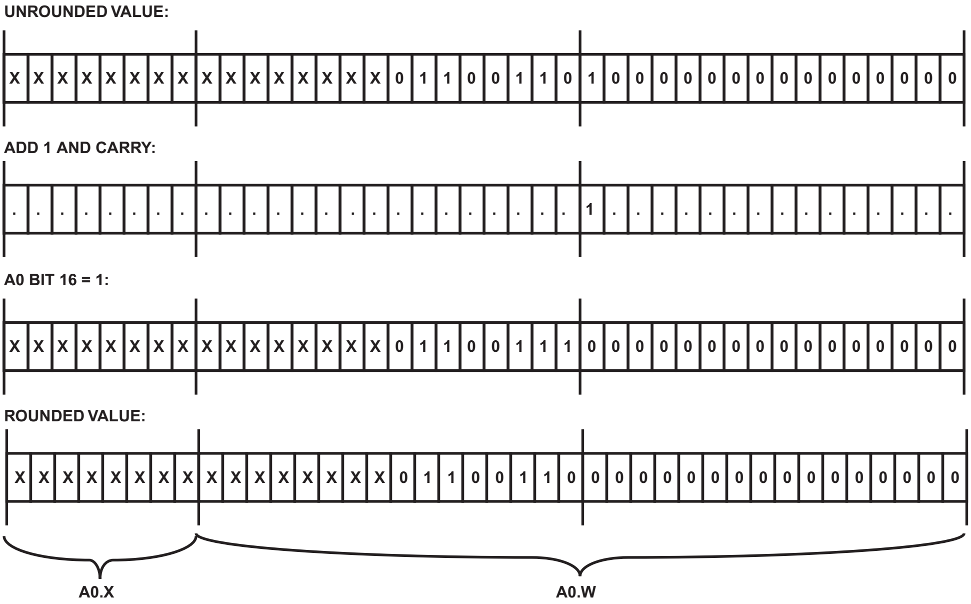

UNROUNDED VALUE:
X X X X X X X X X X X X X X X X 0 1 1 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
ADD 1 AND CARRY:
. . . . 1 . . . .
A0 BIT 16 = 1:
X X X X X X X X X X X X X X X X 0 1 1 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
ROUNDED VALUE:
X X X X X X X X X X X X X X X X 0 1 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
A0.X
A0.W

Figure 2-8. Avoiding Net Bias in 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

A0/A1 Before RND	Biased RND Result	Unbiased RND Result
0x00 0000 8000	0x00 0001 8000	0x00 0000 0000
0x00 0001 8000	0x00 0002 0000	0x00 0002 0000
0x00 0000 8001	0x00 0001 0001	0x00 0001 0001
0x00 0001 8001	0x00 0002 0001	0x00 0002 0001
0x00 0000 7FFF	0x00 0000 FFFF	0x00 0000 FFFF
0x00 0001 7FFF	0x00 0001 FFFF	0x00 0001 FFFF

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.

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) ;
```

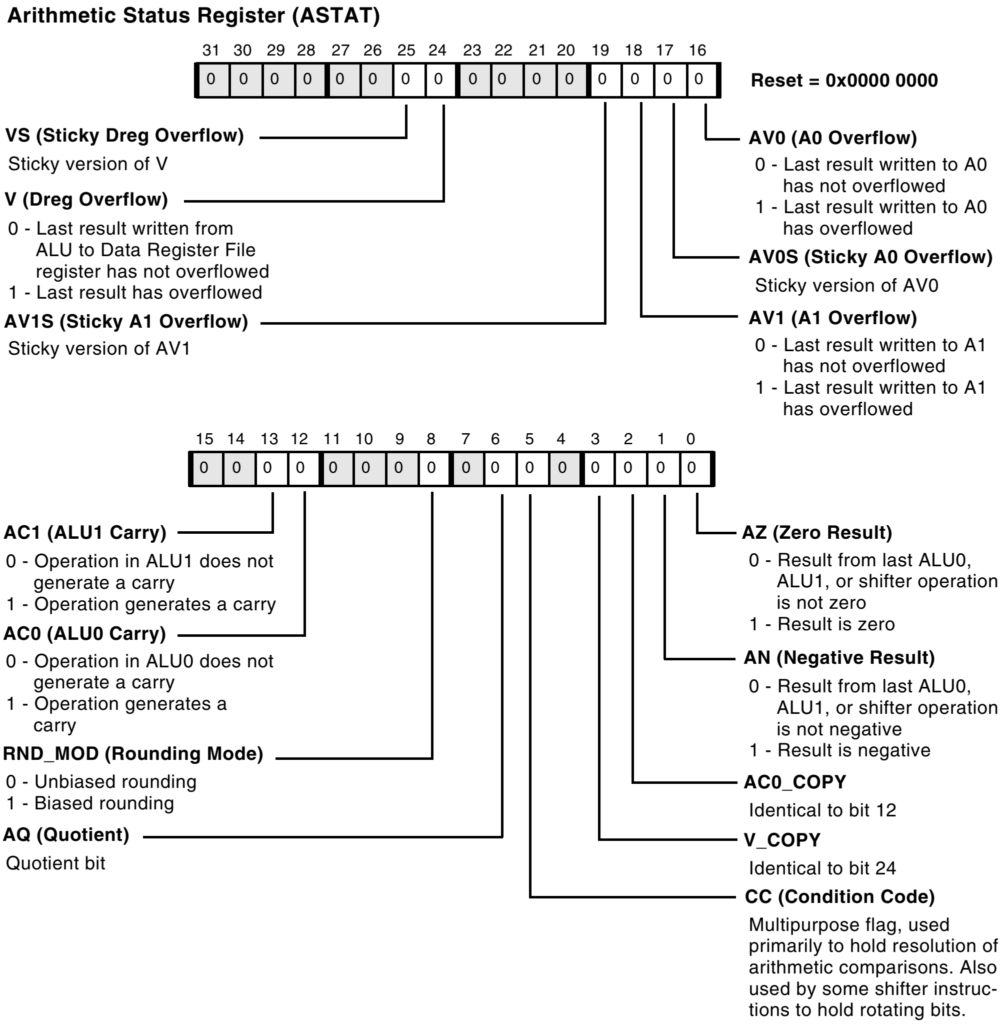
performs an addition of two 32-bit numbers, biased rounding at bit 20, depositing the result in a half word.

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)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
VS (Sticky Dreg Overflow)
Sticky version of V
V (Dreg Overflow)
0 - Last result written from ALU to Data Register File register has not overflowed
1 - Last result has overflowed
AV1S (Sticky A1 Overflow)
Sticky version of AV1
AV0 (A0 Overflow)
0 - Last result written to A0 has not overflowed
1 - Last result written to A0 has overflowed
AV0S (Sticky A0 Overflow)
Sticky version of AV0
AV1 (A1 Overflow)
0 - Last result written to A1 has not overflowed
1 - Last result written to A1 has overflowed
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
AC1 (ALU1 Carry)
0 - Operation in ALU1 does not generate a carry
1 - Operation generates a carry
AC0 (ALU0 Carry)
0 - Operation in ALU0 does not generate a carry
1 - Operation generates a carry
RND_MOD (Rounding Mode)
0 - Unbiased rounding
1 - Biased rounding
AQ (Quotient)
Quotient bit
AZ (Zero Result)
0 - Result from last ALU0, ALU1, or shifter operation is not zero
1 - Result is zero
AN (Negative Result)
0 - Result from last ALU0, ALU1, or shifter operation is not negative
1 - Result is negative
AC0_COPY
Identical to bit 12
V_COPY
Identical to bit 24
CC (Condition Code)
Multipurpose flag, used primarily to hold resolution of arithmetic comparisons. Also used by some shifter instructions to hold rotating bits.

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

Input	Output
Two or four 16-bit operands	One or two 16-bit results
Two 32-bit operands	One 32-bit result
32-bit result from the multiplier	Combination of 32-bit result from the multiplier with a 40-bit accumulation result

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.

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 +|- 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).

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) ;
```

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 (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 (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.

For example:

```
R3 = R1 + R2, R4 = R1 - R2 (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, R4 = A0 - A1 (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.

- A*n* denotes either ALU Result register A0 or A1.
- 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 AQ contains the dividend MSB Exclusive-OR divisor MSB.

Table 2-9. ALU Instruction Summary

Instruction	ASTAT Status Flags						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	V V_COPY VS	AQ
Dreg = Dreg + Dreg ;	*	*	*	–	–	*	–
Dreg = Dreg – Dreg (S) ;	*	*	*	–	–	*	–
Dreg = Dreg + Dreg, Dreg = Dreg – Dreg ;	*	*	*	–	–	*	–
Dreg_lo_hi = Dreg_lo_hi + Dreg_lo_hi ;	*	*	*	–	–	*	–
Dreg_lo_hi = Dreg_lo_hi – Dreg_lo_hi (S) ;	*	*	*	–	–	*	–
Dreg = Dreg +|+ Dreg ;	*	*	*	–	–	*	–
Dreg = Dreg +|– Dreg ;	*	*	*	–	–	*	–
Dreg = Dreg –|+ Dreg ;	*	*	*	–	–	*	–
Dreg = Dreg –|– Dreg ;	*	*	*	–	–	*	–
Dreg = Dreg +|+Dreg, Dreg = Dreg –|– Dreg ;	*	*	–	–	–	*	–
Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg ;	*	*	–	–	–	*	–
Dreg = A*n* + A*n*, Dreg = A*n* – A*n* ;	*	*	*	–	–	*	–
Dreg += imm7 ;	*	*	*	–	–	*	–
Dreg = (A0 += A1) ;	*	*	*	*	–	*	–
Dreg_lo_hi = (A0 += A1) ;	*	*	*	*	–	*	–
A0 += A1 ;	*	*	*	*	–	–	–
A0 –= A1 ;	*	*	*	*	–	–	–
DIVS (Dreg, Dreg) ;	*	*	*	*	–	–	d
DIVQ (Dreg, Dreg) ;	*	*	*	*	–	–	d

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

Instruction	ASTAT Status Flags						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	V V_COPY VS	AQ
Dreg = MAX (Dreg, Dreg) (V) ;	*	*	–	–	–	**/–	–
Dreg = MIN (Dreg, Dreg) (V) ;	*	*	–	–	–	**/–	–
Dreg = ABS Dreg (V) ;	*	**	–	–	–	*	–
A*n* = ABS A*n* ;	*	**	–	*	*	*	–
A*n* = ABS A*n*, A*n* = ABS A*n* ;	*	**	–	*	*	*	–
A*n* = –A*n* ;	*	*	*	*	*	*	–
A*n* = –A*n*, A*n* =– A*n* ;	*	*	*	*	*	*	–
A*n* = A*n* (S) ;	*	*	–	*	*	–	–
A*n* = A*n* (S), A*n* = A*n* (S) ;	*	*	–	*	*	–	–
Dreg_lo_hi = Dreg (RND) ;	*	*	–	–	–	*	–
Dreg_lo_hi = Dreg + Dreg (RND12) ;	*	*	–	–	–	*	–
Dreg_lo_hi = Dreg – Dreg (RND12) ;	*	*	–	–	–	*	–
Dreg_lo_hi = Dreg + Dreg (RND20) ;	*	*	–	–	–	*	–
Dreg_lo_hi = Dreg – Dreg (RND20) ;	*	*	–	–	–	*	–
Dreg_lo = SIGNBITS Dreg ;	–	–	–	–	–	–	–
Dreg_lo = SIGNBITS Dreg_lo_hi ;	–	–	–	–	–	–	–
Dreg_lo = SIGNBITS An ;	–	–	–	–	–	–	–

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

Instruction	ASTAT Status Flags						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	V V_COPY VS	AQ
Dreg_lo = EXPADJ (Dreg, Dreg_lo) (V) ;	–	–	–	–	–	–	–
Dreg_lo = EXPADJ (Dreg_lo_hi, Dreg_lo);	–	–	–	–	–	–	–
Dreg = Dreg & Dreg ;	*	*	**	–	–	**/–	–
Dreg = ~ Dreg ;	*	*	**	–	–	**/–	–
Dreg = Dreg | Dreg ;	*	*	**	–	–	**/–	–
Dreg = Dreg ^ Dreg ;	*	*	**	–	–	**/–	–
Dreg =– Dreg ;	*	*	*	–	–	*	–

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."

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.

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, `A1` or `A0`. 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.

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.

- 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 ASTAT indicate multiplier overflow status:

- Bit 16 (AV0) and bit 18 (AV1) record overflow condition (whether the result has overflowed 32 bits) for the A0 and 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 AV0S and AV1S bits are sticky bits.

- Bit 24 (V) and bit 25 (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.

- A*n* denotes either MAC Accumulator register A0 or A1.
- * 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

Instruction	**ASTAT Status Flags**		
	AV0 **AV0S**	**AV1** **AV1S**	**V** **V_COPY** **VS**
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ;	–	–	*
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi ;	–	–	*
Dreg = Dreg_lo_hi * Dreg_lo_hi ;	–	–	*
A*n* = Dreg_lo_hi * Dreg_lo_hi ;	*	*	–
A*n* += Dreg_lo_hi * Dreg_lo_hi ;	*	*	–
An –= Dreg_lo_hi * Dreg_lo_hi ;	*	*	–
Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_lo = (A0 –= Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_hi = (A1 –= Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg = (A*n* = Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg = (A*n* += Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg = (A*n* –= Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg *= Dreg ;	–	–	–

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.

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 x 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.

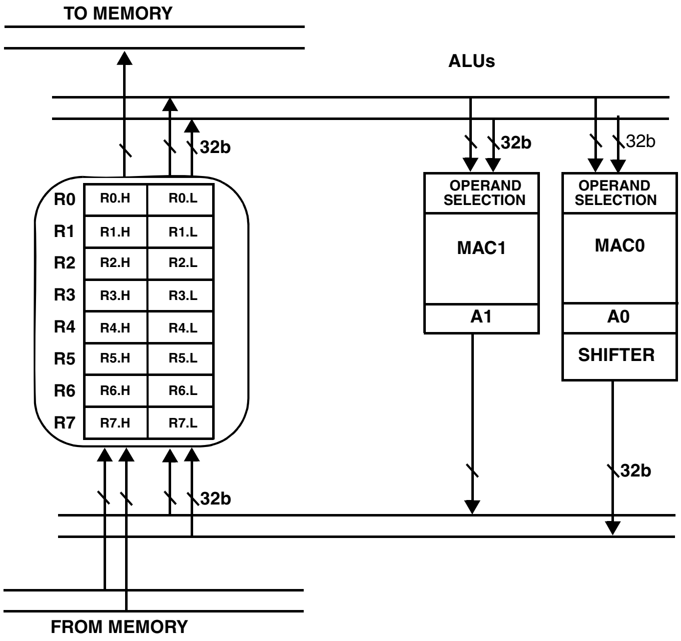


Figure 2-10. Register Files and ALUs

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 chose 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 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.

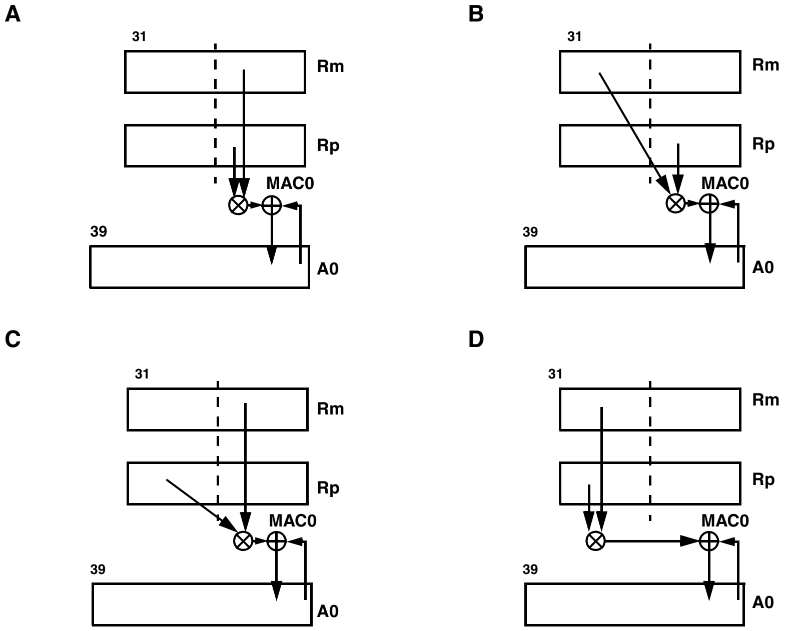


Figure 2-11. Four Possible Combinations of MAC Operations

The 32-bit product is passed to a 40-bit adder/subtracter, 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`.

For example:

```
A1 += R3.H * 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 IH 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).

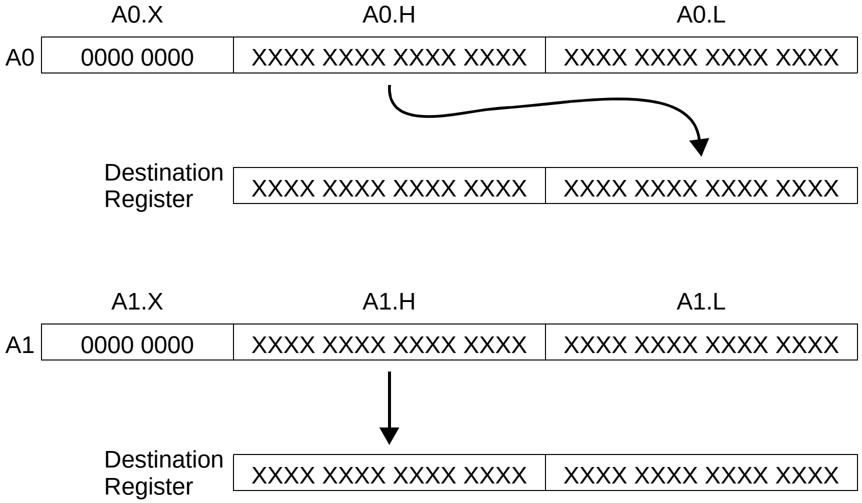


Figure 2-12. Multiplication of Fractional Operands

For example, this instruction uses fractional, unsigned operands:

```
R0.L = R1.L * R2.L (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 * R3.H (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 * R2.L ;
```

Regardless of operand type, the preceding operation deposits 32 bits of the multiplier answer with saturation into R0, using MAC0.

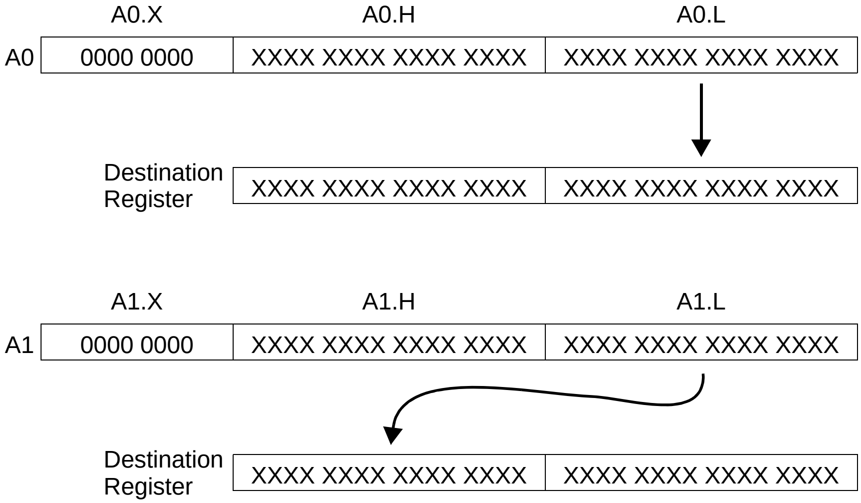


Figure 2-13. Multiplication of Integer Operands

Special 32-Bit Integer MAC Instruction

The processor supports a multicycle 32-bit MAC instruction:

```
Dreg *= 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 * R2.L, A0 += R1.L * 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 * R2.L), R3.L = (A0 += R1.L * 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.

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 * R2.L), R2 = (A0 += R1.L * 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 * R2.L), A0 += R1.L * 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 (>>>, >>, <<, ASHIFT, LSHIFT, ROT) can be used various ways, depending on the underlying arithmetic requirements. The ASHIFT and >>> instructions represent the arithmetic shift. The LSHIFT, <<, and >> instructions represent the logical shift.

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 ASHIFT and 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 >>> and >> 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.

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

results in

```
R0 contains 0000 0B6A ;
```

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 ;
```

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

Instruction	ASTAT Status Flag						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	CC	V V_COPY VS
BITCLR (Dreg, uimm5) ;	*	*	**	–	–	–	**/–
BITSET (Dreg, uimm5) ;	**	*	**	–	–	–	**/–
BITTGL (Dreg, uimm5) ;	*	*	**	–	–	–	**/–
CC = BITTST (Dreg, uimm5) ;	–	–	–	–	–	*	–
CC = !BITTST (Dreg, uimm5) ;	–	–	–	–	–	*	–
Dreg = DEPOSIT (Dreg, Dreg) ;	*	*	**	–	–	–	**/–
Dreg = EXTRACT (Dreg, Dreg) ;	*	*	**	–	–	–	**/–
BITMUX (Dreg, Dreg, A0) ;	–	–	–	–	–	–	–
Dreg_lo = ONES Dreg ;	–	–	–	–	–	–	–
Dreg = PACK (Dreg_lo_hi, Dreg_lo_hi);	–	–	–	–	–	–	–
Dreg >>>= uimm5 ;	*	*	–	–	–	–	**/–
Dreg >>= uimm5 ;	*	*	–	–	–	–	**/–
Dreg <<= uimm5 ;	*	*	–	–	–	–	**/–
Dreg = Dreg >>> uimm5 ;	*	*	–	–	–	–	**/–
Dreg = Dreg >> uimm5 ;	*	*	–	–	–	–	**/–
Dreg = Dreg << uimm5 ;	*	*	–	–	–	–	*
Dreg = Dreg >>> uimm4 (V) ;	*	*	–	–	–	–	**/–
Dreg = Dreg >> uimm4 (V) ;	*	*	–	–	–	–	**/–
Dreg = Dreg << uimm4 (V) ;	*	*	–	–	–	–	*

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

Instruction	ASTAT Status Flag						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	CC	V V_COPY VS
A*n* = A*n* >>> uimm5 ;	*	*	–	** 0/–	** 1/–	–	–
A*n* = A*n* >> uimm5 ;	*	*	–	** 0/–	** 1/–	–	–
A*n* = A*n* << uimm5 ;	*	*	–	* 0	* 1	–	–
Dreg_lo_hi = Dreg_lo_hi >>> uimm4 ;	*	*	–	–	–	–	**/–
Dreg_lo_hi = Dreg_lo_hi >> uimm4 ;	*	*	–	–	–	–	**/–
Dreg_lo_hi = Dreg_lo_hi << uimm4 ;	*	*	–	–	–	–	*
Dreg >>>= Dreg ;	*	*	–	–	–	–	**/–
Dreg >>= Dreg ;	*	*	–	–	–	–	**/–
Dreg <<= Dreg ;	*	*	–	–	–	–	**/–
Dreg = ASHIFT Dreg BY Dreg_lo ;	*	*	–	–	–	–	*
Dreg = LSHIFT Dreg BY Dreg_lo ;	*	*	–	–	–	–	**/–
Dreg = ROT Dreg BY imm6 ;	–	–	–	–	–	***	–
Dreg = ASHIFT Dreg BY Dreg_lo (V) ;	*	*	–	–	–	–	*
Dreg = LSHIFT Dreg BY Dreg_lo (V) ;	*	*	–	–	–	–	**/–
Dreg_lo_hi = ASHIFT Dreg_lo_hi BY Dreg_lo ;	*	*	–	–	–	–	*

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

Instruction	ASTAT Status Flag						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	CC	V V_COPY VS
Dreg_lo_hi = LSHIFT Dreg_lo_hi BY Dreg_lo ;	*	*	–	–	–	–	**/–
A*n* = A*n* ASHIFT BY Dreg _lo ;	*	*	–	* 0	* 1	–	–
A*n* = A*n* ROT BY imm6 ;	–	–	–	–	–	***	–
Dreg = (Dreg + Dreg) << 1 ;	*	*	*	–	–	–	*
Dreg = (Dreg + Dreg) << 2 ;	*	*	*	–	–	–	*

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.

Table 3-1. Identifying the Current Processor Mode

Event	Mode	IPEND
Interrupt	Supervisor	≥ 0x10 but IPEND[0], IPEND[1], IPEND[2], and IPEND[3] = 0.
Exception	Supervisor	≥ 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.
NMI	Supervisor	≥ 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.
Reset	Supervisor	= 0x02 As the reset state is exited, IPEND is set to 0x02, and the reset vector runs in Supervisor mode.
Emulation	Emulator	= 0x01 The processor is in Emulation mode if IPEND[0] = 1, regardless of the state of the remaining bits IPEND[15:1].
None	User	= 0x00

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.

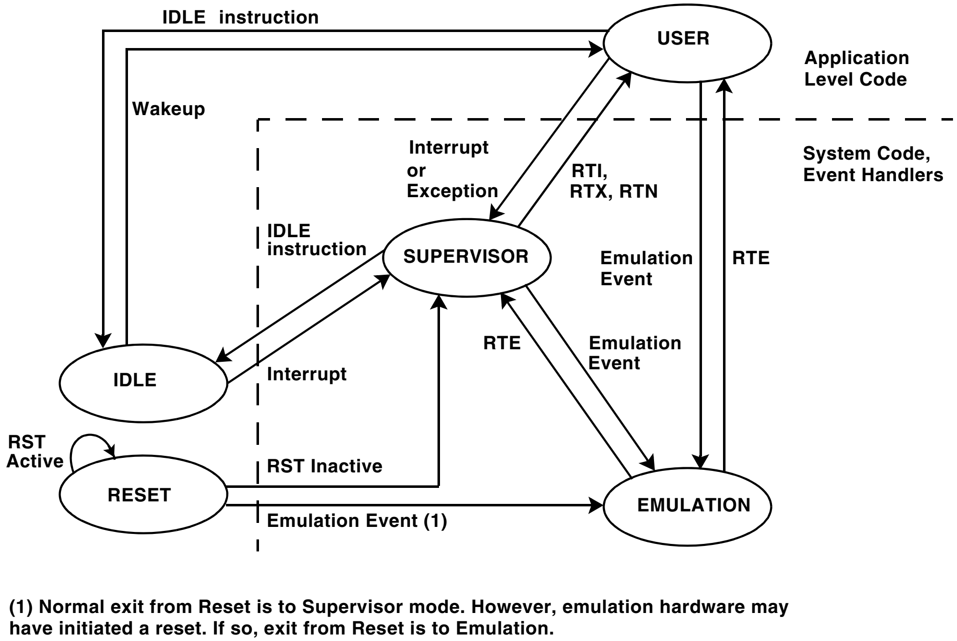


Figure 3-1. Processor Modes and States

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.

Table 3-2. Registers Accessible in User Mode

Processor Registers	Register Names
Data Registers	R[7:0], A[1:0]
Pointer Registers	P[5:0], SP, FP, I[3:0], M[3:0], L[3:0], B[3:0]
Sequencer and Status Registers	RETS, LC[1:0], LT[1:0], LB[1:0], ASTAT, CYCLES, CYCLES2

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

Instruction	Description
RTI	Return from Interrupt
RTX	Return from Exception
RTN	Return from NMI
CLI	Disable Interrupts
STI	Enable Interrupts
RAISE	Force Interrupt/Reset
RTE	Return from Emulation Causes an exception only if executed outside Emulation mode

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

Current Process Activity	Return Instruction to Use	Execution Resumes at Address in This Register
Interrupt Service Routine	RTI	RETI
Exception Service Routine	RTX	RETX
Nonmaskable Interrupt Service Routine	RTN	RETN
Emulation Service Routine	RTE	RETE

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 $\overline{\texttt{RESET}}$ 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.

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) ;

R0 = [P0] ;
R1.L = EVT_IVG15 & 0xFFFF ;

R0 = R0 | R1 ;
[P0] = R0 ;   /* 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 */
```

```
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 EMUEXCPT 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.

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 $\overline{\text{RESET}}$ signal. Upon deassertion, the processor completes the reset sequence and switches to Supervisor mode, where it executes code found at the reset event vector.

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

Item	**Description of Reset State**
Core	
Operating Mode	Supervisor mode in reset event, clocks stopped
Rounding Mode	Unbiased rounding
Cycle Counters	Disabled, zero
DAG Registers (I, L, B, M)	Random values (must be cleared at initialization)
Data and Address Registers	Random values (must be cleared at initialization)
IPEND, IMASK, ILAT	Cleared, interrupts globally disabled with IPEND bit 4
CPLBs	Disabled
L1 Instruction Memory	SRAM (cache disabled)
L1 Data Memory	SRAM (cache disabled)
Cache Validity Bits	Invalid
System	
Booting Methods	Determined by the values of BMODE pins at reset
MSEL Clock Frequency	Reset value = 10
PLL Bypass Mode	Disabled
VCO/Core Clock Ratio	Reset value = 1
VCO/System Clock Ratio	Reset value = 5
Peripheral Clocks	Disabled

System Reset and Powerup

Table 3-6 describes the five types of resets. Note all resets, except System Software, reset the core.

Table 3-6. Resets

Reset	Source	Result
Hardware Reset	The $\overline{\text{RESET}}$ pin causes a hardware reset.	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 "SYSCR Register" on page 3-14.
System Software Reset	Writing b#111 to bits [2:0] in the system MMR SWRST at address 0xFFC0 0100 causes a System Software reset.	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.
Watchdog Timer Reset	Programming the watchdog timer appropriately causes a Watchdog Timer reset.	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.

Table 3-6. Resets (Cont'd)

Reset	Source	Result
Core Double-Fault Reset	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).	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.
Core-Only Software Reset	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.	Resets only the core. The peripherals do not recognize this reset.

Hardware Reset

The processor chip reset is an asynchronous reset event. The $\overline{\texttt{RESET}}$ 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 $\overline{\texttt{RESET}}$ 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 V_{DD} or V_{SS}. The pins and the corresponding bits in SYSCR configure the Boot mode that is employed after hardware reset or

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 $\overline{\text{RESET}}$ 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.

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 processors 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).

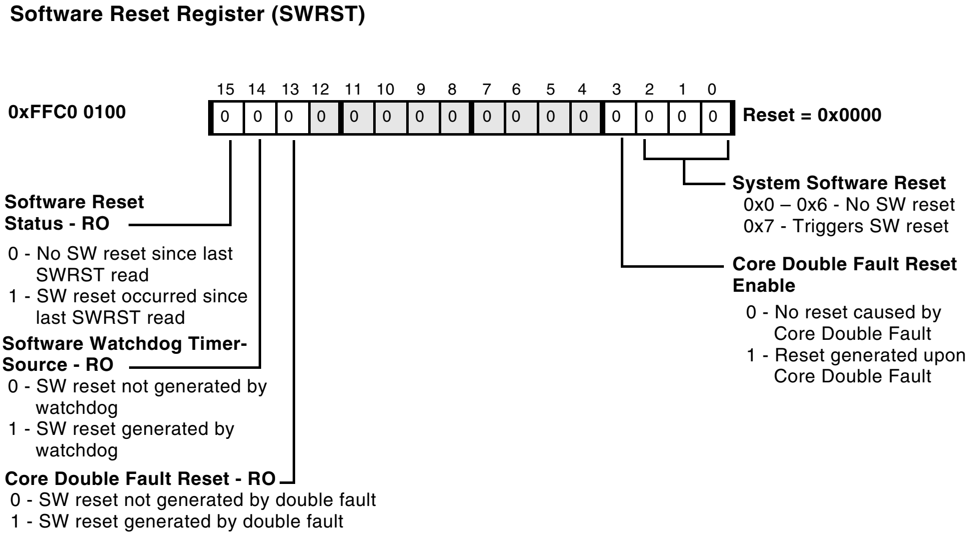

Software Reset Register (SWRST)
0xFFC0 0100
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
System Software Reset
0x0 – 0x6 - No SW reset
0x7 - Triggers SW reset
Core Double Fault Reset Enable
0 - No reset caused by Core Double Fault
1 - Reset generated upon Core Double Fault
Software Reset Status - RO
0 - No SW reset since last SWRST read
1 - SW reset occurred since last SWRST read
Software Watchdog Timer- Source - RO
0 - SW reset not generated by watchdog
1 - SW reset generated by watchdog
Core Double Fault Reset - RO
0 - SW reset not generated by double fault
1 - SW reset generated by double fault

Figure 3-2. Software Reset Register

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. 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 ;
```

4 PROGRAM SEQUENCER

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.

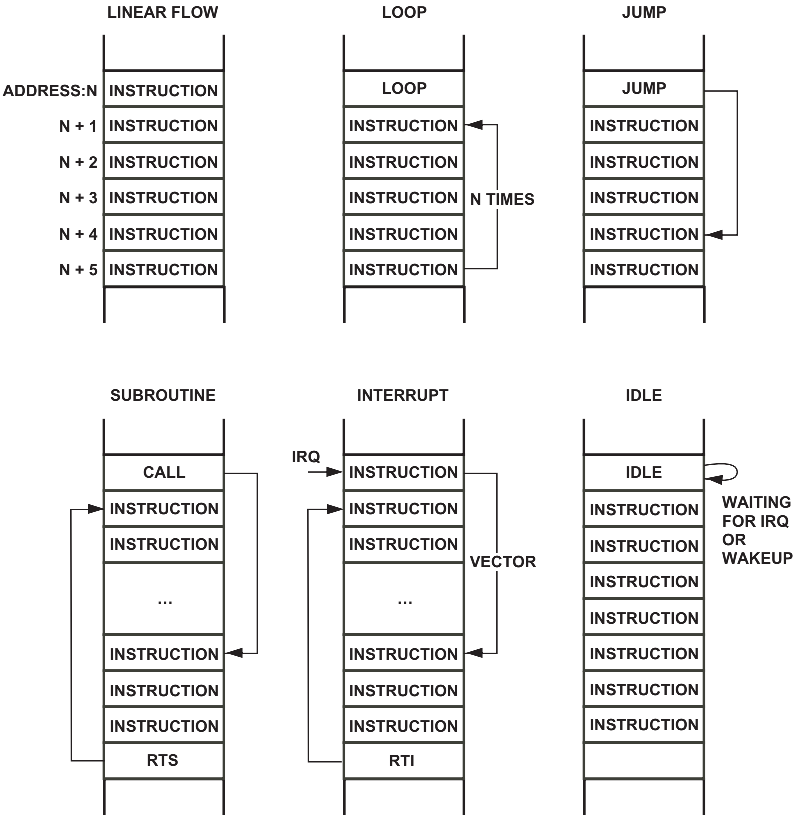


Figure 4-1. Program Flow Variations

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.

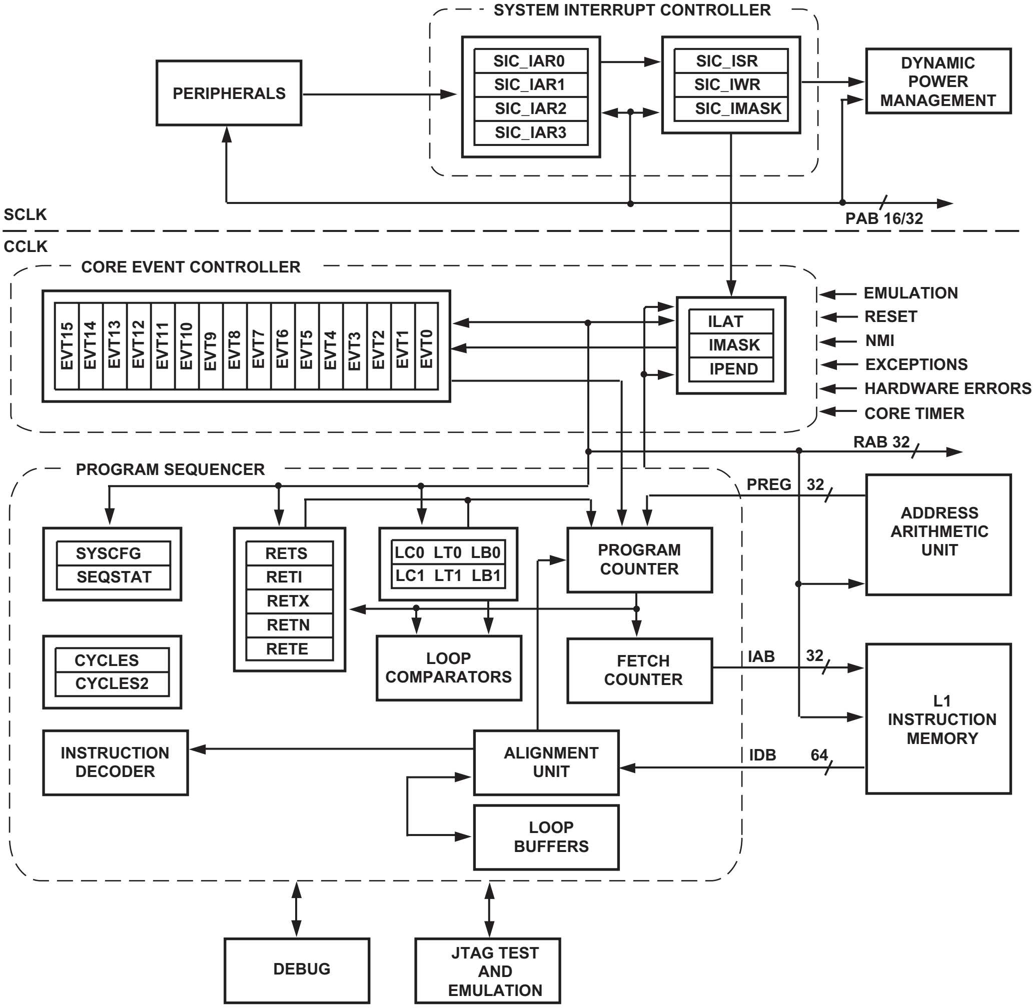


Figure 4-2. Program Sequencing and Interrupt Processing Block Diagram

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

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

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.”

Instruction Pipeline

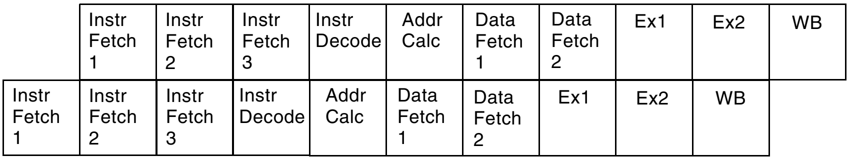
The program sequencer determines the next instruction address by examining both the current instruction being executed and the current state of the processor. If no conditions require otherwise, the processor executes instructions from memory in sequential order by incrementing the look-ahead address.

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

Table 4-2. Stages of Instruction Pipeline

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

Figure 4-3 shows a diagram of the pipeline.



	Instr Fetch 1	Instr Fetch 2	Instr Fetch 3	Instr Decode	Addr Calc	Data Fetch 1	Data Fetch 2	Ex1	Ex2	WB
Instr Fetch 1	Instr Fetch 2	Instr Fetch 3	Instr Decode	Addr Calc	Data Fetch 1	Data Fetch 2	Ex1	Ex2	WB	

Figure 4-3. Processor Pipeline

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.

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

Branches

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.

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`s and `CALL`s 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 0xnnnnnnn` instruction must be used. If the destination offset is unknown and development tools must evaluate the offset, then use the instruction `JUMP 0xnnnnnnn`. 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.

```
    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);
```

```
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 */
```

```
  [--SP] = (R7:7, P5:5);  /* multiple push instruction */
  P5.H = HI(myregister);  /* P5 used locally */
  P5.L = LO(myregister);
  R7 = [P5];  /* R7 used locally */
  R0 = R0 + R7;  /* R0 user for parameter passing */
  (R7:7, P5:5) = [SP++];  /* multiple pop instruction */
  RTS;  /* return from subroutine */
myfunction.end:  /* closing 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 */
 [--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.

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

LINK n;	UNLINK;

[--SP] = RETS;
[--SP] = FP;
FP = SP;
SP += -n;
	SP = FP;
	FP = [SP++];
	RETS = [SP++];

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”.

```
_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:
```

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 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:

```
_sub3:
  LINK 8;
  [--SP] = (R7:0, P5:0);

  R7 = 0 (Z);
  [FP-4] = R7;
  [FP-8] = R7;
  ...
  (R7:0, P5:0) = [SP++];
  UNLINK;
  RTS;
_sub3.end:
```

For more information on the LINK and UNLINK instructions, see "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 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

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 `CC` flag and, if it is positive, jumps to a location identified by the label `dest_address`:

```
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 `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:

```
IF CC R0 = P0 ;
```

Branch Prediction

The sequencer supports static branch prediction to accelerate execution of conditional branches. These branches are executed based on the state of the `CC` bit.

In the EX2 stage, the sequencer compares the actual `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.

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

Instruction	Description
If CC JUMP dest (bp)	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.

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).

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

Registers	Description	Function
LC0, LC1	Loop Counters	Maintains a count of the remaining iterations of the loop
LT0, LT1	Loop Tops	Holds the address of the first instruction within a loop
LB0, LB1	Loop Bottoms	Holds the address of the last instruction of the loop

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 ) LC0 = 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 LBx and the address of the loop's first instruction into 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 LC0 and LC1 registers are unsigned 32-bit registers, each supporting 2^{32} –1 iterations through the loop.

When 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) LC0 = P5;
lp_start:    ...
lp_end:      ...
lp_skip:  /* first instruction outside the loop */
```

Table 4-6. Loop Registers

First/Last Address of the Loop	PC-Relative Offset Used to Compute the Loop Start Address	Effective Range of the Loop Start Instruction
Top / First	5-bit signed immediate; must be a multiple of 2.	0 to 30 bytes away from LSETUP instruction.
Bottom / Last	11-bit signed immediate; must be a multiple of 2.	0 to 2046 bytes away from LSETUP instruction (the defined loop can be 2046 bytes long).

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

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.

```
#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++] ;
    LSETUP (lpi_start, lpi_end) LC1 = P5;
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.

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:

```
#define N 1024
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:

```
[--SP] = LC0;
[--SP] = LB0;
[--SP] = LT0;
```

To restore Loop Unit 0 from system stack, use:

```
LT0 = [SP++];
LB0 = [SP++];
LC0 = [SP++];
```

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.

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.

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 (`IVG7` – `IVG15`) 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 (`IVG14` and `IVG15`) be reserved for software interrupt handlers, leaving seven prioritized interrupt inputs (`IVG7` – `IVG13`) to support the system. Refer to the product data sheet for the default system interrupt mapping.

Table 4-7. Core Event Mapping

	Event Source	**Core Event Name**
Core Events	Emulation (highest priority)	EMU
	Reset	RST
	NMI	NMI
	Exception	EVX
	Reserved	–
	Hardware Error	IVHW
	Core Timer	IVTMR

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.

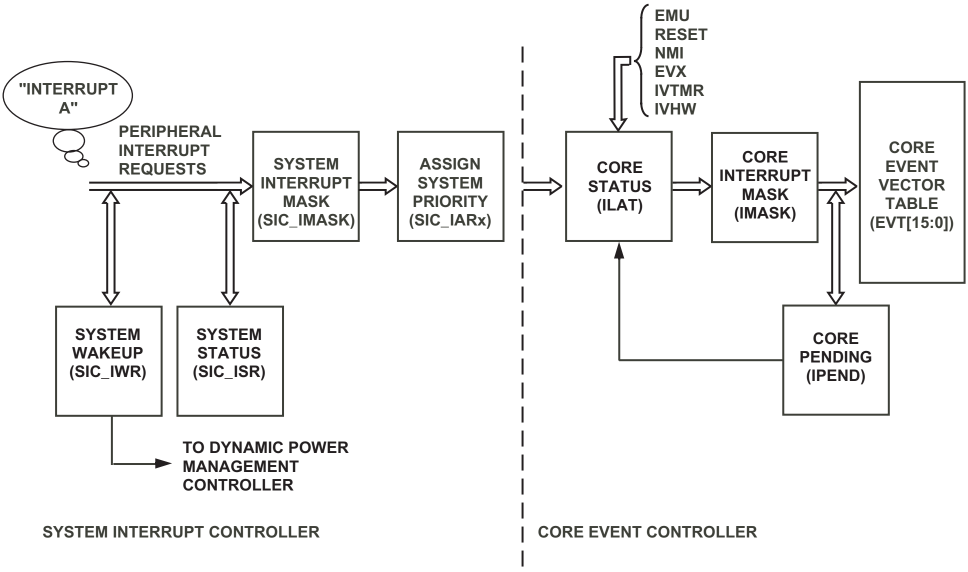
7. The Event Vector Table (EVT) is accessed to look up the appropriate vector for Interrupt A's interrupt service routine (ISR).

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.


"INTERRUPT A"
PERIPHERAL INTERRUPT REQUESTS
SYSTEM INTERRUPT MASK (SIC_IMASK)
ASSIGN SYSTEM PRIORITY (SIC_IARx)
SYSTEM WAKEUP (SIC_IWR)
SYSTEM STATUS (SIC_ISR)
TO DYNAMIC POWER MANAGEMENT CONTROLLER
SYSTEM INTERRUPT CONTROLLER
EMU
RESET
NMI
EVX
IVTMR
IVHW
CORE STATUS (ILAT)
CORE INTERRUPT MASK (IMASK)
CORE EVENT VECTOR TABLE (EVT[15:0])
CORE PENDING (IPEND)
CORE EVENT CONTROLLER

NOTE: NAMES IN PARENTHESES ARE MEMORY-MAPPED REGISTERS.

Figure 4-4. Interrupt Processing Block Diagram

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.

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.

The SIC_IWR register has no effect unless the core is idled. The bits in this register correspond to those of the System Interrupt Mask (SIC_IMASK) and Interrupt Status (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 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.

> ⓘ Note the wakeup function is independent of the interrupt mask function. If an interrupt source is enabled in SIC_IWR but masked off in 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 *Blackfin Processor Hardware Reference* for your part.

SIC_ISR Register

The System Interrupt Controller (SIC) includes a read-only status register, the System Interrupt Status register (SIC_ISR), shown in the System Interrupt Appendix of the *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.

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.

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.

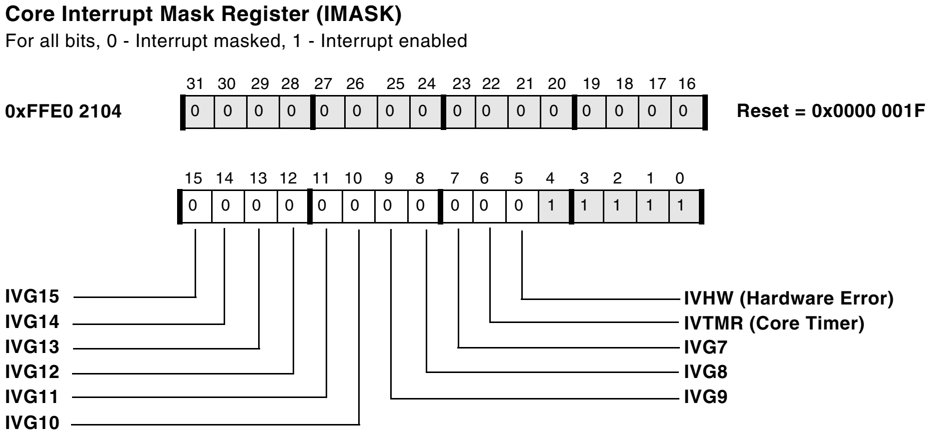


Figure 4-5. Core Interrupt Mask Register

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].

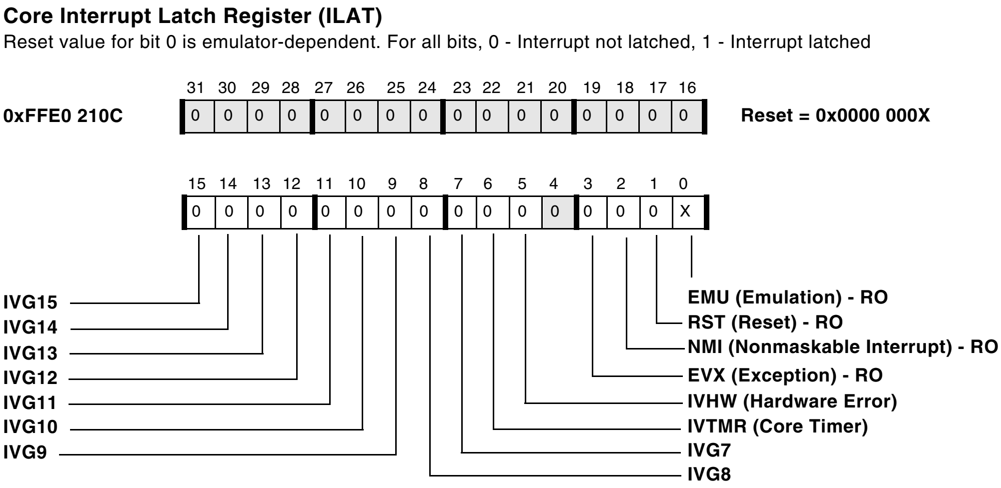


Figure 4-6. Core Interrupt Latch Register

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.

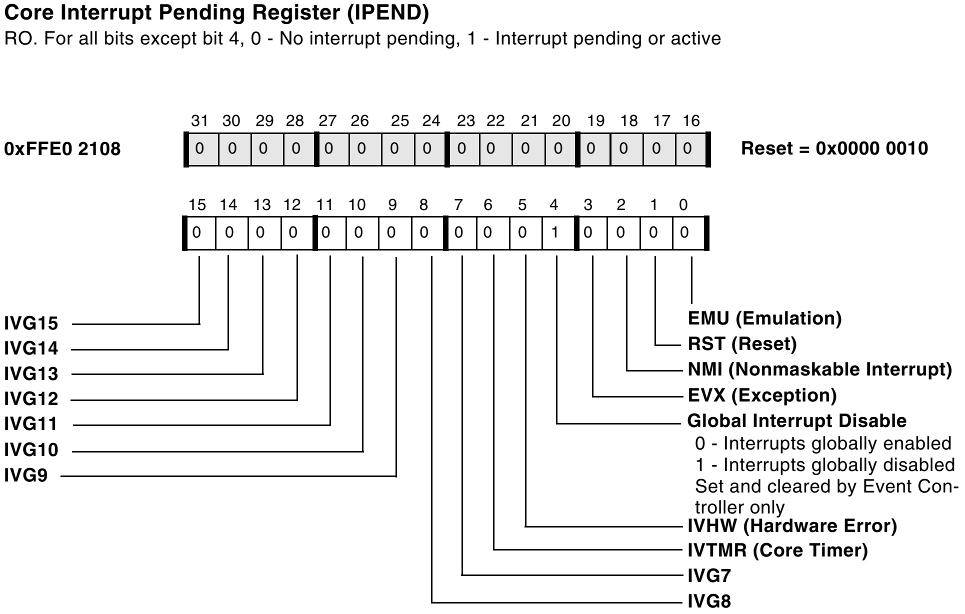


Figure 4-7. Core Interrupt Pending Register

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.

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

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

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 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:
```

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:

```
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 */
  RETI = [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

Name	Event Class	Return Register	Return Instruction
EMU	Emulation	RETE	RTE
RST	Reset	RETI	RTI
NMI	NMI	RETN	RTN
EVX	Exception	RETX	RTX
Reserved	Reserved	-	-
IVHW	Hardware Error	RETI	RTI
IVTMR	Core Timer	RETI	RTI
IVG7	Interrupt 7	RETI	RTI

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

Name	Event Class	Return Register	Return Instruction
IVG8	Interrupt 8	RETI	RTI
IVG9	Interrupt 9	RETI	RTI
IVG10	Interrupt 10	RETI	RTI
IVG11	Interrupt 11	RETI	RTI
IVG12	Interrupt 12	RETI	RTI
IVG13	Interrupt 13	RETI	RTI
IVG14	Interrupt 14	RETI	RTI
IVG15	Interrupt 15	RETI	RTI

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.

Reset Interrupt

The reset interrupt (RST) can be initiated via the $\overline{\text{RESET}}$ 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 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 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 RETX.

If unused, the NMI pin should always be pulled to its deasserted state. On some derivatives, the 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 ILAT, 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 RETI for interrupts, RETX for exceptions, RETN for NMIs, and 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 ILAT is cleared and the corresponding bit in IPEND is set.

   Bit IPEND[4] is also set to disable all interrupts until the return address in RETI is saved.

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

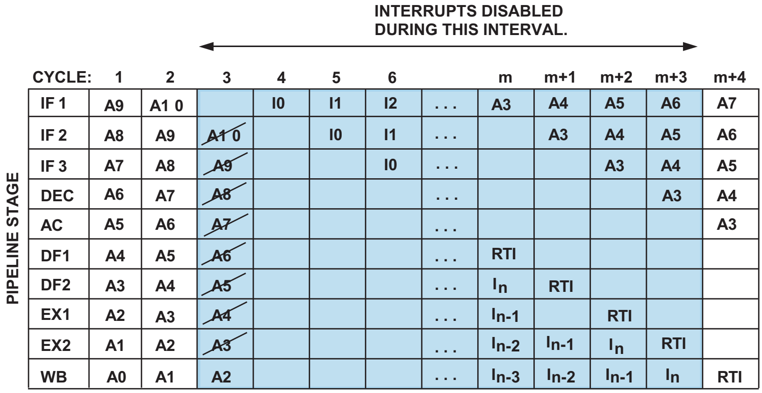
If interrupts do not require nesting, all interrupts are disabled during the interrupt service routine. Note, however, that emulation, NMI, and exceptions are still accepted by the system.

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



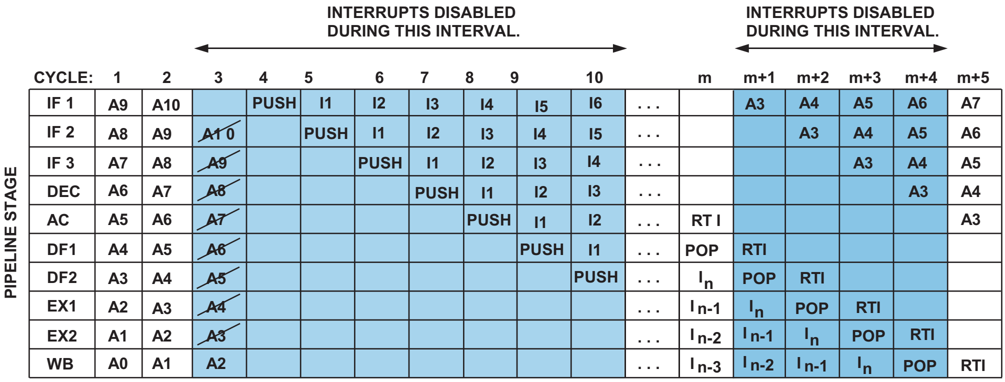
CYCLE:	1	2	3	4	5	6		m	m+1	m+2	m+3	m+4
IF 1	A9	A1 0		I0	I1	I2	. . .	A3	A4	A5	A6	A7
IF 2	A8	A9	~~A1 0~~		I0	I1	. . .		A3	A4	A5	A6
IF 3	A7	A8	~~A9~~			I0	. . .			A3	A4	A5
DEC	A6	A7	~~A8~~				. . .				A3	A4
AC	A5	A6	~~A7~~				. . .					A3
DF1	A4	A5	~~A6~~				. . .	RTI				
DF2	A3	A4	~~A5~~				. . .	I_n	RTI			
EX1	A2	A3	~~A4~~				. . .	I_{n-1}		RTI		
EX2	A1	A2	~~A3~~				. . .	I_{n-2}	I_{n-1}	I_n	RTI	
WB	A0	A1	A2				. . .	I_{n-3}	I_{n-2}	I_{n-1}	I_n	RTI

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.
CYCLE M: WHEN THE RTI INSTRUCTION REACHES THE DF1 STAGE, INSTRUCTION A3 IS FETCHED IN PREPARATION FOR RETURNING FROM INTERRUPT.
CYCLE M+4: RTI HAS REACHED WB STAGE, RE-ENABLING INTERRUPTS.

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` (`[--SP] = RETI`) and pop `RETI` (`RETI = [SP++]`) use the Supervisor stack.

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.



CYCLE:	1	2	3	4	5	6	7	8	9	10		m	m+1	m+2	m+3	m+4	m+5
IF 1	A9	A10		PUSH	I1	I2	I3	I4	I5	I6	. . .		A3	A4	A5	A6	A7
IF 2	A8	A9	~~A10~~		PUSH	I1	I2	I3	I4	I5	. . .			A3	A4	A5	A6
IF 3	A7	A8	~~A9~~			PUSH	I1	I2	I3	I4	. . .				A3	A4	A5
DEC	A6	A7	~~A8~~				PUSH	I1	I2	I3	. . .					A3	A4
AC	A5	A6	~~A7~~					PUSH	I1	I2	. . .	RT I					A3
DF1	A4	A5	~~A6~~						PUSH	I1	. . .	POP	RTI				
DF2	A3	A4	~~A5~~							PUSH	. . .	I_n	POP	RTI			
EX1	A2	A3	~~A4~~								. . .	I_{n-1}	I_n	POP	RTI		
EX2	A1	A2	~~A3~~								. . .	I_{n-2}	I_{n-1}	I_n	POP	RTI	
WB	A0	A1	A2								. . .	I_{n-3}	I_{n-2}	I_{n-1}	I_n	POP	RTI

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.

Figure 4-9. Nested Interrupt Handling

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 ;
```

```
[--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-
rupts, 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 suspends 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].

Logging of Nested Interrupt Requests

The System Interrupt Controller () 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.

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.

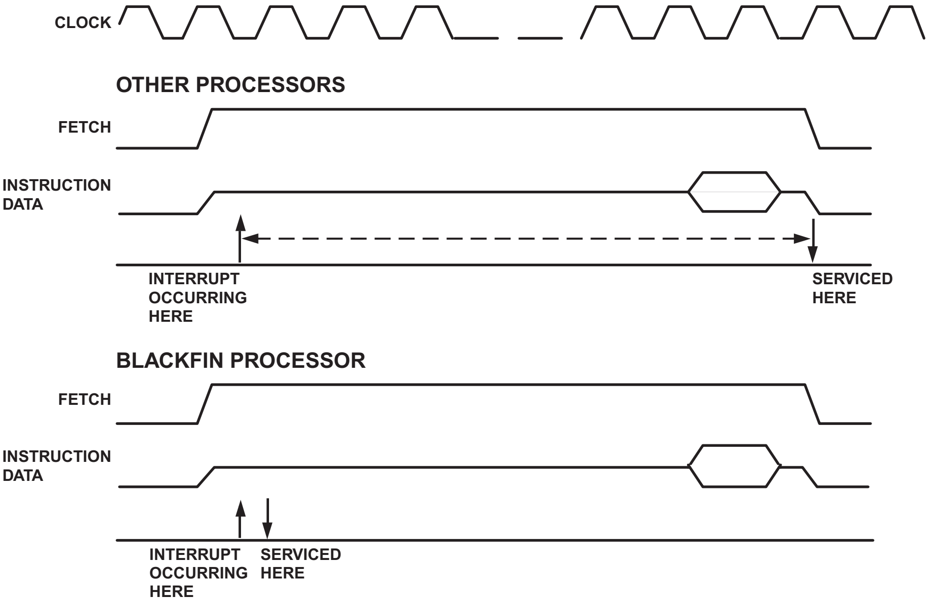


Figure 4-10. Minimizing Latency in Servicing an ISR

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.

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.

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.

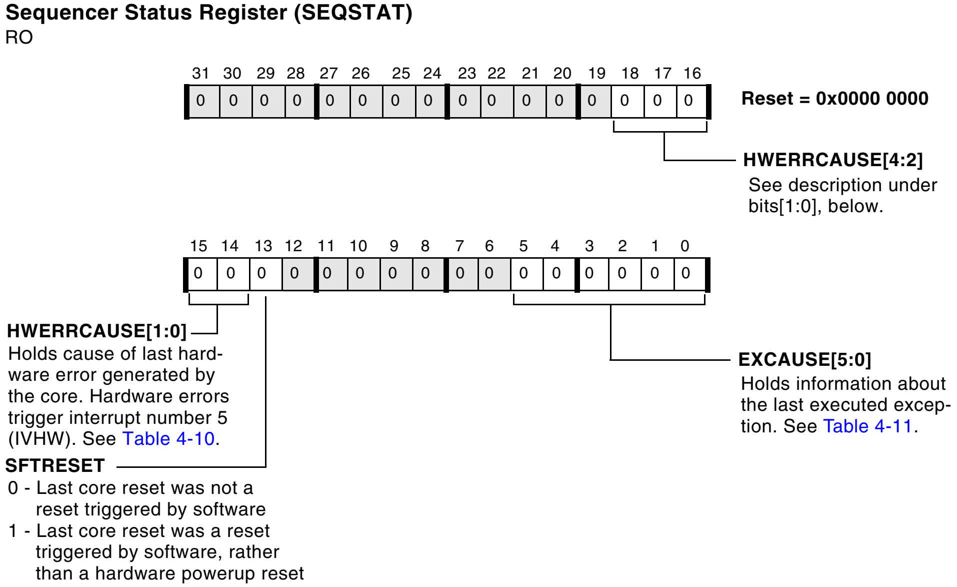

Sequencer Status Register (SEQSTAT)
RO
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
HWERRCAUSE[4:2]
See description under bits[1:0], below.
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
HWERRCAUSE[1:0]
Holds cause of last hardware error generated by the core. Hardware errors trigger interrupt number 5 (IVHW). See Table 4-10.
SFTRESET
0 - Last core reset was not a reset triggered by software
1 - Last core reset was a reset triggered by software, rather than a hardware powerup reset
EXCAUSE[5:0]
Holds information about the last executed exception. See Table 4-11.

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

Interrupt service routine can then read the cause of the error from the 5-bit HWERRCAUSE field appearing in the Sequencer Status register (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 HWERRCAUSE codes, appears in Table 4-10. The bit code for the most recent error appears in the 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.

Table 4-10. Hardware Conditions Causing Hardware Error Interrupts

Hardware Condition	HWERRCAUSE (Binary)	HWERRCAUSE (Hexadecimal)	Notes / Examples
System MMR Error	0b00010	0x02	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.
External Memory Addressing Error	0b00011	0x03	
Performance Monitor Overflow	0b10010	0x12	
RAISE 5 instruction	0b11000	0x18	Software issued a RAISE 5 instruction to invoke the Hardware Error Interrupt (IVHW).
Reserved	All other bit combinations.	All other values.	

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.

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 (SEQSTAT) 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

Exception	**EXCAUSE [5:0]**	**Type:** **(E) Error** **(S) Service** **See note 1.**	**Notes/Examples**
Force Exception instruction EXCPT with 4-bit m field	m field	S	Instruction provides 4 bits of EXCAUSE.
Single step	0x10	S	When the processor is in single step mode, every instruction generates an exception. Primarily used for debugging.
Exception caused by a trace buffer full condition	0x11	S	The processor takes this exception when the trace buffer overflows (only when enabled by the Trace Unit Control register).
Undefined instruction	0x21	E	May be used to emulate instructions that are not defined for a particular processor implementation.
Illegal instruction combination	0x22	E	See section for multi-issue rules in the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*.
Data access CPLB protection violation	0x23	E	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).

Table 4-11. Events That Cause Exceptions (Cont'd)

Exception	**EXCAUSE [5:0]**	**Type:** **(E) Error** **(S) Service** **See note 1.**	**Notes/Examples**
Data access misaligned address violation	0x24	E	Attempted misaligned data memory or data cache access.
Unrecoverable event	0x25	E	For example, an exception generated while processing a previous exception.
Data access CPLB miss	0x26	E	Used by the MMU to signal a CPLB miss on a data access.
Data access multiple CPLB hits	0x27	E	More than one CPLB entry matches data fetch address.
Exception caused by an emulation watchpoint match	0x28	E	There is a watchpoint match, and one of the EMUSW bits in the Watchpoint Instruction Address Control register (WPIACTL) is set.
Instruction fetch misaligned address violation	0x2A	E	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.)
Instruction fetch CPLB protection violation	0x2B	E	Illegal instruction fetch access (memory protection violation).
Instruction fetch CPLB miss	0x2C	E	CPLB miss on an instruction fetch.

Table 4-11. Events That Cause Exceptions (Cont'd)

Exception	EXCAUSE [5:0]	Type: (E) Error (S) Service See note 1.	Notes/Examples
Instruction fetch multiple CPLB hits	0x2D	E	More than one CPLB entry matches instruction fetch address.
Illegal use of supervisor resource	0x2E	E	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.

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

Priority	Exception	EXCAUSE
1	Unrecoverable Event	0x25
2	I-Fetch Multiple CPLB Hits	0x2D
3	I-Fetch Misaligned Access	0x2A
4	I-Fetch Protection Violation	0x2B
5	I-Fetch CPLB Miss	0x2C
6	I-Fetch Access Exception	0x29
7	Watchpoint Match	0x28
8	Undefined Instruction	0x21

Table 4-12. Exceptions by Descending Priority (Cont'd)

Priority	Exception	EXCAUSE
9	Illegal Combination	0x22
10	Illegal Use of Protected Resource	0x2E
11	DAG0 Multiple CPLB Hits	0x27
12	DAG0 Misaligned Access	0x24
13	DAG0 Protection Violation	0x23
14	DAG0 CPLB Miss	0x26
15	DAG1 Multiple CPLB Hits	0x27
16	DAG1 Misaligned Access	0x24
17	DAG1 Protection Violation	0x23
18	DAG1 CPLB Miss	0x26
19	EXCPT Instruction	m field
20	Single Step	0x10
21	Trace Buffer	0x11

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.

- The EXCAUSE field in SEQSTAT is updated with an unrecoverable event code.
- The address of the offending instruction is saved in RETX. Note if the processor were executing, for example, the NMI handler, the RETN register would not have been updated; the excepting instruction address is always stored in RETX.

To determine whether an exception occurred while an exception handler was executing, check SEQSTAT at the end of the exception handler for the code indicating an "unrecoverable event" (EXCAUSE = 0x25). If an unrecoverable event occurred, register 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 PUSH and POP instructions for exceptions, NMIs, and emulation events do not affect the interrupt system.

Note, however, the return instructions for exceptions (RTX, RTN, and RTE) do clear the Least Significant Bit (LSB) currently set in 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

```
/* Determine exception cause by examining EXCAUSE field in
SEQSTAT (first save contents of R0, P0, P1 and ASTAT in Supervi-
sor 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 ;
_EVTABLE:
.byte2 addr_event1;
.byte2 addr_event2;
...
.byte2 addr_eventN;
```

```
/* 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. */
```

5 ADDRESS ARITHMETIC UNIT

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.

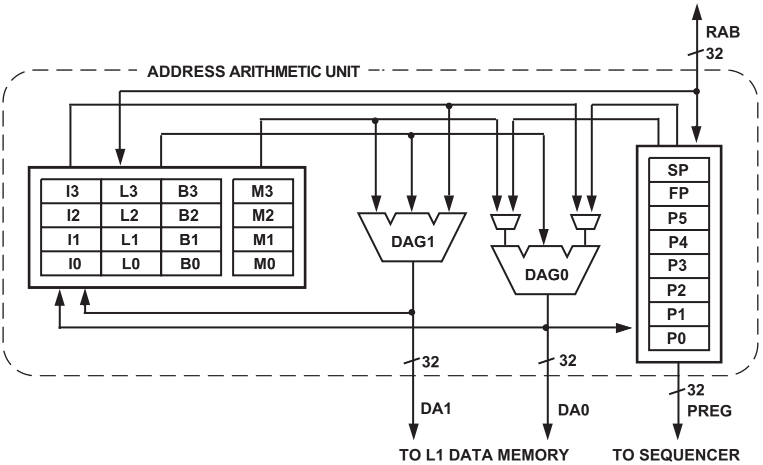


Figure 5-1. AAU Block Diagram

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

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 ++ 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.

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.

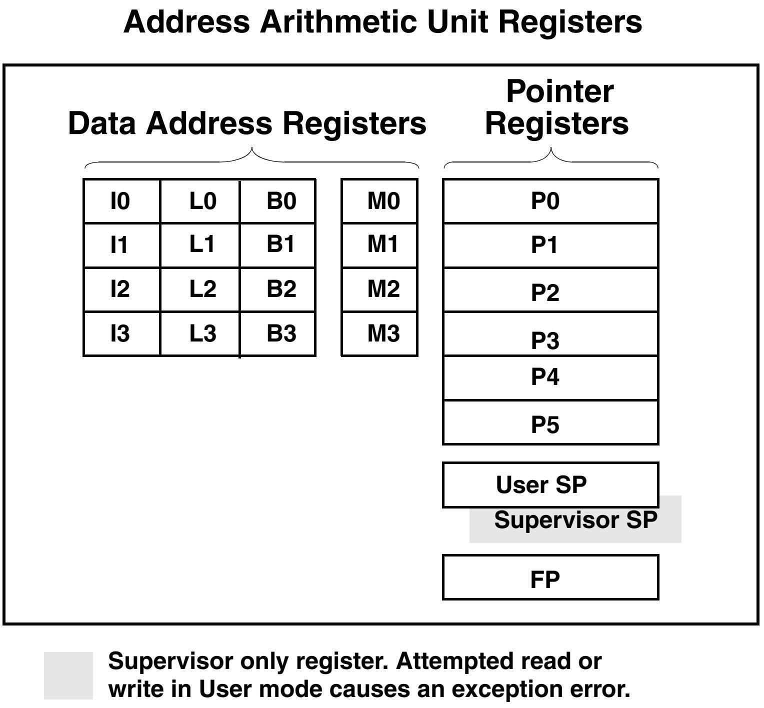

Address Arithmetic Unit Registers
Data Address Registers
Pointer Registers
I0
L0
B0
M0
I1
L1
B1
M1
I2
L2
B2
M2
I3
L3
B3
M3
P0
P1
P2
P3
P4
P5
User SP
Supervisor SP
FP
Supervisor only register. Attempted read or write in User mode causes an exception error.

Figure 5-2. 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++ ] (Z) ;
```

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.

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)

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 `R0 = [ SP++ ] ;`) implicitly uses the `USP` as the effective address.
- In Supervisor mode, the same reference to `SP` (for example, `R0 = [ 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, `R0 = USP ;`) moves the current User Stack Pointer into `R0`. 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

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.

For example:

```
R0 = [ I2 ];
```

loads a 32-bit value from an address pointed to by I2 and stores it in the destination register R0.

```
R0.H = W [ I2 ];
```

loads a 16-bit value from an address pointed to by I2 and stores it in the 16-bit destination register R0.H.

```
[ P1 ] = R0 ;
```

is an example of a 32-bit store operation.

Pointer registers can be used for 8-bit loads and stores.

For example:

```
B [ P1++ ] = R0 ;
```

stores the 8-bit value from the R0 register in the address pointed to by the P1 register, then increments the 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 P1 points to a memory location that contains a value of 0x8080.

```
R0 = W[P1] (Z) ;   /*  R0 = 0x0000 8080 */

R1 = W[P1] (X) ;   /*  R1 = 0xFFFF 8080 */

R2 = B[P1] (Z) ;   /*  R2 = 0x0000 0080 */

R3 = B[P1] (X) ;   /*  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:

```
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.

For example:

```
R0 = [ I2-- ] ;
```

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:

```
R5 = [ P1++P2 ] ;
```

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.

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.

- 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.

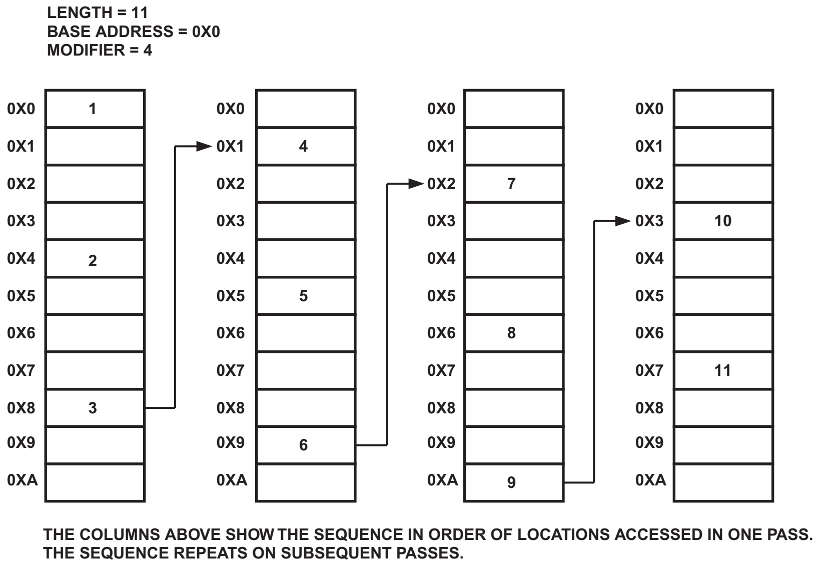

LENGTH = 11
BASE ADDRESS = 0X0
MODIFIER = 4
0X0
0X1
0X2
0X3
0X4
0X5
0X6
0X7
0X8
0X9
0XA
1
2
3
4
5
6
7
8
9
10
11
THE COLUMNS ABOVE SHOW THE SEQUENCE IN ORDER OF LOCATIONS ACCESSED IN ONE PASS.
THE SEQUENCE REPEATS ON SUBSEQUENT PASSES.

Figure 5-3. Circular Data Buffers

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.

In equation form, these post-modify and wraparound operations work as follows, shown for “I+M” operations.

- If M is positive:

  $I_{new} = I_{old} + M$
  if $I_{old} + M <$ buffer base + length (end of buffer)

  $I_{new} = I_{old} + M - L$
  if $I_{old} + M \geq$ buffer base + length (end of buffer)

- If M is negative:

  $I_{new} = I_{old} + M$
  if $I_{old} + M \geq$ buffer base (start of buffer)

  $I_{new} = I_{old} + M + L$
  if $I_{old} + M <$ 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, I1 += M2 ;

This instruction adds M2 to I1 and updates I1 with the new value.

Memory Address Alignment

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 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 b#00.
- 16-bit word load/stores are accessed on two-byte boundaries, meaning the least significant bit of the address must be 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

Addressing Mode	Types of Transfers Supported	Transfer Sizes
Auto-increment Auto-decrement Indirect Indexed	To and from Data Registers	LOADS: 32-bit word 16-bit, zero extended half word 16-bit, sign extended half word 8-bit, zero extended byte 8-bit, sign extended byte STORES: 32-bit word 16-bit half word 8-bit byte
	To and from Pointer Registers	LOAD: 32-bit word STORE: 32-bit word
Post-increment	To and from Data Registers	LOADS: 32-bit word 16-bit half word to Data Register high half 16-bit half word to Data Register low half 16-bit, zero extended half word 16-bit, sign extended half word STORES: 32-bit word 16-bit half word from Data Register high half 16-bit half word from Data Register low half

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.

Table 5-2 summarizes the addressing modes. In the table, an asterisk (*) indicates the processor supports the addressing mode.

Table 5-2. Addressing Modes

	32-bit word	16-bit half-word	8-bit byte	Sign/zero extend	Data Register	Pointer register	Data Register Half
P Auto-inc [P0++]	*	*	*	*	*	*	
P Auto-dec [P0--]	*	*	*	*	*	*	
P Indirect [P0]	*	*	*	*	*	*	*
P Indexed [P0+im]	*	*	*	*	*	*	
FP indexed [FP+im]	*				*	*	
P Post-inc [P0++P1]	*	*		*	*		*
I Auto-inc [I0++]	*	*			*		*
I Auto-dec [I0--]	*	*			*		*
I Indirect [I0]	*	*			*		*
I Post-inc [I0++M0]	*				*		

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 instructions do not affect the ASTAT Status flags.

Table 5-3. AAU Instruction Summary

Instruction
Preg = [Preg] ;
Preg = [Preg ++] ;
Preg = [Preg --] ;
Preg = [Preg + uimm6m4] ;
Preg = [Preg + uimm17m4] ;
Preg = [Preg – uimm17m4] ;
Preg = [FP – uimm7m4] ;
Dreg = [Preg] ;
Dreg = [Preg ++] ;
Dreg = [Preg --] ;
Dreg = [Preg + uimm6m4] ;
Dreg = [Preg + uimm17m4] ;
Dreg = [Preg – uimm17m4] ;
Dreg = [Preg ++ Preg] ;
Dreg = [FP – uimm7m4] ;
Dreg = [Ireg] ;
Dreg = [Ireg ++] ;
Dreg = [Ireg --] ;
Dreg = [Ireg ++ Mreg] ;
Dreg =W [Preg] (Z) ;
Dreg =W [Preg ++] (Z) ;
Dreg =W [Preg --] (Z) ;
Dreg =W [Preg + uimm5m2] (Z) ;

Table 5-3. AAU Instruction Summary (Cont'd)

Instruction
Dreg =W [Preg + uimm16m2] (Z) ;
Dreg =W [Preg – uimm16m2] (Z) ;
Dreg =W [Preg ++ Preg] (Z) ;
Dreg = W [Preg] (X) ;
Dreg = W [Preg ++] (X) ;
Dreg = W [Preg --] (X) ;
Dreg =W [Preg + uimm5m2] (X) ;
Dreg =W [Preg + uimm16m2] (X) ;
Dreg =W [Preg – uimm16m2] (X) ;
Dreg =W [Preg ++ Preg] (X) ;
Dreg_hi = W [Ireg] ;
Dreg_hi = W [Ireg ++] ;
Dreg_hi = W [Ireg --] ;
Dreg_hi = W [Preg] ;
Dreg_hi = W [Preg ++ Preg] ;
Dreg_lo = W [Ireg] ;
Dreg_lo = W [Ireg ++] ;
Dreg_lo = W [Ireg --] ;
Dreg_lo = W [Preg] ;
Dreg_lo = W [Preg ++ Preg] ;
Dreg = B [Preg] (Z) ;
Dreg = B [Preg ++] (Z) ;
Dreg = B [Preg --] (Z) ;
Dreg = B [Preg + uimm15] (Z) ;

Table 5-3. AAU Instruction Summary (Cont'd)

Instruction
Dreg = B [Preg – uimm15] (Z) ;
Dreg = B [Preg] (X) ;
Dreg = B [Preg ++] (X) ;
Dreg = B [Preg --] (X) ;
Dreg = B [Preg + uimm15] (X) ;
Dreg = B [Preg – uimm15] (X) ;
[Preg] = Preg ;
[Preg ++] = Preg ;
[Preg --] = Preg ;
[Preg + uimm6m4] = Preg ;
[Preg + uimm17m4] = Preg ;
[Preg – uimm17m4] = Preg ;
[FP – uimm7m4] = Preg ;
[Preg] = Dreg ;
[Preg ++] = Dreg ;
[Preg --] = Dreg ;
[Preg + uimm6m4] = Dreg ;
[Preg + uimm17m4] = Dreg ;
[Preg – uimm17m4] = Dreg ;
[Preg ++ Preg] = Dreg ;
[FP – uimm7m4] = Dreg ;
[Ireg] = Dreg ;
[Ireg ++] = Dreg ;
[Ireg --] = Dreg ;

Table 5-3. AAU Instruction Summary (Cont'd)

Instruction
[Ireg ++ Mreg] = Dreg ;
W [Ireg] = Dreg_hi ;
W [Ireg ++] = Dreg_hi ;
W [Ireg --] = Dreg_hi ;
W [Preg] = Dreg_hi ;
W [Preg ++ Preg] = Dreg_hi ;
W [Ireg] = Dreg_lo ;
W [Ireg ++] = Dreg_lo ;
W [Ireg --] = Dreg_lo ;
W [Preg] = Dreg_lo ;
W [Preg] = Dreg ;
W [Preg ++] = Dreg ;
W [Preg --] = Dreg ;
W [Preg + uimm5m2] = Dreg ;
W [Preg + uimm16m2] = Dreg ;
W [Preg – uimm16m2] = Dreg ;
W [Preg ++ Preg] = Dreg_lo ;
B [Preg] = Dreg ;
B [Preg ++] = Dreg ;
B [Preg --] = Dreg ;
B [Preg + uimm15] = Dreg ;
B [Preg – uimm15] = Dreg ;
Preg = imm7 (X) ;
Preg = imm16 (X) ;

Table 5-3. AAU Instruction Summary (Cont'd)

Instruction
Preg += Preg (BREV) ;
Ireg += Mreg (BREV) ;
Preg = Preg << 2 ;
Preg = Preg >> 2 ;
Preg = Preg >> 1 ;
Preg = Preg + Preg << 1 ;
Preg = Preg + Preg << 2 ;
Preg –= Preg ;
Ireg –= Mreg ;

ⓘ 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."

6 MEMORY

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 `CCLK` clock domain, they take multiple `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 *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.

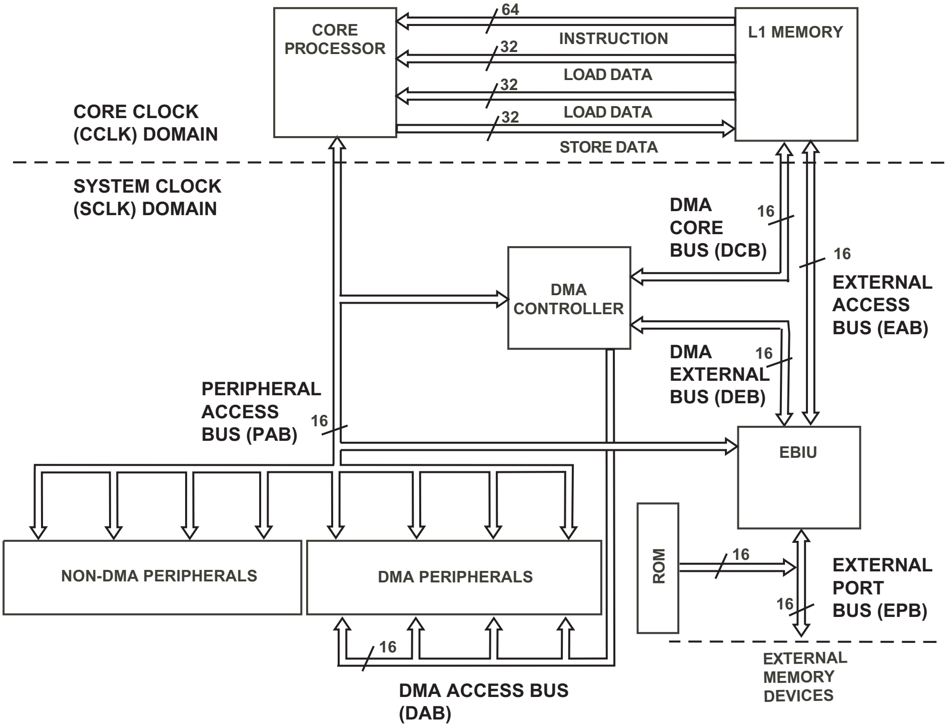


Figure 6-1. Processor Memory Architecture

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 (`CCLK`) for critical DSP algorithms and fast context switching

- Instruction and data cache options for microcontroller code, excellent High Level Language (HLL) support, and ease of programming cache control instructions, such as PREFETCH and FLUSH
- Memory protection

The L1 memories operate at the core clock frequency (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.

The scratchpad data SRAM, like the other L1 blocks, operates at core clock frequency (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 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

ⓘ 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_LRUPRIO 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_LRUPRIO 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_LRUPRIO 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.

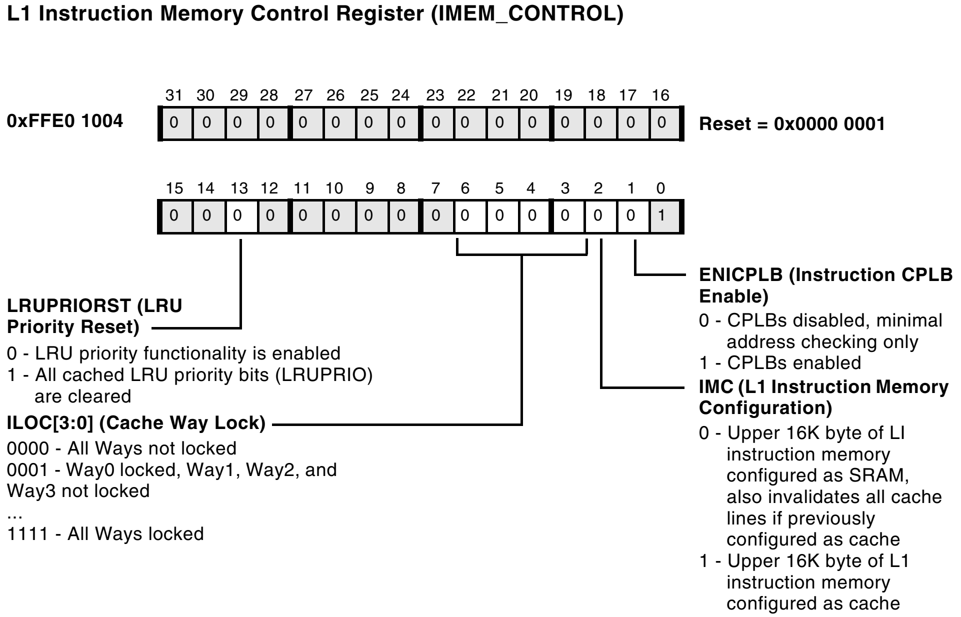

L1 Instruction Memory Control Register (IMEM_CONTROL)
0xFFE0 1004
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0001
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
LRUPRIORST (LRU Priority Reset)
0 - LRU priority functionality is enabled
1 - All cached LRU priority bits (LRUPRIO) are cleared
ILOC[3:0] (Cache Way Lock)
0000 - All Ways not locked
0001 - Way0 locked, Way1, Way2, and Way3 not locked
...
1111 - All Ways locked
ENICPLB (Instruction CPLB Enable)
0 - CPLBs disabled, minimal address checking only
1 - CPLBs enabled
IMC (L1 Instruction Memory Configuration)
0 - Upper 16K byte of LI instruction memory configured as SRAM, also invalidates all cache lines if previously configured as cache
1 - Upper 16K byte of L1 instruction memory configured as cache

Figure 6-2. L1 Instruction Memory Control Register

L1 Instruction SRAM

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.

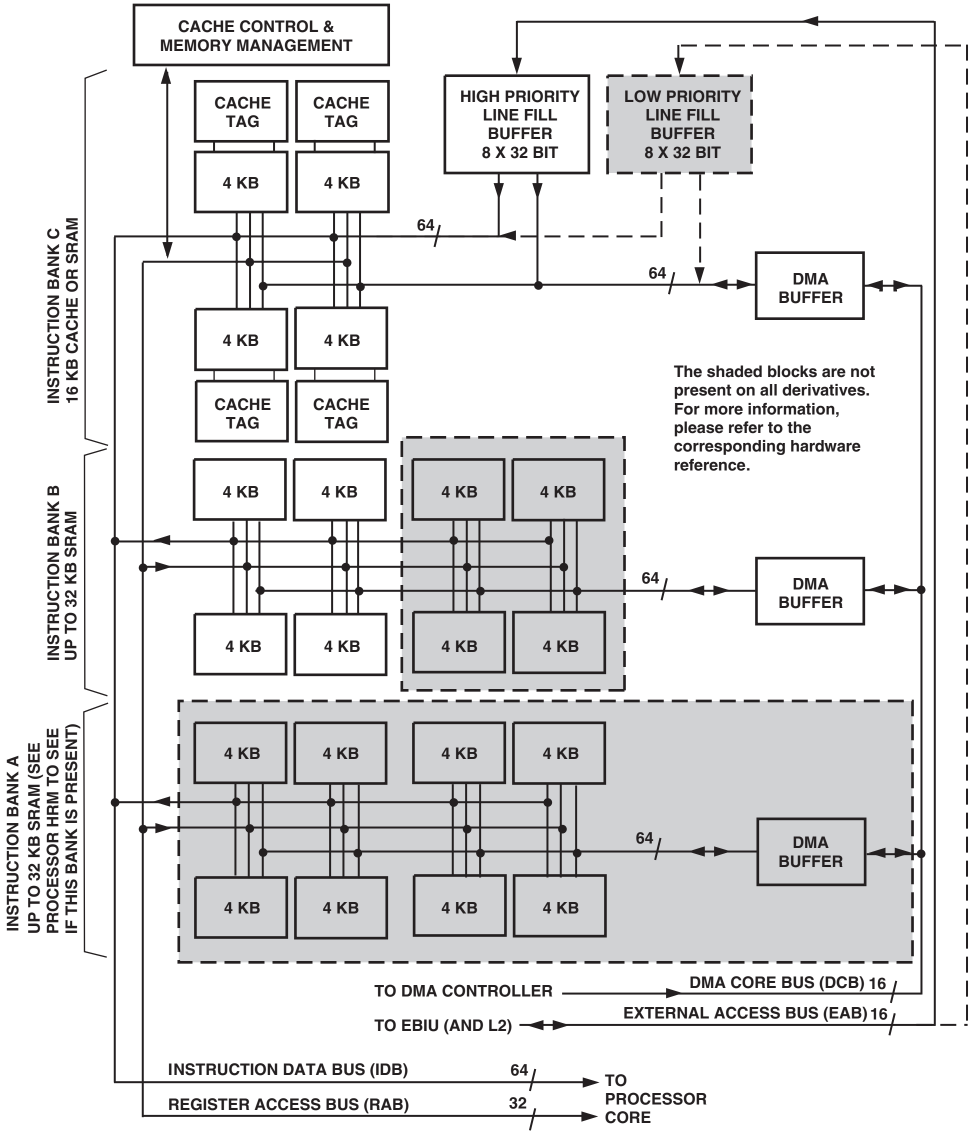

CACHE CONTROL &
MEMORY MANAGEMENT
CACHE TAG
CACHE TAG
4 KB
4 KB
HIGH PRIORITY LINE FILL BUFFER 8 X 32 BIT
LOW PRIORITY LINE FILL BUFFER 8 X 32 BIT
64
64
DMA BUFFER
INSTRUCTION BANK C
16 KB CACHE OR SRAM
4 KB
4 KB
CACHE TAG
CACHE TAG
The shaded blocks are not present on all derivatives. For more information, please refer to the corresponding hardware reference.
INSTRUCTION BANK B
UP TO 32 KB SRAM
4 KB
4 KB
4 KB
4 KB
64
DMA BUFFER
4 KB
4 KB
4 KB
4 KB
INSTRUCTION BANK A
UP TO 32 KB SRAM (SEE PROCESSOR HRM TO SEE IF THIS BANK IS PRESENT)
4 KB
4 KB
4 KB
4 KB
64
DMA BUFFER
4 KB
4 KB
4 KB
4 KB
TO DMA CONTROLLER
DMA CORE BUS (DCB) 16
TO EBIU (AND L2)
EXTERNAL ACCESS BUS (EAB) 16
INSTRUCTION DATA BUS (IDB)
64
REGISTER ACCESS BUS (RAB)
32
TO PROCESSOR CORE

Figure 6-3. L1 Instruction Memory Bank Architecture

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.

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.

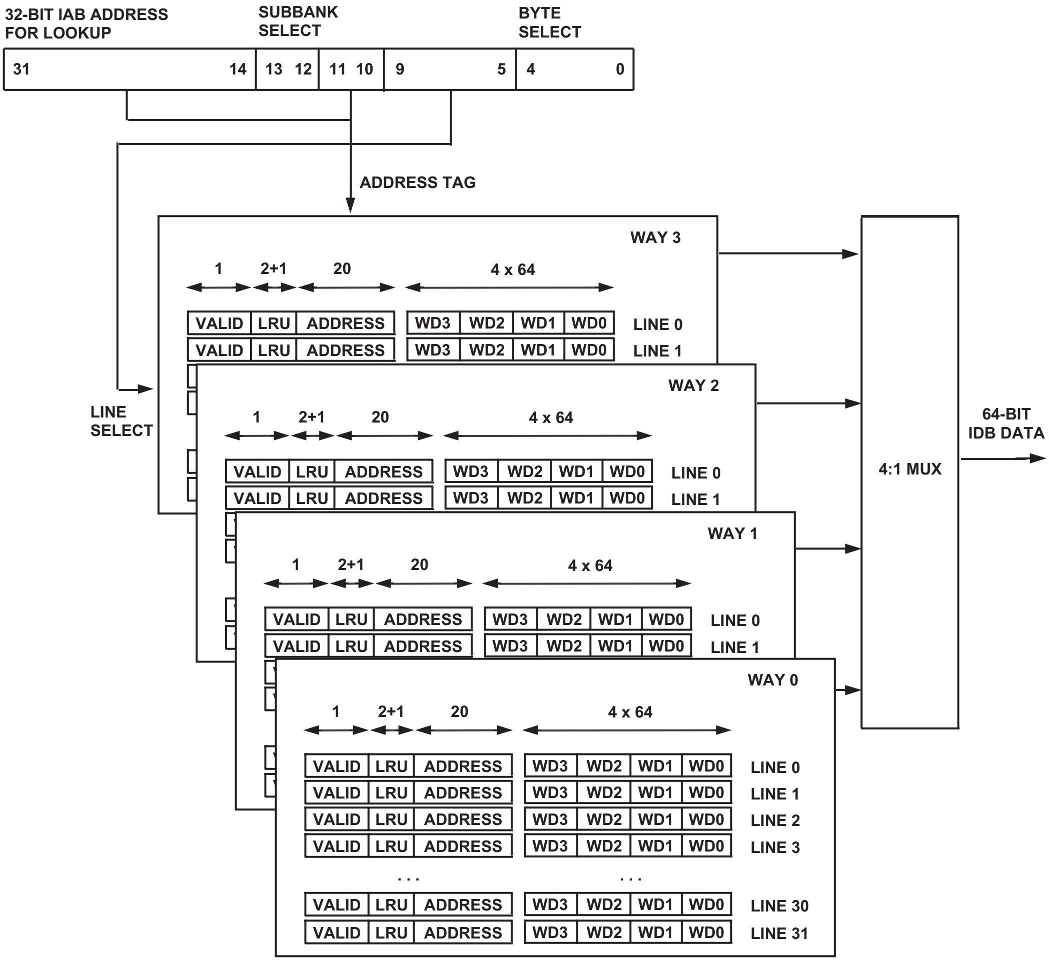


Figure 6-4. Instruction Cache Organization Per Subbank

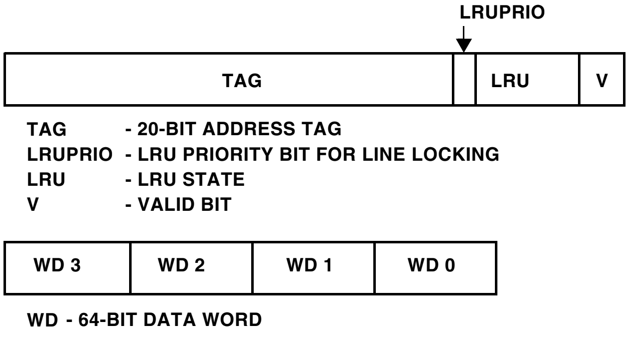


Figure 6-5. Cache Line – Tag and Data Portions

Cache Hits and Misses

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.

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

Target Word	Fetching Order for Next Three Words
WD0	WD0, WD1, WD2, WD3
WD1	WD1, WD2, WD3, WD0
WD2	WD2, WD3, WD0, WD1
WD3	WD3, WD0, WD1, WD2

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.

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

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 `LRUPRIRST` 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.

- 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:

```
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.

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.

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]).

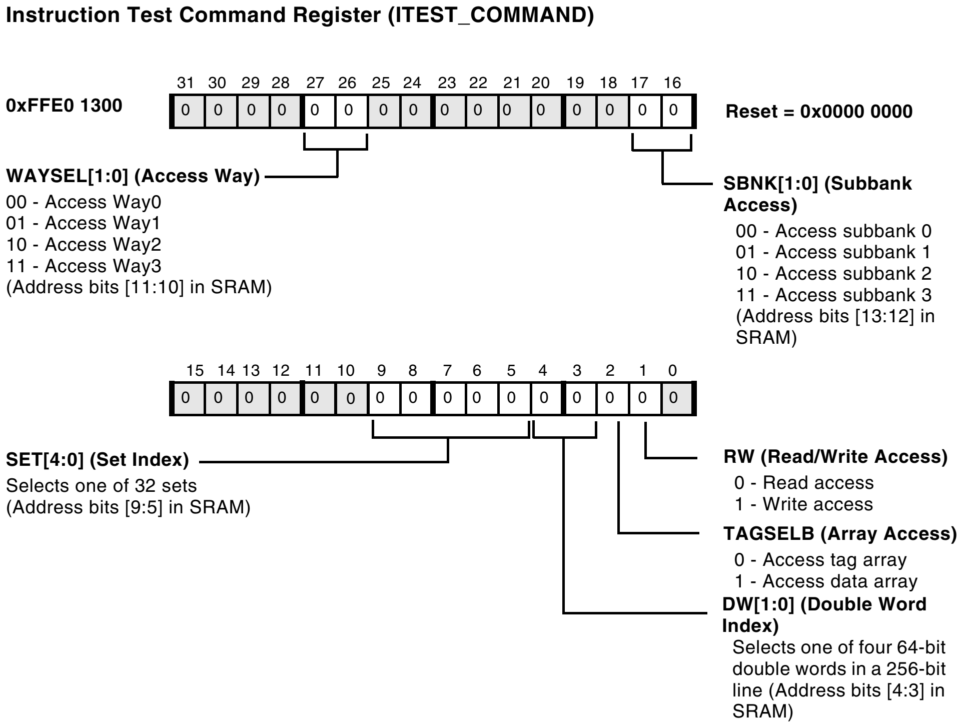
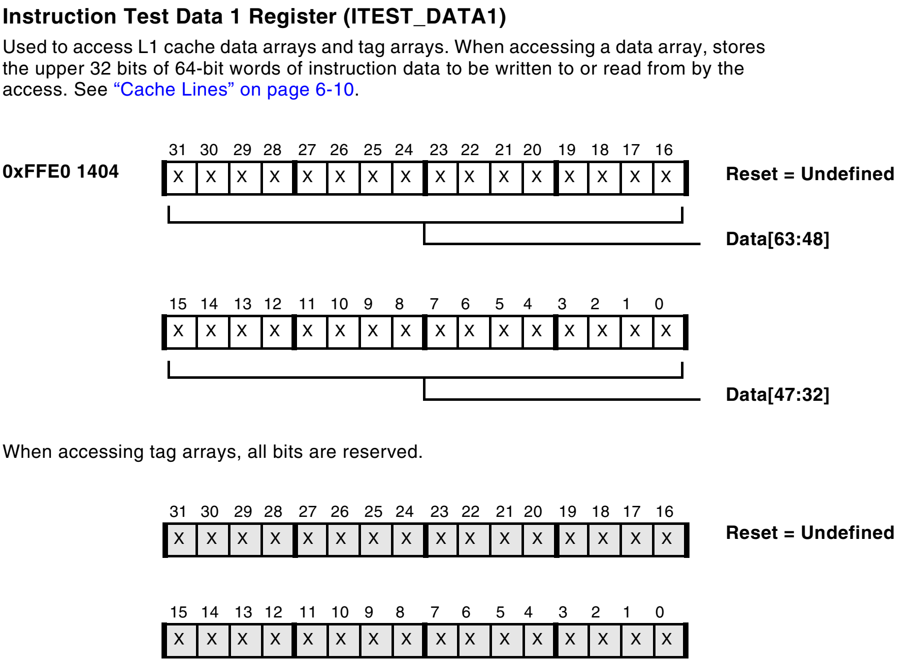


Figure 6-6. Instruction Test Command Register

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)

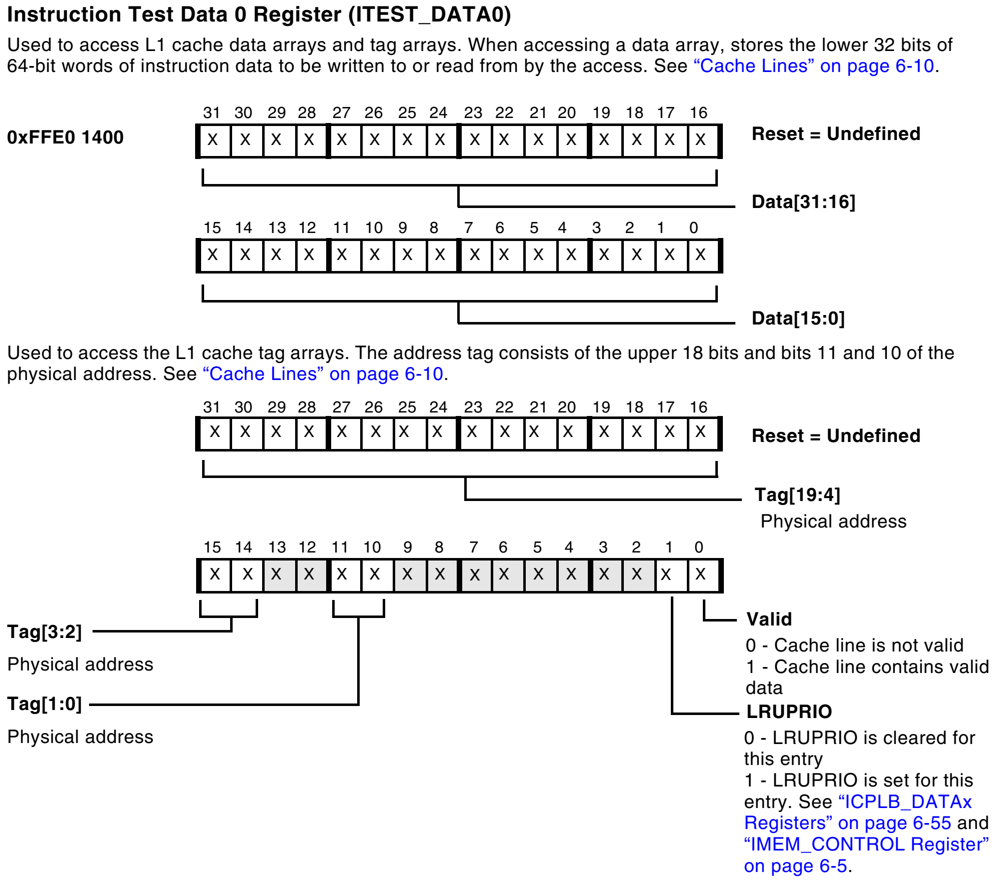
Used to access L1 cache data arrays and tag arrays. When accessing a data array, stores the upper 32 bits of 64-bit words of instruction data to be written to or read from by the access. See "Cache Lines" on page 6-10.

When accessing tag arrays, all bits are reserved.

Figure 6-7. Instruction Test Data 1 Register

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.

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.

0xFFE0 1400
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
Data[31:16]
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Data[15:0]
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
Tag[19:4]
Physical address
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Tag[3:2]
Physical address
Tag[1:0]
Physical address
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:

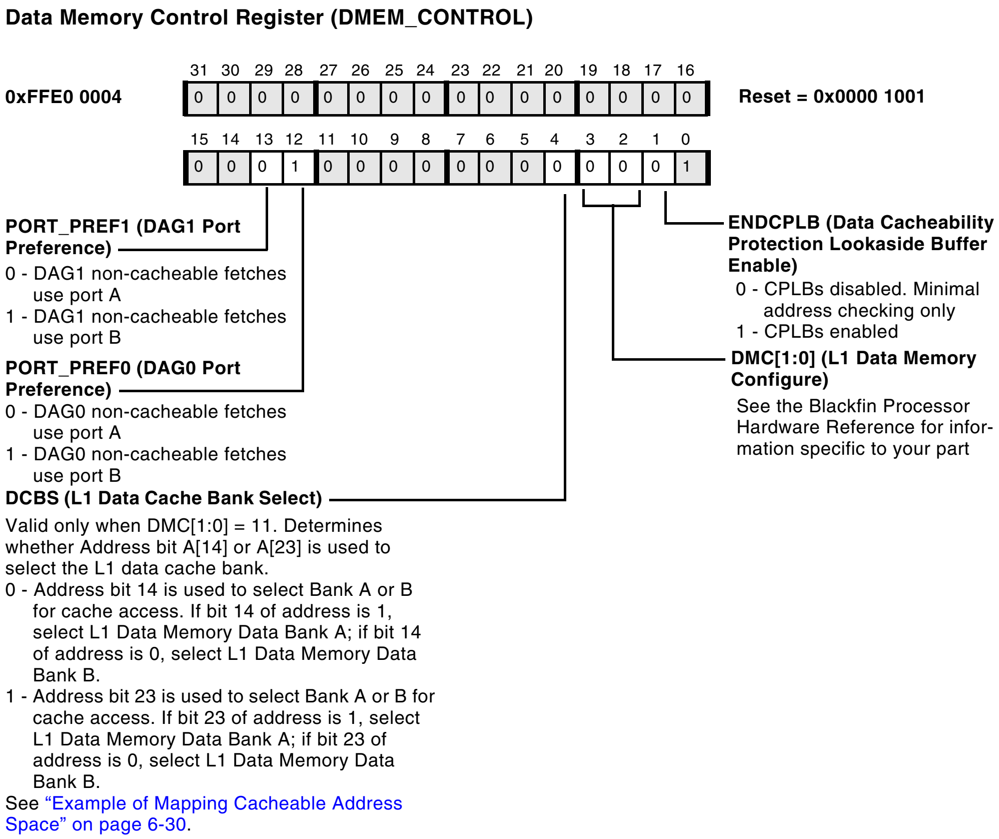
- Two 32-bit data loads
- One pipelined 32-bit data store
- One DMA I/O, up to 64 bits
- One 64-bit cache fill/victim access

ⓘ 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.



PORT_PREF1 (DAG1 Port Preference)

0 - DAG1 non-cacheable fetches use port A

1 - DAG1 non-cacheable fetches use port B

PORT_PREF0 (DAG0 Port Preference)

0 - DAG0 non-cacheable fetches use port A

1 - DAG0 non-cacheable fetches use port B

DCBS (L1 Data Cache Bank Select)

Valid only when DMC[1:0] = 11. Determines whether Address bit A[14] or A[23] is used to select the L1 data cache bank.

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.

ENDCPLB (Data Cacheability Protection Lookaside Buffer Enable)

0 - CPLBs disabled. Minimal address checking only

1 - CPLBs enabled

DMC[1:0] (L1 Data Memory Configure)

See the Blackfin Processor Hardware Reference for information specific to your part

Figure 6-9. L1 Data Memory Control Register

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 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 ENDCPLB bit) and CPLB descriptors (registers DCPLB_DATAx and DCPLB_ADDRx) must specify chosen memory pages as cache-enabled.

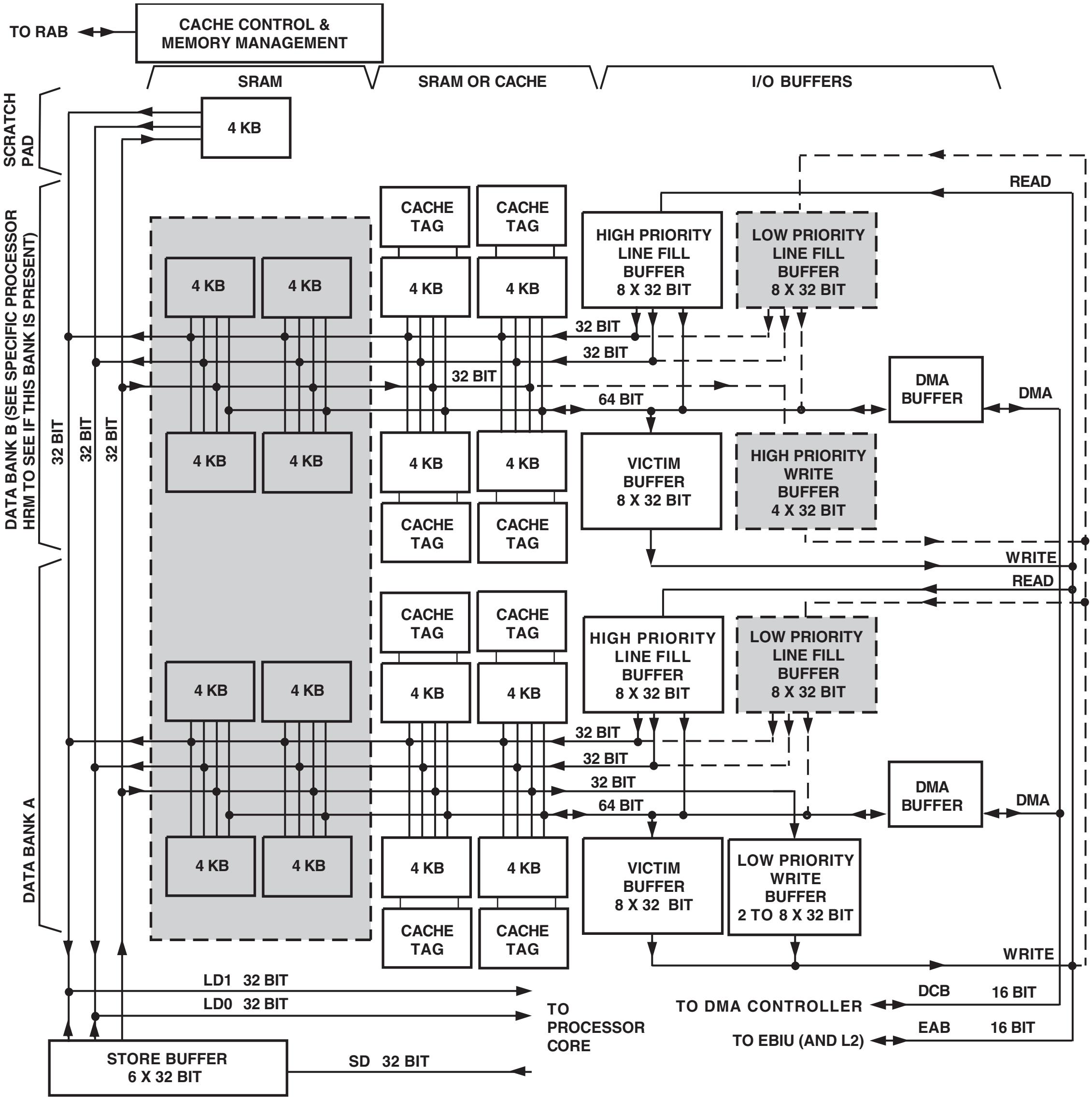
By default after reset, cache and CPLB address checking is disabled.

> 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.



The shaded blocks are not present on all derivatives. For more information, please refer to the corresponding processor hardware reference.

Figure 6-10. L1 Data Memory Architecture

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.

If cache is enabled (controlled by bits `DMC[1:0]` in the `DMEM_CONTROL` register), data CPLBs should also be enabled (controlled by `ENDCPLB` bit in the `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 `DCBS` bit in the `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 `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.

- If `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 `DCBS = 0` or `DCBS = 1` is:

- If `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).

- 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 `DCBS = 1`.

The `DCBS` selection can be changed dynamically; however, to ensure that no data is lost, first flush and invalidate the entire cache.

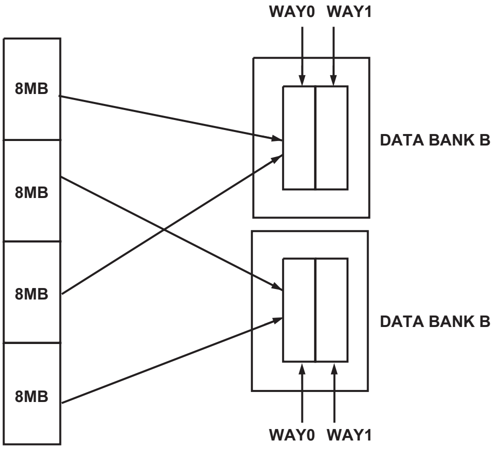


Figure 6-11. Data Cache Mapping When DCBS = 1

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 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.

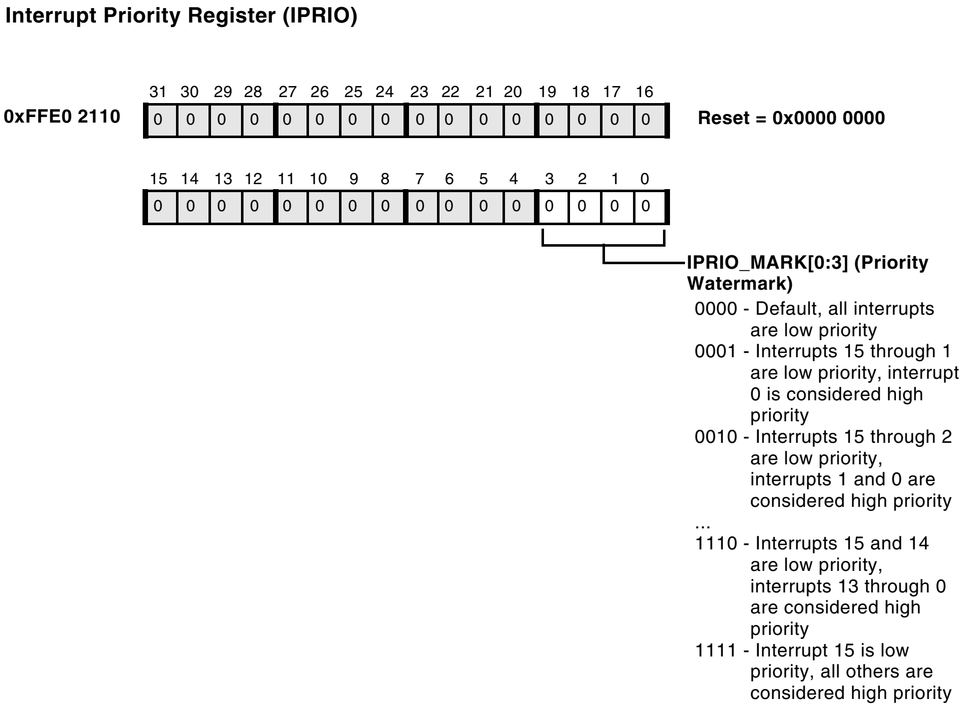
- A write-back cache does not write to external memory until the line is replaced by a load operation that needs the line.

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.



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

Figure 6-12. Interrupt Priority Register

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 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]).

ⓘ The Data/Instruction Access bit allows direct access via the DTEST_COMMAND MMR to L1 instruction SRAM.

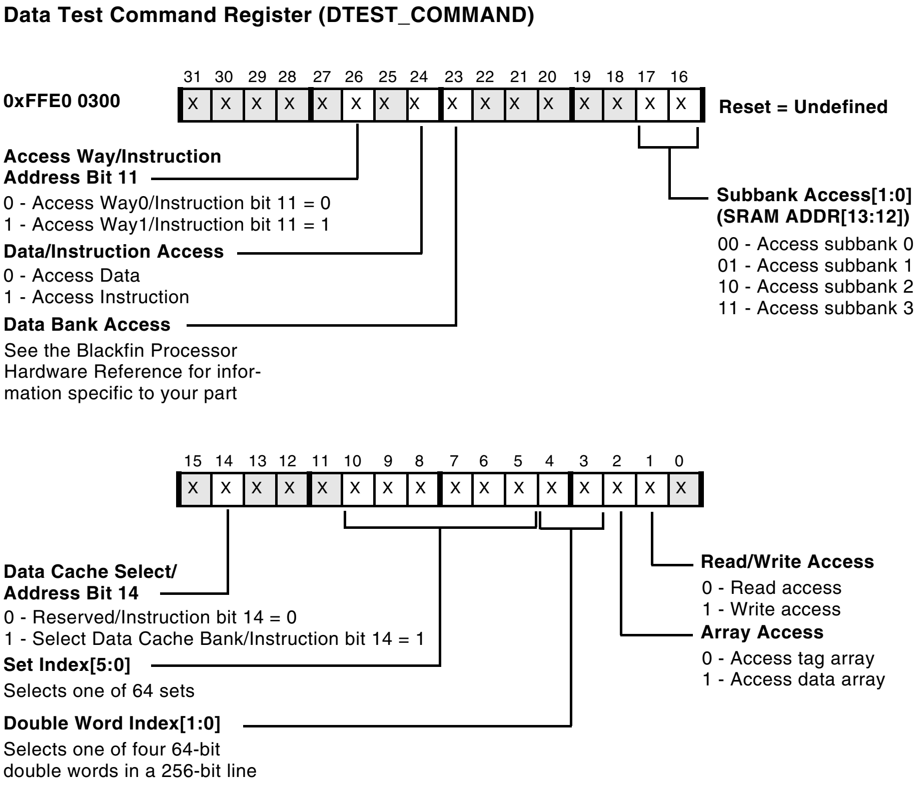

Data Test Command Register (DTEST_COMMAND)
0xFFE0 0300
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
Access Way/Instruction Address Bit 11
0 - Access Way0/Instruction bit 11 = 0
1 - Access Way1/Instruction bit 11 = 1
Data/Instruction Access
0 - Access Data
1 - Access Instruction
Data Bank Access
See the Blackfin Processor Hardware Reference for information specific to your part
Subbank Access[1:0] (SRAM ADDR[13:12])
00 - Access subbank 0
01 - Access subbank 1
10 - Access subbank 2
11 - Access subbank 3
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Data Cache Select/ Address Bit 14
0 - Reserved/Instruction bit 14 = 0
1 - Select Data Cache Bank/Instruction bit 14 = 1
Set Index[5:0]
Selects one of 64 sets
Double Word Index[1:0]
Selects one of four 64-bit double words in a 256-bit line
Read/Write Access
0 - Read access
1 - Write access
Array Access
0 - Access tag array
1 - Access data array

Figure 6-13. Data Test Command Register

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.

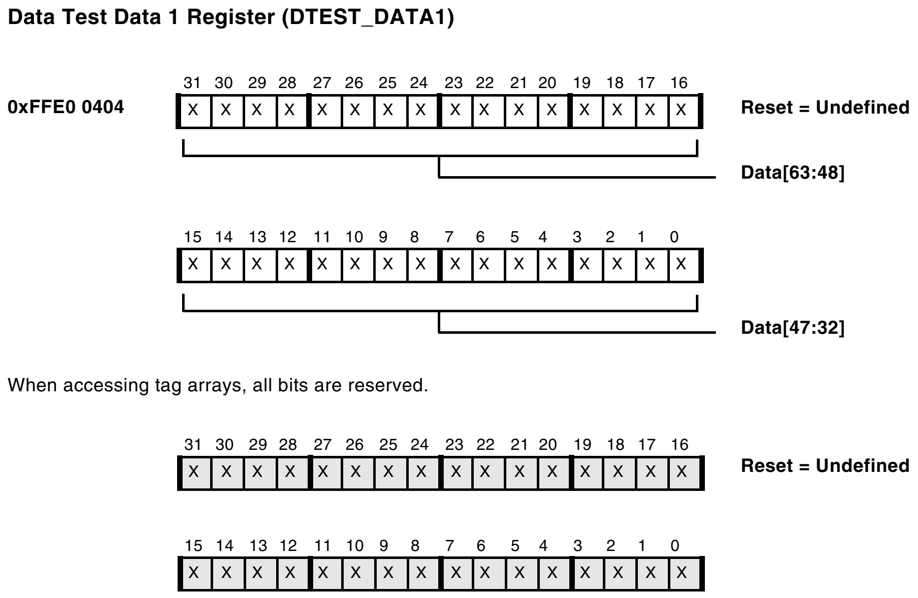
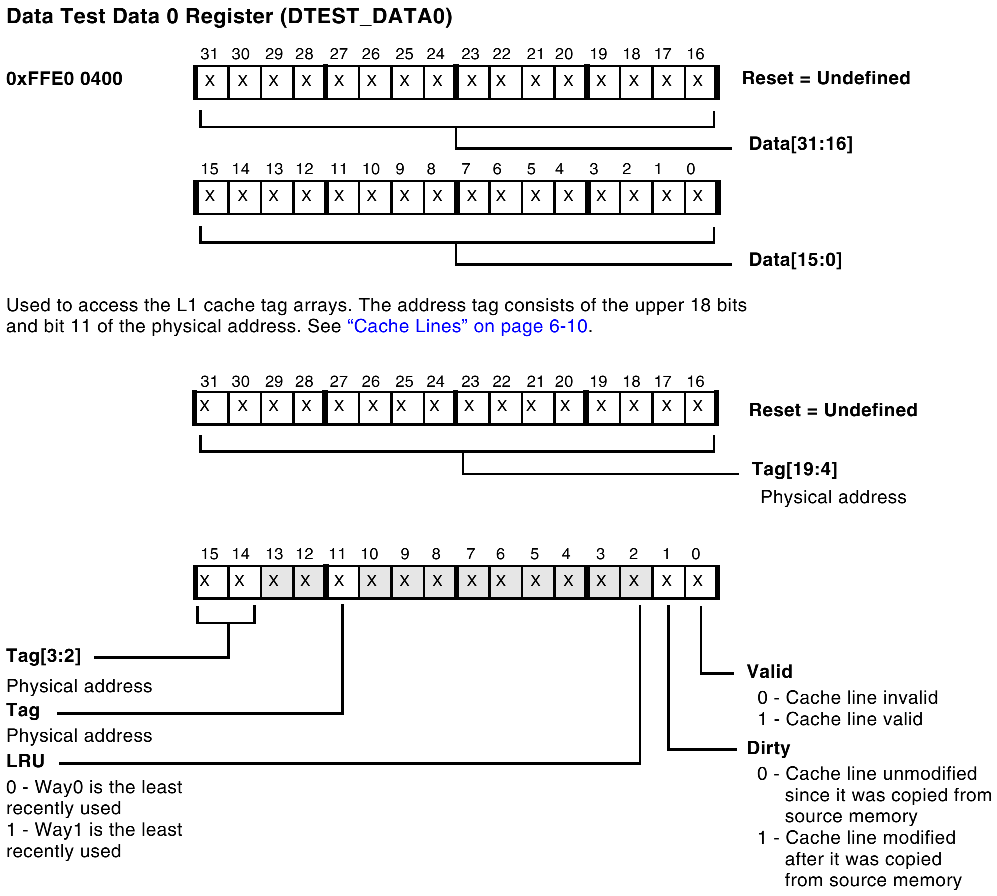


Figure 6-14. Data Test Data 1 Register

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.


Data Test Data 0 Register (DTEST_DATA0)
0xFFE0 0400
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
Data[31:16]
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Data[15:0]
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
Tag[19:4]
Physical address
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Tag[3:2]
Physical address
Tag
Physical address
LRU
0 - Way0 is the least recently used
1 - Way1 is the least recently used
Valid
0 - Cache line invalid
1 - Cache line valid
Dirty
0 - Cache line unmodified since it was copied from source memory
1 - Cache line modified after it was copied from source memory

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.

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.

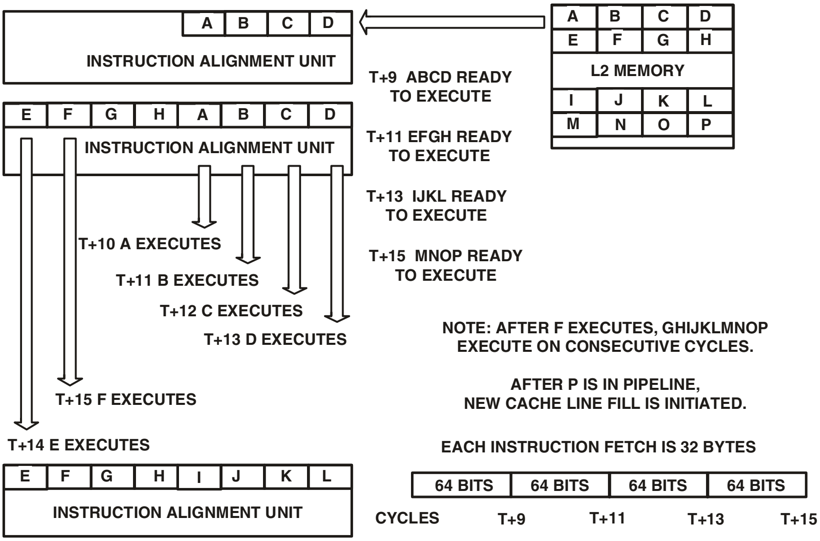

A B C D
INSTRUCTION ALIGNMENT UNIT
A B C D
E F G H
L2 MEMORY
I J K L
M N O P
T+9 ABCD READY TO EXECUTE
T+11 EFGH READY TO EXECUTE
T+13 IJKL READY TO EXECUTE
T+15 MNOP READY TO EXECUTE
E F G H A B C D
INSTRUCTION ALIGNMENT UNIT
T+10 A EXECUTES
T+11 B EXECUTES
T+12 C EXECUTES
T+13 D EXECUTES
T+15 F EXECUTES
T+14 E EXECUTES
NOTE: AFTER F EXECUTES, GHIJKLMNOP EXECUTE ON CONSECUTIVE CYCLES.
AFTER P IS IN PIPELINE, NEW CACHE LINE FILL IS INITIATED.
EACH INSTRUCTION FETCH IS 32 BYTES
E F G H I J K L
INSTRUCTION ALIGNMENT UNIT
64 BITS
64 BITS
64 BITS
64 BITS
CYCLES
T+9
T+11
T+13
T+15

Figure 6-16. L2 Latency With Cache On

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.

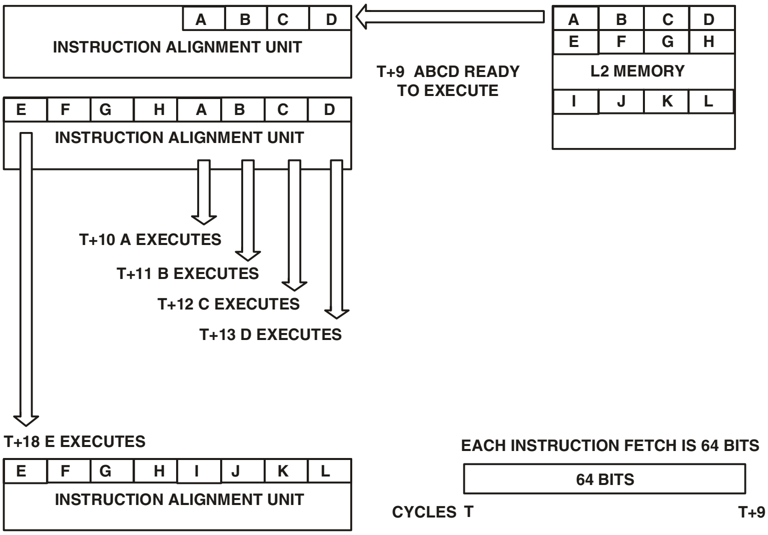

A
B
C
D
INSTRUCTION ALIGNMENT UNIT
T+9 ABCD READY TO EXECUTE
A
B
C
D
E
F
G
H
L2 MEMORY
I
J
K
L
E
F
G
H
A
B
C
D
INSTRUCTION ALIGNMENT UNIT
T+10 A EXECUTES
T+11 B EXECUTES
T+12 C EXECUTES
T+13 D EXECUTES
T+18 E EXECUTES
EACH INSTRUCTION FETCH IS 64 BITS
64 BITS
E
F
G
H
I
J
K
L
INSTRUCTION ALIGNMENT UNIT
CYCLES T
T+9

Figure 6-17. L2 Latency With Cache Off

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 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 `xCPLB_ADDR[n]` and a properties descriptor word `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.

- 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.

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.

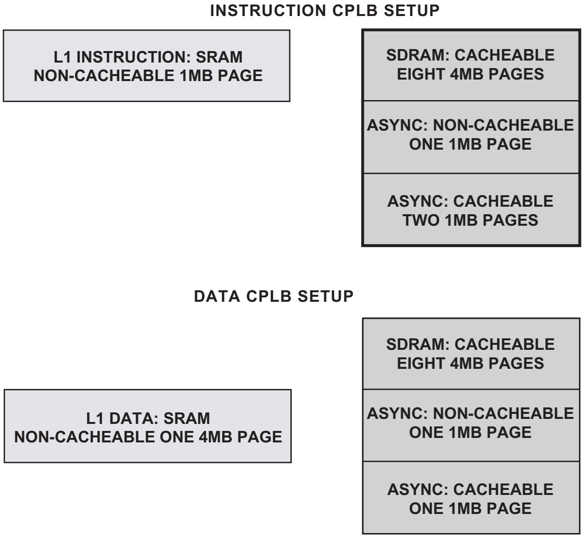

INSTRUCTION CPLB SETUP
L1 INSTRUCTION: SRAM NON-CACHEABLE 1MB PAGE
SDRAM: CACHEABLE EIGHT 4MB PAGES
ASYNC: NON-CACHEABLE ONE 1MB PAGE
ASYNC: CACHEABLE TWO 1MB PAGES
DATA CPLB SETUP
L1 DATA: SRAM NON-CACHEABLE ONE 4MB PAGE
SDRAM: CACHEABLE EIGHT 4MB PAGES
ASYNC: NON-CACHEABLE ONE 1MB PAGE
ASYNC: CACHEABLE ONE 1MB PAGE

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.

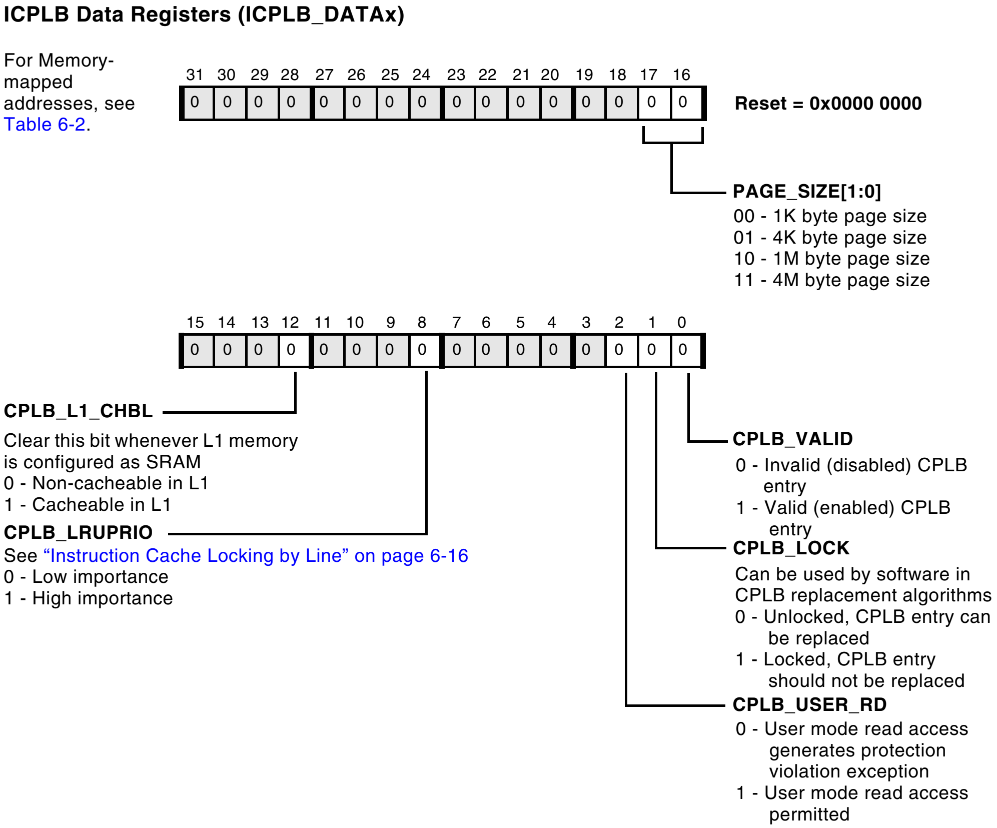

ICPLB Data Registers (ICPLB_DATAx)
For Memory-mapped addresses, see Table 6-2.
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
PAGE_SIZE[1:0]
00 - 1K byte page size
01 - 4K byte page size
10 - 1M byte page size
11 - 4M byte page size
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
CPLB_L1_CHBL
Clear this bit whenever L1 memory is configured as SRAM
0 - Non-cacheable in L1
1 - Cacheable in L1
CPLB_LRUPRIO
See "Instruction Cache Locking by Line" on page 6-16
0 - Low importance
1 - High importance
CPLB_VALID
0 - Invalid (disabled) CPLB entry
1 - Valid (enabled) CPLB entry
CPLB_LOCK
Can be used by software in CPLB replacement algorithms
0 - Unlocked, CPLB entry can be replaced
1 - Locked, CPLB entry should not be replaced
CPLB_USER_RD
0 - User mode read access generates protection violation exception
1 - User mode read access permitted

Figure 6-19. ICPLB Data Registers

Table 6-2. ICPLB Data Register Memory-mapped Addresses

Register Name	Memory-mapped Address
ICPLB_DATA0	0xFFE0 1200
ICPLB_DATA1	0xFFE0 1204
ICPLB_DATA2	0xFFE0 1208
ICPLB_DATA3	0xFFE0 120C
ICPLB_DATA4	0xFFE0 1210
ICPLB_DATA5	0xFFE0 1214
ICPLB_DATA6	0xFFE0 1218
ICPLB_DATA7	0xFFE0 121C
ICPLB_DATA8	0xFFE0 1220
ICPLB_DATA9	0xFFE0 1224
ICPLB_DATA10	0xFFE0 1228
ICPLB_DATA11	0xFFE0 122C
ICPLB_DATA12	0xFFE0 1230
ICPLB_DATA13	0xFFE0 1234
ICPLB_DATA14	0xFFE0 1238
ICPLB_DATA15	0xFFE0 123C

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.

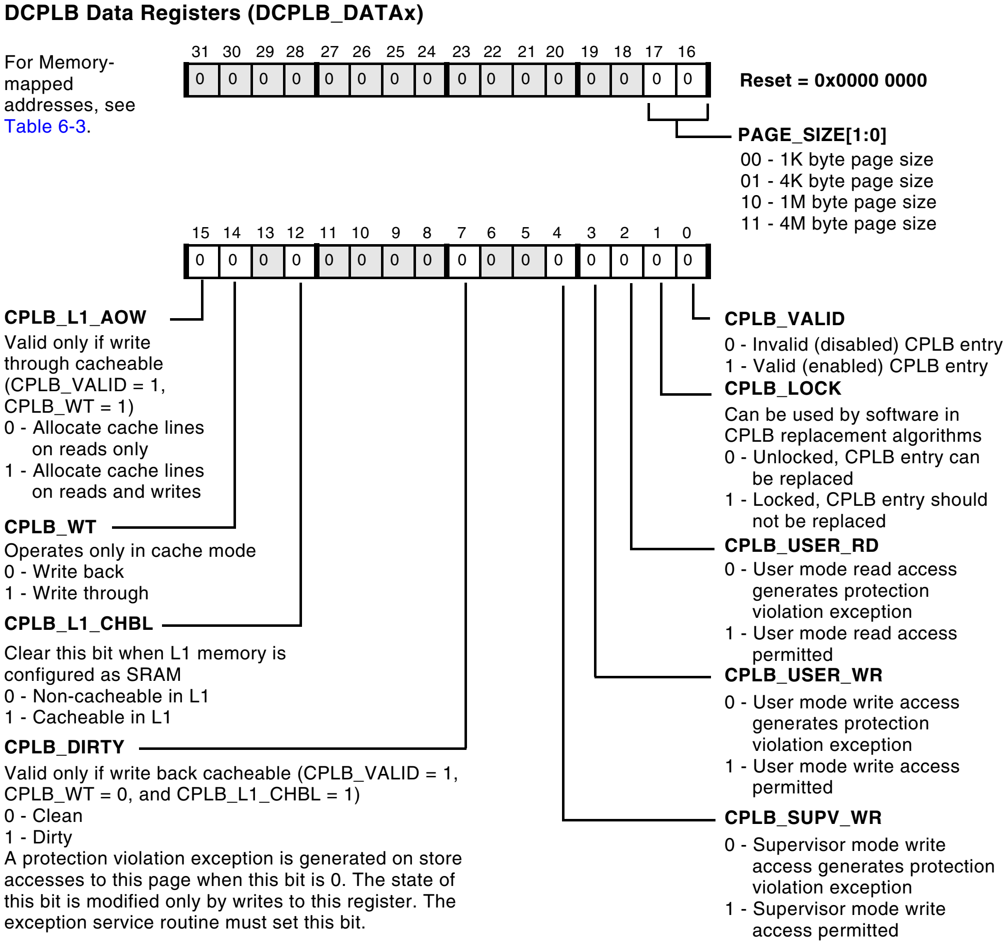

DCPLB Data Registers (DCPLB_DATAx)
For Memory-mapped addresses, see Table 6-3.
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
PAGE_SIZE[1:0]
00 - 1K byte page size
01 - 4K byte page size
10 - 1M byte page size
11 - 4M byte page size
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
CPLB_L1_AOW
Valid only if write through cacheable (CPLB_VALID = 1, CPLB_WT = 1)
0 - Allocate cache lines on reads only
1 - Allocate cache lines on reads and writes
CPLB_WT
Operates only in cache mode
0 - Write back
1 - Write through
CPLB_L1_CHBL
Clear this bit when L1 memory is configured as SRAM
0 - Non-cacheable in L1
1 - Cacheable in L1
CPLB_DIRTY
Valid only if write back cacheable (CPLB_VALID = 1, CPLB_WT = 0, and CPLB_L1_CHBL = 1)
0 - Clean
1 - Dirty
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.
CPLB_VALID
0 - Invalid (disabled) CPLB entry
1 - Valid (enabled) CPLB entry
CPLB_LOCK
Can be used by software in CPLB replacement algorithms
0 - Unlocked, CPLB entry can be replaced
1 - Locked, CPLB entry should not be replaced
CPLB_USER_RD
0 - User mode read access generates protection violation exception
1 - User mode read access permitted
CPLB_USER_WR
0 - User mode write access generates protection violation exception
1 - User mode write access permitted
CPLB_SUPV_WR
0 - Supervisor mode write access generates protection violation exception
1 - Supervisor mode write access permitted

Figure 6-20. DCPLB Data Registers

Table 6-3. DCPLB Data Register Memory-mapped Addresses

Register Name	Memory-mapped Address
DCPLB_DATA0	0xFFE0 0200
DCPLB_DATA1	0xFFE0 0204
DCPLB_DATA2	0xFFE0 0208
DCPLB_DATA3	0xFFE0 020C
DCPLB_DATA4	0xFFE0 0210
DCPLB_DATA5	0xFFE0 0214
DCPLB_DATA6	0xFFE0 0218
DCPLB_DATA7	0xFFE0 021C
DCPLB_DATA8	0xFFE0 0220
DCPLB_DATA9	0xFFE0 0224
DCPLB_DATA10	0xFFE0 0228
DCPLB_DATA11	0xFFE0 022C
DCPLB_DATA12	0xFFE0 0230
DCPLB_DATA13	0xFFE0 0234
DCPLB_DATA14	0xFFE0 0238
DCPLB_DATA15	0xFFE0 023C

DCPLB_ADDRx Registers

Figure 6-21 shows the DCPLB Address registers (DCPLB_ADDRx).

DCPLB Address Registers (DCPLB_ADDRx)

For Memory-mapped addresses, see Table 6-4.

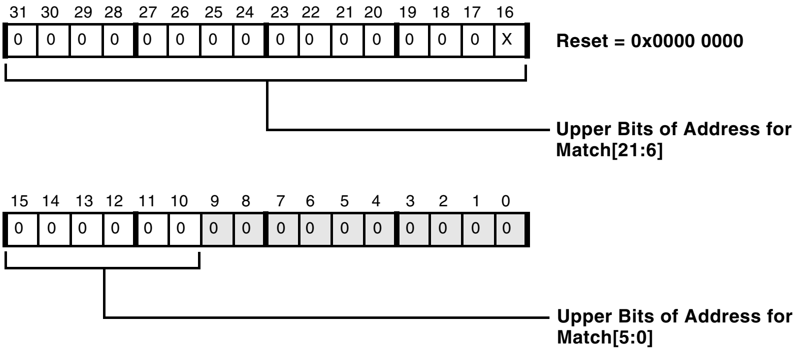


Figure 6-21. DCPLB Address Registers

Table 6-4. DCPLB Address Register Memory-mapped Addresses

Register Name	Memory-mapped Address
DCPLB_ADDR0	0xFFE0 0100
DCPLB_ADDR1	0xFFE0 0104
DCPLB_ADDR2	0xFFE0 0108
DCPLB_ADDR3	0xFFE0 010C
DCPLB_ADDR4	0xFFE0 0110
DCPLB_ADDR5	0xFFE0 0114
DCPLB_ADDR6	0xFFE0 0118
DCPLB_ADDR7	0xFFE0 011C
DCPLB_ADDR8	0xFFE0 0120
DCPLB_ADDR9	0xFFE0 0124

Table 6-4. DCPLB Address Register Memory-mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
DCPLB_ADDR10	0xFFE0 0128
DCPLB_ADDR11	0xFFE0 012C
DCPLB_ADDR12	0xFFE0 0130
DCPLB_ADDR13	0xFFE0 0134
DCPLB_ADDR14	0xFFE0 0138
DCPLB_ADDR15	0xFFE0 013C

ICPLB_ADDRx Registers

Figure 6-22 shows the ICPLB Address registers (ICPLB_ADDRx).

ICPLB Address Registers (ICPLB_ADDRx)

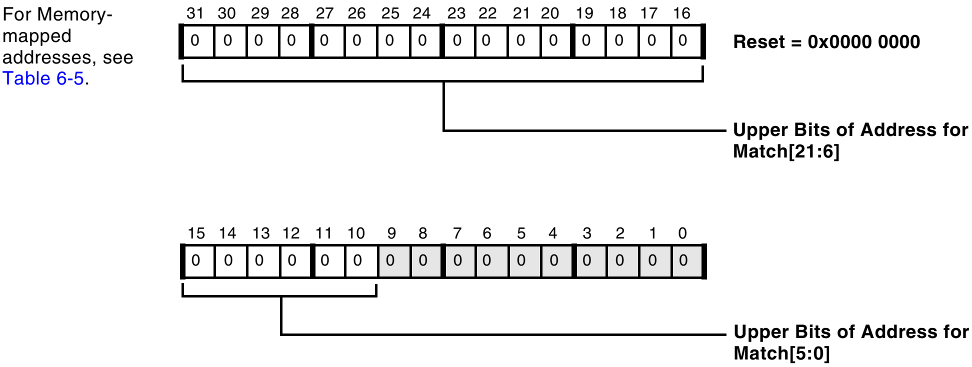


Figure 6-22. ICPLB Address Registers

Table 6-5. ICPLB Address Register Memory-mapped Addresses

Register Name	Memory-mapped Address
ICPLB_ADDR0	0xFFE0 1100
ICPLB_ADDR1	0xFFE0 1104
ICPLB_ADDR2	0xFFE0 1108
ICPLB_ADDR3	0xFFE0 110C
ICPLB_ADDR4	0xFFE0 1110
ICPLB_ADDR5	0xFFE0 1114
ICPLB_ADDR6	0xFFE0 1118
ICPLB_ADDR7	0xFFE0 111C
ICPLB_ADDR8	0xFFE0 1120
ICPLB_ADDR9	0xFFE0 1124
ICPLB_ADDR10	0xFFE0 1128
ICPLB_ADDR11	0xFFE0 112C
ICPLB_ADDR12	0xFFE0 1130
ICPLB_ADDR13	0xFFE0 1134
ICPLB_ADDR14	0xFFE0 1138
ICPLB_ADDR15	0xFFE0 113C

DCPLB_STATUS and ICPLB_STATUS 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.

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).

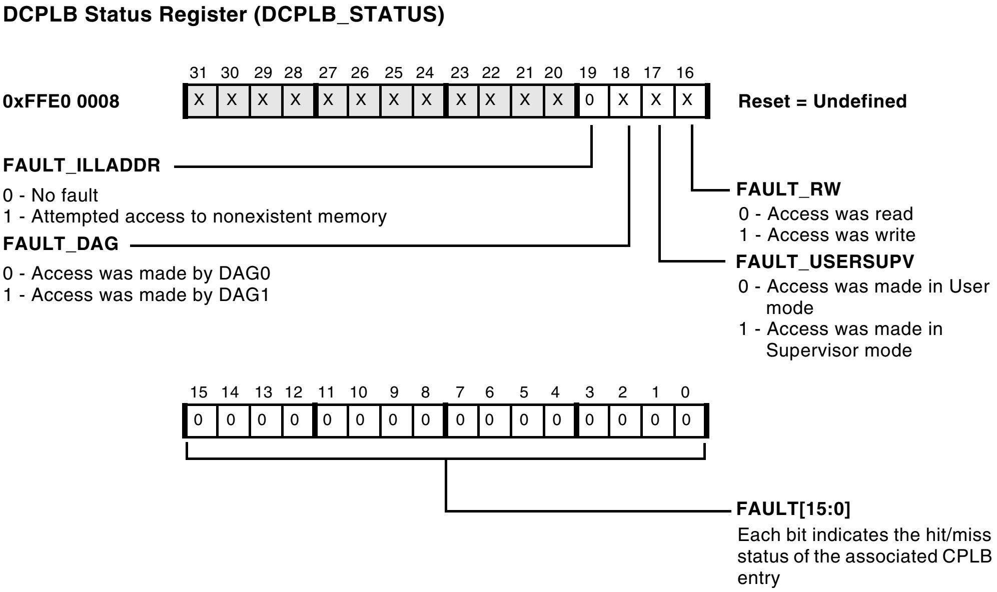
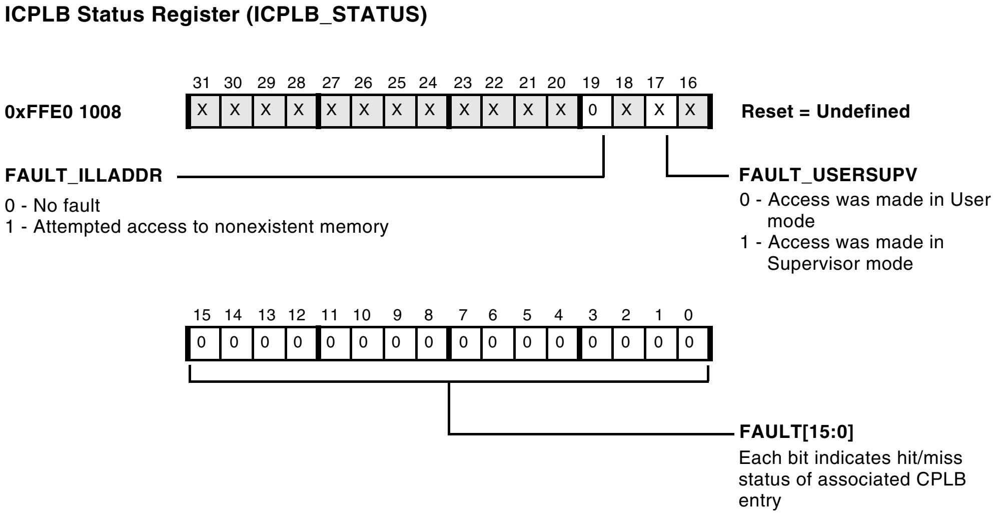

DCPLB Status Register (DCPLB_STATUS)
0xFFE0 0008
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X 0 X X X
Reset = Undefined
FAULT_ILLADDR
0 - No fault
1 - Attempted access to nonexistent memory
FAULT_DAG
0 - Access was made by DAG0
1 - Access was made by DAG1
FAULT_RW
0 - Access was read
1 - Access was write
FAULT_USERSUPV
0 - Access was made in User mode
1 - Access was made in Supervisor mode
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
FAULT[15:0]
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).


ICPLB Status Register (ICPLB_STATUS)
0xFFE0 1008
Reset = Undefined
FAULT_ILLADDR
0 - No fault
1 - Attempted access to nonexistent memory
FAULT_USERSUPV
0 - Access was made in User mode
1 - Access was made in Supervisor mode
FAULT[15:0]
Each bit indicates hit/miss status of associated CPLB entry

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
X	X	X	X	X	X	X	X	X	X	X	X	0	X	X	X

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 6-24. ICPLB Status Register

DCPLB_FAULT_ADDR and ICPLB_FAULT_ADDR Registers

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.

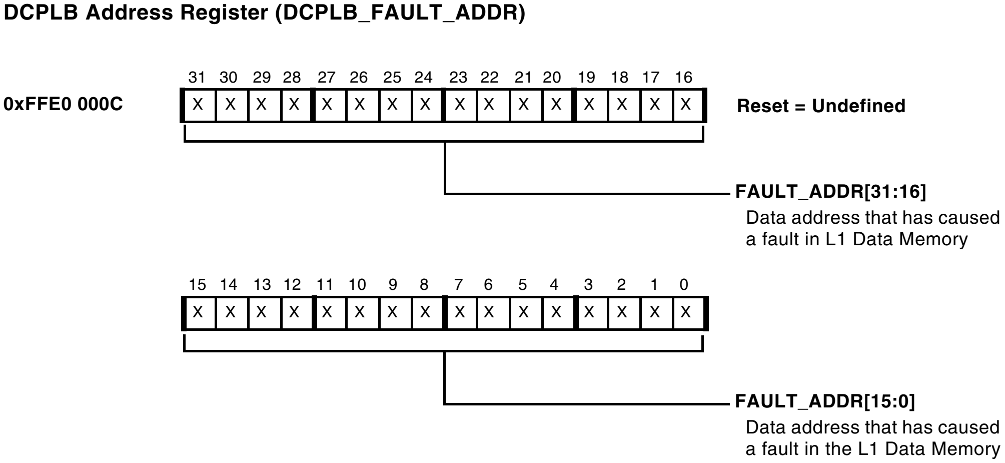

DCPLB Address Register (DCPLB_FAULT_ADDR)
0xFFE0 000C
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
FAULT_ADDR[31:16]
Data address that has caused a fault in L1 Data Memory
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
FAULT_ADDR[15:0]
Data address that has caused a fault in the L1 Data Memory

Figure 6-25. DCPLB Address Register

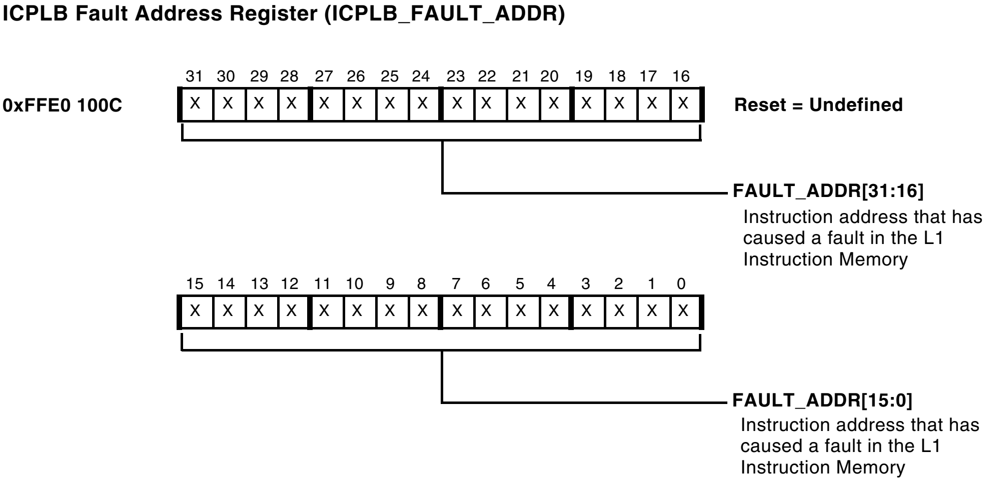

ICPLB Fault Address Register (ICPLB_FAULT_ADDR)
0xFFE0 100C
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
FAULT_ADDR[31:16]
Instruction address that has caused a fault in the L1 Instruction Memory
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
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.

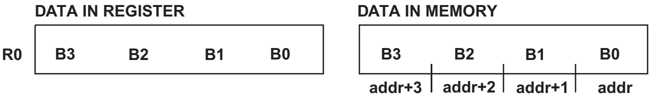


Figure 6-27. Data Stored in Little Endian Order

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*).

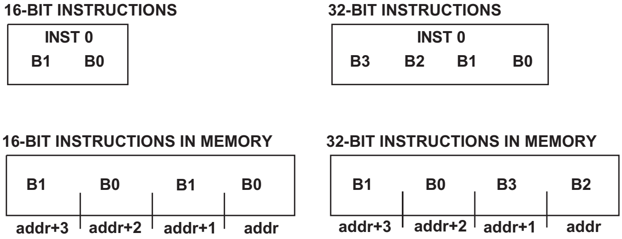


Figure 6-28. Instructions Stored in Little Endian Order

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.

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:

```
  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,

```
            IF CC JUMP away_from_here
            RO = [P2];
            …
away_from_here:
```

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 Misaligned 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.

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 R0;   /*  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 R0;   /*  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).

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

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.

Jump

General Form

```
JUMP (destination_indirect)
JUMP (PC + offset)
JUMP offset
JUMP.S offset
JUMP.L offset
```

Syntax

```
JUMP ( Preg ) ;   /* indirect to an absolute (not PC-relative)
address (a) */
JUMP ( PC + Preg ) ;   /* PC-relative, indexed (a) */
JUMP pcrel25m2 ;   /* PC-relative, immediate (a) or (b) */
```

see "Functional Description" on page 7-3[1]

```
JUMP.S pcrel13m2 ;    /* PC-relative, immediate, short (a) */
JUMP.L pcrel25m2 ;   /* PC-relative, immediate, long (b) */
JUMP 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, FP`

`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)

`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.

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.

Example

```
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

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) */
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 pre-
dicted as taken (a) */
IF CC JUMP user_label ;   /* user-defined absolute address label,
resolved by the assembler/linker to the appropriate PC-relative
instruction (a) */
IF CC JUMP user_label (bp) ;   /* user-defined absolute address
label, resolved by the assembler/linker to the appropriate
PC-relative instruction (a) */
IF !CC JUMP user_label ;   /* user-defined absolute address
label, resolved by the assembler/linker to the appropriate
PC-relative instruction (a) */
IF !CC JUMP user_label (bp) ;   /* user-defined absolute address
label, resolved by the assembler/linker to the appropriate
PC-relative instruction (a) */
```

[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.

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

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

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.

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

```
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

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.

Table 7-1. Types of Return Instruction

Mnemonic	Description
RTS	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.
RTI	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 "nested") 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.
RTX	Forces a return from an exception routine by loading the value of the RETX Register into the PC.
RTN	Forces a return from a non-maskable interrupt (NMI) routine by loading the value of the RETN Register into the PC.
RTE	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.

Table 7-2. Required Mode for the Return Instruction

Mnemonic	Required Mode
RTS	User & Supervisor
RTI, RTX, and RTN	Supervisor only. Any attempt to execute in User mode produces a protection violation exception.
RTE	Emulation only. Any attempt to execute in User mode or Supervisor mode produces an exception.

Example

```
rts ;
rti ;
rtx ;
rtn ;
rte ;
```

Also See

Call, --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 ;   /* autoinitialize LC0 (b) */
LOOP loop_name LC0 = Preg >> 1 ; /* autoinit LC0(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 ( pcrel5m2 , lppcrel11m2 ) LC0 ;   /* (b) */
LSETUP ( pcrel5m2 , lppcrel11m2 ) LC0 = Preg ;   /* autoinitial-
ize LC0 (b) */
LSETUP ( pcrel5m2 , lppcrel11m2 ) LC0 = Preg >> 1 ;   /* autoini-
tialize LC0 (b) */
```

For Loop1

```
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 ;   /* autoinitial-
ize LC1 (b) */
LSETUP ( pcrel5m2 , lppcrel11m2 ) LC1 = Preg >> 1 ;   /* autoini-
tialize 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.

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.

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.

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 LBn, 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.)

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 ;
```

```
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

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

```
register = constant
A1 = A0 = 0
```

Syntax

Half-Word Load

```
reg_lo = uimm16  ;   /* 16-bit value into low-half data or
address register (b) */
reg_hi = uimm16 ;   /* 16-bit value into high-half data or
address register (b) */
```

Zero Extended

```
reg = uimm16 (Z) ;   /* 16-bit value, zero-extended, into data or
address register (b) */
A0 = 0 ;   /* Clear A0 register (b) */
A1 = 0 ;   /* Clear A1 register (b) */
A1 = A0 = 0 ;   /* Clear both A1 and A0 registers (b) */
```

Sign Extended

```
Dreg = imm7 (X) ;   /* 7-bit value, sign extended, into Dreg (a)
*/
Preg = imm7 (X)  ;   /* 7-bit value, sign extended, into Preg
(a) */
reg = imm16 (X)  ;   /* 16-bit value, sign extended, into data or
address register (b) */
```

Syntax Terminology

Dreg: R7–0

Preg: P5–0, SP, FP

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_hi: R7-0.H, P5-0.H, SP.H, FP.H, I3-0.H, M3-0.H, B3-0.H, L3-0.H

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)

uimm16: 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 = foo ;
p3.l = 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

```
r7 = 63 (z) ;
p3 = 12 (z) ;
r0 = -344 (x) ;
r7 = 436 (z) ;
m2 = 0x89ab (z) ;
p1 = 0x1234 (z) ;
m3 = 0x3456 (x) ;
l3.h = 0xbcde ;
```

```
a0 = 0 ;
a1 = 0 ;
a1 = a0 = 0 ;
```

Also See

Load Pointer Register

Special Applications

Use the Load Immediate instruction to initialize registers.

Load Pointer Register

General Form

```
P-register = [ indirect_address ]
```

Syntax

```
Preg = [ Preg ] ;    /* indirect (a) */
Preg = [ Preg ++ ] ;    /* indirect, post-increment (a) */
Preg = [ Preg -- ] ;    /* indirect, post-decrement (a) */
Preg = [ Preg + uimm6m4 ] ;    /* indexed with small offset (a) */
Preg = [ Preg + uimm17m4 ] ;    /* indexed with large offset
(b) */
Preg = [ Preg - uimm17m4 ] ;    /* indexed with large offset
(b) */
Preg = [ FP - uimm7m4 ] ;    /* indexed FP-relative (a) */
```

Syntax Terminology

Preg: P5–0, SP, FP

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 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 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

Load Data Register

General Form

```
D-register = [ indirect_address ]
```

Syntax

```
Dreg = [ Preg ] ;    /* indirect (a) */
Dreg = [ Preg ++ ] ;    /* indirect, post-increment (a) */
Dreg = [ Preg -- ] ;    /* indirect, post-decrement (a) */
Dreg = [ Preg + uimm6m4 ] ;    /* indexed with small offset (a) */
Dreg = [ Preg + uimm17m4 ] ;    /* indexed with large offset
(b) */
Dreg = [ Preg - uimm17m4 ] ;    /* indexed with large offset
(b) */
Dreg = [ Preg ++ Preg ] ;    /* indirect, post-increment index
(a) */[1]
Dreg = [ FP - uimm7m4 ] ;    /* indexed FP-relative (a) */
Dreg = [ Ireg ] ;    /* indirect (a) */
Dreg = [ Ireg ++ ] ;    /* indirect, post-increment (a) */
Dreg = [ Ireg -- ] ;    /* indirect, post-decrement (a) */
Dreg = [ Ireg ++ Mreg ] ;    /* indirect, post-increment index
(a) */[1]
```

Syntax Terminology

Dreg: R7-0

Preg: P5-0, SP, FP

Ireg: I3-0

Mreg: M3-0

[1] See "Indirect and Post-Increment Index Addressing" on page 8-12.

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 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 *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 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. *`Preg`* includes the Frame Pointer and Stack Pointer.

Indirect and Post-Increment Index Addressing

The syntax of the form:

```
Dest = [ Src_1 ++ Src_2 ]
```

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

```
Dest = [Src_1] ;   /* load the 32-bit destination, indirect*/
Src_1 += Src_2 ;   /* post-increment Src_1 by a quantity indexed
by Src_2 */
```

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

```
r3 = [ p0 ] ;
r7 = [ p1 ++ ] ;
r2 = [ sp -- ] ;
r6 = [ p2 + 12 ] ;
r0 = [ p4 + 0x800C ] ;
```

```
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 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 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:

```
Dest = W [ Src_1 ++ Src_2 ]
```

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

```
Dest = [Src_1] ;    /* load the 32-bit destination, indirect*/
Src_1 += Src_2 ;    /* post-increment Src_1 by a quantity indexed
by Src_2 */
```

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 instruction functions as a simple, non-incrementing load. For example, `r0 = W[p2++p2](z)` functions as `r0 = W[p2](z)`.

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

```
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) ;
```

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-register = W [ indirect_address ] (X)
```

Syntax

```
Dreg = W [ Preg ] (X) ;    // indirect (a)
Dreg = W [ Preg ++ ] (X) ;    // indirect, post-increment (a)
Dreg = W [ Preg -- ] (X) ;    // indirect, post-decrement (a)
Dreg = W [ Preg + uimm5m2 ] (X) ;    /* indexed with small offset
(a) */
Dreg = W [ Preg + uimm16m2 ] (X) ;    /* indexed with large offset
(b) */
Dreg = W [ Preg - uimm16m2 ] (X) ;    /* indexed with large offset
(b) */
Dreg = W [ Preg ++ Preg ] (X) ;    /* 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 0xFFFE)

[1] See “Indirect and Post-Increment Index Addressing” on page 8-21.

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 replicated 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:

```
Dest = W [ Src_1 ++ Src_2 ] (X)
```

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

```
Dest = [Src_1] ;   /* load the 32-bit destination, indirect*/
Src_1 += Src_2 ;   /* post-increment Src_1 by a quantity indexed
by Src_2 */
```

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 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

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 High Data Register Half

General Form

```
Dreg_hi = W [ indirect_address ]
```

Syntax

```
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]
```

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.

Indirect and Post-Increment Index Addressing

```
Dst_hi = [ Src_1 ++ Src_2 ]
```

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

```
Dst_hi = [Src_1] ;   /* load the half-word into the upper half of
the destination register, indirect*/
Src_1 += Src_2 ;   /* post-increment Src_1 by a quantity indexed
by Src_2 */
```

where:

- `Dst_hi` is the most significant half of the destination register. (`Dreg_hi` in the syntax example).
- `Src_1` is the memory source pointer register on the right-hand side of the syntax.
- `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

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 [ indirect_address ]
```

Syntax

```
Dreg_lo = W [ Ireg ] ;    /* indirect data addressing (a)*/
Dreg_lo = W [ Ireg ++ ] ;    /* indirect, post-increment data
addressing (a) */
Dreg_lo = W [ Ireg -- ] ;    /* indirect, post-decrement data
addressing (a) */
Dreg_lo = W [ Preg ] ;    /* indirect (a)*/
Dreg_lo = W [ Preg ++ Preg ] ;    /* indirect, post-increment
index (a) */[1]
```

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.

[1] See "Indirect and Post-Increment Index Addressing" on page 8-29.

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] ;   /* load the half-word into the lower half of
the destination register, indirect*/
Src_1 += Src_2 ;   /* 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 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.l = w[ i1 ] ;
r7.l = w[ i3 ++ ] ;
r1.l = w[ i0 -- ] ;
r2.l = w[ p4 ] ;
r5.l = 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-register = B [ indirect_address ] (Z)
```

Syntax

```
Dreg = B [ Preg ] (Z) ;   /* indirect (a)*/
Dreg = B [ Preg ++ ] (Z) ;   /* indirect, post-increment (a)*/
Dreg = B [ Preg -- ] (Z) ;   /* indirect, post-decrement (a)*/
Dreg = B [ Preg + uimm15 ] (Z) ;   /* indexed with offset (b)*/
Dreg = B [ Preg - uimm15 ] (Z) ;   /* indexed with offset (b)*/
```

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 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.

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.

Example

```
r3 = b [ p0 ] (z) ;
r7 = b [ p1 ++ ] (z) ;
r2 = b [ sp -- ] (z) ;
r0 = b [ p4 + 0xFFFF800F ] (z) ;
```

Also See

Load Byte – Sign-Extended

Special Applications

None

Load Byte – Sign-Extended

General Form

```
D-register = B [ indirect_address ] (X)
```

Syntax

```
Dreg = B [ Preg ] (X) ;   /* indirect (a)*/
Dreg = B [ Preg ++ ] (X) ;   /* indirect, post-increment (a)*/
Dreg = B [ Preg -- ] (X) ;   /* indirect, post-decrement (a)*/
Dreg = B [ Preg + uimm15 ] (X) ;   /* indexed with offset (b)*/
Dreg = B [ Preg - uimm15 ] (X) ;   /* indexed with offset (b)*/
```

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 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.

Example

```
r3 = b [ p0 ] (x) ;
r7 = b [ p1 ++ ](x) ;
r2 = b [ sp -- ] (x) ;
r0 = b [ p4 + 0xFFFF800F ](x) ;
```

Also See

Load Byte – Zero-Extended

Special Applications

None

Store Pointer Register

General Form

```
[ indirect_address ] = P-register
```

Syntax

```
[ Preg ] = Preg ;   /* indirect (a)*/
[ Preg ++ ] = Preg ;   /* indirect, post-increment (a)*/
[ Preg -- ] = Preg ;   /* indirect, post-decrement (a)*/
[ Preg + uimm6m4 ] = Preg ;   /* indexed with small offset (a)*/
[ Preg + uimm17m4 ] = Preg ;   /* indexed with large offset (b)*/
[ Preg - uimm17m4 ] = Preg ;   /* indexed with large offset (b)*/
[ FP - uimm7m4 ] = Preg ;   /* indexed FP-relative (a)*/
```

Syntax Terminology

Preg: P5–0, SP, FP

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 (0x000 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 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

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

```
[ p2 ] = p3 ;
[ sp ++ ] = p5 ;
[ p0 -- ] = p2 ;
[ p2 + 8 ] = p3 ;
[ p2 + 0x4444 ] = p0 ;
[ fp -12 ] = p1 ;
```

Also See

--SP (Push), --SP (Push Multiple)

Special Applications

None

Store Data Register

General Form

```
[ indirect_address ] = D-register
```

Syntax

Using Pointer Registers

```
[ Preg ] = Dreg ;    /* indirect (a)*/
[ Preg ++ ] = Dreg ;    /* indirect, post-increment (a)*/
[ Preg -- ] = Dreg ;    /* indirect, post-decrement (a)*/
[ Preg + uimm6m4 ] = Dreg ;    /* indexed with small offset (a)*/
[ Preg + uimm17m4 ] = Dreg ;    /* indexed with large offset (b)*/
[ Preg - uimm17m4 ] = Dreg ;    /* indexed with large offset (b)*/
[ Preg ++ Preg ] = Dreg ;    /* indirect, post-increment index (a)
*/[1]
[ FP - uimm7m4 ] = Dreg ;    /* indexed FP-relative (a)*/
```

Using Data Address Generator (DAG) Registers

```
[ Ireg ] = Dreg ;    /* indirect (a)*/
[ Ireg ++ ] = Dreg ;    /* indirect, post-increment (a)*/
[ Ireg -- ] = Dreg ;    /* indirect, post-decrement (a)*/
[ Ireg ++ Mreg ] = Dreg ;    /* indirect, post-increment index (a)
*/
```

Syntax Terminology

Dreg: R7-0

Preg: P5-0, SP, FP

Ireg: I3-0

[1] See "Indirect and Post-Increment Index Addressing" on page 8-43.

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 0xFFFC)

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.

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 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:

```
[Dst_1 ++ Dst_2] = Src
```

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

```
[Dst_1] = Src ;   /* load the 32-bit source, indirect*/
Dst_1 += Dst_2 ;   /* post-increment Dst_1 by a quantity indexed
by Dst_2 */
```

where:

- `Src` is the source register. (`Dreg` in the syntax example).
- `Dst_1` is the memory destination register on the left side of the equation.
- `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

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

```
[ p0 ] = r3 ;
[ p1 ++ ] = r7 ;
[ sp -- ] = r2 ;
[ p2 + 12 ] = r6 ;
[ p4 - 0x1004 ] = r0 ;
[ p0 ++ p1 ] = r1 ;
[ fp - 28 ] = r5 ;
[ i2 ] = r2 ;
[ i0 ++ ] = r0 ;
[ i0 -- ] = r0 ;
[ i3 ++ m0 ] = r7 ;
```

Also See

Load Immediate

Special Applications

None

Store High Data Register Half

General Form

```
W [ indirect_address ] = Dreg_hi
```

Syntax

```
W [ Ireg ] = Dreg_hi ;    /* indirect data addressing (a)*/
W [ Ireg ++ ] = Dreg_hi ;    /* indirect, post-increment data
addressing (a) */
W [ Ireg -- ] = Dreg_hi ;    /* indirect, post-decrement data
addressing (a) */
W [ Preg ] = Dreg_hi ;    /* indirect (a)*/
W [ Preg ++ Preg ] = Dreg_hi ;    /* indirect, post-increment
index (a) */[1]
```

Syntax Terminology

Dreg_hi: P7-0.H

Preg: P5-0, SP, FP

Ireg: I3-0

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

Functional Description

The 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.

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 *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 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 ++ Dst_2] = Src_hi
```

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

```
[Dst_1] = Src_hi ;   /* store the upper half of the source regis-
ter, indirect*/
Dst_1 += Dst_2 ;   /* post-increment Dst_1 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

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.

Store Low Data Register Half

General Form

```
W [ indirect_address ] = Dreg_lo
W [ indirect_address ] = D-register
```

Syntax

```
W [ Ireg ] = Dreg_lo ;   /* indirect data addressing (a)*/
W [ Ireg ++ ] = Dreg_lo ;   /* indirect, post-increment data
addressing (a) */
W [ Ireg -- ] = Dreg_lo ;   /* indirect, post-decrement data
addressing (a) */
W [ Preg ] = Dreg_lo ;   /* indirect (a)*/
W [ Preg ] = Dreg ;   /* indirect (a)*/
W [ Preg ++ ] = Dreg ;   /* indirect, post-increment (a)*/
W [ Preg -- ] = Dreg ;   /* indirect, post-decrement (a)*/
W [ Preg + uimm5m2 ] = Dreg ;   /* indexed with small offset (a)
*/
W [ Preg + uimm16m2 ] = Dreg ;   /* indexed with large offset (b)
*/
W [ Preg - uimm16m2 ] = Dreg ;   /* indexed with large offset (b)
*/
W [ Preg ++ Preg ] = Dreg_lo ;   /* indirect, post-increment
index (a) */[1]
```

Syntax Terminology

Dreg_lo: R7-0.L

Preg: P5-0, SP, FP

Ireg: I3-0

[1] See "Indirect and Post-Increment Index Addressing" on page 8-51.

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.

> 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 ;   /* store the lower half of the source regis-
ter, indirect*/
Dst_1 += Dst_2 ;   /* post-increment Dst_1 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.

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

```
w [ i1 ] = r3.l ;
w [ p0 ] = r3 ;
w [ i3 ++ ] = r7.l ;
w [ i0 -- ] = r1.l ;
w [ p4 ] = r2.l ;
w [ p1 ++ ] = r7 ;
w [ sp -- ] = r2 ;
w [ p2 + 12 ] = r6 ;
w [ p4 - 0x200C ] = r0 ;
w [ p2 ++ p0 ] = r5.l ;
```

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.

Store Byte

General Form

```
B [ indirect_address ] = D-register
```

Syntax

```
B [ Preg ] = Dreg ;   /* indirect (a)*/
B [ Preg ++ ] = Dreg ;   /* indirect, post-increment (a)*/
B [ Preg -- ] = Dreg ;   /* indirect, post-decrement (a)*/
B [ Preg + uimm15 ] = Dreg ;   /* indexed with offset (b)*/
B [ Preg - uimm15 ] = Dreg ;   /* indexed with offset (b)*/
```

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

```
b [ p0 ] = r3 ;
b [ p1 ++ ] = r7 ;
b [ sp -- ] = r2 ;
b [ p4 + 0x100F ] = r0 ;
b [ p4 - 0x53F ] = r0 ;
```

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

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.

Move Register

General Form

```
dest_reg = src_reg
```

Syntax

```
genreg = genreg ;   /* (a) */
genreg = dagreg ;   /* (a) */
dagreg = genreg ;   /* (a) */
dagreg = dagreg ;   /* (a) */
genreg = USP ;   /* (a)*/
USP = genreg ;   /* (a)*/
Dreg = sysreg ;   /* sysreg to 32-bit D-register (a) */
Preg = sysreg ;   /* sysreg to P-register (c) */
sysreg = Dreg ;   /* 32-bit D-register to sysreg (a) */
sysreg = Preg ;   /* 32-bit P-register to sysreg (a) */
sysreg = USP ;   /* (a) */
A0 = A1 ;   /* move 40-bit Accumulator value (b) */
A1 = A0 ;   /* move 40-bit Accumulator value (b) */
A0 = Dreg ;   /* 32-bit D-register to 40-bit A0, sign extended
(b)*/
A1 = Dreg ;   /* 32-bit D-register to 40-bit A1, sign extended
(b)*/
```

Accumulator to D-register Move:

```
Dreg_even = A0 (opt_mode) ;   /* move 32-bit A0.W to even Dreg
(b) */
Dreg_odd = A1 (opt_mode) ;   /* move 32-bit A1.W to odd Dreg (b)
*/
Dreg_even = A0, Dreg_odd = A1 (opt_mode) ;   /* move both Accumu-
lators to a register pair (b) */
Dreg_odd = A1, Dreg_even = A0 (opt_mode) ;   /* move both Accumu-
lators to a register pair (b) */
```

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, LC0 and LC1, LT0 and LT1, LB0 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.

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 9-1. Accumulator to Data Register Move

Option	Accumulator Copy Formatting
Default	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.
(FU)	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.

Table 9-1. Accumulator to Data Register Move (Cont'd)

Option	Accumulator Copy Formatting
(S2RND)	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.
(ISS2)	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.

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.

- 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 set if result is negative; cleared if non-negative. In the case of two simultaneous operations, AN represents the logical “OR” of the two.
- 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 for most cases.

Explicit accesses to USP, SEQSTAT, SYSCFG, RETI, RETX, RETN and RETE 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

```
r3 = r0 ;
r7 = p2 ;
r2 = a0 ;
a0 = a1 ;
```

```
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

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

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

Move Half to Full Word – Zero-Extended

General Form

```
dest_reg = src_reg (Z)
```

Syntax

```
Dreg = Dreg_lo (Z) ;    /* (a) */
```

Syntax Terminology

Dreg: R7–0

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 – 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

```
   /* If r0.l = 0xFFFF */
r4 = r0.l (z) ;   /* Equivalent to r4.l = r0.l and r4.h = 0 */
   /* . . . then r4 = 0x0000FFFF */
```

Also See

Move Half to Full Word – Sign-Extended, Move Register Half

Special Applications

None

Move Half to Full Word – Sign-Extended

General Form

```
dest_reg = src_reg (X)
```

Syntax

```
Dreg = Dreg_lo (X) ;    /* (a)*/
```

Syntax Terminology

Dreg: R7–0

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.

- 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

```
r4 = r0.l(x) ;
r4 = r0.l ;
```

Also See

Move Half to Full Word – Zero-Extended, Move Register Half

Special Applications

None

Move Register Half

General Form

```
dest_reg_half = src_reg_half
dest_reg_half = accumulator (opt_mode)
```

Syntax

```
A0.X = Dreg_lo ;    /* least significant 8 bits of Dreg into A0.X
(b) */[1]
A1.X = Dreg_lo ;    /* least significant 8 bits of Dreg into A1.X
(b) */
Dreg_lo = A0.X ;    /* 8-bit A0.X, sign-extended, into least sig-
nificant 16 bits of Dreg (b) */
Dreg_lo = A1.X ;    /* 8-bit A1.X, sign-extended, into least sig-
nificant 16 bits of Dreg (b) */
A0.L = Dreg_lo ;    /* least significant 16 bits of Dreg into
least significant 16 bits of A0.W (b) */
A1.L = Dreg_lo ;    /* least significant 16 bits of Dreg into
least significant 16 bits of A1.W (b) */
A0.H = Dreg_hi ;    /* most significant 16 bits of Dreg into most
significant 16 bits of A0.W (b) */
A1.H = Dreg_hi ;    /* most significant 16 bits of Dreg into most
significant 16 bits of A1.W (b) */
```

[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).

Accumulator to Half D-register Moves

```
Dreg_lo = A0 (opt_mode) ; /* move A0 to lower half of Dreg (b) */
Dreg_hi = A1 (opt_mode) ;   /* move A1 to upper half of Dreg (b)
*/

Dreg_lo = A0, Dreg_hi = A1 (opt_mode) ; /* move both values at
once; must go to the lower and upper halves of the same Dreg (b)
*/
Dreg_hi = A1, Dreg_lo = A0 (opt_mode) ;   /* move both values at
once; must go to the upper and lower halves of the same Dreg (b)
*/
```

Syntax Terminology

Dreg_lo: R7-0.L

Dreg_hi: R7-0.H

A0.L: the least significant 16 bits of Accumulator A0.W

A1.L: the least significant 16 bits of Accumulator A1.W

A0.H: the most significant 16 bits of Accumulator A0.W

A1.H: the most significant 16 bits of Accumulator A1.W

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.

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.

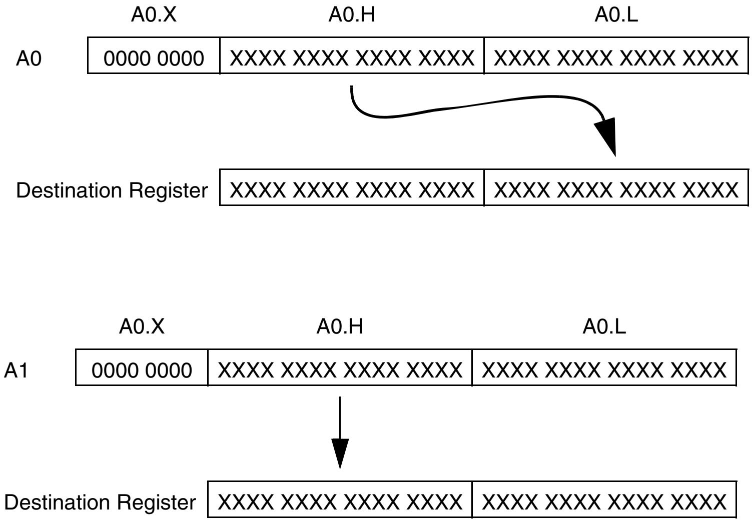


Figure 9-1. Result to Destination Register (Default Option)

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.

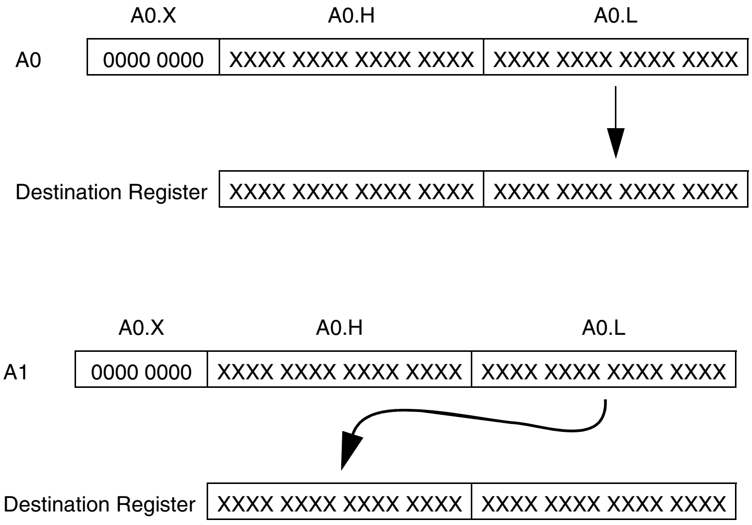


Figure 9-2. Result to Destination Register ((IS) Option)

Some versions of this instruction 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. 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.

Options

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

Option	Accumulator Copy Formatting
Default	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.
(FU)	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.
(IS)	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.
(IU)	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.
(T)	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.

Table 9-2. Accumulator to Half D-Register Move Options (Cont'd)

Option	Accumulator Copy Formatting
(S2RND)	Signed fraction with scaling and rounding. 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). The Accumulator is unaffected by extraction.
(ISS2)	Signed integer with scaling. 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). The Accumulator is unaffected by extraction.
(IH)	Signed integer, high word extract. 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). The Accumulator is unaffected by extraction.

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.

- VS is set if V is set; unaffected otherwise.

- 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. */
```

```
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 satu-
ration. */
r1.l = a0 (s2rnd) ;   /* copy A0.H into R1.L with scaling, round-
ing & saturation. */
r2.h = a1 (iss2) ;   /* copy A1.L into R2.H with scaling and sat-
uration. */
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 Byte – Zero-Extended

General Form

```
dest_reg = src_reg_byte (Z)
```

Syntax

```
Dreg = Dreg_byte (Z) ;   /* (a)*/
```

Syntax Terminology

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 – 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.
- 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 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

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.

- AZ is set if result is zero; cleared if nonzero.
- AN is set if result is negative; cleared if non-negative.
- 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 any other instructions.

Example

```
r7 = r2.b ;
r7 = r2.b(x) ;
```

Also See

Move Byte – Zero-Extended

Special Applications

None

10 STACK CONTROL

Instruction Summary

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.

--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.

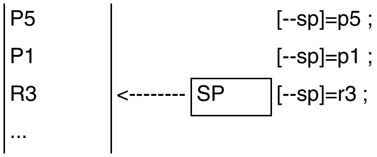
Functional Description

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.

higher memory



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.

Parallel Issue

This instruction cannot be issued in parallel with other instructions.

Example

```
[ -- sp ] = r0 ;
[ -- sp ] = r1 ;
[ -- sp ] = p0 ;
[ -- sp ] = i0 ;
```

Also See

--SP (Push Multiple), SP++ (Pop)

Special Applications

None

--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.

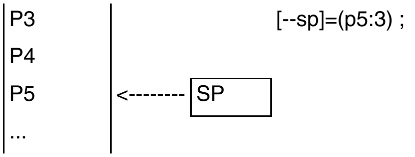
The instructions start by saving the register having the lowest index then advance to the register with the highest index. The index of the first register saved in the stack is specified by the user in the instruction syntax. Data registers are pushed before Pointer registers if both are specified in one instruction.

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



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

```
[ -- sp ] = (r7:5, p5:0) ;    /* D-registers R4:0 excluded */
[ -- sp ] = (r7:2) ;    /* R1:0 excluded */
[ -- sp ] = (p5:4) ;    /* P3:0 excluded */
```

Also See

--SP (Push), SP++ (Pop), SP++ (Pop Multiple)

Special Applications

None

SP++ (Pop)

General Form

```
dest_reg = [ SP ++ ]
```

Syntax

```
mostreg = [ SP ++ ] ;   /* post-increment SP; does not apply to
Data Registers and Pointer Registers (a) */
Dreg = [ SP ++ ] ;   /* Load Data Register instruction (repeated
here for user convenience) (a) */
Preg = [ SP ++ ] ;   /* Load Pointer Register instruction
(repeated here for user convenience) (a) */
```

Syntax Terminology

mostreg: 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, USP, SEQSTAT, and SYSCFG

Dreg: R7-0

Preg: P5-0, FP

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

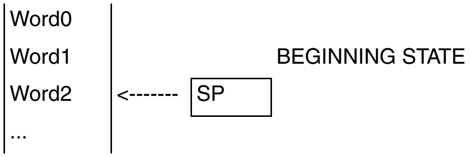
Functional Description

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.

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.

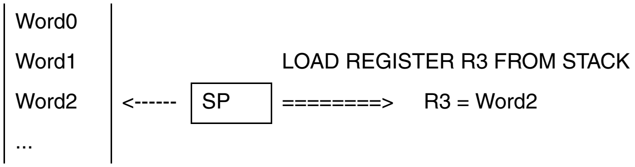
The illustration below shows what the stack would look like when a pop such as R3 = [SP ++] occurs.

higher memory



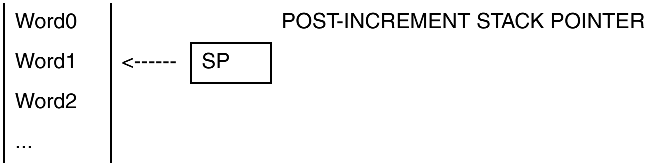
lower memory

higher memory



lower memory

higher memory



lower memory

The value just popped remains on the stack until another push instruction overwrites it.

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.

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

SP++ (Pop Multiple)

General Form

```
(dest_reg_range) = [ SP ++ ]
```

Syntax

```
( R7 : Dreglim, P5 : Preglim ) = [ SP ++ ] ;    /* Dregs and
indexed Pregs (a) */
( R7 : Dreglim ) = [ SP ++ ] ;    /* Dregs, only (a) */
( P5 : Preglim ) = [ SP ++ ] ;    /* 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 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 (R7 and/or P5) plus any contiguous lower index registers specified by the user down to and including R0 and/or P0.

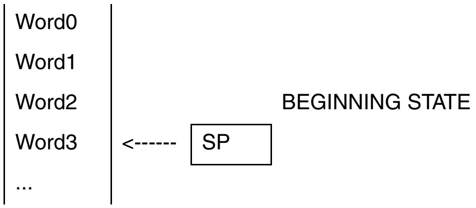
The instructions start by restoring the register having the highest index then descend to the register with the lowest index. The index of the last register restored from the stack is specified by the user in the instruction syntax. Pointer registers are popped before Data registers, if both are specified 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`.

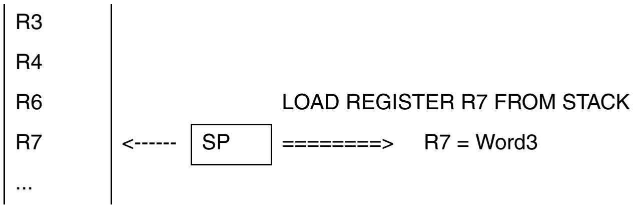
The following graphic shows what the stack would look like when a Pop Multiple such as `(R7:5) = [ SP ++ ]` occurs.

higher memory



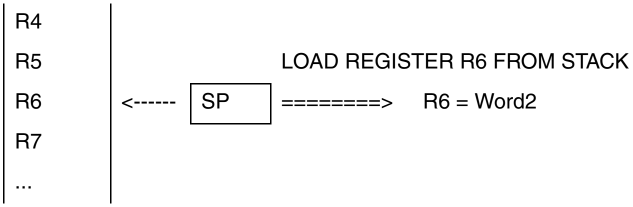
lower memory

higher memory



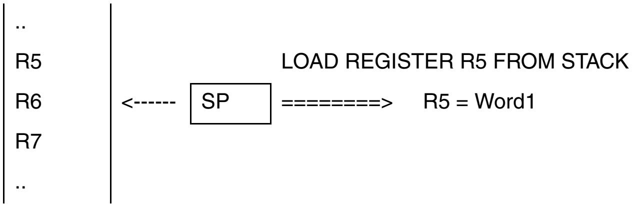
lower memory

higher memory



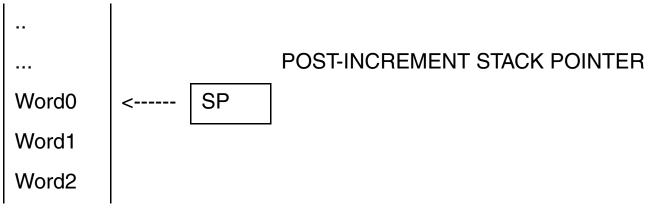
lower memory

higher memory.



lower memory

higher memory



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 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

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

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.

The user-supplied argument for LINK determines the size of the allocated stack frame. LINK always saves RETS and 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.

UNLINK performs the reciprocal of LINK, de-allocating the frame space by moving the current value of FP into SP and restoring previous values into FP and RETS from the stack.

The UNLINK instruction typically follows a Pop Multiple instruction that restores pointer and data registers previously saved to the stack.

The frame values remain on the stack until a subsequent Push, Push Multiple or LINK operation overwrites them.

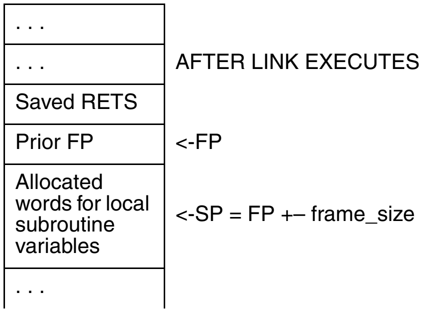
Of course, FP must not be modified by user code between LINK and UNLINK to preserve stack integrity.

Neither LINK nor 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 LINK is executing. In that case, SP and 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 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 UNLINK operation may leave the FP and RETS registers changed because of a load that has already completed before the interruption.

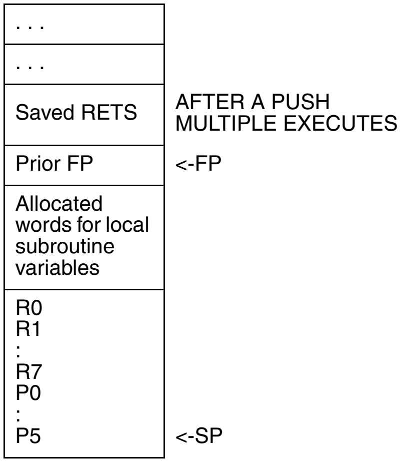
The illustrations below show the stack contents after executing a LINK instruction followed by a Push Multiple instruction.

higher memory



lower memory

higher memory


. . .
. . .
Saved RETS
AFTER A PUSH MULTIPLE EXECUTES
Prior FP
<-FP
Allocated words for local subroutine variables
R0
R1
:
R7
P0
:
P5
<-SP

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.

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

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

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

Comparison	Signed	Unsigned
Equal	AZ=1	n/a
Less than	AN=1	AC0=0
Less than or equal	AN or AZ=1	AC0=0 or AZ=1

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.
- 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.

Example

```
cc = r3 == r2 ;
cc = r7 == 1 ;
   /* If r0 = 0x8FFF FFFF and r3 = 0x0000 0001, then the signed
operation . . . */
cc = r0 < r3 ;
   /* . . . produces cc = 1, because r0 is treated as a negative
value */
cc = r2 < -4 ;
cc = r6 <= r1 ;
cc = r4 <= 3 ;
```

```
   /* If r0 = 0x8FFF FFFF and r3 = 0x0000 0001,then the unsigned
operation . . . */
cc = r0 < r3 (iu) ;
   /* . . . produces CC = 0, because r0 is treated as a large
unsigned value */
cc = r1 < 0x7 (iu) ;
cc = r2 <= r0 (iu) ;
cc = r3 <= 2 (iu) ;
```

Also See

Compare Pointer, Compare Accumulator, IF CC JUMP, BITTST

Special Applications

None

Compare Pointer

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 = Preg == Preg ;    /* equal, register, signed (a) */
CC = Preg == imm3 ;    /* equal, immediate, signed (a) */
CC = Preg < Preg ;     /* less than, register, signed (a) */
CC = Preg < imm3 ;     /* less than, immediate, signed (a) */
CC = Preg <= Preg ;    /* less than or equal, register, signed
(a) */
CC = Preg <= imm3 ;    /* less than or equal, immediate, signed
(a) */
CC = Preg < Preg (IU) ;  /* less than, register, unsigned (a) */
CC = Preg < uimm3 (IU) ; /* less than, immediate, unsigned (a) */
CC = Preg <= Preg (IU) ;    /* less than or equal, register,
unsigned (a) */
CC = Preg <= uimm3 (IU) ;    /* 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.

Example

```
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) ;
```

Also See

Compare Data Register, Compare Accumulator, IF CC JUMP

Special Applications

None

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.

Flags Affected

The Compare Accumulator instruction uses the values shown in Table 11-2 in compare operations.

Table 11-2. Compare Accumulator Instruction Values

Comparison	Signed
Equal	AZ=1
Less than	AN=1
Less than or equal	AN or AZ=1

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.

Example

```
cc = a0 == a1 ;
cc = a0 < a1 ;
cc = a0 <= a1 ;
```

Also See

Compare Pointer, Compare Data Register, IF CC JUMP

Special Applications

None

Move CC

General Form

```
dest = CC
dest |= CC
dest &= CC
dest ^= CC
CC = source
CC |= source
CC &= source
CC ^= source
```

Syntax

```
Dreg = CC ;    /* CC into 32-bit data register, zero-extended (a)
*/
statbit = CC ;     /* status bit equals CC (a) */
statbit |= CC ;    /* status bit equals status bit OR CC (a) */
statbit &= CC ;    /* status bit equals status bit AND CC (a) */
statbit ^= CC ;    /* status bit equals status bit XOR CC (a) */
CC = Dreg ;        /* CC set if the register is non-zero (a) */
CC = statbit ;     /* CC equals status bit (a) */
CC |= statbit ;    /* CC equals CC OR status bit (a) */
CC &= statbit ;    /* CC equals CC AND status bit (a) */
CC ^= statbit ;    /* CC equals CC XOR status bit (a) */
```

Syntax Terminology

Dreg: R7–0

statbit: AZ, AN, AC0, AC1, V, VS, AV0, AV0S, AV1, AV1S, AQ

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

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, AC0, AC1, V, VS, AV0, AV0S, 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.

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

```
r0 = cc ;
az = cc ;
an |= cc ;
ac0 &= cc ;
av0 ^= cc ;
cc = r4 ;
cc = av1 ;
cc |= aq ;
cc &= an ;
cc ^= ac1 ;
```

Also See

Negate CC

Special Applications

None

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.

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 =! CC ;
```

Also See

Move CC

Special Applications

None

12 LOGICAL OPERATIONS

Instruction Summary

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.

& (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.

- 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)

Special Applications

None

~ (NOT One's-Complement)

General Form

```
dest_reg = ~ src_reg
```

Syntax

```
Dreg = ~ 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 *src_reg*. The *dest_reg* and *src_reg* can be the same D-register. Using the same D-register as the *dest_reg* and *src_reg* would explicitly modify the *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.

- 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

```
r3 = ~ r4 ;
```

Also See

Negate (Two's-Complement)

Special Applications

None

| (OR)

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 OR instruction performs a 32-bit, bit-wise logical OR 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 OR 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.

- 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

^ (Exclusive-OR), BXORSHIFT, BXOR

Special Applications

None

^ (Exclusive-OR)

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 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 *dest_reg*.

The XOR instruction does not implicitly modify 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 XOR 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.

- 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

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.

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:

$$Out = (((((B_0 \oplus B_1) \oplus B_2) \oplus B_3) \oplus \dots) \oplus B_{n-1})$$

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.

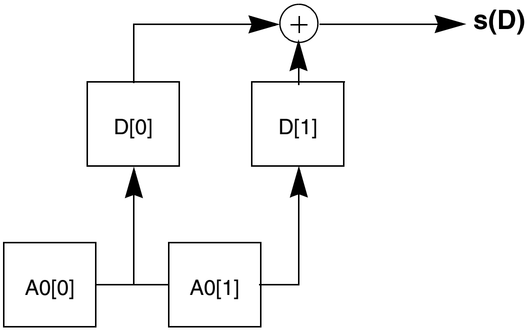


Figure 12-1. Bit-Wise Exclusive-OR Reduction

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.

$$s(D) = (A0[0] \& D[0]) \oplus (A0[1] \& D[1])$$

Modified Type I LFSR (without feedback)

Two instructions support the LSFR with no feedback.

```
Dreg_lo = CC = BXORSHIFT(A0, dreg)
Dreg_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.

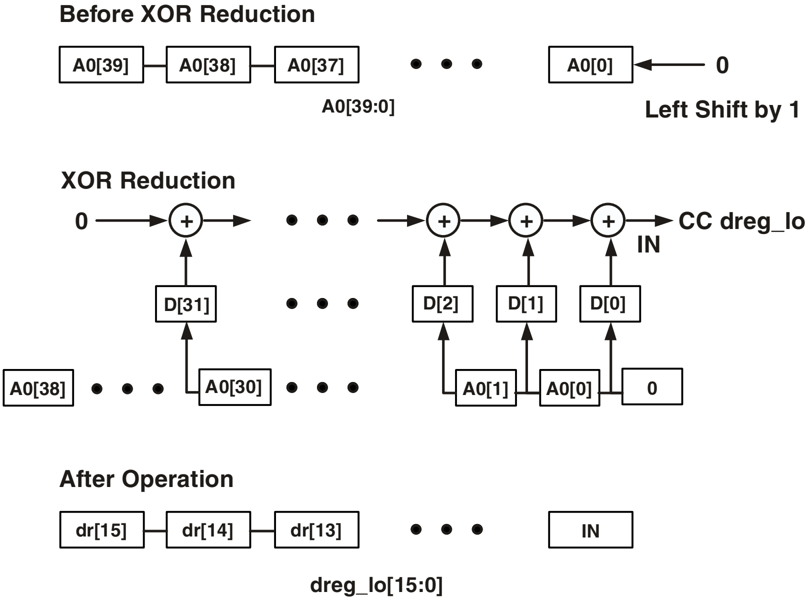


Figure 12-2. A0 Left-Shifted by 1 Followed by XOR Reduction

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.

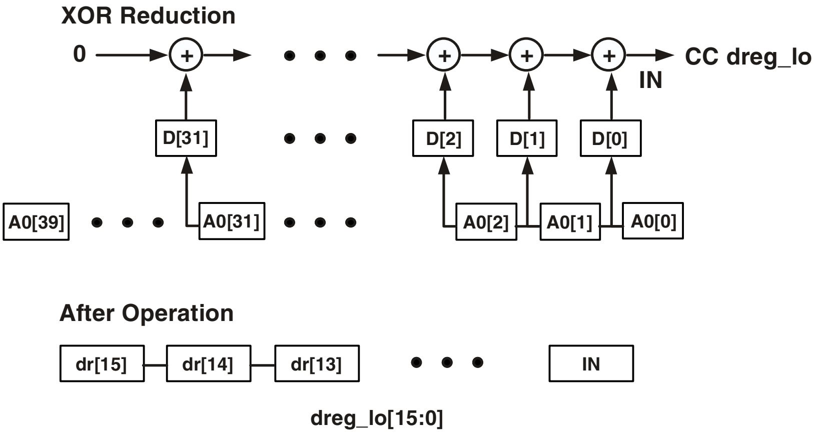


Figure 12-3. XOR of A0, Logical AND with the D-Register

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)
```

The first instruction provides a bit-wise XOR of A0 logically AND'ed with A1. 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 A0 following the operation. This operation is illustrated in Figure 12-4. The CC bit is not modified by this operation.

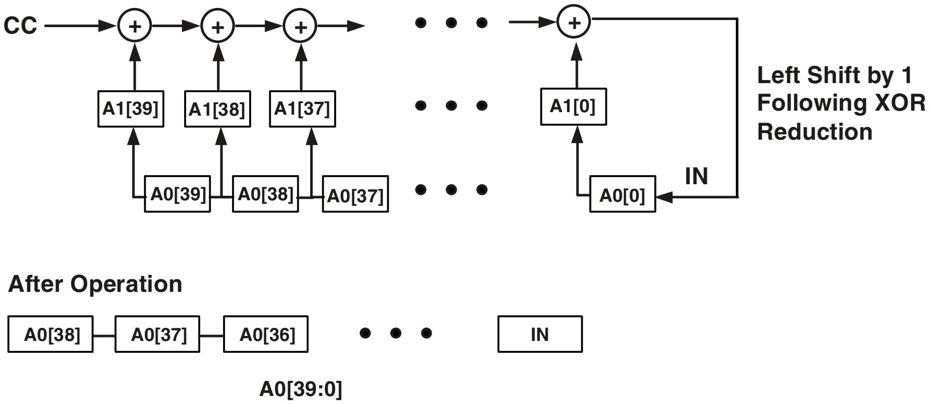


Figure 12-4. XOR of A0 AND A1, Left-Shifted into LSB of A0

The second instruction in this class performs a bit-wise XOR of A0 logically AND'ed with A1. 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 A0 is not modified by this operation. The upper 15 bits of *dreg_lo* are overwritten with zero, and dr[0] = IN.

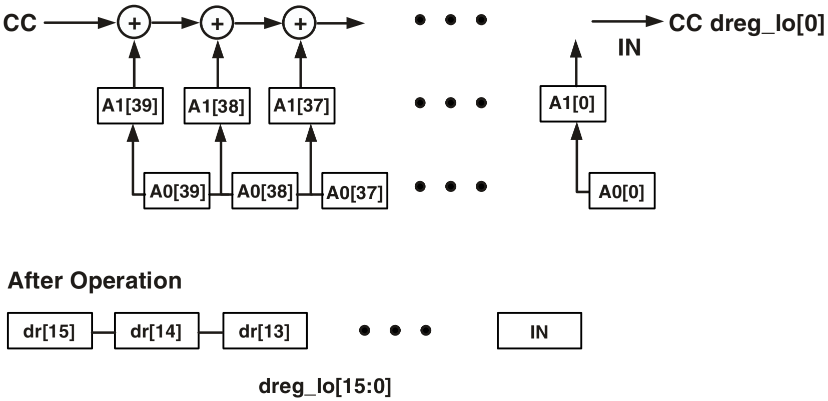


Figure 12-5. XOR of A0 AND A1, to CC Flag and LSB of Dest Register

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

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

```
r0.l = cc = bxorshift (a0, r1) ;
r0.l = cc = bxor (a0, r1) ;
r0.l = cc = bxor (a0, a1, cc) ;
a0 = bxorshift (a0, a1, cc) ;
```

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

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.

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.
- 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

```
bitclr (r2, 3) ;   /* clear bit 3 (the fourth bit from LSB) in
R2 */
```

For example, if R2 contains 0xFFFFFFFF before this instruction, it contains 0xFFFFFFF7 after the instruction.

Also See

BITSET, BITTST, BITTGL

Special Applications

None

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.

- 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

```
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

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.
- 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

```
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

BITTST

General Form

```
CC = BITTST ( register, bit_position )
CC = ! BITTST ( register, bit_position )
```

Syntax

```
CC = BITTST ( Dreg , uimm5 ) ;   /* set CC if bit = 1 (a)*/
CC = ! BITTST ( Dreg , uimm5 ) ;   /* set CC if bit = 0 (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 Test instruction sets or clears the CC bit, based on the bit designated by *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 *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 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 = bittst (r7, 15) ;   /* 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 = ! bittst (r3, 0) ;   /* test bit 0 FALSE in R3 */
```

If R3 contains 0xFFFFFFFF, this instruction clears CC to 0.

Also See

BITCLR, BITSET, BITTGL

Special Applications

None

DEPOSIT

General Form

```
dest_reg = DEPOSIT ( backgnd_reg, foregnd_reg )
dest_reg = DEPOSIT ( backgnd_reg, foregnd_reg ) (X)
```

Syntax

```
Dreg = DEPOSIT ( Dreg, Dreg ) ;    /* no extension (b) */
Dreg = DEPOSIT ( Dreg, Dreg ) (X) ;   /* sign-extended (b) */
```

Syntax Terminology

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 *backgnd_reg* with the foreground bit field in the upper half of *foregnd_reg* and saves the result into *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.

Table 13-1. Input Register Bit Field Definitions

	31................24	23................16	15..................8	7....................0
backgnd_reg:[1]	bbbb bbbb	bbbb bbbb	bbbb bbbb	bbbb bbbb
foregnd_reg:[2]	nnnn nnnn	nnnn nnnn	xxxp pppp	xxxL LLLL

[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.

- 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:

  backgnd_reg = 0x0000 8123,

  L = 0, and

  p = 16,

  then:

  dest_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.

- 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 = deposit (r4, r3) ;
```

- If
  - R4=0b1111 1111 1111 1111 1111 1111 1111 1111
    where this is the background bit field
  - R3=0b0000 0000 0000 0**000** 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=0b1111 1111 1111 1111 1111 11**00** **0**111 1111

- If

  - R4=0b1111 1111 1111 1111 1111 1111 1111 1111
    where this is the background bit field

  - R3=0b0000 000**0 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 11**01 1111 010**1 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 1010 0101 1**010** 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=0b**0000 0000 0000 0000 0000 0001 0**111 1111

- If
  - `R4=0b1111 1111 1111 1111 1111 1111 1111 1111`
    where this is the background bit field
  - `R3=0b0000 100`**`1 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=0b`**`1111 1111 1111 0101 100`**`1 1111 1111 1111`

Also See

EXTRACT

Special Applications

Video image overlay algorithms

EXTRACT

General Form

```
dest_reg = EXTRACT ( scene_reg, pattern_reg ) (Z)
dest_reg = EXTRACT ( scene_reg, pattern_reg ) (X)
```

Syntax

```
Dreg = EXTRACT ( Dreg, Dreg_lo ) (Z) ;   /* zero-extended (b)*/
Dreg = EXTRACT ( Dreg, Dreg_lo ) (X) ;   /* 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 *scene_reg* into the low-order bits of the *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.

Table 13-2. Input Register Bit Field Definitions

	31...............24	23...............16	15.................8	7...................0
scene_reg:[1]	`ssss ssss`	`ssss ssss`	`ssss ssss`	`ssss ssss`
pattern_reg:[2]			`xxxp pppp`	`xxxL LLLL`

[1] where s = scene bit field (32 bits)

[2] where:
–p = position of pattern bit field LSB in scene_reg (valid range 0 through 31)
–L = length of pattern bit field (valid range 0 through 31)

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.

- ACO 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 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 = extract (r4, r3.l) (z) ;   /* zero-extended*/
```

- If
  - R4=0b1010 0101 1010 0101 1100 0**011** **1**010 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 0000 **0111**

- If
  - R4=0b1010 0101 10**10 0101 110**0 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 (unsigned) instruction produces:

  - R7=0b0000 0000 0000 0000 0000 000**1 0010 1110**

Bit Field Extraction Sign-Extended

```
r7 = extract (r4, r3.l) (x) ;   /* sign-extended*/
```

- If
  - R4=0b1010 0101 1010 0101 1100 0**011 1**010 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**

- If
    - R4=0b1010 0101 10**10 0101 110**0 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 111**1 0010 1110**

Also See

DEPOSIT

Special Applications

Video image pattern recognition and separation algorithms

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.

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

IF	39...........32	31...........24	23...........16	15.............8	7................0
source_1:		xxxx xxxx	xxxx xxxx	xxxx xxxx	xxxx xxxx
source_0:		yyyy yyyy	yyyy yyyy	yyyy yyyy	yyyy yyyy
Accumulator A0:	zzzz zzzz	zzzz zzzz	zzzz zzzz	zzzz zzzz	zzzz zzzz

Table 13-4. A Shift Right Instruction

IF	39...........32	31...........24	23...........16	15.............8	7................0
source_1:[1]		0xxx xxxx	xxxx xxxx	xxxx xxxx	xxxx xxxx
source_0:[2]		0yyy yyyy	yyyy yyyy	yyyy yyyy	yyyy yyyy
Accumulator A0:[3]	yxzz zzzz	zzzz zzzz	zzzz zzzz	zzzz zzzz	zzzz zzzz

[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

IF	39...........32	31...........24	23...........16	15.............8	7...............0
source_1:[1]		xxxx xxxx	xxxx xxxx	xxxx xxxx	xxxx xxx0
source_0:[2]		yyyy yyyy	yyyy yyyy	yyyy yyyy	yyyy yyy0
Accumulator A0:[3]	zzzz zzzz	zzzz zzzz	zzzz zzzz	zzzz zzzz	zzzz zzyx

[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.

Example

```
bitmux (r2, r3, a0) (asr) ;   /* right shift*/
```

- If
  - `R2=0b1010 0101 1010 0101 1100 0011 1010 1010`
  - `R3=0b1100 0011 1010 1010 1010 0101 1010 0101`
  - `A0=0b0000 0000 0000 0000 0000 0000 0000 0000 0000 0111`

  then the Shift Right instruction produces:
  - `R2=0b0101 0010 1101 0010 1110 0001 1101 0101`
  - `R3=0b0110 0001 1101 0101 0101 0010 1101 0010`
  - `A0=0b1000 0000 0000 0000 0000 0000 0000 0000 0000 0001`

```
bitmux (r3, r2, a0) (asl) ;   /* left shift*/
```

- If
  - `R3=0b1010 0101 1010 0101 1100 0011 1010 1010`
  - `R2=0b1100 0011 1010 1010 1010 0101 1010 0101`
  - `A0=0b0000 0000 0000 0000 0000 0000 0000 0000 0000 0111`

  then the Shift Left instruction produces:
  - `R2=0b1000 0111 0101 0101 0100 1011 0100 1010`
  - `R3=0b0100 1011 0100 1011 1000 0111 0101 0100`
  - `A0=0b0000 0000 0000 0000 0000 0000 0000 0000 0001 1111`

Also See

None

Special Applications

Convolutional encoder algorithms

ONES (One's-Population Count)

General Form

```
dest_reg = ONES src_reg
```

Syntax

```
Dreg_lo = ONES Dreg ;   /* (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 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

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.

Add with Shift

General Form

```
dest_pntr = (dest_pntr + src_reg) << 1
dest_pntr = (dest_pntr + src_reg) << 2
dest_reg = (dest_reg + src_reg) << 1
dest_reg = (dest_reg + src_reg) << 2
```

Syntax

Pointer Operations

```
Preg = ( Preg + Preg ) << 1 ;    /* dest_reg = (dest_reg +
src_reg) x 2   (a) */
Preg = ( Preg + Preg ) << 2 ;    /* dest_reg = (dest_reg +
src_reg) x 4   (a) */
```

Data Operations

```
Dreg = (Dreg + Dreg) << 1 ;    /* dest_reg = (dest_reg + src_reg)
x 2   (a) */
Dreg = (Dreg + Dreg) << 2 ;    /* dest_reg = (dest_reg + src_reg)
x 4   (a) */
```

Syntax Terminology

Preg: P5–0

Dreg: R7–0

Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

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.

- 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; cleared if no overflow.
- 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.

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.

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

```
dest_pntr = adder_pntr + ( src_pntr << 1 )

dest_pntr = adder_pntr + ( src_pntr << 2 )
```

Syntax

```
Preg = Preg + ( Preg << 1 ) ;    /* adder_pntr + (src_pntr x 2)
(a) */
Preg = Preg + ( Preg << 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

Parallel Issue

This instruction cannot be issued in parallel with other instructions.

Example

```
p3 = p0+(p3<<1) ;    /* p3 = (p3 * 2) + p0 */
p3 = p0+(p3<<2) ;    /* p3 = (p3 * 4) + p0 */
```

Also See

Add with Shift, Logical Shift, Arithmetic Shift, Add, Multiply 32-Bit Operands

Special Applications

None

Arithmetic Shift

General Form

```
dest_reg >>>= shift_magnitude
dest_reg = src_reg >>> shift_magnitude (opt_sat)
dest_reg = src_reg << shift_magnitude (S)
accumulator = accumulator >>> shift_magnitude
dest_reg = ASHIFT src_reg BY shift_magnitude (opt_sat)
accumulator = ASHIFT accumulator BY shift_magnitude
```

Syntax

Constant Shift Magnitude

```
Dreg >>>= uimm5 ;    /* arithmetic right shift (a) */
Dreg <<= uimm5 ;    /* logical left shift (a) */
Dreg_lo_hi = Dreg_lo_hi >>> uimm4 ;    /* arithmetic right shift 
(b) */
Dreg_lo_hi = Dreg_lo_hi << uimm4 (S) ;    /* arithmetic left 
shift (b) */
Dreg = Dreg >>> uimm5 ;    /* arithmetic right shift (b) */
Dreg = Dreg << uimm5 (S) ;    /* arithmetic left shift (b) */
A0 = A0 >>> uimm5 ;    /* arithmetic right shift (b) */
A0 = A0 << uimm5 ;   /* logical left shift (b) */
A1 = A1 >>> uimm5 ;   /* arithmetic right shift (b) */
A1 = A1 << uimm5 ;    /* logical left shift (b) */
```

Registered Shift Magnitude

```
Dreg >>>= Dreg ;    /* arithmetic right shift (a) */
Dreg <<= Dreg ;   /* logical left shift (a) */
Dreg_lo_hi = ASHIFT Dreg_lo_hi BY Dreg_lo (opt_sat) ;    /* 
arithmetic right shift (b) */
Dreg = ASHIFT Dreg BY Dreg_lo (opt_sat) ;    /* arithmetic right 
shift (b) */
```

```
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.

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.

Table 14-1. Arithmetic Shifts

Syntax	Description
“>>>=”	The value in dest_reg is right-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 result in either 0x00000000 (when the input value is positive) or 0xFFFFFFFF (when the input value is negative). Only right shifting is supported in this syntax; there is no equivalent “<<<=” arithmetic left shift syntax. However, logical left shift is supported. See the Logical Shift instruction.
“>>>”, “<<”, and “ASHIFT”	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 “ASHIFT” versions can shift 32-bit Dreg and 40-bit Accumulator registers by up to –32 through +31 places.

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.

- 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; cleared if no overflow.
- VS 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.

- AZ is set if result is zero; cleared if nonzero.
- AN is set if result is negative; cleared if non-negative.
- AV0 is set if result is zero; cleared if nonzero.
- AV0S is set if AV0 is set; unaffected otherwise.
- All other flags are unaffected.

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

```
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 */
```

```
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.l (s) ;    /* shift, word, saturated */
A0 = ashift A0 by r7.l ;    /* shift, Accumulator */
A1 = ashift A1 by r7.l ;    /* 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

Logical Shift

General Form

```
dest_pntr = src_pntr >> 1
dest_pntr = src_pntr >> 2
dest_pntr = src_pntr << 1
dest_pntr = src_pntr << 2
dest_reg >>= shift_magnitude
dest_reg <<= shift_magnitude
dest_reg = src_reg >> shift_magnitude
dest_reg = src_reg << shift_magnitude
dest_reg = LSHIFT src_reg BY shift_magnitude
```

Syntax

Pointer Shift, Fixed Magnitude

```
Preg = Preg >> 1 ;    /* right shift by 1 bit (a) */
Preg = Preg >> 2 ;    /* right shift by 2 bit (a) */
Preg = Preg << 1 ;    /* left shift by 1 bit (a) */
Preg = Preg << 2 ;    /* left shift by 2 bit (a) */
```

Data Shift, Constant Shift Magnitude

```
Dreg >>= uimm5 ;    /* right shift (a) */
Dreg <<= uimm5 ;    /* left shift (a) */
Dreg_lo_hi = Dreg_lo_hi >> uimm4 ;    /* right shift (b) */
Dreg_lo_hi = Dreg_lo_hi << uimm4 ;    /* left shift (b) */
Dreg = Dreg >> uimm5 ;    /* right shift (b) */
Dreg = Dreg << uimm5 ;    /* left shift (b) */
A0 = A0 >> uimm5 ;    /* right shift (b) */
A0 = A0 << uimm5 ;   /* left shift (b) */
A1 = A1 << uimm5 ;    /* left shift (b) */
A1 = A1 >> uimm5 ;   /* right shift (b) */
```

Data Shift, Registered Shift Magnitude

```
Dreg >>= Dreg ;   /* right shift (a) */
Dreg <<= Dreg ;   /* left shift (a) */
Dreg_lo_hi = LSHIFT Dreg_lo_hi BY Dreg_lo ;   /* (b) */
Dreg = LSHIFT Dreg BY Dreg_lo ;   /* (b) */
A0 = LSHIFT A0 BY Dreg_lo ;   /* (b) */
A1 = LSHIFT A1 BY Dreg_lo ;   /* (b) */
```

Syntax Terminology

Dreg: R7–0

Dreg_lo: R7–0.L

Dreg_lo_hi: R7–0.L, R7–0.H

Preg: P5–0

uimm4: 4-bit unsigned field, with a range of 0 through 15

uimm5: 5-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.

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

Syntax	Description
“>>=” and “<<=”	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.
“>>”, “<<”, and “LSHIFT”	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.

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.

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.

The versions of this instruction that send results to an Accumulator A0 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.
- AV0 is cleared.
- All other flags are unaffected.

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 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

```
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.l = r0.l >> 4 ;    /* data right shift, half-word register */
r3.l = r0.h >> 4 ;    /* same as above; half-word register combi-
nations are arbitrary */
r3.h = r0.l << 12 ;    /* data left shift, half-word register */
r3.h = r0.h << 14 ;    /* same as above; half-word register com-
binations 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.l = lshift r0.l by r2.l ;    /* shift direction controlled by
sign of R2.L */
r3.h = lshift r0.l by r2.l ;
a0 = lshift a0 by r7.l ;
a1 = lshift a1 by r7.l ;
   /* If r0.h = -64 (or 0xFFC0), then performing . . . */
r3.h = r0.h >> 4 ;   /* . . . produces r3.h = 0x0FFC (or 4092),
losing the sign */
```

Also See

Arithmetic Shift, ROT (Rotate), Shift with Add, Vector Arithmetic Shift, Vector Logical Shift

Special Applications

None

ROT (Rotate)

General Form

```
dest_reg = ROT src_reg BY rotate_magnitude
accumulator_new = ROT accumulator_old BY rotate_magnitude
```

Syntax

Constant Rotate Magnitude

```
Dreg = ROT Dreg BY imm6 ;    /* (b) */
A0 = ROT A0 BY imm6 ;    /* (b) */
A1 = ROT A1 BY imm6 ;    /* (b) */
```

Registered Rotate Magnitude

```
Dreg = ROT Dreg BY Dreg_lo ;    /* (b) */
A0 = ROT A0 BY Dreg_lo ;    /* (b) */
A1 = ROT A1 BY Dreg_lo ;    */ (b) */
```

Syntax Terminology

Dreg: R7–0

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 CC bit a specified distance and direction. The CC bit is in the rotate chain. Consequently, the first value rotated into the register is the initial value of the 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.

If	31	0
D-register:	`1010 1111 0000 0000 0000 0000 0001 1010`	
CC bit:	N (“1” or “0”)	
Rotate left 1 bit	31	0
D-register:	`0101 1110 0000 0000 0000 0000 0011 010N`	
CC bit:	1	
Rotate left 1 bit again	31	0
D-register:	`1011 1100 0000 0000 0000 0000 0110 10N1`	
CC bit:	0	
If	31	0
D-register:	`1010 1111 0000 0000 0000 0000 0001 1010`	
CC bit:	N (“1” or “0”)	
Rotate right 1 bit	31	0
D-register:	`N101 0111 1000 0000 0000 0000 0000 1101`	
CC bit:	0	
Rotate right 1 bit again	31	0
D-register:	`0N10 1011 1100 0000 0000 0000 0000 0110`	
CC bit:	1	

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.

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

```
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 ;
```

Also See

Arithmetic Shift, Logical Shift

Special Applications

None

15 ARITHMETIC OPERATIONS

Instruction Summary

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.

ABS

General Form

```
dest_reg = ABS src_reg
```

Syntax

```
A0 = ABS A0 ;     /* (b) */
A0 = ABS A1 ;     /* (b) */
A1 = ABS A0 ;     /* (b) */
A1 = ABS A1 ;     /* (b) */
A1 = ABS A1, A0 = ABS A0 ;     /* (b) */
Dreg = ABS Dreg ;     /* (b) */
```

Syntax Terminology

Dreg: R7–0

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

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.
- AV0 is set if result overflows and the *dest_reg* is A0; cleared if no overflow.
- AV0S is set if AV0 is set; unaffected otherwise.

- 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

```
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

Add

General Form

```
dest_reg = src_reg_1 + src_reg_2
```

Syntax

Pointer Registers — 32-Bit Operands, 32-Bit Result

```
Preg = Preg + Preg ;    /* (a) */
```

Data Registers — 32-Bit Operands, 32-bit Result

```
Dreg = Dreg + Dreg ;    /* no saturation support but shorter 
instruction length (a) */
Dreg = Dreg + Dreg (sat_flag) ;   /* saturation optionally sup-
ported, but at the cost of longer instruction length (b) */
```

Data Registers — 16-Bit Operands, 16-Bit Result

```
Dreg_lo_hi = Dreg_lo_hi + Dreg_lo_hi (sat_flag) ;    /* (b) */
```

Syntax Terminology

Preg: P5–0, SP, FP

Dreg: R7–0

Dreg_lo_hi: R7–0.L, R7–0.H

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.

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.

Flags Affected

D-register versions of this instruction 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.
- AC0 is set if the operation generates a carry; cleared if no carry.
- V is set if result overflows; cleared if no overflow.
- 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.

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.

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

Add/Subtract – Prescale Down

General Form

```
dest_reg = src_reg_0 + src_reg_1 (RND20)
dest_reg = src_reg_0 - src_reg_1 (RND20)
```

Syntax

```
Dreg_lo_hi = Dreg + Dreg (RND20) ; // (b)
Dreg_lo_hi = Dreg - Dreg (RND20) ; // (b)
```

Syntax Terminology

Dreg: R7–0

Dreg_lo_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 *dest_reg*

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 "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 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

```
r1.l = r6+r7(rnd20) ;
r1.l = r6-r7(rnd20) ;
r1.h = r6+r7(rnd20) ;
r1.h = r6-r7(rnd20) ;
```

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

```
dest_reg = src_reg_0 + src_reg_1 (RND12)
dest_reg = src_reg_0 - src_reg_1 (RND12)
```

Syntax

```
Dreg_lo_hi = Dreg + Dreg (RND12) ; // (b)
Dreg_lo_hi = Dreg - Dreg (RND12) ; // (b)
```

Syntax Terminology

Dreg: R7-0

Dreg_lo_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 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 *dest_reg*

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) ;
```

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.

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 (*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.

Flags Affected

D-register versions of this instruction 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.
- AC0 is set if the operation generates a carry; cleared if no carry.
- V is set if result overflows; cleared if no overflow.
- 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.

The P-register and I-register versions of this instruction do not affect any flags.

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

```
r0 += 40 ;
p5 += -4 ;    /* decrement by adding a negative value */
i0 += 2 ;
i1 += 4 ;
```

Also See

Subtract Immediate

Special Applications

None

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 sub-
tract 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.

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 binal point of the dividend, and
- NR represent the number of bits to the right of the binal point of the dividend (numerator);
- DL represent the number of bits to the left of the binal point of the divisor, and
- DR represent the number of bits to the right of the binal point of the divisor (denominator).

Then the quotient has NL – DL + 1 bits to the left of the binal point and NR – DR – 1 bits to the right of the binal point. See the following example.

Dividend (numerator)	`BBBB B .`	`BBB BBBB BBBB BBBB BBBB BBBB BBBB`
	`NL bits`	`NR bits`
Divisor (denominator)	`BB .`	`BB BBBB BBBB BBBB`
	`DL bits`	`DR bits`
Quotient	`BBBB .`	`BBBB BBBB BBBB`
	`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 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.

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 divi-
sion */
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 ;
```

```
r0 = r0.l (x) ;   /* Sign extend the 16-bit quotient to 32bits.
*/
/* r0 contains the quotient (70/5 = 14). */
```

Also See

LSETUP, LOOP, Multiply 32-Bit Operands

Special Applications

None

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 sam-
ple (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.

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.

Example

```
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.

```
r5.l = expadj (r4.l, r2.l) ;
```

- Assume `R4.L` = 0x0765 and `R2.L` = 12. Then `R5.L` becomes 4.
- Assume `R4.L` = 0xC765 and `R2.L` = 12. Then `R5.L` becomes 1.

```
r5.l = expadj (r4.h, r2.l) ;
```

- Assume `R4.H` = 0x0765 and `R2.L` = 12. Then `R5.L` becomes 4.
- Assume `R4.H` = 0xC765 and `R2.L` = 12. Then `R5.L` becomes 1.

```
r5.l = expadj (r4, r2.l)(v) ;
```

- Assume `R4.L` = 0x0765, `R4.H` = 0xFF74 and `R2.L` = 12. Then `R5.L` becomes 4.
- Assume `R4.L` = 0x0765, `R4.H` = 0xE722 and `R2.L` = 12. Then `R5.L` becomes 2.

Also See

SIGNBITS

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.

MAX

General Form

```
dest_reg = MAX ( src_reg_0, src_reg_1 )
```

Syntax

```
Dreg = MAX ( Dreg , Dreg ) ;    /* 32-bit operands (b) */
```

Syntax Terminology

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 *src_reg_1* from *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 *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.

- 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 = 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

MIN

General Form

```
dest_reg = MIN ( src_reg_0, src_reg_1 )
```

Syntax

```
Dreg = MIN ( Dreg , Dreg ) ;    /* 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 *dest_reg*. (The minimum value of the source registers is the value closest to – ∞.) The operation subtracts *src_reg_1* from *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 *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.

- 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 = 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

Modify – Decrement

General Form

```
dest_reg -= src_reg
```

Syntax

40-Bit Accumulators

```
A0 -= A1 ;    /* dest_reg_new = dest_reg_old - src_reg, saturate
the result at 40 bits (b) */
A0 -= A1 (W32) ;   /* dest_reg_new = dest_reg_old - src_reg, dec-
rement and saturate the result at 32 bits, sign extended (b) */
```

32-Bit Registers

```
Preg -= Preg ;    /* dest_reg_new = dest_reg_old - src_reg (a) */
Ireg -= Mreg ;    /* dest_reg_new = dest_reg_old - src_reg (a) */
```

Syntax Terminology

Preg: P5–0, SP, FP

Ireg: I3–0

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.

See "Saturation" on page 1-17 for a description of saturation 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.

Flags Affected

The Accumulator versions of this instruction affect 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.
- `AC0` is set if the operation generates a carry; cleared if no carry.
- `AV0` is set if result saturates; cleared if no saturation.
- `AV0S` is set if `AV0` 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.

The P-register and I-register versions do not affect any 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

```
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.

Modify – Increment

General Form

```
dest_reg += src_reg
dest_reg = ( src_reg_0 += src_reg_1 )
```

Syntax

40-Bit Accumulators

```
A0 += A1 ;   /* dest_reg_new = dest_reg_old + src_reg, saturate
the result at 40 bits (b) */
A0 += A1 (W32) ;   /* dest_reg_new = dest_reg_old + src_reg,
signed saturate the result at 32 bits, sign extended (b) */
```

32-Bit Registers

```
Preg += Preg (BREV) ;    /* dest_reg_new = dest_reg_old +
src_reg, bit reversed carry, only (a) */
Ireg += Mreg (opt_brev) ;    /* dest_reg_new = dest_reg_old +
src_reg, optional bit reverse (a) */
Dreg = ( A0 += A1 ) ;   /* increment 40-bit A0 by A1 with satura-
tion 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 ) ;    /* Increment 40-bit A0 by A1 with
saturation at 40 bits, then extract the result into a half regis-
ter. The extraction step involves first rounding the 40-bit
```

```
result at bit 16 (according to the RND_MOD bit in the ASTAT reg-
ister), 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.

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.

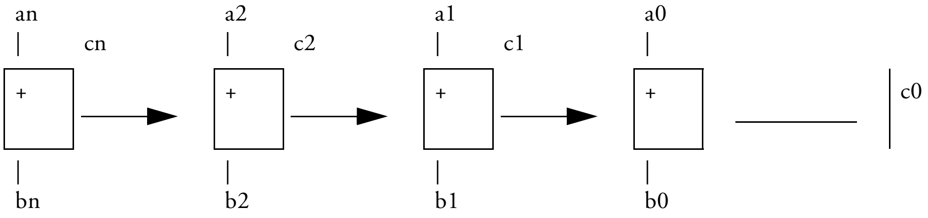
The circular address buffer registers (Index, Length, and Base) are not initialized automatically by Reset. Traditionally, user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes them later, if needed.

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



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.
- AC0 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.
- 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.
- 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.
- AC0 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.
- AV0 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 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

```
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) ;
```

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.

Multiply 16-Bit Operands

General Form

```
dest_reg = src_reg_0 * src_reg_1 (opt_mode)
```

Syntax

Multiply-And-Accumulate Unit 0 (MAC0)

```
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (opt_mode_1) ;   /* 16-bit
result into the destination lower half-word register (b) */
Dreg_even = Dreg_lo_hi * Dreg_lo_hi (opt_mode_2) ;   /* 32-bit
result (b) */
```

Multiply-And-Accumulate Unit 1 (MAC1)

```
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (opt_mode_1) ;   /* 16-bit
result into the destination upper half-word register (b) */
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (opt_mode_2) ;   /* 32-bit
result (b) */
```

Syntax Terminology

Dreg: R7–0

Dreg_lo: R7–0.L

Dreg_hi: R7–0.H

Dreg_lo_hi: R7–0.L, R7–0.H

opt_mode_1: 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. When used together, the option flags must be enclosed in one set of parentheses and separated by a comma. Example: (M, IS) *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).

Table 15-2. Multiply 16-Bit Operands Options

Option	Description for Register Half Destination	Description for 32-Bit Register Destination
Default	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).	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.
(FU)	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).	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.
(IS)	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).	Signed integer. Multiply 16.0 * 16.0 to produce 32.0 results. No shift correction. Saturate integer results between minimum -2^{31} and maximum $2^{31}-1$.
(IU)	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).	Not applicable. Use (IS).

Table 15-2. Multiply 16-Bit Operands Options (Cont'd)

Option	Description for Register Half Destination	Description for 32-Bit Register Destination
(T)	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 $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).	Not applicable. Truncation is meaningless for 32-bit register destinations.
(TFU)	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 $1\text{-}2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).	Not applicable.
(S2RND)	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 $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).	Not applicable.

Table 15-2. Multiply 16-Bit Operands Options (Cont'd)

Option	Description for Register Half Destination	Description for 32-Bit Register Destination
(ISS2)	Signed integer with scaling. Multiply 16.0 * 16.0 to produce 32.0 results. No shift correction. Extract 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).	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.
(IH)	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).	Not applicable.
(M)	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.	

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 frac-
tions. */
r6=r3.h*r4.h ;   /* MAC0. Signed fraction operands, results saved
as 32 bits. */
```

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

Multiply 32-Bit Operands

General Form

```
dest_reg *= multiplier_register
```

Syntax

```
Dreg *= Dreg ; /* 32 x 32 integer multiply (a) */
```

Syntax Terminology

Dreg: 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 (*dest_reg* and *multiplier_register)* and saves the product in *dest_reg*. The instruction mimics multiplication in the C language and effectively performs *Dreg1 = (Dreg1 * Dreg2)* modulo 2^{32}. Since the integer multiply is modulo 2^{32}, the result always fits in a 32-bit *dest_reg*, and overflows are possible but not detected. The overflow flag in the ASTAT register is never set.

Users are required to limit input numbers to ensure that the resulting product does not exceed the 32-bit *dest_reg* capacity. If overflow notification is required, users should write their own multiplication macro with that capability.

Accumulators A0 and A1 are unchanged by this instruction.

The Multiply 32-Bit Operands instruction does not implicitly modify the number in *multiplier_register*.

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 *= 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

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 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.

Table 15-3. Multiply and Multiply-Accumulate to Accumulator Options

Option	Description
Default	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 $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF).
(FU)	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.
(IS)	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 $2^{39}\text{-}1$ (or, expressed in hex, between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF).
(W32)	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 $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0xFF 8000 0000 and maximum 0x00 7FFF FFFF).
(M)	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.

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

```
a0=r3.h*r2.h ;   /* MAC0, only. Both operands are signed frac-
tions. Load the product into A0. */
a1+=r6.h*r4.l (fu) ;   /* MAC1, only. Both operands are unsigned
fractions. Accumulate into A1 */
```

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.

Multiply and Multiply-Accumulate to Half-Register

General Form

```
dest_reg_half = (accumulator = src_reg_0 * src_reg_1) (opt_mode)
dest_reg_half = (accumulator += src_reg_0 * src_reg_1) (opt_mode)
dest_reg_half = (accumulator -= src_reg_0 * src_reg_1) (opt_mode)
```

Syntax

Multiply-And-Accumulate Unit 0 (MAC0)

```
Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (opt_mode) ;    /* multiply and store (b) */
Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (opt_mode) ; /* multiply and add (b) */
Dreg_lo = (A0 -= Dreg_lo_hi * Dreg_lo_hi) (opt_mode) ;    /* multiply and subtract (b) */
```

Multiply-And-Accumulate Unit 1 (MAC1)

```
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (opt_mode) ;    /* multiply and store (b) */
Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (opt_mode) ;    /* multiply and add (b) */
Dreg_hi = (A1 -= Dreg_lo_hi * Dreg_lo_hi) (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

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.

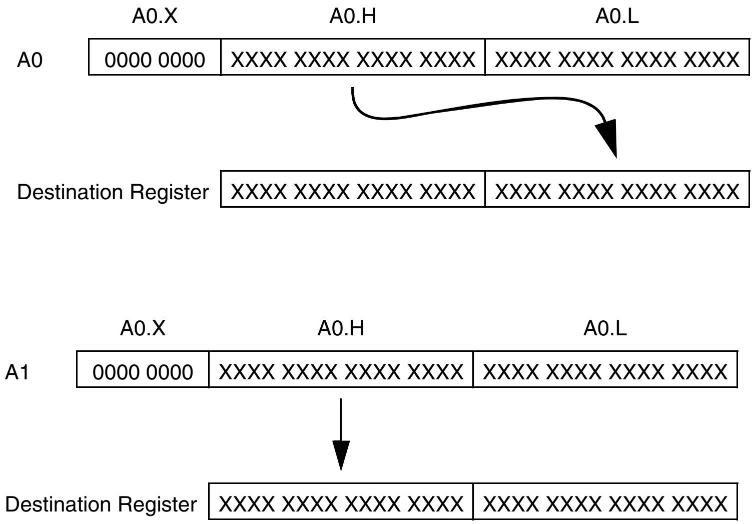


Figure 15-1. Result to Destination Register (Default and (FU) Options)

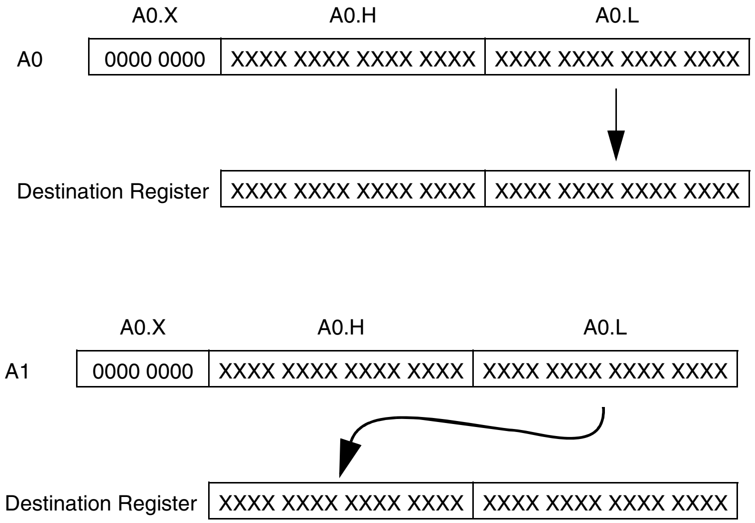


Figure 15-2. Result to Destination Register ((IS) and (IU) Options)

See "Rounding and Truncating" on page 1-19 for a description of rounding behavior.

Options

The Multiply and Multiply-Accumulate to Half-Register instruction supports operand and Accumulator copy options.

The options are listed in Table 15-4.

Table 15-4. Multiply and Multiply-Accumulate to Half-Register Options

Option	Description
Default	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).
(FU)	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).

Table 15-4. Multiply and Multiply-Accumulate to Half-Register Options (Cont'd)

Option	Description
(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).

Table 15-4. Multiply and Multiply-Accumulate to Half-Register Options (Cont'd)

Option	Description
(TFU)	Unsigned fraction with truncation. 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 0xFF FFFF 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).
(S2RND)	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 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).
(ISS2)	Signed integer with scaling. 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).

Table 15-4. Multiply and Multiply-Accumulate to Half-Register Options (Cont'd)

Option	Description
(IH)	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).
(M)	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.

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.

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.

Example

```
r3.l=(a0=r3.h*r2.h) ;   /* MAC0, 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.

Multiply and Multiply-Accumulate to Data Register

General Form

```
dest_reg = (accumulator = src_reg_0 * src_reg_1) (opt_mode)
dest_reg = (accumulator += src_reg_0 * src_reg_1) (opt_mode)
dest_reg = (accumulator -= src_reg_0 * src_reg_1) (opt_mode)
```

Syntax

Multiply-And-Accumulate Unit 0 (MAC0)

```
Dreg_even = (A0 = Dreg_lo_hi * Dreg_lo_hi) (opt_mode) ;   /* multiply and store (b) */
Dreg_even = (A0 += Dreg_lo_hi * Dreg_lo_hi) (opt_mode) ;   /* multiply and add (b) */
Dreg_even = (A0 -= Dreg_lo_hi * Dreg_lo_hi) (opt_mode) ;   /* multiply and subtract (b) */
```

Multiply-And-Accumulate Unit 1 (MAC1)

```
Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (opt_mode) ;   /* multiply and store (b) */
Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (opt_mode) ;   /* multiply and add (b) */
Dreg_odd = (A1 -= Dreg_lo_hi * Dreg_lo_hi) (opt_mode) ;   /* multiply and subtract (b) */
```

Syntax Terminology

Dreg_lo_hi: R7-0.L, R7-0.H

Dreg_even: R0, R2, R4, R6

Dreg_odd: R1, R3, R5, R7

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.

Table 15-5. Multiply and Multiply-Accumulate to Data Register Options

Option	Description
Default	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).
(FU)	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).
(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. 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).
(S2RND)	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).

Table 15-5. Multiply and Multiply-Accumulate to Data Register Options (Cont'd)

Option	Description
(ISS2)	Signed integer with scaling. 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. To extract, shift the Accumulator contents one place to the left (multiply x 2), saturate the result for 32.0 format, 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).
(M)	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.

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.

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.

Example

```
r4=(a0=r3.h*r2.h) ;   /* MAC0, only. Both operands are signed
fractions. Load the product into A0, then into r4. */
r3=(a1+=r6.h*r4.l) (fu) ;   /* MAC1, only. Both operands are
unsigned fractions. Add the product into A1, then into r3. */
```

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.

Negate (Two’s-Complement)

General Form

```
dest_reg = - src_reg
dest_accumulator = - src_accumulator
```

Syntax

```
Dreg = - Dreg ;    /* (a) */
Dreg = - Dreg (sat_flag) ;    /* (b) */
A0 = - A0 ;    /* (b) */
A0 = - A1 ;    /* (b) */
A1 = - A0 ;    /* (b) */
A1 = - A1 ;    /* (b) */
A1 = - A1, A0 = - A0 ;   /* negate both Accumulators simulta-
neously in one 32-bit length instruction (b) */
```

Syntax Terminology

Dreg: R7–0

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.

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.

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.

- 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 ;
```

Also See

Vector Negate (Two’s-Complement)

Special Applications

None

RND (Round to Half-Word)

General Form

```
dest_reg = src_reg (RND)
```

Syntax

```
Dreg_lo_hi =Dreg (RND) ;    /* round and saturate the source to
16 bits. (b) */
```

Syntax Terminology

Dreg: R7- 0

Dreg_lo_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 RND_MOD bit in the 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.

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

```
/* If r6 = 0xFFFC FFFF, then rounding to 16-bits with . . . */
r1.l = r6 (rnd) ;   // . . . produces r1.l = 0xFFFD
// If r7 = 0x0001 8000, then rounding . . .
r1.h = r7 (rnd) ;   // . . . produces r1.h = 0x0002
```

Also See

Add, Add/Subtract – Prescale Up, Add/Subtract – Prescale Down

Special Applications

None

Saturate

General Form

```
dest_reg = src_reg (S)
```

Syntax

```
A0 = A0 (S) ;    /* (b) */
A1 = A1 (S) ;    /* (b) */
A1 = A1 (S), A0 = A0 (S) ;    /* signed saturate both Accumula-
tors at the 32-bit boundary (b) */
```

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.

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 set if result is negative; cleared if non-negative. In the case of two simultaneous operations, AN represents the logical "OR" of the two.
- AV0 is set if result saturates and the *dest_reg* is A0; cleared if no overflow.
- AV0S is set if AV0 is set; unaffected otherwise.
- AV1 is set if result saturates 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

```
a0 = a0 (s) ;
a1 = a1 (s) ;
a1 = a1 (s), a0 = a0 (s) ;
```

Also See

Subtract (saturate options), Add (saturate options)

Special Applications

None

SIGNBITS

General Form

```
dest_reg = SIGNBITS sample_register
```

Syntax

```
Dreg_lo = SIGNBITS Dreg ;    /* 32-bit sample (b) */
Dreg_lo = SIGNBITS Dreg_lo_hi ;    /* 16-bit sample (b) */
Dreg_lo = SIGNBITS A0 ;    /* 40-bit sample (b) */
Dreg_lo = SIGNBITS A1 ;    /* 40-bit sample (b) */
```

Syntax Terminology

Dreg: R7–0

Dreg_lo: R7–0.L

Dreg_lo_hi: R7–0.L, R7–0.H

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.

- 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.MMM MMMM MMMM MMMM
32-bit:	S.MMM MMMM MMMM MMMM MMMM MMMM MMMM MMMM
40-bit:	SSSS SSSS S.MMM 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

```
r2.l = signbits r7 ;
r1.l = signbits r5.l ;
r0.l = signbits r4.h ;
r6.l = signbits a0 ;
r5.l = signbits a1 ;
```

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.

Subtract

General Form

```
dest_reg = src_reg_1 - src_reg_2
```

Syntax

32-Bit Operands, 32-Bit Result

```
Dreg = Dreg - Dreg ;    /* no saturation support but shorter
instruction length (a) */
Dreg = Dreg - Dreg (sat_flag) ;   /* saturation optionally sup-
ported, but at the cost of longer instruction length (b) */
```

16-Bit Operands, 16-Bit Result

```
Dreg_lo_hi = Dreg_lo_hi – Dreg_lo_hi (sat_flag) ; /* (b) */
```

Syntax Terminology

Dreg: R7–0

Dreg_lo_hi: R7–0.L, R7–0.H

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.

Functional Description

The Subtract instruction subtracts *src_reg_2* from *src_reg_1* and places the result in a destination register.

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.

- 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.l - r7.l (ns) ;
r4.l = r0.l - r7.h (s) ;    /* saturate the result */
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 – Decrement, Vector Add / Subtract

Special Applications

None

Subtract Immediate

General Form

```
register -= constant
```

Syntax

```
Ireg -= 2 ;    /* decrement Ireg by 2, half-word address pointer
increment (a) */
Ireg -= 4 ;    /* word address pointer decrement (a) */
```

Syntax Terminology

Ireg: I3–0

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 `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.

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

```
i0 -= 4 ;
i2 -= 2 ;
```

Also See

Add Immediate, Subtract

Special Applications

None

16 EXTERNAL EVENT MANAGEMENT

Instruction Summary

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.

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

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

```
idle ;
```

Also See

System Synchronize

Special Applications

None

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.

`CCYNC` is typically used after core MMR writes to prevent imprecise behavior.

Flags Affected

None

Required Mode

User & Supervisor

Parallel Issue

The Core Synchronize instruction cannot be issued in parallel with other instructions.

Example

Consider the following example code sequence.

```
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.

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.

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.

```
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

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...
```

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.

Example

```
emuexcpt ;
```

Also See

RAISE (Force Interrupt / Reset)

Special Applications

None

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.

Parallel Issue

The Disable Interrupts instruction cannot be issued in parallel with other instructions.

Example

```
cli r3 ;
```

Also See

Enable Interrupts

Special Applications

This instruction is often issued immediately before an `IDLE` instruction.

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.

Example

```
sti r3 ;
```

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.

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

uimm4	Event
0	<reserved>
1	RST
2	NMI

Table 16-1. uimm4 Arguments and Events (Cont'd)

uimm4	Event
3	<reserved>
4	<reserved>
5	IVHW
6	IVTMR
7	IVG7
8	IVG8
9	IVG9
10	IVG10
11	IVG11
12	IVG12
13	IVG13
14	IVG14
15	IVG15

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

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

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

Parallel Issue

The Force Exception instruction cannot be issued in parallel with other instructions.

Example

```
excpt 4 ;
```

Also See

None

Special Applications

None

Test and Set Byte (Atomic)

General Form

```
TESTSET
```

Syntax

```
TESTSET ( Preg ) ;    /* (a) */
```

Syntax Terminology

Preg: `P5–0` (`SP` and `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 `CC` bit. If the byte is originally nonzero the instruction clears the `CC` bit. The sequence of this memory transaction is *atomic.*

`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 `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 `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 `TESTSET` to proceed.

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

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.

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

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

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 FLUSHINV 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.

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.

Flags Affected

None

Required Mode

User & Supervisor

Parallel Issue

This instruction cannot be issued in parallel with other instructions.

Example

```
prefetch [ p2 ] ;
prefetch [ p0 ++ ] ;
```

Also See

None

Special Applications

None

FLUSH

General Form

```
FLUSH
```

Syntax

```
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.

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

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.

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

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.

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

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

```
dest_reg = ALIGN8 ( src_reg_1, src_reg_0 )
dest_reg = ALIGN16 (src_reg_1, src_reg_0 )
dest_reg = ALIGN24 (src_reg_1, src_reg_0 )
```

Syntax

```
Dreg = ALIGN8 ( Dreg, Dreg ) ;    /* overlay 1 byte (b) */
Dreg = ALIGN16 ( Dreg, Dreg ) ;    /* overlay 2 bytes (b) */
Dreg = ALIGN24 ( Dreg, Dreg ) ;    /* overlay 3 bytes (b) */
```

Syntax Terminology

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 *dest_reg* = PACK (*Dreg_lo*, *Dreg_hi*) syntax.

Use the Byte Align instruction to align data bytes for subsequent single-instruction, multiple-data (SIMD) instructions.

Table 18-1. Byte Alignment Options

	src_reg_1				**src_reg_0**			
	byte7	**byte6**	**byte5**	**byte4**	**byte3**	**byte2**	**byte1**	**byte0**
dest_reg for ALIGN8:	▨	▨	▨	byte4	byte3	byte2	byte1	▨
dest_reg for ALIGN16:	▨	▨	byte5	byte4	byte3	byte2	▨	▨
dest_reg for ALIGN24:	▨	byte6	byte5	byte4	byte3	▨	▨	▨

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.

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

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 DISALIGNEXCPT instruction. Also, any modifications performed to the I-register by a parallel instruction are not affected by the DISALIGNEXCPT instruction.

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

```
disalgnexcpt || r1 = [i0++] || r3 = [i1++] ;   /* three instruc-
tions 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 */
```

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.

BYTEOP3P (Dual 16-Bit Add / Clip)

General Form

```
dest_reg = BYTEOP3P ( src_reg_0, src_reg_1 ) (LO)
dest_reg = BYTEOP3P ( src_reg_0, src_reg_1 ) (HI)
dest_reg = BYTEOP3P ( src_reg_0, src_reg_1 ) (LO, R)
dest_reg = BYTEOP3P ( src_reg_0, src_reg_1 ) (HI, R)
```

Syntax

```
/* forward byte order operands */
Dreg = BYTEOP3P (Dreg_pair, Dreg_pair) (LO) ;   /* sum into low 
bytes (b) */
Dreg = BYTEOP3P (Dreg_pair, Dreg_pair) (HI) ;   /* sum into high 
bytes (b) */
/* reverse byte order operands */
Dreg = BYTEOP3P (Dreg_pair, Dreg_pair) (LO, R) ;   /* sum into 
low bytes (b) */
Dreg = BYTEOP3P (Dreg_pair, Dreg_pair) (HI, R) ;   /* sum into 
high bytes (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 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:

	31...............24	23...............16	15.................8	7...................0
aligned_src_reg_0:	y1		y0	
aligned_src_reg_1:	z3	z2	z1	z0

Table 18-3. The versions that load the result into the lower byte–“(LO)”–produce:

	31...............24	23...............16	15.................8	7...................0
dest_reg:	0 0	y1 + z3 clipped to 8 bits	0 0	y0 + z1 clipped to 8 bits

Table 18-4. And the versions that load the result into the higher byte–“(HI)”–produce:

	31...............24	23...............16	15.................8	7...................0
dest_reg:	y1 + z2 clipped to 8 bits	0 0	y0 + z0 clipped to 8 bits	0 0

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

The bytes selected are	**src_reg_pair_HI**				**src_reg_pair_LO**			
Two LSB's of I0 or I1	**byte7**	**byte6**	**byte5**	**byte4**	**byte3**	**byte2**	**byte1**	**byte0**
00b:					byte3	byte2	byte1	byte0
01b:				byte4	byte3	byte2	byte1	
10b:			byte5	byte4	byte3	byte2		
11b:		byte6	byte5	byte4	byte3			

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) 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

The bytes selected are	src_reg_pair_LO				src_reg_pair_HI			
Two LSB’s of I0 or I1	**byte7**	**byte6**	**byte5**	**byte4**	**byte3**	**byte2**	**byte1**	**byte0**
00b:					byte3	byte2	byte1	byte0
01b:				byte4	byte3	byte2	byte1	
10b:			byte5	byte4	byte3	byte2		
11b:		byte6	byte5	byte4	byte3			

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

```
r3 = byteop3p (r1:0, r3:2) (lo) ;
r3 = byteop3p (r1:0, r3:2) (hi) ;
```

```
r3 = byteop3p (r1:0, r3:2) (lo, r) ;
r3 = 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.

Dual 16-Bit Accumulator Extraction with Addition

General Form

```
dest_reg_1 = A1.L + A1.H, dest_reg_0 = A0.L + A0.H
```

Syntax

```
Dreg = A1.L + A1.H, Dreg = A0.L + A0.H ;   /* (b) */
```

Syntax Terminology

Dreg: 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 31through 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

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.l+a1.h, r7=a0.l+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.

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 inTable 18-7 and Table 18-8.

The only valid input source register pairs are R1:0 and R3:2.

Table 18-7. Source Registers Contain

	31...............24	23...............16	15.................8	7...................0
aligned_src_reg_0:	y3	y2	y1	y0
aligned_src_reg_1:	z3	z2	z1	z0

Table 18-8. Destination Registers Receive

	31...............24	23...............16	15.................8	7...................0
aligned_src_reg_0:	y1 + z1		y0 + z0	
aligned_src_reg_1:	y3 + z3		y2 + z2	

The Quad 8-Bit Add 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 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

Table 18-9. I-register Bits and the Byte Alignment

The bytes selected are	src_reg_pair_HI				src_reg_pair_LO			
Two LSB's of I0 or I1	**byte7**	**byte6**	**byte5**	**byte4**	**byte3**	**byte2**	**byte1**	**byte0**
00b:					byte3	byte2	byte1	byte0
01b:				byte4	byte3	byte2	byte1	
10b:			byte5	byte4	byte3	byte2		
11b:		byte6	byte5	byte4	byte3			

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 a source register pair contains the data shown in Table 18-10.

Table 18-10. I-register Bits and the Byte Alignment

The bytes selected are	src_reg_pair_LO				src_reg_pair_HI			
Two LSB's of I0 or I1	**byte7**	**byte6**	**byte5**	**byte4**	**byte3**	**byte2**	**byte1**	**byte0**
00b:					byte3	byte2	byte1	byte0
01b:				byte4	byte3	byte2	byte1	
10b:			byte5	byte4	byte3	byte2		
11b:		byte6	byte5	byte4	byte3			

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.

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)= byteop16p (r3:2,r1:0) ;
(r1,r2)= 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.

BYTEOP1P (Quad 8-Bit Average – Byte)

General Form

```
dest_reg = BYTEOP1P ( src_reg_0, src_reg_1 )
dest_reg = BYTEOP1P ( src_reg_0, src_reg_1 ) (T)
dest_reg = BYTEOP1P ( src_reg_0, src_reg_1 ) (R)
dest_reg = BYTEOP1P ( src_reg_0, src_reg_1 ) (T, R)
```

Syntax

```
/* forward byte order operands */
Dreg = BYTEOP1P (Dreg_pair, Dreg_pair) ;   /* (b) */
Dreg = BYTEOP1P (Dreg_pair, Dreg_pair) (T) ;   /* truncated (b)
*/
/* reverse byte order operands */
Dreg = BYTEOP1P (Dreg_pair, Dreg_pair) (R) ;   /* (b) */
Dreg = BYTEOP1P (Dreg_pair, Dreg_pair) (T, R) ; /* truncated (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 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.

Table 18-11. Source Registers Contain

	31...............24	23...............16	15.................8	7...................0
aligned_src_reg_0:	y3	y2	y1	y0
aligned_src_reg_1:	z3	z2	z1	z0

Table 18-12. Destination Registers Receive

	31...............24	23...............16	15.................8	7...................0
dest_reg:	avg(y3, z3)	avg(y2, z2)	avg(y1, z1)	avg(y0, z0)

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).

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

The bytes selected are	**src_reg_pair_HI**				**src_reg_pair_LO**			
Two LSB's of I0 or I1	**byte7**	**byte6**	**byte5**	**byte4**	**byte3**	**byte2**	**byte1**	**byte0**
00b:					byte3	byte2	byte1	byte0
01b:				byte4	byte3	byte2	byte1	
10b:			byte5	byte4	byte3	byte2		
11b:		byte6	byte5	byte4	byte3			

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

Option	**Description**
Default	Rounds up the arithmetic mean.
(T)	Truncates the arithmetic mean.

Table 18-14. Options for Quad 8-Bit Average – Byte (Cont'd)

Option	Description
(R)	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.
(T, R)	Combines both of the above options.

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 18-15. I-register Bits and the Byte Alignment

The bytes selected are	**src_reg_pair_LO**				**src_reg_pair_HI**			
Two LSB's of I0 or I1	**byte7**	**byte6**	**byte5**	**byte4**	**byte3**	**byte2**	**byte1**	**byte0**
00b:	▨	▨	▨	▨	byte3	byte2	byte1	byte0
01b:	▨	▨	▨	byte4	byte3	byte2	byte1	▨
10b:	▨	▨	byte5	byte4	byte3	byte2	▨	▨
11b:	▨	byte6	byte5	byte4	byte3	▨	▨	▨

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.

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.

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) */
```

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 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

	31...............24	23...............16	15.................8	7...................0
aligned_src_reg_0:	y3	y2	y1	y0
aligned_src_reg_1:	z3	z2	z1	z0

Table 18-17. The versions that load the result into the lower byte – RNDL and TL – produce:

	31...............24	23...............16	15.................8	7...................0
dest_reg:	0 0	avg(y3, y2, z3, z2)	0 0	avg(y1, y0, z1, z0)

In either case, the unused bytes in the destination register are filled with 0x00.

Table 18-18. And the versions that load the result into the higher byte – RNDH and TH – produce:

	31...............24	23...............16	15.................8	7...................0
dest_reg:	avg(y3, y2, z3, z2)	0 0	avg(y1, y0, z1, z0)	0 0

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.

Table 18-19. I-register Bits and the Byte Alignment

The bytes selected are	**src_reg_pair_HI**				**src_reg_pair_LO**			
Two LSB's of I0 or I1	**byte7**	**byte6**	**byte5**	**byte4**	**byte3**	**byte2**	**byte1**	**byte0**
00b:	▨	▨	▨	▨	byte3	byte2	byte1	byte0
01b:	▨	▨	▨	byte4	byte3	byte2	byte1	▨
10b:	▨	▨	byte5	byte4	byte3	byte2	▨	▨
11b:	▨	byte6	byte5	byte4	byte3	▨	▨	▨

Options

The Quad 8-Bit Average – Half-Word instruction supports the following options.

Table 18-20. Options for Quad 8-Bit Average – Half-Word

Option	Description
(RND—)	Rounds up the arithmetic mean.
(T—)	Truncates the arithmetic mean.
(—L)	Loads the results into the lower byte of each destination half-word.
(—H)	Loads the results into the higher byte of each destination half-word.
(,R)	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.

When used together, the order of the options in the syntax makes no difference.

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

The bytes selected are	**src_reg_pair_LO**				**src_reg_pair_HI**			
Two LSB's of I0 or I1	**byte7**	**byte6**	**byte5**	**byte4**	**byte3**	**byte2**	**byte1**	**byte0**
00b:	▨				byte3	byte2	byte1	byte0
01b:	▨			byte4	byte3	byte2	byte1	▨
10b:	▨		byte5	byte4	byte3	byte2	▨	
11b:	▨	byte6	byte5	byte4	byte3	▨		

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

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 = byteop2p (r1:0, r3:2) (rndl) ;
r3 = byteop2p (r1:0, r3:2) (rndh) ;
r3 = byteop2p (r1:0, r3:2) (tl) ;
r3 = byteop2p (r1:0, r3:2) (th) ;
r3 = byteop2p (r1:0, r3:2) (rndl, r) ;
r3 = byteop2p (r1:0, r3:2) (rndh, r) ;
r3 = byteop2p (r1:0, r3:2) (tl, r) ;
r3 = byteop2p (r1:0, r3:2) (th, r) ;
```

Also See

BYTEOP1P (Quad 8-Bit Average – Byte)

Special Applications

This instruction supports binary interpolation used in fractional motion search and motion compensation algorithms.

BYTEPACK (Quad 8-Bit Pack)

General Form

```
dest_reg = BYTEPACK ( src_reg_0, src_reg_1 )
```

Syntax

```
Dreg = 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

	31................24	23................16	15..................8	7....................0
src_reg_0:		byte1		byte0
src_reg_1:		byte3		byte2

Table 18-23. Destination Register Receives

dest_reg:	byte3	byte2	byte1	byte0

This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

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

```
r2 = bytepack (r4,r5) ;
```

- Assuming:
  - R4 = 0xFEED FACE
  - R5 = 0xBEEF BADD

  then this instruction returns:
  - R2 = 0xEFDD EDCE

Also See

BYTEUNPACK (Quad 8-Bit Unpack)

Special Applications

None

BYTEOP16M (Quad 8-Bit Subtract)

General Form

```
(dest_reg_1, dest_reg_0) = BYTEOP16M (src_reg_0, src_reg_1)
(dest_reg_1, dest_reg_0) = BYTEOP16M (src_reg_0, src_reg_1) (R)
```

Syntax

```
/* forward byte order operands */
(Dreg, Dreg) = BYTEOP16M (Dreg_pair, Dreg_pair) ;      /* (b */)
/* reverse byte order operands */
(Dreg, Dreg) = 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

	31................24	23................16	15..................8	7....................0
aligned_src_reg_0:	y3	y2	y1	y0
aligned_src_reg_1:	z3	z2	z1	z0

Table 18-25. Destination Registers Receive

	31...............24	23...............16	15.................8	7....................0
dest_reg_0:	y1 - z1		y0 - z0	
dest_reg_1:	y3 - z3		y2 - z2	

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 18-26. I-register Bits and the Byte Alignment

The bytes selected are	**src_reg_pair_HI**				**src_reg_pair_LO**			
Two LSB's of I0 or I1	**byte7**	**byte6**	**byte5**	**byte4**	**byte3**	**byte2**	**byte1**	**byte0**
00b:					byte3	byte2	byte1	byte0
01b:				byte4	byte3	byte2	byte1	
10b:			byte5	byte4	byte3	byte2		
11b:		byte6	byte5	byte4	byte3			

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) 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

The bytes selected are	**src_reg_pair_LO**				**src_reg_pair_HI**			
Two LSB's of I0 or I1	**byte7**	**byte6**	**byte5**	**byte4**	**byte3**	**byte2**	**byte1**	**byte0**
00b:					byte3	byte2	byte1	byte0
01b:				byte4	byte3	byte2	byte1	
10b:			byte5	byte4	byte3	byte2		
11b:		byte6	byte5	byte4	byte3			

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.

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.

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 oper-
ands (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.

$$SAD = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} |a(i,j) - b(i,j)|$$

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

src_reg_0	a(i, j+3)		a(i, j+2)		a(i, j+1)		a(i, j)
src_reg_1	b(i, j+3)		b(i, j+2)		b(i, j+1)		b(i, j)
A1.H	+=| a(i, j+3) -b(i, j+3) |	A1.L	+=| a(i, j+2) - b(i, j+2) |	A0.H	+=| a(i, j+1) - b(i, j+1) |	A0.L	+=| a(i, j) - b(i, j) |

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

The bytes selected are	**src_reg_pair_HI**				**src_reg_pair_LO**			
Two LSB's of I0 or I1	**byte7**	**byte6**	**byte5**	**byte4**	**byte3**	**byte2**	**byte1**	**byte0**
00b:					byte3	byte2	byte1	byte0
01b:				byte4	byte3	byte2	byte1	
10b:			byte5	byte4	byte3	byte2		
11b:		byte6	byte5	byte4	byte3			

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.

Table 18-30. I-register Bits and the Byte Alignment

The bytes selected are	**src_reg_pair_LO**				**src_reg_pair_HI**			
Two LSB's of I0 or I1	**byte7**	**byte6**	**byte5**	**byte4**	**byte3**	**byte2**	**byte1**	**byte0**
00b:					byte3	byte2	byte1	byte0
01b:				byte4	byte3	byte2	byte1	
10b:			byte5	byte4	byte3	byte2		
11b:		byte6	byte5	byte4	byte3			

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 */
```

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.

BYTEUNPACK (Quad 8-Bit Unpack)

General Form

```
( dest_reg_1, dest_reg_0 ) = BYTEUNPACK src_reg_pair
( dest_reg_1, dest_reg_0 ) = BYTEUNPACK src_reg_pair (R)
```

Syntax

```
( Dreg , Dreg ) = BYTEUNPACK Dreg_pair ;   /* (b) */
( Dreg , Dreg ) = BYTEUNPACK Dreg_pair (R) ;   /* reverse source
order (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 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.

Table 18-31. I-register Bits and the Byte Alignment

The bytes selected are	src_reg_pair_HI				src_reg_pair_LO			
Two LSB’s of I0 or I1	byte7	byte6	byte5	byte4	byte3	byte2	byte1	byte0
00b:					byte3	byte2	byte1	byte0
01b:				byte4	byte3	byte2	byte1	
10b:			byte5	byte4	byte3	byte2		
11b:		byte6	byte5	byte4	byte3			

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

The bytes selected are	src_reg_pair_LO				src_reg_pair_HI			
Two LSB’s of I0 or I1	byte7	byte6	byte5	byte4	byte3	byte2	byte1	byte0
00b:					byte3	byte2	byte1	byte0
01b:				byte4	byte3	byte2	byte1	
10b:			byte5	byte4	byte3	byte2		
11b:		byte6	byte5	byte4	byte3			

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

	31...............24	23...............16	15.................8	7...................0
Aligned bytes:	byte_D	byte_C	byte_B	byte_A

Table 18-34. Destination Registers Receive

	31...............24	23...............16	15.................8	7...................0
dest_reg_0:		byte_B		byte_A
dest_reg_1:		byte_D		byte_C

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.

Example

```
(r6,r5) = byteunpack r1:0 ;    /* non-reversing sources */
```

- Assuming:
  - register I0's two LSBs = 00b,
  - R1 = 0xFEED FACE
  - R0 = 0xBEEF BADD

  then this instruction returns:
  - R6 = 0x00BE 00EF
  - R5 = 0x00BA 00DD
- Assuming:
  - register I0's two LSBs = 01b,
  - R1 = 0xFEED FACE
  - R0 = 0xBEEF BADD

  then this instruction returns:
  - R6 = 0x00CE 00BE
  - R5 = 0x00EF 00BA

- Assuming:
  - register I0's two LSBs = 10b,
  - R1 = 0xFEED FACE
  - R0 = 0xBEEF BADD

  then this instruction returns:
  - R6 = 0x00FA 00CE
  - R5 = 0x00BE 00EF
- Assuming:
  - register I0'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 */
```

- Assuming:
  - register I0's two LSBs = 00b,
  - R1 = 0xFEED FACE
  - R0 = 0xBEEF BADD

  then this instruction returns:
  - R6 = 0x00FE 00ED
  - R5 = 0x00FA 00CE
- Assuming:
  - register I0's two LSBs = 01b,
  - R1 = 0xFEED FACE
  - R0 = 0xBEEF BADD

  then this instruction returns:
  - R6 = 0x00DD 00FE
  - R5 = 0x00ED 00FA
- Assuming:
  - register I0's two LSBs = 10b,
  - R1 = 0xFEED FACE
  - R0 = 0xBEEF BADD

  then this instruction returns:
  - R6 = 0x00BA 00DD
  - R5 = 0x00FE 00ED

- Assuming:
  - register I0's two LSBs = 11b,
  - R1 = 0xFEED FACE
  - R0 = 0xBEEF BADD

  then this instruction returns:
  - R6 = 0x00EF 00BA
  - R5 = 0x00DD 00FE

Also See

BYTEPACK (Quad 8-Bit Pack)

Special Applications

None

19 VECTOR OPERATIONS

Instruction Summary

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.

Add on Sign

General Form

```
dest_hi = dest_lo = SIGN (src0_hi) * src1_hi
                    + SIGN (src0_lo) * src1_lo
```

Syntax

```
Dreg_hi = Dreg_lo = SIGN ( Dreg_hi ) * Dreg_hi
                    + SIGN ( Dreg_lo ) * Dreg_lo ;
/* (b) */
```

Register Consistency

The destination registers *dest_hi* and *dest_lo* must be halves of the same data register. Similarly, *src0_hi* and *src0_lo* must be halves of the same register and *src1_hi* and *src1_lo* must be halves of the same register.

Syntax Terminology

Dreg_hi: R7–0.H

Dreg_lo: R7–0.L

Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

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 *src0* by the corresponding half-word number in *src1*. The arithmetic sign of *src0* is either (+1) or (–1), depending on the sign bit of *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 *src0*, and Z is the number in *src1*. The numbers in *src0* and *src1* may be positive or negative.

Table 19-1. Signed Multiplication Rules

SRC0	SRC1	Sign-Adjusted SRC1
+Y	+Z	+Z
+Y	–Z	–Z
–Y	+Z	–Z
–Y	–Z	+Z

   Note the result always bears the magnitude of Z with only the sign affected.

2. Then, add the sign-adjusted *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.

Table 19-2. Source Registers Contain

	31...............24	23...............16	15.................8	7...................0
src0:	a1		a0	
src1:	b1		b0	

Table 19-3. Destination Register Receives

	31...............24	23...............16	15.................8	7...................0
dest:	(sign_adjusted_b1) + (sign_adjusted_b0)		(sign_adjusted_b1) + (sign_adjusted_b0)	

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

```
r7.h=r7.l=sign(r2.h)*r3.h+sign(r2.l)*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

- 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.

VIT_MAX (Compare-Select)

General Form

```
dest_reg = VIT_MAX ( src_reg_0, src_reg_1 ) (ASL)
dest_reg = VIT_MAX ( src_reg_0, src_reg_1 ) (ASR)
dest_reg_lo = VIT_MAX ( src_reg ) (ASL)
dest_reg_lo = VIT_MAX ( src_reg ) (ASR)
```

Syntax

Dual 16-Bit Operation

```
Dreg = VIT_MAX ( Dreg , Dreg ) (ASL) ;    /* shift history bits 
left (b) */
Dreg = VIT_MAX ( Dreg , Dreg ) (ASR) ;    /* shift history bits 
right (b) */
```

Single 16-Bit Operation

```
Dreg_lo = VIT_MAX ( Dreg ) (ASL) ;    /* shift history bits left 
(b) */
Dreg_lo = VIT_MAX ( Dreg ) (ASR) ;    /* shift history bits right 
(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 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

	31...............24	23...............16	15.................8	7....................0
src_reg_0	y1		y0	
src_reg_1	z1		z0	

Table 19-5. Destination Register Contains

	31...............24	23...............16	15.................8	7....................0
dest_reg	Maximum, y1 or y0		Maximum, z1 or z0	

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.

Table 19-6. ASL Version Shifts

	A0.X	A0.W
A0	00000000	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXBB

Table 19-7. Where

BB	Indicates
00	z0 and y0 are maxima
01	z0 and y1 are maxima
10	z1 and y0 are maxima
11	z1 and y1 are maxima

Conversely, the ASR version shifts `A0` 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

	A0.X	A0.W
A0	00000000	BBXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Table 19-9. Where

BB	Indicates
00	y0 and z0 are maxima
01	y0 and z1 are maxima
10	y1 and z0 are maxima
11	y1 and z1 are maxima

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

	31...............24	23...............16	15.................8	7...................0
src_reg	y1		y0	

Table 19-11. Destination Register Contains

	31...............24	23...............16	15.................8	7...................0
dest_reg_lo			Maximum, y1 or y0	

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

	A0.X	A0.W
A0	00000000	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXB

Conversely, the ASR version shifts A0 right one bit position and appends an MSB to indicate the source of the maximum.

Table 19-13. ASR Version Shifts

	A0.X	A0.W
A0	00000000	BXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Table 19-14. Where

B	Indicates
0	y0 is the maximum
1	y1 is the maximum

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.

Example

```
r5 = vit_max(r3, r2)(asl) ; /* shift left, dual operation */
```

- Assume:
    - R3 = 0xFFFF 0000
    - R2 = 0x0000 FFFF
    - A0 = 0x00 0000 0000

  This example produces:
    - R5 = 0x0000 0000
    - A0 = 0x00 0000 0002

```
r7 = vit_max (r1, r0) (asr) ; /* shift right, dual operation */
```

- Assume:
    - R1 = 0xFEED BEEF
    - R0 = 0xDEAF 0000
    - A0 = 0x00 0000 0000

  This example produces:
    - R7 = 0xFEED 0000
    - A0 = 0x00 8000 0000

```
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 ABS

General Form

```
dest_reg = ABS source_reg (V)
```

Syntax

```
Dreg = ABS Dreg (V) ;    /* (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 `dest_reg`, 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

	31................24	23................16	15..................8	7....................0
src_reg:	x.h		x.l	

Table 19-16. Destination Register Contains

<table>
<tr><th></th><th>31...............24</th><th>23...............16</th><th>15.................8</th><th>7...................0</th></tr>
<tr><td>dest_reg:</td><td colspan="2">| x.h|</td><td colspan="2">| x.l |</td></tr>
</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.

Example

```
/* If r1 = 0xFFFF 7FFF, then . . . */
r3 = abs r1 (v) ;
/* . . . produces 0x0001 7FFF */
```

Also See

ABS

Special Applications

None

Vector Add / Subtract

General Form

```
dest = src_reg_0 +|+ src_reg_1
dest = src_reg_0 -|+ src_reg_1
dest = src_reg_0 +|- src_reg_1
dest = src_reg_0 -|- src_reg_1
dest_0 = src_reg_0 +|+ src_reg_1,
dest_1 = src_reg_0 -|- src_reg_1
dest_0 = src_reg_0 +|- src_reg_1,
dest_1 = src_reg_0 -|+ src_reg_1
dest_0 = src_reg_0 + src_reg_1,
dest_1 = src_reg_0 - src_reg_1
dest_0 = A1 + A0, dest_1 = A1 - A0
dest_0 = A0 + A1, dest_1 = A0 - A1
```

Syntax

Dual 16-Bit Operations

```
Dreg = Dreg +|+ Dreg (opt_mode_0) ; /* add | add (b) */
Dreg = Dreg -|+ Dreg (opt_mode_0) ; /* subtract | add (b) */
Dreg = Dreg +|- Dreg (opt_mode_0) ; /* add | subtract (b) */
Dreg = Dreg -|- Dreg (opt_mode_0) ; /* subtract | subtract (b) */
```

Quad 16-Bit Operations

```
Dreg = Dreg +|+ Dreg, Dreg = Dreg -|- Dreg (opt_mode_0,
opt_mode_2) ;
/* add | add, subtract | subtract; the set of source registers
must be the same for each operation (b) */
Dreg = Dreg +|- Dreg, Dreg = Dreg -|+ Dreg (opt_mode_0,
opt_mode_2) ;
/* add | subtract, subtract | add; the set of source registers
must be the same for each operation (b) */
```

Dual 32-Bit Operations

```
Dreg = Dreg + Dreg,   Dreg = Dreg - Dreg   (opt_mode_1) ;
/* add, subtract; the set of source registers must be the same
for each operation (b) */
```

Dual 40-Bit Accumulator Operations

```
Dreg = A1 + A0,   Dreg = A1 - A0   (opt_mode_1) ;    /* add, sub-
tract Accumulators; subtract A0 from A1 (b) */
Dreg = A0 + A1,   Dreg = A0 - A1   (opt_mode_1) ;    /* add, sub-
tract Accumulators; subtract A1 from A0 (b) */
```

Syntax Terminology

Dreg: `R7-0`

opt_mode_0: optional `(S)`, `(CO)`, or `(SCO)`

opt_mode_1: optional `(S)`

opt_mode_2: 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

Mode	Option	Description
opt_mode_0	S	Saturate the results at 16 bits.
	CO	Cross option. Swap the order of the results in the destination register.
	SCO	Saturate and cross option. Combination of (S) and (CO) options.
opt_mode_1	S	Saturate the results at 16 or 32 bits, depending on the operand size.
opt_mode_2	ASR	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.
	ASL	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.

The options shown for *opt_mode_2* are scaling options.

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 */
```

```
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 Arithmetic Shift

General Form

```
dest_reg = src_reg >>> shift_magnitude (V)
dest_reg = ASHIFT src_reg BY shift_magnitude (V)
```

Syntax

Constant Shift Magnitude

```
Dreg = Dreg >>> uimm4 (V) ;    /* arithmetic shift right, immedi-
ate (b) */
Dreg = Dreg << uimm4 (V,S) ;   /* arithmetic shift left, immedi-
ate with saturation (b) */
```

Registered Shift Magnitude

```
Dreg = ASHIFT Dreg BY Dreg_lo (V) ;    /* arithmetic shift (b) */
Dreg = ASHIFT Dreg BY Dreg_lo (V, S) ;    /* 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 "">>>" and "<<" Syntax" notes for caveats.

Syntax Terminology

Dreg: R7-0

Dreg_lo: R7-0.L

uimm4: unsigned 4-bit field, with a range of 0 through 15

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 *dest_reg* are right 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.

“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.

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.

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

Vector Logical Shift

General Form

```
dest_reg = src_reg >> shift_magnitude (V)
dest_reg = src_reg << shift_magnitude (V)
dest_reg = LSHIFT src_reg BY shift_magnitude (V)
```

Syntax

Constant Shift Magnitude

```
Dreg = Dreg >> uimm4 (V) ;   /* logical shift right, immediate
(b) */
Dreg = Dreg << uimm4 (V) ;   /* logical shift left, immediate
(b) */
```

Registered Shift Magnitude

```
Dreg = LSHIFT Dreg BY Dreg_lo (V) ;   /* logical shift (b) */
```

Syntax Terminology

Dreg: R7-0

Dreg_lo: R7-0.L

uimm4: unsigned 4-bit field, with a range of 0 through 15

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.

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.

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.

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=lshift r7 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

Vector MAX

General Form

```
dest_reg = MAX ( src_reg_0, src_reg_1 ) (V)
```

Syntax

```
Dreg = MAX ( Dreg , Dreg ) (V) ;    /* 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 *dest_reg*.

The instruction compares the upper half-words of *src_reg_0* and *src_reg_1* and returns that maximum 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 maximum to the lower half-word of *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 *dest_reg* can be the same D-register as one of the source registers. Doing this explicitly modifies that source register.

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 = 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.

Also See

Vector SEARCH, Vector MIN, MAX, MIN

Special Applications

None

Vector MIN

General Form

```
dest_reg = MIN ( src_reg_0, src_reg_1 ) (V)
```

Syntax

```
Dreg = 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.

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 = min (r1, r0) (v) ;
```

- Assume R1 = 0x0007 0000 and R0 = 0x0000 000F, then R7 = 0x0000 0000.
- Assume R1 = 0xFFF7 8000 and R0 = 0x000A 7FFF, then R7 = 0xFFF7 8000.
- Assume R1 = 0x1234 5678 and R0 = 0x0000 000F, then R7 = 0x0000 000F.

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, `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.
- 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`*.

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

```
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.
MAC0 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. MAC0
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 MAC0 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 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.

- 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.

- 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

```
a1=r2.l*r3.h, a0=r2.h*r3.h ;
/* both multiply signed fractions into separate Accumulators */
a0=r1.l*r0.l, a1+=r1.h*r0.h ;
/* same as above, but sum result into A1. MAC order is arbitrary.
*/
a1+=r3.h*r3.l, a0-=r3.h*r3.h ;
/* sum product into A1, subtract product from A0 */
a1=r3.h*r2.l (m), a0+=r3.l*r2.l ;
/* MAC1 multiplies a signed fraction in r3.h by an unsigned frac-
tion in r2.l. MAC0 multiplies two signed fractions. */
a1=r7.h*r4.h (m), a0+=r7.l*r4.l (fu) ;
/* MAC1 multiplies signed fraction by unsigned fraction. MAC0
multiplies and accumulates two unsigned fractions. */
a1+=r3.h*r2.h, a0=r3.l*r2.l (is) ;
/* both MACs perform signed integer multiplication */
a1=r6.h*r7.h, a0+=r6.l*r7.l (w32) ;
/* both MACs multiply signed fractions, sign extended, and satu-
rate both Accumulators at bit 31 */
```

Result is 16-bit half D-register

```
r2.h=(a1=r7.l*r6.h), r2.l=(a0=r7.h*r6.h) ;   /* simultaneous MAC0
and MAC1 execution, both are signed fractions, both products load
into the Accumulators,MAC1 into half-word registers. */
r4.l=(a0=r1.l*r0.l), r4.h=(a1+=r1.h*r0.h) ;   /* same as above,
but sum result into A1. ; MAC order is arbitrary. */
r7.h=(a1+=r6.h*r5.l), r7.l=(a0=r6.h*r5.h) ;   /* sum into A1,
subtract into A0  */
r0.h=(a1=r7.h*r4.l) (m), r0.l=(a0+=r7.l*r4.l) ;   /* MAC1 multi-
plies a signed fraction by an unsigned fraction. MAC0 multiplies
two signed fractions. */
r5.h=(a1=r3.h*r2.h) (m), r5.l=(a0+=r3.l*r2.l) (fu) ;   /* MAC1
multiplies signed fraction by unsigned fraction. MAC0 multiplies
two unsigned fractions. */
r0.h=(a1+=r3.h*r2.h), r0.l=(a0=r3.l*r2.l) (is) ;   /* both MACs
perform signed integer multiplication. */
r5.h=(a1=r2.h*r1.h), a0+=r2.l*r1.l ;   /* both MACs multiply
signed fractions. MAC0 does not copy the accum result. */
r3.h=(a1=r2.h*r1.h) (m), a0=r2.l*r1.l ;   /* MAC1 multiplies
signed fraction by unsigned fraction and uses all 40 bits of A1.
MAC0 multiplies two signed fractions. */
r3.h=a1, r3.l=(a0+=r0.l*r1.l) (s2rnd) ;   /* MAC1 copies Accumu-
lator to register half. MAC0 multiplies signed fractions. Both
scale the result and round on the way to the destination regis-
ter. */
r0.l=(a0+=r7.l*r6.l), r0.h=(a1+=r7.h*r6.h) (iss2) ;   /* both
MACs process signed integer the way to the destination half-reg-
isters. */
```

Result is 32-bit D-register

```
r3=(a1=r6.h*r7.h), r2=(a0=r6.l*r7.l) ;   /* simultaneous MAC0 and
MAC1 execution, both are signed fractions, both products load
into the Accumulators */
r4=(a0=r6.l*r7.l), r5=(a1+=r6.h*r7.h) ;   /* same as above, but
sum result into A1. MAC order is arbitrary. */
r7=(a1+=r3.h*r5.h), r6=(a0-=r3.l*r5.l) ;   /* sum into A1, sub-
tract into A0 */
r1=(a1=r7.l*r4.l) (m), r0=(a0+=r7.h*r4.h) ;   /* MAC1 multiplies
a signed fraction by an unsigned fraction. MAC0 multiplies two
signed fractions. */
r5=(a1=r3.h*r7.h) (m), r4=(a0+=r3.l*r7.l) (fu) ;   /* MAC1 multi-
plies signed fraction by unsigned fraction. MAC0 multiplies two
unsigned fractions. */
r1=(a1+=r3.h*r2.h), r0=(a0=r3.l*r2.l) (is) ;   /* both MACs per-
form signed integer multiplication */
r5=(a1-=r6.h*r7.h), a0+=r6.l*r7.l ;    /* both MACs multiply
signed fractions. MAC0 does not copy the accum result */
r3=(a1=r6.h*r7.h) (m), a0-=r6.l*r7.l ;   /* MAC1 multiplies
signed fraction by unsigned fraction and uses all 40 bits of A1.
MAC0 multiplies two signed fractions. */
r3=a1, r2=(a0+=r0.l*r1.l) (s2rnd) ;   /* MAC1 moves Accumulator
to register. MAC0 multiplies signed fractions. Both scale the
result and round on the way to the destination register. */
r0=(a0+=r7.l*r6.l), r1=(a1+=r7.h*r6.h) (iss2) ;   /* both MACs
process signed integer operands and scale the result on the way
to the destination registers. */
```

Vector Negate (Two's-Complement)

General Form

```
dest_reg = - source_reg (V)
```

Syntax

```
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 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.

- VS is set if V is set; unaffected otherwise.
- AC0 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 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 (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
= 0xFFFC 8001 */
```

Also See

Negate (Two’s-Complement)

Special Applications

None

Vector PACK

General Form

```
Dest_reg = PACK ( src_half_0, src_half_1 )
```

Syntax

```
Dreg = PACK ( Dreg_lo_hi , Dreg_lo_hi ) ;     /* (b) */
```

Syntax Terminology

`Dreg`: `R7–0`

`Dreg_lo_hi`: `R7–0.L`, `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

		15..................8	7....................0
src_half_0		half_word_0	
src_half_1		half_word_1	

Table 19-19. Destination Register Contains

	31................24	23................16	15..................8	7....................0
dest_reg:	half_word_0		half_word_1	

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=pack(r4.l, r5.l) ;    /* pack low / low half-words */
r1=pack(r6.l, r4.h) ;    /* pack low / high half-words */
r0=pack(r2.h, r4.l) ;    /* pack high / low half-words */
r5=pack(r7.h, r2.h) ;    /* pack high / high half-words */
```

Also See

BYTEPACK (Quad 8-Bit Pack)

Special Applications

```
/* If r4.l = 0xDEAD and r5.l = 0xBEEF, then . . . */
r3 = pack (r4.l, r5.l) ;
/* . . . produces r3 = 0xDEAD BEEF */
```

Vector SEARCH

General Form

```
(dest_pointer_hi, dest_pointer_lo ) = SEARCH src_reg (searchmode)
```

Syntax

```
(Dreg, Dreg) = SEARCH Dreg (searchmode) ;   /* (b) */
```

Syntax Terminology

Dreg: R7–0

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 *src_reg* is compared in magnitude with the 16 low-order bits in A1. If *src_reg_hi* meets the comparison criterion, then A1 is updated with *src_reg_hi*, and the value in pointer register P0 is stored in *dest_pointer_hi*. The same operation is performed for *src_reg_low* and A0.

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

Mode	Description
(GT)	Greater than. Find the location of the first maximum number in an array.
(GE)	Greater than or equal. Find the location of the last maximum number in an array.
(LT)	Less than. Find the location of the first minimum number in an array.
(LE)	Less than or equal. Find the location of the last minimum number in an array.

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`*.

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. */
```

```
   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. */
cc = 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 according to the element's value.

20 ISSUING PARALLEL INSTRUCTIONS

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

The diagram in Table 20-1 illustrates the combinations for parallel issue that the Blackfin processor supports.

Table 20-1. Parallel Issue Combinations

32-bit ALU/MAC instruction	16-bit Instruction	16-bit Instruction

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.

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

Instruction Name	Notes
Arithmetic Operations	
ABS (Absolute Value)	
Add	Only the versions that support optional saturation.
Add/Subtract – Prescale Up	
Add/Subtract – Prescale Down	
EXPADJ (Exponent Detection)	
MAX (Maximum)	
MIN (Minimum)	
Modify – Decrement (for Accumulators, only)	
Modify – Increment (for Accumulators, only)	Accumulator versions only.
Negate (Two’s-Complement)	Accumulator versions only.
RND (Round to Half-Word)	
Saturate	
SIGNBITS	
Subtract	Saturating versions only.
Load Store	
Load Immediate	Accumulator versions only.

Table 20-2. 32-Bit DSP Instructions (Cont'd)

Instruction Name	Notes
Bit Operations	
DEPOSIT (Bit Field Deposit)	
EXTRACT (Bit Field Extract)	
BITMUX (Bit Multiplex)	
ONES (One's-Population Count)	
Logical Operations	
^ (Exclusive-OR) (Bit-Wise XOR)	
Move	
Move Register	40-bit Accumulator versions only.
Move Register Half	
Shift / Rotate Operations	
Arithmetic Shift	Saturating and Accumulator versions only.
Logical Shift	32-bit instruction size versions only.
ROT (Rotate)	
External Event Management	
No Op	32-bit MNOP only

Table 20-2. 32-Bit DSP Instructions (Cont'd)

Instruction Name	Notes
Vector Operations	
VIT_MAX (Compare-Select)	
Add on Sign	
Multiply and Multiply-Accumulate to Accumulator	
Multiply and Multiply-Accumulate to Half-Register	
Multiply and Multiply-Accumulate to Data Register	
Vector ABS (Vector Absolute Value)	
Vector Add / Subtract	
Vector Arithmetic Shift	
Vector Logical Shift	
Vector MAX (Vector Maximum)	
Vector MIN (Vector Minimum)	
Multiply 16-Bit Operands	
Vector Negate (Two's-Complement)	
Vector PACK	
Vector SEARCH	

Table 20-2. 32-Bit DSP Instructions (Cont'd)

Instruction Name	Notes
Video Pixel Operations	
ALIGN8, ALIGN16, ALIGN24 (Byte Align)	
DISALGNEXCPT (Disable Alignment Exception for Load)	
SAA (Quad 8-Bit Subtract-Absolute-Accumulate)	
Dual 16-Bit Accumulator Extraction with Addition	
BYTEOP16P (Quad 8-Bit Add)	
BYTEOP16M (Quad 8-Bit Subtract)	
BYTEOP1P (Quad 8-Bit Average – Byte)	
BYTEOP2P (Quad 8-Bit Average – Half-Word)	
BYTEOP3P (Dual 16-Bit Add / Clip)	
BYTEPACK (Quad 8-Bit Pack)	
BYTEUNPACK (Quad 8-Bit Unpack)	

16-Bit Instructions

The two16-bit instructions in a multi-issue instruction must each be from Group1 and Group2 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 20-3. Group1 Compatible 16-Bit Instructions

Instruction Name	Notes
Arithmetic Operations	
Add Immediate	Ireg versions only.
Modify – Decrement	Ireg versions only.
Modify – Increment	Ireg versions only.
Subtract Immediate	Ireg versions only.
Load / Store	
Load Pointer Register	
Load Data Register	
Load Half-Word – Zero-Extended	
Load Half-Word – Sign-Extended	
Load High Data Register Half	
Load Low Data Register Half	
Load Byte – Zero-Extended	
Load Byte – Sign-Extended	
Store Pointer Register	
Store Data Register	
Store High Data Register Half	
Store Low Data Register Half	
Store Byte	

Table 20-4. Group2 Compatible 16-Bit Instructions

Instruction Name	Notes
Load / Store	
Load Data Register	Ireg versions only.
Load High Data Register Half	Ireg versions only.
Load Low Data Register Half	Ireg versions only.
Store Data Register	Ireg versions only.
Store High Data Register Half	Ireg versions only.
Store Low Data Register Half	Ireg versions only.
External Event Management	
No Op	16-bit NOP only.

Examples

Two Parallel Memory Access Instructions

```
/* Subtract-Absolute-Accumulate issued in parallel with the mem-
ory 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++] ;
```

One *Ireg* and One Memory Access Instruction in Parallel

```
/* Add on Sign while incrementing an Ireg and loading a data reg-
ister 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. */
A1=R2.L*R1.L, A0=R2.H*R1.H || R2.H=W[I2++] || [I3++]=R3 ;
/* 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--] ;
R3.H=(A1+=R0.L*R1.H), R3.L=(A0+=R0.L*R1.L) || R0=[P0++] ||
R1=[I0] ;
/* 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 ; */
```

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

Debug Feature	Description
Watchpoints	Specify address ranges and conditions that halt the processor when satisfied.
Trace History	Stores the last 16 discontinuous values of the Program Counter in an on-chip trace buffer.
Cycle Count	Provides functionality for all code profiling functions.
Performance Monitoring	Allows internal resources to be monitored and measured non-intrusively.

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.

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 <= WPIA1

WPIA2 <= WPIA3

WPIA4 <= 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.

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

Bit Name	Description
EMUSWx	Determines whether an instruction-address match causes either an emulation event or an exception event.
WPICNTENx	Enables the 16-bit counter that counts the number of address matches. If the counter is disabled, then every match causes an event.
WPIAENx	Enables the address watchpoint activity.

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

Bit Name	Description
WPIRENxy	Indicates the two watchpoints that are to be associated to form a range.
WPIRINVxy	Determines whether an event is caused by an address within the range identified or outside of the range identified.

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].

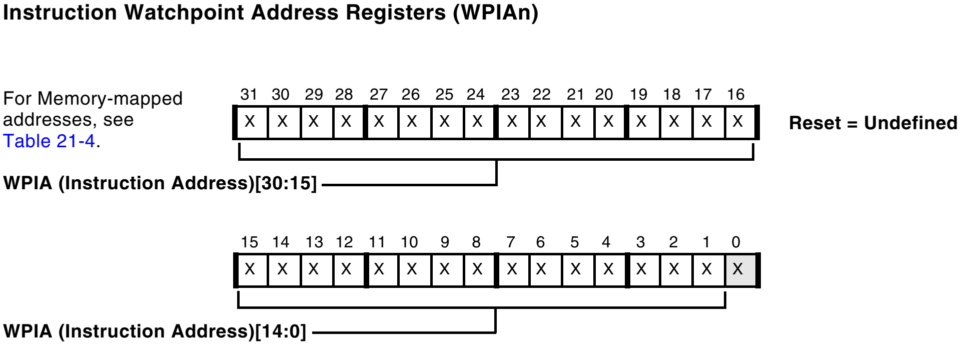

Instruction Watchpoint Address Registers (WPIAn)
For Memory-mapped addresses, see Table 21-4.
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
WPIA (Instruction Address)[30:15]
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
WPIA (Instruction Address)[14:0]

Figure 21-1. Instruction Watchpoint Address Registers

Table 21-4. Instruction Watchpoint Register Memory-mapped Addresses

Register Name	Memory-mapped Address
WPIA0	0xFFE0 7040
WPIA1	0xFFE0 7044
WPIA2	0xFFE0 7048
WPIA3	0xFFE0 704C
WPIA4	0xFFE0 7050
WPIA5	0xFFE0 7054

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.

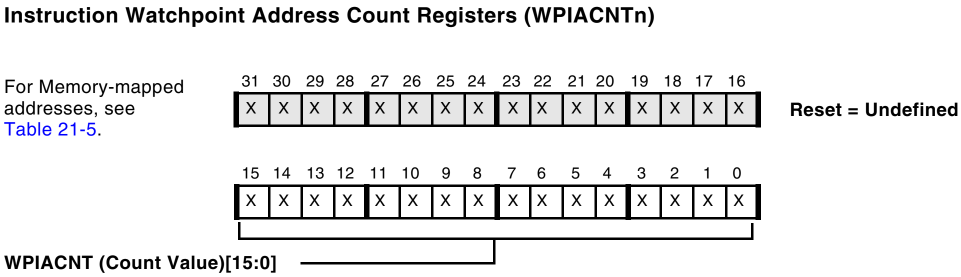

Instruction Watchpoint Address Count Registers (WPIACNTn)
For Memory-mapped addresses, see Table 21-5.
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
WPIACNT (Count Value)[15:0]

Figure 21-2. Instruction Watchpoint Address Count Registers

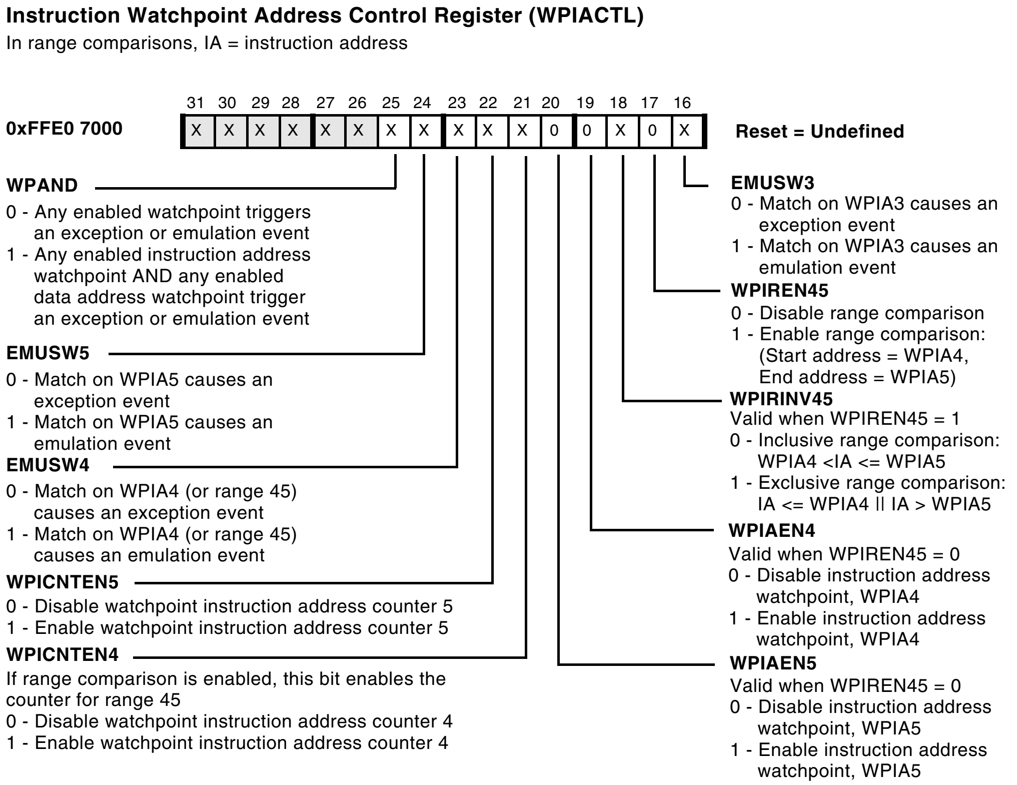
Table 21-5. Instruction Watchpoint Address Count Register Memory-mapped Addresses

Register Name	Memory-mapped Address
WPIACNT0	0xFFE0 7080
WPIACNT1	0xFFE0 7084
WPIACNT2	0xFFE0 7088
WPIACNT3	0xFFE0 708C
WPIACNT4	0xFFE0 7090
WPIACNT5	0xFFE0 7094

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.


Instruction Watchpoint Address Control Register (WPIACTL)
In range comparisons, IA = instruction address
0xFFE0 7000
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X 0 0 X 0 X
Reset = Undefined

WPAND
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

EMUSW5
0 - Match on WPIA5 causes an exception event
1 - Match on WPIA5 causes an emulation event

EMUSW4
0 - Match on WPIA4 (or range 45) causes an exception event
1 - Match on WPIA4 (or range 45) causes an emulation event

WPICNTEN5
0 - Disable watchpoint instruction address counter 5
1 - Enable watchpoint instruction address counter 5

WPICNTEN4
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

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
Valid when WPIREN45 = 0
0 - Disable instruction address watchpoint, WPIA4
1 - Enable instruction address watchpoint, WPIA4

WPIAEN5
Valid when WPIREN45 = 0
0 - Disable instruction address watchpoint, WPIA5
1 - Enable instruction address watchpoint, WPIA5

Figure 21-3. Instruction Watchpoint Address Control Register (WPIACTL)[31:16]

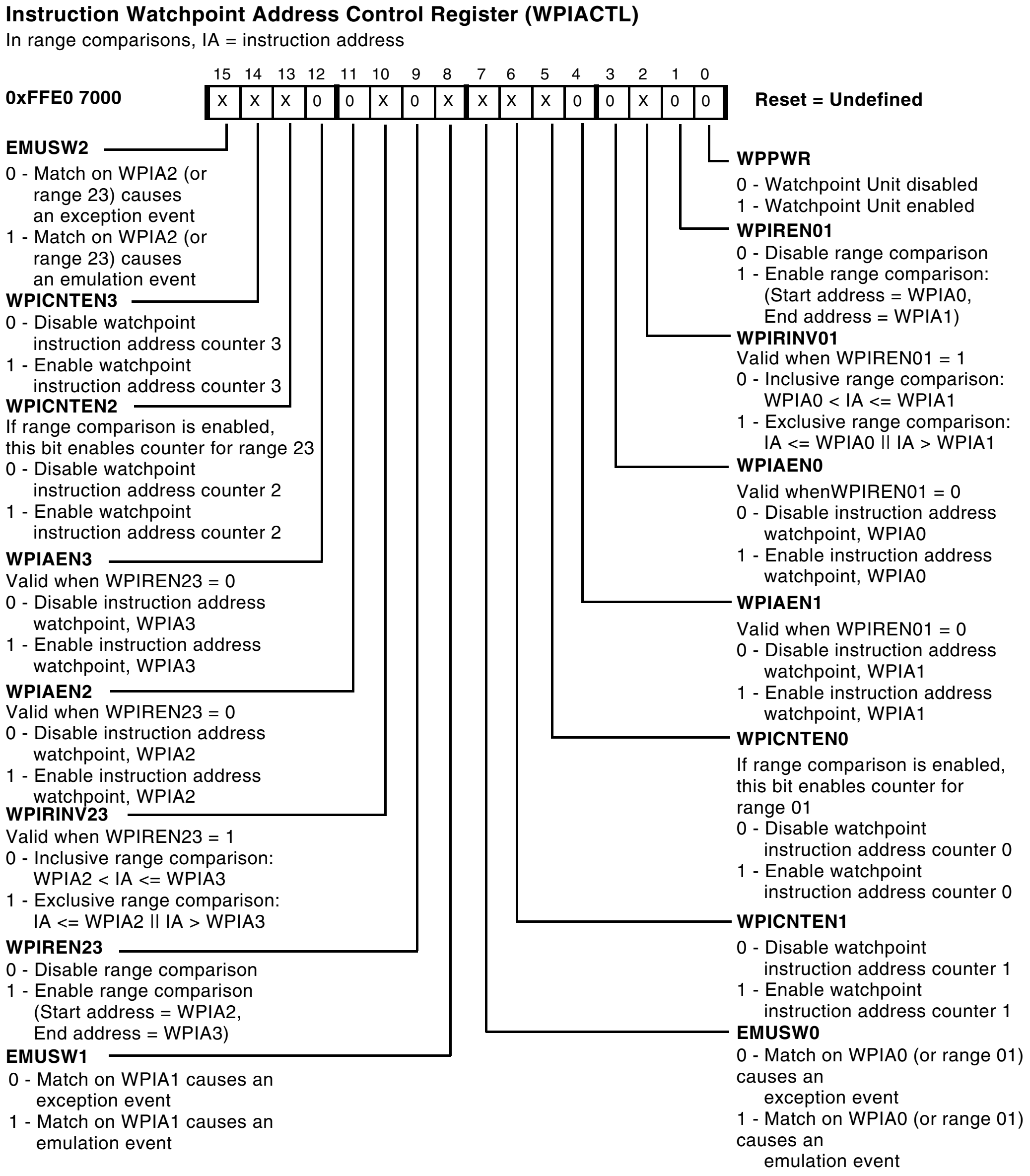

Instruction Watchpoint Address Control Register (WPIACTL)
In range comparisons, IA = instruction address
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFE0 7000
X X X 0 0 X 0 X X X X 0 0 X 0 0
Reset = Undefined
EMUSW2
0 - Match on WPIA2 (or range 23) causes an exception event
1 - Match on WPIA2 (or range 23) causes an emulation event
WPICNTEN3
0 - Disable watchpoint instruction address counter 3
1 - Enable watchpoint instruction address counter 3
WPICNTEN2
If range comparison is enabled, this bit enables counter for range 23
0 - Disable watchpoint instruction address counter 2
1 - Enable watchpoint instruction address counter 2
WPIAEN3
Valid when WPIREN23 = 0
0 - Disable instruction address watchpoint, WPIA3
1 - Enable instruction address watchpoint, WPIA3
WPIAEN2
Valid when WPIREN23 = 0
0 - Disable instruction address watchpoint, WPIA2
1 - Enable instruction address watchpoint, WPIA2
WPIRINV23
Valid when WPIREN23 = 1
0 - Inclusive range comparison: WPIA2 < IA <= WPIA3
1 - Exclusive range comparison: IA <= WPIA2 || IA > WPIA3
WPIREN23
0 - Disable range comparison
1 - Enable range comparison (Start address = WPIA2, End address = WPIA3)
EMUSW1
0 - Match on WPIA1 causes an exception event
1 - Match on WPIA1 causes an emulation event
WPPWR
0 - Watchpoint Unit disabled
1 - Watchpoint Unit enabled
WPIREN01
0 - Disable range comparison
1 - Enable range comparison: (Start address = WPIA0, End address = WPIA1)
WPIRINV01
Valid when WPIREN01 = 1
0 - Inclusive range comparison: WPIA0 < IA <= WPIA1
1 - Exclusive range comparison: IA <= WPIA0 || IA > WPIA1
WPIAEN0
Valid whenWPIREN01 = 0
0 - Disable instruction address watchpoint, WPIA0
1 - Enable instruction address watchpoint, WPIA0
WPIAEN1
Valid when WPIREN01 = 0
0 - Disable instruction address watchpoint, WPIA1
1 - Enable instruction address watchpoint, WPIA1
WPICNTEN0
If range comparison is enabled, this bit enables counter for range 01
0 - Disable watchpoint instruction address counter 0
1 - Enable watchpoint instruction address counter 0
WPICNTEN1
0 - Disable watchpoint instruction address counter 1
1 - Enable watchpoint instruction address counter 1
EMUSW0
0 - Match on WPIA0 (or range 01) causes an exception event
1 - Match on WPIA0 (or range 01) causes an emulation event

Figure 21-4. Instruction Watchpoint Address Control Register (WPIACTL)[15:0]

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

Bit Name	Description
WPDACCn	Determines whether the match should be on a read or write access.
WPDSRCn	Determines which DAG the unit should monitor.
WPDCNTENn	Enables the counter that counts the number of address matches. If the counter is disabled, then every match causes an event.
WPDAENn	Enables the data watchpoint activity.

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

Bit Name	Description
WPDREN01	Indicates the two watchpoints associated to form a range.
WPDRINV01	Determines whether an event is caused by an address within the range identified or outside the range.

ⓘ Note data address watchpoints always trigger emulation events.

WPDAn Registers

When the Watchpoint Unit is enabled, the values in the Data Watchpoint Address registers (WPDAn) are compared to the address on the data buses. Corresponding count values in the Data Watchpoint Address Count registers (WPDACNTn) are decremented on each match.

Figure 21-5 shows the Data Watchpoint Address registers, WPDA[1:0].

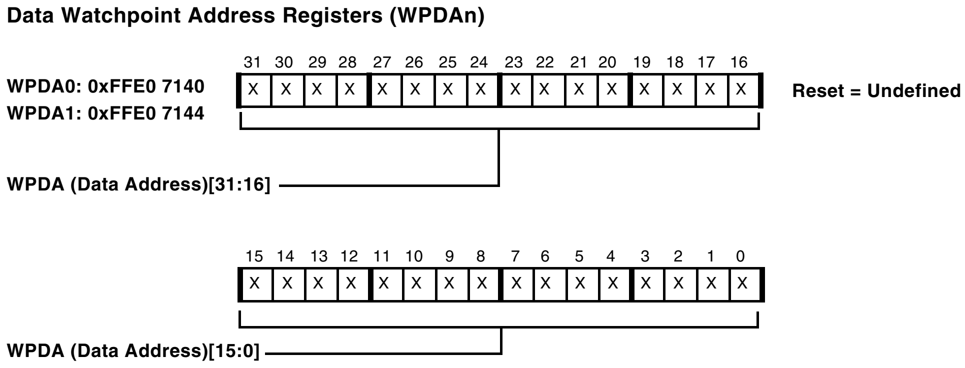

Data Watchpoint Address Registers (WPDAn)
WPDA0: 0xFFE0 7140
WPDA1: 0xFFE0 7144
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
WPDA (Data Address)[31:16]
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
WPDA (Data Address)[15:0]

Figure 21-5. Data Watchpoint Address Registers

WPDACNTn Registers

When the Watchpoint Unit is enabled, the count values in the Data Watchpoint Address Count Value registers (WPDACNTn) are decremented each time the address or the address bus matches a value in the WPDAn registers. Load this WPDACNTn register with a value that is one less than the number of times the watchpoint must match before triggering an event.

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].

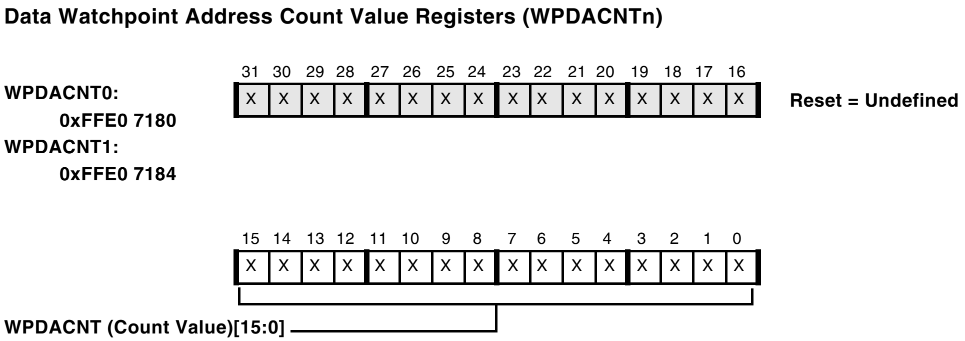
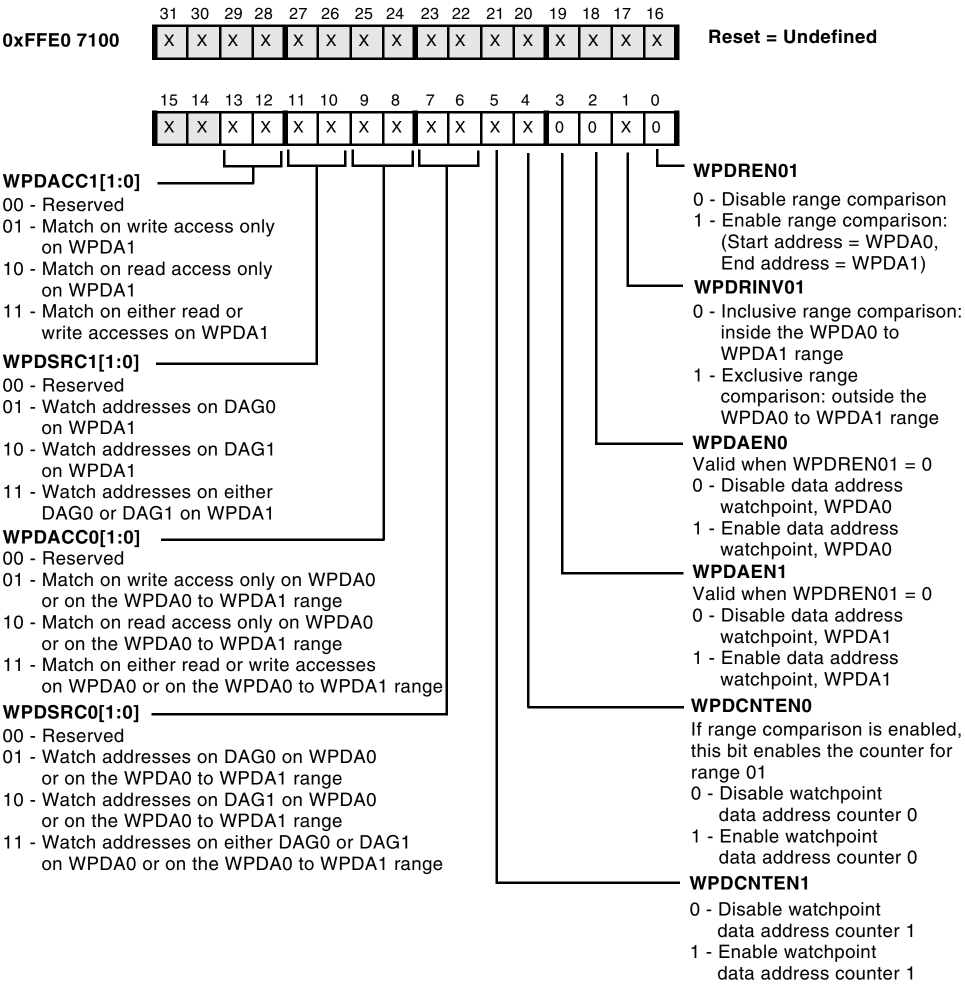

Data Watchpoint Address Count Value Registers (WPDACNTn)
WPDACNT0:
0xFFE0 7180
WPDACNT1:
0xFFE0 7184
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
WPDACNT (Count Value)[15:0]

Figure 21-6. Data Watchpoint Address Count Value Registers

WPDACTL Register

For more information about the bits in the Data Watchpoint Address Control register (WPDACTL), see “Data Address Watchpoints” on page 21-10.

Data Watchpoint Address Control Register (WPDACTL)



WPDACC1[1:0]
00 - Reserved
01 - Match on write access only on WPDA1
10 - Match on read access only on WPDA1
11 - Match on either read or write accesses on WPDA1

WPDSRC1[1:0]
00 - Reserved
01 - Watch addresses on DAG0 on WPDA1
10 - Watch addresses on DAG1 on WPDA1
11 - Watch addresses on either DAG0 or DAG1 on WPDA1

WPDACC0[1:0]
00 - Reserved
01 - Match on write access only on WPDA0 or on the WPDA0 to WPDA1 range
10 - Match on read access only on WPDA0 or on the WPDA0 to WPDA1 range
11 - Match on either read or write accesses on WPDA0 or on the WPDA0 to WPDA1 range

WPDSRC0[1:0]
00 - Reserved
01 - Watch addresses on DAG0 on WPDA0 or on the WPDA0 to WPDA1 range
10 - Watch addresses on DAG1 on WPDA0 or on the WPDA0 to WPDA1 range
11 - Watch addresses on either DAG0 or DAG1 on WPDA0 or on the WPDA0 to WPDA1 range

WPDREN01
0 - Disable range comparison
1 - Enable range comparison: (Start address = WPDA0, End address = WPDA1)

WPDRINV01
0 - Inclusive range comparison: inside the WPDA0 to WPDA1 range
1 - Exclusive range comparison: outside the WPDA0 to WPDA1 range

WPDAEN0
Valid when WPDREN01 = 0
0 - Disable data address watchpoint, WPDA0
1 - Enable data address watchpoint, WPDA0

WPDAEN1
Valid when WPDREN01 = 0
0 - Disable data address watchpoint, WPDA1
1 - Enable data address watchpoint, WPDA1

WPDCNTEN0
If range comparison is enabled, this bit enables the counter for range 01
0 - Disable watchpoint data address counter 0
1 - Enable watchpoint data address counter 0

WPDCNTEN1
0 - Disable watchpoint data address counter 1
1 - Enable watchpoint data address counter 1

Figure 21-7. Data Watchpoint Address Control Register

WPSTAT Register

The 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.

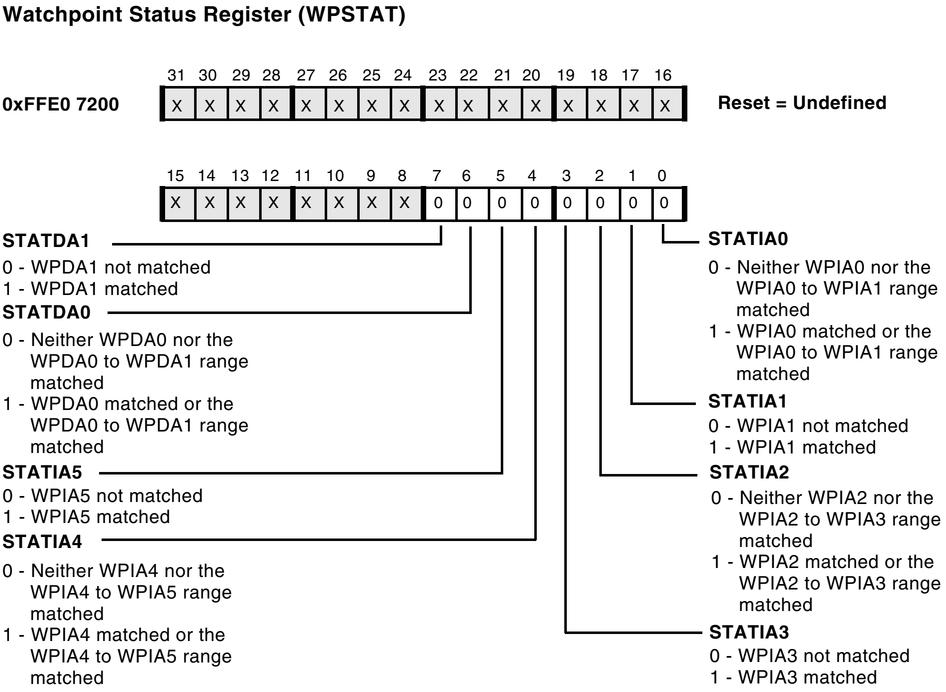

Watchpoint Status Register (WPSTAT)
0xFFE0 7200
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X 0 0 0 0 0 0 0 0
STATDA1
0 - WPDA1 not matched
1 - WPDA1 matched
STATDA0
0 - Neither WPDA0 nor the WPDA0 to WPDA1 range matched
1 - WPDA0 matched or the WPDA0 to WPDA1 range matched
STATIA5
0 - WPIA5 not matched
1 - WPIA5 matched
STATIA4
0 - Neither WPIA4 nor the WPIA4 to WPIA5 range matched
1 - WPIA4 matched or the WPIA4 to WPIA5 range matched
STATIA0
0 - Neither WPIA0 nor the WPIA0 to WPIA1 range matched
1 - WPIA0 matched or the WPIA0 to WPIA1 range matched
STATIA1
0 - WPIA1 not matched
1 - WPIA1 matched
STATIA2
0 - Neither WPIA2 nor the WPIA2 to WPIA3 range matched
1 - WPIA2 matched or the WPIA2 to WPIA3 range matched
STATIA3
0 - WPIA3 not matched
1 - WPIA3 matched

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.

The number of valid entries in TBUF is held in the TBUFCNT field of the TBUFSTAT register. On every second read, TBUFCNT is decremented. Because each entry corresponds to two pieces of data, a total of 2 x TBUFCNT reads empties the 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 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 (TBUFCTL) register. First, the Trace Unit must be activated by setting the TBUFPWR bit. If TBUFPWR = 1, then setting TBUFEN to 1 enables the Trace Unit.

Figure 21-9 describes the Trace Buffer Control register (TBUFCTL). If TBUFOVF = 1, then the Trace Unit does not record discontinuities in the exception, NMI, and reset routines.

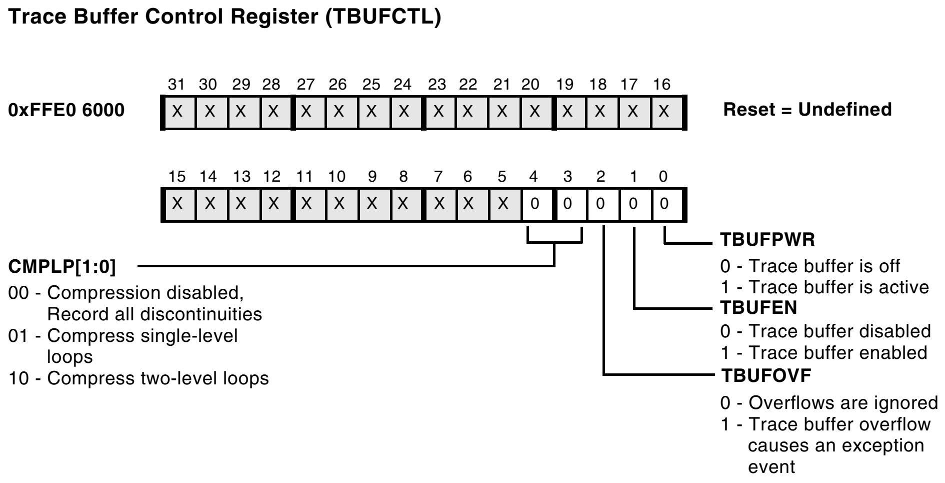

Trace Buffer Control Register (TBUFCTL)
0xFFE0 6000
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X 0 0 0 0 0
CMPLP[1:0]
00 - Compression disabled, Record all discontinuities
01 - Compress single-level loops
10 - Compress two-level loops
TBUFPWR
0 - Trace buffer is off
1 - Trace buffer is active
TBUFEN
0 - Trace buffer disabled
1 - Trace buffer enabled
TBUFOVF
0 - Overflows are ignored
1 - Trace buffer overflow causes an exception event

Figure 21-9. Trace Buffer Control Register

TBUFSTAT Register

Figure 21-10 shows the Trace Buffer Status register (TBUFSTAT). Two reads from TBUF decrements TBUFCNT by one.

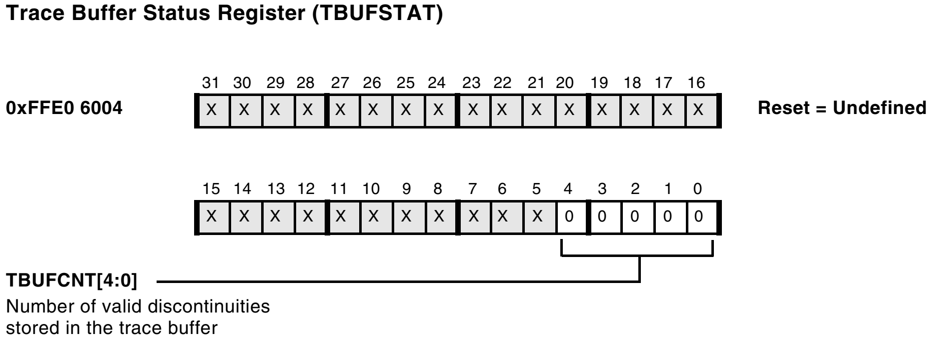

Trace Buffer Status Register (TBUFSTAT)
0xFFE0 6004
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X 0 0 0 0 0
TBUFCNT[4:0]
Number of valid discontinuities stored in the trace buffer

Figure 21-10. Trace Buffer Status Register

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.

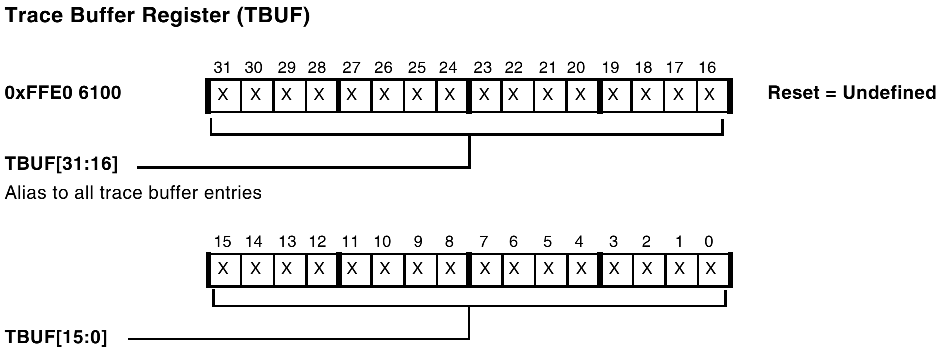


Figure 21-11. Trace Buffer Register

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.

Listing 21-1. Recreating the Execution Trace in Memory

```
[--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.

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.

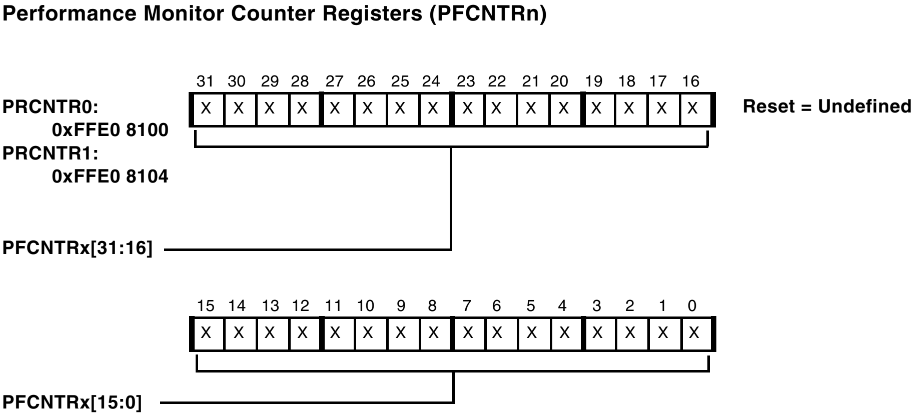


Figure 21-12. Performance Monitor Counter Registers

PFCTL Register

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.

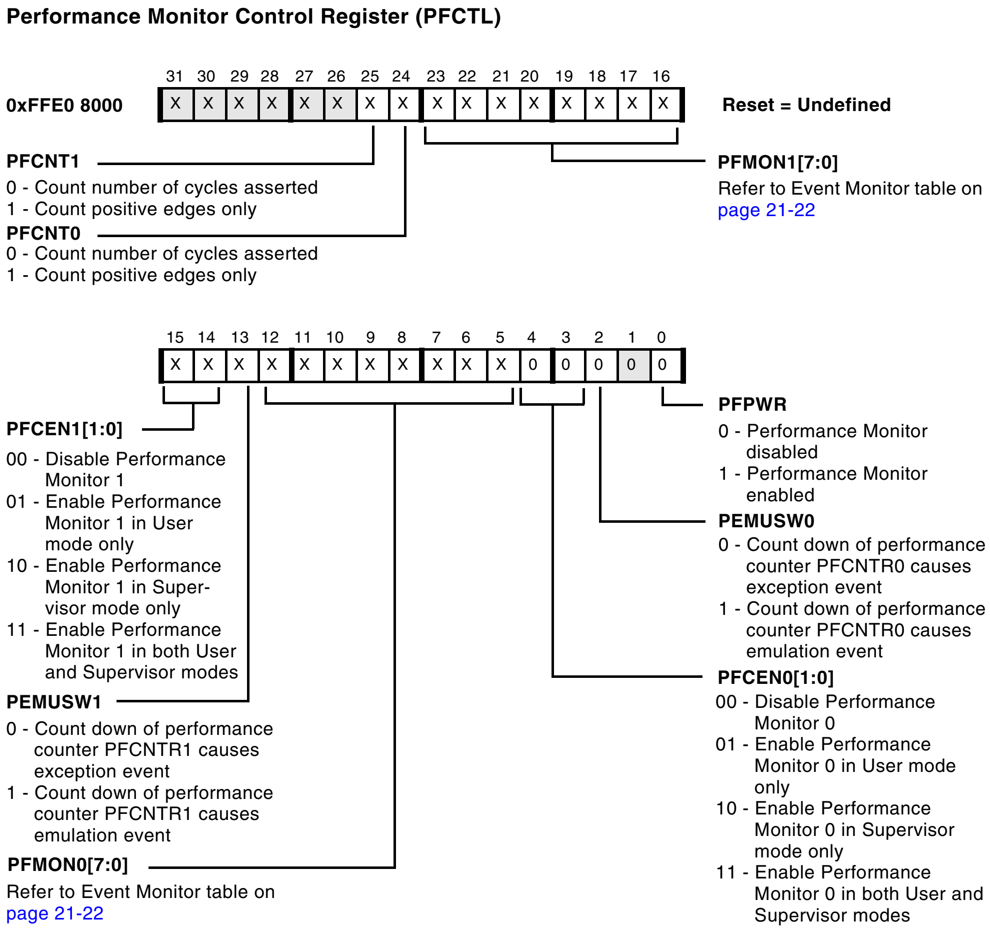

Performance Monitor Control Register (PFCTL)
0xFFE0 8000
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
PFCNT1
0 - Count number of cycles asserted
1 - Count positive edges only
PFCNT0
0 - Count number of cycles asserted
1 - Count positive edges only
PFMON1[7:0]
Refer to Event Monitor table on page 21-22
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X 0 0 0 0 0
PFCEN1[1:0]
00 - Disable Performance Monitor 1
01 - Enable Performance Monitor 1 in User mode only
10 - Enable Performance Monitor 1 in Super-visor mode only
11 - Enable Performance Monitor 1 in both User and Supervisor modes
PEMUSW1
0 - Count down of performance counter PFCNTR1 causes exception event
1 - Count down of performance counter PFCNTR1 causes emulation event
PFMON0[7:0]
Refer to Event Monitor table on page 21-22
PFPWR
0 - Performance Monitor disabled
1 - Performance Monitor enabled
PEMUSW0
0 - Count down of performance counter PFCNTR0 causes exception event
1 - Count down of performance counter PFCNTR0 causes emulation event
PFCEN0[1:0]
00 - Disable Performance Monitor 0
01 - Enable Performance Monitor 0 in User mode only
10 - Enable Performance Monitor 0 in Supervisor mode only
11 - Enable Performance Monitor 0 in both User and Supervisor modes

Figure 21-13. Performance Monitor Control Register

Event Monitor Table

Table 21-8 identifies events that cause the Performance Monitor Counter registers (PFMON0 or PFMON1) to increment.

Table 21-8. Event Monitor Table

PFMONx Fields	Events That Cause the Count Value to Increment
0x00	Loop 0 iterations
0x01	Loop 1 iterations
0x02	Loop buffer 0 not optimized
0x03	Loop buffer 1 not optimized
0x04	PC invariant branches (requires trace buffer to be enabled, see "TBUFCTL Register" on page 21-16)
0x06	Conditional branches
0x09	Total branches including calls, returns, branches, but not interrupts (requires trace buffer to be enabled, see "TBUFCTL Register" on page 21-16)
0x0A	Stalls due to CSYNC, SSYNC
0x0B	EXCPT instructions
0x0C	CSYNC, SSYNC instructions
0x0D	Committed instructions
0x0E	Interrupts taken
0x0F	Misaligned address violation exceptions
0x10	Stall cycles due to read after write hazards on DAG registers
0x13	Stall cycles due to RAW data hazards in computes
0x80	Code memory fetches postponed due to DMA collisions (minimum count of two per event)
0x81	Code memory TAG stalls (cache misses, or FlushI operations, count of 3 per FlushI). Note code memory stall results in a processor stall only if instruction assembly unit FIFO empties.
0x82	Code memory fill stalls (cacheable or non-cacheable). Note code memory stall results in a processor stall only if instruction assembly unit FIFO empties.
0x83	Code memory 64-bit words delivered to processor instruction assembly unit

Table 21-8. Event Monitor Table (Cont'd)

PFMONx Fields	Events That Cause the Count Value to Increment
0x90	Processor stalls to memory
0x91	Data memory stalls to processor not hidden by processor stall
0x92	Data memory store buffer full stalls
0x93	Data memory write buffer full stalls due to high-to-low priority code transition
0x94	Data memory store buffer forward stalls due to lack of committed data from processor
0x95	Data memory fill buffer stalls
0x96	Data memory array or TAG collision stalls (DAG to DAG, or DMA to DAG)
0x97	Data memory array collision stalls (DAG to DAG or DMA to DAG)
0x98	Data memory stalls
0x99	Data memory stalls sent to processor
0x9A	Data memory cache fills completed to Bank A
0x9B	Data memory cache fills completed to Bank B
0x9C	Data memory cache victims delivered from Bank A
0x9D	Data memory cache victims delivered from Bank B
0x9E	Data memory cache high priority fills requested
0x9F	Data memory cache low priority fills requested

Cycle Counter

The cycle counter counts `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.

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:

```
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.

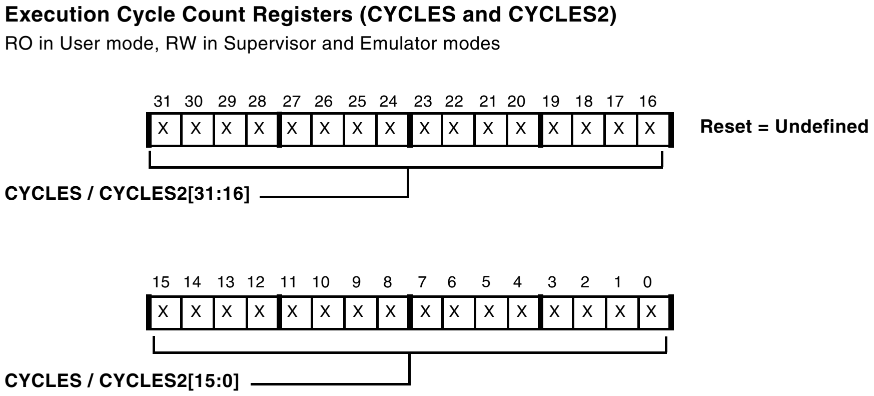

Execution Cycle Count Registers (CYCLES and CYCLES2)
RO in User mode, RW in Supervisor and Emulator modes
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
CYCLES / CYCLES2[31:16]
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
CYCLES / CYCLES2[15:0]

Figure 21-14. Execution Cycle Count Registers

SYSCFG Register

The System Configuration register (`SYSCFG`) controls the configuration of the processor. This register is accessible only from the Supervisor mode.

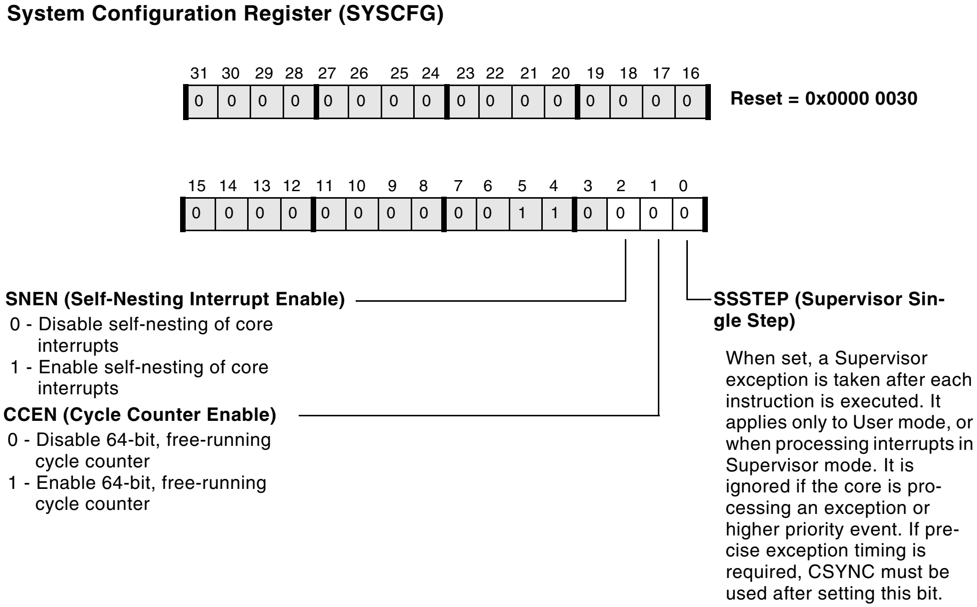

System Configuration Register (SYSCFG)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0030
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0
SNEN (Self-Nesting Interrupt Enable)
0 - Disable self-nesting of core interrupts
1 - Enable self-nesting of core interrupts
CCEN (Cycle Counter Enable)
0 - Disable 64-bit, free-running cycle counter
1 - Enable 64-bit, free-running cycle counter
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.

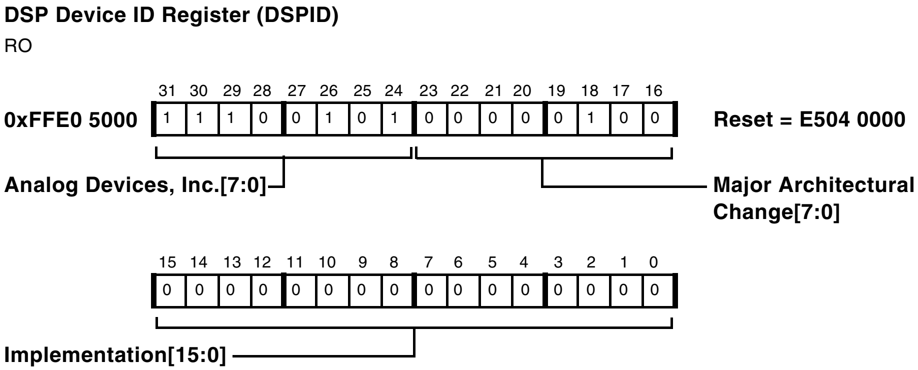


Figure 21-16. DSP Device ID Register

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.

Table A-1. ASTAT Flag Behavior for the ADSP-BF535

Instruction	CC	AZ	AN	AC0_ COPY	V_ COPY	AQ
Jump	–	–	–	–	–	–
IF CC JUMP	–	–	–	–	–	–
Call	–	–	–	–	–	–
RTS, RTI, RTX, RTN, RTE (Return)	–	–	–	–	–	–
LSETUP, LOOP	–	–	–	–	–	–
Load Immediate	–	–	–	–	–	–
Load Pointer Register	–	–	–	–	–	–
Load Data Register	–	–	–	–	–	–
Load Half-Word – Zero-Extended	–	–	–	–	–	–
Load Half-Word – Sign-Extended	–	–	–	–	–	–
Load High Data Register Half	–	–	–	–	–	–
Load Low Data Register Half	–	–	–	–	–	–
Load Byte – Zero-Extended	–	–	–	–	–	–
Load Byte – Sign-Extended	–	–	–	–	–	–
Store Pointer Register	–	–	–	–	–	–
Store Data Register	–	–	–	–	–	–
Store High Data Register Half	–	–	–	–	–	–
Store Low Data Register Half	–	–	–	–	–	–
Store Byte	–	–	–	–	–	–
Move Register (except acc to dreg)	–	–	–	–	–	–

– 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)

Instruction	CC	AZ	AN	AC0_ COPY	V_ COPY	AQ
Move Register (acc to dreg)	–	U	U	–	U	–
Move Conditional	–	–	–	–	–	–
Move Half to Full Word – Zero-Extended	–	*	**	**	**	–
Move Half to Full Word – Sign-Extended	–	*	*	**	**	–
Move Register Half (except acc to half dreg)	–	–	–	–	–	–
Move Register Half (acc to half dreg)	–	U	U	–	U	–
Move Byte – Zero-Extended	–	*	*	**	**	–
Move Byte – Sign-Extended	–	*	*	**	**	–
--SP (Push)	–	–	–	–	–	–
--SP (Push Multiple)	–	–	–	–	–	–
SP++ (Pop)	–	–	–	–	–	–
SP++ (Pop Multiple)	–	–	–	–	–	–
LINK, UNLINK	–	–	–	–	–	–
Compare Data Register	*	*	*	*	U	–
Compare Pointer	*	–	–	–	–	–
Compare Accumulator	*	*	*	*	U	–
Move CC	–	*	*	*	*	*
Negate CC	*	–	–	–	–	–
& (AND)	–	*	*	**	**	–
~ (NOT One's-Complement)	–	*	*	**	**	–

– 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)

Instruction	CC	AZ	AN	AC0_ COPY	V_ COPY	AQ
| (OR)	–	*	*	**	**	–
^ (Exclusive-OR)	–	*	*	**	**	–
BXORSHIFT, BXOR	*	–	–	–	–	–
BITCLR	–	*	*	U	U	–
BITSET	–	U	U	U	U	–
BITTGL	–	*	*	U	U	–
BITTST	*	–	–	–	–	–
DEPOSIT	–	*	*	U	U	–
EXTRACT	–	*	*	U	U	–
BITMUX	–	U	U	–	–	–
ONES (One's-Population Count)	–	U	U	–	–	–
Add with Shift (preg version)	–	–	–	–	–	–
Add with Shift (dreg version)	–	*	*	U	*	–
Shift with Add	–	–	–	–	–	–
Arithmetic Shift (to dreg)	–	*	*	U	*	–
Arithmetic Shift (to A0)	–	*	*	U	–	–
Arithmetic Shift (to A1)	–	*	*	U	–	–
Logical Shift (to preg)	–	U	U	U	U	–
Logical Shift (to dreg)	–	*	*	–	U	–
Logical Shift (to A0)	–	*	*	U	U	–
– 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)

Instruction	CC	AZ	AN	AC0_ COPY	V_ COPY	AQ
Logical Shift (to A1)	–	*	*	U	U	–
ROT (Rotate)	*	–	–	–	–	–
ABS (to dreg)	–	*	**	U	*	–
ABS (to A0)	–	*	**	U	U	–
ABS (to A1)	–	*	**	U	U	–
Add (preg version)	–	–	–	–	–	–
Add (dreg version)	–	*	*	*	*	–
Add/Subtract – Prescale Down	–	–	–	–	–	–
Add/Subtract – Prescale Up	–	–	–	–	–	–
Add Immediate (to preg or ireg)	–	–	–	–	–	–
Add Immediate (to dreg)	–	*	*	*	*	–
DIVS, DIVQ (Divide Primitive)	–	U	U	U	U	*
EXPADJ	–	U	U	–	–	–
MAX	–	*	*	U	U	–
MIN	–	*	*	U	U	–
Modify – Decrement (to preg or ireg)	–	–	–	–	–	–
Modify – Decrement (to acc)	–	U	U	U	–	–
Modify – Increment (to preg or ireg)	–	–	–	–	–	–
Modify – Increment (extracted to dreg)	–	*	*	*	*	–
Modify – Increment (to acc)	–	U	U	U	U	–

– 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)

Instruction	CC	AZ	AN	AC0_ COPY	V_ COPY	AQ
Multiply 16-Bit Operands	–	–	–	–	U	–
Multiply 32-Bit Operands	–	–	–	–	–	–
Multiply and Multiply-Accumulate to Accumulator	–	–	–	–	U	–
Multiply and Multiply-Accumulate to Half-Register	–	–	–	–	U	–
Multiply and Multiply-Accumulate to Data Register	–	–	–	–	U	–
Negate (Two's-Complement) (to dreg)	–	*	*	U	*	–
Negate (Two's-Complement) (to A0)	–	*	*	U	U	–
Negate (Two's-Complement) (to A1)	–	*	*	U	U	–
RND (Round to Half-Word)	–	*	*	U	*	–
Saturate	–	*	*	U	U	–
SIGNBITS	–	U	U	–	–	–
Subtract	–	*	*	*	*	–
Subtract Immediate (to ireg)	–	–	–	–	–	–
Idle	–	–	–	–	–	–
Core Synchronize	–	–	–	–	–	–
System Synchronize	–	–	–	–	–	–
EMUEXCPT (Force Emulation)	–	–	–	–	–	–
Disable Interrupts	–	–	–	–	–	–
Enable Interrupts	–	–	–	–	–	–
RAISE (Force Interrupt / Reset)	–	–	–	–	–	–

– 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)

Instruction	CC	AZ	AN	AC0_ COPY	V_ COPY	AQ
EXCPT (Force Exception)	–	–	–	–	–	–
Test and Set Byte (Atomic)	*	–	–	–	–	–
No Op	–	–	–	–	–	–
PREFETCH	–	–	–	–	–	–
FLUSH	–	–	–	–	–	–
FLUSHINV	–	–	–	–	–	–
IFLUSH	–	–	–	–	–	–
ALIGN8, ALIGN16, ALIGN24	–	U	U	–	–	–
DISALGNEXCPT	–	–	–	–	–	–
BYTEOP3P (Dual 16-Bit Add / Clip)	–	–	–	–	–	–
Dual 16-Bit Accumulator Extraction with Addition	–	–	–	–	–	–
BYTEOP16P (Quad 8-Bit Add)	–	U	U	U	U	–
BYTEOP1P (Quad 8-Bit Average – Byte)	–	U	U	U	U	–
BYTEOP2P (Quad 8-Bit Average – Half-Word)	–	U	U	U	U	–
BYTEPACK (Quad 8-Bit Pack)	–	U	U	U	U	–
BYTEOP16M (Quad 8-Bit Subtract)	–	U	U	U	U	–
SAA (Quad 8-Bit Subtract-Absolute-Accumulate)	–	U	U	U	U	–
BYTEUNPACK (Quad 8-Bit Unpack)	–	U	U	U	U	–
Add on Sign	–	U	U	U	U	–
VIT_MAX (Compare-Select)	–	U	U	–	–	–

– 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)

Instruction	CC	AZ	AN	AC0_ COPY	V_ COPY	AQ
Vector ABS	–	*	**	U	*	–
Vector Add / Subtract	–	*	**	*	*	–
Vector Arithmetic Shift	–	*	*	U	*	–
Vector Logical Shift	–	*	*	U	**	–
Vector MAX	–	*	*	U	**	–
Vector MIN	–	*	*	U	**	–
Vector Multiply	–	–	–	–	U	–
Vector Multiply and Multiply-Accumulate	–	–	–	–	*	–
Vector Negate (Two's-Complement)	–	*	*	*	*	–
Vector PACK	–	U	U	–	–	–
Vector SEARCH	–	U	U	–	–	–

– 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.

B CORE MMR ASSIGNMENTS

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

Memory-mapped Address	Register Name	See Section
0xFFE0 0004	DMEM_CONTROL	"DMEM_CONTROL Register" on page 6-24
0xFFE0 0008	DCPLB_STATUS	"DCPLB_STATUS and ICPLB_STATUS Registers" on page 6-61
0xFFE0 000C	DCPLB_FAULT_ADDR	"DCPLB_FAULT_ADDR and ICPLB_FAULT_ADDR Registers" on page 6-63
0xFFE0 0100	DCPLB_ADDR0	"DCPLB_ADDRx Registers" on page 6-59

Table B-1. L1 Data Memory Controller Registers (Cont'd)

Memory-mapped Address	Register Name	See Section
0xFFE0 0104	DCPLB_ADDR1	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 0108	DCPLB_ADDR2	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 010C	DCPLB_ADDR3	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 0110	DCPLB_ADDR4	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 0114	DCPLB_ADDR5	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 0118	DCPLB_ADDR6	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 011C	DCPLB_ADDR7	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 0120	DCPLB_ADDR8	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 0124	DCPLB_ADDR9	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 0128	DCPLB_ADDR10	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 012C	DCPLB_ADDR11	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 0130	DCPLB_ADDR12	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 0134	DCPLB_ADDR13	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 0138	DCPLB_ADDR14	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 013C	DCPLB_ADDR15	"DCPLB_ADDRx Registers" on page 6-59
0xFFE0 0200	DCPLB_DATA0	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 0204	DCPLB_DATA1	"DCPLB_DATAx Registers" on page 6-57
0 xFFE0 0208	DCPLB_DATA2	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 020C	DCPLB_DATA3	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 0210	DCPLB_DATA4	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 0214	DCPLB_DATA5	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 0218	DCPLB_DATA6	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 021C	DCPLB_DATA7	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 0220	DCPLB_DATA8	"DCPLB_DATAx Registers" on page 6-57

Table B-1. L1 Data Memory Controller Registers (Cont'd)

Memory-mapped Address	Register Name	See Section
0xFFE0 0224	DCPLB_DATA9	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 0228	DCPLB_DATA10	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 022C	DCPLB_DATA11	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 0230	DCPLB_DATA12	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 0234	DCPLB_DATA13	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 0238	DCPLB_DATA14	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 023C	DCPLB_DATA15	"DCPLB_DATAx Registers" on page 6-57
0xFFE0 0300	DTEST_COMMAND	"DTEST_COMMAND Register" on page 6-39
0xFFE0 0400	DTEST_DATA0	"DTEST_DATA0 Register" on page 6-42
0xFFE0 0404	DTEST_DATA1	"DTEST_DATA1 Register" on page 6-41

L1 Instruction Memory Controller Registers

L1 Instruction Memory Controller registers (0xFFE0 1004 –0xFFE0 1404)

Table B-2. L1 Instruction Memory Controller Registers

Memory-mapped Address	Register Name	See Section
0xFFE0 1004	IMEM_CONTROL	"IMEM_CONTROL Register" on page 6-5
0xFFE0 1008	ICPLB_STATUS	"DCPLB_STATUS and ICPLB_STATUS Registers" on page 6-61
0xFFE0 100C	ICPLB_FAULT_ADDR	"DCPLB_FAULT_ADDR and ICPLB_FAULT_ADDR Registers" on page 6-63
0xFFE0 1100	ICPLB_ADDR0	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 1104	ICPLB_ADDR1	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 1108	ICPLB_ADDR2	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 110C	ICPLB_ADDR3	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 1110	ICPLB_ADDR4	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 1114	ICPLB_ADDR5	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 1118	ICPLB_ADDR6	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 111C	ICPLB_ADDR7	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 1120	ICPLB_ADDR8	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 1124	ICPLB_ADDR9	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 1128	ICPLB_ADDR10	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 112C	ICPLB_ADDR11	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 1130	ICPLB_ADDR12	"ICPLB_ADDRx Registers" on page 6-60

Table B-2. L1 Instruction Memory Controller Registers (Cont'd)

Memory-mapped Address	Register Name	See Section
0xFFE0 1134	ICPLB_ADDR13	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 1138	ICPLB_ADDR14	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 113C	ICPLB_ADDR15	"ICPLB_ADDRx Registers" on page 6-60
0xFFE0 1200	ICPLB_DATA0	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1204	ICPLB_DATA1	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1208	ICPLB_DATA2	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 120C	ICPLB_DATA3	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1210	ICPLB_DATA4	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1214	ICPLB_DATA5	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1218	ICPLB_DATA6	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 121C	ICPLB_DATA7	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1220	ICPLB_DATA8	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1224	ICPLB_DATA9	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1228	ICPLB_DATA10	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 122C	ICPLB_DATA11	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1230	ICPLB_DATA12	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1234	ICPLB_DATA13	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1238	ICPLB_DATA14	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 123C	ICPLB_DATA15	"ICPLB_DATAx Registers" on page 6-55
0xFFE0 1300	ITEST_COMMAND	"ITEST_COMMAND Register" on page 6-21
0XFFE0 1400	ITEST_DATA0	"ITEST_DATA0 Register" on page 6-23
0XFFE0 1404	ITEST_DATA1	"ITEST_DATA1 Register" on page 6-22

Interrupt Controller Registers

Interrupt Controller registers (0xFFE0 2000 – 0xFFE0 2110)

Table B-3. Interrupt Controller Registers

Memory-mapped Address	Register Name	See Section
0xFFE0 2000	EVT0 (EMU)	"Core Event Vector Table" on page 4-42
0xFFE0 2004	EVT1 (RST)	"Core Event Vector Table" on page 4-42
0xFFE0 2008	EVT2 (NMI)	"Core Event Vector Table" on page 4-42
0xFFE0 200C	EVT3 (EVX)	"Core Event Vector Table" on page 4-42
0xFFE0 2010	EVT4	"Core Event Vector Table" on page 4-42
0xFFE0 2014	EVT5 (IVHW)	"Core Event Vector Table" on page 4-42
0xFFE0 2018	EVT6 (TMR)	"Core Event Vector Table" on page 4-42
0xFFE0 201C	EVT7 (IVG7)	"Core Event Vector Table" on page 4-42
0xFFE0 2020	EVT8 (IVG8)	"Core Event Vector Table" on page 4-42
0xFFE0 2024	EVT9 (IVG9)	"Core Event Vector Table" on page 4-42
0xFFE0 2028	EVT10 (IVG10)	"Core Event Vector Table" on page 4-42
0xFFE0 202C	EVT11 (IVG11)	"Core Event Vector Table" on page 4-42
0xFFE0 2030	EVT12 (IVG12)	"Core Event Vector Table" on page 4-42

Table B-3. Interrupt Controller Registers (Cont'd)

Memory-mapped Address	Register Name	See Section
0xFFE0 2034	EVT13 (IVG13)	"Core Event Vector Table" on page 4-42
0xFFE0 2038	EVT14 (IVG14)	"Core Event Vector Table" on page 4-42
0xFFE0 203C	EVT15 (IVG15)	"Core Event Vector Table" on page 4-42
0xFFE0 2104	IMASK	"IMASK Register" on page 4-38
0xFFE0 2108	IPEND	"IPEND Register" on page 4-40
0xFFE0 210C	ILAT	"ILAT Register" on page 4-39
0xFFE0 2110	IPRIO	"IPRIO Register and Write Buffer Depth" on page 6-35

Debug, MP, and Emulation Unit Registers

Debug, MP, and Emulation Unit registers (0xFFE0 5000 – 0xFFE0 5008)

Table B-4. Debug and Emulation Unit Registers

Memory-mapped Address	Register Name	See Section
0xFFE0 5000	DSPID	"DSPID Register" on page 21-27

Trace Unit Registers

Trace Unit registers (0xFFE0 6000 – 0xFFE0 6100)

Table B-5. Trace Unit Registers

Memory-mapped Address	Register Name	See Section
0xFFE0 6000	TBUFCTL	“TBUFCTL Register” on page 21-16
0xFFE0 6004	TBUFSTAT	“TBUFSTAT Register” on page 21-17
0xFFE0 6100	TBUF	“TBUF Register” on page 21-18

Watchpoint and Patch Registers

Watchpoint and Patch registers (0xFFE0 7000 – 0xFFE0 7200)

Table B-6. Watchpoint and Patch Registers

Memory-mapped Address	Register Name	See Section
0xFFE0 7000	WPIACTL	“WPIACTL Register” on page 21-7
0xFFE0 7040	WPIA0	“WPIAn Registers” on page 21-5
0xFFE0 7044	WPIA1	“WPIAn Registers” on page 21-5
0xFFE0 7048	WPIA2	“WPIAn Registers” on page 21-5
0xFFE0 704C	WPIA3	“WPIAn Registers” on page 21-5
0xFFE0 7050	WPIA4	“WPIAn Registers” on page 21-5
0xFFE0 7054	WPIA5	“WPIAn Registers” on page 21-5
0xFFE0 7080	WPIACNT0	“WPIACNTn Registers” on page 21-6
0xFFE0 7084	WPIACNT1	“WPIACNTn Registers” on page 21-6
0xFFE0 7088	WPIACNT2	“WPIACNTn Registers” on page 21-6

Table B-6. Watchpoint and Patch Registers (Cont'd)

Memory-mapped Address	Register Name	See Section
0xFFE0 708C	WPIACNT3	"WPIACNTn Registers" on page 21-6
0xFFE0 7090	WPIACNT4	"WPIACNTn Registers" on page 21-6
0xFFE0 7094	WPIACNT5	"WPIACNTn Registers" on page 21-6
0xFFE0 7100	WPDACTL	"WPDACTL Register" on page 21-12
0xFFE0 7140	WPDA0	"WPDAn Registers" on page 21-10
0xFFE0 7144	WPDA1	"WPDAn Registers" on page 21-10
0xFFE0 7180	WPDACNT0	"WPDACNTn Registers" on page 21-11
0xFFE0 7184	WPDACNT1	"WPDACNTn Registers" on page 21-11
0xFFE0 7200	WPSTAT	"WPSTAT Register" on page 21-14

Performance Monitor Registers

Performance Monitor registers (0xFFE0 8000 – 0xFFE0 8104)

Table B-7. Performance Monitor Registers

Memory-mapped Address	Register Name	See Section
0xFFE0 8000	PFCTL	"PFCTL Register" on page 21-20
0xFFE0 8100	PFCNTR0	"PFCNTRn Registers" on page 21-20
0xFFE0 8104	PFCNTR1	"PFCNTRn Registers" on page 21-20

C INSTRUCTION OPCODES

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.

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

Register	Description
Accumulators	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.
Data Registers	The set of 32-bit registers R0, R1, …, R6, 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.
Pointer Registers	The set of 32-bit registers P0, P1, …, 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.
Stack Pointer	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.
Frame Pointer	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.
Loop Top	LT0 and LT1; contains 32-bit address of the top of a zero overhead loop.
Loop Count	LC0 and LC1; contains 32-bit counter of the zero overhead loop executions.
Loop Bottom	LB0 and LB1; contains 32-bit address of the bottom of a zero overhead loop.

Table C-1. Registers (Cont'd)

Register	Description
Index Register	The set of 32-bit registers I0, I1, I2, I3 that normally contain byte addresses of data structures. Abbreviated I-register or Ireg.
Modify Registers	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.
Length Registers	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.
Base Registers	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.

Functional Units

The architecture includes three processor sections.

Table C-2. Processor Sections

Processor	Description
Data Address Generator (DAG)	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.
Multiply and Accumulate Unit (MAC)	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.
Arithmetic Logical Unit (ALU)	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.

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.

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.

- 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.

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

Flag	Description
AC0	Carry (ALU0)
AC1	Carry (ALU1)
AN	Negative
AQ	Quotient
AV0	Accumulator 0 Overflow
AVS0	Accumulator 0 Sticky Overflow; set when AV0 is set, but remains set until explicitly cleared by user code
AV1	Accumulator 1 Overflow
AVS1	Accumulator 1 Sticky Overflow; set when AV1 is set, but remains set until explicitly cleared by user code
AZ	Zero
CC	Control Code bit; multipurpose flag set, cleared and tested by specific instructions
V	Overflow for Data Register results
VS	Sticky Overflow for Data Register results; set when V is set, but remains set until explicitly cleared by user code

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

	REGISTER NUMBER							
REGISTER GROUP	0	1	2	3	4	5	6	7
0	R0	R1	R2	R3	R4	R5	R6	R7
1	P0	P1	P2	P3	P4	P5	SP	FP
2	I0	I1	I2	I3	M0	M1	M2	M3
3	B0	B1	B2	B3	L0	L1	L2	L3
4	A0.x	A0.w	A1.x	A1.w	<res.>	<res.>	ASTAT	RETS
5	<res.>	<res.>	<res.>	<res.>	<res.>	<res.>	<res.>	<res.>
6	LC0	LT0	LB0	LC1	LT1	LB1	CYCLES	CYCLES2
7	USP	SEQSTAT	SYSCFG	RETI	RETX	RETN	RETE	EMUDAT

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

Instruction									Bin								
and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Instruction Name	Single Hex Value	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit
Syntax without variable arguments (16-bit Instruction)																	
Instruction Name	Min. Value—Max. Value	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit
Syntax with variable arguments (16-bit Instruction)																	
Instruction Name	Single Hex Value	bit	bit	bit	bit	bit	Most significant bits					bit	bit	bit	bit	bit	
		bit	bit	bit	bit	bit	Least significant bits					bit	bit	bit	bit	bit	
Syntax without variable arguments (32-bit Instruction)																	
Instruction Name	Min. Value—Max. Value	bit	bit	bit	bit	bit	Most significant bits					bit	bit	bit	bit	bit	
		bit	bit	bit	bit	bit	Least significant bits					bit	bit	bit	bit	bit	
Syntax with variable arguments (32-bit Instruction)																	

Opcode Bit Terminology

The following conventions describe the instruction opcode bit states.

Table C-6.

SYMBOL	MEANING
0	Binary zero bit, logical “low”
1	Binary one bit, logical “high”
x	“don’t care” bit

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.

Table C-7. Sample Opcode Holes Representation

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Zero Overhead Loop Setup	0xE080 0000— 0xE08F 03FF	1	1	1	0	0	0	0	0	1	0	0	0	pcrel5m2 divided by 2			
		0	0	0	0	0	0	pcrel11m2 divided by 2									
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.																	

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

Relative Byte Address	Data
0	0xED
1	0xFE
2	0xCE
3	0xFA

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)

Instruction		Bin															
and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Jump JUMP (Preg)	0x0050—0x0057	0	0	0	0	0	0	0	0	0	1	0	1	Preg #			
Jump JUMP (PC+Preg)	0x0080—0x0087	0	0	0	0	0	0	0	0	1	0	0	0	Preg #			
Jump JUMP.S pcrel13m2	0x2000—0x2FFF	0	0	1	0	pcrel13m2 divided by 2											
Jump JUMP.L pcrel25m2	0xE200 0000—0xE2FF FFFF	1	1	1	0	0	0	1	0	Most significant bits of pcrel25m2							
		Least significant bits of pcrel25m2 divided by 2															
Conditional Jump IF CC JUMP pcrel11m2	0x1800—0x17FF	0	0	0	1	x	x	pcrel11m2 divided by 2									
Conditional Jump IF CC JUMP pcrel11m2 (bp)	0x1C00—0x1FFF	0	0	0	1	1	1	pcrel11m2 divided by 2									
Conditional Jump IF !CC JUMP pcrel11m2	0x1000—0x13FF	0	0	0	1	0	0	pcrel11m2 divided by 2									
Conditional Jump IF !CC JUMP pcrel11m2 (bp)	0x1400—0x1BFF	0	0	0	1	0	1	pcrel11m2 divided by 2									
Call CALL (Preg)	0x0060—0x0067	0	0	0	0	0	0	0	0	0	1	1	0	Preg #			

Table C-9. Program Flow Control Instructions (Sheet 2 of 3)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Call	0x0070— 0x0077	0	0	0	0	0	0	0	0	0	1	1	1	Preg #			
CALL (PC+Preg)																	
Call	0xE300 0000— 0xE3FF FFFF	1	1	1	0	0	0	1	1	Most significant bits of pcrel25m2							
		Least significant bits of pcrel25m2 divided by 2															
CALL pcrel25m2																	
Return	0x0010—	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
RTS																	
Return	0x0011—	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
RTI																	
Return	0x0012—	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0
RTX																	
Return	0x0013—	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1
RTN																	
Return	0x0014—	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0
RTE																	
Zero Overhead Loop Setup	0xE080 0000— 0xE08F 03FF	1	1	1	0	0	0	0	0	1	0	0	0	pcrel5m2 divided by 2			
		0	0	0	0	x	x	pcrel11m2 divided by 2									
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.																	
Zero Overhead Loop Setup	0xE0A0 0000— 0xE0AF F3FF	1	1	1	0	0	0	0	0	1	0	1	0	pcrel5m2 divided by 2			
		Preg #				x	x	pcrel11m2 divided by 2									
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.																	

Table C-9. Program Flow Control Instructions (Sheet 3 of 3)

Instruction		Bin															
and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Zero Overhead Loop Setup	0xE0E0 0000— 0xE0AF F3FF	1	1	1	0	0	0	0	0	1	1	1	0	pcrel5m2 divided by 2			
		Preg #				x	x	pcrel11m2 divided by 2									
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.																	
Zero Overhead Loop Setup	0xE090 0000— 0xE09F 03FF	1	1	1	0	0	0	0	0	1	0	0	1	pcrel5m2 divided by 2			
		0	0	0	0	x	x	pcrel11m2 divided by 2									
LOOP loop_name LC1 LOOP_BEGIN loop_name LOOP_END loop_name ... is mapped to... LSETUP (pcrel5m2, pcrel11m2) LC1 ... where the address of LOOP_BEGIN determines pcrel5m2, and the address of LOOP_END determines pcrel11m2.																	
Zero Overhead Loop Setup	0xE0B0 0000— 0xE0BF F3FF	1	1	1	0	0	0	0	0	1	0	1	1	pcrel5m2 divided by 2			
		Preg #				x	x	pcrel11m2 divided by 2									
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.																	
Zero Overhead Loop Setup	0xE0F0 0000— 0xE0FF F3FF	1	1	1	0	0	0	0	0	1	1	1	1	pcrel5m2 divided by 2			
		Preg #				x	x	pcrel11m2 divided by 2									
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)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Load Immediate	0xE100 0000— 0xE11F FFFF	1	1	1	0	0	0	0	1	0	0	0	Reg grp. #		Reg #		
		uimm16															
reg_lo = uimm16																	
Load Immediate	0xE140 0000— 0xE15F FFFF	1	1	1	0	0	0	0	1	0	1	0	Reg grp. #		Reg #		
		uimm16															
reg_hi = uimm16																	
Load Immediate	0xE180 0000— 0xE19F FFFF	1	1	1	0	0	0	0	1	1	0	0	Reg grp. #		Reg #		
		uimm16															
reg = uimm16 (Z)																	
Load Immediate	0xC408 003F	1	1	0	0	0	1	0	x	x	x	0	0	1	0	0	0
		0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
A0 = 0																	
Load Immediate	0xC408 403F	1	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0
		0	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1
A1 = 0																	
Load Immediate	0xC408 803F	1	1	0	0	0	1	0	x	x	x	0	0	1	0	0	0
		1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
A1 = A0 = 0																	
Load Immediate	0x6000— 0x63FF	0	1	1	0	0	0	imm7							Dreg #		
Dreg = imm7 (X)																	
Load Immediate	0x6800— 0x6BFF	0	1	1	0	1	0	imm7							Preg #		
Preg = imm7 (X)																	

Table C-10. Load / Store Instructions (Sheet 2 of 12)

Instruction		Bin															
and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Load Immediate	0xE120 0000— 0xE13F FFFF	1	1	1	0	0	0	0	1	0	0	1	Reg grp. #		Reg #		
		imm16															
reg = imm16 (X)																	
Load Pointer Register	0x9140— 0x917F	1	0	0	1	0	0	0	1	0	1	Source Preg #			Dest. Preg #		
Preg = [Preg]																	
Load Pointer Register	0x9040— 0x907F	1	0	0	1	0	0	0	0	0	1	Source Preg #			Dest. Preg #		
Preg = [Preg ++]																	
Load Pointer Register	0x90C0— 0x90FF	1	0	0	1	0	0	0	0	1	1	Source Preg #			Dest. Preg #		
Preg = [Preg – –]																	
Load Pointer Register	0xAC00— 0xAFFF	1	0	1	0	1	1	uimm6m4 divided by 4				Source Preg #			Dest. Preg #		
Preg = [Preg + uimm6m4]																	
Load Pointer Register	0xE500 0000— 0xE53F 7FFF	1	1	1	0	0	1	0	1	0	0	Source Preg #			Dest. Preg #		
		uimm17m4 divided by 4															
Preg = [Preg + uimm17m4]																	
Load Pointer Register	0xE500 8000— 0xE53F FFFF	1	1	1	0	0	1	0	1	0	0	Source Preg #			Dest. Preg #		
		uimm17m4 divided by 4															
Preg = [Preg – uimm17m4]																	
Load Pointer Register	0xB808— 0xB9FF	1	0	1	1	1	0	0	uimm7m4 divided by 4						Preg #		
Preg = [FP – uimm7m4]																	
Load Data Register	0x9100— 0x913F	1	0	0	1	0	0	0	1	0	0	Preg #			Dreg #		
Dreg = [Preg]																	

Table C-10. Load / Store Instructions (Sheet 3 of 12)

Instruction		Bin															
and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Load Data Register	0x9000— 0x903F	1	0	0	1	0	0	0	0	0	0	Preg #			Dreg #		
Dreg = [Preg ++]																	
Load Data Register	0x9080— 0x90BF	1	0	0	1	0	0	0	0	1	0	Preg #			Dreg #		
Dreg = [Preg – –]																	
Load Data Register	0xA000— 0xA3FF	1	0	1	0	0	0	uimm6m4 divided by 4				Preg #			Dreg #		
Dreg = [Preg + uimm6m4]																	
Load Data Register	0xE400 0000— 0xE4EF 7FFF	1	1	1	0	0	1	0	0	0	0	Preg #			Dreg #		
		uimm17m4 divided by 4															
Dreg = [Preg + uimm17m4]																	
Load Data Register	0xE400 8000— 0xE43F FFFF	1	1	1	0	0	1	0	0	0	0	Preg #			Dreg #		
		uimm17m4 divided by 4															
Dreg = [Preg – uimm17m4]																	
Load Data Register	0x8000— 0x81FF	1	0	0	0	0	0	0	Dest. Dreg #			Index Preg #			Pointer Preg #		
Dreg = [Preg ++ Preg]																	
NOTE: Pointer Preg number cannot be the same as Index Preg number. If so, this opcode represents a non-post-modify instruction version.																	
Load Data Register	0xB800— 0xB9F7	1	0	1	1	1	0	0	uimm7m4 divided by 4					Dreg #			
Dreg = [FP – uimm7m4]																	
Load Data Register	0x9D00— 0x9D1F	1	0	0	1	1	1	0	1	0	0	0	Ireg #		Dreg #		
Dreg = [Ireg]																	
Load Data Register	0x9C00— 0x9C1F	1	0	0	1	1	1	0	0	0	0	0	Ireg #		Dreg #		
Dreg = [Ireg ++]																	

Table C-10. Load / Store Instructions (Sheet 4 of 12)

Instruction									Bin								
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Load Data Register	0x9C80— 0x9C9F	1	0	0	1	1	1	0	0	1	0	0	Ireg #		Dreg #		
Dreg = [Ireg – –]																	
Load Data Register	0x9D80— 0x9DFF	1	0	0	1	1	1	0	1	1	Mreg #		Ireg #		Dreg #		
Dreg = [Ireg ++ Mreg]																	
Load Half Word, Zero Extended	0x9500— 0x953F	1	0	0	1	0	1	0	1	0	0	Preg #			Dreg #		
Dreg = W [Preg] (Z)																	
Load Half Word , Zero Extended	0x9400— 0x943F	1	0	0	1	0	1	0	0	0	0	Preg #			Dreg #		
Dreg = W [Preg ++] (Z)																	
Load Half Word, Zero Extended	0x9480— 0x94BF	1	0	0	1	0	1	0	0	1	0	Preg #			Dreg #		
Dreg = W [Preg —] (Z)																	
Load Half Word, Zero Extended	0xA400— 0xA7FF	1	0	1	0	0	1	uimm5m2 divided by 2				Preg #			Dreg #		
Dreg = W [Preg + uimm5m2] (Z)																	
Load Half Word, Zero Extended	0xE440 0000— 0xE47F 8FFF	1	1	1	0	0	1	0	0	0	1	Preg #			Dreg #		
		uimm16m2 divided by 2															
Dreg = W [Preg + uimm16m2] (Z)																	
Load Half Word, Zero Extended	0xE440 8000— 0xE47F FFFF	1	1	1	0	0	1	0	0	0	1	Preg #			Dreg #		
		uimm16m2 divided by 2															
Dreg = W [Preg – uimm16m2] (Z)																	
Load Half Word, Zero Extended	0x8601— 0x87FE	1	0	0	0	0	1	1	Dest. Dreg #			Index Preg #			Pointer Preg #		
Dreg = W [Preg ++ Preg] (Z)																	

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 5 of 12)

Instruction		Bin															
and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Load Half Word, Sign Extended	0x9540— 0x957F	1	0	0	1	0	1	0	1	0	1	Preg #			Dreg #		
Dreg = W [Preg] (X)																	
Load Half Word, Sign Extended	0x9440— 0x947F	1	0	0	1	0	1	0	0	0	1	Preg #			Dreg #		
Dreg = W [Preg ++] (X)																	
Load Half Word, Sign Extended	0x94C0— 0x94FF	1	0	0	1	0	1	0	0	1	1	Preg #			Dreg #		
Dreg = W [Preg ––] (X)																	
Load Half Word, Sign Extended	0xA800— 0xABFF	1	0	1	0	1	0	uimm5m2 divided by 2				Preg #			Dreg #		
Dreg = W [Preg + uimm5m2] (X)																	
Load Half Word, Sign Extended	0xE540 0000— 0xE57F 8FFF	1	1	1	0	0	1	0	1	0	1	Preg #			Dreg #		
		uimm16m2 divided by 2															
Dreg = W [Preg + uimm16m2] (X)																	
Load Half Word, Sign Extended	0xE540 8000— 0xE57F FFFF	1	1	1	0	0	1	0	1	0	1	Preg #			Dreg #		
		uimm16m2 divided by 2															
Dreg = W [Preg – uimm16m2] (X)																	
Load Half Word, Sign Extended	0x8E00— 0x8FFF	1	0	0	0	1	1	1	Dest. Dreg #			Index Preg #			Pointer Preg #		
Dreg = W [Preg ++ Preg] (X)																	
NOTE: Pointer Preg number cannot be the same as Index Preg number. If so, this opcode represents a non-post-modify instruction version.																	
Load High Data Register Half	0x9D40— 0x9D5F	1	0	0	1	1	1	0	1	0	1	0	Ireg #		Dreg #		
Dreg_hi = W [Ireg]																	
Load High Data Register Half	0x9C40— 0x9C5F	1	0	0	1	1	1	0	0	0	1	0	Ireg #		Dreg #		
Dreg_hi = W [Ireg ++]																	

Table C-10. Load / Store Instructions (Sheet 6 of 12)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Load High Data Register Half Dreg_hi = W [Ireg – –]	0x9CC0— 0x9CDF	1	0	0	1	1	1	0	0	1	1	0	Ireg #		Dreg #		
Load High Data Register Half Dreg_hi = W [Preg] 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.	0x8400— 0x85FF	1	0	0	0	0	1	0	Dest. Dreg #			Pointer Preg #			Pointer Preg #		
Load High Data Register Half Dreg_hi = W [Preg ++ Preg] NOTE: Pointer Preg number cannot be the same as Index Preg number. If so, this opcode represents a non-post-modify instruction version.	0x8401— 0x85FE	1	0	0	0	0	1	0	Dest. Dreg #			Index Preg #			Pointer Preg #		
Load Low Data Register Half Dreg_lo = W [Ireg]	0x9D20— 0x9D3F	1	0	0	1	1	1	0	1	0	0	1	Ireg #		Dreg #		
Load Low Data Register Half Dreg_lo = W [Ireg ++]	0x9C20— 0x9C3F	1	0	0	1	1	1	0	0	0	0	1	Ireg #		Dreg #		
Load Low Data Register Half Dreg_lo = W [Ireg – –]	0x9CA0— 0x9CBF	1	0	0	1	1	1	0	0	1	0	1	Ireg #		Dreg #		
Load Low Data Register Half Dreg_lo = W [Preg] NOTE: Both Pointer Preg # fields must refer to the same Preg number. Otherwise, this opcode represents a post-modify version of this instruction.	0x8200— 0x83FF	1	0	0	0	0	0	1	Dest. Dreg #			Pointer Preg #			Pointer Preg #		
Load Low Data Register Half Dreg_lo = W [Preg ++ Preg] NOTE: Pointer Preg number cannot be the same as Index Preg number. If so, this opcode represents a non-post-modify instruction version.	0x8201— 0x83FE	1	0	0	0	0	0	1	Dest. Dreg #			Index Preg #			Pointer Preg #		

Table C-10. Load / Store Instructions (Sheet 7 of 12)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Load Byte, Zero Extended	0x9900— 0x993F	1	0	0	1	1	0	0	1	0	0	Preg #			Dreg #		
Dreg = B [Preg] (Z)																	
Load Byte, Zero Extended	0x9800— 0x983F	1	0	0	1	1	0	0	0	0	0	Preg #			Dreg #		
Dreg = B [Preg ++] (Z)																	
Load Byte, Zero Extended	0x9880— 0x98BF	1	0	0	1	1	0	0	0	1	0	Preg #			Dreg #		
Dreg = B [Preg —] (Z)																	
Load Byte, Zero Extended	0xE480 0000— 0xE4BF 7FFF	1	1	1	0	0	1	0	0	1	0	Preg #			Dreg #		
		uimm15															
Dreg = B [Preg + uimm15] (Z)																	
Load Byte, Zero Extended	0xE480 8000— 0xE4BF FFFF	1	1	1	0	0	1	0	0	1	0	Preg #			Dreg #		
		uimm15															
Dreg = B [Preg – uimm15] (Z)																	
Load Byte, Sign Extended	0x9940— 0x997F	1	0	0	1	1	0	0	1	0	1	Preg #			Dreg #		
Dreg = B [Preg] (X)																	
Load Byte, Sign Extended	0x9840— 0x987F	1	0	0	1	1	0	0	0	0	1	Preg #			Dreg #		
Dreg = B [Preg ++] (X)																	
Load Byte, Sign Extended	0x98C0— 0x98FF	1	0	0	1	1	0	0	0	1	1	Preg #			Dreg #		
Dreg = B [Preg —] (X)																	
Load Byte, Sign Extended	0xE580 0000— 0xE5BF 7FFF	1	1	1	0	0	1	0	1	1	0	Preg #			Dreg #		
		uimm15															
Dreg = B [Preg + uimm15] (X)																	

Table C-10. Load / Store Instructions (Sheet 8 of 12)

Instruction		Bin															
and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Load Byte, Sign Extended	0xE580 8000— 0xE5BF FFFF	1	1	1	0	0	1	0	1	1	0	Preg #			Dreg #		
		uimm15															
Dreg = B [Preg – uimm15] (X)																	
Store Pointer Register	0x9340— 0x937F	1	0	0	1	0	0	1	1	0	1	Dest. Pointer Preg #			Source Preg #		
[Preg] = Preg																	
Store Pointer Register	0x9240— 0x927F	1	0	0	1	0	0	1	0	0	1	Dest. Pointer Preg #			Source Preg #		
[Preg ++] = Preg																	
Store Pointer Register	0x92C0— 0x92FF	1	0	0	1	0	0	1	0	1	1	Dest. Pointer Preg #			Source Preg #		
[Preg – –] = Preg																	
Store Pointer Register	0xBC00— 0xBFFF	1	0	1	1	1	1	uimm6m4 divided by 4				Source Pointer Preg #			Dest. Preg #		
[Preg + uimm6m4] = Preg																	
Store Pointer Register	0xE700 0000— 0xE7EF 8FFF	1	1	1	0	0	1	1	1	0	0	Dest. Pointer Preg #			Source Preg #		
		uimm17m4 divided by 4															
[Preg + uimm17m4] = Preg																	
Store Pointer Register	0xE700 8000— 0xE73F FFFF	1	1	1	0	0	1	1	1	0	0	Dest. Pointer Preg #			Source Preg #		
		uimm17m4 divided by 4															
[Preg – uimm17m4] = Preg																	

Table C-10. Load / Store Instructions (Sheet 9 of 12)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Store Pointer Register [FP – uimm7m4] = Preg	0xBA08— 0xBBFF	1	0	1	1	1	0	1	uimm7m4 divided by 4					Preg #			
Store Data Register [Preg] = Dreg	0x9300— 0x933F	1	0	0	1	0	0	1	1	0	0	Preg #			Dreg #		
Store Data Register [Preg ++] = Dreg	0x9200— 0x923F	1	0	0	1	0	0	1	0	0	0	Preg #			Dreg #		
Store Data Register [Preg – –] = Dreg	0x9280— 0x92BF	1	0	0	1	0	0	1	0	1	0	Preg #			Dreg #		
Store Data Register [Preg + uimm6m4] = Dreg	0xB000— 0xB3FF	1	0	1	1	0	0	uimm6m4 divided by 4				Preg #			Dreg #		
Store Data Register	0xE600 0000— 0xE63F 7FFF	1	1	1	0	0	1	1	0	0	0	Preg #			Dreg #		
[Preg + uimm17m4] = Dreg		uimm17m4 divided by 4															
Store Data Register	0xE600 8000— 0xE63F FFFF	1	1	1	0	0	1	1	0	0	0	Preg #			Dreg #		
[Preg – uimm17m4] = Dreg		uimm17m4 divided by 4															
Store Data Register [Preg ++ Preg] = Dreg	0x8800— 0x89FF	1	0	0	0	1	0	0	Source Dreg #			Index Preg #			Pointer Preg #		
NOTE: Pointer Preg number cannot be the same as Index Preg number. If so, this opcode represents a non-post-modify instruction version.																	
Store Data Register [FP – uimm7m4] = Dreg	0xBA00— 0xBBF7	1	0	1	1	1	0	1	uimm7m4 divided by 4					Dreg #			

Table C-10. Load / Store Instructions (Sheet 10 of 12)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Store Data Register [Ireg] = Dreg	0x9F00—0x9F1F	1	0	0	1	1	1	1	1	0	0	0	Ireg #		Dreg #		
Store Data Register [Ireg ++] = Dreg	0x9E00—0x9E1F	1	0	0	1	1	1	1	0	0	0	0	Ireg #		Dreg #		
Store Data Register [Ireg – –] = Dreg	0x9E80—0x9E9F	1	0	0	1	1	1	1	0	1	0	0	Ireg #		Dreg #		
Store Data Register [Ireg ++ Mreg] = Dreg	0x9F80—0x9FFF	1	0	0	1	1	1	1	1	1	Mreg #		Ireg #		Dreg #		
Store High Data Register Half W [Ireg] = Dreg_hi	0x9F40—0x9F5F	1	0	0	1	1	1	1	1	0	1	0	Ireg #		Dreg #		
Store High Data Register Half W [Ireg ++] = Dreg_hi	0x9E40—0x9E5F	1	0	0	1	1	1	1	0	0	1	0	Ireg #		Dreg #		
Store High Data Register Half W [Ireg – –] = Dreg_hi	9EC0—0x9EDF	1	0	0	1	1	1	1	0	1	1	0	Ireg #		Dreg #		
Store High Data Register Half W [Preg] = Dreg_hi	0x8C00—0x8DFF	1	0	0	0	1	1	0	Source Dreg #			Pointer Preg #			Pointer Preg #		
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 W [Preg ++ Preg] = Dreg_hi	0x8C01—0x8DFE	1	0	0	0	1	1	0	Source Dreg #			Index Preg #			Pointer Preg #		
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)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Store Low Data Register Half W [Ireg] = Dreg_lo	0x9F20— 0x9F3F	1	0	0	1	1	1	1	1	0	0	1	Ireg #		Dreg #		
Store Low Data Register Half W [Ireg ++] = Dreg_lo	0x9E20— 0x9E3F	1	0	0	1	1	1	1	0	0	0	1	Ireg #		Dreg #		
Store Low Data Register Half W [Ireg – –] = Dreg_lo	0x9EA0— 0x9EBF	1	0	0	1	1	1	1	0	1	0	1	Ireg #		Dreg #		
Store Low Data Register Half W [Preg] = Dreg_lo	0x8A00— 0x8BFF	1	0	0	0	1	0	1	Source Dreg #			Pointer Preg #			Pointer Preg #		
NOTE: Both Pointer Preg # fields must refer to the same Preg number. Otherwise, this opcode represents a post-modify version of this instruction.																	
Store Low Data Register Half W [Preg] = Dreg	0x9700— 0x973F	1	0	0	1	0	1	1	1	0	0	Preg #			Dreg #		
Store Low Data Register Half W [Preg ++] = Dreg	0x9600— 0x963F	1	0	0	1	0	1	1	0	0	0	Preg #			Dreg #		
Store Low Data Register Half W [Preg – –] = Dreg	0x9680— 0x96BF	1	0	0	1	0	1	1	0	1	0	Preg #			Dreg #		
Store Low Data Register Half W [Preg + uimm5m2] = Dreg	0xB400— 0xB7FF	1	0	1	1	0	1	uimm5m2 divided by 2				Preg #			Dreg #		
Store Low Data Register Half W [Preg + uimm16m2] = Dreg	0xE640 0000— 0xE67F 7FFF	1	1	1	0	0	1	1	0	0	1	Preg #			Dreg #		
		uimm16m2 divided by 2															

Table C-10. Load / Store Instructions (Sheet 12 of 12)

Instruction		Bin															
and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Store Low Data Register Half	0xE640 8000— 0xE67F FFFF	1	1	1	0	0	1	1	0	0	1	Preg #			Dreg #		
		uimm16m2 divided by 2															
W [Preg – uimm16m2] = Dreg																	
Store Low Data Register Half	0x8A01— 0x8BFE	1	0	0	0	1	0	1	Source Dreg #			Index Preg #			Pointer Preg #		
W [Preg ++ Preg] = Dreg_lo																	
NOTE: Pointer Preg number cannot be the same as Index Preg number. If so, this opcode represents a non-post-modify instruction version.																	
Store Byte	0x9B00— 0x9B3F	1	0	0	1	1	0	1	1	0	0	Preg #			Dreg #		
B [Preg] = Dreg																	
Store Byte	0x9A00— 0x9A3F	1	0	0	1	1	0	1	0	0	0	Preg #			Dreg #		
B [Preg ++] = Dreg																	
Store Byte	0x9A80— 0x9ABF	1	0	0	1	1	0	1	0	1	0	Preg #			Dreg #		
B [Preg – –] = Dreg																	
Store Byte	0xE680 0000— 0xE6BF 7FFF	1	1	1	0	0	1	1	0	1	0	Preg #			Dreg #		
		uimm15															
B [Preg + uimm15] = Dreg																	
Store Byte	0xE680 8000— 0xE6BF FFFF	1	1	1	0	0	1	1	0	1	0	Preg #			Dreg #		
		uimm15															
B [Preg – uimm15] = Dreg																	

Move Instructions

Table C-11. Move Instructions (Sheet 1 of 9)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Move Register	0x3000— 0x3FFF	0	0	1	1	Dest. reg. group			Source reg. group			Dest. reg. #			Source reg. #		
genreg = genreg genreg = dagreg dagreg = genreg dagreg = dagreg genreg = USP USP = genreg Dreg = sysreg sysreg = Dreg sysreg = Preg sysreg = USP																	
Move Register	0xC408 C000— 0xC408 C038	1	1	0	0	0	1	0	x	x	x	0	0	1	0	0	0
		1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1
A0 = A1																	
Move Register	0xC408 E000— 0xC408 E000	1	1	0	0	0	1	0	x	x	x	0	0	1	0	0	0
		1	1	1	0	0	0	0	0	0	0	1	1	1	1	1	1
A1 = A0																	
Move Register	0xC409 2000— 0xC409 2038	1	1	0	0	0	1	0	x	x	x	0	0	1	0	0	1
		0	0	1	0	0	0	0	0	0	0	Source Dreg #			0	0	0
A0 = Dreg																	
Move Register	0xC409 A000— 0xC409 A038	1	1	0	0	0	1	0	x	x	x	0	0	1	0	0	1
		1	0	1	0	0	0	0	0	0	0	Source Dreg #			0	0	0
A1 = Dreg																	

Table C-11. Move Instructions (Sheet 2 of 9)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Move Register	0xC00B 3800— 0xC00B 39C0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1
		0	0	1	1	1	0	0	Dreg_even #			0	0	0	0	0	0
Dreg_even = A0																	
Move Register	0xC08B 3800— 0xC08B 39C0	1	1	0	0	0	0	0	0	1	0	0	0	1	0	1	1
		0	0	1	1	1	0	0	Dreg_even #			0	0	0	0	0	0
Dreg_even = A0 (FU)																	
Move Register	0xC12B 3800— 0xC12B 39C0	1	1	0	0	0	0	0	1	0	0	1	0	1	0	1	1
		0	0	1	1	1	0	0	Dreg_even #			0	0	0	0	0	0
Dreg_even = A0 (ISS2)																	
Move Register	0xC00F 1800— 0xC00F 19C0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1
		0	0	0	1	1	0	0	Dreg_even # of the pair containing Dreg_odd			0	0	0	0	0	0
Dreg_odd = A1																	
Move Register	0xC08F 1800— 0xC08F 19C0	1	1	0	0	0	0	0	0	1	0	0	0	1	1	1	1
		0	0	0	1	1	0	0	Dreg_even # of the pair containing Dreg_odd			0	0	0	0	0	0
Dreg_odd = A1 (FU)																	

Table C-11. Move Instructions (Sheet 3 of 9)

Instruction and Version	Opcode Range	Bin 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Move Register	0xC12F 1800— 0xC12F 19C0	1	1	0	0	0	0	0	1	0	0	1	0	1	1	1	1
		0	0	0	1	1	0	0	Dreg_even # of the pair containing Dreg_odd			0	0	0	0	0	0
Dreg_odd = A1 (ISS2)																	
Move Register	0xC00F 3800— 0xC00F 39C0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1
		0	0	1	1	1	0	0	Dreg_even # of the register pair			0	0	0	0	0	0
Dreg_even = A0, Dreg_odd = A1 Dreg_odd = A1, Dreg_even =A0																	
Move Register	0xC08F 3800— 0xC08F 39C0	1	1	0	0	0	0	0	0	1	0	0	0	1	1	1	1
		0	0	1	1	1	0	0	Dreg_even # of the register pair			0	0	0	0	0	0
Dreg_even = A0, Dreg_odd = A1 (FU) Dreg_odd = A1, Dreg_even =A0 (FU)																	
Move Register	0xC12F 3800— 0xC12F 39C0	1	1	0	0	0	0	0	1	0	0	1	0	1	1	1	1
		0	0	1	1	1	0	0	Dreg_even # of the register pair			0	0	0	0	0	0
Dreg_even = A0, Dreg_odd = A1 (ISS2) Dreg_odd = A1, Dreg_even =A0 (ISS2)																	

Table C-11. Move Instructions (Sheet 4 of 9)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Move Conditional IF CC Dreg=Dreg	0x0700— 0x073F	0	0	0	0	0	1	1	1	0	0	Dest. Dreg #			Source Dreg #		
Move Conditional IF CC Dreg=Preg	0x0740— 0x077F	0	0	0	0	0	1	1	1	0	1	Dest. Dreg #			Source Preg #		
Move Conditional IF CC Preg=Dreg	0x0780— 0x07BF	0	0	0	0	0	1	1	1	1	0	Dest. Preg #			Source Dreg #		
Move Conditional IF CC Preg=Preg	0x07C0— 0x07FF	0	0	0	0	0	1	1	1	1	1	Dest. Preg #			Source Preg #		
Move Conditional IF !CC Dreg=Dreg	0x0600— 0x063F	0	0	0	0	0	1	1	0	0	0	Dest. Dreg #			Source Dreg #		
Move Conditional IF !CC Dreg=Preg	0x0640— 0x067F	0	0	0	0	0	1	1	0	0	1	Dest. Dreg #			Source Preg #		
Move Conditional IF !CC Preg=Dreg	0x0680— 0x06BF	0	0	0	0	0	1	1	0	1	0	Dest. Preg #			Source Dreg #		
Move Conditional IF !CC Preg=Preg	0x06C0— 0x06FF	0	0	0	0	0	1	1	0	1	1	Dest. Preg #			Source Preg #		
Move Half to Full Word, Zero Extended Dreg = Dreg_lo (Z)	0x42C0— 0x42FF	0	1	0	0	0	0	1	0	1	1	Source Dreg #			Dest. Dreg #		
Move Half to Full Word, Sign Extended Dreg = Dreg_lo (X)	0x4280— 0x42BF	0	1	0	0	0	0	1	0	1	0	Source Dreg #			Dest. Dreg #		

Table C-11. Move Instructions (Sheet 5 of 9)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Move Register Half	0xC409 4000— 0xC409 4038	1	1	0	0	0	1	0	x	x	x	0	0	1	0	0	1
		0	1	0	0	0	0	0	0	0	0	Source Dreg #			0	0	0
A0.X = Dreg_lo																	
Move Register Half	0xC409 C000— 0xC409 C038	1	1	0	0	0	1	0	x	x	x	0	0	1	0	0	1
		1	1	0	0	0	0	0	0	0	0	Source Dreg #			0	0	0
A1.X = Dreg_lo																	
Move Register Half	0xC40A 0000— 0xC40A 0E00	1	1	0	0	0	1	0	x	x	x	0	0	1	0	1	0
		0	0	0	0	Dest Dreg #			0	0	0	1	1	1	1	1	1
Dreg_lo = A0.X																	
Move Register Half	0xC40A 4000— 0xC40A 4E00	1	1	0	0	0	1	0	x	x	x	0	0	1	0	1	0
		0	1	0	0	Dest Dreg #			0	0	0	1	1	1	1	1	1
Dreg_lo = A1.X																	
Move Register Half	0xC409 0000— 0xC409 0038	1	1	0	0	0	1	0	x	x	x	0	0	1	0	0	1
		0	0	0	0	0	0	0	0	0	0	Source 0 Dreg #			0	0	0
A0.L = Dreg_lo																	
Move Register Half	0xC409 8000— 0xC409 8038	1	1	0	0	0	1	0	x	x	x	0	0	1	0	0	1
		1	0	0	0	0	0	0	0	0	0	Source 0 Dreg #			0	0	0
A1.L = Dreg_lo																	

Table C-11. Move Instructions (Sheet 6 of 9)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Move Register Half	0xC429 0000— 0xC429 0038	1	1	0	0	0	1	0	x	x	x	1	0	1	0	0	1
		0	0	0	0	0	0	0	0	0	0	Source 0 Dreg #			0	0	0
A0.H = Dreg_hi																	
Move Register Half	0xC429 8000— 0xC429 8038	1	1	0	0	0	1	0	x	x	x	1	0	1	0	0	1
		1	0	0	0	0	0	0	0	0	0	Source 0 Dreg #			0	0	0
A1.H = Dreg_hi																	
Move Register Half	0xC003 3800— 0xC003 39C0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1
		0	0	1	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_lo = A0																	
Move Register Half	0xC083 3800— 0xC083 39C0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	1	1
		0	0	1	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_lo = A0 (FU)																	
Move Register Half	0xC103 3800— 0xC103 39C0	1	1	0	0	0	0	0	1	0	0	0	0	0	0	1	1
		0	0	1	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_lo = A0 (IS)																	
Move Register Half	0xC183 3800— 0xC183 39C0	1	1	0	0	0	0	0	1	1	0	0	0	0	0	1	1
		0	0	1	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_lo = A0 (IU)																	
Move Register Half	0xC043 3800— 0xC043 39C0	1	1	0	0	0	0	0	0	0	1	0	0	0	0	1	1
		0	0	1	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_lo = A0 (T)																	

Table C-11. Move Instructions (Sheet 7 of 9)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Move Register Half	0xC023 3800— 0xC023 39C0	1	1	0	0	0	0	0	0	0	0	1	0	0	0	1	1
		0	0	1	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_lo = A0 (S2RND)																	
Move Register Half	0xC123 3800— 0xC123 39C0	1	1	0	0	0	0	0	1	0	0	1	0	0	0	1	1
		0	0	1	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_lo = A0 (ISS2)																	
Move Register Half	0xC163 3800— 0xC163 39C0	1	1	0	0	0	0	0	1	0	1	1	0	0	0	1	1
		0	0	1	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_lo = A0 (IH)																	
Move Register Half	0xC007 1800— 0xC007 19C0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1
		0	0	0	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_hi = A1																	
Move Register Half	0xC107 1800— 0xC107 19C0	1	1	0	0	0	0	0	1	0	0	0	0	0	1	1	1
		0	0	0	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_hi = A1 (IS)																	
Move Register Half	0xC087 1800— 0xC087 19C0	1	1	0	0	0	0	0	0	1	0	0	0	0	1	1	1
		0	0	0	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_hi = A1 (FU)																	
Move Register Half	0xC187 1800— 0xC187 19C0	1	1	0	0	0	0	0	1	1	0	0	0	0	1	1	1
		0	0	0	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_hi = A1 (IU)																	

Table C-11. Move Instructions (Sheet 8 of 9)

Instruction		Bin															
and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Move Register Half	0xC047 1800— 0xC047 19C0	1	1	0	0	0	0	0	0	0	1	0	0	0	1	1	1
		0	0	0	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_hi = A1 (T)																	
Move Register Half	0xC027 1800— 0xC027 19C0	1	1	0	0	0	0	0	0	0	0	1	0	0	1	1	1
		0	0	0	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_hi = A1 (S2RND)																	
Move Register Half	0xC127 1800— 0xC127 19C0	1	1	0	0	0	0	0	1	0	0	1	0	0	1	1	1
		0	0	0	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_hi = A1 (ISS2)																	
Move Register Half	0xC167 1800— 0xC167 19C0	1	1	0	0	0	0	0	1	0	1	1	0	0	1	1	1
		0	0	0	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_hi = A1 (IH)																	
Move Register Half	0xC007 3800— 0xC007 39C0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1
		0	0	1	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_lo = A0, Dreg_hi = A1 Dreg_hi = A1, Dreg_lo = A0																	
Move Register Half	0xC087 3800— 0xC087 39C0	1	1	0	0	0	0	0	0	1	0	0	0	0	1	1	1
		0	0	1	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_lo = A0, Dreg_hi = A1 (FU) Dreg_hi = A1, Dreg_lo = A0 (FU)																	
Move Register Half	0xC107 3800— 0xC107 39C0	1	1	0	0	0	0	0	1	0	0	0	0	0	1	1	1
		0	0	1	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_lo = A0, Dreg_hi = A1 (IS) Dreg_hi = A1, Dreg_lo = A0 (IS)																	

Table C-11. Move Instructions (Sheet 9 of 9)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Move Register Half	0xC187 3800— 0xC187 39C0	1	1	0	0	0	0	0	1	1	0	0	0	0	1	1	1
		0	0	1	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_lo = A0, Dreg_hi = A1 (IU) Dreg_hi = A1, Dreg_lo = A0 (IU)																	
Move Register Half	0xC047 3800— 0xC047 39C0	1	1	0	0	0	0	0	0	0	1	0	0	0	1	1	1
		0	0	1	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_lo = A0, Dreg_hi = A1 (T) Dreg_hi = A1, Dreg_lo = A0 (T)																	
Move Register Half	0xC027 3800— 0xC027 39C0	1	1	0	0	0	0	0	0	0	0	1	0	0	1	1	1
		0	0	1	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_lo = A0, Dreg_hi = A1 (S2RND) Dreg_hi = A1, Dreg_lo = A0 (S2RND)																	
Move Register Half	0xC127 3800— 0xC127 39C0	1	1	0	0	0	0	0	1	0	0	1	0	0	1	1	1
		0	0	1	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_lo = A0, Dreg_hi = A1 (ISS2) Dreg_hi = A1, Dreg_lo = A0 (ISS2)																	
Move Register Half	0xC167 3800— 0xC167 39C0	1	1	0	0	0	0	0	1	0	1	1	0	0	1	1	1
		0	0	1	1	1	0	0	Dreg #			0	0	0	0	0	0
Dreg_lo = A0, Dreg_hi = A1 (IH) Dreg_hi = A1, Dreg_lo = A0 (IH)																	
Move Byte, Zero Extended	0x4340— 0x437F	0	1	0	0	0	0	1	1	0	1	Source Dreg #			Dest. Dreg #		
Dreg = Dreg_byte (Z)																	
Move Byte, Sign Extended	0x4300— 0x433F	0	1	0	0	0	0	1	1	0	0	Source Dreg #			Dest. Dreg #		
Dreg = Dreg_byte (X)																	

Stack Control Instructions

Table C-12. Stack Control Instructions (Sheet 1 of 2)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Push	0x0140— 0x017F	0	0	0	0	0	0	0	1	0	1	Reg. group			Reg. #		
[– –SP]=allreg																	
Push Multiple	0x05C0— 0x05FD	0	0	0	0	0	1	0	1	1	1	Dreg #			Preg #		
NOTE: See two above notes on interpretation of the register number fields.																	
[– –SP]=(R7:Dreglim, P5:Preglim)																	
Push Multiple	0x0540— 0x0578	0	0	0	0	0	1	0	1	0	1	Dreg #			0	0	0
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.																	
[– –SP]=(R7:Dreglim)																	
Push Multiple	0x04C0— 0x04C5	0	0	0	0	0	1	0	0	1	1	0	0	0	Preg #		
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.																	
[– –SP]=(P5:Preglim)																	
Pop	0x0100— 0x013F	0	0	0	0	0	0	0	1	0	0	Reg. group			Reg. #		
NOTE: Dreg and Preg not supported by this instruction. See Load Data Register for Dreg and Load Pointer Register for Preg.																	
mostreg=[SP++]																	
Pop Multiple	0x0580— 0x05BD	0	0	0	0	0	1	0	1	1	0	Dreg #			Preg #		
NOTE: See two above notes on interpretation of the register number fields.																	
(R7:Dreglim, P5:Preglim)=[SP++]																	

Table C-12. Stack Control Instructions (Sheet 2 of 2)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Pop Multiple	0x0500— 0x0538	0	0	0	0	0	1	0	1	0	0	Dreg #			0	0	0
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++]																	
Pop Multiple	0x0480— 0x0485	0	0	0	0	0	1	0	0	1	0	0	0	0	Preg #		
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++]																	
Linkage	0xE800 0000— 0xE800 FFFF	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0
		uimm18m4 divided by 4															
LINK uimm18m4																	
Linkage	0xE801 0000	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	1
		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
UNLINK																	

Control Code Bit Management Instructions

Table C-13. Control Code Bit Management Instructions (Sheet 1 of 4)

Instruction		Bin															
and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Compare Data Register CC = Dreg == Dreg	0x0800— 0x083F	0	0	0	0	1	0	0	0	0	0	Source reg #			Dest reg #		
Compare Data Register CC = Dreg == imm3	0x0C00— 0x0C3F	0	0	0	0	1	1	0	0	0	0	imm3			Dest reg #		
Compare Data Register CC = Dreg < Dreg	0x0880— 0x08BF	0	0	0	0	1	0	0	0	1	0	Source reg #			Dest reg #		
Compare Data Register CC = Dreg < imm3	0x0C80— 0x0CBF	0	0	0	0	1	1	0	0	1	0	imm3			Dest reg #		
Compare Data Register CC = Dreg <= Dreg	0x0900— 0x093F	0	0	0	0	1	0	0	1	0	0	Source reg #			Dest reg #		
Compare Data Register CC = Dreg <= imm3	0x0D00— 0x0D3F	0	0	0	0	1	1	0	1	0	0	imm3			Dest reg #		
Compare Data Register CC = Dreg < Dreg (IU)	0x0980— 0x09BF	0	0	0	0	1	0	0	1	1	0	Source reg #			Dest reg #		
Compare Data Register CC = Dreg < uimm3 (IU)	0x0D80— 0x0DBF	0	0	0	0	1	1	0	1	1	0	uimm3			Dest reg #		
Compare Data Register CC = Dreg <= Dreg (IU)	0x0A00— 0x0A3F	0	0	0	0	1	0	1	0	0	0	Source reg #			Dest reg #		

Table C-13. Control Code Bit Management Instructions (Sheet 2 of 4)

Instruction		Bin															
and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Compare Data Register CC = Dreg <= uimm3 (IU)	0x0E00— 0x0E3F	0	0	0	0	1	1	1	0	0	0	uimm3			Dest reg #		
Compare Pointer Register CC = Preg == Preg	0x0840— 0x087F	0	0	0	0	1	0	0	0	0	1	Source reg #			Dest reg #		
Compare Pointer Register CC = Preg == imm3	0x0C40— 0x0C7F	0	0	0	0	1	1	0	0	0	1	imm3			Dest reg #		
Compare Pointer Register CC = Preg < Preg	0x08C0— 0x08FF	0	0	0	0	1	0	0	0	1	1	Source reg #			Dest reg #		
Compare Pointer Register CC = Preg < imm3	0x0CC0— 0x0CFF	0	0	0	0	1	1	0	0	1	1	imm3			Dest reg #		
Compare Pointer Register CC = Preg <= Preg	0x0940— 0x097F	0	0	0	0	1	0	0	1	0	1	Source reg #			Dest reg #		
Compare Pointer Register CC = Preg <= imm3	0x0D40— 0x0D7F	0	0	0	0	1	1	0	1	0	1	imm3			Dest reg #		
Compare Pointer Register CC = Preg < Preg (IU)	0x09C0— 0x09FF	0	0	0	0	1	0	0	1	1	1	Source reg #			Dest reg #		
Compare Pointer Register CC = Preg < uimm3 (IU)	0x0DC0— 0x0DFF	0	0	0	0	1	1	0	1	1	1	uimm3			Dest reg #		
Compare Pointer Register CC = Preg <= Preg (IU)	0x0A40— 0x0A7F	0	0	0	0	1	0	1	0	0	1	Source reg #			Dest reg #		

Table C-13. Control Code Bit Management Instructions (Sheet 3 of 4)

Instruction and Version	Opcode Range	Bin 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Compare Pointer Register CC = Preg <= uimm3 (IU)	0x0E40— 0x0E7F	0	0	0	0	1	1	1	0	0	1	uimm3			Dest reg #		
Compare Accumulator CC = A0 == A1	0x0A80	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0
Compare Accumulator CC = A0 < A1	0x0B00	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0
Compare Accumulator CC = A0 <= A1	0x0B80	0	0	0	0	1	0	1	1	1	0	0	0	0	0	0	0
Move CC Dreg = CC	0x0200— 0x0207	0	0	0	0	0	0	1	0	0	0	0	0	0	Dreg #		
Move CC statbit = CC	0x0380— 0x039F	0	0	0	0	0	0	1	1	1	0	0	ASTAT bit #				
Move CC statbit |= CC	0x03A0— 0x03BF	0	0	0	0	0	0	1	1	1	0	1	ASTAT bit #				
Move CC statbit &= CC	0x03C0— 0x03DF	0	0	0	0	0	0	1	1	1	1	0	ASTAT bit #				
Move CC statbit ^= CC	0x03E0— 0x03FF	0	0	0	0	0	0	1	1	1	1	1	ASTAT bit #				
Move CC CC = Dreg	0x0208— 0x020F	0	0	0	0	0	0	1	0	0	0	0	0	1	Dreg #		
Move CC CC = statbit	0x0300— 0x031F	0	0	0	0	0	0	1	1	0	0	0	ASTAT bit #				

Table C-13. Control Code Bit Management Instructions (Sheet 4 of 4)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Move CC CC |= statbit	0x0320— 0x033F	0	0	0	0	0	0	1	1	0	0	1	ASTAT bit #				
Move CC CC &= statbit	0x0340— 035F	0	0	0	0	0	0	1	1	0	1	0	ASTAT bit #				
Move CC CC ^= statbit	0x0360— 0x037F	0	0	0	0	0	0	1	1	0	1	1	ASTAT bit #				
Negate CC CC = !CC	0x0218	0	0	0	0	0	0	1	0	0	0	0	1	1	0	0	0

Logical Operations Instructions

Table C-14. Logical Operations Instructions

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AND	0x5400— 0x55FF	0	1	0	1	0	1	0	Dest. Dreg #			Src 1 Dreg #			Src 0 Dreg #		
Dreg = Dreg & Dreg																	
NOT (One's-Complement)	0x43C0— 0x43FF	0	1	0	0	0	0	1	1	1	1	Source Dreg #			Dest. Dreg #		
Dreg = ~ Dreg																	
OR	0x5600— 0x57FF	0	1	0	1	0	1	1	Dest. Dreg #			Src 1 Dreg #			Src 0 Dreg #		
Dreg = Dreg | Dreg																	
Exclusive OR	0x5800— 0x59FF	0	1	0	1	1	0	0	Dest. Dreg #			Src 1 Dreg #			Src 0 Dreg #		
Dreg = Dreg ^ Dreg																	
Bit Wise Exclusive OR	0xC60B 0000— 0xC60B 0E38	1	1	0	0	0	1	1	0	0	0	x	x	1	0	1	1
		0	0	0	0	Dest. Dreg #			x	x	x	Source Dreg #			0	0	0
Dreg_lo = CC = BXORSHIFT (A0, Dreg)																	
Bit Wise Exclusive OR	0xC60B 4000— 0xC60B 4E38	1	1	0	0	0	1	1	0	0	0	x	x	1	0	1	1
		0	1	0	0	Dest. Dreg #			x	x	x	Source Dreg #			0	0	0
Dreg_lo = CC = BXOR (A0, Dreg)																	
Bit Wise Exclusive OR	C60C 4000— C60C 4E00	1	1	0	0	0	1	1	0	0	x	x	0	1	1	0	0
		0	1	0	0	Dest. Dreg #			x	x	x	0	0	0	0	0	0
Dreg_lo = CC = BXOR (A0, A1, CC)																	
Bit Wise Exclusive OR	C60C 0000	1	1	0	0	0	1	1	0	0	x	x	0	1	1	0	0
		0	0	0	0	0	0	0	x	x	x	0	0	0	0	0	0
A0 = BXORSHIFT (A0, A1, CC)																	

Bit Operations Instructions

Table C-15. Bit Operations Instructions (Sheet 1 of 2)

Instruction		Bin															
and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Bit Clear	0x4C00— 0x4CFF	0	1	0	0	1	1	0	0	uimm5					Dest. Dreg #		
BITCLR (Dreg, uimm5)																	
Bit Set	0x4A00— 0x4AFF	0	1	0	0	1	0	1	0	uimm5					Dest. Dreg #		
BITSET (Dreg, uimm5)																	
Bit Toggle	0x4B00— 0x4BFF	0	1	0	0	1	0	1	1	uimm5					Dest. Dreg #		
BITTGL (Dreg, uimm5)																	
Bit Test	0x4900— 0x49FF	0	1	0	0	1	0	0	1	uimm5					Dest. Dreg #		
CC = BITTST (Dreg, uimm5)																	
Bit Test	0x4800— 0x48FF	0	1	0	0	1	0	0	0	uimm5					Dest. Dreg #		
CC = ! BITTST (Dreg, uimm5)																	
Bit Field Deposit	0xC60A 8000— 0xC60A 8E3F	1	1	0	0	0	1	1	0	0	x	x	0	1	0	1	0
		1	0	0	0	Dest. Dreg #			x	x	x	foregnd Dreg #			backgnd Dreg #		
Dreg = DEPOSIT (Dreg, Dreg)																	
Bit Field Deposit	0xC60A C000— 0xC60A CE3F	1	1	0	0	0	1	1	0	0	x	x	0	1	0	1	0
		1	1	0	0	Dest. Dreg #			x	x	x	foregnd Dreg #			backgnd Dreg #		
Dreg = DEPOSIT (Dreg, Dreg) (X)																	
Bit Field Extraction	0xC60A 0000— 0xC60A 0E3F	1	1	0	0	0	1	1	0	0	x	x	0	1	0	1	0
		0	0	0	0	Dest. Dreg #			x	x	x	pattern Dreg #			scene Dreg #		
Dreg = EXTRACT (Dreg, Dreg_lo) (Z)																	

Table C-15. Bit Operations Instructions (Sheet 2 of 2)

Instruction		Bin															
and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Bit Field Extraction	0xC60A 4000— 0xC60A 4E3F	1	1	0	0	0	1	1	0	0	x	x	0	1	0	1	0
		0	1	0	0	Dest. Dreg #			x	x	x	pattern Dreg #			scene Dreg #		
Dreg = EXTRACT (Dreg, Dreg_lo) (X)																	
Bit Multiplex	0xC608 0000— 0xC608 003F	1	1	0	0	0	1	1	0	0	x	x	0	1	0	0	0
		0	0	0	0	0	0	0	x	x	x	Source 0 Dreg #			Source 1 Dreg #		
BITMUX (Dreg, Dreg, A0) (ASR)																	
Bit Multiplex	0xC608 4000— 0xC608 403F	1	1	0	0	0	1	1	0	0	x	x	0	1	0	0	0
		0	1	0	0	0	0	0	x	x	x	Source 0 Dreg #			Source 1 Dreg #		
BITMUX (Dreg, Dreg, A0) (ASL)																	
One's-Population Count	0xC606 C000— 0xC606 CE07	1	1	0	0	0	1	1	0	0	x	x	0	0	1	1	0
		1	1	0	0	Dest. Dreg #			x	x	x	0	0	0	Source Dreg #		
Dreg_lo = ONES Dreg																	

Shift / Rotate Operations Instructions

Table C-16. Shift / Rotate Operations Instructions (Sheet 1 of 9)

Instruction and Version	Opcode Range	Bin															
		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	Source Preg #			Dest. Preg #		
Preg = (Preg + Preg) << 1																	
Add with Shift	0x45C0— 0x45FF	0	1	0	0	0	1	0	1	1	1	Source Preg #			Dest. Preg #		
Preg = (Preg + Preg) << 2																	
Add with Shift	0x4100— 0x413F	0	1	0	0	0	0	0	1	0	0	Source Dreg #			Dest. Dreg #		
Dreg = (Dreg + Dreg) << 1																	
Add with Shift	0x4140— 0x417F	0	1	0	0	0	0	0	1	0	1	Source Dreg #			Dest. Dreg #		
Dreg = (Dreg + Dreg) << 2																	
Shift with Add	0x5C00— 0x5DFF	0	1	0	1	1	1	0	Dest. Preg #			Src 1 Preg #			Src 0 Preg #		
Preg = Preg + (Preg <<1)																	
Shift with Add	0x5E00— 0x5FFF	0	1	0	1	1	1	1	Dest. Preg #			Src 1 Preg #			Src 0 Preg #		
Preg = Preg + (Preg <<2)																	
Arithmetic Shift	0x4D00— 0x4DFF	0	1	0	0	1	1	0	uimm5						Dest. Dreg #		
Dreg >>>= uimm5																	
Arithmetic Shift	0xC680 0180— 0xC680 0FFF	1	1	0	0	0	1	1	0	1	x	x	0	0	0	0	0
		0	0	0	0	Dest. Dreg #			2's comp. of uimm4						Source Dreg #		
Dreg_lo = Dreg_lo >>> uimm4																	
Arithmetic Shift	0xC680 1180— 0xC680 1FFF	1	1	0	0	0	1	1	0	1	x	x	0	0	0	0	0
		0	0	0	1	Dest. Dreg #			2's comp. of uimm4						Source Dreg #		
Dreg_lo = Dreg_hi >>> uimm4																	

Table C-16. Shift / Rotate Operations Instructions (Sheet 2 of 9)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Arithmetic Shift	0xC680 2180—	1	1	0	0	0	1	1	0	1	x	x	0	0	0	0	0
	0xC680 2FFF	0	0	1	0	Dest. Dreg #			2’s comp. of uimm4						Source Dreg #		
Dreg_hi = Dreg_lo >>> uimm4																	
Arithmetic Shift	0xC680 3180—	1	1	0	0	0	1	1	0	1	x	x	0	0	0	0	0
	0xC680 3FFF	0	0	1	1	Dest. Dreg #			2’s comp. of uimm4						Source Dreg #		
Dreg_hi = Dreg_hi >>> uimm4																	
Arithmetic Shift	0xC680 4000—	1	1	0	0	0	1	1	0	1	x	x	0	0	0	0	0
	0xC680 4E7F	0	1	0	0	Dest. Dreg #			uimm4						Source Dreg #		
Dreg_lo = Dreg_lo << uimm4 (S)																	
Arithmetic Shift	0xC680 5000—	1	1	0	0	0	1	1	0	1	x	x	0	0	0	0	0
	0xC680 5E7F	0	1	0	1	Dest. Dreg #			uimm4						Source Dreg #		
Dreg_lo = Dreg_hi << uimm4 (S)																	
Arithmetic Shift	0xC680 6000—	1	1	0	0	0	1	1	0	1	x	x	0	0	0	0	0
	0xC680 6E7F	0	1	1	0	Dest. Dreg #			uimm4						Source Dreg #		
Dreg_hi = Dreg_lo << uimm4 (S)																	
Arithmetic Shift	0xC680 7000—	1	1	0	0	0	1	1	0	1	x	x	0	0	0	0	0
	0xC680 7E7F	0	1	1	1	Dest. Dreg #			uimm4						Source Dreg #		
Dreg_hi = Dreg_hi << uimm4 (S)																	
Arithmetic Shift	0xC682 0100—	1	1	0	0	0	1	1	0	1	x	x	0	0	0	1	0
	0xC682 0FFF	0	0	0	0	Dest. Dreg #			2’s complement of uimm5						Source Dreg #		
Dreg = Dreg >>> uimm5																	

Table C-16. Shift / Rotate Operations Instructions (Sheet 3 of 9)

Instruction		Bin															
and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Arithmetic Shift	0xC682 4000— 0xC680 4EFF	1	1	0	0	0	1	1	0	1	x	x	0	0	0	1	0
		0	1	0	0	Dest. Dreg #			uimm5						Source Dreg #		
Dreg = Dreg << uimm5 (S)																	
Arithmetic Shift	0xC683 0100— 0xC683 01F8	1	1	0	0	0	1	1	0	1	x	x	0	0	0	1	1
		0	0	0	0	0	0	0	2's complement of uimm5						0	0	0
A0 = A0 >>> uimm5																	
Arithmetic Shift	0xC683 1100— 0xC683 11F8	1	1	0	0	0	1	1	0	1	x	x	0	0	0	1	1
		0	0	0	1	0	0	0	2's complement of uimm5						0	0	0
A1 = A1 >>> uimm5																	
Arithmetic Shift	0x4000— 0x403F	0	1	0	0	0	0	0	0	0	0	Source Dreg #			Dest. Dreg #		
Dreg >>>= Dreg																	
Arithmetic Shift	0xC600 0000— 0xC600 0E3F	1	1	0	0	0	1	1	0	0	x	x	0	0	0	0	0
		0	0	0	0	Dest. Dreg #			x	x	x	Source Dreg #			sh_mag Dreg #		
Dreg_lo = ASHIFT Dreg_lo BY Dreg_lo																	
Arithmetic Shift	0xC600 1000— 0xC600 1E3F	1	1	0	0	0	1	1	0	0	x	x	0	0	0	0	0
		0	0	0	1	Dest. Dreg #			x	x	x	Source Dreg #			sh_mag Dreg #		
Dreg_lo = ASHIFT Dreg_hi BY Dreg_lo																	
Arithmetic Shift	0xC600 2000— 0xC600 2E3F	1	1	0	0	0	1	1	0	0	x	x	0	0	0	0	0
		0	0	1	0	Dest. Dreg #			x	x	x	Source Dreg #			sh_mag Dreg #		
Dreg_hi = ASHIFT Dreg_lo BY Dreg_lo																	
Arithmetic Shift	0xC600 3000— 0xC600 3E3F	1	1	0	0	0	1	1	0	0	x	x	0	0	0	0	0
		0	0	1	1	Dest. Dreg #			x	x	x	Source Dreg #			sh_mag Dreg #		
Dreg_hi = ASHIFT Dreg_hi BY Dreg_lo																	

Table C-16. Shift / Rotate Operations Instructions (Sheet 4 of 9)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Arithmetic Shift	0xC600 4000—	1	1	0	0	0	1	1	0	0	x	x	0	0	0	0	0
	0xC600 4E3F	0	1	0	0	Dest. Dreg #			x	x	x	Source Dreg #			sh_mag Dreg #		
Dreg_lo = ASHIFT Dreg_lo BY Dreg_lo (S)																	
Arithmetic Shift	0xC600 5000—	1	1	0	0	0	1	1	0	0	x	x	0	0	0	0	0
	0xC600 5E3F	0	1	0	1	Dest. Dreg #			x	x	x	Source Dreg #			sh_mag Dreg #		
Dreg_lo = ASHIFT Dreg_hi BY Dreg_lo (S)																	
Arithmetic Shift	0xC600 6000—	1	1	0	0	0	1	1	0	0	x	x	0	0	0	0	0
	0xC600 6E3F	0	1	1	0	Dest. Dreg #			x	x	x	Source Dreg #			sh_mag Dreg #		
Dreg_hi = ASHIFT Dreg_lo BY Dreg_lo (S)																	
Arithmetic Shift	0xC600 7000—	1	1	0	0	0	1	1	0	0	x	x	0	0	0	0	0
	0xC600 7E3F	0	1	1	1	Dest. Dreg #			x	x	x	Source Dreg #			sh_mag Dreg #		
Dreg_hi = ASHIFT Dreg_hi BY Dreg_lo (S)																	
Arithmetic Shift	0xC602 0000—	1	1	0	0	0	1	1	0	0	x	x	0	0	0	1	0
	0xC602 0E3F	0	0	0	0	Dest. Dreg #			x	x	x	Source Dreg #			sh_mag Dreg #		
Dreg = ASHIFT Dreg BY Dreg_lo																	
Arithmetic Shift	0xC602 4000—	1	1	0	0	0	1	1	0	0	x	x	0	0	0	1	0
	0xC602 4E3F	0	1	0	0	Dest. Dreg #			x	x	x	Source Dreg #			sh_mag Dreg #		
Dreg = ASHIFT Dreg BY Dreg_lo (S)																	
Arithmetic Shift	0xC603 0000—	1	1	0	0	0	1	1	0	0	x	x	0	0	0	1	1
	0xC603 0038	0	0	0	0	0	0	0	x	x	x	Source Dreg #			0	0	0
A0 = ASHIFT A0 BY Dreg_lo																	

Table C-16. Shift / Rotate Operations Instructions (Sheet 5 of 9)

Instruction and Version	Opcode Range	Bin 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Arithmetic Shift	0xC603 1000—	1	1	0	0	0	1	1	0	0	x	x	0	0	0	1	1
	0xC603 1038	0	0	0	1	0	0	0	x	x	x	Source Dreg #			0	0	0
A1 = ASHIFT A1 BY Dreg_lo																	
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	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	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	1	uimm5						Dest. Dreg #		
Dreg <<= uimm5																	
Logical Shift	0xC680 8180—	1	1	0	0	0	1	1	0	1	0	0	0	0	0	0	0
	0xC680 8FFF	1	0	0	0	Dest. Dreg #			2's comp. of uimm4						Source Dreg #		
Dreg_lo = Dreg_lo >> uimm4																	

Table C-16. Shift / Rotate Operations Instructions (Sheet 6 of 9)

Instruction									Bin								
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Logical Shift	0xC680 9180—	1	1	0	0	0	1	1	0	1	x	x	0	0	0	0	0
	0xC680 9FFF	1	0	0	1	Dest. Dreg #			2’s comp. of uimm4						Source Dreg #		
Dreg_lo = Dreg_hi >> uimm4																	
Logical Shift	0xC680 A180—	1	1	0	0	0	1	1	0	1	x	x	0	0	0	0	0
	0xC680 AFFF	1	0	1	0	Dest. Dreg #			2’s comp. of uimm4						Source Dreg #		
Dreg_hi = Dreg_lo >> uimm4																	
Logical Shift	0xC680 B180—	1	1	0	0	0	1	1	0	1	x	x	0	0	0	0	0
	0xC680 BFFF	1	0	1	1	Dest. Dreg #			2’s comp. of uimm4						Source Dreg #		
Dreg_hi = Dreg_hi >> uimm4																	
Logical Shift	0xC680 8000—	1	1	0	0	0	1	1	0	1	x	x	0	0	0	0	0
	0xC680 8E7F	1	0	0	0	Dest. Dreg #			uimm4						Source Dreg #		
Dreg_lo = Dreg_lo << uimm4																	
Logical Shift	0xC680 9000—	1	1	0	0	0	1	1	0	1	x	x	0	0	0	0	0
	0xC680 9E7F	1	0	0	1	Dest. Dreg #			uimm4						Source Dreg #		
Dreg_lo = Dreg_hi << uimm4																	
Logical Shift	0xC680 A000—	1	1	0	0	0	1	1	0	1	x	x	0	0	0	0	0
	0xC680 AE7F	1	0	1	0	Dest. Dreg #			uimm4						Source Dreg #		
Dreg_hi = Dreg_lo << uimm4																	
Logical Shift	0xC680 B000—	1	1	0	0	0	1	1	0	1	x	x	0	0	0	0	0
	0xC680 BE7F	1	0	1	1	Dest. Dreg #			uimm4						Source Dreg #		
Dreg_hi = Dreg_hi << uimm4																	

Table C-16. Shift / Rotate Operations Instructions (Sheet 7 of 9)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Logical Shift	0xC682 8100—	1	1	0	0	0	1	1	0	1	x	x	0	0	0	1	0
	0xC682 8FFF	1	0	0	0	Dest. Dreg #			2's comp. of uimm5						Source Dreg #		
Dreg = Dreg >> uimm5																	
Logical Shift	0xC682 8000—	1	1	0	0	0	1	1	0	1	x	x	0	0	0	1	0
	0xC682 8EFF	1	0	0	0	Dest. Dreg #			uimm5						Source Dreg #		
Dreg = Dreg << uimm5																	
Logical Shift	0xC683 4100—	1	1	0	0	0	1	1	0	1	x	x	0	0	0	1	1
	0xC683 41F8	0	1	0	0	0	0	0	2's comp of uimm5						0	0	0
A0 = A0 >> uimm5																	
Logical Shift	0xC683 4000—	1	1	0	0	0	1	1	0	1	x	x	0	0	0	1	1
	0xC683 40F8	0	1	0	0	0	0	0	uimm5						0	0	0
A0 = A0 << uimm5																	
Logical Shift	0xC683 5100—	1	1	0	0	0	1	1	0	1	x	x	0	0	0	1	1
	0xC683 51F8	0	1	0	1	0	0	0	2's comp of uimm5						0	0	0
A1 = A1 >> uimm5																	
Logical Shift	0xC683 5000—	1	1	0	0	0	1	1	0	1	x	x	0	0	0	1	1
	0xC683 50F8	0	1	0	1	0	0	0	uimm5						0	0	0
A1 = A1 << uimm5																	
Logical Shift	0x4080— 0x40BF	0	1	0	0	0	0	0	0	1	0	Source Dreg #			Dest. Dreg #		
Dreg <<= Dreg																	
Logical Shift	0x4040— 0x407F	0	1	0	0	0	0	0	0	0	1	Source Dreg #			Dest. Dreg #		
Dreg >>= Dreg																	

Table C-16. Shift / Rotate Operations Instructions (Sheet 8 of 9)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Logical Shift	0xC600 8000— 0xC600 8E3F	1	1	0	0	0	1	1	0	0	x	x	0	0	0	0	0
		1	0	0	0	Dest. Dreg #			x	x	x	Source Dreg #			sh_mag Dreg #		
Dreg_lo = LSHIFT Dreg_lo BY Dreg_lo																	
Logical Shift	0xC600 9000— 0xC600 9E3F	1	1	0	0	0	1	1	0	0	x	x	0	0	0	0	0
		1	0	0	1	Dest. Dreg #			x	x	x	Source Dreg #			sh_mag Dreg #		
Dreg_lo = LSHIFT Dreg_hi BY Dreg_lo																	
Logical Shift	0xC600 A000— 0xC600 AE3F	1	1	0	0	0	1	1	0	0	x	x	0	0	0	0	0
		1	0	1	0	Dest. Dreg #			x	x	x	Source Dreg #			sh_mag Dreg #		
Dreg_hi = LSHIFT Dreg_lo BY Dreg_lo																	
Logical Shift	0xC600 B000— 0xC600 BE3F	1	1	0	0	0	1	1	0	0	x	x	0	0	0	0	0
		1	0	1	1	Dest. Dreg #			x	x	x	Source Dreg #			sh_mag Dreg #		
Dreg_hi = LSHIFT Dreg_hi BY Dreg_lo																	
Logical Shift	0xC602 8000— 0xC602 8E3F	1	1	0	0	0	1	1	0	0	x	x	0	0	0	1	0
		1	0	0	0	Dest. Dreg #			x	x	x	Source Dreg #			sh_mag Dreg #		
Dreg = LSHIFT Dreg BY Dreg_lo																	
Logical Shift	0xC603 4000— 0xC603 4038	1	1	0	0	0	1	1	0	0	x	x	0	0	0	1	1
		0	1	0	0	0	0	0	x	x	x	Source Dreg #			0	0	0
A0 = LSHIFT A0 BY Dreg_lo																	
Logical Shift	0xC603 5000— 0xC603 5038	1	1	0	0	0	1	1	0	0	x	x	0	0	0	1	1
		0	1	0	1	0	0	0	x	x	x	Source Dreg #			0	0	0
A1 = LSHIFT A1 BY Dreg_lo																	

Table C-16. Shift / Rotate Operations Instructions (Sheet 9 of 9)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rotate	0xC682 C000— 0xC682 CFFF	1	1	0	0	0	1	1	0	1	x	x	0	0	0	1	0
		1	1	0	0	Dest. Dreg #			imm6						Source Dreg #		
Dreg = ROT Dreg BY imm6																	
Rotate	0xC683 8000— 0xC683 81F8	1	1	0	0	0	1	1	0	1	x	x	0	0	0	1	1
		1	0	0	0	0	0	0	imm6						0	0	0
A0 = ROT A0 BY imm6																	
Rotate	0xC683 9000— 0xC683 91F8	1	1	0	0	0	1	1	0	1	x	x	0	0	0	1	1
		1	0	0	1	0	0	0	imm6						0	0	0
A1 = ROT A1 BY imm6																	
Rotate	0xC602 C000— 0xC602 CE3F	1	1	0	0	0	1	1	0	0	x	x	0	0	0	1	0
		1	1	0	0	Dest. Dreg #			x	x	x	Source Dreg #			rot_mag Dreg #		
Dreg = ROT Dreg BY Dreg_lo																	
Rotate	0xC603 8000— 0xC603 8038	1	1	0	0	0	1	1	0	0	x	x	0	0	0	1	1
		1	0	0	0	0	0	0	x	x	x	Source Dreg #			0	0	0
A0 = ROT A0 BY Dreg_lo																	
Rotate	0xC603 9000— 0xC603 9038	1	1	0	0	0	1	1	0	0	x	x	0	0	0	1	1
		1	0	0	1	0	0	0	x	x	x	Source Dreg #			0	0	0
A1 = ROT A1 BY Dreg_lo																	

Arithmetic Operations Instructions

Table C-17. Arithmetic Operations Instructions (Sheet 1 of 44)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Absolute Value	0xC410 403F	1	1	0	0	0	1	0	x	x	x	0	1	0	0	0	0
		0	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1
A0 = ABS A1																	
Absolute Value	0xC430 003F	1	1	0	0	0	1	0	x	x	x	1	1	0	0	0	0
		0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
A1 = ABS A0																	
Absolute Value	0xC430 403F	1	1	0	0	0	1	0	x	x	x	1	1	0	0	0	0
		0	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1
A1 = ABS A1																	
Absolute Value	0xC410 C03F	1	1	0	0	0	1	0	x	x	x	0	1	0	0	0	0
		1	1	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	1	0	x	x	x	0	0	0	1	1	1
		1	0	0	0	Dest Dreg #			0	0	0	Source Dreg #			0	0	0
Dreg = ABS Dreg																	
Add	0x5A00— 0x5BFF	0	1	0	1	1	0	1	Dest. Dreg #			Src 1 Dreg #			Src 0 Dreg #		
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	0	1	0	0	0	Dest. Dreg #			Src 1 Dreg #			Src 0 Dreg #		
Dreg = Dreg + Dreg																	

Table C-17. Arithmetic Operations Instructions (Sheet 2 of 44)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Add	0xC404 0000—	1	1	0	0	0	1	0	x	x	x	0	0	0	1	0	0
	0xC404 0E3F	0	0	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg + Dreg (NS)																	
Add	0xC404 2000—	1	1	0	0	0	1	0	x	x	x	0	0	0	1	0	0
	0xC404 2E3F	0	0	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg + Dreg (S)																	
Add	0xC402 0000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	1	0
	0xC402 0E3F	0	0	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = Dreg_lo + Dreg_lo (NS)																	
Add	0xC402 4000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	1	0
	0xC402 4E3F	0	1	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = Dreg_lo + Dreg_hi (NS)																	
Add	0xC402 8000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	1	0
	0xC402 8E3F	1	0	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = Dreg_hi + Dreg_lo (NS)																	
Add	0xC402 C000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	1	0
	0xC402 CE3F	1	1	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = Dreg_hi + Dreg_hi (NS)																	
Add	0xC422 0000—	1	1	0	0	0	1	0	x	x	x	1	0	0	0	1	0
	0xC422 0E3F	0	0	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_hi = Dreg_lo + Dreg_lo (NS)																	

Table C-17. Arithmetic Operations Instructions (Sheet 3 of 44)

Instruction		Bin															
and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Add	0xC422 4000—	1	1	0	0	0	1	0	x	x	x	1	0	0	0	1	0
	0xC422 4E3F	0	1	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_hi = Dreg_lo + Dreg_hi (NS)																	
Add	0xC422 8000—	1	1	0	0	0	1	0	x	x	x	1	0	0	0	1	0
	0xC422 8E3F	1	0	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_hi = Dreg_hi + Dreg_lo (NS)																	
Add	0xC422 C000—	1	1	0	0	0	1	0	x	x	x	1	0	0	0	1	0
	0xC422 CE3F	1	1	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_hi = Dreg_hi + Dreg_hi (NS)																	
Add	0xC402 2000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	1	0
	0xC402 2E3F	0	0	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = Dreg_lo + Dreg_lo (S)																	
Add	0xC402 6000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	1	0
	0xC402 6E3F	0	1	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = Dreg_lo + Dreg_hi (S)																	
Add	0xC402 A000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	1	0
	0xC402 AE3F	1	0	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = Dreg_hi + Dreg_lo (S)																	
Add	0xC402 E000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	1	0
	0xC402 EE3F	1	1	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = Dreg_hi + Dreg_hi (S)																	

Table C-17. Arithmetic Operations Instructions (Sheet 4 of 44)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Add	0xC422 2000— 0xC422 2E3F	1	1	0	0	0	1	0	x	x	x	1	0	0	0	1	0
		0	0	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_hi = Dreg_lo + Dreg_lo (S)																	
Add	0xC422 6000— 0xC422 6E3F	1	1	0	0	0	1	0	x	x	x	1	0	0	0	1	0
		0	1	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_hi = Dreg_lo + Dreg_hi (S)																	
Add	0xC422 A000— 0xC422 AE3F	1	1	0	0	0	1	0	x	x	x	1	0	0	0	1	0
		1	0	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_hi = Dreg_hi + Dreg_lo (S)																	
Add	0xC422 E000— 0xC422 EE3F	1	1	0	0	0	1	0	x	x	x	1	0	0	0	1	0
		1	1	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_hi = Dreg_hi + Dreg_hi (S)																	
Add/Subtract, Prescale Down	0xC405 9000— 0xC405 9E3F	1	1	0	0	0	1	0	x	x	x	0	0	0	1	0	1
		1	0	0	1	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = Dreg + Dreg (RND20)																	
Add/Subtract, Prescale Down	0xC425 9000— 0xC425 9E3F	1	1	0	0	0	1	0	x	x	x	1	0	0	1	0	1
		1	0	0	1	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_hi = Dreg + Dreg (RND20)																	
Add/Subtract, Prescale Down	0xC405 D000— 0xC405 DE3F	1	1	0	0	0	1	0	x	x	x	0	0	0	1	0	1
		1	1	0	1	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = Dreg – Dreg (RND20)																	

Table C-17. Arithmetic Operations Instructions (Sheet 5 of 44)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Add/Subtract, Prescale Down	0xC425 D000—	1	1	0	0	0	1	0	x	x	x	1	0	0	1	0	1
	0xC425 DE3F	1	1	0	1	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_hi = Dreg – Dreg (RND20)																	
Add/Subtract, Prescale Up	0xC405 0000—	1	1	0	0	0	1	0	x	x	x	0	0	0	1	0	1
	0xC405 0E3F	0	0	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = Dreg + Dreg (RND12)																	
Add/Subtract, Prescale Up	0xC425 0000—	1	1	0	0	0	1	0	x	x	x	1	0	0	1	0	1
	0xC425 0E3F	0	0	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_hi = Dreg + Dreg (RND12)																	
Add/Subtract, Prescale Up	0xC405 4000—	1	1	0	0	0	1	0	x	x	x	0	0	0	1	0	1
	0xC405 4E3F	0	1	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = Dreg – Dreg (RND12)																	
Add/Subtract, Prescale Up	0xC425 4000—	1	1	0	0	0	1	0	x	x	x	1	0	0	1	0	1
	0xC425 4E3F	0	1	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_hi = Dreg – Dreg (RND12)																	
Add Immediate	0x6400— 0x6700	0	1	1	0	0	1	imm7							Dreg #		
Dreg += imm7																	
Add Immediate	0x6C00— 0x6FFF	0	1	1	0	1	1	imm7							Preg #		
Preg += imm7																	
Add Immediate	0x9F60— 0x9F63	1	0	0	1	1	1	1	1	0	1	1	0	0	0	Ireg #	
Ireg += 2																	

Table C-17. Arithmetic Operations Instructions (Sheet 6 of 44)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Add Immediate	0x9F68— 0x9F6B	1	0	0	1	1	1	1	1	0	1	1	0	1	0	Ireg #	
Ireg += 4																	
Divide Primitive	0x4240— 0x427F	0	1	0	0	0	0	1	0	0	1	Source Dreg #			Dest. Dreg #		
DIVS (Dreg, Dreg)																	
Divide Primitive	0x4200— 0x423F	0	1	0	0	0	0	1	0	0	0	Source Dreg #			Dest. Dreg #		
DIVQ (Dreg, Dreg)																	
Exponent Detection	0xC607 0000— 0xC607 0E3F	1	1	0	0	0	1	1	0	0	x	x	0	0	1	1	1
		0	0	0	0	Dest. Dreg #			x	x	x	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = EXPADJ (Dreg, Dreg_lo)																	
Exponent Detection	0xC607 8000— 0xC607 8E3F	1	1	0	0	0	1	1	0	0	x	x	0	0	1	1	1
		1	0	0	0	Dest. Dreg #			x	x	x	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = EXPADJ (Dreg_lo, Dreg_lo)																	
Exponent Detection	0xC607 C000— 0xC607 CE3F	1	1	0	0	0	1	1	0	0	x	x	0	0	1	1	1
		1	1	0	0	Dest. Dreg #			x	x	x	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = EXPADJ (Dreg_hi, Dreg_lo)																	
Exponent Detection	0xC607 4000— 0xC607 4E3F	1	1	0	0	0	1	1	0	0	x	x	0	0	1	1	1
		0	1	0	0	Dest. Dreg #			x	x	x	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = EXPADJ (Dreg, Dreg_lo) (V)																	
Maximum	0xC407 0000— 0xC407 0E3F	1	1	0	0	0	1	0	0	0	x	x	0	0	1	1	1
		0	0	0	0	Dest Dreg #			x	x	x	Source 0 Dreg #			Source 1 Dreg #		
Dreg = MAX (Dreg, Dreg)																	

Table C-17. Arithmetic Operations Instructions (Sheet 7 of 44)

Instruction		Bin															
and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Minimum	0xC407 4000— 0xC407 4E3F	1	1	0	0	0	1	0	x	x	x	0	0	0	1	1	1
		0	1	0	0	Dest Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = MIN (Dreg, Dreg)																	
Modify, Decrement	0xC40B C03F	1	1	0	0	0	1	0	x	x	x	0	0	1	0	1	1
		1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1
A0 – = A1																	
Modify, Decrement	0xC40B E03F	1	1	0	0	0	1	0	x	x	x	0	0	1	0	1	1
		1	1	1	0	0	0	0	0	0	0	1	1	1	1	1	1
A0 – = A1 (W32)																	
Modify, Decrement	0x4400— 0x443F	0	1	0	0	0	1	0	0	0	0	Source Preg #			Dest. Preg #		
Preg – = Preg																	
Modify, Decrement	0x9E70— 0x9E7F	1	0	0	1	1	1	1	0	0	1	1	1	Mreg #		Ireg #	
Ireg – = Mreg																	
Modify, Increment	0xC40B 803F	1	1	0	0	0	1	0	x	x	x	0	0	1	0	1	1
		1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
A0 += A1																	
Modify, Increment	0xC40B A03F	1	1	0	0	0	1	0	x	x	x	0	0	1	0	1	1
		1	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1
A0 += A1 (W32)																	
Modify, Increment	0x4540— 0x457F	0	1	0	0	0	1	0	1	0	1	Source Preg #			Dest. Preg #		
Preg += Preg (BREV)																	
Modify, Increment	0x9E60— 0x9E6F	1	0	0	1	1	1	1	0	0	1	1	0	Mreg #		Ireg #	
Ireg += Mreg																	

Table C-17. Arithmetic Operations Instructions (Sheet 8 of 44)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Modify, Increment	0x9EE0— 0x9EEF	1	0	0	1	1	1	1	0	1	1	1	0	Mreg #		Ireg #	
Ireg += Mreg (brev)																	
Modify, Increment	0xC40B 003F— 0xC40B 0E00	1	1	0	0	0	1	0	x	x	x	0	0	1	0	1	1
		0	0	0	0	Dest Dreg #			0	0	0	1	1	1	1	1	1
Dreg = (A0 += A1)																	
Modify, Increment	0xC40B 403F— 0xC40B 4E00	1	1	0	0	0	1	0	x	x	x	0	0	1	0	1	1
		0	1	0	0	Dest Dreg #			0	0	0	1	1	1	1	1	1
Dreg_lo = (A0 += A1)																	
Modify, Increment	0xC42B 403F— 0xC42B 4E00	1	1	0	0	0	1	0	x	x	x	1	0	1	0	1	1
		0	1	0	0	Dest Dreg #			0	0	0	1	1	1	1	1	1
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)																	
Multiply 16-Bit Operands	0xC200 2000— 0xC200 27FF	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0
		0	0	1	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi																	
Multiply 16-Bit Operands	0xC280 2000— 0xC280 27FF	1	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0
		0	0	1	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (FU)																	
Multiply 16-Bit Operands	C300 2000— 0xC300 27FF	1	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0
		0	0	1	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (IS)																	

Table C-17. Arithmetic Operations Instructions (Sheet 9 of 44)

Instruction		Bin															
and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Multiply 16-Bit Operands	0xC380 2000—	1	1	0	0	0	0	1	1	1	0	0	0	0	0	0	0
	0xC380 27FF	0	0	1	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (IU)																	
Multiply 16-Bit Operands	0xC240 2000—	1	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0
	0xC240 27FF	0	0	1	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (T)																	
Multiply 16-Bit Operands	0xC2C0 2000—	1	1	0	0	0	0	1	0	1	1	0	0	0	0	0	0
	0xC2C0 27FF	0	0	1	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (TFU)																	
Multiply 16-Bit Operands	0xC220 2000—	1	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0
	0xC220 27FF	0	0	1	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (S2RND)																	
Multiply 16-Bit Operands	0xC320 200—	1	1	0	0	0	0	1	1	0	0	1	0	0	0	0	0
	0xC320 27FF0	0	0	1	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (ISS2)																	
Multiply 16-Bit Operands	0xC360 2000—	1	1	0	0	0	0	1	1	0	1	1	0	0	0	0	0
	0xC360 27FF	0	0	1	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
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_lo = Dreg_lo_hi * Dreg_lo_hi (IH)																	
Multiply 16-Bit Operands	0xC208 2000—	1	1	0	0	0	0	1	0	0	0	0	0	1	0	0	0
	0xC208 27FF	0	0	1	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = Dreg_lo_hi * Dreg_lo_hi																	

Table C-17. Arithmetic Operations Instructions (Sheet 10 of 44)

Instruction		Bin															
and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Multiply 16-Bit Operands	0xC288 2000— 0xC288 27FF	1	1	0	0	0	0	1	0	1	0	0	0	1	0	0	0
		0	0	1	0	0	Dreg half		Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_even = Dreg_lo_hi * Dreg_lo_hi (FU)																	
Multiply 16-Bit Operands	0xC308 2000— 0xC308 27FF	1	1	0	0	0	0	1	1	0	0	0	0	1	0	0	0
		0	0	1	0	0	Dreg half		Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_even = Dreg_lo_hi * Dreg_lo_hi (IS)																	
Multiply 16-Bit Operands	0xC228 2000— 0xC228 27FF	1	1	0	0	0	0	1	0	0	0	1	0	1	0	0	0
		0	0	1	0	0	Dreg half		Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_even = Dreg_lo_hi * Dreg_lo_hi (S2RND)																	
Multiply 16-Bit Operands	0xC328 2000— 0xC328 27FF	1	1	0	0	0	0	1	1	0	0	1	0	1	0	0	0
		0	0	1	0	0	Dreg half		Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
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_even = Dreg_lo_hi * Dreg_lo_hi (ISS2)																	
Multiply 16-Bit Operands	0xC204 0000— 0xC204 C1FF	1	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0
		Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi																	
Multiply 16-Bit Operands	0xC284 0000— 0xC284 C1FF	1	1	0	0	0	0	1	0	1	0	0	0	0	1	0	0
		Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (FU)																	
Multiply 16-Bit Operands	0xC304 0000— 0xC304 C1FF	1	1	0	0	0	0	1	1	0	0	0	0	0	1	0	0
		Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IS)																	

Table C-17. Arithmetic Operations Instructions (Sheet 11 of 44)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Multiply 16-Bit Operands	0xC384 0000— 0xC384 C1FF	1	1	0	0	0	0	1	1	1	0	0	0	0	1	0	0
		Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IU)																	
Multiply 16-Bit Operands	0xC244 0000— 0xC244 C1FF	1	1	0	0	0	0	1	0	0	1	0	0	0	1	0	0
		Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (T)																	
Multiply 16-Bit Operands	0xC2C4 0000— 0xC2C4 C1FF	1	1	0	0	0	0	1	0	1	1	0	0	0	1	0	0
		Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (TFU)																	
Multiply 16-Bit Operands	0xC224 0000— 0xC224 C1FF	1	1	0	0	0	0	1	0	0	0	1	0	0	1	0	0
		Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (S2RND)																	
Multiply 16-Bit Operands	0xC324 0000— 0xC324 C1FF	1	1	0	0	0	0	1	1	0	0	1	0	0	1	0	0
		Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (ISS2)																	
Multiply 16-Bit Operands	0xC364 0000— 0xC364 C1FF	1	1	0	0	0	0	1	1	0	1	1	0	0	1	0	0
		Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IH)																	
Multiply 16-Bit Operands	0xC214 0000— 0xC214 C1FF	1	1	0	0	0	0	1	0	0	0	0	1	0	1	0	0
		Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (M)																	

Table C-17. Arithmetic Operations Instructions (Sheet 12 of 44)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Multiply 16-Bit Operands	0xC294 0000—	1	1	0	0	0	0	1	0	1	0	0	1	0	1	0	0
	0xC294 C1FF	Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (FU, M)																	
Multiply 16-Bit Operands	0xC314 0000—	1	1	0	0	0	0	1	1	0	0	0	1	0	1	0	0
	0xC314 C1FF	Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IS, M)																	
Multiply 16-Bit Operands	0xC394 0000—	1	1	0	0	0	0	1	1	1	0	0	1	0	1	0	0
	0xC394 C1FF	Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IU, M)																	
Multiply 16-Bit Operands	0xC254 0000—	1	1	0	0	0	0	1	0	0	1	0	1	0	1	0	0
	0xC254 C1FF	Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (T, M)																	
Multiply 16-Bit Operands	0xC2D4 0000—	1	1	0	0	0	0	1	0	1	1	0	1	0	1	0	0
	0xC2D4 C1FF	Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (TFU, M)																	
Multiply 16-Bit Operands	0xC234 0000—	1	1	0	0	0	0	1	0	0	0	1	1	0	1	0	0
	0xC234 C1FF	Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (S2RND, M)																	
Multiply 16-Bit Operands	0xC334 0000—	1	1	0	0	0	0	1	1	0	0	1	1	0	1	0	0
	0xC334 C1FF	Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (ISS2, M)																	

Table C-17. Arithmetic Operations Instructions (Sheet 13 of 44)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Multiply 16-Bit Operands	0xC374 0000— 0xC374 C1FF	1	1	0	0	0	0	1	1	0	1	1	1	0	1	0	0
		Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
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)																	
Multiply 16-Bit Operands	0xC20C 0000— 0xC20C C1FF	1	1	0	0	0	0	1	0	0	0	0	0	1	1	0	0
		Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi																	
Multiply 16-Bit Operands	0xC28C 0000— 0xC28C C1FF	1	1	0	0	0	0	1	0	1	0	0	0	1	1	0	0
		Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (FU)																	
Multiply 16-Bit Operands	0xC30C 0000— 0xC30C C1FF	1	1	0	0	0	0	1	1	0	0	0	0	1	1	0	0
		Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (IS)																	
Multiply 16-Bit Operands	0xC22C 0000— 0xC22C C1FF	1	1	0	0	0	0	1	0	0	0	1	0	1	1	0	0
		Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (S2RND)																	
Multiply 16-Bit Operands	0xC32C 0000— 0xC32C C1FF	1	1	0	0	0	0	1	1	0	0	1	0	1	1	0	0
		Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (ISS2)																	
Multiply 16-Bit Operands	0xC21C 0000— 0xC21C C1FF	1	1	0	0	0	0	1	0	0	0	0	1	1	1	0	0
		Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (M)																	

Table C-17. Arithmetic Operations Instructions (Sheet 14 of 44)

Instruction									Bin								
and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Multiply 16-Bit Operands	0xC29C 0000— 0xC29C C1FF	1	1	0	0	0	0	1	0	1	0	0	1	1	1	0	0
		Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (FU, M)																	
Multiply 16-Bit Operands	0xC31C 0000— 0xC31C C1FF	1	1	0	0	0	0	1	1	0	0	0	1	1	1	0	0
		Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (IS, M)																	
Multiply 16-Bit Operands	0xC239 0000— 0xC239 C1FF	1	1	0	0	0	0	1	0	0	0	1	1	1	1	0	0
		Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (S2RND, M)																	
Multiply 16-Bit Operands	0xC33C 0000— 0xC33C C1FF	1	1	0	0	0	0	1	1	0	0	1	1	1	1	0	0
		Dreg half		0	0	0	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
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	0	1	1	Source Dreg #			Dest. Dreg #		
Dreg *= Dreg																	

Table C-17. Arithmetic Operations Instructions (Sheet 15 of 44)

Instruction		Bin															
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 Accumulator																	
Legend: Dreg half determines which halves of the input operand registers to use.		Dreg half															
Dreg_lo * Dreg_lo		0	0														
Dreg_lo * Dreg_hi		0	1														
Dreg_hi * Dreg_lo		1	0														
Dreg_hi * Dreg_hi		1	1														
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.																	
Multiply and Multiply-Accumulate to Accumulator	0xC003 0000—0xC003 063F	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1
		0	0	0	0	0	Dreg half		0	0	0	src_reg_0 Dreg #			src_reg_1 Dreg #		
A0 = Dreg_lo_hi * Dreg_lo_hi																	
Multiply and Multiply-Accumulate to Accumulator	0xC083 0000—0xC083 063F	1	1	0	0	0	0	0	0	1	0	0	0	0	0	1	1
		0	0	0	0	0	Dreg half		0	0	0	src_reg_0 Dreg #			src_reg_1 Dreg #		
A0 = Dreg_lo_hi * Dreg_lo_hi (FU)																	
Multiply and Multiply-Accumulate to Accumulator	0xC103 0000—0xC103 063F	1	1	0	0	0	0	0	1	0	0	0	0	0	0	1	1
		0	0	0	0	0	Dreg half		0	0	0	src_reg_0 Dreg #			src_reg_1 Dreg #		
A0 = Dreg_lo_hi * Dreg_lo_hi (IS)																	
Multiply and Multiply-Accumulate to Accumulator	0xC063 0000—0xC063 063F	1	1	0	0	0	0	0	0	0	1	1	0	0	0	1	1
		0	0	0	0	0	Dreg half		0	0	0	src_reg_0 Dreg #			src_reg_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.																	
A0 = Dreg_lo_hi * Dreg_lo_hi (W32)																	

Table C-17. Arithmetic Operations Instructions (Sheet 16 of 44)

Instruction		Bin															
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 Accumulator	0xC003 0800— 0xC003 0E3F	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1
		0	0	0	0	1	Dreg half		0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A0 += Dreg_lo_hi * Dreg_lo_hi																	
Multiply and Multiply-Accumulate to Accumulator	0xC083 0800— 0xC083 0E3F	1	1	0	0	0	0	0	0	1	0	0	0	0	0	1	1
		0	0	0	0	1	Dreg half		0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A0 += Dreg_lo_hi * Dreg_lo_hi (FU)																	
Multiply and Multiply-Accumulate to Accumulator	0xC103 0800— 0xC103 0E3F	1	1	0	0	0	0	0	1	0	0	0	0	0	0	1	1
		0	0	0	0	1	Dreg half		0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A0 += Dreg_lo_hi * Dreg_lo_hi (IS)																	
Multiply and Multiply-Accumulate to Accumulator	0xC063 0800— 0xC063 0E3F	1	1	0	0	0	0	0	0	0	1	1	0	0	0	1	1
		0	0	0	0	1	Dreg half		0	0	0	src_reg_ 0 Dreg #			src_reg_ 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. A0 += Dreg_lo_hi * Dreg_lo_hi (W32)																	
Multiply and Multiply-Accumulate to Accumulator	0xC003 1000— 0xC003 163F	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1
		0	0	0	1	0	Dreg half		0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A0 – = Dreg_lo_hi * Dreg_lo_hi																	
Multiply and Multiply-Accumulate to Accumulator	0xC083 1000— 0xC083 163F	1	1	0	0	0	0	0	0	1	0	0	0	0	0	1	1
		0	0	0	1	0	Dreg half		0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A0 – = Dreg_lo_hi * Dreg_lo_hi (FU)																	
Multiply and Multiply-Accumulate to Accumulator	0xC103 1000— 0xC103 163F	1	1	0	0	0	0	0	1	0	0	0	0	0	0	1	1
		0	0	0	1	0	Dreg half		0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A0 – = Dreg_lo_hi * Dreg_lo_hi (IS)																	

Table C-17. Arithmetic Operations Instructions (Sheet 17 of 44)

Instruction		Bin															
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 Accumulator	0xC063 1000—	1	1	0	0	0	0	0	0	0	1	1	0	0	0	1	1
	0xC063 163F	0	0	0	1	0	Dreg half		0	0	0	src_reg_ 0 Dreg #			src_reg_ 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.																	
A0 – = Dreg_lo_hi * Dreg_lo_hi (W32)																	
Multiply and Multiply-Accumulate to Accumulator	0xC000 1800—	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0xC000 D83F	Dreg half		0	1	1	0	0	0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A1 = Dreg_lo_hi * Dreg_lo_hi																	
Multiply and Multiply-Accumulate to Accumulator	0xC080 1800—	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
	0xC080 D83F	Dreg half		0	1	1	0	0	0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A1 = Dreg_lo_hi * Dreg_lo_hi (FU)																	
Multiply and Multiply-Accumulate to Accumulator	0xC100 1800—	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
	0xC100 D83F	Dreg half		0	1	1	0	0	0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A1 = Dreg_lo_hi * Dreg_lo_hi (IS)																	
Multiply and Multiply-Accumulate to Accumulator	0xC060 1800—	1	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0
	0xC060 D83F	Dreg half		0	1	1	0	0	0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A1 = Dreg_lo_hi * Dreg_lo_hi (W32)																	
Multiply and Multiply-Accumulate to Accumulator	0xC010 1800—	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0
	0xC010 D83F	Dreg half		0	1	1	0	0	0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A1 = Dreg_lo_hi * Dreg_lo_hi (M)																	

Table C-17. Arithmetic Operations Instructions (Sheet 18 of 44)

Instruction		Bin															
and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Multiply and Multiply-Accumulate	0xC070 1800—	1	1	0	0	0	0	0	0	0	1	1	1	0	0	0	0
to Accumulator	0xC070 D83F	Dreg half		0	1	1	0	0	0	0	0	src_reg_ 0 Dreg #			src_reg_ 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.																	
A1 = Dreg_lo_hi * Dreg_lo_hi (W32, M)																	
Multiply and Multiply-Accumulate	0xC001 1800—	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
to Accumulator	0xC001 D83F	Dreg half		0	1	1	0	0	0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A1 += Dreg_lo_hi * Dreg_lo_hi																	
Multiply and Multiply-Accumulate	0xC081 1800—	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1
to Accumulator	0xC081 D83F	Dreg half		0	1	1	0	0	0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A1 += Dreg_lo_hi * Dreg_lo_hi (FU)																	
Multiply and Multiply-Accumulate	0xC101 1800—	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1
to Accumulator	0xC101 D83F	Dreg half		0	1	1	0	0	0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A1 += Dreg_lo_hi * Dreg_lo_hi (IS)																	
Multiply and Multiply-Accumulate	0xC061 1800—	1	1	0	0	0	0	0	0	0	1	1	0	0	0	0	1
to Accumulator	0xC061 D83F	Dreg half		0	1	1	0	0	0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A1 += Dreg_lo_hi * Dreg_lo_hi (W32)																	
Multiply and Multiply-Accumulate	0xC011 1800—	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1
to Accumulator	0xC011 D83F	Dreg half		0	1	1	0	0	0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A1 += Dreg_lo_hi * Dreg_lo_hi (M)																	

Table C-17. Arithmetic Operations Instructions (Sheet 19 of 44)

Instruction		Bin															
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 Accumulator	0xC071 1800— 0xC071 D83F	1	1	0	0	0	0	0	0	0	1	1	1	0	0	0	1
		Dreg half		0	1	1	0	0	0	0	0	src_reg_ 0 Dreg #			src_reg_ 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.																	
A1 += Dreg_lo_hi * Dreg_lo_hi (W32, M)																	
Multiply and Multiply-Accumulate to Accumulator	0xC002 1800— 0xC002 D83F	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
		Dreg half		0	1	1	0	0	0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A1 – = Dreg_lo_hi * Dreg_lo_hi																	
Multiply and Multiply-Accumulate to Accumulator	0xC082 1800— 0xC082 D83F	1	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0
		Dreg half		0	1	1	0	0	0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A1 – = Dreg_lo_hi * Dreg_lo_hi (FU)																	
Multiply and Multiply-Accumulate to Accumulator	0xC102 1800— 0xC102 D83F	1	1	0	0	0	0	0	1	0	0	0	0	0	0	1	0
		Dreg half		0	1	1	0	0	0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A1 – = Dreg_lo_hi * Dreg_lo_hi (IS)																	
Multiply and Multiply-Accumulate to Accumulator	0xC062 1800— 0xC062 D83F	1	1	0	0	0	0	0	0	0	1	1	0	0	0	1	0
		Dreg half		0	1	1	0	0	0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A1 – = Dreg_lo_hi * Dreg_lo_hi (W32)																	
Multiply and Multiply-Accumulate to Accumulator	0xC022 1800— 0xC022 D83F	1	1	0	0	0	0	0	0	0	0	0	1	0	0	1	0
		Dreg half		0	1	1	0	0	0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A1 – = Dreg_lo_hi * Dreg_lo_hi (M)																	

Table C-17. Arithmetic Operations Instructions (Sheet 20 of 44)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Multiply and Multiply-Accumulate to Accumulator	0xC072 1800— 0xC072 D83F	1	1	0	0	0	0	0	0	0	1	1	1	0	0	1	0
		Dreg half		0	1	1	0	0	0	0	0	src_reg_ 0 Dreg #			src_reg_ 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.

A1 – = Dreg_lo_hi * Dreg_lo_hi (W32, M)

Multiply and Multiply-Accumulate to Accumulator																
LEGEND: Dreg half determines which halves of the input operand registers to use.	Dreg half															
Dreg_lo * Dreg_lo	0	0														
Dreg_lo * Dreg_hi	0	1														
Dreg_hi * Dreg_lo	1	0														
Dreg_hi * Dreg_hi	1	1														

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.

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 Half Register	0xC003 2000— 0xC003 27FF	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1
		0	0	1	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		

Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi)

Multiply and Multiply-Accumulate to Half Register	0xC083 2000— 0xC083 27FF	1	1	0	0	0	0	0	0	1	0	0	0	0	0	1	1
		0	0	1	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		

Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (FU)

Multiply and Multiply-Accumulate to Half Register	0xC103 2000— 0xC103 27FF	1	1	0	0	0	0	0	1	0	0	0	0	0	0	1	1
		0	0	1	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		

Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (IS)

Table C-17. Arithmetic Operations Instructions (Sheet 21 of 44)

Instruction		Bin															
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	0xC183 2000— 0xC183 27FF	1	1	0	0	0	0	0	1	1	0	0	0	0	0	1	1
		0	0	1	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (IU)																	
Multiply and Multiply-Accumulate to Half Register	0xC043 2000— 0xC043 27FF	1	1	0	0	0	0	0	0	0	1	0	0	0	0	1	1
		0	0	1	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (T)																	
Multiply and Multiply-Accumulate to Half Register	0xC0C3 2000— 0xC0C3 27FF	1	1	0	0	0	0	0	0	1	1	0	0	0	0	1	1
		0	0	1	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (TFU)																	
Multiply and Multiply-Accumulate to Half Register	0xC023 2000— 0xC023 27FF	1	1	0	0	0	0	0	0	0	0	1	0	0	0	1	1
		0	0	1	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (S2RND)																	
Multiply and Multiply-Accumulate to Half Register	0xC123 2000— 0xC123 27FF	1	1	0	0	0	0	0	1	0	0	1	0	0	0	1	1
		0	0	1	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (ISS2)																	
Multiply and Multiply-Accumulate to Half Register	0xC163 2000— 0xC163 27FF	1	1	0	0	0	0	0	1	0	1	1	0	0	0	1	1
		0	0	1	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 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_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (IH)																	
Multiply and Multiply-Accumulate to Half Register	0xC003 2800— 0xC003 2FFF	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1
		0	0	1	0	1	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi)																	

Table C-17. Arithmetic Operations Instructions (Sheet 22 of 44)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Multiply and Multiply-Accumulate to Half Register	0xC083 2800— 0xC083 2FFF	1	1	0	0	0	0	0	0	1	0	0	0	0	0	1	1
		0	0	1	0	1	Dreg half		Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (FU)																	
Multiply and Multiply-Accumulate to Half Register	0xC103 2800— 0xC103 2FFF	1	1	0	0	0	0	0	1	0	0	0	0	0	0	1	1
		0	0	1	0	1	Dreg half		Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (IS)																	
Multiply and Multiply-Accumulate to Half Register	0xC183 2800— 0xC183 2FFF	1	1	0	0	0	0	0	1	1	0	0	0	0	0	1	1
		0	0	1	0	1	Dreg half		Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (IU)																	
Multiply and Multiply-Accumulate to Half Register	0xC043 2800— 0xC043 2FFF	1	1	0	0	0	0	0	0	0	1	0	0	0	0	1	1
		0	0	1	0	1	Dreg half		Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (T)																	
Multiply and Multiply-Accumulate to Half Register	0xC0C3 2800— 0xC0C3 2FFF	1	1	0	0	0	0	0	0	1	1	0	0	0	0	1	1
		0	0	1	0	1	Dreg half		Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (TFU)																	
Multiply and Multiply-Accumulate to Half Register	0xC023 2800— 0xC023 2FFF	1	1	0	0	0	0	0	0	0	0	1	0	0	0	1	1
		0	0	1	0	1	Dreg half		Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (S2RND)																	
Multiply and Multiply-Accumulate to Half Register	0xC123 2800— 0xC123 2FFF	1	1	0	0	0	0	0	1	0	0	1	0	0	0	1	1
		0	0	1	0	1	Dreg half		Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (ISS2)																	

Table C-17. Arithmetic Operations Instructions (Sheet 23 of 44)

Instruction		Bin															
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	0xC163 2800—0xC163 2FFF	1	1	0	0	0	0	0	1	0	1	1	0	0	0	1	1
		0	0	1	0	1	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 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_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (IH)																	
Multiply and Multiply-Accumulate to Half Register	0xC003 3000—0xC003 37FF	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1
		0	0	1	1	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 – = Dreg_lo_hi * Dreg_lo_hi)																	
Multiply and Multiply-Accumulate to Half Register	0xC083 3000—0xC083 37FF	1	1	0	0	0	0	0	0	1	0	0	0	0	0	1	1
		0	0	1	1	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 – = Dreg_lo_hi * Dreg_lo_hi) (FU)																	
Multiply and Multiply-Accumulate to Half Register	0xC103 3000—0xC103 37FF	1	1	0	0	0	0	0	1	0	0	0	0	0	0	1	1
		0	0	1	1	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 – = Dreg_lo_hi * Dreg_lo_hi) (IS)																	
Multiply and Multiply-Accumulate to Half Register	0xC183 3000—0xC183 37FF	1	1	0	0	0	0	0	1	1	0	0	0	0	0	1	1
		0	0	1	1	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 – = Dreg_lo_hi * Dreg_lo_hi) (IU)																	
Multiply and Multiply-Accumulate to Half Register	0xC043 3000—0xC043 37FF	1	1	0	0	0	0	0	0	0	1	0	0	0	0	1	1
		0	0	1	1	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 – = Dreg_lo_hi * Dreg_lo_hi) (T)																	
Multiply and Multiply-Accumulate to Half Register	0xC0C3 3000—0xC0C3 37FF	1	1	0	0	0	0	0	0	1	1	0	0	0	0	1	1
		0	0	1	1	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 – = Dreg_lo_hi * Dreg_lo_hi) (TFU)																	

Table C-17. Arithmetic Operations Instructions (Sheet 24 of 44)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Multiply and Multiply-Accumulate to Half Register	0xC023 3000— 0xC023 37FF	1	1	0	0	0	0	0	0	0	0	1	0	0	0	1	1
		0	0	1	1	0	Dreg half		Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_lo = (A0 – = Dreg_lo_hi * Dreg_lo_hi) (S2RND)																	
Multiply and Multiply-Accumulate to Half Register	0xC123 3000— 0xC123 37FF	1	1	0	0	0	0	0	1	0	0	1	0	0	0	1	1
		0	0	1	1	0	Dreg half		Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_lo = (A0 – = Dreg_lo_hi * Dreg_lo_hi) (ISS2)																	
Multiply and Multiply-Accumulate to Half Register	0xC163 3000— 0xC163 37FF	1	1	0	0	0	0	0	1	0	1	1	0	0	0	1	1
		0	0	1	1	0	Dreg half		Dest. Dreg #			src_reg_0 Dreg #			src_reg_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_lo = (A0 – = Dreg_lo_hi * Dreg_lo_hi) (IH)																	
Multiply and Multiply-Accumulate to Half Register	0xC004 1800— 0xC004 D9FF	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi)																	
Multiply and Multiply-Accumulate to Half Register	0xC084 1800— 0xC084 D9FF	1	1	0	0	0	0	0	0	1	0	0	0	0	1	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (FU)																	
Multiply and Multiply-Accumulate to Half Register	0xC104 1800— 0xC104 D9FF	1	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IS)																	
Multiply and Multiply-Accumulate to Half Register	0xC184 1800— 0xC184 D9FF	1	1	0	0	0	0	0	1	1	0	0	0	0	1	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IU)																	

Table C-17. Arithmetic Operations Instructions (Sheet 25 of 44)

Instruction		Bin															
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	0xC044 1800—0xC044 D9FF	1	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (T)																	
Multiply and Multiply-Accumulate to Half Register	0xC0C4 1800—0xC0C4 D9FF	1	1	0	0	0	0	0	0	1	1	0	0	0	1	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (TFU)																	
Multiply and Multiply-Accumulate to Half Register	0xC024 1800—0xC024 D9FF	1	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (S2RND)																	
Multiply and Multiply-Accumulate to Half Register	0xC124 1800—0xC124 D9FF	1	1	0	0	0	0	0	1	0	0	1	0	0	1	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (ISS2)																	
Multiply and Multiply-Accumulate to Half Register	0xC164 1800—0xC164 D9FF	1	1	0	0	0	0	0	1	0	1	1	0	0	1	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IH)																	
Multiply and Multiply-Accumulate to Half Register	0xC014 1800—0xC014 D9FF	1	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (M)																	
Multiply and Multiply-Accumulate to Half Register	0xC094 1800—0xC094 D9FF	1	1	0	0	0	0	0	0	1	0	0	1	0	1	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (FU, M)																	

Table C-17. Arithmetic Operations Instructions (Sheet 26 of 44)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Multiply and Multiply-Accumulate to Half Register	0xC114 1800— 0xC114 D9FF	1	1	0	0	0	0	0	1	0	0	0	1	0	1	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IS, M)																	
Multiply and Multiply-Accumulate to Half Register	0xC194 1800— 0xC194 D9FF	1	1	0	0	0	0	0	1	1	0	0	1	0	1	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IU, M)																	
Multiply and Multiply-Accumulate to Half Register	0xC054 1800— 0xC054 D9FF	1	1	0	0	0	0	0	0	0	1	0	1	0	1	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (T, M)																	
Multiply and Multiply-Accumulate to Half Register	0xC0D4 1800— 0xC0D4 D9FF	1	1	0	0	0	0	0	0	1	1	0	1	0	1	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (TFU, M)																	
Multiply and Multiply-Accumulate to Half Register	0xC034 1800— 0xC034 D9FF	1	1	0	0	0	0	0	0	0	0	1	1	0	1	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)																	
Multiply and Multiply-Accumulate to Half Register	0xC134 1800— 0xC134 D9FF	1	1	0	0	0	0	0	1	0	0	1	1	0	1	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)																	
Multiply and Multiply-Accumulate to Half Register	0xC174 1800— 0xC174 D9FF	1	1	0	0	0	0	0	1	0	1	1	1	0	1	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 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_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IH, M)																	

Table C-17. Arithmetic Operations Instructions (Sheet 27 of 44)

Instruction		Bin															
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	0xC005 1800— 0xC005 D9FF	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi)																	
Multiply and Multiply-Accumulate to Half Register	0xC085 1800— 0xC085 D9FF	1	1	0	0	0	0	0	0	1	0	0	0	0	1	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (FU)																	
Multiply and Multiply-Accumulate to Half Register	0xC105 1800— 0xC105 D9FF	1	1	0	0	0	0	0	1	0	0	0	0	0	1	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IS)																	
Multiply and Multiply-Accumulate to Half Register	0xC185 1800— 0xC185 D9FF	1	1	0	0	0	0	0	1	1	0	0	0	0	1	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IU)																	
Multiply and Multiply-Accumulate to Half Register	0xC045 1800— 0xC045 D9FF	1	1	0	0	0	0	0	0	0	1	0	0	0	1	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (T)																	
Multiply and Multiply-Accumulate to Half Register	0xC0C5 1800— 0xC0C5 D9FF	1	1	0	0	0	0	0	0	1	1	0	0	0	1	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (TFU)																	
Multiply and Multiply-Accumulate to Half Register	0xC025 1800— 0xC025 D9FF	1	1	0	0	0	0	0	0	0	0	1	0	0	1	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (S2RND)																	

Table C-17. Arithmetic Operations Instructions (Sheet 28 of 44)

Instruction									Bin								
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	0	1	0	0	1	0	0	1	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			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	0	1	0	1	1	0	0	1	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			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	0	0	1	0	1	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			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	0	1	0	0	1	0	1	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			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	0	1	0	0	0	1	0	1	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			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	0	1	1	0	0	1	0	1	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			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	0	0	0	1	0	1	0	1	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (T, M)																	

Table C-17. Arithmetic Operations Instructions (Sheet 29 of 44)

Instruction									Bin								
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	0xC0D5 1800— 0xC0D5 D9FF	1	1	0	0	0	0	0	0	1	1	0	1	0	1	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (TFU, M)																	
Multiply and Multiply-Accumulate to Half Register	0xC035 1800— 0xC035 D9FF	1	1	0	0	0	0	0	0	0	0	1	1	0	1	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)																	
Multiply and Multiply-Accumulate to Half Register	0xC135 1800— 0xC135 D9FF	1	1	0	0	0	0	0	1	0	0	1	1	0	1	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)																	
Multiply and Multiply-Accumulate to Half Register	0xC175 1800— 0xC175 D9FF	1	1	0	0	0	0	0	1	0	1	1	1	0	1	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 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_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IH, M)																	
Multiply and Multiply-Accumulate to Half Register	0xC006 1800— 0xC006 D9FF	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi)																	
Multiply and Multiply-Accumulate to Half Register	0xC086 1800— 0xC086 D9FF	1	1	0	0	0	0	0	0	1	0	0	0	0	1	1	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (FU)																	
Multiply and Multiply-Accumulate to Half Register	0xC106 1800— 0xC106 D9FF	1	1	0	0	0	0	0	1	0	0	0	0	0	1	1	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (IS)																	

Table C-17. Arithmetic Operations Instructions (Sheet 30 of 44)

Instruction										Bin							
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	0xC186 1800— 0xC186 D9FF	1	1	0	0	0	0	0	1	1	0	0	0	0	1	1	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (IU)																	
Multiply and Multiply-Accumulate to Half Register	0xC046 1800— 0xC046 D9FF	1	1	0	0	0	0	0	0	0	1	0	0	0	1	1	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (T)																	
Multiply and Multiply-Accumulate to Half Register	0xC0C6 1800— 0xC0C6 D9FF	1	1	0	0	0	0	0	0	1	1	0	0	0	1	1	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (TFU)																	
Multiply and Multiply-Accumulate to Half Register	0xC026 1800— 0xC026 D9FF	1	1	0	0	0	0	0	0	0	0	1	0	0	1	1	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (S2RND)																	
Multiply and Multiply-Accumulate to Half Register	0xC126 1800— 0xC126 D9FF	1	1	0	0	0	0	0	1	0	0	1	0	0	1	1	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (ISS2)																	
Multiply and Multiply-Accumulate to Half Register	0xC166 1800— 0xC166 D9FF	1	1	0	0	0	0	0	1	0	1	1	0	0	1	1	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (IH)																	
Multiply and Multiply-Accumulate to Half Register	0xC016 1800— 0xC016 D9FF	1	1	0	0	0	0	0	0	0	0	0	1	0	1	1	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (M)																	

Table C-17. Arithmetic Operations Instructions (Sheet 31 of 44)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Multiply and Multiply-Accumulate	0xC096 1800—	1	1	0	0	0	0	0	0	1	0	0	1	0	1	1	0
to Half Register	0xC096 D9FF	Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (FU, M)																	
Multiply and Multiply-Accumulate	0xC116 1800—	1	1	0	0	0	0	0	1	0	0	0	1	0	1	1	0
to Half Register	0xC116 D9FF	Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (IS, M)																	
Multiply and Multiply-Accumulate	0xC196 1800—	1	1	0	0	0	0	0	1	1	0	0	1	0	1	1	0
to Half Register	0xC196 D9FF	Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (IU, M)																	
Multiply and Multiply-Accumulate	0xC056 1800—	1	1	0	0	0	0	0	0	0	1	0	1	0	1	1	0
to Half Register	0xC056 D9FF	Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (T, M)																	
Multiply and Multiply-Accumulate	0xC0D6 1800—	1	1	0	0	0	0	0	0	1	1	0	1	0	1	1	0
to Half Register	0xC0D6 D9FF	Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (TFU, M)																	
Multiply and Multiply-Accumulate	0xC036 1800—	1	1	0	0	0	0	0	0	0	0	1	1	0	1	1	0
to Half Register	0xC036 D9FF	Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)																	
Multiply and Multiply-Accumulate	0xC136 1800—	1	1	0	0	0	0	0	1	0	0	1	1	0	1	1	0
to Half Register	0xC136 D9FF	Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)																	

Table C-17. Arithmetic Operations Instructions (Sheet 32 of 44)

Instruction and Version	Opcode Range	Bin 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Multiply and Multiply-Accumulate to Half Register	0xC176 1800— 0xC176 D9FF	1	1	0	0	0	0	0	1	0	1	1	1	0	1	1	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 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_hi = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (IH, M)

Multiply and Multiply-Accumulate to Half Register																
LEGEND: Dreg half determines which halves of the input operand registers to use.	Dreg half															
Dreg_lo * Dreg_lo	0	0														
Dreg_lo * Dreg_hi	0	1														
Dreg_hi * Dreg_lo	1	0														
Dreg_hi * Dreg_hi	1	1														

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.

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	0xC00D 0000— 0xC00D 07FF	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1
		0	0	0	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		

Dreg_even = (A0 = Dreg_lo_hi * Dreg_lo_hi)

Multiply and Multiply-Accumulate to Data Register	0xC08D 0000— 0xC08D 07FF	1	1	0	0	0	0	0	0	1	0	0	0	1	0	1	1
		0	0	0	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		

Dreg_even = (A0 = Dreg_lo_hi * Dreg_lo_hi) (FU)

Table C-17. Arithmetic Operations Instructions (Sheet 33 of 44)

Instruction and Version	Opcode Range	Bin 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	0	1	0	1	1
		0	0	0	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = (A0 = Dreg_lo_hi * Dreg_lo_hi) (IS)																	
Multiply and Multiply-Accumulate to Data Register	0xC02D 0000— 0xC02D 07FF	1	1	0	0	0	0	0	0	0	0	1	0	1	0	1	1
		0	0	0	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = (A0 = Dreg_lo_hi * Dreg_lo_hi) (S2RND)																	
Multiply and Multiply-Accumulate to Data Register	0xC12D 0000— 0xC12D 07FF	1	1	0	0	0	0	0	1	0	0	1	0	1	0	1	1
		0	0	0	0	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 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.

Instruction and Version	Opcode Range	Bin 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Dreg_even = (A0 = Dreg_lo_hi * 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	0	1	0	1	1
		0	0	0	0	1	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = (A0 += Dreg_lo_hi * Dreg_lo_hi)																	
Multiply and Multiply-Accumulate to Data Register	0xC08D 0800— 0xC08D 0FFF	1	1	0	0	0	0	0	0	1	0	0	0	1	0	1	1
		0	0	0	0	1	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = (A0 += Dreg_lo_hi * Dreg_lo_hi) (FU)																	
Multiply and Multiply-Accumulate to Data Register	0xC10D 0800— 0xC10D 0FFF	1	1	0	0	0	0	0	1	0	0	0	0	1	0	1	1
		0	0	0	0	1	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = (A0 += Dreg_lo_hi * 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	1	0	1	0	1	1
		0	0	0	0	1	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = (A0 += Dreg_lo_hi * Dreg_lo_hi) (S2RND)																	

Table C-17. Arithmetic Operations Instructions (Sheet 34 of 44)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Multiply and Multiply-Accumulate to Data Register	0xC12D 0800— 0xC12D 0FFF	1	1	0	0	0	0	0	1	0	0	1	0	1	0	1	1
		0	0	0	0	1	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 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 = (A0 += Dreg_lo_hi * Dreg_lo_hi) (ISS2)																	
Multiply and Multiply-Accumulate to Data Register	0xC00D 1000— 0xC00D 17FF	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1
		0	0	0	1	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = (A0 – = Dreg_lo_hi * Dreg_lo_hi)																	
Multiply and Multiply-Accumulate to Data Register	0xC08D 1000— 0xC08D 17FF	1	1	0	0	0	0	0	0	1	0	0	0	1	0	1	1
		0	0	0	1	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = (A0 – = Dreg_lo_hi * Dreg_lo_hi) (FU)																	
Multiply and Multiply-Accumulate to Data Register	0xC10D 1000— 0xC10D 17FF	1	1	0	0	0	0	0	1	0	0	0	0	1	0	1	1
		0	0	0	1	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = (A0 – = Dreg_lo_hi * Dreg_lo_hi) (IS)																	
Multiply and Multiply-Accumulate to Data Register	0xC02D 1000— 0xC02D 17FF	1	1	0	0	0	0	0	0	0	0	1	0	1	0	1	1
		0	0	0	1	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = (A0 – = Dreg_lo_hi * Dreg_lo_hi) (S2RND																	
Multiply and Multiply-Accumulate to Data Register	0xC12D 1000— 0xC12D 17FF	1	1	0	0	0	0	0	1	0	0	1	0	1	0	1	1
		0	0	0	1	0	Dreg half		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 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 = (A0 – = Dreg_lo_hi * Dreg_lo_hi) (ISS2)																	

Table C-17. Arithmetic Operations Instructions (Sheet 35 of 44)

Instruction									Bin								
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 Data Register	0xC008 1800— 0xC008 D9FF	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi)																	
Multiply and Multiply-Accumulate to Data Register	0xC088 1800— 0xC088 D9FF	1	1	0	0	0	0	0	0	1	0	0	0	1	0	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (FU)																	
Multiply and Multiply-Accumulate to Data Register	0xC108 1800— 0xC108 D9FF	1	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IS)																	
Multiply and Multiply-Accumulate to Data Register	0xC028 1800— 0xC028 D9FF	1	1	0	0	0	0	0	0	0	0	1	0	1	0	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (S2RND)																	
Multiply and Multiply-Accumulate to Data Register	0xC128 1800— 0xC128 D9FF	1	1	0	0	0	0	0	1	0	0	1	0	1	0	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (ISS2)																	
Multiply and Multiply-Accumulate to Data Register	0xC018 1800— 0xC018 D9FF	1	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (M)																	
Multiply and Multiply-Accumulate to Data Register	0xC098 1800— 0xC098 D9FF	1	1	0	0	0	0	0	0	1	0	0	1	1	0	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (FU, M)																	

Table C-17. Arithmetic Operations Instructions (Sheet 36 of 44)

Instruction									Bin								
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 Data Register	0xC118 1800— 0xC118 D9FF	1	1	0	0	0	0	0	1	0	0	0	1	1	0	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IS, M)																	
Multiply and Multiply-Accumulate to Data Register	0xC038 1800— 0xC038 D9FF	1	1	0	0	0	0	0	0	0	0	1	1	1	0	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)																	
Multiply and Multiply-Accumulate to Data Register	0xC138 1800— 0xC138 D9FF	1	1	0	0	0	0	0	1	0	0	1	1	1	0	0	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 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_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)																	
Multiply and Multiply-Accumulate to Data Register	0xC009 1800— 0xC009 D9FF	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi)																	
Multiply and Multiply-Accumulate to Data Register	0xC089 1800— 0xC089 D9FF	1	1	0	0	0	0	0	0	1	0	0	0	1	0	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (FU)																	
Multiply and Multiply-Accumulate to Data Register	0xC109 1800— 0xC109 D9FF	1	1	0	0	0	0	0	1	0	0	0	0	1	0	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IS)																	
Multiply and Multiply-Accumulate to Data Register	0xC029 1800— 0xC029 D9FF	1	1	0	0	0	0	0	0	0	0	1	0	1	0	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (S2RND)																	

Table C-17. Arithmetic Operations Instructions (Sheet 37 of 44)

Instruction									Bin								
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 Data Register	0xC129 1800— 0xC129 D9FF	1	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (ISS2)																	
Multiply and Multiply-Accumulate to Data Register	0xC019 1800— 0xC019 D9FF	1	1	0	0	0	0	0	0	0	0	0	1	1	0	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (M)																	
Multiply and Multiply-Accumulate to Data Register	0xC099 1800— 0xC099 D9FF	1	1	0	0	0	0	0	0	1	0	0	1	1	0	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (FU, M)																	
Multiply and Multiply-Accumulate to Data Register	0xC119 1800— 0xC119 D9FF	1	1	0	0	0	0	0	1	0	0	0	1	1	0	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IS, M)																	
Multiply and Multiply-Accumulate to Data Register	0xC039 1800— 0xC039 D9FF	1	1	0	0	0	0	0	0	0	0	1	1	1	0	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)																	
Multiply and Multiply-Accumulate to Data Register	0xC139 1800— 0xC139 D9FF	1	1	0	0	0	0	0	1	0	0	1	1	1	0	0	1
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 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_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)																	
Multiply and Multiply-Accumulate to Data Register	0xC00A 1800— 0xC00A D9FF	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 – = Dreg_lo_hi * Dreg_lo_hi)																	

Table C-17. Arithmetic Operations Instructions (Sheet 38 of 44)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Multiply and Multiply-Accumulate	0xC08A 1800—	1	1	0	0	0	0	0	0	1	0	0	0	1	0	1	0
to Data Register	0xC08A D9FF	Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (FU)																	
Multiply and Multiply-Accumulate	0xC10A 1800—	1	1	0	0	0	0	0	1	0	0	0	0	1	0	1	0
to Data Register	0xC10A D9FF	Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (IS)																	
Multiply and Multiply-Accumulate	0xC02A 1800—	1	1	0	0	0	0	0	0	0	0	1	0	1	0	1	0
to Data Register	0xC02A D9FF	Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (S2RND)																	
Multiply and Multiply-Accumulate	0xC12A 1800—	1	1	0	0	0	0	0	1	0	0	1	0	1	0	1	0
to Data Register	0xC12A D9FF	Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (ISS2)																	
Multiply and Multiply-Accumulate	0xC01A 1800—	1	1	0	0	0	0	0	0	0	0	0	1	1	0	1	0
to Data Register	0xC01A D9FF	Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (M)																	
Multiply and Multiply-Accumulate	0xC09A 1800—	1	1	0	0	0	0	0	0	1	0	0	1	1	0	1	0
to Data Register	0xC09A D9FF	Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (FU, M)																	
Multiply and Multiply-Accumulate	0xC11A 1800—	1	1	0	0	0	0	0	1	0	0	0	1	1	0	1	0
to Data Register	0xC11A D9FF	Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (IS, M)																	

Table C-17. Arithmetic Operations Instructions (Sheet 39 of 44)

Instruction		Bin															
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 Data Register	0xC03A 1800— 0xC03A D9FF	1	1	0	0	0	0	0	0	0	0	1	1	1	0	1	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_odd = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)																	
Multiply and Multiply-Accumulate to Data Register	0xC13A 1800— 0xC13A D9FF	1	1	0	0	0	0	0	1	0	0	1	1	1	0	1	0
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 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_odd = (A1 – = Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)																	
Multiply and Multiply-Accumulate to Data Register																	
LEGEND: Dreg half determines which halves of the input operand registers to use.		Dreg half															
Dreg_lo * Dreg_lo		0	0														
Dreg_lo * Dreg_hi		0	1														
Dreg_hi * Dreg_lo		1	0														
Dreg_hi * Dreg_hi		1	1														
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)	0x4380— 0x43BF	0	1	0	0	0	0	1	1	1	0	Source Dreg #			Dest. Dreg #		
Dreg = – Dreg																	
Negate (Two's-Complement)	0xC407 C000— 0xC407 CFC0	1	1	0	0	0	1	0	x	x	x	0	0	0	1	1	1
		1	1	0	0	Dest. Dreg #			0	0	0	Source Dreg #			0	0	0
Dreg = – Dreg (NS)																	

Table C-17. Arithmetic Operations Instructions (Sheet 40 of 44)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Negate (Two's-Complement)	0xC407 E000—	1	1	0	0	0	1	0	x	x	x	0	0	0	1	1	1
	0xC407 EFC0	1	1	1	0	Dest. Dreg #			0	0	0	Source Dreg #			0	0	0
Dreg = – Dreg (S)																	
Negate (Two's-Complement)	0xC40E 003F	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	0	1	1	1	1	1	1
A0 = – A0																	
Negate (Two's-Complement)	0xC40E 403F	1	1	0	0	0	1	0	x	x	x	0	0	1	1	1	0
		0	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1
A0 = – A1																	
Negate (Two's-Complement)	0xC42E 003F	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	0	1	1	1	1	1	1
A1 = – A0																	
Negate (Two's-Complement)	0xC42E 403F	1	1	0	0	0	1	0	x	x	x	1	0	1	1	1	0
		0	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1
A1 = – A1																	
Negate (Two's-Complement)	0xC40E C03F	1	1	0	0	0	1	0	x	x	x	0	0	1	1	1	0
		1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1
A1 = – A1, A0 = – A0																	
Round to Half Word	0xC40C C000—	1	1	0	0	0	1	0	x	x	x	0	0	1	1	0	0
	0xC40C CE38	1	1	0	0	Dest. Dreg #			0	0	0	Source Dreg #			0	0	0
Dreg_lo = Dreg (RND)																	
Round to Half Word	0xC42C C000—	1	1	0	0	0	1	0	x	x	x	1	0	1	1	0	0
	0xC42C CE38	1	1	0	0	Dest. Dreg #			0	0	0	Source Dreg #			0	0	0
Dreg_hi = Dreg (RND)																	

Table C-17. Arithmetic Operations Instructions (Sheet 41 of 44)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Saturate	0xC408 203F	1	1	0	0	0	1	0	x	x	x	0	0	1	0	0	0
		0	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1
A0 = A0 (S)																	
Saturate	0xC408 603F	1	1	0	0	0	1	0	x	x	x	0	0	1	0	0	0
		0	1	1	0	0	0	0	0	0	0	1	1	1	1	1	1
A1 = A1 (S)																	
Saturate	0xC408 A03F	1	1	0	0	0	1	0	x	x	x	0	0	1	0	0	0
		1	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1
A1 = A1 (S), A0 = A0 (S)																	
Sign Bit	0xC605 0000— 0xC605 0E07	1	1	0	0	0	1	1	0	0	x	x	0	0	1	0	1
		0	0	0	0	Dest. Dreg #			x	x	x	0	0	0	Source Dreg #		
Dreg_lo = SIGNBITS Dreg																	
Sign Bit	0xC605 4000— 0xC605 4E07	1	1	0	0	0	1	1	0	0	x	x	0	0	1	0	1
		0	1	0	0	Dest. Dreg #			x	x	x	0	0	0	Source Dreg #		
Dreg_lo = SIGNBITS Dreg_lo																	
Sign Bit	0xC605 8000— 0xC605 8E07	1	1	0	0	0	1	1	0	0	x	x	0	0	1	0	1
		1	0	0	0	Dest. Dreg #			x	x	x	0	0	0	Source Dreg #		
Dreg_lo = SIGNBITS Dreg_hi																	
Sign Bit	0xC606 0000— 0xC606 0E00	1	1	0	0	0	1	1	0	0	x	x	0	0	1	1	0
		0	0	0	0	Dest. Dreg #			x	x	x	0	0	0	0	0	0
Dreg_lo = SIGNBITS A0																	
Sign Bit	0xC606 4000— 0xC606 4E00	1	1	0	0	0	1	1	0	0	x	x	0	0	1	1	0
		0	1	0	0	Dest. Dreg #			x	x	x	0	0	0	0	0	0
Dreg_lo = SIGNBITS A1																	

Table C-17. Arithmetic Operations Instructions (Sheet 42 of 44)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Subtract	0x5200— 0x53FF	0	1	0	1	0	0	1	Dest. Dreg #			Src 1 Dreg #			Src 0 Dreg #		
Dreg = Dreg – Dreg																	
Subtract	0xC404 4000— 0xC404 4E3F	1	1	0	0	0	1	0	x	x	x	0	0	0	1	0	0
		0	1	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg – Dreg (NS)																	
Subtract	0xC404 6000— 0xC404 6E3F	1	1	0	0	0	1	0	x	x	x	0	0	0	1	0	0
		0	1	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg – Dreg (S)																	
Subtract	0xC403 0000— 0xC403 0E3F	1	1	0	0	0	1	0	x	x	x	0	0	0	0	1	1
		0	0	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = Dreg_lo – Dreg_lo (NS)																	
Subtract	0xC403 4000— 0xC403 4E3F	1	1	0	0	0	1	0	x	x	x	0	0	0	0	1	1
		0	1	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = Dreg_lo – Dreg_hi (NS)																	
Subtract	0xC403 8000— 0xC403 8E3F	1	1	0	0	0	1	0	x	x	x	0	0	0	0	1	1
		1	0	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = Dreg_hi – Dreg_lo (NS)																	
Subtract	0xC403 C000— 0xC403 CE3F	1	1	0	0	0	1	0	x	x	x	0	0	0	0	1	1
		1	1	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = Dreg_hi – Dreg_hi (NS)																	
Subtract	0xC423 0000— 0xC423 0E3F	1	1	0	0	0	1	0	x	x	x	1	0	0	0	1	1
		0	0	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_hi = Dreg_lo – Dreg_lo (NS)																	

Table C-17. Arithmetic Operations Instructions (Sheet 43 of 44)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Subtract	0xC423 4000—	1	1	0	0	0	1	0	x	x	x	1	0	0	0	1	1
	0xC423 4E3F	0	1	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_hi = Dreg_lo – Dreg_hi (NS)																	
Subtract	0xC423 8000—	1	1	0	0	0	1	0	x	x	x	1	0	0	0	1	1
	0xC423 8E3F	1	0	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_hi = Dreg_hi – Dreg_lo (NS)																	
Subtract	0xC423 C000—	1	1	0	0	0	1	0	x	x	x	1	0	0	0	1	1
	0xC423 CE3F	1	1	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_hi = Dreg_hi – Dreg_hi (NS)																	
Subtract	0xC403 2000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	1	1
	0xC403 2E3F	0	0	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = Dreg_lo – Dreg_lo (S)																	
Subtract	0xC403 6000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	1	1
	0xC403 6E3F	0	1	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = Dreg_lo – Dreg_hi (S)																	
Subtract	0xC403 A000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	1	1
	0xC403 AE3F	1	0	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = Dreg_hi – Dreg_lo (S)																	
Subtract	0xC403 E000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	1	1
	0xC403 EE3F	1	1	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_lo = Dreg_hi – Dreg_hi (S)																	

Table C-17. Arithmetic Operations Instructions (Sheet 44 of 44)

Instruction										Bin							
and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Subtract	0xC423 2000— 0xC423 2E3F	1	1	0	0	0	1	0	x	x	x	1	0	0	0	1	1
		0	0	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_hi = Dreg_lo – Dreg_lo (S)																	
Subtract	0xC423 6000— 0xC423 6E3F	1	1	0	0	0	1	0	x	x	x	1	0	0	0	1	1
		0	1	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_hi = Dreg_lo – Dreg_hi (S)																	
Subtract	0xC423 A000— 0xC423 AE3F	1	1	0	0	0	1	0	x	x	x	1	0	0	0	1	1
		1	0	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_hi = Dreg_hi – Dreg_lo (S)																	
Subtract	0xC423 E000— 0xC423 EE3F	1	1	0	0	0	1	0	x	x	x	1	0	0	0	1	1
		1	1	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_hi = Dreg_hi – Dreg_hi (S)																	
Subtract Immediate	0x9F64— 0x9F67	1	0	0	1	1	1	1	1	0	1	1	0	0	1	Ireg #	
Ireg – = 2																	
Subtract Immediate	0x9F6C— 0x9F6F	1	0	0	1	1	1	1	1	0	1	1	0	1	1	Ireg #	
Ireg – = 4																	

External Event Management Instructions

Table C-18. External Event Management Instructions (Sheet 1 of 2)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Idle IDLE	0x0020	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
Core Synchronize CSYNC	0x0023	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1
System Synchronize SSYNC	0x0024	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0
Force Emulation EMUEXCPT	0x0025	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1
Disable Interrupts CLI Dreg	0x0030— 0x0037	0	0	0	0	0	0	0	0	0	0	1	1	Dreg #			
Enable Interrupts STI Dreg	0x0040— 0x0047	0	0	0	0	0	0	0	0	0	1	0	0	Dreg #			
Force Interrupt / Reset RAISE uimm4	0x0090— 0x009F	0	0	0	0	0	0	0	0	1	0	0	1	uimm4			
Force Exception EXCPT uimm4	0x00A0— 0x00AF	0	0	0	0	0	0	0	0	1	0	1	0	uimm4			
Test and Set Byte (Atomic) NOTE: SP and FP are not allowed as the register for this instruction. Therefore, the highest valid Preg number is 5. TESTSET (Preg)	0x00B0— 0x00B5	0	0	0	0	0	0	0	0	1	0	1	1	Preg #			
No Op NOP	0x0000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table C-18. External Event Management Instructions (Sheet 2 of 2)

Instruction		Bin															
and Version	Opcode Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
No Op	0xC003 1800	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1
		0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
MNOP																	
No Op	0xC803 1800	1	1	0	0	1	0	0	0	0	0	0	0	0	0	1	1
		0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
MNOP when issued in parallel with two compatible load/store instructions																	
Abort	0x002F	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	1
NOTE: Abort is only valid on a simulator.																	
ABORT																	

Cache Control Instructions

Table C-19. Cache Control Instructions

		Bin															
Instruction and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Data Cache Prefetch PREFETCH [Preg]	0x0240— 0x0247	0	0	0	0	0	0	1	0	0	1	0	0	0	Preg #		
Data Cache Prefetch PREFETCH [Preg++]	0x0260— 0x0267	0	0	0	0	0	0	1	0	0	1	1	0	0	Preg #		
Data Cache Flush FLUSH [Preg]	0x0250— 0x0257	0	0	0	0	0	0	1	0	0	1	0	1	0	Preg #		
Data Cache Line Invalidate FLUSHINV [Preg]	0x0248— 0x024F	0	0	0	0	0	0	1	0	0	1	0	0	1	Preg #		
Instruction Cache Flush IFLUSH [Preg]	0x0258— 0x025F	0	0	0	0	0	0	1	0	0	1	0	1	1	Preg #		

Video Pixel Operations Instructions

Table C-20. Video Pixel Operations Instructions (Sheet 1 of 5)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Byte Align	0xC60D 0000— 0xC60D 0E3F	1	1	0	0	0	1	1	0	0	x	x	0	1	1	0	1
		0	0	0	0	Dest. Dreg #			x	x	x	Source 0 Dreg #			Source 1 Dreg #		
Dreg = ALIGN8 (Dreg, Dreg)																	
Byte Align	0xC60D 4000— 0xC60D 4E3F	1	1	0	0	0	1	1	0	0	x	x	0	1	1	0	1
		0	1	0	0	Dest. Dreg #			x	x	x	Source 0 Dreg #			Source 1 Dreg #		
Dreg = ALIGN16 (Dreg, Dreg)																	
Byte Align	0xC60D 800— 0xC60D 8E3F0	1	1	0	0	0	1	1	0	0	x	x	0	1	1	0	1
		1	0	0	0	Dest. Dreg #			x	x	x	Source 0 Dreg #			Source 1 Dreg #		
Dreg = ALIGN24 (Dreg, Dreg)																	
Disable Alignment Exception for Load	0xC412 C000	1	1	0	0	0	1	0	x	x	x	0	1	0	0	1	0
		1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
NOTE: 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.																	
DISALGNEXCPT																	
Dual 16-Bit Add / Clip	0xC417 0000— 0xC417 0E3F	1	1	0	0	0	1	0	x	x	x	0	1	0	1	1	1
		0	0	0	0	Dest. 0 Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = BYTEOP3P (Dreg_pair, Dreg_pair) (LO)																	
Dual 16-Bit Add / Clip	0xC437 0000— 0xC437 0E3F	1	1	0	0	0	1	0	x	x	x	1	1	0	1	1	1
		0	0	0	0	Dest. 0 Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = BYTEOP3P (Dreg_pair, Dreg_pair) (HI)																	

Table C-20. Video Pixel Operations Instructions (Sheet 2 of 5)

		Bin															
Instruction and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Dual 16-Bit Add / Clip	0xC417 2000— 0xC417 1E3F	1	1	0	0	0	1	0	x	x	x	0	1	0	1	1	1
		0	0	1	0	Dest. 0 Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = BYTEOP3P (Dreg_pair, Dreg_pair) (LO, R)																	
Dual 16-Bit Add / Clip	0xC437 2000— 0xC437 1E3F	1	1	0	0	0	1	0	x	x	x	1	1	0	1	1	1
		0	0	1	0	Dest. 0 Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
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.																	
Dreg = BYTEOP3P (Dreg_pair, Dreg_pair) (HI, R)																	
Dual 16-Bit Accumulator Extraction with Addition	0xC40C 403F— 0xC40C 4FC0	1	1	0	0	0	1	0	x	x	x	0	0	1	1	0	0
		0	1	0	0	Dest. of A1 Op Dreg #			Dest of A0 Op Dreg #			1	1	1	1	1	1
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.																	
Dreg = A1.L + A1.H, Dreg = A0.L + A0.H																	
Quad 8-Bit Add	0xC415 0000— 0xC415 0FFF	1	1	0	0	0	1	0	x	x	x	0	1	0	1	0	1
		0	0	0	0	Dest. 1 Dreg #			Dest. 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
(Dreg, Dreg) = BYTEOP16P (Dreg_pair, Dreg_pair)																	
Quad 8-Bit Add	0xC415 2000— 0xC415 2FFF	1	1	0	0	0	1	0	x	x	x	0	1	0	1	0	1
		0	0	1	0	Dest. 1 Dreg #			Dest. 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
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.																	
(Dreg, Dreg) = BYTEOP16P (Dreg_pair, Dreg_pair) (R)																	

Table C-20. Video Pixel Operations Instructions (Sheet 3 of 5)

									Bin								
Instruction and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Quad 8-Bit Average-Byte	0xC414 0000—	1	1	0	0	0	1	0	x	x	x	0	1	0	1	0	0
	0xC414 0E3F	0	0	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = BYTEOP1P (Dreg_pair, Dreg_pair)																	
Quad 8-Bit Average-Byte	0xC414 4000—	1	1	0	0	0	1	0	x	x	x	0	1	0	1	0	0
	0xC414 4E3F	0	1	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = BYTEOP1P (Dreg_pair, Dreg_pair) (T)																	
Quad 8-Bit Average-Byte	0xC414 200—	1	1	0	0	0	1	0	x	x	x	0	1	0	1	0	0
	0xC414 2E3F0	0	0	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = BYTEOP1P (Dreg_pair, Dreg_pair) (R)																	
Quad 8-Bit Average-Byte	0xC414 6000—	1	1	0	0	0	1	0	x	x	x	0	1	0	1	0	0
	0xC414 6E3F	0	1	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
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.																	
Dreg = BYTEOP1P (Dreg_pair, Dreg_pair) (T, R)																	
Quad 8-Bit Average-Half Word	0xC416 0000—	1	1	0	0	0	1	0	x	x	x	0	1	0	1	1	0
	0xC416 0E3F	0	0	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (RNDL)																	
Quad 8-Bit Average-Half Word	0xC436 0000—	1	1	0	0	0	1	0	x	x	x	1	1	0	1	1	0
	0xC436 0E3F	0	0	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (RNDH)																	
Quad 8-Bit Average-Half Word	0xC416 4000—	1	1	0	0	0	1	0	x	x	x	0	1	0	1	1	0
	0xC416 6E3F	0	1	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (TL)																	

Table C-20. Video Pixel Operations Instructions (Sheet 4 of 5)

		Bin															
Instruction and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Quad 8-Bit Average-Half Word	0xC436 4000— 0xC436 6E3F	1	1	0	0	0	1	0	x	x	x	1	1	0	1	1	0
		0	1	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (TH)																	
Quad 8-Bit Average-Half Word	0xC416 2000— 0xC416 2E3F	1	1	0	0	0	1	0	x	x	x	0	1	0	1	1	0
		0	0	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (RNDL, R)																	
Quad 8-Bit Average-Half Word	0xC436 2000— 0xC436 2E3F	1	1	0	0	0	1	0	x	x	x	1	1	0	1	1	0
		0	0	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (RNDH, R)																	
Quad 8-Bit Average-Half Word	0xC416 6000— 0xC416 7E3F	1	1	0	0	0	1	0	x	x	x	0	1	0	1	1	0
		0	1	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (TL, R)																	
Quad 8-Bit Average-Half Word	0xC436 6000— 0xC436 7E3F	1	1	0	0	0	1	0	x	x	x	1	1	0	1	1	0
		0	1	1	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
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.																	
Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (TH, R)																	
Quad 8-Bit Pack	0xC418 0000— 0xC418 0E3F	1	1	0	0	0	1	0	x	x	x	0	1	1	0	0	0
		0	0	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
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.																	
Dreg = BYTEPACK (Dreg, Dreg)																	

Table C-20. Video Pixel Operations Instructions (Sheet 5 of 5)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Quad 8-Bit Subtract	0xC415 4000—	1	1	0	0	0	1	0	x	x	x	0	1	0	1	0	1
	0xC415 4FFF	0	1	0	0	Dest. 1 Dreg #			Dest. 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
(Dreg, Dreg) = BYTEOP16M (Dreg_pair, Dreg_pair)																	
Quad 8-Bit Subtract	0xC415 6000—	1	1	0	0	0	1	0	x	x	x	0	1	0	1	0	1
	0xC415 6FFF	0	1	1	0	Dest. 1 Dreg #			Dest. 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
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.																	
(Dreg, Dreg) = BYTEOP16M (Dreg_pair, Dreg_pair) (R)																	
Quad 8-Bit Subtract-Absolute-Accumulate	0xC412 0000—	1	1	0	0	0	1	0	x	x	x	0	1	0	0	1	0
	0xC412 003F	0	0	0	0	0	0	0	0	0	0	Source 0 Dreg #			Source 1 Dreg #		
SAA (Dreg_pair, Dreg_pair)																	
Quad 8-Bit Subtract-Absolute-Accumulate	0xC412 2000—	1	1	0	0	0	1	0	x	x	x	0	1	0	0	1	0
	0xC412 203F	0	0	1	0	0	0	0	0	0	0	Source 0 Dreg #			Source 1 Dreg #		
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	0xC418 4000—	1	1	0	0	0	1	0	x	x	x	0	1	1	0	0	0
	0xC418 4FF8	0	1	0	0	Dest. 1 Dreg #			Dest. 0 Dreg #			Source 0 Dreg #			0	0	0
(Dreg, Dreg) = BYTEUNPACK Dreg_pair																	
Quad 8-Bit Unpack	0xC418 6000—	1	1	0	0	0	1	0	x	x	x	0	1	1	0	0	0
	0xC418 6FF8	0	1	1	0	Dest. 1 Dreg #			Dest. 0 Dreg #			Source 0 Dreg #			0	0	0
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)																	

Vector Operations Instructions

Table C-21. Vector Operations Instructions (Sheet 1 of 33)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Add on Sign	0xC40C 0000—	1	1	0	0	0	1	0	x	x	x	0	0	1	1	0	0
	0xC40C 0E38	0	0	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg_hi = Dreg_lo = SIGN (Dreg_hi) * Dreg_hi + SIGN (Dreg_lo) * Dreg_lo																	
Compare Select (VIT_MAX)	0xC609 C000—	1	1	0	0	0	1	1	0	0	x	x	0	1	0	0	1
	0xC609 CE07	1	1	0	0	Dest. Dreg #			x	x	x	Source 0 Dreg #			Source 1 Dreg #		
Dreg = VIT_MAX (Dreg, Dreg) (ASR)																	
Compare Select (VIT_MAX)	0xC609 8000—	1	1	0	0	0	1	1	0	0	x	x	0	1	0	0	1
	0xC609 8E07	1	0	0	0	Dest. Dreg #			x	x	x	Source 0 Dreg #			Source 1 Dreg #		
Dreg = VIT_MAX (Dreg, Dreg) (ASL)																	
Compare Select (VIT_MAX)	0xC609 4000—	1	1	0	0	0	1	1	0	0	x	x	0	1	0	0	1
	0xC609 4E07	0	1	0	0	Dest. Dreg #			x	x	x	0	0	0	Source Dreg #		
Dreg_lo = VIT_MAX (Dreg) (ASR)																	
Compare Select (VIT_MAX)	0xC609 0000—	1	1	0	0	0	1	1	0	0	x	x	0	1	0	0	1
	0xC609 0E07	0	0	0	0	Dest. Dreg #			x	x	x	0	0	0	Source Dreg #		
Dreg_lo = VIT_MAX (Dreg) (ASL)																	
Vector Absolute Value	0xC406 8000—	1	1	0	0	0	1	0	x	x	x	0	0	0	1	1	0
	0xC406 8E38	1	0	0	0	Dest. Dreg #			0	0	0	Source Dreg #			0	0	0
Dreg = ABS Dreg (V)																	
Vector Add / Subtract	0xC400 0000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	0
	0xC400 0E3F	0	0	0	0	Dest Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|+ Dreg																	

Table C-21. Vector Operations Instructions (Sheet 2 of 33)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Add / Subtract	0xC400 2000— 0xC400 2E3F	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	0
Dreg = Dreg +|+ Dreg (S)		0	0	1	0	Dest Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Vector Add / Subtract	0xC400 1000— 0xC400 1E3F	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	0
Dreg = Dreg +|+ Dreg (CO)		0	0	0	1	Dest Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Vector Add / Subtract	0xC400 3000— 0xC400 3E3F	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	0
Dreg = Dreg +|+ Dreg (SC0)		0	0	1	1	Dest Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Vector Add / Subtract	0xC400 8000— 0xC400 8E3F	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	0
Dreg = Dreg –|+ Dreg		1	0	0	0	Dest Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Vector Add / Subtract	0xC400 A000— 0xC400 AE3F	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	0
Dreg = Dreg –|+ Dreg (S)		1	0	1	0	Dest Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Vector Add / Subtract	0xC400 9000— 0xC400 9E3F	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	0
Dreg = Dreg –|+ Dreg (CO)		1	0	0	1	Dest Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Vector Add / Subtract	0xC400 B000— 0xC400 BE3F	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	0
Dreg = Dreg –|+ Dreg (SC0)		1	0	1	1	Dest Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		

Table C-21. Vector Operations Instructions (Sheet 3 of 33)

Instruction	Opcode	Bin															
and Version	Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Add / Subtract	0xC400 4000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	0
	0xC400 4E3F	0	1	0	0	Dest Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|– Dreg																	
Vector Add / Subtract	0xC400 6000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	0
	0xC400 6E3F	0	1	1	0	Dest Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|– Dreg (S)																	
Vector Add / Subtract	0xC400 5000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	0
	0xC400 5E3F	0	1	0	1	Dest Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|– Dreg (CO)																	
Vector Add / Subtract	0xC400 7000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	0
	0xC400 7E3F	0	1	1	1	Dest Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|– Dreg (SC0)																	
Vector Add / Subtract	0xC400 C000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	0
	0xC400 CE3F	1	1	0	0	Dest Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg –|– Dreg																	
Vector Add / Subtract	0xC400 E000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	0
	0xC400 EE3F	1	1	1	0	Dest Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg –|– Dreg (S)																	
Vector Add / Subtract	0xC400 D000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	0
	0xC400 DE3F	1	1	0	1	Dest Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg –|– Dreg (CO)																	

Table C-21. Vector Operations Instructions (Sheet 4 of 33)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Add / Subtract	0xC400 F000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	0
	0xC400 FE3F	1	1	1	1	Dest Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg –|– Dreg (SC0)																	
Vector Add / Subtract	0xC401 0000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	1
	0xC401 0FFF	0	0	0	0	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg																	
Vector Add / Subtract	0xC401 8000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	1
	0xC401 8FFF	1	0	0	0	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg (ASR)																	
Vector Add / Subtract	0xC401 C000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	1
	0xC401 CFFF	1	1	0	0	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg (ASL)																	
Vector Add / Subtract	0xC401 2000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	1
	0xC401 2FFF	0	0	1	0	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg (S)																	
Vector Add / Subtract	0xC401 A000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	1
	0xC401 AFFF	1	0	1	0	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg (S, ASR)																	
Vector Add / Subtract	0xC401 E000—	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	1
	0xC401 EFFF	1	1	1	0	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg (S, ASL)																	

Table C-21. Vector Operations Instructions (Sheet 5 of 33)

Instruction	Opcode	Bin															
and Version	Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Add / Subtract	0xC401 1000— 0xC401 1FFF	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	1
		0	0	0	1	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg (CO)																	
Vector Add / Subtract	0xC401 9000— 0xC401 9FFF	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	1
		1	0	0	1	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg (CO, ASR)																	
Vector Add / Subtract	0xC401 D000— 0xC401 DFFF	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	1
		1	1	0	1	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg (CO, ASL)																	
Vector Add / Subtract	0xC401 3000— 0xC401 3FFF	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	1
		0	0	1	1	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg (SCO)																	
Vector Add / Subtract	0xC401 B000— 0xC401 BFFF	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	1
		1	0	1	1	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg (SCO, ASR)																	
Vector Add / Subtract	0xC401 F000— 0xC401 FFFF	1	1	0	0	0	1	0	x	x	x	0	0	0	0	0	1
		1	1	1	1	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg (SCO, ASL)																	
Vector Add / Subtract	0xC421 0000— 0xC421 0FFF	1	1	0	0	0	1	0	x	x	x	1	0	0	0	0	1
		0	0	0	0	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg																	

Table C-21. Vector Operations Instructions (Sheet 6 of 33)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Add / Subtract	0xC421 8000— 0xC421 8FFF	1	1	0	0	0	1	0	x	x	x	1	0	0	0	0	1
		1	0	0	0	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg (ASR)																	
Vector Add / Subtract	0xC421 C000— 0xC421 CFFF	1	1	0	0	0	1	0	x	x	x	1	0	0	0	0	1
		1	1	0	0	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg (ASL)																	
Vector Add / Subtract	0xC421 2000— 0xC421 2FFF	1	1	0	0	0	1	0	x	x	x	1	0	0	0	0	1
		0	0	1	0	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg (S)																	
Vector Add / Subtract	0xC421 A000— 0xC421 AFFF	1	1	0	0	0	1	0	x	x	x	1	0	0	0	0	1
		1	0	1	0	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg (S, ASR)																	
Vector Add / Subtract	0xC421 E000— 0xC421 EFFF	1	1	0	0	0	1	0	x	x	x	1	0	0	0	0	1
		1	1	1	0	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg (S, ASL)																	
Vector Add / Subtract	0xC421 1000— 0xC421 1FFF	1	1	0	0	0	1	0	x	x	x	1	0	0	0	0	1
		0	0	0	1	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg (CO)																	
Vector Add / Subtract	0xC421 9000— 0xC421 9FFF	1	1	0	0	0	1	0	x	x	x	1	0	0	0	0	1
		1	0	0	1	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg (CO, ASR)																	

Table C-21. Vector Operations Instructions (Sheet 7 of 33)

Instruction	**Opcode**	**Bin**															
and Version	**Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Vector Add / Subtract	0xC421 D000—	1	1	0	0	0	1	0	x	x	x	1	0	0	0	0	1
	0xC421 DFFF	1	1	0	1	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg (CO, ASL)																	
Vector Add / Subtract	0xC421 3000—	1	1	0	0	0	1	0	x	x	x	1	0	0	0	0	1
	0xC421 3FFF	0	0	1	1	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg (SCO)																	
Vector Add / Subtract	0xC421 B000—	1	1	0	0	0	1	0	x	x	x	1	0	0	0	0	1
	0xC421 BFFF	1	0	1	1	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg (SCO, ASR)																	
Vector Add / Subtract	0xC421 F000—	1	1	0	0	0	1	0	x	x	x	1	0	0	0	0	1
	0xC421 FFFF	1	1	1	1	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg (SCO, ASL)																	
Vector Add / Subtract	0xC404 8000—	1	1	0	0	0	1	0	x	x	x	0	0	0	1	0	0
	0xC404 8FFF	1	0	0	0	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg + Dreg, Dreg = Dreg – Dreg																	
Vector Add / Subtract	0xC404 A000—	1	1	0	0	0	1	0	x	x	x	0	0	0	1	0	0
	0xC404 AFFF	1	0	1	0	Dest 1 Dreg #			Dest 0 Dreg #			Source 0 Dreg #			Source 1 Dreg #		
Dreg = Dreg + Dreg, Dreg = Dreg – Dreg (S)																	
Vector Add / Subtract	0xC411 003F—	1	1	0	0	0	1	0	x	x	x	0	1	0	0	0	1
	0xC411 0FC0	0	0	0	0	Dest 1 Dreg #			Dest 0 Dreg #			1	1	1	1	1	1
Dreg = A1 + A0, Dreg = A1 – A0																	

Table C-21. Vector Operations Instructions (Sheet 8 of 33)

Instruction	Opcode	Bin															
and Version	Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Add / Subtract	0xC411 203F—	1	1	0	0	0	1	0	x	x	x	0	1	0	0	0	1
	0xC411 2FC0	0	0	1	0	Dest 1 Dreg #			Dest 0 Dreg #			1	1	1	1	1	1
Dreg = A1 + A0, Dreg = A1 – A0 (S)																	
Vector Add / Subtract	0xC411 403F—	1	1	0	0	0	1	0	x	x	x	0	1	0	0	0	1
	0xC411 4FC0	0	1	0	0	Dest 1 Dreg #			Dest 0 Dreg #			1	1	1	1	1	1
Dreg = A0 + A1, Dreg = A0 – A1																	
Vector Add / Subtract	0xC411 603F—	1	1	0	0	0	1	0	x	x	x	0	1	0	0	0	1
	0xC411 6FC0	0	1	1	0	Dest 1 Dreg #			Dest 0 Dreg #			1	1	1	1	1	1
Dreg = A0 + A1, Dreg = A0 – A1 (S)																	
Vector Arithmetic Shift	0xC681 0100—	1	1	0	0	0	1	1	0	1	x	x	0	0	0	0	1
	0xC681 0FFF	0	0	0	0	Dest. Dreg #			2's complement of uimm5						Source Dreg #		
Dreg = Dreg >>> uimm5 (V)																	
Vector Arithmetic Shift	0xC681 4000—	1	1	0	0	0	1	1	0	1	x	x	0	0	0	0	1
	0xC681 4EFF	0	1	0	0	Dest. Dreg #			uimm5						Source Dreg #		
Dreg = Dreg << uimm5 (V, S)																	
Vector Arithmetic Shift	0xC601 0000—	1	1	0	0	0	1	1	0	0	x	x	0	0	0	0	1
	0xC601 0E3F	0	0	0	0	Dest. Dreg #			x	x	x	Source 0 Dreg #			Source 1 Dreg #		
Dreg = ASHIFT Dreg BY Dreg_lo (V)																	
Vector Arithmetic Shift	0xC601 4000—	1	1	0	0	0	1	1	0	0	x	x	0	0	0	0	1
	0xC601 4E3F	0	1	0	0	Dest. Dreg #			x	x	x	Source 0 Dreg #			Source 1 Dreg #		
Dreg = ASHIFT Dreg BY Dreg_lo (V, S)																	

Table C-21. Vector Operations Instructions (Sheet 9 of 33)

Instruction	Opcode	Bin															
and Version	Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Logical Shift	0xC681 8180— 0xC681 8FFF	1	1	0	0	0	1	1	0	1	x	x	0	0	0	0	1
		1	0	0	0	Dest. Dreg #			2's comp of uimm4						Source Dreg #		
Dreg = Dreg >> uimm4 (V)																	
Vector Logical Shift	0xC681 8000— 0xC681 8E7F	1	1	0	0	0	1	1	0	1	x	x	0	0	0	0	1
		1	0	0	0	Dest. Dreg #			uimm4						Source Dreg #		
Dreg = Dreg << uimm4 (V)																	
Vector Logical Shift	0xC601 8000— 0xC601 8E3F	1	1	0	0	0	1	1	0	0	x	x	0	0	0	0	1
		1	0	0	0	Dest. Dreg #			x	x	x	Source 0 Dreg #			Source 1 Dreg #		
Dreg = LSHIFT Dreg BY Dreg_lo (V)																	
Vector Maximum	0xC406 0000— 0xC406 0E3F	1	1	0	0	0	1	0	x	x	x	0	0	0	1	1	0
		0	0	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = MAX (Dreg, Dreg) (V)																	
Vector Minimum	0xC406 4000— 0xC406 4E3F	1	1	0	0	0	1	0	x	x	x	0	0	0	1	1	0
		0	1	0	0	Dest. Dreg #			0	0	0	Source 0 Dreg #			Source 1 Dreg #		
Dreg = MIN (Dreg, Dreg) (V)																	
Vector Multiply	0xC204 2000— 0xC204 E7FF	1	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi																	
Vector Multiply	0xC284 2000— 0xC284 E7FF	1	1	0	0	0	0	1	0	1	0	0	0	0	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (FU)																	

Table C-21. Vector Operations Instructions (Sheet 10 of 33)

Instruction	Opcode	Bin															
and Version	Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Multiply	0xC304 2000— 0xC304 E7FF	1	1	0	0	0	0	1	1	0	0	0	0	0	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IS)																	
Vector Multiply	0xC384 2000— 0xC384 E7FF	1	1	0	0	0	0	1	1	1	0	0	0	0	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IU)																	
Vector Multiply	0xC244 2000— 0xC244 E7FF	1	1	0	0	0	0	1	0	0	1	0	0	0	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (T)																	
Vector Multiply	0xC2C4 2000— 0xC2C4 E7FF	1	1	0	0	0	0	1	0	1	1	0	0	0	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (TFU)																	
Vector Multiply	0xC224 2000— 0xC224 E7FF	1	1	0	0	0	0	1	0	0	0	1	0	0	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (S2RND)																	
Vector Multiply	0xC324 2000— 0xC324 E7FF	1	1	0	0	0	0	1	1	0	0	1	0	0	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			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)

Instruction	Opcode	Bin															
and Version	Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Multiply	0xC364 2000— 0xC364 E7FF	1	1	0	0	0	0	1	1	0	1	1	0	0	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IH)																	
Vector Multiply	0xC214 2000— 0xC214 E7FF	1	1	0	0	0	0	1	0	0	0	0	1	0	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (M)																	
Vector Multiply	0xC294 2000— 0xC294 E7FF	1	1	0	0	0	0	1	0	1	0	0	1	0	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (FU, M)																	
Vector Multiply	0xC314 2000— 0xC314 E7FF	1	1	0	0	0	0	1	1	0	0	0	1	0	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IS, M)																	
Vector Multiply	0xC394 2000— 0xC394 E7FF	1	1	0	0	0	0	1	1	1	0	0	1	0	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IU, M)																	
Vector Multiply	0xC254 2000— 0xC254 E7FF	1	1	0	0	0	0	1	0	0	1	0	1	0	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (T, M)																	

Table C-21. Vector Operations Instructions (Sheet 12 of 33)

		Bin															
Instruction and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Vector Multiply	0xC2D4 2000— 0xC2D4 E7FF	1	1	0	0	0	0	1	0	1	1	0	1	0	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (TFU, M)																	
Vector Multiply	0xC234 2000— 0xC234 E7FF	1	1	0	0	0	0	1	0	0	0	1	1	0	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (S2RND, M)																	
Vector Multiply	0xC334 2000— 0xC334 E7FF	1	1	0	0	0	0	1	1	0	0	1	1	0	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (ISS2, M)																	
Vector Multiply	0xC374 2000— 0xC374 E7FF	1	1	0	0	0	0	1	1	0	1	1	1	0	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
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_lo = Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IH, M)																	
Vector Multiply	0xC20C 2000— 0xC20C E7FF	1	1	0	0	0	0	1	0	0	0	0	0	1	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = Dreg_lo_hi * Dreg_lo_hi , Dreg_odd = Dreg_lo_hi * Dreg_lo_hi																	

Table C-21. Vector Operations Instructions (Sheet 13 of 33)

Instruction	Opcode	Bin															
and Version	Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Multiply	0xC28C 2000— 0xC28C E7FF	1	1	0	0	0	0	1	0	1	0	0	0	1	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = Dreg_lo_hi * Dreg_lo_hi , Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (FU)																	
Vector Multiply	0xC30C 2000— 0xC30C E7FF	1	1	0	0	0	0	1	1	0	0	0	0	1	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = Dreg_lo_hi * Dreg_lo_hi , Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (IS)																	
Vector Multiply	0xC22C 2000— 0xC22C E7FF	1	1	0	0	0	0	1	0	0	0	1	0	1	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = Dreg_lo_hi * Dreg_lo_hi , Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (S2RND)																	
Vector Multiply	0xC32C 2000— 0xC32C E7FF	1	1	0	0	0	0	1	1	0	0	1	0	1	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = Dreg_lo_hi * Dreg_lo_hi , Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (ISS2)																	
Vector Multiply	0xC21C 2000— 0xC21C E7FF	1	1	0	0	0	0	1	0	0	0	0	1	1	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = Dreg_lo_hi * Dreg_lo_hi , Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (M)																	
Vector Multiply	0xC29C 2000— 0xC29C E7FF	1	1	0	0	0	0	1	0	1	0	0	1	1	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = Dreg_lo_hi * Dreg_lo_hi , Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (FU, M)																	

Table C-21. Vector Operations Instructions (Sheet 14 of 33)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Multiply	0xC31C 2000— 0xC31C E7FF	1	1	0	0	0	0	1	1	0	0	0	1	1	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = Dreg_lo_hi * Dreg_lo_hi , Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (IS, M)																	
Vector Multiply	0xC239 2000— 0xC239 E7FF	1	1	0	0	0	0	1	0	0	0	1	1	1	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = Dreg_lo_hi * Dreg_lo_hi , Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (S2RND, M)																	
Vector Multiply	0xC33C 2000— 0xC33C E7FF	1	1	0	0	0	0	1	1	0	0	1	1	1	1	0	0
		Dreg half 1		1	0	0	Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
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)																	

Table C-21. Vector Operations Instructions (Sheet 15 of 33)

Instruction	Opcode	Bin															
and Version	**Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Vector Multiply and Multiply-Accumulate																	
LEGEND: Dreg half 0 and Dreg half 1 determine which halves of the input operand registers to use. 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.		Dreg half 0 and Dreg half 1															
Dreg_lo * Dreg_lo		0	0														
Dreg_lo * Dreg_hi		0	1														
Dreg_hi * Dreg_lo		1	0														
Dreg_hi * Dreg_hi		1	1														
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																	
Vector Multiply and Multiply-Accumulate	0xC000 0000— 0xC003 DE3F	1	1	0	0	0	0	0	0	0	0	0	0	0	0	op1	
		Dreg half 1		0	op0		Dreg half 0		0	0	0	src_reg_0 Dreg #			src_reg_1 Dreg #		
A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi , A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi																	

Table C-21. Vector Operations Instructions (Sheet 16 of 33)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Multiply and Multiply-Accumulate	0xC080 0000— 0xC083 DE3F	1	1	0	0	0	0	0	0	1	0	0	0	0	0	op1	
		Dreg half 1		0	op0		Dreg half 0		0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi , A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi (FU)																	
Vector Multiply and Multiply-Accumulate	0xC100 0000— 0xC103 DE3F	1	1	0	0	0	0	0	1	0	0	0	0	0	0	op1	
		Dreg half 1		0	op0		Dreg half 0		0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi , A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi (IS)																	
Vector Multiply and Multiply-Accumulate	0xC060 0000— 0xC063 DE3F	1	1	0	0	0	0	0	0	0	1	1	0	0	0	op1	
		Dreg half 1		0	op0		Dreg half 0		0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi , A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi (W32)																	
Vector Multiply and Multiply-Accumulate	0xC010 0000— 0xC013 DE3F	1	1	0	0	0	0	0	0	0	0	0	1	0	0	op1	
		Dreg half 1		0	op0		Dreg half 0		0	0	0	src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi , A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi (M)																	
Vector Multiply and Multiply-Accumulate	0xC070 0000— 0xC073 DE3F	1	1	0	0	0	0	0	0	0	1	1	1	0	0	op1	
		Dreg half 1		0	op0		Dreg half 0		0	0	0	src_reg_ 0 Dreg #			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. A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi , A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi (W32, M)																	

Table C-21. Vector Operations Instructions (Sheet 17 of 33)

Instruction	Opcode	Bin															
and Version	Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Multiply and Multiply-Accumulate to Accumulator																	
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.		op0 and op1															
“=”		0	0														
“+=”		0	1														
“–=”		1	0														
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.		Dreg half 0 and Dreg half 1															
Dreg_lo * Dreg_lo		0	0														
Dreg_lo * Dreg_hi		0	1														
Dreg_hi * Dreg_lo		1	0														
Dreg_hi * Dreg_hi		1	1														
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 Half Register with Multiply and Multiply-Accumulate to Half Register																	

Table C-21. Vector Operations Instructions (Sheet 18 of 33)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Multiply and Multiply-Accumulate	0xC004 2000— 0xC007 FFFF	1	1	0	0	0	0	0	0	0	0	0	0	0	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi)																	
Vector Multiply and Multiply-Accumulate	0xC084 2000— 0xC087 FFFF	1	1	0	0	0	0	0	0	1	0	0	0	0	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (FU)																	
Vector Multiply and Multiply-Accumulate	0xC104 2000— 0xC107 FFFF	1	1	0	0	0	0	0	1	0	0	0	0	0	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (IS)																	
Vector Multiply and Multiply-Accumulate	0xC184 2000— 0xC187 FFFF	1	1	0	0	0	0	0	1	1	0	0	0	0	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (IU)																	
Vector Multiply and Multiply-Accumulate	0xC044 2000— 0xC047 FFFF	1	1	0	0	0	0	0	0	0	1	0	0	0	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (T)																	
Vector Multiply and Multiply-Accumulate	0xC0C4 2000— 0xC0C7 FFFF	1	1	0	0	0	0	0	0	1	1	0	0	0	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (TFU)																	

Table C-21. Vector Operations Instructions (Sheet 19 of 33)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Multiply and Multiply-Accumulate	0xC024 2000— 0xC027 FFFF	1	1	0	0	0	0	0	0	0	0	1	0	0	1	op1	
		Dreg half		0	1	1	0	0	Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (S2RND)																	
Vector Multiply and Multiply-Accumulate	0xC124 2000— 0xC127 FFFF	1	1	0	0	0	0	0	1	0	0	1	0	0	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (ISS2)																	
Vector Multiply and Multiply-Accumulate	0xC164 2000— 0xC167 FFFF	1	1	0	0	0	0	0	1	0	1	1	0	0	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (IH)																	
Vector Multiply and Multiply-Accumulate	0xC014 2000— 0xC017 FFFF	1	1	0	0	0	0	0	0	0	0	0	1	0	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (M)																	
Vector Multiply and Multiply-Accumulate	0xC094 2000— 0xC097 FFFF	1	1	0	0	0	0	0	0	1	0	0	1	0	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (FU, M)																	
Vector Multiply and Multiply-Accumulate	0xC114 2000— 0xC117 FFFF	1	1	0	0	0	0	0	1	0	0	0	1	0	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (IS, M)																	

Table C-21. Vector Operations Instructions (Sheet 20 of 33)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Multiply and Multiply-Accumulate	0xC194 2000— 0xC197 FFFF	1	1	0	0	0	0	0	1	1	0	0	1	0	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (IU, M)																	
Vector Multiply and Multiply-Accumulate	0xC054 2000— 0xC057 FFFF	1	1	0	0	0	0	0	0	0	1	0	1	0	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (T, M)																	
Vector Multiply and Multiply-Accumulate	0xC0D4 2000— 0xC0D7 FFFF	1	1	0	0	0	0	0	0	1	1	0	1	0	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (TFU, M)																	
Vector Multiply and Multiply-Accumulate	0xC034 2000— 0xC037 FFFF	1	1	0	0	0	0	0	0	0	0	1	1	0	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)																	
Vector Multiply and Multiply-Accumulate	0xC134 2000— 0xC137 FFFF	1	1	0	0	0	0	0	1	0	0	1	1	0	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)																	

Table C-21. Vector Operations Instructions (Sheet 21 of 33)

Instruction and Version	Opcode Range	Bin 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Multiply and Multiply-Accumulate	0xC174 2000— 0xC177 FFFF	1	1	0	0	0	0	0	1	0	1	1	1	0	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			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_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) ,
Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (IH, M)

Table C-21. Vector Operations Instructions (Sheet 22 of 33)

Instruction	Opcode	Bin															
and Version	**Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Vector Multiply and Multiply-Accumulate																	
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.		op0 and op1															
“=”		0	0														
“+=”		0	1														
“–=”		1	0														
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.		Dreg half 0 and Dreg half 1															
Dreg_lo * Dreg_lo		0	0														
Dreg_lo * Dreg_hi		0	1														
Dreg_hi * Dreg_lo		1	0														
Dreg_hi * Dreg_hi		1	1														

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 Data Register with Multiply and Multiply-Accumulate to Data Register

Table C-21. Vector Operations Instructions (Sheet 23 of 33)

Instruction	Opcode	Bin															
and Version	**Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Vector Multiply and Multiply-Accumulate	0xC00C 2000— 0xC00F FFFF	1	1	0	0	0	0	0	0	0	0	0	0	1	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , Dreg_odd = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi)																	
Vector Multiply and Multiply-Accumulate	0xC08C 2000— 0xC08F FFFF	1	1	0	0	0	0	0	0	1	0	0	0	1	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , Dreg_odd = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (FU)																	
Vector Multiply and Multiply-Accumulate	0xC10C 2000— 0xC10F FFFF	1	1	0	0	0	0	0	1	0	0	0	0	1	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			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)																	
Vector Multiply and Multiply-Accumulate	0xC02C 2000— 0xC02F FFFF	1	1	0	0	0	0	0	0	0	0	1	0	1	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			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)																	
Vector Multiply and Multiply-Accumulate	0xC12C 2000— 0xC12F FFFF	1	1	0	0	0	0	0	1	0	0	1	0	1	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , Dreg_odd = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (ISS2)																	
Vector Multiply and Multiply-Accumulate	0xC01C 2000— 0xC01F FFFF	1	1	0	0	0	0	0	0	0	0	0	1	1	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_even = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , Dreg_odd = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (M)																	

Table C-21. Vector Operations Instructions (Sheet 24 of 33)

		Bin															
Instruction and Version	**Opcode Range**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Vector Multiply and Multiply-Accumulate	0xC09C 2000— 0xC09F FFFF	1	1	0	0	0	0	0	0	1	0	0	1	1	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
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 Multiply-Accumulate	0xC11C 2000— 0xC11F FFFF	1	1	0	0	0	0	0	1	0	0	0	1	1	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			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 Multiply-Accumulate	0xC03C 2000— 0xC03F FFFF	1	1	0	0	0	0	0	0	0	0	1	1	1	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			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 Multiply-Accumulate	0xC13C 2000— 0xC13F FFFF	1	1	0	0	0	0	0	1	0	0	1	1	1	1	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			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)																	

Table C-21. Vector Operations Instructions (Sheet 25 of 33)

Instruction	Opcode								Bin								
and Version	Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Multiply and Multiply-Accumulate																	
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.		op0 and op1															
“=”		0	0														
“+=”		0	1														
“–=”		1	0														
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.		Dreg half 0 and Dreg half 1															
Dreg_lo * Dreg_lo		0	0														
Dreg_lo * Dreg_hi		0	1														
Dreg_hi * Dreg_lo		1	0														
Dreg_hi * Dreg_hi		1	1														
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 Half Register																	

Table C-21. Vector Operations Instructions (Sheet 26 of 33)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Multiply and Multiply-Accumulate	0xC004 0000— 0xC007 DFFF	1	1	0	0	0	0	0	0	0	0	0	0	0	1	op1	
		Dreg half 1		0	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi)																	
Vector Multiply and Multiply-Accumulate	0xC084 0000— 0xC087 DFFF	1	1	0	0	0	0	0	0	1	0	0	0	0	1	op1	
		Dreg half 1		0	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (FU)																	
Vector Multiply and Multiply-Accumulate	0xC104 0000— 0xC107 DFFF	1	1	0	0	0	0	0	1	0	0	0	0	0	1	op1	
		Dreg half 1		0	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (IS)																	
Vector Multiply and Multiply-Accumulate	0xC000 2000— 0xC003 FFFF	1	1	0	0	0	0	0	0	0	0	0	0	0	0	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi																	
Vector Multiply and Multiply-Accumulate	0xC080 2000— 0xC083 FFFF	1	1	0	0	0	0	0	0	1	0	0	0	0	0	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			src_reg_ 1 Dreg #		
Dreg_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi (FU)																	

Table C-21. Vector Operations Instructions (Sheet 27 of 33)

Instruction	Opcode	Bin															
and Version	Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Multiply and Multiply-Accumulate	0xC100 2000—0xC103 FFFF	1	1	0	0	0	0	0	1	0	0	0	0	0	0	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			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_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) ,
A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi (IS)

Table C-21. Vector Operations Instructions (Sheet 28 of 33)

Instruction	Opcode	Bin															
and Version	Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Multiply and Multiply-Accumulate																	
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.		op0 and op1															
“=”		0	0														
“+=”		0	1														
“–=”		1	0														
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.		Dreg half 0 and Dreg half 1															
Dreg_lo * Dreg_lo		0	0														
Dreg_lo * Dreg_hi		0	1														
Dreg_hi * Dreg_lo		1	0														
Dreg_hi * Dreg_hi		1	1														
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																	

Table C-21. Vector Operations Instructions (Sheet 29 of 33)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Multiply and Multiply-Accumulate	0xC00C 0000— 0xC00F DFFF	1	1	0	0	0	0	0	0	0	0	0	0	1	1	op1	
		Dreg half 1		0	op0		Dreg half 0		Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi , Dreg_odd = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi)																	
Vector Multiply and Multiply-Accumulate	0xC08C 0000— 0xC08F DFFF	1	1	0	0	0	0	0	0	1	0	0	0	1	1	op1	
		Dreg half 1		0	op0		Dreg half 0		Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi , Dreg_odd = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (FU)																	
Vector Multiply and Multiply-Accumulate	0xC10C 0000— 0xC10F DFFF	1	1	0	0	0	0	0	1	0	0	0	0	1	1	op1	
		Dreg half 1		0	op0		Dreg half 0		Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi , Dreg_odd = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (IS)																	
Vector Multiply and Multiply-Accumulate	0xC008 2000— 0xC00B FFFF	1	1	0	0	0	0	0	0	0	0	0	0	1	0	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_even = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi																	
Vector Multiply and Multiply-Accumulate	0xC088 2000— 0xC08B FFFF	1	1	0	0	0	0	0	0	1	0	0	0	1	0	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_0 Dreg #			src_reg_1 Dreg #		
Dreg_even = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi (FU)																	

Table C-21. Vector Operations Instructions (Sheet 30 of 33)

Instruction and Version	Opcode Range	Bin															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Multiply and Multiply-Accumulate	0xC108 2000— 0xC10B FFFF	1	1	0	0	0	0	0	1	0	0	0	0	1	0	op1	
		Dreg half 1		1	op0		Dreg half 0		Dest. Dreg #			src_reg_ 0 Dreg #			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) ,
A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi (IS)

Table C-21. Vector Operations Instructions (Sheet 31 of 33)

Instruction and Version	Opcode Range	Bin 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Multiply and Multiply-Accumulate																	
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.		op0 and op1															
“=”		0	0														
“+=”		0	1														
“–=”		1	0														
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.		Dreg half 0 and Dreg half 1															
Dreg_lo * Dreg_lo		0	0														
Dreg_lo * Dreg_hi		0	1														
Dreg_hi * Dreg_lo		1	0														
Dreg_hi * Dreg_hi		1	1														
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																	
Multiply and Multiply-Accumulate to Accumulator with Multiply and Multiply-Accumulate to Data Register																	

Table C-21. Vector Operations Instructions (Sheet 32 of 33)

Instruction and Version	Opcode Range	Bin 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Negate (Two's-Complement)	0xC40F C000— 0xC40F CE38	1	1	0	0	0	1	0	x	x	x	0	0	1	1	1	1
		1	1	0	0	Dest. Dreg #			0	0	0	Source Dreg #			0	0	0
Dreg = – Dreg (V)																	
Vector Pack	0xC604 0000— 0xC604 0E3F	1	1	0	0	0	1	1	0	0	x	x	0	0	1	0	0
		0	0	0	0	Dest. Dreg #			x	x	x	Source 0 Dreg #			Source 1 Dreg #		
Dreg = PACK (Dreg_lo, Dreg_lo)																	
Vector Pack	0xC604 4000— 0xC604 4E3F	1	1	0	0	0	1	1	0	0	x	x	0	0	1	0	0
		0	1	0	0	Dest. Dreg #			x	x	x	Source 0 Dreg #			Source 1 Dreg #		
Dreg = PACK (Dreg_lo, Dreg_hi)																	
Vector Pack	0xC604 8000— 0xC604 8E3F	1	1	0	0	0	1	1	0	0	x	x	0	0	1	0	0
		1	0	0	0	Dest. Dreg #			x	x	x	Source 0 Dreg #			Source 1 Dreg #		
Dreg = PACK (Dreg_hi, Dreg_lo)																	
Vector Pack	0xC604 C000— 0xC604 CE3F	1	1	0	0	0	1	1	0	0	x	x	0	0	1	0	0
		1	1	0	0	Dest. Dreg #			x	x	x	Source 0 Dreg #			Source 1 Dreg #		
Dreg = PACK (Dreg_hi, Dreg_hi)																	
Vector Search	0xC40D 0000— 0xC40D 2FFF	1	1	0	0	0	1	0	x	x	x	0	0	1	1	0	1
		0	0	0	0	Dest. 1 Dreg #			Dest. 0 Dreg #			Source Dreg #			0	0	0
(Dreg, Dreg) = SEARCH Dreg (GT)																	
Vector Search	0xC40D 4000— 0xC40D 6FFF	1	1	0	0	0	1	0	x	x	x	0	0	1	1	0	1
		0	1	0	0	Dest. 1 Dreg #			Dest. 0 Dreg #			Source Dreg #			0	0	0
(Dreg, Dreg) = SEARCH Dreg (GE)																	

Table C-21. Vector Operations Instructions (Sheet 33 of 33)

Instruction	Opcode								Bin								
and Version	Range	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vector Search	0xC40D 8000— 0xC40D AFF8	1	1	0	0	0	1	0	x	x	x	0	0	1	1	0	1
		1	0	0	0	Dest. 1 Dreg #			Dest. 0 Dreg #			Source Dreg #			0	0	0
(Dreg, Dreg) = SEARCH Dreg (LT)																	
Vector Search		1	1	0	0	0	1	0	x	x	x	0	0	1	1	0	1
	0xC40D C000— 0xC40D EFF8	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 C-22. 16-Bit Opcode Instructions (Sheet 1 of 14)

Instruction and Version	Opcode Range
No Op NOP	0x0000—
Return RTS	0x0010—
Return RTI	0x0011—
Return RTX	0x0012—
Return RTN	0x0013—
Return RTE	0x0014—
Idle IDLE	0x0020—
Core Synchronize CSYNC	0x0023—
System Synchronize SSYNC	0x0024—
Force Emulation EMUEXCPT	0x0025—
Abort ABORT	0x002F—
Disable Interrupts CLI Dreg	0x0030— 0x0037
Enable Interrupts STI Dreg	0x0040 — 0x0047
Jump JUMP (Preg)	0x0050— 0x0057

Table C-22. 16-Bit Opcode Instructions (Sheet 2 of 14)

Instruction and Version	Opcode Range
Call CALL (Preg)	0x0060— 0x0067
Call CALL (PC+Preg)	0x0070— 0x0077
Jump JUMP (PC+Preg)	0x0080— 0x0087
Force Interrupts / Reset RAISE uimm4	0x0090— 0x009F
Force Exception EXCPT uimm4	0x00A0— 0x00AF
Test and Set Byte (Atomic) TESTSET (Preg)	0x00B0— 0x00B5
Pop mostreg=[SP++]	0x0100— 0x013F
Push [– –SP]=allreg	0x0140— 0x017F
Move CC Dreg = CC	0x0200— 0x0207
Move CC CC = Dreg	0x0208— 0x020F
Negate CC CC = !CC	0x0218—
Data Cache Prefetch PREFETCH [Preg]	0x0240— 0x0247
Data Cache Line Invalidate FLUSHINV [Preg]	0x0248— 0x024F
Data Cache Flush FLUSH [Preg]	0x0250— 0x0257
Instruction Cache Flush IFLUSH [Preg]	0x0258— 0x025F
Data Cache Prefetch PREFETCH [Preg++]	0x0260— 0x0267

Table C-22. 16-Bit Opcode Instructions (Sheet 3 of 14)

Instruction and Version	Opcode Range
Data Cache Line Invalidate FLUSHINV [Preg++]	0x0268— 0x026F
Data Cache Flush FLUSH [Preg++]	0x0270— 0x0277
Instruction Cache Flush IFLUSH [Preg++]	0x0278— 0x027F
Move CC CC = statbit	0x0300— 0x031F
Move CC CC |= statbit	0x0320— 0x033F
Move CC CC &= statbit	0x0340— 0x035F
Move CC CC ^= statbit	0x0360— 0x037F
Move CC statbit = CC	0x0380— 0x039F
Move CC statbit |= CC	0x03A0— 0x03BF
Move CC statbit &= CC	0x03C0— 0x03DF
Move CC statbit ^= CC	0x03E0— 0x03FF
Pop Multiple (P5:Preglim)=[SP++]	0x0480— 0x0485
Push Multiple [– –SP]=(P5:Preglim)	0x04C0— 0x04C5
Pop Multiple (R7:Dreglim)=[SP++]	0x0500— 0x0538
Push Multiple [– –SP]=(R7:Dreglim)	0x0540— 0x0578
Pop Multiple (R7:Dreglim, P5:Preglim)=[SP++]	0x0580— 0x05BD

Table C-22. 16-Bit Opcode Instructions (Sheet 4 of 14)

Instruction and Version	Opcode Range
Push Multiple [– –SP]=(R7:Dreglim, P5:Preglim)	0x05C0— 0x05FD
Move Conditional IF !CC Dreg=Dreg	0x0600— 0x063F
Move Conditional IF !CC Dreg=Preg	0x0640— 0x067F
Move Conditional IF !CC Preg=Dreg	0x0680— 0x06BF
Move Conditional IF !CC Preg=Preg	0x06C0— 0x06FF
Move Conditional IF CC Dreg=Dreg	0x0700— 0x073F
Move Conditional IF CC Dreg=Preg	0x0740— 0x077F
Move Conditional IF CC Preg=Dreg	0x0780— 0x07BF
Move Conditional IF CC Preg=Preg	0x07C0— 0x07FF
Compare Data Register CC = Dreg == Dreg	0x0800— 0x083F
Compare Pointer Register CC = Preg == Preg	0x0840— 0x087F
Compare Data Register CC = Dreg < Dreg	0x0880— 0x08BF
Compare Pointer Register CC = Preg < Preg	0x08C0— 0x08FF
Compare Data Register CC = Dreg <= Dreg	0x0900— 0x093F
Compare Pointer Register CC = Preg <= Preg	0x0940— 0x097F
Compare Data Register CC = Dreg < Dreg (IU)	0x0980— 0x09BF

Table C-22. 16-Bit Opcode Instructions (Sheet 5 of 14)

Instruction **and Version**	**Opcode** **Range**
Compare Pointer Register CC = Preg < Preg (IU)	0x09C0— 0x09FF
Compare Data Register CC = Dreg <= Dreg (IU)	0x0A00— 0x0A3F
Compare Pointer Register CC = Preg <= Preg (IU)	0x0A40— 0x0A7F
Compare Accumulator CC = A0 == A1	0x0A80—
Compare Accumulator CC = A0 < A1	0x0B00—
Compare Accumulator CC = A0 <= A1	0x0B80—
Compare Data Register CC = Dreg == imm3	0x0C00— 0x0C3F
Compare Pointer Register CC = Preg == imm3	0x0C40— 0x0C7F
Compare Data Register CC = Dreg < imm3	0x0C80— 0x0CBF
Compare Pointer Register CC = Preg < imm3	0x0CC0— 0x0CFF
Compare Data Register CC = Dreg <= imm3	0x0D00— 0x0D3F
Compare Pointer Register CC = Preg <= imm3	0x0D40— 0x0D7F
Compare Data Register CC = Dreg < uimm3 (IU)	0x0D80— 0x0DBF
Compare Pointer Register CC = Preg < uimm3 (IU)	0x0DC0— 0x0DFF
Compare Data Register CC = Dreg <= uimm3 (IU)	0x0E00— 0x0E3F
Compare Pointer Register CC = Preg <= uimm3 (IU)	0x0E40— 0x0E7F

Table C-22. 16-Bit Opcode Instructions (Sheet 6 of 14)

Instruction and Version	Opcode Range
Conditional Jump IF !CC JUMP pcrel11m2	0x1000— 0x13FF
Conditional Jump IF CC JUMP pcrel11m2	0x1800— 0x17FF
Conditional Jump IF !CC JUMP pcrel11m2 (bp)	0x1400— 0x1BFF
Conditional Jump IF CC JUMP pcrel11m2 (bp)	0x1C00— 0x1FFF
Jump JUMP.S pcrel13m2	0x2000— 0x2FFF

Move Register
genreg = genreg
genreg = dagreg
dagreg = genreg
dagreg = dagreg
genreg = USP
USP = genreg
Dreg = sysreg
sysreg = Dreg
sysreg = Preg
sysreg = USP
	0x3000—
	0x3FFF

Arithmetic Shift Dreg >>>= Dreg	0x4000— 0x403F
Logical Shift Dreg >>= Dreg	0x4040— 0x407F
Logical Shift Dreg <<= Dreg	0x4080— 0x40BF
Multiply 32-Bit Operands Dreg *= Dreg	0x40C0— 0x40FF
Add with Shift Dreg = (Dreg + Dreg) << 1	0x4100— 0x413F
Add with Shift Dreg = (Dreg + Dreg) << 2	0x4140— 0x417F
Divide Primitive DIVQ (Dreg, Dreg)	0x4200— 0x423F

Table C-22. 16-Bit Opcode Instructions (Sheet 7 of 14)

Instruction and Version	**Opcode Range**
Divide Primitive DIVS (Dreg, Dreg)	0x4240— 0x427F
Move Half to Full Word, Sign Extended Dreg = Dreg_lo (X)	0x4280— 0x42BF
Move Half to Full Word – Zero Extended Dreg = Dreg_lo (Z)	0x42C0— 0x42FF
Move Byte, Sign Extended Dreg = Dreg_byte (X)	0x4300— 0x433F
Move Byte, Zero Extended Dreg = Dreg_byte (Z)	0x4340— 0x437F
Negate (Two's-Complement) Dreg = – Dreg	0x4380— 0x43BF
NOT (One's-Complement) Dreg = ~ Dreg	0x43C0— 0x43FF
Modify-Decrement Preg –= Preg	0x4400— 0x443F
Logical Shift Preg = Preg << 2	0x4440— 0x447F
Logical Shift Preg = Preg >> 2	0x44C0— 0x44FF
Logical Shift Preg = Preg >> 1	0x4500— 0x453F
Modify-Increment Preg += Preg (BREV)	0x4540— 0x457F
Add with Shift Preg = (Preg + Preg) << 1	0x4580— 0x45BF
Add with Shift Preg = (Preg + Preg) << 2	0x45C0— 0x45FF
Bit Test CC = ! BITTST (Dreg, uimm5)	0x4800— 0x48FF
Bit Test CC = BITTST (Dreg, uimm5)	0x4900— 0x49FF

Table C-22. 16-Bit Opcode Instructions (Sheet 8 of 14)

Instruction and Version	**Opcode Range**
Bit Set BITSET (Dreg, uimm5)	0x4A00— 0x4AFF
Bit Toggle BITTGL (Dreg, uimm5)	0x4B00— 0x4BFF
Bit Clear BITCLR (Dreg, uimm5)	0x4C00— 0x4CFF
Arithmetic Shift Dreg >>>= uimm5	0x4D00— 0x4DFF
Logical Shift Dreg >>= uimm5	0x4E00— 0x4EFF
Logical Shift Dreg <<= uimm5	0x4F00— 0x4FFF
Add Dreg = Dreg + Dreg	0x5000— 0x51FF
Subtract Dreg = Dreg – Dreg	0x5200— 0x53FF
AND Dreg = Dreg & Dreg	0x5400— 0x55FF
OR Dreg = Dreg | Dreg	0x5600— 0x57FF
Exclusive-OR Dreg = Dreg ^ Dreg	0x5800— 0x59FF
Add Preg = Preg + Preg	0x5A00— 0x5BFF
Logical Shift Preg = Preg << 1	0x5A00— 0x5BFF
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.	
Shift with Add Preg = Preg + (Preg <<1)	0x5C00— 0x5DFF
Shift with Add Preg = Preg + (Preg <<2)	0x5E00— 0x5FFF

Table C-22. 16-Bit Opcode Instructions (Sheet 9 of 14)

Instruction and Version	Opcode Range
Load Immediate Dreg = imm7 (X)	0x6000— 0x63FF
Add Immediate Dreg += imm7	0x6400— 0x6700
Load Immediate Preg = imm7 (X)	0x6800— 0x6BFF
Add Immediate Preg += imm7	0x6C00— 0x6FFF—
Load Data Register Dreg = [Preg ++ Preg]	0x8000— 0x81FF—
Load Low Data Register Half Dreg_lo = W [Preg]	0x8200— 0x83FF—
Load Low Data Register Half Dreg_lo = W [Preg ++ Preg]	0x8201— 0x83FE—
Load High Data Register Half Dreg_hi = W [Preg]	0x8400— 0x85FF—
Load High Data Register Half Dreg_hi = W [Preg ++ Preg]	0x8401— 0x85FE—
Load Half Word, Zero Extended Dreg = W [Preg ++ Preg] (Z)	0x8601— 0x87FE—
Load Half Word, Sign Extended Dreg = W [Preg ++ Preg] (X)	0x8E00— 0x8FFF—
Store Data Register [Preg ++ Preg] = Dreg	0x8800— 0x89FF
Store Low Data Register Half W [Preg] = Dreg_lo	0x8A00— 0x8BFF
Store Low Data Register Half W [Preg ++ Preg] = Dreg_lo	0x8A01— 0x8BFE
Store High Data Register Half W [Preg] = Dreg_hi	0x8C00— 0x8DFF
Store High Data Register Half W [Preg ++ Preg] = Dreg_hi	0x8C01— 0x8DFE

Table C-22. 16-Bit Opcode Instructions (Sheet 10 of 14)

Instruction and Version	**Opcode Range**
Load Data Register Dreg = [Preg ++]	0x9000— 0x903F
Load Pointer Register Preg = [Preg ++]	0x9040— 0x907F
Load Data Register Dreg = [Preg – –]	0x9080— 0x90BF
Load Pointer Register Preg = [Preg – –]	0x90C0— 0x90FF
Load Data Register Dreg = [Preg]	0x9100— 0x913F
Load Pointer Register Preg = [Preg]	0x9140— 0x917F
Store Data Register [Preg ++] = Dreg	0x9200— 0x923F
Store Pointer Register [Preg ++] = Preg	0x9240— 0x927F
Store Data Register [Preg – –] = Dreg	0x9280— 0x92BF
Store Pointer Register [Preg – –] = Preg	0x92C0— 0x92FF
Store Data Register [Preg] = Dreg	0x9300— 0x933F
Store Pointer Register [Preg] = Preg	0x9340— 0x937F
Load Half Word, Zero Extended Dreg = W [Preg ++] (Z)	0x9400— 0x943F
Load Half Word, Sign Extended Dreg = W [Preg ++] (X)	0x9440— 0x947F
Load Half Word, Zero Extended Dreg = W [Preg – –] (Z)	0x9480— 0x94BF
Load Half Word, Sign Extended Dreg = W [Preg – –] (X)	0x94C0— 0x94FF

Table C-22. 16-Bit Opcode Instructions (Sheet 11 of 14)

Instruction and Version	Opcode Range
Load Half Word, Zero Extended Dreg = W [Preg] (Z)	0x9500— 0x953F
Load Half Word, Sign Extended Dreg = W [Preg] (X)	0x9540— 0x957F
Store Low Data Register Half W [Preg ++] = Dreg	0x9600— 0x963F
Store Low Data Register Half W [Preg – –] = Dreg	0x9680— 0x96BF
Store Low Data Register Half W [Preg] = Dreg	0x9700— 0x973F
Load Byte, Zero Extended Dreg = B [Preg ++] (Z)	0x9800— 0x983F
Load Byte, Sign Extended Dreg = B [Preg ++] (X)	0x9840— 0x987F
Load Byte, Zero Extended Dreg = B [Preg – –] (Z)	0x9880— 0x98BF
Load Byte, Sign Extended Dreg = B [Preg – –] (X)	0x98C0— 0x98FF
Load Byte, Zero Extended Dreg = B [Preg] (Z)	0x9900— 0x993F
Load Byte, Sign Extended Dreg = B [Preg] (X)	0x9940— 0x997F
Store Byte B [Preg ++] = Dreg	0x9A00— 0x9A3F
Store Byte B [Preg – –] = Dreg	0x9A80— 0x9ABF
Store Byte B [Preg] = Dreg	0x9B00— 0x9B3F
Load Data Register Dreg = [Ireg ++]	0x9C00— 0x9C1F
Load Low Data Register Half Dreg_lo = W [Ireg ++]	0x9C20— 0x9C3F

Table C-22. 16-Bit Opcode Instructions (Sheet 12 of 14)

Instruction and Version	Opcode Range
Load High Data Register Half Dreg_hi = W [Ireg ++]	0x9C40— 0x9C5F
Load Data Register Dreg = [Ireg – –]	0x9C80— 0x9C9F
Load Low Data Register Half Dreg_lo = W [Ireg – –]	0x9CA0— 0x9CBF
Load High Data Register Half Dreg_hi = W [Ireg – –]	0x9CC0— 0x9CDF
Load Data Register Dreg = [Ireg]	0x9D00— 0x9D1F
Load Low Data Register Half Dreg_lo = W [Ireg]	0x9D20— 0x9D3F
Load High Data Register Half Dreg_hi = W [Ireg]	0x9D40— 0x9D5F
Load Data Register Dreg = [Ireg ++ Mreg]	0x9D80— 0x9DFF
Store Data Register [Ireg ++] = Dreg	0x9E00— 0x9E1F
Store Low Data Register Half W [Ireg ++] = Dreg_lo	0x9E20— 0x9E3F
Store High Data Register Half W [Ireg ++] = Dreg_hi	0x9E40— 0x9E5F
Modify-Increment Ireg += Mreg	0x9E60— 0x9E6F
Modify-Decrement Ireg –= Mreg	0x9E70— 0x9E7F
Store Data Register [Ireg – –] = Dreg	0x9E80— 0x9E9F
Store Low Data Register Half W [Ireg – –] = Dreg_lo	0x9EA0— 0x9EBF
Store High Data Register Half W [Ireg – –] = Dreg_hi	0x9EC0— 0x9EDF

Table C-22. 16-Bit Opcode Instructions (Sheet 13 of 14)

Instruction and Version	Opcode Range
Modify-Increment Ireg += Mreg (brev)	0x9EE0— 0x9EEF
Store Data Register [Ireg] = Dreg	0x9F00— 0x9F1F
Store Low Data Register Half W [Ireg] = Dreg_lo	0x9F20— 0x9F3F
Store High Data Register Half W [Ireg] = Dreg_hi	0x9F40— 0x9F5F
Add Immediate Ireg += 2	0x9F60— 0x9F63
Subtract Immediate Ireg –= 2	0x9F64— 0x9F67
Add Immediate Ireg += 4	0x9F68— 0x9F6B
Subtract Immediate Ireg –= 4	0x9F6C— 0x9F6F
Store Data Register [Ireg ++ Mreg] = Dreg	0x9F80— 0x9FFF
Load Data Register Dreg = [Preg + uimm6m4]	0xA000— 0xA3FF
Load Half Word, Zero Extended Dreg = W [Preg + uimm5m2] (Z)	0xA400— 0xA7FF
Load Half Word, Sign Extended Dreg = W [Preg + uimm5m2] (X)	0xA800— 0xABFF
Load Pointer Register Preg = [Preg + uimm6m4]	0xAC00— 0xAFFF
Store Data Register [Preg + uimm6m4] = Dreg	0xB000— 0xB3FF

Table C-22. 16-Bit Opcode Instructions (Sheet 14 of 14)

Instruction and Version	**Opcode Range**
Store Low Data Register Half W [Preg + uimm5m2] = Dreg	0xB400— 0xB7FF
Load Data Register Dreg = [FP – uimm7m4]	0xB800— 0xB9F7
Load Pointer Register Preg = [FP – uimm7m4]	0xB808— 0xB9FF
Store Data Register [FP – uimm7m4] = Dreg	0xBA00— 0xBBF7
Store Pointer Register [FP – uimm7m4] = Preg	0xBA08— 0xBBFF
Store Pointer Register [Preg + uimm6m4] = Preg	0xBC00— 0xBFFF

32-Bit Opcode Instructions

Table C-23 lists the instructions that are represented by 32-bit opcodes.

Table C-23. 32-Bit Opcode Instructions (Sheet 1 of 40)

Instruction and Version	Opcode Range
Vector Multiply and Multiply-Accumulate A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi , A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi	0xC000 0000— 0xC003 DE3F
Multiply and Multiply-Accumulate to Accumulator A1 = Dreg_lo_hi * Dreg_lo_hi	0xC000 1800— 0xC000 D83F
Vector Multiply and Multiply-Accumulate Dreg_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi	0xC000 2000— 0xC003 FFFF
Multiply and Multiply-Accumulate to Accumulator A1 += Dreg_lo_hi * Dreg_lo_hi	0xC001 1800— 0xC001 D83F
Multiply and Multiply-Accumulate to Accumulator A1 –= Dreg_lo_hi * Dreg_lo_hi	0xC002 1800— 0xC002 D83F
Multiply and Multiply-Accumulate to Accumulator A0 = Dreg_lo_hi * Dreg_lo_hi	0xC003 0000— 0xC003 063F
Multiply and Multiply-Accumulate to Accumulator A0 += Dreg_lo_hi * Dreg_lo_hi	0xC003 0800— 0xC003 0E3F
Multiply and Multiply-Accumulate to Accumulator A0 –= Dreg_lo_hi * Dreg_lo_hi	0xC003 1000— 0xC003 163F
No Op MNOP	0xC003 1800
Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi)	0xC003 2000— 0xC003 27FF
Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi)	0xC003 2800— 0xC003 0FFF
Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 –= Dreg_lo_hi * Dreg_lo_hi)	0xC003 3000— 0xC003 37FF
Move Register Half Dreg_lo = A0	0xC003 3800— 0xC003 39C0

Table C-23. 32-Bit Opcode Instructions (Sheet 2 of 40)

Instruction and Version	Opcode Range
Vector Multiply and Multiply-Accumulate A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi)	0xC004 0000— 0xC007 DFFF
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi)	0xC004 1800— 0xC004 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)
	0xC004 2000—
	0xC007 FFFF

Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi)	0xC005 1800— 0xC005 D9FF
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 –= Dreg_lo_hi * Dreg_lo_hi)	0xC006 1800— 0xC006 D9FF
Move Register Half Dreg_hi = A1	0xC007 1800— 0xC007 19C0

Move Register Half
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

Table C-23. 32-Bit Opcode Instructions (Sheet 3 of 40)

Instruction and Version	Opcode Range
Multiply and Multiply-Accumulate to Data Register Dreg_even = (A0 += Dreg_lo_hi * Dreg_lo_hi)	0xC00D 0800— 0xC00D 0FFF
Multiply and Multiply-Accumulate to Data Register Dreg_even = (A0 –= Dreg_lo_hi * Dreg_lo_hi)	0xC00D 1000— 0xC00D 17FF
Vector Multiply and Multiply-Accumulate A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi , A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi (M)	0xC010 0000— 0xC013 DE3F
Multiply and Multiply-Accumulate to Accumulator A1 = Dreg_lo_hi * Dreg_lo_hi (M)	0xC010 1800— 0xC010 D83F
Multiply and Multiply-Accumulate to Accumulator A1 += Dreg_lo_hi * Dreg_lo_hi (M)	0xC011 1800— 0xC011 D83F
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (M)	0xC014 1800— 0xC014 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) (M)
	0xC014 2000—
	0xC017 FFFF

Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (M)	0xC015 1800— 0xC015 D9FF
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (M)	0xC016 1800— 0xC016 D9FF
Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (M)	0xC018 1800— 0xC018 D9FF
Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (M)	0xC019 1800— 0xC019 D9FF
Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (M)	0xC01A 1800— 0xC01A D9FF

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) (M)
	0xC01C 2000—
	0xC01F FFFF

Multiply and Multiply-Accumulate to Accumulator A1 –= Dreg_lo_hi * Dreg_lo_hi (M)	0xC022 1800— 0xC022 D83F
Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (S2RND)	0xC023 2000— 0xC023 27FF

Table C-23. 32-Bit Opcode Instructions (Sheet 4 of 40)

Instruction and Version	Opcode Range
Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (S2RND)	0xC023 2800— 0xC023 2FFF
Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 –= Dreg_lo_hi * Dreg_lo_hi) (S2RND)	0xC023 3000— 0xC023 37FF
Move Register Half Dreg_lo = A0 (S2RND)	0xC023 3800— 0xC023 39C0
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (S2RND)	0xC024 1800— 0xC024 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) (S2RND)
	0xC024 2000—
	0xC027 FFFF

Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (S2RND)	0xC025 1800— 0xC025 D9FF
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (S2RND)	0xC026 1800— 0xC026 D9FF
Move Register Half Dreg_hi = A1 (S2RND)	0xC027 1800— 0xC027 19C0

Move Register Half
Dreg_lo = A0, Dreg_hi = A1 (S2RND)
Dreg_hi = A1, Dreg_lo = A0 (S2RND)
	0xC027 3800—
	0xC027 39C0

Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (S2RND)	0xC028 1800— 0xC028 D9FF
Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (S2RND)	0xC029 1800— 0xC029 D9FF
Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (S2RND)	0xC02A 1800— 0xC02A D9FF

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) (S2RND)
	0xC02C 2000—
	0xC02F FFFF

Multiply and Multiply-Accumulate to Data Register Dreg_even = (A0 = Dreg_lo_hi * Dreg_lo_hi) (S2RND)	0xC02D 0000— 0xC02D 07FF
Multiply and Multiply-Accumulate to Data Register Dreg_even = (A0 += Dreg_lo_hi * Dreg_lo_hi) (S2RND)	0xC02D 0800— 0xC02D 0FFF

Table C-23. 32-Bit Opcode Instructions (Sheet 5 of 40)

Instruction and Version	Opcode Range
Multiply and Multiply-Accumulate to Data Register Dreg_even = (A0 –= Dreg_lo_hi * Dreg_lo_hi) (S2RND)	0xC02D 1000— 0xC02D 17FF
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)	0xC034 1800— 0xC034 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) (S2RND, M)
	0xC034 2000—
	0xC037 FFFF

Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)	0xC035 1800— 0xC035 D9FF
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)	0xC036 1800— 0xC036 D9FF
Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)	0xC038 1800— 0xC038 D9FF
Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)	0xC039 1800— 0xC039 D9FF
Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (S2RND, M)	0xC03A 1800— 0xC03A D9FF

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) (S2RND, M)
	0xC03C 2000—
	0xC03F FFFF

Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (T)	0xC043 2000— 0xC043 27FF
Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (T)	0xC043 2800— 0xC043 2FFF
Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 –= Dreg_lo_hi * Dreg_lo_hi) (T)	0xC043 3000— 0xC043 37FF
Move Register Half Dreg_lo = A0 (T)	0xC043 3800— 0xC043 39C0
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (T)	0xC044 1800— 0xC044 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)
	0xC044 2000—
	0xC047 FFFF

Table C-23. 32-Bit Opcode Instructions (Sheet 6 of 40)

Instruction and Version	Opcode Range
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)
Dreg_hi = A1, Dreg_lo = A0 (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 0E3F
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)

Instruction and Version	Opcode Range
Vector Multiply and Multiply-Accumulate A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi , A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi (W32, M)	0xC070 0000— 0xC073 DE3F
Multiply and Multiply-Accumulate to Accumulator A1 = Dreg_lo_hi * Dreg_lo_hi (W32, M)	0xC070 1800— 0xC070 D83F
Multiply and Multiply-Accumulate to Accumulator A1 += Dreg_lo_hi * Dreg_lo_hi (W32, M)	0xC071 1800— 0xC071 D83F
Multiply and Multiply-Accumulate to Accumulator A1 –= Dreg_lo_hi * Dreg_lo_hi (W32, M)	0xC072 1800— 0xC072 D83F
Vector Multiply and Multiply-Accumulate A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi , A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi (FU)	0xC080 0000— 0xC083 DE3F
Multiply and Multiply-Accumulate to Accumulator A1 = Dreg_lo_hi * Dreg_lo_hi (FU)	0xC080 1800— 0xC080 D83F
Vector Multiply and Multiply-Accumulate Dreg_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi (FU)	0xC080 2000— 0xC083 FFFF
Multiply and Multiply-Accumulate to Accumulator A1 += Dreg_lo_hi * Dreg_lo_hi (FU)	0xC081 1800— 0xC081 D83F
Multiply and Multiply-Accumulate to Accumulator A1 –= Dreg_lo_hi * Dreg_lo_hi (FU)	0xC082 1800— 0xC082 D83F
Multiply and Multiply-Accumulate to Accumulator A0 = Dreg_lo_hi * Dreg_lo_hi (FU)	0xC083 0000— 0xC083 063F
Multiply and Multiply-Accumulate to Accumulator A0 += Dreg_lo_hi * Dreg_lo_hi (FU)	0xC083 0800— 0xC083 0E3F
Multiply and Multiply-Accumulate to Accumulator A0 –= Dreg_lo_hi * Dreg_lo_hi (FU)	0xC083 1000— 0xC083 163F
Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (FU)	0xC083 2000— 0xC083 27FF
Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (FU)	0xC083 2800— 0xC083 2FFF
Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 –= Dreg_lo_hi * Dreg_lo_hi) (FU)	0xC083 3000— 0xC083 37FF

Table C-23. 32-Bit Opcode Instructions (Sheet 8 of 40)

Instruction and Version	Opcode Range
Move Register Half Dreg_lo = A0 (FU)	0xC083 3800— 0xC083 39C0
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (FU)	0xC084 1800— 0xC084 D9FF
Vector Multiply and Multiply-Accumulate A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (FU)	0xC084 0000— 0xC087 DFFF

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) (FU)
	0xC084 2000—
	0xC087 FFFF

Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (FU)	0xC085 1800— 0xC085 D9FF
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (FU)	0xC086 1800— 0xC086 D9FF
Move Register Half Dreg_hi = A1 (FU)	0xC087 1800— 0xC087 19C0

Move Register Half
Dreg_lo = A0, Dreg_hi = A1 (FU)
Dreg_hi = A1, Dreg_lo = A0 (FU)
	0xC087 3800—
	0xC087 39C0

Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (FU)	0xC088 1800— 0xC088 D9FF
Vector Multiply and Multiply-Accumulate Dreg_even = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi (FU)	0xC088 2000— 0xC08B FFFF
Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (FU)	0xC089 1800— 0xC089 D9FF
Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (FU)	0xC08A 1800— 0xC08A D9FF
Vector Multiply and Multiply-Accumulate A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi , Dreg_odd = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (FU)	0xC08C 0000— 0xC08F 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) (FU)
	0xC08C 2000—
	0xC08F FFFF

Table C-23. 32-Bit Opcode Instructions (Sheet 9 of 40)

Instruction and Version	Opcode Range
Multiply and Multiply-Accumulate to Data Register Dreg_even = (A0 = Dreg_lo_hi * Dreg_lo_hi) (FU)	0xC08D 0000— 0xC08D 07FF
Multiply and Multiply-Accumulate to Data Register Dreg_even = (A0 += Dreg_lo_hi * Dreg_lo_hi) (FU)	0xC08D 0800— 0xC08D 0FFF
Multiply and Multiply-Accumulate to Data Register Dreg_even = (A0 –= Dreg_lo_hi * Dreg_lo_hi) (FU)	0xC08D 1000— 0xC08D 17FF
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (FU, M)	0xC094 1800— 0xC094 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) (FU, M)
	0xC094 2000—
	0xC097 FFFF

Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (FU, M)	0xC095 1800— 0xC095 D9FF
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (FU, M)	0xC096 1800— 0xC096 D9FF
Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (FU, M)	0xC098 1800— 0xC098 D9FF
Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (FU, M)	0xC099 1800— 0xC099 D9FF
Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (FU, M)	0xC09A 1800— 0xC09A D9FF

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) (FU, M)
	0xC09C 2000—
	0xC09F FFFF

Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (TFU)	0xC0C3 2000— 0xC0C3 27FF
Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (TFU)	0xC0C3 2800— 0xC0C3 2FFF
Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 –= Dreg_lo_hi * Dreg_lo_hi) (TFU)	0xC0C3 3000— 0xC0C3 37FF
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (TFU)	0xC0C4 1800— 0xC0C4 D9FF

Table C-23. 32-Bit Opcode Instructions (Sheet 10 of 40)

Instruction and Version	Opcode Range

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) (TFU)
	0xC0C4 2000—
	0xC0C7 FFFF

Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (TFU)	0xC0C5 1800— 0xC0C5 D9FF
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (TFU)	0xC0C6 1800— 0xC0C6 D9FF
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (TFU, M)	0xC0D4 1800— 0xC0D4 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) (TFU, M)
	0xC0D4 2000—
	0xC0D7 FFFF

Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (TFU, M)	0xC0D5 1800— 0xC0D5 D9FF
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (TFU, M)	0xC0D6 1800— 0xC0D6 D9FF
Vector Multiply and Multiply-Accumulate A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi , A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi (IS)	0xC100 0000— 0xC103 DE3F
Multiply and Multiply-Accumulate to Accumulator A1 = Dreg_lo_hi * Dreg_lo_hi (IS)	0xC100 1800— 0xC100 D83F
Vector Multiply and Multiply-Accumulate Dreg_lo = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi (IS)	0xC100 2000— 0xC103 FFFF
Multiply and Multiply-Accumulate to Accumulator A1 += Dreg_lo_hi * Dreg_lo_hi (IS)	0xC101 1800— 0xC101 D83F
Multiply and Multiply-Accumulate to Accumulator A1 –= Dreg_lo_hi * Dreg_lo_hi (IS)	0xC102 1800— 0xC102 D83F
Multiply and Multiply-Accumulate to Accumulator A0 = Dreg_lo_hi * Dreg_lo_hi (IS)	0xC103 0000— 0xC103 063F
Multiply and Multiply-Accumulate to Accumulator A0 += Dreg_lo_hi * Dreg_lo_hi (IS)	0xC103 0800— 0xC103 0E3F
Multiply and Multiply-Accumulate to Accumulator A0 –= Dreg_lo_hi * Dreg_lo_hi (IS)	0xC103 1000— 0xC103 163F

Table C-23. 32-Bit Opcode Instructions (Sheet 11 of 40)

Instruction and Version	Opcode Range
Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (IS)	0xC103 2000— 0xC103 27FF
Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (IS)	0xC103 2800— 0xC103 2FFF
Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 –= Dreg_lo_hi * Dreg_lo_hi) (IS)	0xC103 3000— 0xC103 37FF
Move Register Half Dreg_lo = A0 (IS)	0xC103 3800— 0xC103 39C0
Vector Multiply and Multiply-Accumulate A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi , Dreg_hi = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (IS)	0xC104 0000— 0xC107 DFFF
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IS)	0xC104 1800— 0xC104 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) (IS)
	0xC104 2000—
	0xC107 FFFF

Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IS)	0xC105 1800— 0xC105 D9FF
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (IS)	0xC106 1800— 0xC106 D9FF
Move Register Half Dreg_hi = A1 (IS)	0xC107 1800— 0xC107 19C0

Move Register Half
Dreg_lo = A0, Dreg_hi = A1 (IS)
Dreg_hi = A1, Dreg_lo = A0 (IS)
	0xC107 3800—
	0xC107 39C0

Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IS)	0xC108 1800— 0xC108 D9FF
Vector Multiply and Multiply-Accumulate Dreg_even = (A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) , A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi (IS)	0xC108 2000— 0xC10B FFFF
Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IS)	0xC109 1800— 0xC109 D9FF
Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (IS)	0xC10A 1800— 0xC10A D9FF

Table C-23. 32-Bit Opcode Instructions (Sheet 12 of 40)

Instruction and Version	Opcode Range
Vector Multiply and Multiply-Accumulate A0 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi , Dreg_odd = (A1 {=, +=, or –=} Dreg_lo_hi * Dreg_lo_hi) (IS)	0xC10C 0000— 0xC10F 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) (IS)
	0xC10C 2000—
	0xC10F FFFF

Multiply and Multiply-Accumulate to Data Register Dreg_even = (A0 = Dreg_lo_hi * Dreg_lo_hi) (IS)	0xC10D 0000— 0xC10D 07FF
Multiply and Multiply-Accumulate to Data Register Dreg_even = (A0 += Dreg_lo_hi * Dreg_lo_hi) (IS)	0xC10D 0800— 0xC10D 0FFF
Multiply and Multiply-Accumulate to Data Register Dreg_even = (A0 –= Dreg_lo_hi * Dreg_lo_hi) (IS)	0xC10D 1000— 0xC10D 17FF
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IS, M)	0xC114 1800— 0xC114 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) (IS, M)
	0xC114 2000—
	0xC117 FFFF

Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IS, M)	0xC115 1800— 0xC115 D9FF
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (IS, M)	0xC116 1800— 0xC116 D9FF
Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IS, M)	0xC118 1800— 0xC118 D9FF
Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IS, M)	0xC119 1800— 0xC119 D9FF
Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (IS, M)	0xC11A 1800— 0xC11A D9FF

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) (IS, M)
	0xC11C 2000—
	0xC11F FFFF

Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (ISS2)	0xC123 2000— 0xC123 27FF
Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (ISS2)	0xC123 2800— 0xC123 2FFF

Table C-23. 32-Bit Opcode Instructions (Sheet 13 of 40)

Instruction and Version	Opcode Range
Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 –= Dreg_lo_hi * Dreg_lo_hi) (ISS2)	0xC123 3000— 0xC123 37FF
Move Register Half Dreg_lo = A0 (ISS2)	0xC123 3800— 0xC123 39C0
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (ISS2)	0xC124 1800— 0xC124 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) (ISS2)
	0xC124 2000—
	0xC127 FFFF

Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (ISS2)	0xC125 1800— 0xC125 D9FF
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (ISS2)	0xC126 1800— 0xC126 D9FF
Move Register Half Dreg_hi = A1 (ISS2)	0xC127 1800— 0xC127 19C0

Move Register Half
Dreg_lo = A0, Dreg_hi = A1 (ISS2)
Dreg_hi = A1, Dreg_lo = A0 (ISS2)
	0xC127 3800—
	0xC127 39C0

Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (ISS2)	0xC128 1800— 0xC128 D9FF
Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (ISS2)	0xC129 1800— 0xC129 D9FF
Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (ISS2)	0xC12A 1800— 0xC12A D9FF

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) (ISS2)
	0xC12C 2000—
	0xC12F FFFF

Multiply and Multiply-Accumulate to Data Register Dreg_even = (A0 = Dreg_lo_hi * Dreg_lo_hi) (ISS2)	0xC12D 0000— 0xC12D 07FF
Multiply and Multiply-Accumulate to Data Register Dreg_even = (A0 += Dreg_lo_hi * Dreg_lo_hi) (ISS2)	0xC12D 0800— 0xC12D 0FFF
Multiply and Multiply-Accumulate to Data Register Dreg_even = (A0 –= Dreg_lo_hi * Dreg_lo_hi) (ISS2)	0xC12D 1000— 0xC12D 17FF

Table C-23. 32-Bit Opcode Instructions (Sheet 14 of 40)

Instruction and Version	Opcode Range
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)	0xC134 1800— 0xC134 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) (ISS2, M)
	0xC134 2000—
	0xC137 FFFF

Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)	0xC135 1800— 0xC135 D9FF
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)	0xC136 1800— 0xC136 D9FF
Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)	0xC138 1800— 0xC138 D9FF
Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)	0xC139 1800— 0xC139 D9FF
Multiply and Multiply-Accumulate to Data Register Dreg_odd = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (ISS2, M)	0xC13A 1800— 0xC13A D9FF

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) (ISS2, M)
	0xC13C 2000—
	0xC13F FFFF

Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (IH)	0xC163 2000— 0xC163 27FF
Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (IH)	0xC163 2800— 0xC163 2FFF
Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 –= Dreg_lo_hi * Dreg_lo_hi) (IH)	0xC163 3000— 0xC163 37FF
Move Register Half Dreg_lo = A0 (IH)	0xC163 3800— 0xC163 39C0
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IH)	0xC164 1800— 0xC164 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) (IH)
	0xC164 2000—
	0xC167 FFFF

Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IH)	0xC165 1800— 0xC165 D9FF

Table C-23. 32-Bit Opcode Instructions (Sheet 15 of 40)

Instruction and Version	Opcode Range
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (IH)	0xC166 1800— 0xC166 D9FF
Move Register Half Dreg_hi = A1 (IH)	0xC167 1800— 0xC167 19C0

Move Register Half
Dreg_lo = A0, Dreg_hi = A1 (IH)
Dreg_hi = A1, Dreg_lo = A0 (IH)
	0xC167 3800—
	0xC167 39C0

Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IH, M)	0xC174 1800— 0xC174 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) (IH, M)
	0xC174 2000—
	0xC177 FFFF

Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IH, M)	0xC175 1800— 0xC175 D9FF
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (IH, M)	0xC176 1800— 0xC176 D9FF
Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) (IU)	0xC183 2000— 0xC183 27FF
Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) (IU)	0xC183 2800— 0xC183 2FFF
Multiply and Multiply-Accumulate to Half Register Dreg_lo = (A0 –= Dreg_lo_hi * Dreg_lo_hi) (IU)	0xC183 3000— 0xC183 37FF
Move Register Half Dreg_lo = A0 (IU)	0xC183 3800— 0xC183 39C0
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IU)	0xC184 1800— 0xC184 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) (IU)
	0xC184 2000—
	0xC187 FFFF

Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IU)	0xC185 1800— 0xC185 D9FF
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (IU)	0xC186 1800— 0xC186 D9FF

Table C-23. 32-Bit Opcode Instructions (Sheet 16 of 40)

Instruction and Version	Opcode Range
Move Register Half Dreg_hi = A1 (IU)	0xC187 1800— 0xC187 19C0

Move Register Half
Dreg_lo = A0, Dreg_hi = A1 (IU)
Dreg_hi = A1, Dreg_lo = A0 (IU)
	0xC187 3800—
	0xC187 39C0

Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) (IU, M)	0xC194 1800— 0xC194 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) (IU, M)
	0xC194 2000—
	0xC197 FFFF

Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) (IU, M)	0xC195 1800— 0xC195 D9FF
Multiply and Multiply-Accumulate to Half Register Dreg_hi = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (IU, M)	0xC196 1800— 0xC196 D9FF
Multiply 16-Bit Operands Dreg_lo = Dreg_lo_hi * Dreg_lo_hi	0xC200 2000— 0xC200 27FF
Multiply 16-Bit Operands Dreg_hi = Dreg_lo_hi * Dreg_lo_hi	0xC204 0000— 0xC204 C1FF

Vector Multiply
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi
	0xC204 2000—
	0xC204 E7FF

Multiply 16-Bit Operands Dreg_even = Dreg_lo_hi * Dreg_lo_hi	0xC208 2000— 0xC208 27FF
Multiply 16-Bit Operands Dreg_odd = Dreg_lo_hi * Dreg_lo_hi	0xC20C 0000— 0xC20C C1FF

Vector Multiply
Dreg_even = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi
	0xC20C 2000—
	0xC20C E7FF

Multiply 16-Bit Operands Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (M)	0xC214 0000— 0xC214 C1FF

Vector Multiply
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (M)
	0xC214 2000—
	0xC214 E7FF

Table C-23. 32-Bit Opcode Instructions (Sheet 17 of 40)

Instruction and Version	Opcode Range
Multiply 16-Bit Operands Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (M)	0xC21C 0000— 0xC21C C1FF

Vector Multiply
Dreg_even = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (M)
	0xC21C 2000—
	0xC21C E7FF

Multiply 16-Bit Operands Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (S2RND)	0xC220 2000— 0xC220 27FF
Multiply 16-Bit Operands Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (S2RND)	0xC224 0000— 0xC224 C1FF

Vector Multiply
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (S2RND)
	0xC224 2000—
	0xC224 E7FF

Multiply 16-Bit Operands Dreg_even = Dreg_lo_hi * Dreg_lo_hi (S2RND)	0xC228 2000— 0xC228 27FF
Multiply 16-Bit Operands Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (S2RND)	0xC22C 0000— 0xC22C C1FF

Vector Multiply
Dreg_even = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (S2RND)
	0xC22C 2000—
	0xC22C E7FF

Multiply 16-Bit Operands Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (S2RND, M)	0xC234 0000— 0xC234 C1FF

Vector Multiply
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (S2RND, M)
	0xC234 2000—
	0xC234 E7FF

Multiply 16-Bit Operands Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (S2RND, M)	0xC239 0000— 0xC239 C1FF

Vector Multiply
Dreg_even = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (S2RND, M)
	0xC239 2000—
	0xC239 E7FF

Multiply 16-Bit Operands Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (T)	0xC240 2000— 0xC240 27FF
Multiply 16-Bit Operands Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (T)	0xC244 0000— 0xC244 C1FF

Table C-23. 32-Bit Opcode Instructions (Sheet 18 of 40)

Instruction and Version	Opcode Range

Vector Multiply
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (T)
	0xC244 2000—
	0xC244 E7FF

Multiply 16-Bit Operands Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (T, M)	0xC254 0000— 0xC254 C1FF

Vector Multiply
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (T, M)
	0xC254 2000—
	0xC254 E7FF

Multiply 16-Bit Operands Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (FU)	0xC280 2000— 0xC280 27FF
Multiply 16-Bit Operands Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (FU)	0xC284 0000— 0xC284 C1FF

Vector Multiply
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (FU)
	0xC284 2000—
	0xC284 E7FF

Multiply 16-Bit Operands Dreg_even = Dreg_lo_hi * Dreg_lo_hi (FU)	0xC288 2000— 0xC288 27FF
Multiply 16-Bit Operands Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (FU)	0xC28C 0000— 0xC28C C1FF

Vector Multiply
Dreg_even = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (FU)
	0xC28C 2000—
	0xC28C E7FF

Multiply 16-Bit Operands Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (FU, M)	0xC294 0000— 0xC294 C1FF

Vector Multiply
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (FU, M)
	0xC294 2000—
	0xC294 E7FF

Multiply 16-Bit Operands Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (FU, M)	0xC29C 0000— 0xC29C C1FF

Vector Multiply
Dreg_even = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (FU, M)
	0xC29C 2000—
	0xC29C E7FF

Multiply 16-Bit Operands Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (TFU)	0xC2C0 2000— 0xC2C0 27FF

Table C-23. 32-Bit Opcode Instructions (Sheet 19 of 40)

Instruction and Version	**Opcode Range**
Multiply 16-Bit Operands Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (TFU)	0xC2C4 0000— 0xC2C4 C1FF

Vector Multiply
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (TFU)
	0xC2C4 2000—
	0xC2C4 E7FF

Multiply 16-Bit Operands Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (TFU, M)	0xC2D4 0000— 0xC2D4 C1FF

Vector Multiply
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (TFU, M)
	0xC2D4 2000—
	0xC2D4 E7FF

Multiply 16-Bit Operands Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (IS)	0xC300 2000— 0xC300 27FF
Multiply 16-Bit Operands Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IS)	0xC304 0000— 0xC304 C1FF

Vector Multiply
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IS)
	0xC304 2000—
	0xC304 E7FF

Multiply 16-Bit Operands Dreg_even = Dreg_lo_hi * Dreg_lo_hi (IS)	0xC308 2000— 0xC308 27FF
Multiply 16-Bit Operands Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (IS)	0xC30C 0000— 0xC30C C1FF

Vector Multiply
Dreg_even = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (IS)
	0xC30C 2000—
	0xC30C E7FF

Multiply 16-Bit Operands Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IS, M)	0xC314 0000— 0xC314 C1FF

Vector Multiply
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IS, M)
	0xC314 2000—
	0xC314 E7FF

Multiply 16-Bit Operands Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (IS, M)	0xC31C 0000— 0xC31C C1FF

Vector Multiply
Dreg_even = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (IS, M)
	0xC31C 2000—
	0xC31C E7FF

Table C-23. 32-Bit Opcode Instructions (Sheet 20 of 40)

Instruction and Version	Opcode Range
Multiply 16-Bit Operands Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (ISS2)	0xC320 2000— 0xC320 27FF
Multiply 16-Bit Operands Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (ISS2)	0xC324 0000— 0xC324 C1FF

Vector Multiply
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (ISS2)
	0xC324 2000—
	0xC324 E7FF

Multiply 16-Bit Operands Dreg_even = Dreg_lo_hi * Dreg_lo_hi (ISS2)	0xC328 2000— 0xC328 27FF
Multiply 16-Bit Operands Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (ISS2)	0xC32C 0000— 0xC32C C1FF

Vector Multiply
Dreg_even = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (ISS2)
	0xC32C 2000—
	0xC32C E7FF

Multiply 16-Bit Operands Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (ISS2, M)	0xC334 0000— 0xC334 C1FF

Vector Multiply
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (ISS2, M)
	0xC334 2000—
	0xC334 E7FF

Multiply 16-Bit Operands Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (ISS2, M)	0xC33C 0000— 0xC33C C1FF

Vector Multiply
Dreg_even = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_odd = Dreg_lo_hi * Dreg_lo_hi (ISS2, M)
	0xC33C 2000—
	0xC33C E7FF

Multiply 16-Bit Operands Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (IH)	0xC360 2000— 0xC360 27FF
Multiply 16-Bit Operands Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IH)	0xC364 0000— 0xC364 C1FF

Vector Multiply
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IH)
	0xC364 2000—
	0xC364 E7FF

Multiply 16-Bit Operands Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IH, M)	0xC374 0000— 0xC374 C1FF

Table C-23. 32-Bit Opcode Instructions (Sheet 21 of 40)

Instruction and Version	Opcode Range

Vector Multiply
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IH, M)
	0xC374 2000—
	0xC374 E7FF

Multiply 16-Bit Operands Dreg_lo = Dreg_lo_hi * Dreg_lo_hi (IU)	0xC380 2000— 0xC380 27FF
Multiply 16-Bit Operands Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IU)	0xC384 0000— 0xC384 C1FF

Vector Multiply
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IU)
	0xC384 2000—
	0xC384 E7FF

Multiply 16-Bit Operands Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IU, M)	0xC394 0000— 0xC394 C1FF

Vector Multiply
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ,
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi (IU, M)
	0xC394 2000—
	0xC394 E7FF

Vector Add / Subtract Dreg = Dreg +|+ Dreg	0xC400 0000— 0xC400 0E3F
Vector Add / Subtract Dreg = Dreg +|+ Dreg (CO)	0xC400 1000— 0xC400 1E3F
Vector Add / Subtract Dreg = Dreg +|+ Dreg (S)	0xC400 2000— 0xC400 2E3F
Vector Add / Subtract Dreg = Dreg +|+ Dreg (SC0)	0xC400 3000— 0xC400 3E3F
Vector Add / Subtract Dreg = Dreg +|– Dreg	0xC400 4000— 0xC400 4E3F
Vector Add / Subtract Dreg = Dreg +|– Dreg (CO)	0xC400 5000— 0xC400 5E3F
Vector Add / Subtract Dreg = Dreg +|– Dreg (S)	0xC400 6000— 0xC400 6E3F
Vector Add / Subtract Dreg = Dreg +|– Dreg (SC0)	0xC400 7000— 0xC400 7E3F
Vector Add / Subtract Dreg = Dreg –|+ Dreg	0xC400 8000— 0xC400 8E3F

Table C-23. 32-Bit Opcode Instructions (Sheet 22 of 40)

Instruction and Version	Opcode Range
Vector Add / Subtract Dreg = Dreg –|+ Dreg (CO)	0xC400 9000— 0xC400 9E3F
Vector Add / Subtract Dreg = Dreg –|+ Dreg (S)	0xC400 A000— 0xC400 AE3F
Vector Add / Subtract Dreg = Dreg –|+ Dreg (SC0)	0xC400 B000— 0xC400 BE3F
Vector Add / Subtract Dreg = Dreg –|– Dreg	0xC400 C000— 0xC400 CE3F
Vector Add / Subtract Dreg = Dreg –|– Dreg (CO)	0xC400 D000— 0xC400 DE3F
Vector Add / Subtract Dreg = Dreg –|– Dreg (S)	0xC400 E000— 0xC400 EE3F
Vector Add / Subtract Dreg = Dreg –|– Dreg (SC0)	0xC400 F000— 0xC400 FE3F
Vector Add / Subtract Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg	0xC401 0000— 0xC401 0FFF
Vector Add / Subtract Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg (CO)	0xC401 1000— 0xC401 1FFF
Vector Add / Subtract Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg (S)	0xC401 2000— 0xC401 2FFF
Vector Add / Subtract Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg (SCO)	0xC401 3000— 0xC401 3FFF
Vector Add / Subtract Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg (ASR)	0xC401 8000— 0xC401 8FFF
Vector Add / Subtract Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg (CO, ASR)	0xC401 9000— 0xC401 9FFF
Vector Add / Subtract Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg (S, ASR)	0xC401 A000— 0xC401 AFFF
Vector Add / Subtract Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg (SCO, ASR)	0xC401 B000— 0xC401 BFFF
Vector Add / Subtract Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg (ASL)	0xC401 C000— 0xC401 CFFF

Table C-23. 32-Bit Opcode Instructions (Sheet 23 of 40)

Instruction and Version	Opcode Range
Vector Add / Subtract Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg (CO, ASL)	0xC401 D000— 0xC401 DFFF
Vector Add / Subtract Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg (S, ASL)	0xC401 E000— 0xC401 EFFF
Vector Add / Subtract Dreg = Dreg +|+ Dreg, Dreg = Dreg –|– Dreg (SCO, ASL)	0xC401 F000— 0xC401 FFFF
Add Dreg_lo = Dreg_lo + Dreg_lo (NS)	0xC402 0000— 0xC402 0E3F
Add Dreg_lo = Dreg_lo + Dreg_lo (S)	0xC402 2000— 0xC402 2E3F
Add Dreg_lo = Dreg_lo + Dreg_hi (NS)	0xC402 4000— 0xC402 4E3F
Add Dreg_lo = Dreg_lo + Dreg_hi (S)	0xC402 6000— 0xC402 6E3F
Add Dreg_lo = Dreg_hi + Dreg_lo (NS)	0xC402 8000— 0xC402 8E3F
Add Dreg_lo = Dreg_hi + Dreg_lo (S)	0xC402 A000— 0xC402 AE3F
Add Dreg_lo = Dreg_hi + Dreg_hi (NS)	0xC402 C000— 0xC402 CE3F
Add Dreg_lo = Dreg_hi + Dreg_hi (S)	0xC402 E000— 0xC402 EE3F
Subtract Dreg_lo = Dreg_lo – Dreg_lo (NS)	0xC403 0000— 0xC403 0E3F
Subtract Dreg_lo = Dreg_lo – Dreg_lo (S)	0xC403 2000— 0xC403 2E3F
Subtract Dreg_lo = Dreg_lo – Dreg_hi (NS)	0xC403 4000— 0xC403 4E3F
Subtract Dreg_lo = Dreg_lo – Dreg_hi (S)	0xC403 6000— 0xC403 6E3F
Subtract Dreg_lo = Dreg_hi – Dreg_lo (NS)	0xC403 8000— 0xC403 8E3F

Table C-23. 32-Bit Opcode Instructions (Sheet 24 of 40)

Instruction **and Version**	**Opcode** **Range**
Subtract Dreg_lo = Dreg_hi – Dreg_lo (S)	0xC403 A000— 0xC403 AE3F
Subtract Dreg_lo = Dreg_hi – Dreg_hi (NS)	0xC403 C000— 0xC403 CE3F
Subtract Dreg_lo = Dreg_hi – Dreg_hi (S)	0xC403 E000— 0xC403 EE3F
Add Dreg = Dreg + Dreg (NS)	0xC404 0000— 0xC404 0E3F
Add Dreg = Dreg + Dreg (S)	0xC404 2000— 0xC404 2E3F
Subtract Dreg = Dreg – Dreg (NS)	0xC404 4000— 0xC404 4E3F
Subtract Dreg = Dreg – Dreg (S)	0xC404 6000— 0xC404 6E3F
Vector Add / Subtract Dreg = Dreg + Dreg, Dreg = Dreg – Dreg	0xC404 8000— 0xC404 8FFF
Vector Add / Subtract Dreg = Dreg + Dreg, Dreg = Dreg – Dreg (S)	0xC404 A000— 0xC404 AFFF
Add/Subtract-Prescale Up Dreg_lo = Dreg + Dreg (RND12)	0xC405 0000— 0xC405 0E3F
Add/Subtract-Prescale Up Dreg_lo = Dreg – Dreg (RND12)	0xC405 4000— 0xC405 4E3F
Add/Subtract-Prescale Down Dreg_lo = Dreg + Dreg (RND20)	0xC405 9000— 0xC405 9E3F
Add/Subtract-Prescale Down Dreg_lo = Dreg – Dreg (RND20)	0xC405 D000— 0xC405 DE3F
Vector Maximum Dreg = MAX (Dreg, Dreg) (V)	0xC406 0000— 0xC406 0E3F
Vector Minimum Dreg = MIN (Dreg, Dreg) (V)	0xC406 4000— 0xC406 4E3F
Vector Absolute Value Dreg = ABS Dreg (V)	0xC406 8000— 0xC406 8E38

Table C-23. 32-Bit Opcode Instructions (Sheet 25 of 40)

Instruction and Version	Opcode Range
Maximum Dreg = MAX (Dreg, Dreg)	0xC407 0000— 0xC407 0E3F
Minimum Dreg = MIN (Dreg, Dreg)	0xC407 4000— 0xC407 4E3F
Absolute Value Dreg = ABS Dreg	0xC407 8000— 0xC407 8E38
Negate (Two's-Complement) Dreg = – Dreg (NS)	0xC407 C000— 0xC407 CFC0
Negate (Two's-Complement) Dreg = – Dreg (S)	0xC407 E000— 0xC407 EFC0
Load Immediate A0 = 0	0xC408 003F
Saturate A0 = A0 (S)	0xC408 203F
Load Immediate A1 = 0	0xC408 403F
Saturate A1 = A1 (S)	0xC408 603F
Load Immediate A1 = A0 = 0	0xC408 803F
Saturate A1 = A1 (S), A0 = A0 (S)	0xC408 A03F
Move Register Half A0.L = Dreg_lo	0xC409 0000— 0xC409 0038
Move Register Half A0.X = Dreg_lo	0xC409 4000— 0xC409 4038
Move Register Half A1.L = Dreg_lo	0xC409 8000— 0xC409 8038
Move Register Half A1.X = Dreg_lo	0xC409 C000— 0xC409 C038
Move Register Half Dreg_lo = A0.X	0xC40A 003F— 0xC40A 0E00

Table C-23. 32-Bit Opcode Instructions (Sheet 26 of 40)

Instruction and Version	Opcode Range
Move Register Half Dreg_lo = A1.X	0xC40A 403F— 0xC40A 4E00
Modify-Increment Dreg = (A0 += A1)	0xC40B 003F— 0xC40B 0E00
Modify-Increment Dreg_lo = (A0 += A1)	0xC40B 403F— 0xC40B 4E00
Modify-Increment A0 += A1	0xC40B 803F
Modify-Increment A0 += A1 (W32)	0xC40B A03F
Modify-Decrement A0 –= A1	0xC40B C03F
Modify-Decrement A0 –= A1 (W32)	0xC40B E03F
Add on Sign Dreg_hi = Dreg_lo = SIGN (Dreg_hi) * Dreg_hi + SIGN (Dreg_lo) * Dreg_lo	0xC40C 0000— 0xC40C 0E38
Dual 16-Bit Accumulator Extraction with Addition Dreg = A1.L + A1.H, Dreg = A0.L + A0.H	0xC40C 403F— 0xC40C 4FC0
Round to Half Word Dreg_lo = Dreg (RND)	0xC40C C000— 0xC40C CE38
Vector Search (Dreg, Dreg) = SEARCH Dreg (GT)	0xC40D 0000— 0xC40D 2FFF
Vector Search (Dreg, Dreg) = SEARCH Dreg (GE)	0xC40D 4000— 0xC40D 6FFF
Vector Search (Dreg, Dreg) = SEARCH Dreg (LT)	0xC40D 8000— 0xC40D AFF8
Vector Search (Dreg, Dreg) = SEARCH Dreg (LE)	0xC40D C000— 0xC40D EFF8
Negate (Two's-Complement) A0 = – A0	0xC40E 003F
Negate (Two's-Complement) A0 = – A1	0xC40E 403F

Table C-23. 32-Bit Opcode Instructions (Sheet 27 of 40)

Instruction and Version	Opcode Range
Negate (Two's-Complement) A1 = – A1, A0 = – A0	0xC40E C03F
Vector Negate (Two's-Complement) Dreg = – Dreg (V)	0xC40F C000— 0xC40F CE38
Absolute Value A0 = ABS A0	0xC410 0000
Absolute Value A0 = ABS A1	0xC410 403F
Absolute Value A1 = ABS A1, A0 = ABS A0	0xC410 C03F
Vector Add / Subtract Dreg = A1 + A0, Dreg = A1 – A0	0xC411 003F— 0xC411 0FC0
Vector Add / Subtract Dreg = A1 + A0, Dreg = A1 – A0 (S)	0xC411 203F— 0xC411 2FC0
Vector Add / Subtract Dreg = A0 + A1, Dreg = A0 – A1	0xC411 403F— 0xC411 4FC0
Vector Add / Subtract Dreg = A0 + A1, Dreg = A0 – A1 (S)	0xC411 603F— 0xC411 6FC0
Quad 8-Bit Subtract-Absolute-Accumulate SAA (Dreg_pair, Dreg_pair)	0xC412 0000— 0xC412 003F
Quad 8-Bit Subtract-Absolute-Accumulate SAA (Dreg_pair, Dreg_pair) (R)	0xC412 2000— 0xC412 203F
Disable Alignment Exception for Load DISALGNEXCPT	0xC412 C000
Quad 8-Bit Average-Byte Dreg = BYTEOP1P (Dreg_pair, Dreg_pair)	0xC414 0000— 0xC414 0E3F
Quad 8-Bit Average-Byte Dreg = BYTEOP1P (Dreg_pair, Dreg_pair) (R)	0xC414 2000— 0xC414 2E3F
Quad 8-Bit Average-Byte Dreg = BYTEOP1P (Dreg_pair, Dreg_pair) (T)	0xC414 4000— 0xC414 4E3F
Quad 8-Bit Average-Byte Dreg = BYTEOP1P (Dreg_pair, Dreg_pair) (T, R)	0xC414 6000— 0xC414 6E3F

Table C-23. 32-Bit Opcode Instructions (Sheet 28 of 40)

Instruction and Version	Opcode Range
Quad 8-Bit Add (Dreg, Dreg) = BYTEOP16P (Dreg_pair, Dreg_pair)	0xC415 0000— 0xC415 0FFF
Quad 8-Bit Add (Dreg, Dreg) = BYTEOP16P (Dreg_pair, Dreg_pair) (R)	0xC415 2000— 0xC415 2FFF
Quad 8-Bit Subtract (Dreg, Dreg) = BYTEOP16M (Dreg_pair, Dreg_pair)	0xC415 4000— 0xC415 4FFF
Quad 8-Bit Subtract (Dreg, Dreg) = BYTEOP16M (Dreg_pair, Dreg_pair) (R)	0xC415 6000— 0xC415 6FFF
Quad 8-Bit Average-Half Word Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (RNDL)	0xC416 0000— 0xC416 0E3F
Quad 8-Bit Average-Half Word Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (RNDL, R)	0xC416 2000— 0xC416 2E3F
Quad 8-Bit Average-Half Word Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (TL)	0xC416 4000— 0xC416 6E3F
Quad 8-Bit Average-Half Word Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (TL, R)	0xC416 6000— 0xC416 7E3F
Dual 16-Bit Add / Clip Dreg = BYTEOP3P (Dreg_pair, Dreg_pair) (LO)	0xC417 0000— 0xC417 0E3F
Dual 16-Bit Add / Clip Dreg = BYTEOP3P (Dreg_pair, Dreg_pair) (LO, R)	0xC417 2000— 0xC417 1E3F
Quad 8-Bit Pack Dreg = BYTEPACK (Dreg, Dreg)	0xC418 0000— 0xC418 0E3F
Quad 8-Bit Unpack (Dreg, Dreg) = BYTEUNPACK Dreg_pair	0xC418 4000— 0xC418 4FF8
Quad 8-Bit Unpack (Dreg, Dreg) = BYTEUNPACK Dreg_pair (R)	0xC418 6000— 0xC418 6FF8
Vector Add / Subtract Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg	0xC421 0000— 0xC421 0FFF
Vector Add / Subtract Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg (CO)	0xC421 1000— 0xC421 1FFF
Vector Add / Subtract Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg (S)	0xC421 2000— 0xC421 2FFF

Table C-23. 32-Bit Opcode Instructions (Sheet 29 of 40)

Instruction and Version	Opcode Range
Vector Add / Subtract Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg (SCO)	0xC421 3000— 0xC421 3FFF
Vector Add / Subtract Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg (ASR)	0xC421 8000— 0xC421 8FFF
Vector Add / Subtract Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg (CO, ASR)	0xC421 9000— 0xC421 9FFF
Vector Add / Subtract Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg (S, ASR)	0xC421 A000— 0xC421 AFFF
Vector Add / Subtract Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg (SCO, ASR)	0xC421 B000— 0xC421 BFFF
Vector Add / Subtract Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg (ASL)	0xC421 C000— 0xC421 CFFF
Vector Add / Subtract Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg (CO, ASL)	0xC421 D000— 0xC421 DFFF
Vector Add / Subtract Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg (S, ASL)	0xC421 E000— 0xC421 EFFF
Vector Add / Subtract Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg (SCO, ASL)	0xC421 F000— 0xC421 FFFF
Add Dreg_hi = Dreg_lo + Dreg_lo (NS)	0xC422 0000— 0xC422 0E3F
Add Dreg_hi = Dreg_lo + Dreg_lo (S)	0xC422 2000— 0xC422 2E3F
Add Dreg_hi = Dreg_lo + Dreg_hi (NS)	0xC422 4000— 0xC422 4E3F
Add Dreg_hi = Dreg_lo + Dreg_hi (S)	0xC422 6000— 0xC422 6E3F
Add Dreg_hi = Dreg_hi + Dreg_lo (NS)	0xC422 8000— 0xC422 8E3F
Add Dreg_hi = Dreg_hi + Dreg_lo (S)	0xC422 A000— 0xC422 AE3F
Add Dreg_hi = Dreg_hi + Dreg_hi (NS)	0xC422 C000— 0xC422 CE3F

Table C-23. 32-Bit Opcode Instructions (Sheet 30 of 40)

Instruction and Version	Opcode Range
Add Dreg_hi = Dreg_hi + Dreg_hi (S)	0xC422 E000— 0xC422 EE3F
Subtract Dreg_hi = Dreg_lo – Dreg_lo (NS)	0xC423 0000— 0xC423 0E3F
Subtract Dreg_hi = Dreg_lo – Dreg_lo (S)	0xC423 2000— 0xC423 2E3F
Subtract Dreg_hi = Dreg_lo – Dreg_hi (NS)	0xC423 4000— 0xC423 4E3F
Subtract Dreg_hi = Dreg_lo – Dreg_hi (S)	0xC423 6000— 0xC423 6E3F
Subtract Dreg_hi = Dreg_hi – Dreg_lo (NS)	0xC423 8000— 0xC423 8E3F
Subtract Dreg_hi = Dreg_hi – Dreg_lo (S)	0xC423 A000— 0xC423 AE3F
Subtract Dreg_hi = Dreg_hi – Dreg_hi (NS)	0xC423 C000— 0xC423 CE3F
Subtract Dreg_hi = Dreg_hi – Dreg_hi (S)	0xC423 E000— 0xC423 EE3F
Add/Subtract-Prescale Up Dreg_hi = Dreg + Dreg (RND12)	0xC425 0000— 0xC425 0E3F
Add/Subtract-Prescale Up Dreg_hi = Dreg – Dreg (RND12)	0xC425 4000— 0xC425 4E3F
Add/Subtract-Prescale Down Dreg_hi = Dreg + Dreg (RND20)	0xC425 9000— 0xC425 9E3F
Add/Subtract-Prescale Down Dreg_hi = Dreg – Dreg (RND20)	0xC425 D000— 0xC425 DE3F
Move Register Half A0.H = Dreg_hi	0xC429 0000— 0xC429 0038
Move Register Half A1.H = Dreg_hi	0xC429 8000— 0xC429 8038
Modify-Increment Dreg_hi = (A0 += A1)	0xC42B 403F— 0xC42B 4E00

Table C-23. 32-Bit Opcode Instructions (Sheet 31 of 40)

Instruction and Version	Opcode Range
Round to Half Word Dreg_hi = Dreg (RND)	0xC42C C000— 0xC42C CE38
Negate (Two's-Complement) A1 = – A0	0xC42E 003F
Negate (Two's-Complement) A1 = – A1	0xC42E 403F
Absolute Value A1 = ABS A0	0xC430 003F
Absolute Value A1 = ABS A1	0xC430 403F
Quad 8-Bit Average-Half Word Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (RNDH)	0xC436 0000— 0xC436 0E3F
Quad 8-Bit Average-Half Word Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (RNDH, R)	0xC436 2000— 0xC436 2E3F
Quad 8-Bit Average-Half Word Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (TH)	0xC436 4000— 0xC436 6E3F
Quad 8-Bit Average-Half Word Dreg = BYTEOP2P (Dreg_pair, Dreg_pair) (TH, R)	0xC436 6000— 0xC436 7E3F
Dual 16-Bit Add / Clip Dreg = BYTEOP3P (Dreg_pair, Dreg_pair) (HI)	0xC437 0000— 0xC437 0E3F
Dual 16-Bit Add / Clip Dreg = BYTEOP3P (Dreg_pair, Dreg_pair) (HI, R)	0xC437 2000— 0xC437 1E3F
Arithmetic Shift Dreg_lo = ASHIFT Dreg_lo BY Dreg_lo	0xC600 0000— 0xC600 0E3F
Arithmetic Shift Dreg_lo = ASHIFT Dreg_hi BY Dreg_lo	0xC600 1000— 0xC600 1E3F
Arithmetic Shift Dreg_hi = ASHIFT Dreg_lo BY Dreg_lo	0xC600 2000— 0xC600 2E3F
Arithmetic Shift Dreg_hi = ASHIFT Dreg_hi BY Dreg_lo	0xC600 3000— 0xC600 3E3F
Arithmetic Shift Dreg_lo = ASHIFT Dreg_lo BY Dreg_lo (S)	0xC600 4000— 0xC600 4E3F

Table C-23. 32-Bit Opcode Instructions (Sheet 32 of 40)

Instruction and Version	Opcode Range
Arithmetic Shift Dreg_lo = ASHIFT Dreg_hi BY Dreg_lo (S)	0xC600 5000— 0xC600 5E3F
Arithmetic Shift Dreg_hi = ASHIFT Dreg_lo BY Dreg_lo (S)	0xC600 6000— 0xC600 6E3F
Arithmetic Shift Dreg_hi = ASHIFT Dreg_hi BY Dreg_lo (S)	0xC600 7000— 0xC600 7E3F
Logical Shift Dreg_lo = LSHIFT Dreg_lo BY Dreg_lo	0xC600 8000— 0xC600 8E3F
Logical Shift Dreg_lo = LSHIFT Dreg_hi BY Dreg_lo	0xC600 9000— 0xC600 9E3F
Logical Shift Dreg_hi = LSHIFT Dreg_lo BY Dreg_lo	0xC600 A000— 0xC600 AE3F
Logical Shift Dreg_hi = LSHIFT Dreg_hi BY Dreg_lo	0xC600 B000— 0xC600 BE3F
Vector Arithmetic Shift Dreg = ASHIFT Dreg BY Dreg_lo (V)	0xC601 0000— 0xC601 0E3F
Vector Arithmetic Shift Dreg = ASHIFT Dreg BY Dreg_lo (V, S)	0xC601 4000— 0xC601 4E3F
Vector Logical Shift Dreg = LSHIFT Dreg BY Dreg_lo (V)	0xC601 8000— 0xC601 8E3F
Arithmetic Shift Dreg = ASHIFT Dreg BY Dreg_lo	0xC602 0000— 0xC602 0E3F
Arithmetic Shift Dreg = ASHIFT Dreg BY Dreg_lo (S)	0xC602 4000— 0xC602 4E3F
Logical Shift Dreg = LSHIFT Dreg BY Dreg_lo	0xC602 8000— 0xC602 8E3F
Rotate Dreg = ROT Dreg BY Dreg_lo	0xC602 C000— 0xC602 CE3F
Arithmetic Shift A0 = ASHIFT A0 BY Dreg_lo	0xC603 0000— 0xC603 0038
Arithmetic Shift A1 = ASHIFT A1 BY Dreg_lo	0xC603 1000— 0xC603 1038

Table C-23. 32-Bit Opcode Instructions (Sheet 33 of 40)

Instruction and Version	Opcode Range
Logical Shift A0 = LSHIFT A0 BY Dreg_lo	0xC603 4000— 0xC603 4038
Logical Shift A1 = LSHIFT A1 BY Dreg_lo	0xC603 5000— 0xC603 5038
Rotate A0 = ROT A0 BY Dreg_lo	0xC603 8000— 0xC603 8038
Rotate A1 = ROT A1 BY Dreg_lo	0xC603 9000— 0xC603 9038
Vector Pack Dreg = PACK (Dreg_lo, Dreg_lo)	0xC604 0000— 0xC604 0E3F
Vector Pack Dreg = PACK (Dreg_lo, Dreg_hi)	0xC604 4000— 0xC604 4E3F
Vector Pack Dreg = PACK (Dreg_hi, Dreg_lo)	0xC604 8000— 0xC604 8E3F
Vector Pack Dreg = PACK (Dreg_hi, Dreg_hi)	0xC604 C000— 0xC604 CE3F
Sign Bit Dreg_lo = SIGNBITS Dreg	0xC605 0000— 0xC605 0E07
Sign Bit Dreg_lo = SIGNBITS Dreg_lo	0xC605 4000— 0xC605 4E07
Sign Bit Dreg_lo = SIGNBITS Dreg_hi	0xC605 8000— 0xC605 8E07
Sign Bit Dreg_lo = SIGNBITS A0	0xC606 0000— 0xC606 0E00
Sign Bit Dreg_lo = SIGNBITS A1	0xC606 4000— 0xC606 4E00
Ones-Population Count Dreg_lo = ONES Dreg	0xC606 C000— 0xC606 CE07
Exponent Detection Dreg_lo = EXPADJ (Dreg, Dreg_lo)	0xC607 0000— 0xC607 0E3F
Exponent Detection Dreg_lo = EXPADJ (Dreg, Dreg_lo) (V)	0xC607 4000— 0xC607 4E3F

Table C-23. 32-Bit Opcode Instructions (Sheet 34 of 40)

Instruction and Version	Opcode Range
Exponent Detection Dreg_lo = EXPADJ (Dreg_lo, Dreg_lo)	0xC607 8000— 0xC607 8E3F
Exponent Detection Dreg_lo = EXPADJ (Dreg_hi, Dreg_lo)	0xC607 C000— 0xC607 CE3F
Bit Multiplex BITMUX (Dreg, Dreg, A0) (ASR)	0xC608 0000— 0xC608 003F
Bit Multiplex BITMUX (Dreg, Dreg, A0) (ASL)	0xC608 4000— 0xC608 403F
Compare-Select (VIT_MAX) Dreg_lo = VIT_MAX (Dreg) (ASL)	0xC609 0000— 0xC609 0E07
Compare-Select (VIT_MAX) Dreg_lo = VIT_MAX (Dreg) (ASR)	0xC609 4000— 0xC609 4E07
Compare-Select (VIT_MAX) Dreg = VIT_MAX (Dreg, Dreg) (ASL)	0xC609 8000— 0xC609 8E07
Compare-Select (VIT_MAX) Dreg = VIT_MAX (Dreg, Dreg) (ASR)	0xC609 C000— 0xC609 CE07
Bit Field Extraction Dreg = EXTRACT (Dreg, Dreg_lo) (Z)	0xC60A 0000— 0xC60A 0E3F
Bit Field Extraction Dreg = EXTRACT (Dreg, Dreg_lo) (X)	0xC60A 4000— 0xC60A 4E3F
Bit Field Deposit Dreg = DEPOSIT (Dreg, Dreg)	0xC60A 8000— 0xC60A 8E3F
Bit Field Deposit Dreg = DEPOSIT (Dreg, Dreg) (X)	0xC60A C000— 0xC60A CE3F
Bit Wise Exclusive OR Dreg_lo = CC = BXORSHIFT (A0, Dreg)	0xC60B 0000— 0xC60B 0E38
Bit Wise Exclusive OR Dreg_lo = CC = BXOR (A0, Dreg)	0xC60B 4000— 0xC60B 4E38
Bit Wise Exclusive OR A0 = BXORSHIFT (A0, A1, CC)	0xC60C 0000
Bit Wise Exclusive OR Dreg_lo = CC = BXOR (A0, A1, CC)	0xC60C 4000— 0xC60C 4E00

Table C-23. 32-Bit Opcode Instructions (Sheet 35 of 40)

Instruction and Version	Opcode Range
Byte Align Dreg = ALIGN8 (Dreg, Dreg)	0xC60D 0000— 0xC60D 0E3F
Byte Align Dreg = ALIGN16 (Dreg, Dreg)	0xC60D 4000— 0xC60D 4E3F
Byte Align Dreg = ALIGN24 (Dreg, Dreg)	0xC60D 8000— 0xC60D 8E3F
Arithmetic Shift Dreg_lo = Dreg_lo >>> uimm4	0xC680 0180— 0xC680 0FFF
Arithmetic Shift Dreg_lo = Dreg_hi >>> uimm4	0xC680 1180— 0xC680 1FFF
Arithmetic Shift Dreg_hi = Dreg_lo >>> uimm4	0xC680 2180— 0xC680 2FFF
Arithmetic Shift Dreg_hi = Dreg_hi >>> uimm4	0xC680 3180— 0xC680 3FFF
Arithmetic Shift Dreg_lo = Dreg_lo << uimm4 (S)	0xC680 4000— 0xC680 4E7F
Arithmetic Shift Dreg_lo = Dreg_hi << uimm4 (S)	0xC680 5000— 0xC680 5E7F
Arithmetic Shift Dreg_hi = Dreg_lo << uimm4 (S)	0xC680 6000— 0xC680 6E7F
Arithmetic Shift Dreg_hi = Dreg_hi << uimm4 (S)	0xC680 7000— 0xC680 7E7F
Logical Shift Dreg_lo = Dreg_lo << uimm4	0xC680 8000— 0xC680 8E7F
Logical Shift Dreg_lo = Dreg_lo >> uimm4	0xC680 8180— 0xC680 8FFF
Logical Shift Dreg_lo = Dreg_hi << uimm4	0xC680 9000— 0xC680 9E7F
Logical Shift Dreg_lo = Dreg_hi >> uimm4	0xC680 9180— 0xC680 9FFF
Logical Shift Dreg_hi = Dreg_lo << uimm4	0xC680 A000— 0xC680 AE7F

Table C-23. 32-Bit Opcode Instructions (Sheet 36 of 40)

Instruction and Version	Opcode Range
Logical Shift Dreg_hi = Dreg_lo >> uimm4	0xC680 A180— 0xC680 AFFF
Logical Shift Dreg_hi = Dreg_hi << uimm4	0xC680 B000— 0xC680 BE7F
Logical Shift Dreg_hi = Dreg_hi >> uimm4	0xC680 B180— 0xC680 BFFF
Vector Arithmetic Shift Dreg = Dreg >>> uimm5 (V)	0xC681 0100— 0xC681 0FFF
Vector Arithmetic Shift Dreg = Dreg << uimm5 (V, S)	0xC681 4000— 0xC681 4EFF
Vector Logical Shift Dreg = Dreg << uimm4 (V)	0xC681 8000— 0xC681 8E7F
Vector Logical Shift Dreg = Dreg >> uimm4 (V)	0xC681 8180— 0xC681 8FFF
Arithmetic Shift Dreg = Dreg >>> uimm5	0xC682 0100— 0xC682 0FFF
Arithmetic Shift Dreg = Dreg << uimm5 (S)	0xC682 4000— 0xC680 4EFF
Logical Shift Dreg = Dreg << uimm5	0xC682 8000— 0xC682 8EFF
Logical Shift Dreg = Dreg >> uimm5	0xC682 8100— 0xC682 8FFF
Rotate Dreg = ROT Dreg BY imm6	0xC682 C000— 0xC682 CFFF
Arithmetic Shift A0 = A0 >>> uimm5	0xC683 0100— 0xC683 01F8
Arithmetic Shift A1 = A1 >>> uimm5	0xC683 1100— 0xC683 11F8
Logical Shift A0 = A0 << uimm5	0xC683 4000— 0xC683 40F8
Logical Shift A0 = A0 >> uimm5	0xC683 4100— 0xC683 41F8

Table C-23. 32-Bit Opcode Instructions (Sheet 37 of 40)

Instruction and Version	Opcode Range
Logical Shift A1 = A1 << uimm5	0xC683 5000— 0xC683 50F8
Logical Shift A1 = A1 >> uimm5	0xC683 5100— 0xC683 51F8
Rotate A0 = ROT A0 BY imm6	0xC683 8000— 0xC683 81F8
Rotate A1 = ROT A1 BY imm6	0xC683 9000— 0xC683 91F8
No Op MNOP when issued in parallel with two compatible load/store instructions	0xC803 1800
Zero Overhead Loop Setup 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.	0xE080 0000— 0xE08F 03FF
Zero Overhead Loop Setup LOOP loop_name LC1 LOOP_BEGIN loop_name LOOP_END loop_name ... is mapped to... LSETUP (pcrel5m2, pcrel11m2) LC1 ... where the address of LOOP_BEGIN determines pcrel5m2, and the address of LOOP_END determines pcrel11m2.	0xE090 0000— 0xE09F 03FF
Zero Overhead Loop Setup 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.	0xE0A0 0000— 0xE0AF F3FF

Table C-23. 32-Bit Opcode Instructions (Sheet 38 of 40)

Instruction and Version	Opcode Range
Zero Overhead Loop Setup 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.	0xE0B0 0000— 0xE0BF F3FF
Zero Overhead Loop Setup 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.	0xE0E0 0000— 0xE0AF F3FF
Zero Overhead Loop Setup 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.	0xE0F0 0000— 0xE0FF F3FF
Load Immediate reg_lo = uimm16	0xE100 0000— 0xE11F FFFF
Load Immediate reg = imm16 (X)	0xE120 0000— 0xE13F FFFF
Load Immediate reg_hi = uimm16	0xE140 0000— 0xE15F FFFF
Load Immediate reg = uimm16 (Z)	0xE180 0000— 0xE19F FFFF
Jump JUMP.L pcrel25m2	0xE200 0000— 0xE2FF FFFF
Call CALL pcrel25m2	0xE300 0000— 0xE3FF FFFF

Table C-23. 32-Bit Opcode Instructions (Sheet 39 of 40)

Instruction **and Version**	**Opcode** **Range**
Load Data Register Dreg = [Preg + uimm17m4]	0xE400 0000— 0xE4EF 7FFF
Load Data Register Dreg = [Preg – uimm17m4]	0xE400 8000— 0xE43F FFFF
Load Half Word, Zero Extended Dreg = W [Preg + uimm16m2] (Z)	0xE440 0000— 0xE47F 8FFF
Load Half Word, Zero Extended Dreg = W [Preg – uimm16m2] (Z)	0xE440 8000— 0xE47F FFFF
Load Byte, Zero Extended Dreg = B [Preg + uimm15] (Z)	0xE480 0000— 0xE4BF 7FFF
Load Byte, Zero Extended Dreg = B [Preg – uimm15] (Z)	0xE480 8000— 0xE4BF FFFF
Load Pointer Register Preg = [Preg + uimm17m4]	0xE500 0000— 0xE53F 7FFF
Load Pointer Register Preg = [Preg – uimm17m4]	0xE500 8000— 0xE53F FFFF
Load Half Word, Sign Extended Dreg = W [Preg + uimm16m2] (X)	0xE540 0000— 0xE57F 8FFF
Load Half Word, Sign Extended Dreg = W [Preg – uimm16m2] (X)	0xE540 8000— 0xE57F FFFF
Load Byte, Sign Extended Dreg = B [Preg + uimm15] (X)	0xE580 0000— 0xE5BF 7FFF
Load Byte, Sign Extended Dreg = B [Preg – uimm15] (X)	0xE580 8000— 0xE5BF FFFF
Store Data Register [Preg + uimm17m4] = Dreg	0xE600 0000— 0xE63F 7FFF
Store Data Register [Preg – uimm17m4] = Dreg	0xE600 8000— 0xE63F FFFF
Store Low Data Register Half W [Preg + uimm16m2] = Dreg	0xE640 0000— 0xE67F 7FFF
Store Low Data Register Half W [Preg – uimm16m2] = Dreg	0xE640 8000— 0xE67F FFFF

Table C-23. 32-Bit Opcode Instructions (Sheet 40 of 40)

Instruction and Version	Opcode Range
Store Byte B [Preg + uimm15] = Dreg	0xE680 0000— 0xE6BF 7FFF
Store Byte B [Preg – uimm15] = Dreg	0xE680 8000— 0xE6BF FFFF
Store Pointer Register [Preg + uimm17m4] = Preg	0xE700 0000— 0xE7EF 8FFF
Store Pointer Register [Preg – uimm17m4] = Preg	0xE700 8000— 0xE73F FFFF
Linkage LINK uimm18m4	0xE800 0000— 0xE800 FFFF
Linkage UNLINK	0xE801 0000

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.

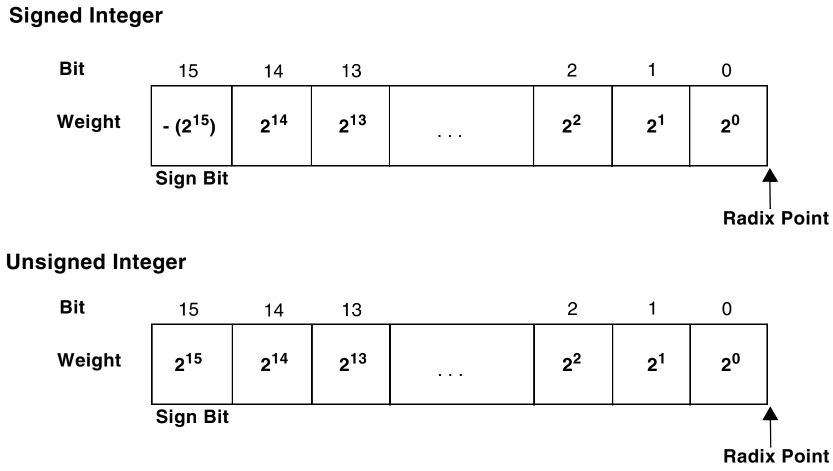
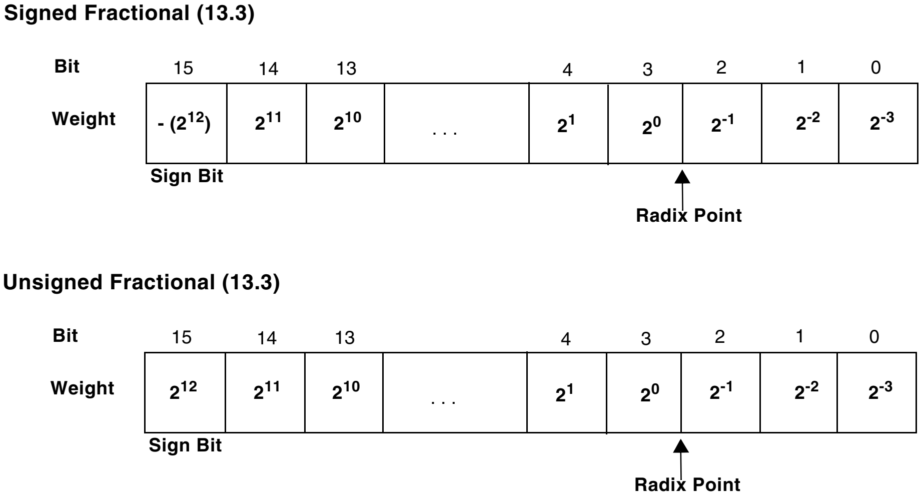


Figure D-1. Integer Format

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.



Signed Fractional (13.3)

Bit	15	14	13		4	3	2	1	0
Weight	$-(2^{12})$	2^{11}	2^{10}	. . .	2^{1}	2^{0}	2^{-1}	2^{-2}	2^{-3}

Unsigned Fractional (13.3)

Bit	15	14	13		4	3	2	1	0
Weight	2^{12}	2^{11}	2^{10}	. . .	2^{1}	2^{0}	2^{-1}	2^{-2}	2^{-3}

Figure D-2. Example of Fractional Format

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

Format	# of Integer Bits	# of Fractional Bits	Max Positive Value (0x7FFF) In Decimal	Max Negative Value (0x8000) In Decimal	Value of 1 LSB (0x0001) In Decimal
1.15	1	15	0.999969482421875	–1.0	0.000030517578125
2.14	2	14	1.999938964843750	–2.0	0.000061035156250
3.13	3	13	3.999877929687500	–4.0	0.000122070312500
4.12	4	12	7.999755859375000	–8.0	0.000244140625000
5.11	5	11	15.999511718750000	–16.0	0.000488281250000
6.10	6	10	31.999023437500000	–32.0	0.000976562500000
7.9	7	9	63.998046875000000	–64.0	0.001953125000000
8.8	8	8	127.996093750000000	–128.0	0.003906250000000
9.7	9	7	255.992187500000000	–256.0	0.007812500000000
10.6	10	6	511.984375000000000	–512.0	0.015625000000000
11.5	11	5	1023.968750000000000	–1024.0	0.031250000000000
12.4	12	4	2047.937500000000000	–2048.0	0.062500000000000
13.3	13	3	4095.875000000000000	–4096.0	0.125000000000000
14.2	14	2	8191.750000000000000	–8192.0	0.250000000000000
15.1	15	1	16383.500000000000000	–16384.0	0.500000000000000
16.0	16	0	32767.000000000000000	–32768.0	1.000000000000000

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.

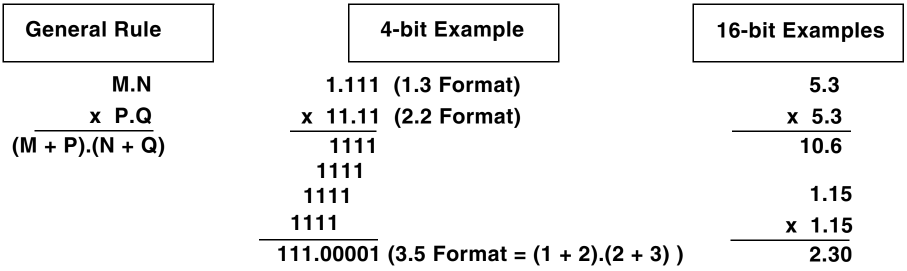


Figure D-3. Format of Multiplier Result

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.

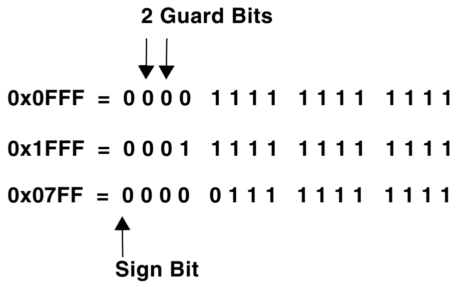
When both operands are in 1.15 format, the result is 2.30 (30 fractional bits). A left shift causes the multiplier result to be 1.31 which can be rounded to 1.15. Thus, if you use a signed fractional data format, it is most convenient to use the 1.15 format.

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.



```
0x0FFF = 0000 1111 1111 1111
0x1FFF = 0001 1111 1111 1111
0x07FF = 0000 0111 1111 1111
```

To detect bit growth into two guard bits, set SB = –2

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 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.

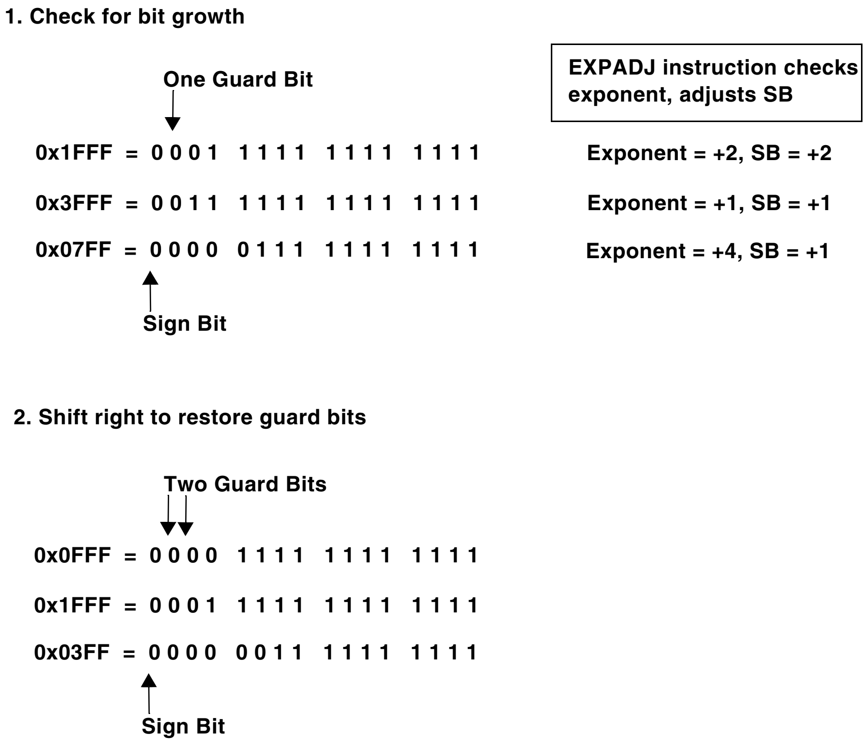

1. Check for bit growth
One Guard Bit
EXPADJ instruction checks exponent, adjusts SB
0x1FFF = 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
Exponent = +2, SB = +2
0x3FFF = 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Exponent = +1, SB = +1
0x07FF = 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1
Exponent = +4, SB = +1
Sign Bit
2. Shift right to restore guard bits
Two Guard Bits
0x0FFF = 0 0 0 0 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
0x03FF = 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
Sign Bit

Figure D-5. Block Floating-point Adjustment

I INDEX