

ADSP-TS101 TigerSHARC® Processor Programming Reference

Revision 1.1, February 2005

Part Number
82-001997-01

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



Copyright Information

© 2005 Analog Devices, Inc., ALL RIGHTS RESERVED. This document may not be reproduced in any form without prior, express written consent from Analog Devices, Inc.

Printed in the USA.

Disclaimer

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

Trademark and Service Mark Notice

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

SuperScalar is a trademark of Analog Devices, Inc.

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

CONTENTS

PREFACE

Purpose of This Manual	xvii
Intended Audience	xvii
Manual Contents	xviii
What's New in This Manual	xix
Technical or Customer Support	xx
Supported Processors	xx
Product Information	xxi
MyAnalog.com	xxii
Processor Product Information	xxii
Related Documents	xxiii
Online Technical Documentation	xxiv
Accessing Documentation From VisualDSP++	xxv
Accessing Documentation From Windows	xxv
Accessing Documentation From the Web	xxvi
Printed Manuals	xxvi
VisualDSP++ Documentation Set	xxvi
Hardware Tools Manuals	xxvi

CONTENTS

Processor Manuals	xxvi
Data Sheets	xxvii
Conventions	xxviii

INTRODUCTION

DSP Architecture	1-6
Compute Blocks	1-8
Arithmetic Logic Unit (ALU)	1-9
Multiply Accumulator (Multiplier)	1-11
Bit Wise Barrel Shifter (Shifter)	1-11
Integer Arithmetic Logic Unit (IALU)	1-12
Program Sequencer	1-13
Quad Instruction Execution	1-15
Relative Addresses for Relocation	1-16
Nested Call and Interrupt	1-16
Context Switching	1-16
Internal Memory and Other Internal Peripherals	1-16
Internal Buses	1-17
Internal Transfer	1-18
Data Accesses	1-18
Quad Data Access	1-18
Booting	1-19
Scalability and Multiprocessing	1-19
Emulation and Test Support	1-20

Instruction Line Syntax and Structure	1-20
Instruction Notation Conventions	1-22
Unconditional Execution Support	1-23
Conditional Execution Support	1-24
Instruction Parallelism Rules	1-24
General Restriction	1-36
Compute Block Instruction Restrictions	1-37
IALU Instruction Restrictions	1-39
Sequencer Instruction Restrictions	1-45

COMPUTE BLOCK REGISTERS

Register File Registers	2-5
Compute Block Selection	2-7
Register Width Selection	2-8
Operand Size and Format Selection	2-10
Registers File Syntax Summary	2-13
Numeric Formats	2-16
IEEE Single-Precision Floating-Point Data Format	2-16
Extended Precision Floating-Point Format	2-19
Fixed-Point Formats	2-19

CONTENTS

ALU

ALU Operations	3-5
ALU Instruction Options	3-7
Signed/Unsigned Option	3-8
Saturation Option	3-8
Extension (ABS) Option	3-9
Truncation Option	3-9
Return Zero (MAX/MIN) Option	3-10
Fractional/Integer Option	3-11
ALU Execution Status	3-11
AN — ALU Negative	3-13
AV — ALU Overflow	3-13
AI — ALU Invalid	3-14
AC — ALU Carry	3-14
ALU Execution Conditions	3-14
ALU Static Flags	3-15
ALU Examples	3-16
Example Parallel Addition of Byte Data	3-18
Example Sideways Addition of Byte Data	3-19
Example Parallel Result (PR) Register Usage	3-19
CLU Examples	3-21
CLU Data Types and Sizes	3-22
TMAX Function	3-23
Trellis Function	3-24

Despread Function	3-26
CLU Execution Status	3-27
ALU Instruction Summary	3-28

MULTIPLIER

Multiplier Operations	4-4
Multiplier Instruction Options	4-8
Signed/Unsigned Option	4-10
Fractional/Integer Option	4-10
Saturation Option	4-11
Truncation Option	4-12
Clear/Round Option	4-14
Complex Conjugate Option	4-16
Multiplier Result Overflow (MR4) Register	4-17
Multiplier Execution Status	4-18
Multiplier Execution Conditions	4-20
Multiplier Static Flags	4-21
Multiplier Examples	4-21
Multiplier Instruction Summary	4-23

SHIFTER

Shifter Operations	5-3
Logical Shift Operation	5-5
Arithmetic Shift Operation	5-6
Bit Manipulation Operations	5-7

CONTENTS

Bit Field Manipulation Operations	5-8
Bit Field Conversion Operations	5-11
Bit Stream Manipulation Operations	5-11
Shifter Instruction Options	5-14
Sign Extended Option	5-15
Zero Filled Option	5-15
Shifter Execution Status	5-15
Shifter Execution Conditions	5-16
Shifter Static Flags	5-17
Shifter Examples	5-17
Shifter Instruction Summary	5-19

IALU

IALU Operations	6-5
IALU Arithmetic, Logical, and Function Operations	6-5
IALU Instruction Options	6-6
Integer Data	6-7
Signed/Unsigned Option	6-8
Circular Buffer Option	6-8
Bit Reverse Option	6-9
Computed Jump Option	6-9
IALU Execution Status	6-10
JN/KN–IALU Negative	6-11
JV/KV–IALU Overflow	6-11
JC/KC–IALU Carry	6-11

IALU Execution Conditions	6-12
IALU Static Flags	6-13
IALU Data Addressing and Transfer Operations	6-13
Direct and Indirect Addressing	6-14
Normal, Merged, and Broadcast Memory Accesses	6-16
Data Alignment Buffer (DAB) Accesses	6-23
Circular Buffer Addressing	6-27
Bit Reverse Addressing	6-31
Universal Register Transfer Operations	6-35
Immediate Extension Operations	6-36
IALU Examples	6-37
IALU Instruction Summary	6-39

PROGRAM SEQUENCER

Sequencer Operations	7-7
Conditional Execution	7-12
Branching Execution	7-16
Looping Execution	7-19
Interrupting Execution	7-20
Instruction Pipeline Operations	7-26
Instruction Alignment Buffer (IAB)	7-31
Branch Target Buffer (BTB)	7-34
Conditional Branch Effects on Pipeline	7-44

CONTENTS

Dependency and Resource Effects on Pipeline	7-55
Stall From Compute Block Dependency	7-56
Stall from Bus Conflict	7-59
Stall From Compute Block Load Dependency	7-62
Stall From IALU Load Dependency	7-63
Stall From Load (From External Memory) Dependency	7-64
Stall From Conditional IALU Load Dependency	7-64
Interrupt Effects on Pipeline	7-66
Interrupt During Conditional Instruction	7-68
Interrupt During Interrupt Disable Instruction	7-70
Exception Effects on Pipeline	7-72
Sequencer Examples	7-72
Sequencer Instruction Summary	7-76

INSTRUCTION SET

ALU Instructions	8-2
Add/Subtract	8-3
Add/Subtract With Carry/Borrow	8-6
Average	8-8
Absolute Value/Absolute Value of Sum or Difference	8-10
Negate	8-13
Maximum/Minimum	8-14
Viterbi Maximum/Minimum	8-17
Increment/Decrement	8-20
Compare	8-22

Clip	8-24
Sum	8-26
Ones Counting	8-28
Parallel Result Register	8-29
Bit FIFO Increment	8-30
Parallel Absolute Value of Difference	8-32
Sideways Sum	8-34
Add/Subtract (Dual Operation)	8-36
Pass	8-37
Logical AND/AND NOT/OR/XOR/NOT	8-38
Expand	8-40
Compact	8-45
Merge	8-49
Add/Subtract (Floating-Point)	8-51
Average (Floating-Point)	8-53
Maximum/Minimum (Floating-Point)	8-55
Absolute Value (Floating-Point)	8-57
Negate (Floating-Point)	8-60
Compare (Floating-Point)	8-62
Floating- to Fixed-Point Conversion	8-64
Fixed- to Floating-Point Conversion	8-66
Floating-Point Normal to Extended Word Conversion	8-68
Floating-Point Extended to Normal Word Conversion	8-70
Clip (Floating-Point)	8-72

CONTENTS

Copysign (Floating-Point)	8-74
Scale (Floating-Point)	8-76
Pass (Floating-Point)	8-78
Reciprocal (Floating-Point)	8-80
Reciprocal Square Root (Floating-Point)	8-82
Mantissa (Floating-Point)	8-85
Logarithm (Floating-Point)	8-87
Add/Subtract (Dual Operation, Floating-Point)	8-89
CLU Instructions	8-91
Trellis Maximum (CLU)	8-92
Maximum (CLU)	8-99
Trellis Registers (CLU)	8-104
Despread (CLU)	8-106
Add/Compare/Select (CLU)	8-113
Permute (Byte Word, CLU)	8-117
Permute (Short Word, CLU)	8-119
Multiplier Instructions	8-121
Multiply (Normal Word)	8-122
Multiply-Accumulate (Normal Word)	8-125
Multiply-Accumulate/Move (Dual Operation, Normal Word)	8-130
Multiply (Quad-Short Word)	8-138
Multiply-Accumulate (Quad-Short Word)	8-141
Multiply-Accumulate (Dual Operation, Quad-Short Word)	8-146

Complex Multiply-Accumulate (Short Word)	8-152
Complex Multiply-Accumulate/Move (Dual Operation, Short Word)	8-156
Multiply (Floating-Point, Normal/Extended Word)	8-163
Multiplier Result Register	8-165
Compact Multiplier Result	8-171
Shifter Instructions	8-175
Arithmetic/Logical Shift	8-176
Rotate	8-179
Field Extract	8-181
Field Deposit	8-183
Field/Bit Mask	8-185
Get Bits	8-187
Put Bits	8-189
Bit Test	8-191
Bit Clear/Set/Toggle	8-192
Extract Leading Zeros	8-194
Extract Exponent	8-195
XSTAT/YSTAT Register	8-196
Block Floating-Point	8-197
BFOTMP Register	8-199
IALU (Integer) Instructions	8-200
Add/Subtract (Integer)	8-202
Add/Subtract With Carry/Borrow (Integer)	8-204
Average (Integer)	8-206

CONTENTS

Compare (Integer)	8-208
Maximum/Minimum (Integer)	8-210
Absolute Value (Integer)	8-212
Logical AND/AND NOT/OR/XOR/NOT (Integer)	8-213
Arithmetic Shift/Logical Shift (Integer)	8-215
Left Rotate/Right Rotate (Integer)	8-217
IALU (Load/Store/Transfer) Instructions	8-218
Universal Register Load (Data Addressing)	8-220
Universal Register Store (Data Addressing)	8-221
Data Register Load and DAB Operation (Data Addressing)	8-222
Data Register Store (Data Addressing)	8-224
Universal Register Transfer	8-226
Sequencer Instructions	8-228
Jump/Call	8-230
Computed Jump/Call	8-232
Return (from Interrupt)	8-234
Reduce (Interrupt to Subroutine)	8-236
If – Do (Conditional Execution)	8-237
If – Else (Conditional Sequencing and Execution)	8-238
Static Flag Registers	8-239
Idle	8-240
BTB Invalid	8-241

Trap	8-242
Emulator Trap	8-243
No Operation	8-244

QUICK REFERENCE

ALU Quick Reference	A-2
Multiplier Quick Reference	A-6
Shifter Quick Reference	A-8
IALU Quick Reference	A-10
Sequencer Quick Reference	A-13

REGISTER/BIT DEFINITIONS

INSTRUCTION DECODE

Instruction Structure	C-1
Compute Block Instruction Format	C-3
ALU Instructions	C-4
ALU Fixed-Point, Arithmetic and Logical Instructions (CU=00)	C-5
ALU Fixed-Point, Data Conversion Instructions (CU=01)	C-7
ALU Floating-Point, Arithmetic and Logical Instructions (CU=01)	C-10
CLU Instructions	C-12
Multiplier Instructions	C-14

CONTENTS

Shifter Instructions	C-18
Shifter Instructions Using Single Normal-Word Operands and Single Register	C-18
Shifter Instructions Using Single Long-Word or Dual Normal-Word Operands and Dual Register	C-19
Shifter Instructions Using Short or Byte Operands and Single or Dual Registers	C-20
Shifter Instructions Using Single Operand	C-22
IALU (Integer) Instruction Format	C-24
IALU Move Instruction Format	C-25
IALU Load Data Instruction Format	C-27
IALU Load/Store Instruction Format	C-28
IALU Immediate Extension Format	C-32
Sequencer Instruction Format	C-33
Sequencer Flow Control Instructions	C-33
Sequencer Direct Jump/Call Instruction Format	C-34
Sequencer Indirect Jump Instruction Format	C-36
Condition Codes	C-39
Compute Block Conditions	C-39
IALU Conditions	C-40
Sequencer and External Conditions	C-40
Sequencer Immediate Extension Format	C-41
Miscellaneous Instruction Format	C-42

INDEX

PREFACE

Thank you for purchasing and developing systems using TigerSHARC® processors from Analog Devices.

Purpose of This Manual

The *ADSP-TS101 TigerSHARC Processor Programming Reference* contains information about the DSP architecture and DSP assembly language for TigerSHARC processors. These are 32-bit, fixed- and floating-point digital signal processors from Analog Devices for use in computing, communications, and consumer applications.

The manual provides information on how assembly instructions execute on the TigerSHARC processor's architecture along with reference information about DSP operations.

Intended Audience

The primary audience for this manual is a programmer who is familiar with Analog Devices processors. This manual assumes that the audience has a working knowledge of the appropriate processor 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 the appropriate hardware reference manuals and data sheets) that describe your target architecture.

Manual Contents

The manual consists of:

- Chapter 1, “[Introduction](#)”
Provides a general description of the DSP architecture, instruction slot/line syntax, and instruction parallelism rules.
- Chapter 2, “[Compute Block Registers](#)”
Provides a description of the compute block register file, register naming syntax, and numeric formats.
- Chapter 3, “[ALU](#)”
Provides a description of the arithmetic logic unit (ALU) and communications logic unit (CLU) operation, includes ALU/CLU instruction examples, and provides the ALU instruction summary.
- Chapter 4, “[Multiplier](#)”
Provides a description of the multiply-accumulator (multiplier) operation, includes multiplier instruction examples, and provides the multiplier instruction summary.
- Chapter 5, “[Shifter](#)”
Provides a description of the bit wise, barrel shifter (shifter) operation, includes shifter instruction examples, and provides the shifter instruction summary.
- Chapter 6, “[IALU](#)”
Provides a description of the integer arithmetic logic unit (IALU) and data alignment buffer (DAB) operation, includes IALU instruction examples, and provides the IALU instruction summary.
- Chapter 7, “[Program Sequencer](#)”
Provides a description of the program sequencer operation, the instruction alignment buffer (IAB), the branch target buffer (BTB), and the instruction pipeline. This chapter also includes a program sequencer instruction summary.

- Chapter 8, “[Instruction Set](#)”
Describes the ADSP-TS101 processor instruction set in detail, starting with an overview of the instruction line and instruction types.
- Appendix A, “[Quick Reference](#)”
Contains a concise description of the ADSP-TS101 processor assembly language. It is intended to be used as an assembly programming reference.
- Appendix B, “[Register/Bit Definitions](#)”
Provides register and bit name definitions to be used in ADSP-TS101 processor programs.
- Appendix C, “[Instruction Decode](#)”
Identifies operation codes (opcodes) for instructions. Use this chapter to learn how to construct opcodes.



This programming reference is a companion document to the *ADSP-TS101 TigerSHARC Processor Hardware Reference*.

What’s New in This Manual

Revision 1.1 of the *ADSP-TS101 TigerSHARC Processor Programming Reference* corrects and closes all open Tool Anomaly Reports (TARs) against this manual, adds figure titles that were missing, and updates Web site and contact numbers. These changes affect the preface, various chapters, appendices, and the index.

Technical or Customer Support

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

- Visit the Embedded Processing and DSP products Web site at <http://www.analog.com/processors/technicalSupport>
- E-mail tools questions to dsptools.support@analog.com
- E-mail processor questions to embedded.support@analog.com
dsp.support@analog.com
- 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++®.

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 processors:

ADSP-TS101, ADSP-TS201, ADSP-TS202, and ADSP-TS203

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 processors:

ADSP-21020, ADSP-21060, ADSP-21061, ADSP-21062,
ADSP-21065L, ADSP-21160, ADSP-21161, ADSP-21261,
ADSP-21262, ADSP-21266, ADSP-21267, ADSP-21363, ADSP-21364,
and ADSP-21365

Blackfin® (ADSP-BFxxx) Processors

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

ADSP-BF531, ADSP-BF532 (formerly ADSP-21532), ADSP-BF533,
ADSP-BF535 (formerly ADSP-21535), ADSP-BF561, AD6532, and
AD90747

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.

Product Information

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 the Analog Devices 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
embedded.support@analog.com
dsp.support@analog.com
- 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](ftp://ftp.analog.com) (or [ftp 137.71.25.69](ftp://137.71.25.69))
<ftp://ftp.analog.com>

Related Documents

The following publications that describe the ADSP-TS101 TigerSHARC processor (and related processors) can be ordered from any Analog Devices sales office:

- *ADSP-TS101S TigerSHARC Embedded Processor Data Sheet*
- *ADSP-TS101 TigerSHARC Processor Hardware Reference*
- *ADSP-TS101 TigerSHARC Processor Programming Reference*

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

- *VisualDSP++ User's Guide for TigerSHARC Processors*
- *VisualDSP++ C/C++ Compiler and Library Manual for TigerSHARC Processors*
- *VisualDSP++ Assembler and Preprocessor Manual for TigerSHARC Processors*

Product Information

- *VisualDSP++ Linker and Utilities Manual for TigerSHARC Processors*
- *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/technical_library

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.

Product Information

Accessing Documentation From the Web

Download manuals at the following Web site:

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

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 appear within curly brackets and separated by vertical bars; read the example as <i>this</i> or <i>that</i> . One or the other is required.
[this that]	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional <i>this</i> or <i>that</i> .
[this,...]	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipse; read the example as an optional comma-separated list of <i>this</i> .
.SECTION	Commands, directives, keywords, and feature names are in text with letter gothic font.
<i>filename</i>	Non-keyword placeholders appear in text with italic style format.
	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 device users. In the online version of this book, the word Warning appears instead of this symbol.

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

Conventions

1 INTRODUCTION

The *ADSP-TS101 TigerSHARC Processor Programming Reference* describes the Digital Signal Processor (DSP) architecture and instruction set. These descriptions provide the information required for programming TigerSHARC processor systems. This chapter introduces programming concepts for the DSP with the following information:

- “[DSP Architecture](#)” on page 1-6
- “[Instruction Line Syntax and Structure](#)” on page 1-20
- “[Instruction Parallelism Rules](#)” on page 1-24

The TigerSHARC processor is a 128-bit, high performance, next generation version of the ADSP-2106x SHARC DSP. The TigerSHARC processor sets a new standard of performance for digital signal processors, combining multiple computation units for floating-point and fixed-point processing as well as very wide word widths. The TigerSHARC processor maintains a ‘system-on-a-chip’ scalable computing design philosophy, including 6M bit of on-chip SRAM, integrated I/O peripherals, a host processor interface, DMA controllers, link ports, and shared bus connectivity for glueless MDSP (Multi Digital Signal Processing).

In addition to providing unprecedented performance in DSP applications in raw MFLOPS and MIPS, the TigerSHARC processor boosts performance measures such as MFLOPS/Watt and MFLOPS/square inch in multiprocessing applications.

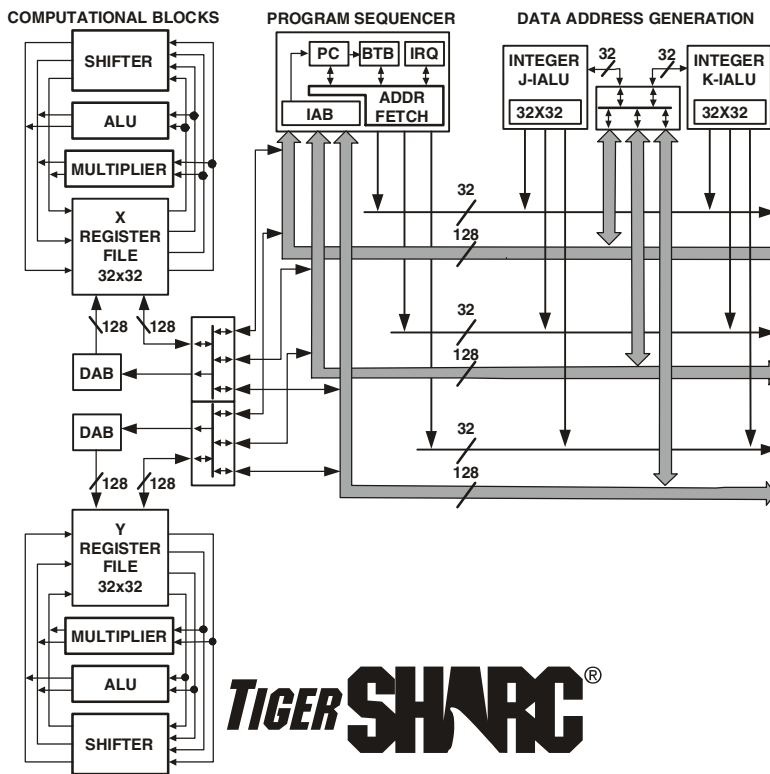


Figure 1-1. ADSP-TS101 TigerSHARC Processor Core Diagram

As shown in [Figure 1-1](#) and [Figure 1-2](#), the processor has the following architectural features:

- Dual computation blocks—X and Y—each consisting of a multiplier, ALU, shifter, and a 32-word register file
- Dual integer ALUs—J and K—each containing a 32-bit IALU and 32-word register file

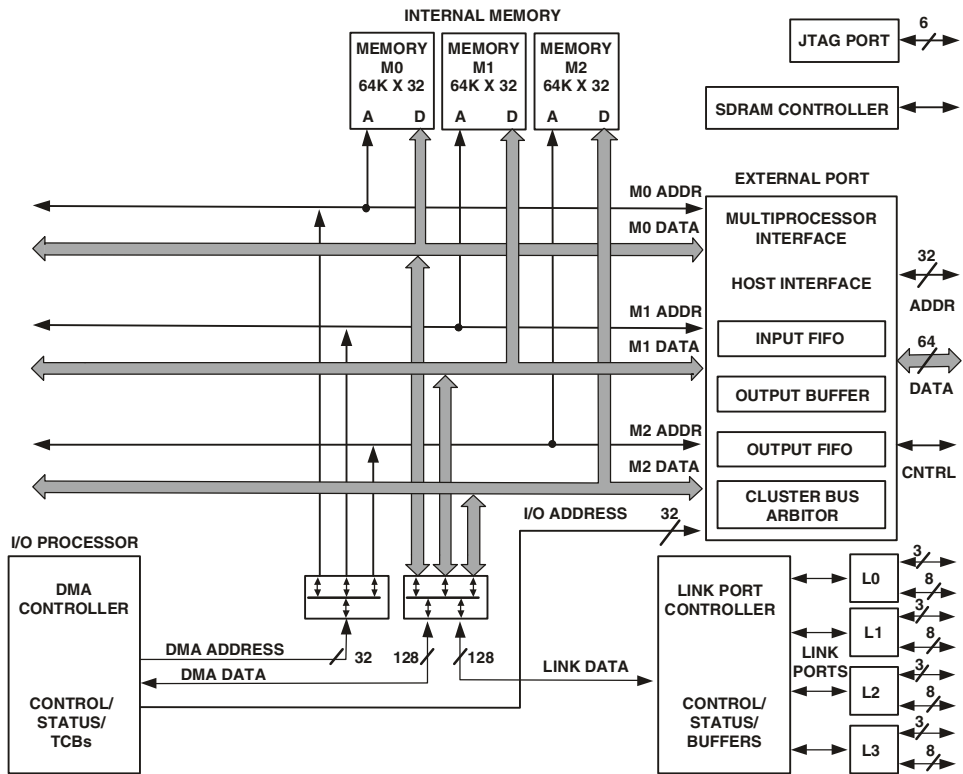


Figure 1-2. ADSP-TS101 TigerSHARC Processor Peripherals Diagram

- Program sequencer—Controls the program flow and contains an instruction alignment buffer (IAB) and a branch target buffer (BTB)
- Three 128-bit buses providing high bandwidth connectivity between all blocks
- External port interface including the host interface, SDRAM controller, static pipelined interface, four DMA channels, four link ports (each with two DMA channels), and multiprocessing support

- 6M bits of internal memory organized as three blocks—M0, M1 and M2—each containing 16K rows and 128 bits wide (a total of 2M bit).
- Debug features
- JTAG Test Access Port

The TigerSHARC processor external port provides an interface to external memory, to memory-mapped I/O, to host processor, and to additional TigerSHARC processors. The external port performs external bus arbitration and supplies control signals to shared, global memory and I/O devices.

[Figure 1-3](#) illustrates a typical single-processor system. A multiprocessor system is illustrated in [Figure 1-4 on page 1-6](#) and is discussed later in “[Scalability and Multiprocessing](#)” on page 1-19.

The TigerSHARC processor includes several features that simplify system development. The features lie in three key areas:

- Support of IEEE floating-point formats
- IEEE 1149.1 JTAG serial scan path and on-chip emulation features
- Architectural features supporting high-level languages and operating systems

The features of the TigerSHARC processor architecture that directly support high-level language compilers and operating systems include:

- Simple, orthogonal instruction allowing the compiler to efficiently use the multi-instruction slots
- General-purpose data and IALU register files
- 32- and 40-bit floating-point and 8-, 16-, 32-, and 64-bit fixed-point native data types

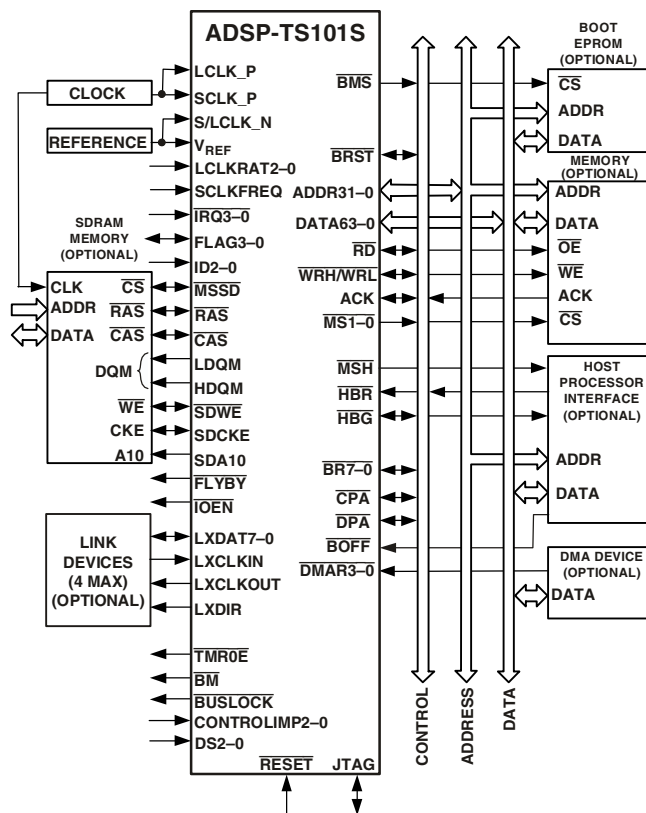


Figure 1-3. Single Processor Configuration

- Large address space
- Immediate address modify fields
- Easily supported relocatable code and data
- Fast save and restore of processor registers onto internal memory stacks

DSP Architecture

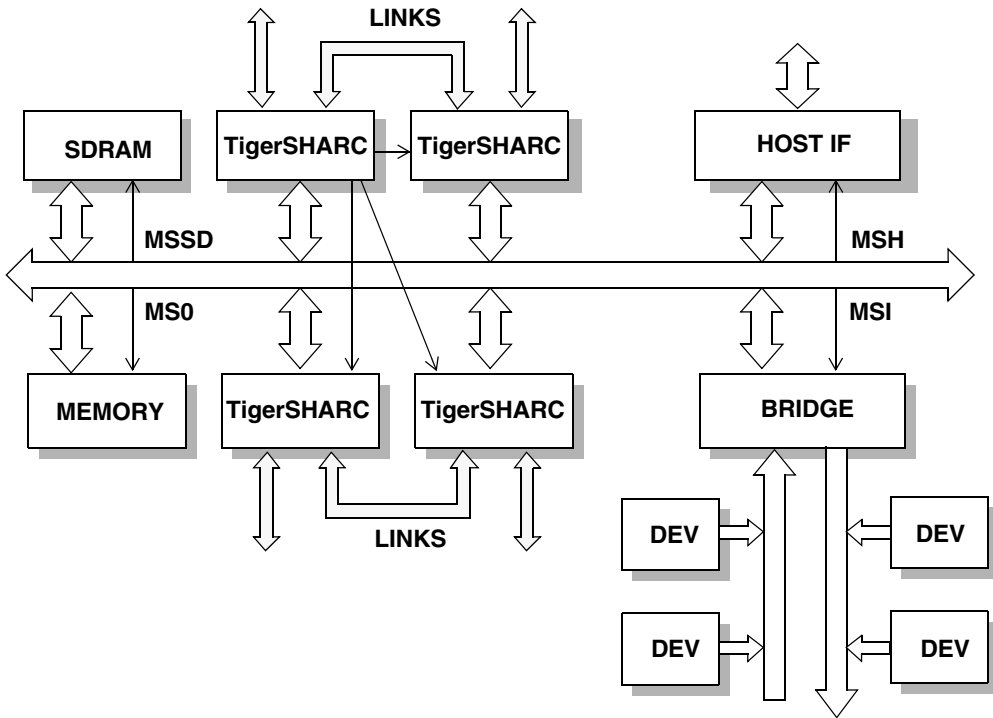


Figure 1-4. Multiprocessing Cluster Configuration

DSP Architecture

As shown in [Figure 1-1 on page 1-2](#) and [Figure 1-2 on page 1-3](#), the DSP architecture consists of two divisions: the DSP core (where instructions execute) and the I/O peripherals (where data is stored and off-chip I/O is processed). The following discussion provides a high-level description of the DSP core and peripherals architecture. More detail on the core appears in other sections of this reference. For more information on I/O peripherals, see the *ADSP-TS101 TigerSHARC Processor Hardware Reference*.

High performance is facilitated by the ability to execute up to four 32-bit wide instructions per cycle. The TigerSHARC processor uses a variation of a *Static Superscalar*[™] architecture to allow the programmer to specify which instructions are executed in parallel in each cycle. The instructions do not have to be aligned in memory so that program memory is not wasted.

The 6M bit internal memory is divided into three 128-bit wide memory blocks. Each of the three internal address/data bus pairs connect to one of the three memory blocks. The three memory blocks can be used for triple accesses every cycle where each memory block can access up to four, 32-bit words in a cycle.

The external port cluster bus is 64 bits wide. The high I/O bandwidth complements the high processing speeds of the core. To facilitate the high clock rate, the TigerSHARC processor uses a pipelined external bus with programmable pipeline depth for interprocessor communications and for Synchronous SRAM and DRAM (SSRAM and SDRAM).

The four link ports support point-to-point high bandwidth data transfers. Link ports have hardware supported two-way communication.

The processor operates with a two cycle arithmetic pipeline. The branch pipeline is two to six cycles. A branch target buffer (BTB) is implemented to reduce branch delay. The two identical computation units support floating-point as well as fixed-point arithmetic.

During compute intensive operations, one or both integer ALUs compute or generate addresses for fetching up to two quad operands from two memory blocks, while the program sequencer simultaneously fetches the next quad instruction from the third memory block. In parallel, the computation units can operate on previously fetched operands while the sequencer prepares for a branch.

While the core processor is doing the above, the DMA channels can be replenishing the internal memories in the background with quad data from either the external port or the link ports.

DSP Architecture

The processing core of the TigerSHARC processor reaches exceptionally high DSP performance through using these features:

- Computation pipeline
- Dual computation units
- Execution of up to four instructions per cycle
- Access of up to eight words per cycle from memory

The two computation units (compute blocks) perform up to 6 floating-point or 24 fixed-point operations per cycle.

Each multiplier and ALU unit can execute four 16-bit fixed-point operations per cycle, using Single-Instruction, Multiple-Data (SIMD) operation. This operation boosts performance of critical imaging and signal processing applications that use fixed-point data.

Compute Blocks

The TigerSHARC processor core contains two computation units called *compute blocks*. Each compute block contains a register file and three independent computation units—an ALU, a multiplier, and a shifter. For meeting a wide variety of processing needs, the computation units process data in several fixed- and floating-point formats listed here and shown in [Figure 1-5](#):

- **Fixed-point format**
These include 64-bit long word, 32-bit normal word, 16-bit short word, and 8-bit byte word. For short word fixed-point arithmetic,

quad parallel operations on quad-aligned data allow fast processing of array data. Byte operations are also supported for octal-aligned data.

- **Floating-point format**

These include 32-bit normal word and 40-bit extended word. Floating-point operations are single or extended precision. The normal word floating-point format is the standard IEEE format, and the 40-bit extended-precision format occupies a double word (64 bits) with eight additional LSBs of mantissa for greater accuracy.

Each compute block has a general-purpose, multi-port, 32-word data register file for transferring data between the computation units and the data buses and storing intermediate results. All of these registers can be accessed as single-, dual-, or quad-aligned registers. For more information on the register file, see [“Compute Block Registers” on page 2-1](#).

Arithmetic Logic Unit (ALU)

The ALU performs arithmetic operations on fixed-point and floating-point data and logical operations on fixed-point data. The source and destination of most ALU operations is the compute block register file.

On the ADSP-TS101 processor, the ALU includes a special sub-block, which is referred to as the *communications logic unit (CLU)*. The CLU instructions are designed to support different algorithms used for communications applications. The algorithms that are supported by the CLU instructions are:

- Viterbi Decoding
- Turbo-code Decoding
- Despreading for code-division multiple access (CDMA) systems

DSP Architecture

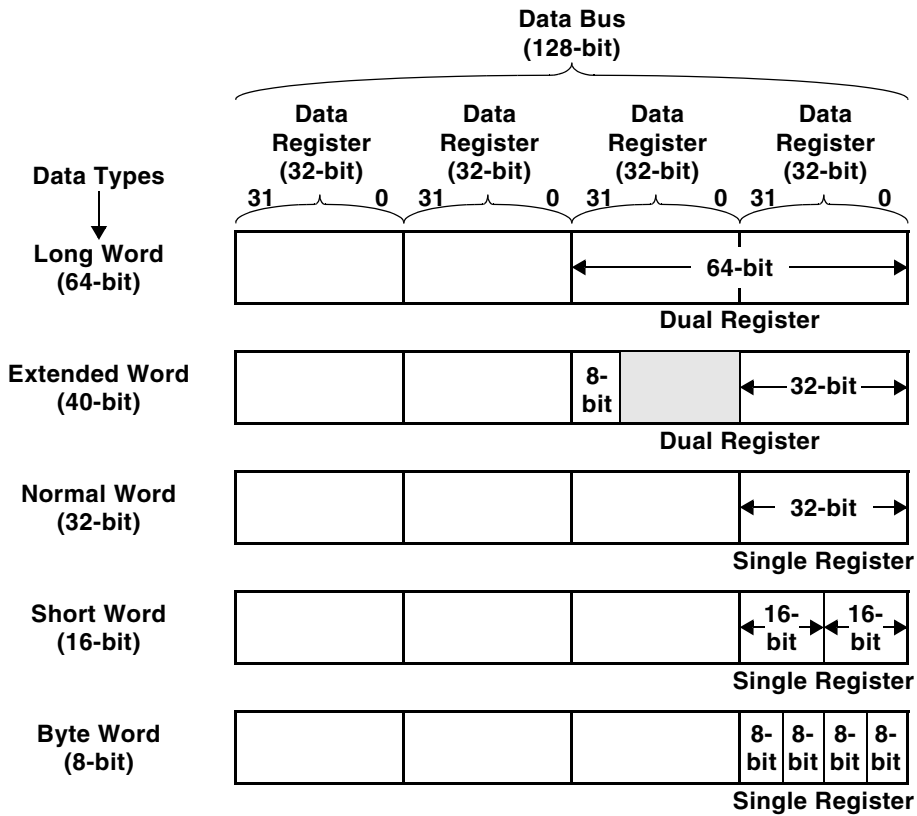


Figure 1-5. Word Format Definitions¹

- 1 The TigerSHARC processor internal data buses are 128 bits (one quad word) wide. In a quad word, the DSP can move 16 byte words, 8 short words, 4 normal words, or 2 long words over the bus at the same time.

For more information on the ALU (and CLU features), see [“ALU” on page 3-1](#).

Multiply Accumulator (Multiplier)

The multiplier performs fixed-point or floating-point multiplication and fixed-point multiply/accumulate operations. The multiplier supports several data types in fixed- and floating-point. The floating-point formats are float and float-extended, as in the ALU. The source and destination of most operations is the compute block register file.

The TigerSHARC processor's multiplier supports complex multiply-accumulate operations. Complex numbers are represented by a pair of 16-bit short words within a 32-bit word. The least significant bits (LSBs) of the input operand represents the real part, and the most significant bits (MSBs) of the input operand represent the imaginary part.

For more information on the multiplier, see [“Multiplier” on page 4-1](#).

Bit Wise Barrel Shifter (Shifter)

The shifter performs logical and arithmetic shifts, bit manipulation, field deposit, and field extraction. The shifter operates on one 64-bit, one or two 32-bit, two or four 16-bit, and four or eight 8-bit fixed-point operands. Shifter operations include:

- Shifts and rotates from off-scale left to off-scale right
- Bit manipulation operations, including bit set, clear, toggle and test
- Bit field manipulation operations, including field extract and deposit, using register `BF0TMP` (which is internal to the shifter)
- Bit FIFO operations to support bit streams with fields of varying length
- Support for ADSP-2100 family compatible fixed-point/floating-point conversion operations (such as exponent extract, number of leading 1s or 0s)

For more information on the shifter, see [“Shifter” on page 5-1](#).

Integer Arithmetic Logic Unit (IALU)

The IALUs can execute standard standalone ALU operations on IALU register files. The IALUs also provide memory addresses when data is transferred between memory and registers. The DSP has dual IALUs (the J-IALU and the K-IALU) that enable simultaneous addresses for multiple operand reads or writes. The IALUs allow computational operations to execute with maximum efficiency because the computation units can be devoted exclusively to processing data.

Each IALU has a multiport, 32-word register file. Operations in the IALU are not pipelined. The IALUs support pre-modify with no update and post-modify with update address generation. Circular data buffers are implemented in hardware. The IALUs support the following types of instructions:

- Regular IALU instructions
- Move Data instructions
- Load Data instructions
- Load/Store instructions with register update
- Load/Store instructions with immediate update

For indirect addressing (instructions with update), one of the registers in the register file can be modified by another register in the file or by an immediate 8- or 32-bit value, either before (pre-modify) or after (post-modify) the access. For circular buffer addressing, a length value can be associated with the first four registers to perform automatic modulo addressing for circular data buffers; the circular buffers can be located at arbitrary boundaries in memory. Circular buffers allow efficient implementation of delay lines and other data structures, which are commonly

used in digital filters and Fourier transformations. The TigerSHARC processor circular buffers automatically handle address pointer wraparounds, reducing overhead and simplifying implementation.

The IALUs also support bit reverse addressing, which is useful for the FFT algorithm. Bit reverse addressing is implemented using a reverse carry addition that is similar to regular additions, but the carry is taken from the upper bits and is driven into lower bits.

The IALU provides flexibility in moving data as single-, dual-, or quad-words. Every instruction can execute with a throughput of one per cycle. IALU instructions execute with a single cycle of latency while computation units have two cycles of latency. Normally, there are no dependency delays between IALU instructions, but if there are, three or four cycles of latency can occur.

For more information on the IALUs, see [“IALU” on page 6-1](#).

Program Sequencer

The program sequencer supplies instruction addresses to memory and, together with the IALUs, allows computational operations to execute with maximum efficiency. The sequencer supports efficient branching using the branch target buffer (BTB), which reduces branch delays for conditional and unconditional instructions. The sequencer and IALU's control flow instructions divide into two types:

- *Control flow instructions.* These instructions are used to direct program execution by means of jumps and to execute individual instructions conditionally.
- *Immediate extension instructions.* These instructions are used to extend the numeric fields used in immediate operands for the sequencer and the IALU.

Control flow instructions divide into two types:

- Direct jumps and calls based on an immediate address operand specified in the instruction encoding. For example: ‘if <cond> jump 100;’ always jumps to address 100, if the <cond> evaluates as true.
- Indirect jumps based on an address supplied by a register. The instructions used for specifying conditional execution of a line are a subcategory of indirect jumps. For example: ‘if <cond> cjmp;’ is a jump to the address pointed to by the CJMP register.



The control flow instruction must use the first instruction slot in the instruction line.

The TigerSHARC processor achieves its fast execution rate by means of an eight-cycle pipeline.

Two stages of the sequencer’s pipeline actually execute in the computation units. The computation units perform single-cycle operations with a two-cycle computation pipeline, meaning that results are available for use two cycles after the operation is begun. Hardware causes a stall if a result is not available in a given cycle (register dependency check). Up to two computation instructions per compute block can be issued in each cycle, instructing the ALU, multiplier or shifter to perform independent, simultaneous operations.

The TigerSHARC processor has four general-purpose external interrupts, $\overline{IRQ3-0}$. The processor also has internally generated interrupts for the two timers, DMA channels, link ports, arithmetic exceptions, multiprocessor vector interrupts, and user-defined software interrupts. Interrupts can be nested through instruction commands. Interrupts have a short latency and do not abort currently executing instructions. Interrupts vector directly to a user-supplied address in the interrupt table register file, removing the overhead of a second branch.

The branch penalty in a deeply pipelined processor such as the TigerSHARC processor can be compensated for by the use of a branch target buffer (BTB) and branch prediction. The branch target address is stored in the BTB. When the address of a jump instruction, which is predicted by the user to be taken in most cases, is recognized (the tag address), the corresponding jump address is read from the BTB and is used as the jump address on the next cycle. Thus the latency of a jump is reduced from three to six wasted cycles to zero wasted cycles. If this address is not stored in the BTB, the instruction must be fetched from memory.

Other instructions also use the BTB to speed up these types of branches. These instructions are interrupt return, call return, and computed jump instructions.

Immediate extensions are associated with IALU or sequencer (control flow) instructions. These instructions are not specified by the programmer, but are implied by the size of the immediate data used in the instructions. The programmer must place the instruction that requires an immediate extension in the first instruction slot and leave an empty instruction slot in the line (use only three slots), so the assembler can place the immediate extension in the second instruction slot of the instruction line.



Note that only one immediate extension may be in a single instruction line.

For more information on the sequencer, BTB, and immediate extensions, see [“Program Sequencer” on page 7-1](#).

Quad Instruction Execution

The TigerSHARC processor can execute up to four instructions per cycle from a single memory block, due to the 128-bit wide access per cycle. The ability to execute several instructions in a single cycle derives from a *Static Superscalar* architectural concept. This is not strictly a superscalar architecture because the instructions executed in each cycle are specified in the

DSP Architecture

instruction by the programmer or by the compiler, and not by the chip hardware. There is also no instruction reordering. Register dependencies are, however, examined by the hardware and stalls are generated where appropriate. Code is fully compacted in memory and there are no alignment restrictions for instruction lines.

Relative Addresses for Relocation

Most instructions in the TigerSHARC processor support PC relative branches to allow code to be relocated easily. Also, most data references are *register relative*, which means they allow programs to access data blocks relative to a base register.

Nested Call and Interrupt

Nested call and interrupt return addresses (along with other registers as needed) are saved by specific instructions onto the on-chip memory stack, allowing more generality when used with high-level languages. Non-nested calls and interrupts do not need to save the return address in internal memory, making these more efficient for short, non-nested routines.

Context Switching

The TigerSHARC processor provides the ability to save and restore up to eight registers per cycle onto a stack in two internal memory blocks when using load/store instructions. This fast save/restore capability permits efficient interrupts and fast context switching. It also allows the TigerSHARC processor to dispense with on-chip PC stack or alternate registers for register files or status registers.

Internal Memory and Other Internal Peripherals

The on-chip memory consists of three blocks of 2M bits each. Each block is 128 bits (four words) wide, thus providing high bandwidth sufficient to support both computation units, the instruction stream and external I/O,

even in very intensive operations. The TigerSHARC processor provides access to program and two data operands without memory or bus constraints. The memory blocks can store instructions and data interchangeably.

Each memory block is organized as 64K words of 32 bits each. The accesses are pipelined to meet one clock cycle access time needed by the core, DMA, or by the external bus. Each access can be up to four words. Memories (and their associated buses) are a resource that must be shared between the compute blocks, the IALUs, the sequencer, the external port, and the link ports. In general, if during a particular cycle more than one unit in the processor attempts to access the same memory, one of the competing units is granted access, while the other is held off for further arbitration until the following cycle—*see* “Bus Arbitration Protocol” in the *ADSP-TS101 TigerSHARC Processor Hardware Reference*. This type of conflict only has a small impact on performance due to the very high bandwidth afforded by the internal buses.

An important benefit of large on-chip memory is that by managing the movement of data on and off chip with DMA, a system designer can realize high levels of determinism in execution time. Predictable and deterministic execution time is a central requirement in DSP and real-time systems.

Internal Buses

The processor core has three buses, each one connected to one of the internal memories. These buses are 128 bits wide to allow up to four instructions, or four aligned data words, to be transferred in each cycle on each bus. On-chip system elements also use these buses to access memory. Only one access to each memory block is allowed in each cycle, so DMA or external port transfers must compete with core accesses on the same block. Because of the large bandwidth available from each memory block, not all the memory bandwidth can be used by the core units, which leaves

DSP Architecture

some memory bandwidth available for use by the DSP's DMA processes or by the bus interface to serve other DSPs bus master transfers to the DSP's memory.

Internal Transfer

Most registers of the TigerSHARC processor are classified as universal registers (Uregs). Instructions are provided for transferring data between any two Uregs, between a Ureg and memory, or for the immediate load of a Ureg. This includes control registers and status registers, as well as the data registers in the register files. These transfers occur with the same timing as internal memory load/store.

Data Accesses

Each move instruction specifies the number of words accessed from each memory block. Two memory blocks can be accessed on each cycle because of the two IALUs. For a discussion of data and register widths and the syntax that specifies these accesses, see [“Register File Registers” on page 2-5](#).

Quad Data Access

Instructions specify whether one, two or four words are to be loaded or stored. Quad words¹ can be aligned on a quad-word boundary and long words aligned on a long-word boundary. This, however, is not necessary when loading data to computation units because a data alignment buffer (DAB) automatically aligns quad words that are not aligned in memory.

¹ A memory quad word is comprised of four 32-bit words or 128 bits of data.

Up to four data words from each memory block can be supplied to each computation unit, meaning that new data is not required on every cycle and leaving alternate cycles for I/O to the memories. This is beneficial in applications with high I/O requirements since it allows the I/O to occur without degrading core processor performance.

Booting

The internal memory of the TigerSHARC processor can be loaded from an 8-bit EPROM using a boot mechanism at system powerup. The DSP can also be booted using another master or through one of the link ports. Selection of the boot source is controlled by external pins. For information on booting the DSP, see the *ADSP-TS101 TigerSHARC Processor Hardware Reference*.

Scalability and Multiprocessing

The TigerSHARC processor, like the related Analog Devices product the SHARC DSP, is designed for multiprocessing applications. The primary multiprocessing architecture supported is a cluster of up to eight TigerSHARC processors that share a common bus, a global memory, and an interface to either a host processor or to other clusters. In large multiprocessing systems, this cluster can be considered an element and connected in configurations such as torroid, mesh, tree, crossbar, or others. The user can provide a personal interconnect method or use the on-chip communication ports.

The TigerSHARC processor improves on most of the multiprocessing capabilities of the SHARC DSP and enhances the data transfer bandwidth. These capabilities include:

- On-chip bus arbitration for glueless multiprocessing
- Globally accessible internal memory and registers

Instruction Line Syntax and Structure

- Semaphore support
- Powerful, in-circuit multiprocessing emulation

Emulation and Test Support

The TigerSHARC processor supports the IEEE standard P1149.1 Joint Test Action Group (JTAG) standard for system test. This standard defines a method for serially scanning the I/O status of each component in a system. The JTAG serial port is also used by the TigerSHARC processor EZ-ICE® to gain access to the processor's on-chip emulation features.

Instruction Line Syntax and Structure

TigerSHARC processor is a static superscalar DSP processor that executes from one to four 32-bit *instruction slots* in an *instruction line*. With few exceptions, an instruction line executes with a throughput of one cycle in an eight-deep pipeline. [Figure 1-6](#) shows the instruction slot and line structure.

There are some important things to note about the instruction slot and instruction line structure and how this structure relates to instruction execution.

- Each instruction line consists of up to four 32-bit instruction slots.
- Instruction slots are delimited with one semicolon “;”.
- Instruction lines are terminated with two semicolons “;;”.
- The up to four instructions on an instruction line are executed in parallel.
- Every instruction slot consists of a 32-bit opcode.

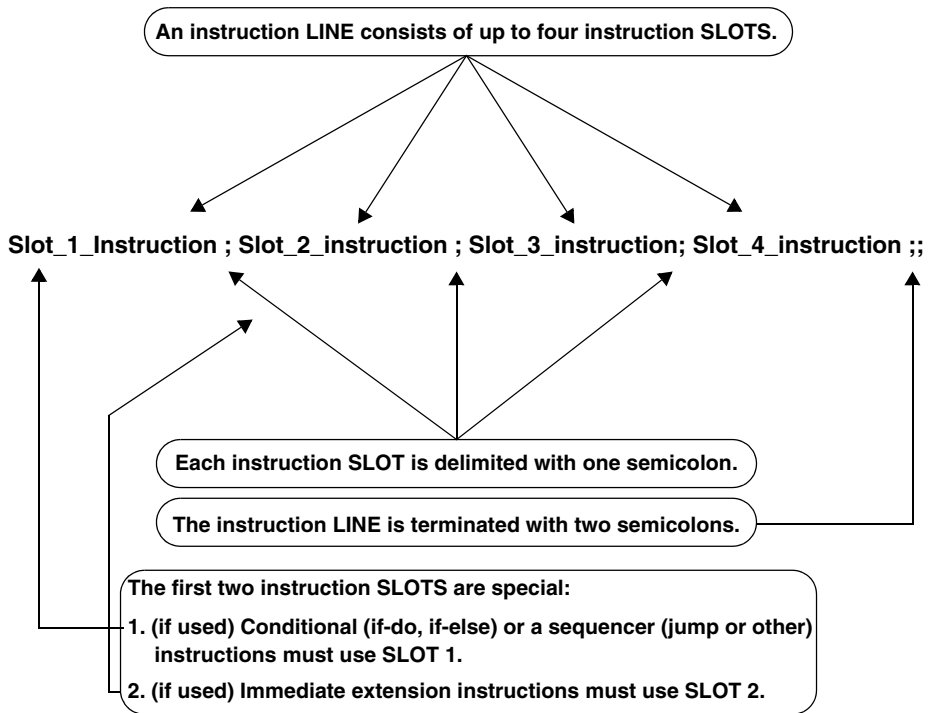


Figure 1-6. Instruction Line and Slot Structure

- Some instructions (such as immediate extensions) require two 32-bit opcodes (instruction slots) to execute.
- Some instructions (program sequencer, conditional, and immediate extension) require specific instruction slots.

An instruction is a 32-bit word that activates one or more of the TigerSHARC processor's execution units to carry out an operation. *The DSP executes or stalls the instructions in the same instruction line together.*

Although the DSP fetches quad words from memory, instruction lines do not have to be aligned to quad-word boundaries. Regardless of size (one to four instructions), instruction lines follow one after the other in memory

Instruction Line Syntax and Structure

with a new instruction line beginning one word from where the previous instruction line ended. The end of an instruction line is identified by the most significant bit (MSB) in the instruction word.

Instruction Notation Conventions

The TigerSHARC processor assembly language is based on an algebraic syntax for ease of coding and readability. The syntax for TigerSHARC processor instructions selects the *operation* that the DSP executes and the *mode* in which the DSP executes the operation. Operations include computations, data movements, and program flow controls. Modes include Single-Instruction, Single-Data (SISD) versus Single-Instruction, Multiple-Data (SIMD) selection, data format selection, word size selection, enabling saturation, and enabling truncation. All controls on instruction execution are included in the DSP's instruction syntax—there are no mode bits to set in control registers for this DSP.

This book presents instructions in summary format. This format presents all the selectable items and optional items available for an instruction. The conventions for these are:

<code>this that other</code>	Lists of items delimited with a vertical bar “ ” indicate that syntax permits selection of one of the items. One item from the list must be selected. The vertical bar is not part of instruction syntax.
<code>{option}</code>	An item or a list of items enclosed within curly braces “{}” indicate an optional item. The item may be included or omitted. The curly braces are not part of instruction syntax.
<code>() [] , ; ;;</code>	Parenthesis, square bracket, comma, semicolon, double semicolon, and other symbols are required items in the instruction syntax and must appear

where shown in summary syntax with *one exception*. Empty parenthesis (no options selected) may not appear in an instruction.

Rm Rmd Rmq

Register names are replaceable items in the summary syntax and appear in italics. Register names indicate that the syntax requires a single (*Rm*), double (*Rmd*), or quad (*Rmq*) register. For more information on register name syntax, compute block selection, and data format selection, see “[Register File Registers](#)” on page 2-5.

<*imm#*>

Immediate data (literal values) in the summary syntax appears as <*imm#*> with # indicating the bit width of the value.

For example, the following instruction in summary format:

```
{X|Y|XY}{S|B}Rs = MIN|MAX (Rm, Rn) {{U}{Z}} ;
```

could be coded as any of the following instructions:

```
XR3 = MIN (R2, R1) ;
YBR2 = MAX (R1, R0) (UZ);
XYSR2 = MAX (R3, R4) (U);
```

Unconditional Execution Support

The DSP supports unconditional execution of up to four instructions in parallel. This support lets programmers use simultaneous computations with data transfers and branching or looping. These operations can be combined with few restrictions. The following example code shows three instruction lines containing 2, 4, and 1 instruction slots each, respectively:

```
XR3:0=Q[J5+=J9]; YR1:0=R3:2+R1:0;;
XR3:0=Q[J5+=J9]; YR3:0=Q[K5+=K9]; XYR7:6=R3:2+R1:0; XYR8=R4*R5;;
J5=J9-J10;;
```

Instruction Parallelism Rules

It is important to note that the above instructions execute unconditionally. Their execution does not depend on computation-based conditions. For a description of condition dependent (conditional) execution, see [“Conditional Execution Support” on page 1-24](#).

Conditional Execution Support

All instructions can be executed conditionally (a mechanism also known as predicated execution). The condition field exists in one instruction slot in an instruction line, and all the remaining instructions in that line either execute or not, depending on the outcome of the condition.

In a conditional computational instruction, the execution of the entire instruction line can depend on the specified condition at the beginning of the instruction line. Conditional instructions take one of the following forms:

```
IF Condition;  
    D0, Instruction; D0, Instruction; D0, Instruction ;;  
  
IF Condition, Sequencer_Instruction;  
    ELSE, Instruction; ELSE, Instruction; ELSE, Instruction ;;
```

This syntax permits up to three instructions to be controlled by a condition. For more information, see [“Conditional Execution” on page 7-12](#).

Instruction Parallelism Rules

The TigerSHARC processor executes from one to four 32-bit instructions per line. The compiler or programmer determines which instructions may execute in parallel in the same line prior to runtime (leading to the name *Static Superscalar*). The DSP architecture places several constraints on the application of different instructions and various instruction combinations.

Note that all the restrictions refer to combinations of instructions within the same line. There is no restriction of combinations between lines. There are, however, cases in which certain combinations between lines may cause stall cycles (see “[Conditional Branch Effects on Pipeline](#)” on [page 7-44](#)), mostly because of data conflicts (operand of an instruction in line $n+1$ is the result of instruction in line $\#n$, which is not ready when fetched).

[Table 1-1 on page 1-29](#) and [Table 1-2 on page 1-34](#) identify instruction parallelism rules for the TigerSHARC processor. The following sections provide more details on each type of constraint and accompany the details with examples:

- “[General Restriction](#)” on [page 1-36](#)
- “[IALU Instruction Restrictions](#)” on [page 1-39](#)
- “[Compute Block Instruction Restrictions](#)” on [page 1-37](#)
- “[Sequencer Instruction Restrictions](#)” on [page 1-45](#)

The instruction parallelism rules in [Table 1-1](#) and [Table 1-2](#) present the resource usage constraints for instructions that occupy instruction slots in the same instruction line. The horizontal axis lists *resources*—portions of the DSP architecture that are active during an instruction—and lists the number of resources that are available. The vertical axis lists *instruction types*—descriptive names for classes of instructions. For resources, a ‘1’ indicates that a particular instruction uses one unit of the resource, and a ‘2’ indicates that the instruction uses two units of the resource. Typical instructions of most classes are listed with the descriptive name for the instruction type.

It is important to note that [Table 1-1](#) and [Table 1-2](#) identify static restrictions for the TigerSHARC processor. *Static* restrictions are distinguished from *dynamic* restrictions, in that static restrictions can be resolved by the

Instruction Parallelism Rules

assembler. For example, the assembler flags the instruction `xr3:0 = Q[J0 += 3];;` because the modifier is not a multiple of 4—this is a static violation.

Dynamic restrictions cannot be resolved by the assembler because these restrictions represent runtime conditions, such as stray pointers. When the processor encounters a dynamic (runtime) violation, an exception is issued when the violation runs through the core. Whatever the case, the processor does not arrive at a deadlock situation, although unpredictable results may be written into registers or memory.

As a dynamic restriction example, examine the instruction `xr3:0 = Q[J0 += 4];;`. Although this instruction looks correct to the assembler, it may violate hardware restrictions if `J0` is not quad aligned. Because the assembler cannot predict what the code will do to `J0` up to the point of this instruction, this violation is dynamic, since it occurs at runtime.

Further, [Table 1-1](#) and [Table 1-2](#) cover restrictions that arise from the interaction of instructions that share a line, but mostly omits restrictions of single instructions. An example of the former occurs when two instructions attempt to use the same unit in the same line. An example of an individual instruction restriction is an attempt to use a register that is not valid for the instruction. For example, the instruction `xr0 = CB[J5+=1];;` is illegal because circular buffer accesses can only use IALU registers `J0` through `J3`.

For most instruction types, you can locate the instruction in [Table 1-1](#) or [Table 1-2](#) and read across to find out the resources it uses. Resource usage for data movement instructions is more complicated to analyze. Resource usage for these instructions is calculated by adding together base resources, where base resources are determined by the type of move instruction. Move instructions are Ureg transfer (register to register), immediate load (immediate values to register), memory load (memory to register), and

memory store (register to memory). Source resources are determined by the resource register and are only applicable when the source itself is a register (Ureg transfer and stores). Destination resources may be of two types:

- Address pointer in post-modify (for example, `XR0 = [J0 += 2];;`)
- Destination register—only applicable when the destination is a register (Ureg transfer, memory loads and immediate loads)

If a particular combination of base, source, and destination uses more resources than are available, that combination is illegal. Consider, for example, the following instruction:

```
XR3:0 = Q[K31+0x40000];;
```

This is a memory load instruction, or specifically, a K-IALU load using a 32-bit offset. Reading across the table, the base resources used by the instruction are two slots in the line—the K-IALU instruction and the second instruction slot (for the immediate extension). The destination is `XR3:0`, which are X-compute block registers. The ‘X-Register File, Dreg = XR31–0’ line under ‘Ureg transfer and Store (Source Register) Resources’ in the table indicates that the instruction also uses an X-compute block port and an X-compute block input port.

The following Ureg transfer instruction provides another example:

```
XYR0=CJMP;;
```

This example uses the following resources:

- One instruction slot
- Base resources—an IALU instruction (no matter whether J-IALU or K-IALU) and the Ureg transfer resource (base resources) for the IALU instruction

Instruction Parallelism Rules

- Source resources—the sequencer I/O port
- Destination resources—an X-compute block port, an X-compute block input port, a Y-compute block port, and a Y-compute block input port

By comparison, the instruction `R3:0 = j7:4;;` uses an instruction slot, an IALU slot (no matter whether J or K), the Ureg transfer slot, and the J-IALU input port and output port.

Table 1-1. Parallelism Rules for Register File, DAB, J/K-IALU, and Port Access Instructions

	Resources:																					
	Inst. slots used	First inst. slot ¹	Second inst. slot ²	IALU inst.	Imm. load or Ureg xfer	J-IALU	K-IALU	J-IALU-port I/O	K-IALU-port I/O	X-ports I/O ³	X-ports input	X-ports output	X-DAB	Y-ports I/O ³	Y-ports input	Y-ports output	Y-DAB	Seq.-port I/O	Ext. Port I/O	IOP-port I/O ³	Link Port I/O ³	
⇒ Resources Available: ⇒	4	1	1	2	1	1	1	1	1	2	2	1	1	2	2	1	1	1	1	3	3	
↓ Instruction Types: ↓																						
<i>IALU Arithmetic</i>																						
J-IALU $J_s = J_m \text{ Op } J_n Imm8$	1			1		1																
J-IALU, 32-bit immediate $J_s = J_m \text{ Op } Imm32$	2		1	1		1																
K-IALU $K_s = K_m \text{ Op } K_n Imm8$	1			1			1															
K-IALU, 32-bit immediate $K_s = K_m \text{ Op } Imm32$	2		1	1			1															
<i>Data Move (resource total = instr. + Uregs)</i>																						
Ureg Transfer $Ureg = Ureg$	1			1	1																	
<i>Immediate Load (resource total = instr. + Ureg)</i>																						
Immediate 16-bit Load $Ureg = Imm16$	1			1	1																	
Immediate 32-bit Load $Ureg = Imm32$	2		1	1	1																	

Instruction Parallelism Rules

Table 1-1. Parallelism Rules for Register File, DAB, J/K-IALU, and Port Access Instructions (Cont'd)

	Resources:																				
	Inst. slots used	First inst. slot ¹	Second inst. slot ²	IALU inst.	Imm. load or Ureg xfer	J-IALU	K-IALU	J-IALU-port I/O	K-IALU-port I/O	X-ports I/O ³	X-ports input	X-ports output	X-DAB	Y-ports I/O ³	Y-ports input	Y-ports output	Y-DAB	Seq.-port I/O	Ext. Port I/O	IOP-port I/O ³	Link Port I/O ³
⇒ Resources Available: ⇒	4	1	1	2	1	1	1	1	1	2	2	1	1	2	2	1	1	1	1	3	3
↓ Instruction Types: ↓																					
<i>Memory Load (resource total = instr. + Ureg)</i>																					
J-IALU Load $Ureg = [Jm + += Jn imm8]$	1		1	1		1															
J-IALU Load, 32-bit offset $Ureg = [Jm + += imm32]$	2		1	1		1															
K-IALU Load $Ureg = [Km + += Kn imm8]$	1			1			1														
K-IALU Load, 32-bit offset $Ureg = [Km + += imm32]$	2		1	1			1														
<i>Memory Store (resource total = instr. + Ureg)</i>																					
J-IALU Store $[Jm + += Jn imm8] = Ureg$	1			1		1															
J-IALU Store, 32-bit offset $[Jm + += imm32] = Ureg$	2		1	1		1															
K-IALU Store $[Km + += Kn imm8] = Ureg$	1			1			1														
K-IALU Store, 32-bit offset $[Km + += imm32] = Ureg$	2		1	1			1														

Table 1-1. Parallelism Rules for Register File, DAB, J/K-IALU, and Port Access Instructions (Cont'd)

	Resources:																					
	Inst. slots used	First inst. slot ¹	Second inst. slot ²	IALU inst.	Imm. load or Ureg xfer	J-IALU	K-IALU	J-IALU-port I/O	K-IALU-port I/O	X-ports I/O ³	X-ports input	X-ports output	X-DAB	Y-ports I/O ³	Y-ports input	Y-ports output	Y-DAB	Seq.-port I/O	Ext. Port I/O	IOP-port I/O ³	Link Port I/O ³	
⇒ Resources Available: ⇒	4	1	1	2	1	1	1	1	1	2	2	1	1	2	2	1	1	1	1	3	3	
⇓ Instruction Types: ⇓																						
<i>Ureg transfer and Store (Source Register) Resources</i>																						
J-IALU <i>Ureg = J30-0 JB3-0 JL3-0</i>								1														
K-IALU <i>Ureg = K30-0 KB3-0 KL3-0</i>									1													
X-Register File <i>Dreg = XR31-0</i>										1		1										
Y-Register File <i>Dreg = XR31-0</i>														1		1						
XY-Register Files (SIMD) <i>Ureg = XYR31-0</i>										1		1		1		1						
Sequencer <i>Ureg = CJMP RETI RETS ...⁴</i>																			1			
External Port Control/Status <i>Ureg = SYSCON BUSLK ...⁵</i>																				1		
I/O Processor (DMA) <i>Ureg = DCS0 DCD0 ...⁶</i>																					1	
Link Port Control/Status/Buf. <i>Ureg = LCTL0 LCTL1 ...⁷</i>																						1

Instruction Parallelism Rules

Table 1-1. Parallelism Rules for Register File, DAB, J/K-IALU, and Port Access Instructions (Cont'd)

	Resources:																					
	Inst. slots used	First inst. slot ¹	Second inst. slot ²	IALU inst.	Imm. load or Ureg xfer	J-IALU	K-IALU	J-IALU-port I/O	K-IALU-port I/O	X-ports I/O ³	X-ports input	X-ports output	X-DAB	Y-ports I/O ³	Y-ports input	Y-ports output	Y-DAB	Seq.-port I/O	Ext. Port I/O	IOP-port I/O ³	Link Port I/O ³	
⇒ Resources Available: ⇒	4	1	1	2	1	1	1	1	1	2	2	1	1	2	2	1	1	1	1	3	3	
⇓ Instruction Types: ⇓																						
<i>Ureg Transfer and Load (Destination Register) Resources</i>																						
J-IALU <i>Ureg = J30-0 JB3-0 JL3-0</i>							1															
K-IALU <i>Ureg = K30-0 KB3-0 KL3-0</i>								1														
X-Register File <i>Dreg = XR31-0</i>										1	1											
Y-Register File <i>Dreg = XR31-0</i>														1	1							
XY-Register Files (SIMD) <i>Ureg = XYR31-0</i>										1	1			1	1							
Sequencer <i>Ureg = CJMP RETI RETS ...⁴</i>																		1				
External Port Control/Status <i>Ureg = SYSCON BUSLK ...⁵</i>																			1			
I/O Processor (DMA) <i>Ureg = DCS0 DCD0 ...⁶</i>																					1	
Link Port Control/Status/Buf. <i>Ureg = LCTL0 LCTL1 ...⁷</i>																						1

Table 1-1. Parallelism Rules for Register File, DAB, J/K-IALU, and Port Access Instructions (Cont'd)

	Resources:																				
	Inst. slots used	First inst. slot ¹	Second inst. slot ²	IALU inst.	Imm. load or Ureg xfer	J-IALU	K-IALU	J-IALU-port I/O	K-IALU-port I/O	X-ports I/O ³	X-ports input	X-ports output	X-DAB	Y-ports I/O ³	Y-ports input	Y-ports output	Y-DAB	Seq.-port I/O	Ext. Port I/O	IOP-port I/O ³	Link Port I/O ³
⇒ Resources Available: ⇒	4	1	1	2	1	1	1	1	1	2	2	1	1	2	2	1	1	1	1	3	3
↓ Instruction Types: ↓																					
<i>Memory Load Ureg (Destination Register) Resources</i>																					
X-Register File DAB/SDAB <i>XDreg = DAB q[addr]</i> <i>XDreg = XR31-0</i>										1	1		1								
Y-Register File DAB/SDAB <i>YDreg = DAB q[addr]</i> <i>YDreg = YR31-0</i>														1	1		1				
XY-Register Files DAB/SDAB <i>XYDreg = DAB q[addr]</i> <i>XYDreg = XYR31-0</i>										1	1		1	1	1		1				

- 1 If a conditional instruction is present on the instruction line, it must use the first instruction slot.
- 2 If an immediate extension is present on the instruction line, it must use the second instruction slot.
- 3 These resources are listed for informational purposes only. These constraints can not be exceeded within the core.
- 4 Complete list is all registers in register groups 0x1A, 0x38, and 0x39: CJMP, RETI, RETIB, RETS, DBGE, ILATSTL, ILATSTH, LC0, LC1, ILATL, ILATH, IMASKL, IMASKH, PMASKL, PMASKH, TIMER0L, TIMER0H, TIMER1L, TIMER1H, TMRIN0L, TMRIN0H, TMRIN1L, TMRIN1H, SQCTL, SQCTLST, SQCTLCL, SQSTAT, SFREG, ILATCLL, and ILATCLH.
- 5 Complete list is all registers in register groups 0x24 and 0x3A: SYSCON, BUSLK, SDRCON, SYSTAT, SYSTATCL, BMAX, BMAXC, AUTODMA0, and AUTODMA1.
- 6 Complete list is all registers in register groups 0x20 and 0x23: DCS0, DCD0, DCS1, DCD1, DCS2, DCD2, DCS3, DCD3, DCNT, DCNTST, DCNTCL, CSTAT, and DSTATC.
- 7 Complete list is all registers in register groups 0x25 and 0x27: LBUFTX0, LBUFRX0, LBUFTX1, LBUFRX1, LBUFTX2, LBUFRX2, LBUFTX3, LBUFRX3, LCTL0, LCTL1, LCTL2, LCTL3, LSTAT0, LSTAT1, LSTAT2, and LSTAT3.

Instruction Parallelism Rules

Table 1-2. Parallelism Rules for Compute Block and Sequencer Instructions

	Resources:										
	Inst. slots used	First inst. slot ¹	Second inst. slot ²	X-Comp Block Inst.	X-ALU	X-Multiplier	X-Shifter	Y-Comp Block Inst.	Y-ALU	Y-Multiplier	Y-Shifter
⇒ Resources Available: ⇒	4	1	1	2	1	1	1	2	1	1	1
⇓ Instruction Types: ⇓											
<i>Sequencer Instructions</i>											
Conditional Jump/Call, 16-bit offset <i>IF cond, JUMP CALL Imm16</i>	1	1									
Conditional Jump/Call, 32-bit offset <i>IF cond, JUMP CALL Imm32</i>	2	1	1								
Other Conditionals, Indirect Jumps, Static Flag Ops	1	1									
<i>X Compute Block Operations</i>											
X-ALU instruction, except quad output <i>XDreg = Dreg + Dreg</i>	1			1	1						
X-Multiplier instruction, except quad output <i>XDreg = Dreg * Dreg</i>	1			1		1					
X-Shifter instruction, except MASK, FDEP, STAT	1			1			1				
X-ALU instruction with quad output (<i>add_sub, EXPAND, MERGE</i>)	1			1	1		1				
X-Multiplier instruction with quad output	1			1		1	1				
X-Shifter instructions MASK, FDEP, XSTAT	1			2							

Table 1-2. Parallelism Rules for Compute Block and Sequencer Instructions

	Resources:										
	Inst. slots used	First inst. slot ¹	Second inst. slot ²	X-Comp Block Inst.	X-ALU	X-Multiplier	X-Shifter	Y-Comp Block Inst.	Y-ALU	Y-Multiplier	Y-Shifter
⇒ Resources Available: ⇒	4	1	1	2	1	1	1	2	1	1	1
⇓ Instruction Types: ⇓											
<i>Y Compute Block Operations</i>											
Y-ALU instruction, except quad output $YDreg = Dreg + Dreg$	1							1	1		
Y-Multiplier instruction, except quad output $YDreg = Dreg * Dreg$	1							1		1	
Y-Shifter instruction, except MASK, FDEP, STAT	1							1			1
Y-ALU instruction with quad output (add_sub, EXPAND, MERGE)	1							1	1		1
Y-Multiplier instruction with quad output	1							1		1	1
Y-Shifter instructions MASK, FDEP, YSTAT	1							2			
<i>X and Y Compute Block Operations (SIMD)</i>											
XY-ALU instruction, except quad output $XYDreg = Dreg + Dreg$	1			1	1			1	1		
XY-Multiplier instruction, except quad output $XYDreg = Dreg * Dreg$	1			1		1		1		1	
XY-Shifter instruction, except MASK, FDEP, STAT	1			1				1	1		1
XY-ALU instruction with quad output (add_sub, EXPAND, MERGE)	1			1	1			1	1	1	1
XY-Multiplier instruction with quad output	1			1		1		1	1	1	1
XY-Shifter instructions MASK, FDEP, X/YSTAT	1			2				2			

- 1 If a conditional instruction is present on the instruction line, it must use the first instruction slot.
- 2 If an immediate extension is present on the instruction line, it must use the second instruction slot.

Instruction Parallelism Rules

General Restriction

There is a general restriction that applies to all types of instructions: *Two instructions may not write to the same register.* This restriction is checked statically by the assembler. For example:

```
XR0 = R1 + R2 ; XR0 = R5 * R6 ;;  
/* Invalid; these instructions cannot be on the same instruction  
line */
```

```
XR1 = R2 + R3 , XR1 = R2 - R3 ;;  
/* Invalid; add-subtract to the same register */
```

Consequently, a load instruction may not be targeted to a register that is updated in the same line by another instruction. For example:

```
XR0 = [J17 + 1] ; R0 = R3 * R8 ;; /* Invalid */
```

A load/store instruction in that uses post-modify and update addressing cannot load the same register that is used as the index Jm/Km (pointer to memory). For example:

```
J0 = [J0 += 1] ;;  
/* Invalid; J0 cannot be used as both destination ( $J_s$ ) and index  
( $J_m$ ) in a post-modify (+=) load or store */
```

No instruction can write to the CJMP register in the same line as a CALL instruction (which also updates the CJMP register). For example:

```
if ALE, CALL label ; J6 = J0 + J1 (CJMP) ;; /* Invalid */
```

There are two types of loop counter updates, where combining them is illegal. For example:

```
IF LCOE; D0 ... ; LCO = [J0 + J1] ;; /* Invalid */
```

Compute Block Instruction Restrictions

There are two compute blocks, and instructions can be issued to either or both.

- Instructions in the format $XR_s = R_m \text{ op } R_n$ are issued to the X-compute block
- Instructions in the format $YR_s = R_m \text{ op } R_n$ are issued to the Y-compute block
- Instructions in the format $R_s = R_m \text{ op } R_n$ or $XYR_s = R_m \text{ op } R_n$ are issued to both the X- and Y-compute blocks

The following conditions apply when issuing instructions to the compute blocks. Note that the assembler statically checks all of these restrictions.

- Up to two instructions can be issued to each compute block (making that a maximum of four compute block instructions in one line). Note, however, that for this rule, the instructions of type $R_s = R_m \text{ op } R_n$ count as one instruction for each compute block. For example:

```
R0 = R1 + R2 ; R3 = R4 * R5 ;;
/* Valid; a total of four instructions */
```

```
XR0 = R1 + R2 ; XR3 = R4 * R5 ; XR6 = LSHIFT R1 BY R7 ;;
/* Invalid; three instructions to compute block X */
```

- Only one instruction can be issued to each unit (ALU, multiplier, or shifter) in a cycle. Each of the two instructions must be issued to a different unit (ALU, multiplier or shifter). For example:

```
XR0 = R1 + R2 ; XR6 = R1 + R2 ;; /* Invalid */
```

```
XR0 = R1 + R2 ; YR0 = R1 + R2 ;; /* Valid */
```

Instruction Parallelism Rules

- When one of the shifter instructions listed below is executed, it must be the only instruction in that line for the particular compute block. The instructions are: FDEP, MASK, GETBITS, PUTBITS and access to XSTAT/YSTAT registers. For example:

```
XR0 += MASK R1 BY R2 ; XR6 = R1 + R2 ;;  
/* Invalid; three operand shifter instruction in same line  
with an ALU operation; both issued to compute block X */
```

- Only one unit (ALU or multiplier) can use two result buses. A unit uses two result buses either when the result is quad word or when there are two results (dual ADD and SUB instructions— $R0 = R1+R2$, $R5 = R1-R2$;). Another instruction is allowed in the same line, as long as it is not a shifter instruction. For example:

```
R0 = R1 + R2 , R5 = R1 - R2 ; XR6 = R1 * R2 ;; /* Valid */
```

```
R0 = R1 + R2 , R5 = R1 - R2 ; XR6 = LSHIFT R1 BY R2 ;;  
/* Invalid; shifter instruction and two result ALU instruc-  
tion */
```

```
R0 = R1 + R2 , R5 = R1 - R2 ; XR3:0 = MR3:0 ;;  
/* Invalid; two instructions using two buses */
```

- There can be no other compute block instruction with Shifter load/store of X/YSTAT.
- In the multiplier, the option CR (clear and set round bit) and the option I (integer – not fractional) may not be used in the same multiply-accumulate instruction.
- The CR option of multiplier may be used only in these instructions:

```
MR3:2|MR1:0 +|- = Rm * Rn 32-bit fractional multiply-accumulate  
MR3:0 +|- = Rmd * Rnd Quad 16-bit fractional multiply-accumulate  
MR3:2|MR1:0 += Rm ** Rn Complex multiply-accumulate
```

- Communications Logic Unit (CLU) register load instructions have the same restrictions as shifter instructions, with one exception—a CLU register load instruction can be executed in the same instruction line with another compute instruction that has a quad result.
- All CLU instructions, except for load of CLU registers, refer to the same rules as compute ALU instructions.

IALU Instruction Restrictions

There are four types of IALU instructions:

- Memory load/store—for example: $R0 = [J0 + 1]$;
- IALU operations—for example: $J0 = J1 + J2$;
- Load data—for example: $R1 = 0xABCD$;
- Ureg transfer—for example: $XR0 = YR0$;

These restrictions apply when issuing instructions to the IALU. Except for the load data restriction, the assembler flags all of these restrictions.

- Up to one J-IALU and up to one K-IALU instruction can be issued in the same instruction line. For example:

```
R0 = [J0 += 1] ; R1 = [K0 += 1] ;;
/* It's recommended that J0 and K0 point to different mem-
memory blocks to avoid stall */

[J0 += 1] = XR0 ; [K0 += 1] = YR0;;
J0 = [J5 + 1] ; XR0 = [K6 + 1] ;;
R1 = 0xABCD ; R0 = [J0 += 1] ;;
/* One load data instruction (in K-IALU) and one J-IALU
operation */

XR0 = YR0 ; XR1 = [J0 += 1] ; YR1 = [K0 += 1] ;;
```

Instruction Parallelism Rules

```
/* Invalid; three IALU instructions */
```

```
XR0 = [J0 + 1] ; YR0 = [J1 + 1] ;;
```

```
/* Invalid; both use the same IALU (J-IALU) */
```

```
XR0 = [J0 + 1] ; J5 = J1 + 1 ;;
```

```
/* Invalid; both use the same IALU (J-IALU) */
```

- Two accesses to the same memory address in the same line, when one of them is a store instruction is liable to give unpredictable results.
- Loading from external memory is only allowed to the compute block and IALU register files.
- Reading from a multiprocessing broadcast zone is illegal.
- Move register to register instruction: if one of the registers is compute block merged, the other may not be compute block register. For example:

```
XYR1:0 = XR11:8 ; /* Invalid */
```

```
XR11:8 = XYR1:0 ; /* Invalid */
```

```
XYR1:0 = J11:8 ; /* Valid */
```

```
J11:8 = XYR1:0 ; /* Valid */
```

- A line of instructions may contain at the most one of either “load immediate data to register” or “Ureg to Ureg transfer” instructions. For example:

```
XR0 = YR0 ;; /* Valid */
```

```
XR5 = YR5 ; YR8 = [J3 + J6] ;; /* Valid */
```

```
R0 = 0xFFFFFFFF ;;
/* Valid; one load immediate data and one immediate extension */
```

```
XR0 = YR0 ; J5 = 0xFFFF ;;
/* Invalid; one Ureg to Ureg transfer and one load immediate data instruction */
```

```
XR0 = YR0 ; J0 = XR1 ;;
/* Invalid; two Ureg to Ureg transfers */
```

```
R0 = 0xFFFF ; J1 = 0xFF ;;
/* Invalid; two load immediate data instructions */
```

- Access via DAB must be through a quad word load. It can not be via “merged” Ureg groups. For example:

```
R3:0 = DAB Q[J0 += 4] ;; /* Valid; broadcast */
```

```
R1:0 = DAB Q[J0 += 4] ;; /* Invalid; merged */
```

- DAB and circular buffer access to memory is allowed only with post-modify with update. For example:

```
XR1:0 = CB L[J2 + 2] ;; /* Invalid */
```

- Register groups 0x20 to 0x3F can be accessed via Ureg transfer only.
- In a register-to-register move, XY register may not be used as source or destination of the transaction, unless it is both source and destination. For example:

```
R1:0 = R11:10 ;; /* Valid */
```

```
J1:0 = R11:10 ;; /* Invalid */
```

```
R3:0 = J3:0 ;; /* Invalid */
```

Instruction Parallelism Rules

- There can be up to two load instructions to the same compute block register file or up to one load to and one store from the same compute block register file. (A compute block register file has one input port and one input/output port.) If two store instructions are issued, none of them will be executed. For example:

```
[J0 + 1] = XR0 ; [K0 + 1] = XR1 ;;  
/* Invalid; attempts to use two output ports */
```

```
R0 = [J0 + 1] ; R1 = [K1 + 1] ;;  
/* Valid; uses two input ports in compute block X and Y */
```

```
R0 = [J0 + 1] ; [K1 + 1] = XR1 ;; /* Valid */
```

- A Ureg transfer within the same compute register file cannot be used with any other store to that register file. For example:

```
XR3:0 = R7:4 ; [J17 + 2] = YR4 ;;  
/* Valid; different register files */
```

```
XR3:0 = R7:4 ; XR0 = [J17 + 2] ;;  
/* Valid; one Ureg trans. and one load to compute block X  
*/
```

```
XR3:0 = R7:4 ; [J17 + 2] = XR4 ;;  
/* Invalid; one Ureg transfer and one store from compute  
block X */
```

```
R3:0 = R31:28 ;; /* Valid—SIMD Ureg transfer */
```

```
R3:0 = R31:28 ; [J17 + 2] = YR8;;  
/* Invalid—SIMD Ureg transfer (in both RFs) and store from  
compute block Y */
```


- Only one DAB load per Compute Block is allowed. For example:

```
XR3:0 = DAB Q[J0 += 4] ; XR7:4 = DAB Q[K0 += 4] ;;
/* Invalid */
```

```
XR3:0 = DAB Q[J0 += 4] ; YR7:4 = DAB Q[K0 += 4] ;; /* Valid
*/
```

- Only one memory load/store to and from the same single port register files is allowed. The single port register files are:
 - J-IALU registers: groups 0xC and 0xE
 - K-IALU registers: groups 0xD and 0xF
 - Bus Control registers: groups 0x24 and 0x3A
 - Sequencer, Interrupt and BTB registers: groups 0x1A, 0x30–0x39, and 0x3B
 - Debug logic registers: groups 0x1B, 0x3D–0x3F

For example:

```
J0 = [J5 + 1] ; K0 = [K6 + 1] ;; /* Valid */
```

```
J0 = [J5 + 1] ; [K6 + 1] = K0 ;; /* Valid */
```

```
J0 = [J5 + 1] ; [K6 + 1] = J1 ;;
/* Invalid; one load to J-IALU register file and one store
from J-IALU register file */
```

- Access to memory must be aligned to its size. For example, quad word access must be quad-word aligned. The long access must be aligned to an even address. This excludes load to compute block via

Instruction Parallelism Rules

DAB. In addition, the immediate address modifier must be a multiple of four in quad accesses and of two in long accesses. For example:

```
XR3:0 = Q[J0 += 3] ;; /* Invalid */
```

```
XR3:0 = Q[J0 += 4] ;; /* Valid */
```

- A Ureg store instruction and an instruction that updates the same Ureg may not be issued in the same instruction line, because the store instruction may be stalled and by the time it progresses, the contents may have been modified by the update instruction. For example:

```
XR0 = R1 + R3 ; Q[J7 += 4] = XR3:0 ;; /* Invalid */
```

```
IF ALE, CALL label ; [J0 += 1] = CJMP ;;  
/* Invalid; CJMP is updated by the call instruction */
```

- For the following J-IALU circular buffer or bit-reversed addressing operations, J_m (the index) only may be J0, J1, J2, or J3:

$$J_s = J_m + | - J_n \text{ (CB)}$$
$$Ureg = \text{CB [L] [Q] } (J_m + | += J_n | Imm)$$
$$\text{CB [L] [Q] } (J_m + / += J_n | Imm) = Ureg$$
$$Ureg = \text{DAB [L] [Q] } (J_m + | += J_n | Imm)$$
$$Ureg = \text{BR [L] [Q] } (J_m + | += J_n | imm)$$
$$\text{BR [L] [Q] } (J_m + | += J_n | imm) = Ureg$$
$$Ureg = \text{BR [L] [Q] } (J_m + | += J_n | Imm)$$


The same restrictions apply to K-IALU instructions that use circular buffer or bit-reversed addressing operations.

- On load or store instructions the memory address may not be a register. For example, the address may not be a memory mapped register address in the range of 0x180000 to 0x1FFFFFF. For example:

```
Q[J2 + 0] = XR3:0 ;;
/* Invalid if J2 is in the range of 0x180000 to 0x1FFFFFF */
```

- If one IALU is used to access the other IALU register, there may not be an immediate load instruction in the same line. For example:

```
Q[J2 + 0] = K3:0 ; XR0 = 100 ;; /* Invalid */

Q[K2 + 0] = K3:0 ; XR0 = 100 ;; /* Valid */
```

Sequencer Instruction Restrictions

There can be one sequencer instruction and one immediate extension per line, where the sequencer instruction can be jump, indirect jump, and other instructions. The assembler statically checks all of these restrictions:

- The sequencer instruction must be the first instruction in the four-slot instruction line.
- The immediate extension must be the second instruction in the four-slot instruction line.
- The immediate extension is counted as one of the four instructions in the line.

Instruction Parallelism Rules

- There cannot be two instructions that end in the same quad-word boundary, and where both have branch instructions with a predicted bit set. For example:

```
IF MLE, JUMP + 100 ;; /* begin address 100 */
IF NALE JUMP -50 ;
XR0 = R5 + R6 ; J0 = J2 + J3 ; YR4 = [K3 + 40] ;;
/* Valid; first instruction line ends on 1001; second
instruction line ends on 1005 */
```

```
IF MLE, JUMP + 100 ;; /* begin address 100 */
IF NALE JUMP - 50 ;;
/* Invalid; both lines within the same quad word */
```

- For instruction `SCFx += op Cond`, there can be no operation between compute block static flags (XSF0/1, YSF0/1, and XYSF0/1) and non-compute block conditions.

2 COMPUTE BLOCK REGISTERS

The TigerSHARC processor core contains two compute blocks. Each compute block contains a register file and three independent computation units—an ALU, a multiplier, and a shifter. Because the execution of all computational instructions in the TigerSHARC DSP depends on the input and output data formats and depends on whether the instruction is executed on one computational block or both, it is important to understand how to use the TigerSHARC DSP's compute block registers. This chapter describes the registers in the compute blocks, shows how the register name syntax controls data format and execution location, and defines the available data formats.

The DSP has two compute blocks—compute block X and compute block Y. Each block contains a register file and three independent computation units. The units are the ALU, multiplier, and shifter.

A general-purpose, multiport, 32-word data register file in each compute block serves for transferring data between the computation units and the data buses and stores intermediate results. [Figure 2-1](#) shows how each of the register files provide the interface between the internal buses and the computational units within the compute blocks.

As shown in [Figure 2-1](#), data input to the register file passes through the data alignment buffer (DAB). The DAB is a two quad-word FIFO that provides aligned data for registers when dual- or quad-register loads receive misaligned data from memory. For more information on using the DAB, see [“IALU” on page 6-1](#).

COMPUTE BLOCK X

COMPUTE BLOCK Y

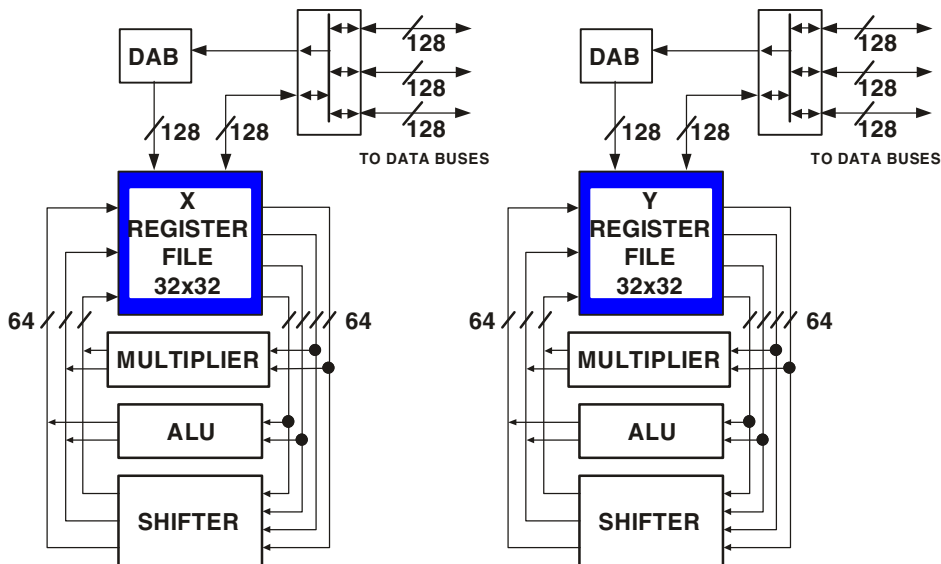



Figure 2-1. Data Register Files in Compute Block X and Y

Within the compute block, there are two types of registers—memory-mapped registers and non-memory-mapped registers. The memory-mapped registers in each of the compute blocks are the general-purpose data register file registers XR31-0 and YR31-0. Because these registers are memory mapped, they are accessible to external bus devices.

For operations within a single DSP, the distinction between memory-mapped and non-memory-mapped compute block registers is important because the memory-mapped registers are *Universal registers* (*Ureg*). The *Ureg* group of registers is available for many types of operations working with portions of the DSP's core other than the portion of the core where the *Ureg* resides. The compute block *Ureg* registers can be

used for additional operations unavailable to other *Ureg* registers. To distinguish the compute block register file registers from other *Ureg* registers, the XR31-0 and YR31-0 registers are also referred to as *Data registers (Dreg)*.

For operations in a multiprocessing DSP system, it is very useful that 90% of the registers in the TigerSHARC processor are memory-mapped registers. The memory-mapped registers have absolute addresses associated with them, meaning that they can be accessed by other processors through multiprocessor space or accessed by any other bus masters in the system.

 A DSP can access its own registers by using the multiprocessor memory space, but the DSP would have to tie up the external bus to access its own registers this way.

The compute blocks have a few registers that are non-memory mapped. These registers do not have absolute addresses associated with them. The non-memory-mapped registers are special registers that are dedicated for special instructions in each compute block. The unmapped registers in the compute blocks include:

- Compute block status (XSTAT and YSTAT) registers
- Parallel Result (XPR1-0 and YPR1-0) registers—ALU
- Multiplier Result (XMR3-0 and YMR3-0) registers—Multiplier
- Multiplier Result Overflow (XMR4 and YMR4) registers—Multiplier
- Bit FIFO Overflow Temporary (XBFOTMP and YBFOTMP) registers—Shifter

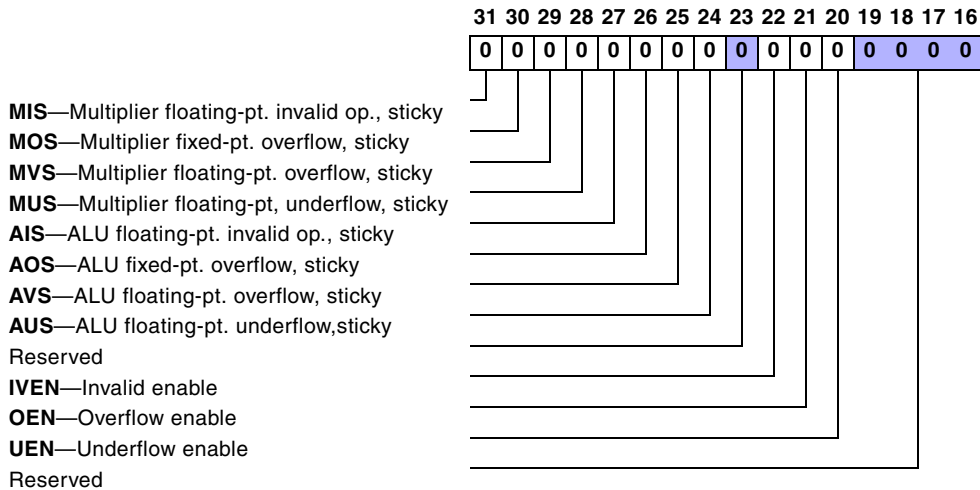


Figure 2-2. XSTAT/YSTAT (Upper) Register Bit Descriptions

The non-memory-mapped registers serve special purposes in each compute block. The $X/YSTAT$ registers (shown in [Figure 2-2](#) and [Figure 2-3](#)) hold the status flags for each compute block. These flags are set or reset to indicate the status of an instruction's execution a compute block's ALU, multiplier, and shifter. The $X/YPR1-0$ registers hold parallel results from the ALU's `SUM`, `ABS`, `VMAX`, and `VMIN` instructions. The $X/YMR3-0$ registers optionally hold results from fixed-point multiply operations, and the $X/YMR4$ register holds overflow from those operations. The $X/YBFOTMP$ registers temporarily store or return overflow from `GETBITS` and `PUTBITS` instructions.

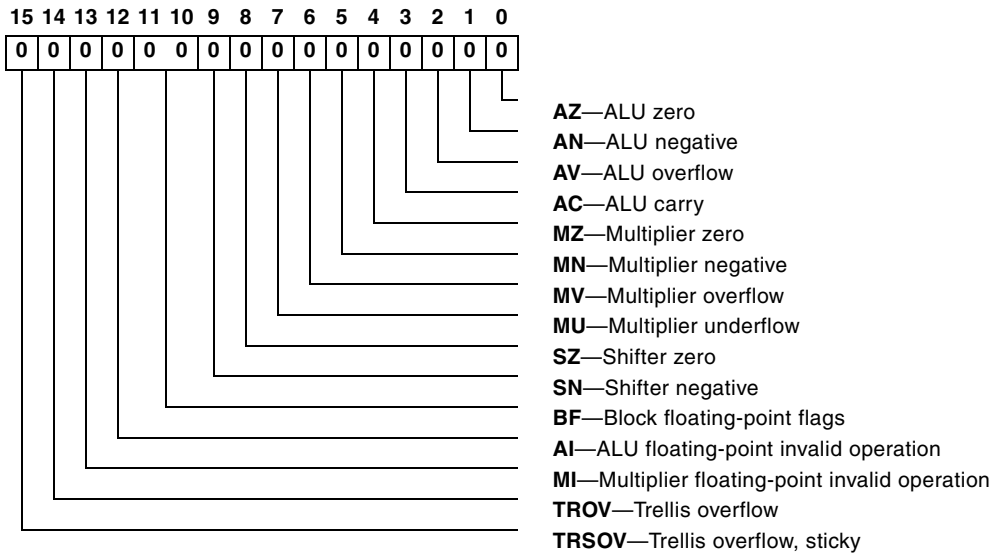


Figure 2-3. XSTAT/YSTAT (Lower) Register Bit Descriptions

Register File Registers

The compute block X and Y register files contain thirty-two 32-bit registers, which serve as a compute block's interface between DSP internal bus and the computational units. The register file registers—XR31-0 and YR31-0—are both universal registers (*Ureg*) and data registers (*Dreg*).

All inputs for computations come from the register file and all results are sent to the register file, except for fixed-point multiplies which can optionally be sent to the MR3-0 registers.



It is important to note that a register may be used once in an *instruction slot*, but the assembly syntax permits using registers multiple times within an *instruction line* (which contains up to four instruction slots). The register file registers are hardware interlocked, meaning that there is dependency checking during each computation to make sure the correct values are being used. When

Register File Registers

a computation accesses a register, the DSP performs a register check to make sure there are no other dependencies on that register. For more information on instruction lines and dependencies, see [“Instruction Line Syntax and Structure”](#) on page 1-20 and [“Instruction Parallelism Rules”](#) on page 1-24.

There are many ways to name registers in the TigerSHARC DSP’s assembly syntax. The register name syntax provides selection of many features of computational instructions. Using the register name syntax in an instruction, you can specify:

- Compute block selection
- Register width selection
- Operand size selection
- Data format selection

Figure 2-4 shows the parts of the register name syntax and the features that the syntax selects.

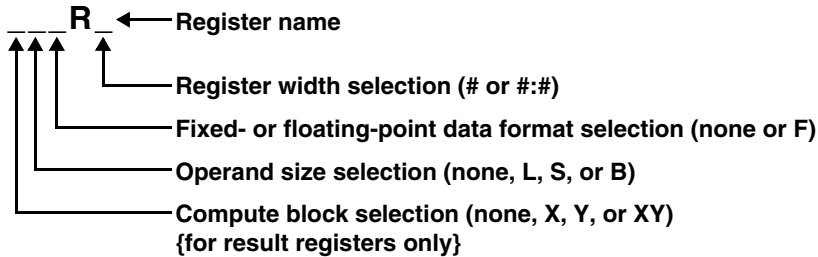


Figure 2-4. Register File Register Name Syntax

i The DSP's assembly syntax also supports selection of integer or fractional and real or complex data types. These selections are provided as options to instructions and are not part of register file register name syntax.

Compute Block Selection

As shown in Figure 2-4, the assembly syntax for naming registers lets you select the compute block of the register with which you are working.

The X and Y register-name prefixes denote in which compute block the register resides: X = compute block X only, Y = compute block Y only, and XY (or no prefix) = both. The following ALU instructions provide some register name syntax examples.

```
XR0 = R1 + R2 ;; /* This instruction executes in block X */
This instruction uses registers XR0, XR1, and XR2.
```

```
YR1 = R5 + R6 ;; /* This instruction executes in block Y */
This instruction uses registers YR1, YR5, and YR6.
```

Register File Registers

```
XYR0 = R0 + R2 ;; /* This instruction executes in block X & Y */  
This instruction uses registers XR0, XR2, YR0, and YR2.
```

```
R0 = R22 + R3 ;; /* This instruction executes in block X & Y */  
This instruction uses registers XR0, XR22, XR3, YR0, YR22, and YR3.
```

Because the compute block prefix lets you select between executing the instruction in one or both compute blocks, this prefix provides the selection between *Single-Instruction, Single-Data* (SISD) execution and *Single-Instruction, Multiple-Data* (SIMD) execution. Using SIMD execution is a powerful way to optimize execution if the same algorithm is being used to process multiple channels of data.

It is important to note that SISD and SIMD are not modes that are turned on or off with some latency in the change. SISD and SIMD execution are always available as execution options simply through register name selection.

To represent optional items, instruction syntax definitions use curly braces { } around the item. To represent choices between items, instruction syntax definitions place a vertical bar | between items. The following syntax definition example and comparable instruction indicates the difference for compute block selection:

```
{X|Y|XY}Rs = Rm + Rn ;;  
/* the curly braces enclose options */  
/* the vertical bars separate choices */  
  
XYR0 = R1 + R0 ;;  
/* code, no curly braces – no vertical bars */
```

Register Width Selection

As shown in [Figure 2-4 on page 2-7](#), the assembly syntax for naming registers lets you select the width of the register with which you are working.

Each individual register file register (XR31-0 and YR31-0) is 32 bits wide. To support data sizes larger than a 32-bit word, the DSP's assembly syntax lets you combine registers to hold larger words. The register name syntax for register width works as follows:

- *Rs*, *Rm*, or *Rn* indicates a *Single register* containing a 32-bit word (or smaller).

For example, these are register names such as R1, XR2, and so on.

- *Rsd*, *Rmd*, or *Rnd* indicates a *Double register* containing a 64-bit word (or smaller).

For example, these are register names such as R1:0, XR3:2, and so on. The lower register must be evenly divisible by two.

- *Rsq*, *Rmq*, or *Rnq* indicates a *Quad register* containing a 128-bit word (or smaller).

For example, these are register names such as R3:0, XR7:4, and so on. The lowest register must be evenly divisible by 4.

The combination of italic and code font in the register name syntax above indicates a user-substitutable value. Instruction syntax definitions use this convention to represent multiple register names. The following syntax definition example and comparable instruction indicates the difference for register width selection.

```
{X|Y|XY}Rsd = Rmd + Rnd ;;  
/* replaceable register names, italics are variables */  
  
XR1:0 = R3:2 + R1:0 ;;  
/* code, no substitution */
```

Operand Size and Format Selection

As shown in [Figure 2-4 on page 2-7](#), the assembly syntax for naming registers lets you select the operand size and fixed- or floating-point format of the data placed within the register with which you are working.

Single, double, and quad register file registers (R_s , R_{sd} , R_{sq}) hold *operands* (inputs and outputs) for instructions. Depending on the operand size and fixed- or floating-point format, there may be more than one operand in a register.

To select the operand size within a register file register, a register name prefix selects *a size that is equal or less than the size of the register*. These operand size prefixes for fixed-point data work as follows.

- B — Indicates Byte (8-bit) word data. The data in a single 32-bit register is treated as four 8-bit words. Example register names with byte word operands are BR1, BR1:0, and BR3:0.
- S — Indicates Short (16-bit) word data. The data in a single 32-bit register is treated as two 16-bit words. Example register names with short word operands are SR1, SR1:0, and SR3:0.
- None — Indicates Normal (32-bit) word data. Example register names with normal word operands are R0 R1:0, and R3:0.
- L — Indicates Long (64-bit) word data. An example register name with a long word operand is LR1:0.



The B, S, and L options apply for ALU and Shifter operations. Operand size selection differs slightly for the multiplier. For more information, see [“Multiplier Operations” on page 4-4](#).

To distinguish between fixed- and floating-point data, the register name prefix F indicates that the register contains floating-point data. The DSP supports the following floating-point data formats.

- None — Indicates fixed-point data
- *FRs*, *FRm*, or *FRn* (floating-point data in a single register) — Indicates normal (IEEE format, 32-bit) word data. An example register name with a normal word, floating-point operand is *FR3*.
- *FRsd*, *FRmd*, or *FRnd* (floating-point data in a double register) — Indicates extended (40-bit) word data. An example register name with an extended word, floating-point operand is *FR1:0*.

Register File Registers

It is important to note that the operand size influences the execution of the instruction. For example, $SRsd = Rmd + Rnd;;$ is an addition of four short data operands, stored in two register pairs. An example of this type of instruction follows and has the results shown in [Figure 2-5](#).

$SR1:0 = R31:30 + R25:24;;$

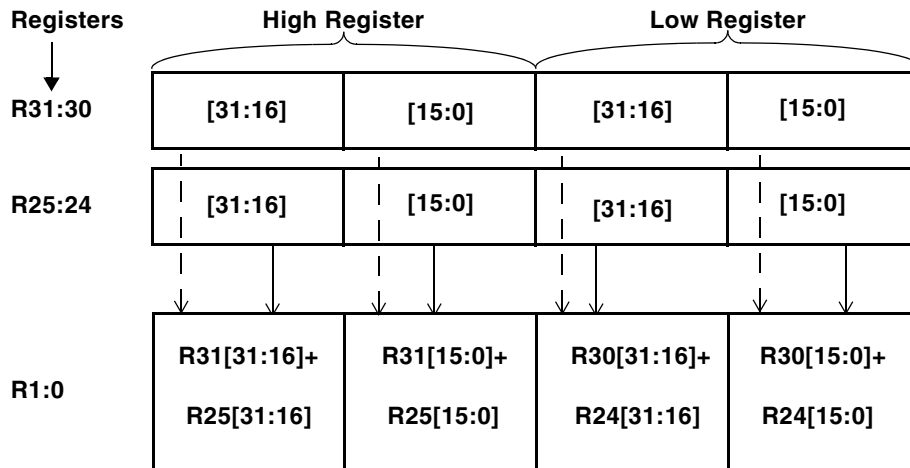


Figure 2-5. Addition of Four Short Word Operands in Double Registers

As shown in [Figure 2-5](#), this instruction executes the operation on all 64 bits in this example. The operation is executed on every group of 16 bits separately.

Registers File Syntax Summary

Data register file registers are used in computational instructions and memory load/store instructions. The syntax for those instructions is described in:

- “ALU” on page 3-1
- “Multiplier” on page 4-1
- “Shifter” on page 5-1

The following ALU instruction syntax description shows the conventions that all syntax descriptions use for data register file names:

$$\{X|Y|XY\}\{F\}Rsd = Rmd + Rnd ;;$$

Where:

- $\{X|Y|XY\}$ — The X, Y, or XY (none is same as XY) prefix on the register name selects the compute block or blocks to execute the instruction. The curly braces around these items indicate they are optional, and the vertical bars indicate that only one may be chosen.
- $\{F\}$ — The F prefix on the register name selects floating-point format for the operation. Omitting the prefix selects fixed-point format.
- Rsd — The result is a double register as indicated by the d . The register name takes the form $R\#:\#$, where the lower number is evenly divisible by two (as in $R1:0$).
- Rmd, Rnd — The inputs are double registers. The m and n indicate that these must be different registers.

Register File Registers

Here are some examples of register naming. In [Figure 2-6](#), the register name $XBR3$ indicates the operation uses four fixed-point 8-bit words in the X compute block $R3$ data register. In [Figure 2-7](#), the register name $XSR3$ indicates the operation uses two fixed-point 16-bit words in the X compute block $R3$ data register. In [Figure 2-8](#), the register name $XR3$ indicates the operation uses one fixed-point 32-bit word in the X compute block $R3$ data register. In [Figure 2-8](#), the register name $XFR3$ indicates floating-point data.

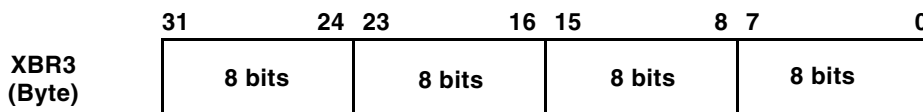


Figure 2-6. Register $R3$ in Compute Block X, Treated as Byte Data

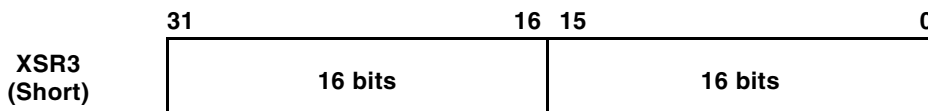


Figure 2-7. Register $R3$ in Compute Block X, Treated as Short Data

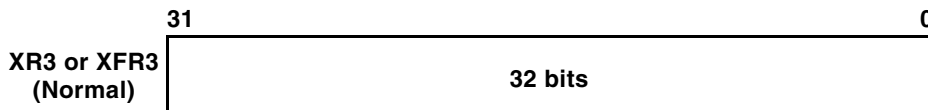


Figure 2-8. Register $R3$ in Compute Block X, Treated as Normal Data

Compute Block Registers

Here are additional examples of register naming. [Figure 2-9](#), [Figure 2-10](#), and [Figure 2-11](#) show examples of operand size in double registers, which are similar to the examples in [Figure 2-6](#), [Figure 2-7](#), and [Figure 2-8](#).

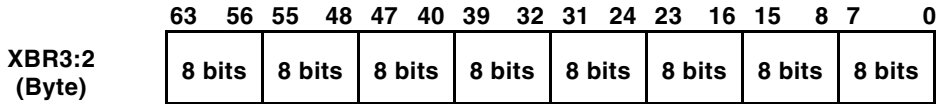


Figure 2-9. Register R3:2 in Compute Block X, Treated as Byte Data

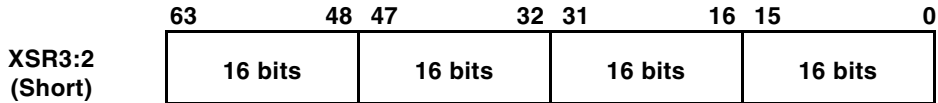


Figure 2-10. Register R3:2 in Compute Block X, Treated as Short Data

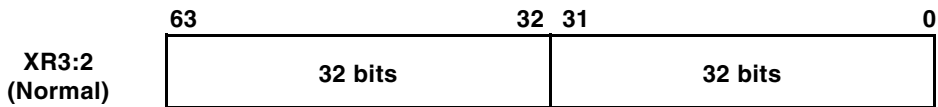


Figure 2-11. Register R3:2 in Compute Block X, Treated as Normal Data

The examples in [Figure 2-12](#) and [Figure 2-13](#) refer to two registers, but hold a single data word.

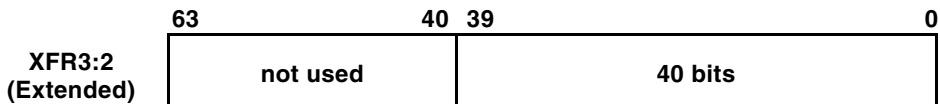


Figure 2-12. Register R3:2 in Compute Block X, Treated as Extended (Floating-Point) Data

Numeric Formats

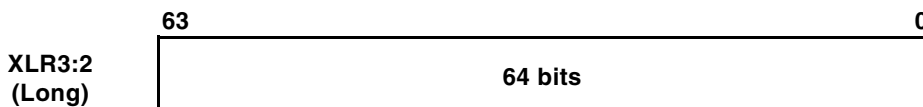


Figure 2-13. Register R3:2 in Compute Block X, Treated as Long Data

Numeric Formats

The DSP supports the 32-bit single-precision floating-point data format defined in the IEEE Standard 754/854. In addition, the DSP supports a 40-bit extended-precision version of the same format with eight additional bits in the mantissa. The DSP also supports 8-, 16-, 32-, and 64-bit fixed-point formats—fractional and integer—which can be signed (two’s-complement) or unsigned.

IEEE Single-Precision Floating-Point Data Format

IEEE Standard 754/854 specifies a 32-bit single-precision floating-point format, shown in [Figure 2-14](#). A number in this format consists of a sign bit s , a 24-bit significand, and an 8-bit unsigned-magnitude exponent e .

For normalized numbers, the significand consists of a 23-bit fraction f and a hidden bit of 1 that is implicitly presumed to precede f_{22} in the significand. The binary point is presumed to lie between this hidden bit and f_{22} . The least significant bit (LSB) of the fraction is f_0 ; the LSB of the exponent is e_0 .

The hidden bit effectively increases the precision of the floating-point significand to 24 bits from the 23 bits actually stored in the data format. This bit also insures that the significand of any number in the IEEE normalized number format is always greater than or equal to 1 and less than 2.

The unsigned exponent e can range between $1 \leq e \leq 254$ for normal numbers in the single-precision format. This exponent is biased by $+127$ ($254/2$). To calculate the true unbiased exponent, 127 must be subtracted from e .

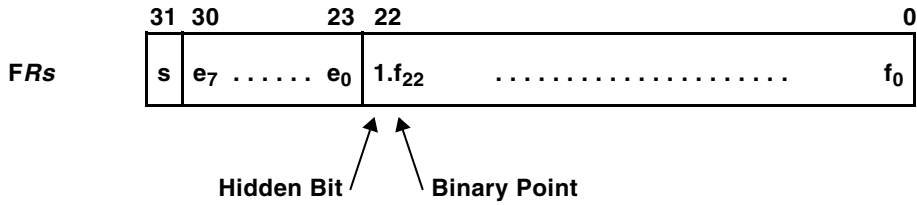


Figure 2-14. IEEE 32-Bit Single-Precision Floating-Point Format (Normal Word)

The IEEE standard also provides for several special data types in the single-precision floating-point format:

- An exponent value of 255 (all ones) with a nonzero fraction is a Not-A-Number (NaN). NaNs are usually used as flags for data flow control, for the values of uninitialized variables, and for the results of invalid operations such as $0 * \infty$.
- Infinity is represented as an exponent of 255 and a zero fraction. Note that because the fraction is signed, both positive and negative Infinity can be represented.
- Zero is represented by a zero exponent and a zero fraction. As with Infinity, both positive zero and negative zero can be represented.

The IEEE single-precision floating-point data types supported by the DSP and their interpretations are summarized in [Table 2-1](#).

Numeric Formats

Table 2-1. IEEE Single-Precision Floating-Point Data Types

Type	Exponent	Fraction	Value
NAN	255	Nonzero	Undefined
Infinity	255	0	$(-1)^s$ Infinity
Normal	$1 \leq e \leq 254$	Any	$(-1)^s (1.f_{22-0}) 2^{e-127}$
Zero	0	0 $(-1)^s$ Zero	

The TigerSHARC processor is compatible with the IEEE single-precision floating-point data format in all respects, except for:

- The TigerSHARC processor does not provide inexact flags.
- NAN inputs generate an invalid exception and return a quiet NAN.
- Denormal operands are flushed to zero when input to a computation unit and do not generate an underflow exception. Any denormal or underflow result from an arithmetic operation is flushed to zero and an underflow exception is generated.
- Round-to-nearest and round-towards-zero are supported. Round-to- \pm infinity are not supported.

Extended Precision Floating-Point Format

The extended precision floating-point format is 40 bits wide, with the same 8-bit exponent as in the standard format but with a 32-bit significand. This format is shown in Figure 2-15. In all other respects, the extended floating-point format is the same as the IEEE standard format.

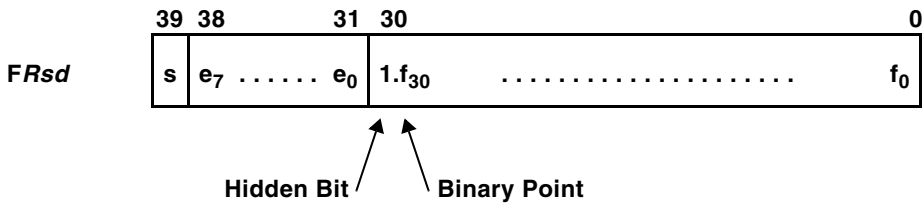


Figure 2-15. 40-Bit Extended-Precision Floating-Point Format (Extended Word)

Fixed-Point Formats

The DSP supports fixed-point fractional and integer formats for 16-, 32-, and 64-bit data. In these formats, numbers can be signed (two's-complement) or unsigned. The possible combinations are shown in Figure 2-20 through Figure 2-27. In the fractional format, there is an implied binary point to the left of the most significant magnitude bit. In integer format, the binary point is understood to be to the right of the LSB. Note that the sign bit is negatively weighted in a two's-complement format.

- i The DSP supports a fixed-point, signed, integer format for 8-bit data. Data in the 8- and 16-bit formats is always *packed* in 32-bit registers as follows—a single register holds four 8-bit or two 16-bit words, a dual register holds eight 8-bit or four 16-bit words, and a quad register holds sixteen 8-bit or eight 16-bit words.

Numeric Formats

ALU outputs always have the same width and data format as the inputs. The multiplier, however, produces a 64-bit product from two 32-bit inputs. If both operands are unsigned integers, the result is a 64-bit unsigned integer. If both operands are unsigned fractions, the result is a 64-bit unsigned fraction. These formats are shown in [Figure 2-30](#) and [Figure 2-31](#).

If one operand is signed and the other unsigned, the result is signed. If both inputs are signed, the result is signed and automatically shifted left one bit. The LSB becomes zero and bit 62 moves into the sign bit position. Normally bit 63 and bit 62 are identical when both operands are signed. (The only exception is full-scale negative multiplied by itself.) Thus, the left shift normally removes a redundant sign bit, increasing the precision of the most significant product. Also, if the data format is fractional, a single bit left shift renormalizes the MSB to a fractional format. The signed formats with and without left shifting are shown in [Figure 2-28](#) and [Figure 2-29](#).

The multiplier has an 80-bit accumulator to allow the accumulation of 64-bit products. For more information on the multiplier and accumulator, see [“Multiplier” on page 4-1](#).

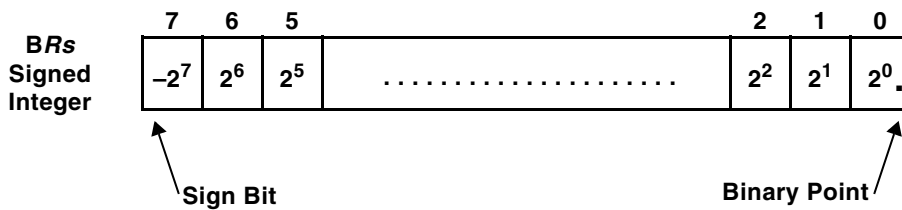


Figure 2-16. 8-Bit Fixed-Point Format, Signed Integer (Byte Word)

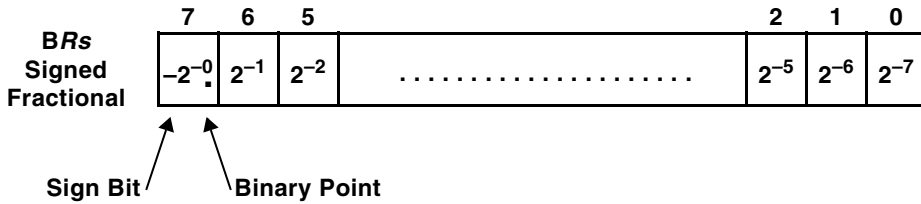


Figure 2-17. 8-Bit Fixed-Point Format, Signed Fractional (Byte Word)

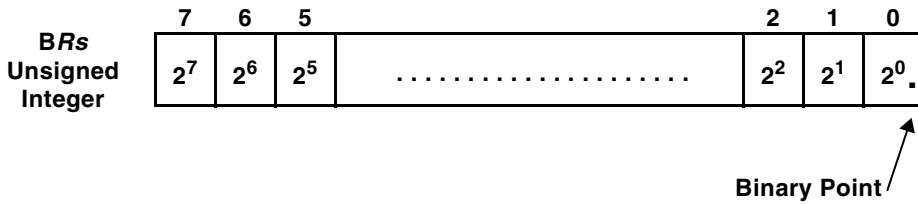


Figure 2-18. 8-Bit Fixed-Point Format, Unsigned Integer (Byte Word)

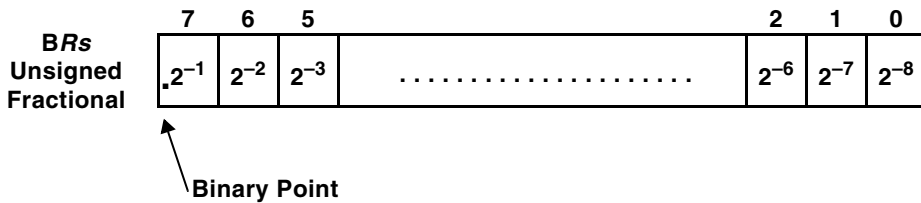


Figure 2-19. 8-Bit Fixed-Point Format, Unsigned Fractional (Byte Word)

Numeric Formats

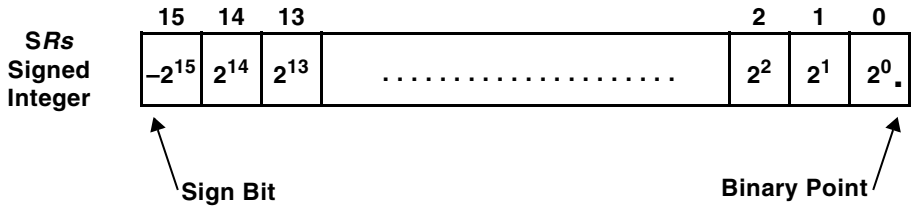


Figure 2-20. 16-Bit Fixed-Point Format, Signed Integer (Short Word)

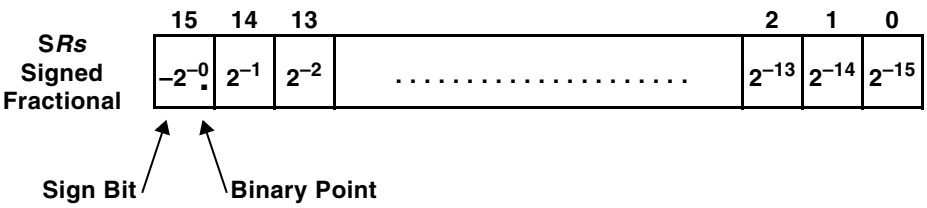


Figure 2-21. 16-Bit Fixed-Point Format, Signed Fractional (Short Word)

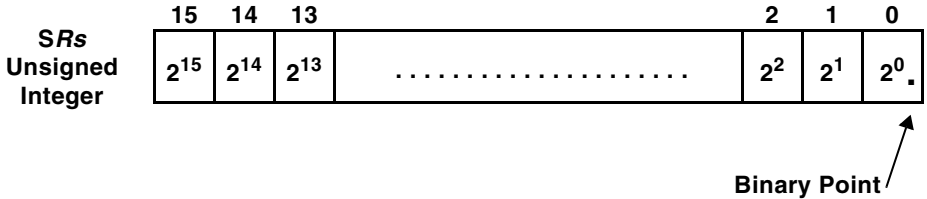


Figure 2-22. 16-Bit Fixed-Point Format, Unsigned Integer (Short Word)

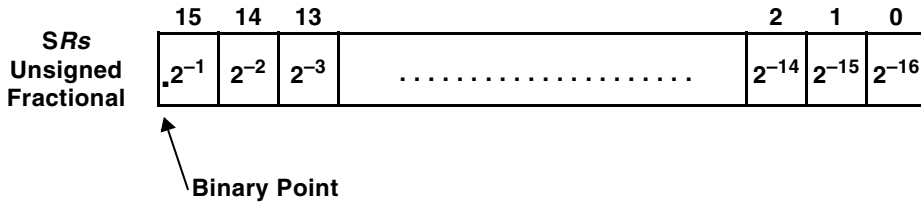


Figure 2-23. 16-Bit Fixed-Point Format, Unsigned Fractional (Short Word)

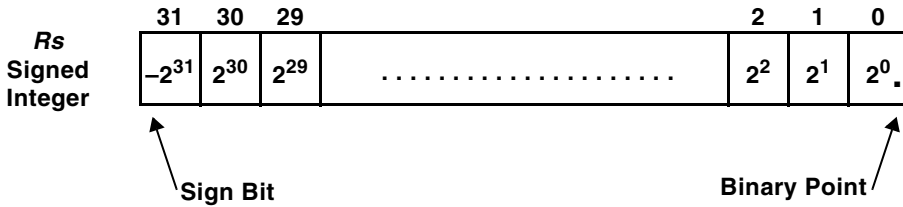


Figure 2-24. 32-Bit Fixed-Point Format, Signed Integer (Normal Word)

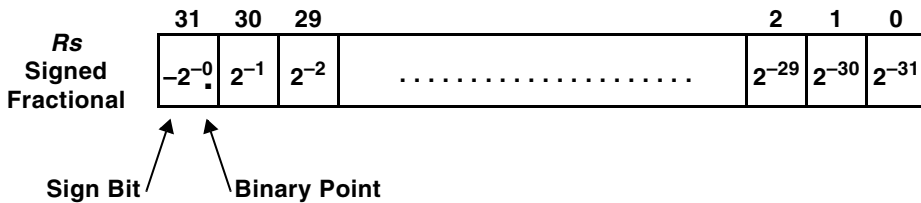


Figure 2-25. 32-Bit Fixed-Point Format, Signed Fractional (Normal Word)

Numeric Formats

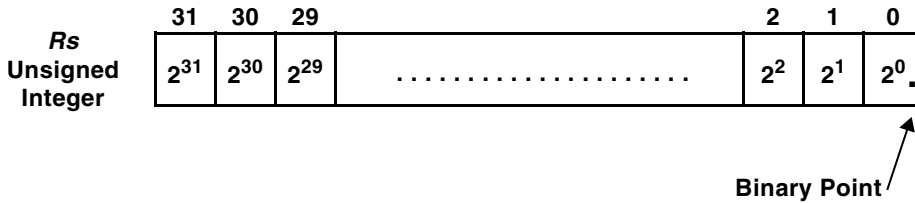


Figure 2-26. 32-Bit Fixed-Point Format, Unsigned Integer (Normal Word)

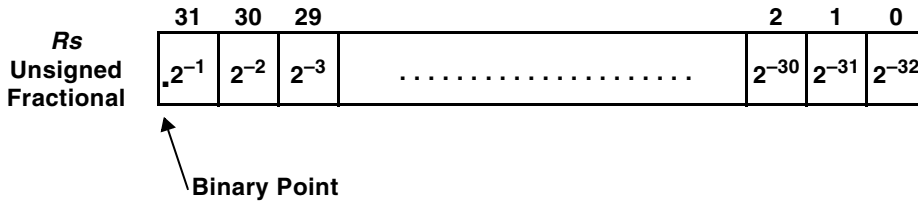


Figure 2-27. 32-Bit Fixed-Point Format, Unsigned Fractional (Normal Word)

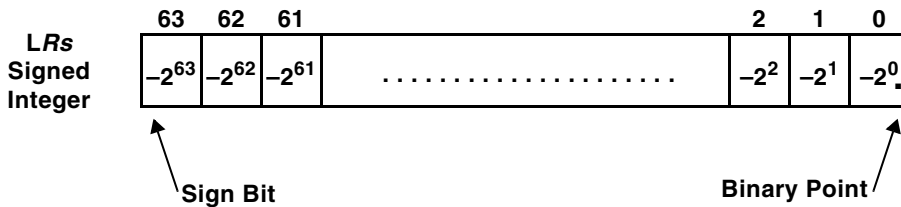


Figure 2-28. 64-Bit Fixed-Point Format, Signed Integer (Long Word)

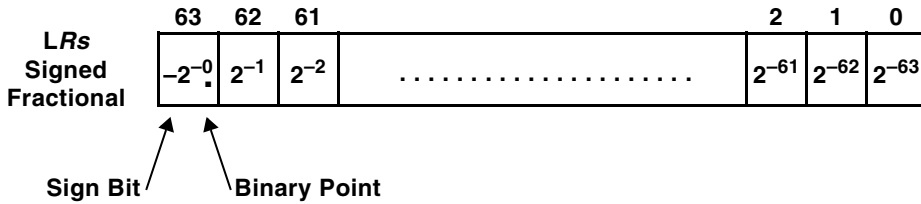


Figure 2-29. 64-Bit Fixed-Point Format, Signed Fractional (Long Word)

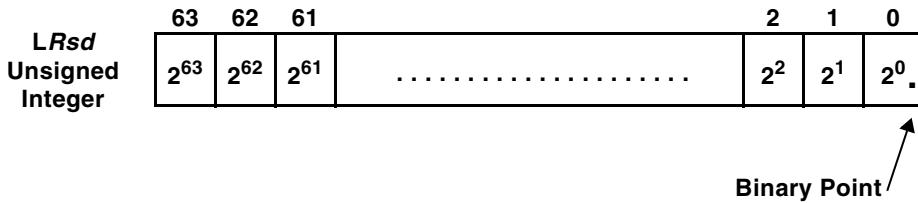


Figure 2-30. 64-Bit Fixed-Point Format, Unsigned Integer (Long Word)

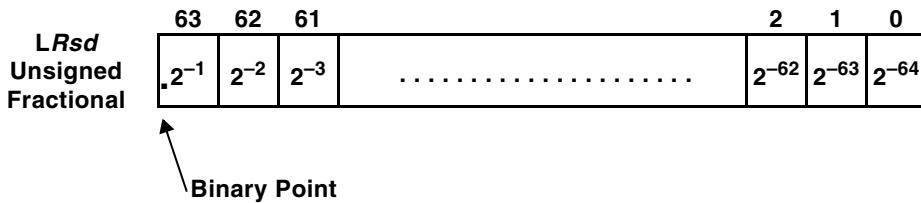


Figure 2-31. 64-Bit Fixed-Point Format, Unsigned Fractional (Long Word)

Numeric Formats

3 ALU

The TigerSHARC processor core contains two computation units known as compute blocks. Each compute block contains a register file and three independent computation units—an ALU, a multiplier, and a shifter. The Arithmetic Logic Unit (ALU) is highlighted in Figure 3-1. The ALU takes its inputs from the register file, and returns its outputs to the register file.

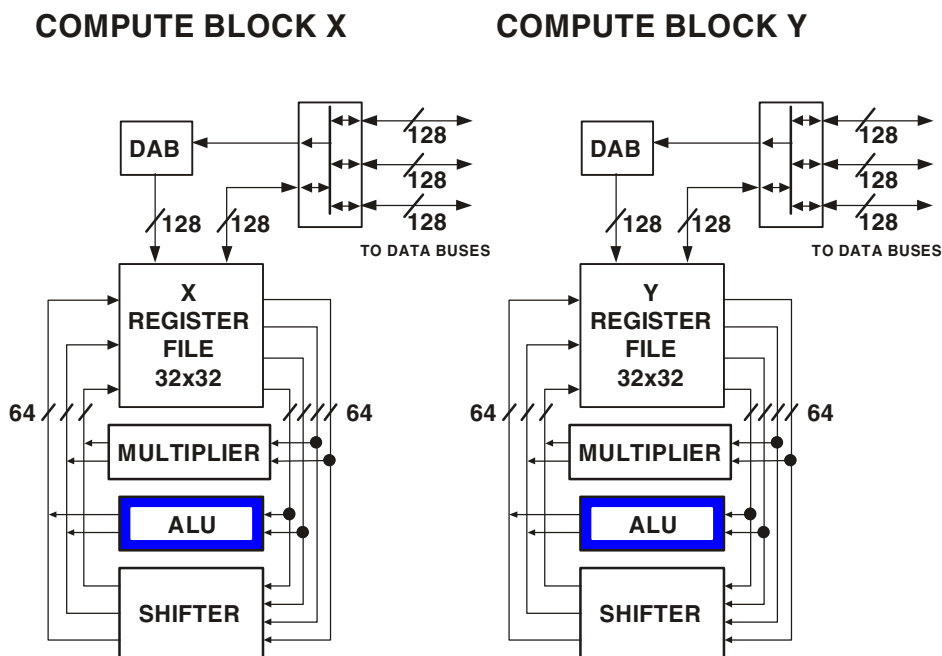


Figure 3-1. ALUs in Compute Block X and Y

This unit performs all *arithmetic operations* (addition/subtraction) for the processor on data in fixed-point and floating-point formats and performs *logical operations* for the processor on data in fixed-point formats. The ALU also executes *data conversion operations* such as expand/compact on data in fixed-point formats.

On the ADSP-TS101 processor, the ALU also performs specialized communications functions, primarily to support decoding and CDMA despreading operations. This functionality within the ALU is referred to as the *communication logic unit (CLU)*.

Not all ALU operations can be applied to both fixed- and floating-point data. Relating ALU operations and supported data types shows that the 64-Bit ALU unit within each compute block supports:

- Fixed- and floating-point *arithmetic operations* — add (+), subtract (-), minimum (MIN), maximum (MAX), Viterbi maximum (VMAX), comparison (COMP), clipping (CLIP), and absolute value (ABS)
- Fixed-point only *arithmetic operations* — increment (INC), decrement (DEC), sideways add (SUM), parallel result of sideways add (PRX=SUM), one's complement (ONES), and bit FIFO pointer increment (BFOINC)
- Floating-point only *arithmetic operations* — floating-point conversion (FLOAT), fixed-point conversion (FIX), copy sign (COPYSIGN), scaling (SCALB), inverse or division seed (RECIPS), square root or inverse square root seed (RSQRTS), extract mantissa (MANT), extract exponent (LOGB), extend operand (EXTD), and translate extended to a single operand (SNGL)
- Fixed-point only *logical operations* — AND, AND NOT, OR, XOR, and PASS

- Fixed-point only *data conversion (promotion/demotion) operations* — expand (EXPAND), compact (COMPACT), and merge (MERGE)
- Fixed-point only *CLU operations* — maximum of values for Viterbi decode (VMAX), Jacobian logarithm for turbo decode (TMAX), CDMA despreader (DESPREAD), polynomial reordering (PERMUTE), and trellis add/compare/select (ACS)

Examining the supported operands for each operation shows that the ALU operations support these data types:

- Fixed-point *arithmetic operations* and *logical operations* support:
 - 8-bit (byte) input operands
 - 16-bit (short) input operands
 - 32-bit (normal) input operands
 - 64-bit (long) input operands
 - output 8-, 16-, 32- or 64-bit results
- Floating-point *arithmetic operations* support:
 - 32-bit (normal) input operands (IEEE standard)
 - 40-bit (extended) input operands
 - output 32- or 40-bit results

- Fixed-point *data conversion operations* support:
 - 8-bit (byte) input operands
 - 16-bit (short) input operands
 - 32-bit (normal) input operands
 - 64-bit (long) input operands
 - 128-bit (quad) input operands
 - output 8-, 16-, 32-, 64-, or 128-bit results; 128-bit input and output operands only apply for `EXPAND` and `COMPACT`
- Fixed-point *CLU operations* support:
 - one or two 32-bit operands
 - two or four 16-bit operands
 - four or eight 8-bit operands
 - output 8-, 16-, or 32-bit results

Within instructions, the register name syntax identifies the input operand and output result data size and type. For more information on data size and type selection for ALU instructions, see [“Register File Registers” on page 2-5](#).

The remainder of this chapter presents descriptions of ALU instructions, options, and results using instruction syntax. For an explanation of the instruction syntax conventions used in ALU and other instructions, see [“Instruction Line Syntax and Structure” on page 1-20](#). For a list of ALU instructions and their syntax, see [“ALU Instruction Summary” on page 3-28](#).

ALU Operations

The ALU performs arithmetic operations on fixed-point and floating-point data and logical operations on fixed-point data. The DSP uses compute block registers for the input operands and output result from ALU operations. The compute block register file registers are XR31 through XR0 and YR31 through YR0. The ALU has one special-purpose double register—the PR register—for parallel results. The DSP uses the PR register with the different types of SUM, VMAX, and VMIN instructions. For more information on the register files and register naming syntax for selecting data type and width, see [“Register File Registers” on page 2-5](#). The following examples are ALU instructions that demonstrate arithmetic operations.

```
XR2 = R1 + R0 ;;
/* This is a fixed-point add of the 32-bit input operands XR1 and
XR0; the DSP places the result in XR2. */
```

```
YLR1:0 = ABS( R3:2 - R5:4 ) ::
/* This is a fixed-point subtract of the 64-bit input operand
XR5:4 from XR3:2; the DSP places the absolute value of the result
in XR1:0; the “L” in the result register name directs the DSP to
treat the input and output as 64-bit long data. */
```

```
XYFR2 = ( R1 + R0 ) / 2 ;;
/* This is a floating-point add and divide by 2 of the 32-bit
input operands XR1+XR0 and YR1+YR0; the DSP places the results in
XR2 and YR2; this is a Single-Instruction, Multiple-Data (SIMD)
operation, executing in both compute blocks simultaneously. */
```

ALU Operations

When multiple input operands are held in a single register, the DSP processes the data in parallel. For example, assume that the YR0 register contains 0x00050003 and the YR1 register contains 0x00040008 (as shown in [Figure 3-2](#)).

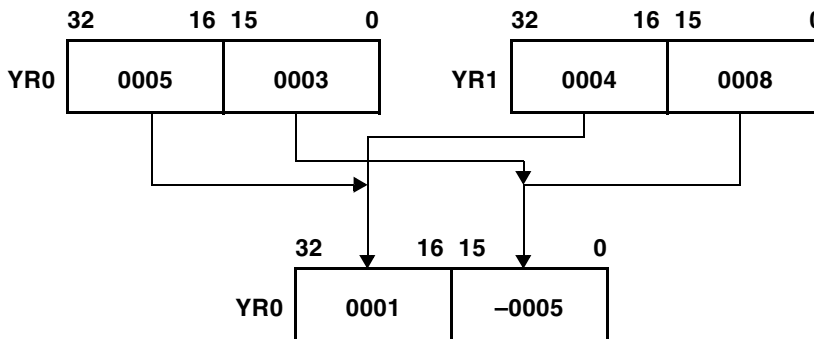


Figure 3-2. Input Operands for Parallel Subtract

After executing the instruction `YSR2 = R0 - R1 ; ;`, the YR2 register contains 0x0001FFFB (0x1 in upper half and -0x5 in lower half).

All ALU instructions generate status flags to indicate the status of the result. Because multiple operations are occurring in a parallel instruction, the value of the flag is an ORing of the results of all of the operations. The instruction demonstrated in [Figure 3-2](#) sets the YAN flag (Y compute block, ALU result negative) because one of the two subtractions resulted in a negative value. For more information on ALU status, see [“ALU Execution Status”](#) on page 3-11.

ALU Instruction Options

Most of the ALU instructions have options associated with them that permit flexibility in how the instructions execute. It is important to note that these options modify the detailed execution of instructions and options that are particular to a group of instructions—not all options are applicable to all instructions. Instruction options appear in parenthesis at the end of the instruction’s slot. For a list indicating which options apply for particular ALU instructions, see [“ALU Instruction Summary” on page 3-28](#). The ALU instruction options include:

- () signed operation, no saturation¹, round-to-nearest even², fractional mode³
- (S) signed operation, saturation¹
- (U) unsigned operation, no saturation¹, round-to-nearest even²
- (SU) unsigned operation, saturation¹
- (X) extend operation for ABS
- (T) signed operation, truncate⁴
- (TU) unsigned operation, truncate⁴
- (Z) signed result returns zero operation for MIN/MAX
- (UZ) unsigned result returns zero operation for MIN/MAX
- (I) signed operation, integer mode³
- (IU) unsigned operation, integer mode³

¹ Where saturation applies

² Where rounding applies

³ Where applies for floating-point operations

⁴ Where truncation applies

ALU Operations

- (IS) signed operation, saturation, integer mode³
- (ISU) unsigned operation, saturation, integer mode³

The following examples are ALU instructions that demonstrate arithmetic operations with options applied.

```
XR2 = R1 + R0 (S);;  
/* This is a fixed-point add of the 32-bit input operands with  
saturation. */
```

```
YLR1:0 = ABS( R3:2 - R5:4 ) (T) ::  
/* This is a fixed-point subtract of the 64-bit input operands  
with truncation. */
```

```
XYFR2 = ( R1 + R0 ) / 2 ( ) ;;  
/* This is a floating-point add and divide by 2 of the 32-bit  
input operands without truncation; this is the same as omitting  
the parenthesis. */
```

Signed/Unsigned Option

The DSP always represents fixed-point numbers in 8, 16, 32, or 64 bits, using up to four 32-bit data registers. Fixed- and floating-point data in the ALU may be unsigned or two's-complement. For information on the supported numeric formats, see [“Numeric Formats” on page 2-16](#).

Saturation Option

There are two types of saturation arithmetic that may be enabled for an instruction — signed or unsigned. For signed saturation, whenever overflow occurs (AV flag is set), the maximum positive value or the minimum negative value is replaced as the output of the operation. For unsigned saturation, overflow causes the maximum value or zero to be replaced as the output of the operation. Maximum and minimum values refer to the maximum and minimum values representable in the output format. For

example, the maximum positive, minimum negative, and maximum unsigned values in 16-bit short word arithmetic are $0x7fff$, $0x8000$, and $0xffff$ respectively.

Under saturation arithmetic, the flags AV and AC reflect the state of the ALU operation *prior* to saturation. For example, with signed saturation when an operation overflows, the maximum or minimum value is returned and AV remains set. On the other hand, the flags AN and AZ are set according to the final saturated result, therefore they correctly reflect the sign and any equivalence to zero of the final result. This allows the correct evaluation of the conditions AEQ , ALT , and ALE even during overflow, when using saturation arithmetic.

Extension (ABS) Option

For the ABS instruction, the X option provides an extended output range. Without the X , the output range is 0 to the maximum positive signed value ($0x0$ through $0x7F...F$). When ABS with the X option is used, the output range is extended from $0x0$ to $0xFF...FF$. The output numbers are unsigned in the extended range.

Truncation Option

For ALU instructions that support truncation as the T option, this option permits selection of the results rounding mode. The DSP supports two modes of rounding — round-toward-zero and round-toward-nearest. The rounding modes comply with the IEEE 754 standard and have these definitions:

- Round-Toward-Nearest (not using T option). If the result before rounding is not exactly representable in the destination format, the rounded result is the number that is nearer to the result before

ALU Operations

rounding. If the result before rounding is exactly halfway between two numbers in the destination format (differing by an LSB), the rounded result is the number that has an LSB equal to zero.

- Round-Toward-Zero (using T option). If the result before rounding is not exactly representable in the destination format, the rounded result is the number that is nearer to zero. This is equivalent to truncation.

Statistically, rounding up occurs as often as rounding down, so there is no large sample bias. Because the maximum floating-point value is one LSB less than the value that represents Infinity, a result that is halfway between the maximum floating-point value and Infinity rounds to Infinity in this mode.

Though these rounding modes comply with standards set for floating-point data, they also apply for fixed-point multiplier operations on fractional data. The same two rounding modes are supported, but only the round-to-nearest operation is actually performed by the multiplier. Using its local result register for fixed-point operations, the multiplier rounds-to-zero by reading only the upper bits of the result and discarding the lower bits.

Return Zero (MAX/MIN) Option

For the MAX/MIN instructions, the Z option changes the operation, returning zero if the second input register contains the maximum (for MAX) or minimum (for MIN) value. For example, *without the Z option*, the pseudo code for the MAX instruction is:

```
Rsd = MAX (Rmd, Rnd) ; ;
```

The ALU determines whether *Rmd* or *Rnd* contains the maximum and places the maximum in *Rsd*.

For example, *with the Z option*, the pseudo code for the MAX instruction is:

```
Rsd = MAX (Rmd, Rnd) (Z) ;;
```

The ALU determines whether *Rmd* or *Rnd* contains the maximum. If *Rmd* contains the maximum, the ALU places the maximum in *Rsd*. If *Rnd* contains the maximum, the ALU places zero in *Rsd*.

Fractional/Integer Option

The DSP always represents fixed-point numbers in 8, 16, 32, or 64 bits, using up to four 32-bit data registers. In the ALU, fractional or integer format is available for the EXPAND and COMPACT instructions. The default is fractional format. Use the I option for integer format. For information on the supported numeric formats, see “[Numeric Formats](#)” on page 2-16.

ALU Execution Status

ALU operations update status flags in the compute block’s Arithmetic Status (XSTAT and YSTAT) register (see [Figure 2-2 on page 2-4](#) and [Figure 2-3 on page 2-5](#)). Programs can use status flags to control execution of conditional instructions and initiate software exception interrupts. For more information, see “[ALU Execution Conditions](#)” on page 3-14.

[Table 3-1](#) shows the flags in XSTAT or YSTAT that indicate ALU status (a 1 indicates the condition) for the most recent ALU operation.

Table 3-1. ALU Status Flags

Flag	Definition	Updated By...
AZ	ALU fixed-point zero and floating-point underflow	All ALU ops
AN	ALU negative	All ALU ops
AV	ALU overflow	All arithmetic ops

ALU Operations

Table 3-1. ALU Status Flags (Cont'd)

Flag	Definition	Updated By...
AC	ALU carry	All fixed-point ops; cleared by floating-point ops
AI	ALU floating-point invalid operation	All floating-point ops; cleared by fixed-point ops

ALU operations also update sticky status flags in the compute block's Arithmetic Status (XSTAT and YSTAT) register. Table 3-2 shows the flags in XSTAT or YSTAT that indicate ALU sticky status (a 1 indicates the condition) for the most recent ALU operation. Once set, a sticky flag remains high until explicitly cleared.

Table 3-2. ALU Status Sticky Flags

Flag	Definition	Updated By...
AUS	ALU floating-point underflow, sticky	All floating-point ops
AVS	ALU floating-point overflow, sticky	All floating-point ops
AOS	ALU fixed-point overflow, sticky	All fixed-point ops
AIS	ALU floating-point invalid operation, sticky	All floating-point ops

Flag update occurs at the end of each operation and is available on the next instruction cycle. A program cannot write the Arithmetic Status register explicitly in the same cycle that the multiplier is performing an operation.

Multi-operand instructions (for example, $BRs = Rm + Rn$) produce multiple sets of results. In this case, the DSP determines a flag by ORing the result flag values from individual results.

AN — ALU Negative

The AN flag is set whenever the result of an ALU operation is negative. The AN flag is set to the most significant bit of the result. An exception is the instructions below, in which the AN flag is set differently:

- $R_s = \text{ABS } R_m$; AN is R_m (input data) sign
- $FR_s = \text{ABS } R_m$; AN is R_m (input data) sign
- $R_s = \text{ABS } (R_m \{+|- \} R_n)$; AN is set to be the sign of the result prior to ABS operation
- $R_s = \text{MANT } FR_m$; AN is R_m (input data) sign
- $FR_s = \text{ABS } (R_m \{+|- \} R_n)$; AN is set to be the sign of the result prior to ABS operation

The result sign of the above instructions is not indicated as it is always positive.

AV — ALU Overflow

The AV flag is an overflow indication. In all ALU operations, this bit is set when the correct result of the operation is too large to be represented by the result format. The overflow check is done always as signed operands, unless the instruction defines otherwise.

ALU Operations

If in the following example $R5$ and $R6$ are $0x70...0$ (large positive numbers), the result of the `Add` instruction (above) will produce a result that is larger than the maximum at the given format.

```
R10 = R5 + R6;;
```

As shown in the following example, an instruction can be composed of more than one operation.

```
R11:10 = expand (Rm + Rn)(I);
```

If Rm and Rn are $0x70...0$, the overflow is defined by the final result and is not defined by intermediate results. In the case above, there is no overflow.

AI — ALU Invalid

The `AI` flag indicates an invalid floating-point operation as defined by IEEE floating-point standard.

AC — ALU Carry

The `AC` flag is used as carry out of add or subtract instructions that can be chained. It can also be used as an indication for unsigned overflow in these operations. `AV` is set when there is signed overflow.

ALU Execution Conditions

In a conditional ALU instruction, the execution of the entire instruction line can depend on the specified condition at the beginning of the instruction line. Conditional ALU instructions take the form:

```
IF cond; D0, instr.; D0, instr.; D0, instruct. ;;
```

This syntax permits up to three instructions to be controlled by a condition. Omitting the `D0` before the instruction makes the instruction unconditional.

Table 3-3 lists the ALU conditions. For more information on conditional instructions, see [“Conditional Execution” on page 7-12](#).

Table 3-3. ALU Conditions

Condition	Description	Flags Set
AEQ	ALU equal to zero	AZ = 1
ALT	ALU less than zero	AN and AZ = 1
ALE	ALU less than or equal to zero	AN or AZ = 1
NAEQ	NOT (ALU equal to zero)	AZ = 0
NALT	NOT (ALU less than zero)	AN and AZ = 0
NALE	NOT (ALU less than or equal to zero)	AN or AZ = 0

ALU Static Flags

In the program sequencer, the static flag (SFREG) can store status flag values for later usage in conditional instructions. With SFREG, each compute block has two dedicated static flags X/YSCF0 (condition is SF0) and X/YSCF1 (condition is SF1). The following example shows how to load a compute block condition value into a static flag register.

```
XSCF0 = XAEQ ;; /* Load X-compute block SEQ flag into XSCF0 bit
in static flags (SFREG) register */
IF SF0, XR5 = R4 + R3 ;; /* the SF0 condition tests the XSCF0
static flag */
```

For more information on static flags, see [“Conditional Execution” on page 7-12](#).

ALU Examples

[Listing 3-1](#) provides a number of example ALU arithmetic instructions. The comments with the instructions identify the key features of the instruction, such as fixed- or floating-point format, input operand size, and register usage.

Listing 3-1. ALU Instruction Examples

```
LR5:4 = R11:10 + R1:0 ;;  
/* This is a fixed-point add of the 64-bit input operands  
XR11:10 + XR1:0 and YR11:10 + YR1:0; the DSP places the result in  
XR5:4 and YR5:4. */
```

```
YSR1:0 = R31:30 + R25:24 ;;  
/* This is a fixed-point add of the four 16-bit input operands  
YR31:30 and the four operands in YR25:24; the DSP places the four  
results in YR1:0. */
```

```
XR3 = R5 AND R7 ;;  
/* This is a logical AND of the 32-bit input operands XR5 and  
XR7; the DSP places the result in XR3. */
```

```
YR4 = SUM SR3:2 ;;  
/* This is a signed sideways sum of the four 16-bit input oper-  
ands in YR3:2; the DSP places the result in YR4. */
```

```
R9 = R4 + R8, R2 = R4 - R8 ;;  
/* This is a dual instruction (two instructions in one instruc-  
tion slot); the first instruction is a fixed-point add of the  
32-bit input operands XR4 + XR8 and YR4 + YR8; the DSP places the  
results in XR9 and YR9; the second instruction is a fixed-point  
subtract of the 32-bit input operands XR4 - XR8 and YR4 - YR8;  
the DSP places the results in XR2 and YR2. */
```

```
FR9 = R4 + R8 ;;
```

```
/* This is a floating-point add of the 32-bit input operands in  
XR4 +XR8 and YR4 + YR8; the DSP places the results in XR9 and  
YR9. */
```

```
XFR9:8 = R3:2 + R5:4 ;;
```

```
/* This is a floating-point add of the 40-bit (Extended Word)  
input operands in XR3:2 and XR5:4; the DSP places the result in  
XR9:8. */
```

ALU Examples

Example Parallel Addition of Byte Data

Figure 3-3 shows an ALU add using Byte input operands and dual registers. The syntax for the instruction is:

```
XBR11:10 = R9:8 + R7:6 ;;
```

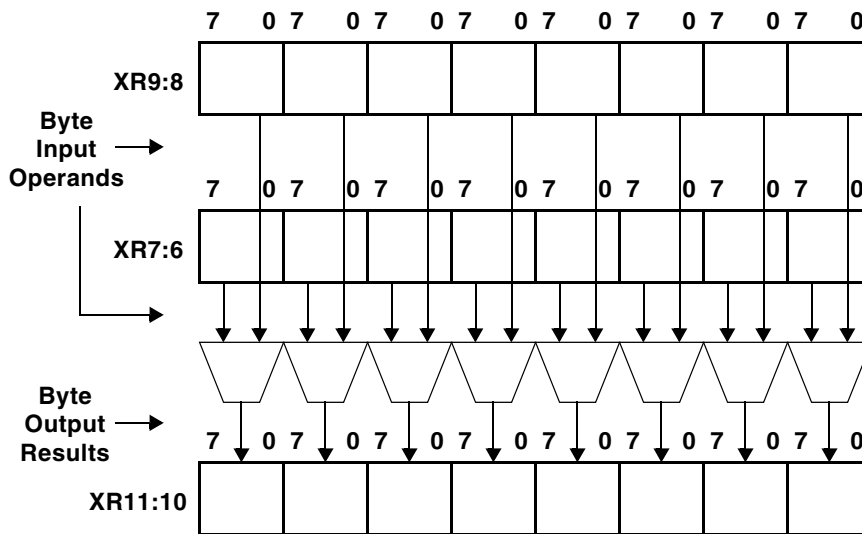


Figure 3-3. Input Operands for Parallel Add

It is important to note that the ALU processes the eight add operations independent of each other, but updates the arithmetic status based on an ORing of the status of all eight operations.

Example Sideways Addition of Byte Data

Figure 3-4 shows an ALU sideways sum using Byte input operands and a dual register. The syntax for the instruction is:

```
XR11 = SUM BR9:8 (U) ;; /* unsigned sideways sum */
```

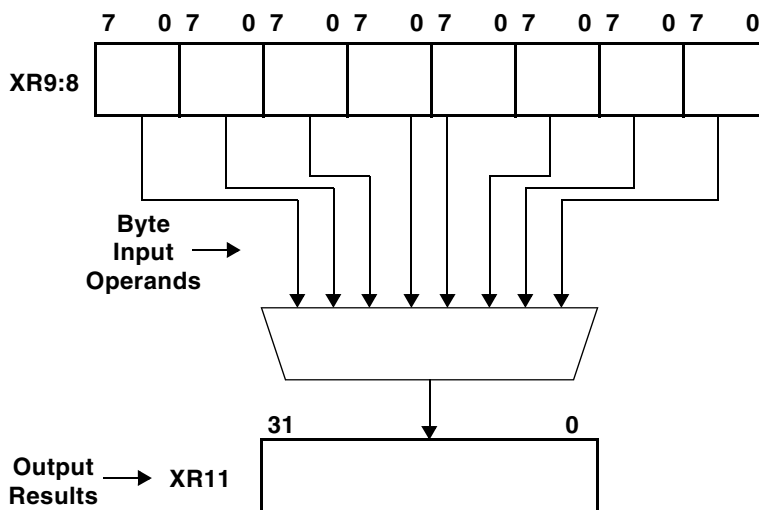


Figure 3-4. Input Operands for Sideways Add

Example Parallel Result (PR) Register Usage

The ALU supports a set of special instructions such as `SUM`, `VMAX`, `VMIN`, and `ABS` that can use the `PR1:0` register. The `PR1:0` register is an ALU register that is not memory mapped the way that data register file registers are mapped. To load or store, programs must load `PR1:0` from data registers,

ALU Examples

or store $PR1:0$ to data registers—there is no memory load or store for the $PR1:0$ register. To access the $PR1:0$ registers, the application must use instructions with the pseudo code:

```
PR1:0 = Rmd ;;  
Rsd = PR1:0 ;;
```



The instruction must operate on double registers even if only $PR0$ or $PR1$ is required.

The `SUM` instruction is one of the instructions that can use the $PR1:0$ register to hold parallel results. When using the $PR1:0$ register, the `SUM` instruction performs a short or byte wise parallel add of the input operands, adds this quantity to the contents of one of the PR registers, then stores the parallel result to the PR register.

This instruction performs four 16-bit additions and adds the result to the current contents of the PRO register.

```
PRO += SUM SR5:4;;
```

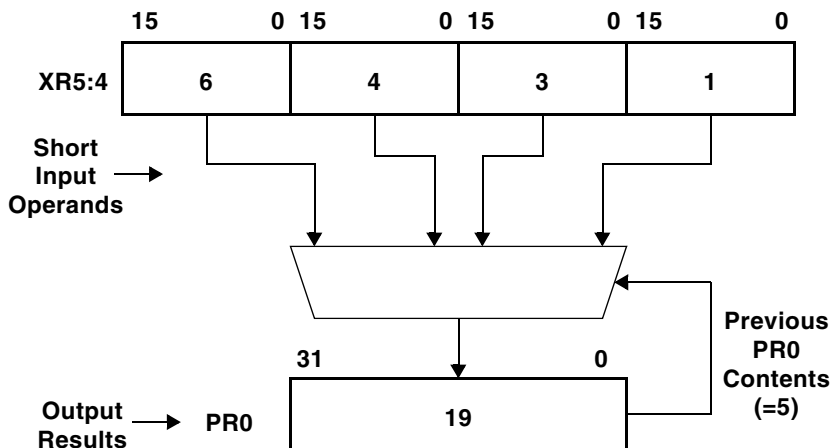


Figure 3-5. Input Operands for Parallel/Sideways Add

CLU Examples

The communications logic unit (CLU) instructions are designed to support different algorithms used for communications applications. These instructions were designed primarily with the following algorithms in mind (although many other uses are possible):

- Viterbi Decoding
- Decoding of turbo codes
- Despreading for code-division multiple access (CDMA) systems

CLU Examples

The inclusion of the CLU instructions simplifies the programming of these algorithms, yet still retains the flexibility of a software approach. In this way, it is easy to tune the algorithm according to a user's specific requirements. Additionally, the instructions can be used for a variety of purposes; for example, the `TMAX` instruction, included to support the decoding of turbo codes, is also very useful in the decoding of low-density parity-check codes.

The major strength of the TigerSHARC processor is the huge data transfer rate—two 128-bit memory accesses every cycle. For despreading, this enables 16 complex multiply-accumulate operations per cycle of 16-bit complex data (8-bit real, 8-bit imaginary). This enables calculation of a whole 16-bit 64-state trellis calculation every two cycles in both compute blocks together.

CLU Data Types and Sizes

For turbo and Viterbi decoding, the communications logic unit (CLU) input data sizes of 8- and 16-bit soft values are supported; output value data sizes of 16 and 32 bits are supported. Care should be taken when choosing the data size to prevent overflow in the calculation.

The `DESPREAD` function works with 16-bit complex numbers. Each 16-bit complex is composed of the real part (bits 7–0) and the imaginary part (bits 15–8). The result is always one or two complex words, each consisting of two shorts. Bits 15–0 represent the real part, and bits 31–16 represent the imaginary part (as complex numbers in the multiplier).

The CLU instructions refer to three types of registers:

- $R_{m,n,s}$ —register file (data) registers
- $TR_{m,n,s}$ —16 (trellis) registers that are dedicated to the CLU instructions
- THR —two (trellis history) registers used by the `ACS` and `DESPREAD` instructions for shifted data

TMAX Function

The $TMAX$ function is commonly used in the decoding of turbo codes and other high-performance error-correcting codes. The function is:

$$TMAX(A, B) = \max(A, B) + \ln(1 + e^{-|a-b|})$$

The second term is implemented as a table:

$$\ln(1 + e^{-|a-b|})$$

(for large $|a - b|$, \ln of 1 is 0).

The $TMAX$ table input is the result of the subtraction on clock high of the `execute1` (EX1) pipe stage. If the result is negative, it is inverted (assuming that the difference between one's complement and two's complement is within the allowed error). The input to [Table 3-4](#) is the seven LSB's of the compare subtract result. If the compare subtract result is larger than seven bits, the output is zero. Note that the implied decimal point is always placed before the five least significant bits.

[Table 3-4](#) shows the $TMAX$ values. The maximum output error is one LSB.

CLU Examples

Table 3-4. TMAX Values

Negative Input	Positive Input	Output (Hex)
11..111.1111X	000.0000X	000.10110
11..111.1110X	000.0001X	000.10101
11..111.1101X	000.0010X	000.10100
11..111.1100X	000.0011X	000.10011
11..111.1011X	000.0100X	000.10010
11..111.1010X	000.0101X	000.10001
11..111.100XX	000.011XX	000.10000
11..111.011XX	000.100XX	000.01110
11..111.010XX	000.101XX	000.01101
11..111.001XX	000.110XX	000.01100
11..111.000XX	000.111XX	000.01011
11..110.111XX	001.000XX	000.01010
11..110.110XX	001.001XX	000.01000
11..110.10XXX	001.01XXX	000.00111
11..110.01XXX	001.10XXX	000.00110
11..110.00XXX	001.11XXX	000.00101
11..101.1XXXX	010.0XXXX	000.00011
11..101.0XXXX	010.1XXXX	000.00010
11..100.XXXXX	011.XXXXX	000.00001
<= 11..10XX.XXXXX	>= 1XX.XXXXX	000.00000

Trellis Function

The trellis diagram is a widely-used tool in communications systems. For example, the Viterbi and turbo decoding algorithms both operate on trellises. The ADSP-TS101 processor provides specialized instructions for

trellises with binary transitions and up to eight states. Trellis with larger numbers of states can often be broken up into subtrellises with eight states or fewer and then applied to these instructions.

A typical trellis diagram is shown in [Figure 3-6 on page 3-26](#), and represents the set of possible state transitions from one stage to the next. At each stage n , we need to compute the accumulated metrics for every state. For a particular state, this value will depend on the metrics of each of its two possible previous states (at stage $n-1$) as well as transition metrics ($\gamma(n)$) corresponding to particular input/output combinations. The particular symmetry among the transition metrics shown in the figure is typical for practical error-correcting codes.

The ACS (Add-Compare-Select) instruction has options for two types of state metric computations. In the Viterbi algorithm, the metrics for each of the two possible previous state are updated, and the one with maximum value is selected. For example, the metric for state "100" (binary for 4) at stage n is computed as:

$$\begin{aligned} \text{Metrics}("100", n) = \\ \max((\text{Metrics}("010", n-1) - \gamma_3(n)), \\ (\text{Metrics}("110", n-1) + \gamma_3(n))) \end{aligned}$$

The ACS instruction computes the metrics for four or eight states in parallel, and additionally records information specifying the selected transitions for use in a trace back routine.

A second option, used in turbo decoding, replaces the MAX operation above with the TMAX operation defined in ["TMAX Function" on page 3-23](#).

CLU Examples

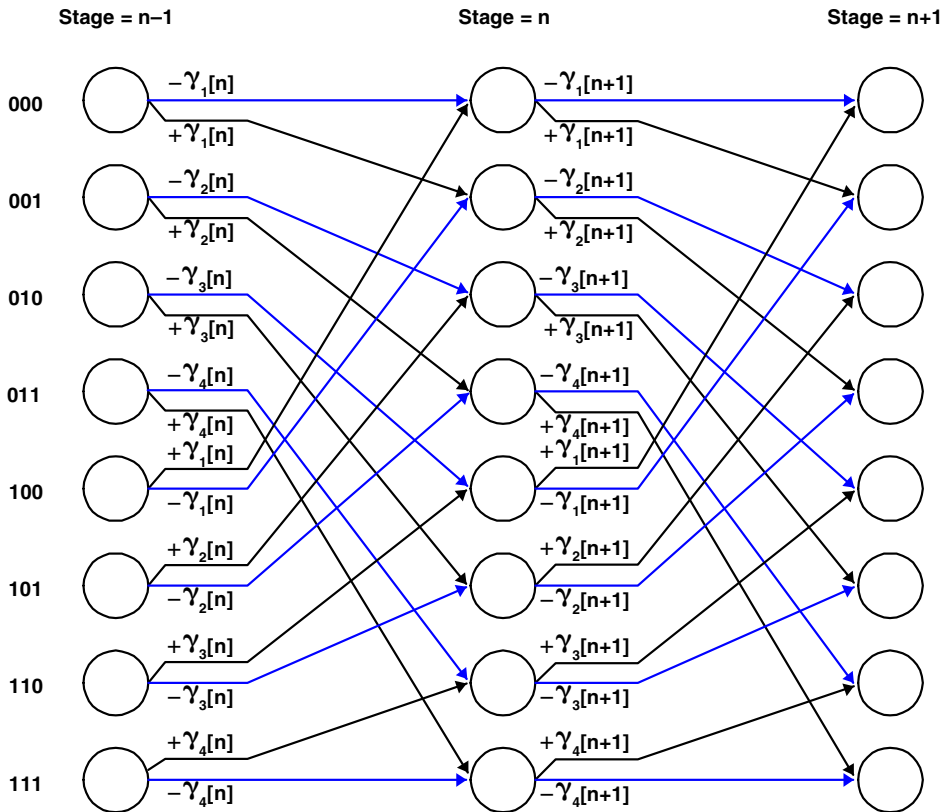


Figure 3-6. Trellis Diagram

Despread Function

The `DESPREAD` instruction implements a highly parallel complex multiply-and-accumulate operation that is optimized for CDMA systems. Despreading involves computing samples of a correlation between complex input data and a precomputed complex spreading/scrambling code sequence.

The input data consists of samples with 8-bit real and imaginary parts. The code sequence samples, on the other hand, are always members of $\{ 1+j, -1+j, -1-j, 1-j \}$, and are therefore specified by 1-bit real and imaginary parts. The `DESPREAD` instruction takes advantage of this property and is able to compute eight parallel complex multiply-and-accumulates in each block in a single cycle.

The `DESPREAD` instruction supports accumulations over lengths (spread factors) of four, eight, and multiples of eight samples.

CLU Execution Status

CLU operations update status flags in the compute block's Arithmetic Status (`XSTAT` and `YSTAT`) register (see [Figure 2-2 on page 2-4](#) and [Figure 2-3 on page 2-5](#)). Programs can use status flags to control execution of conditional instructions and initiate software exception interrupts.

[Table 3-5](#) shows the flags in `XSTAT` or `YSTAT` that indicate CLU status (a 1 indicates the condition) for the most recent CLU operation.

Table 3-5. CLU Status Flags

Flag	Definition	Updated By...
TROV	CLU overflow	All CLU ops

CLU operations also update sticky status flags in the compute block's Arithmetic Status (`XSTAT` and `YSTAT`) register. [Table 3-6](#) shows the flags in `XSTAT` or `YSTAT` that indicate CLU sticky status (a 1 indicates the condition) for the most recent CLU operation. Once set, a sticky flag remains high until explicitly cleared.

Table 3-6. CLU Status Sticky Flags

Flag	Definition	Updated By...
TRSOV	CLU overflow, sticky	All CLU ops

ALU Instruction Summary

Flag update occurs at the end of each operation and is available on the next instruction cycle. A program cannot write the arithmetic status register explicitly in the same cycle that the CLU is performing an operation.

Multi-operand instructions (for example, $STRsd = TMAX(TRmd + Rmq_h, TRnd + Rmq_l)$;) produce multiple sets of results. In this case, the DSP determines a flag by ORing the result flag values from individual results.

ALU Instruction Summary

The following listings show the ALU instructions' syntax:

- [Listing 3-2](#) “ALU Fixed-Point Instructions”
- [Listing 3-3](#) “ALU Logical Operation Instructions”
- [Listing 3-4](#) “ALU Fixed-Point Miscellaneous”
- [Listing 3-5](#) “Floating-Point ALU Instructions”
- [Listing 3-6](#) “Fixed-Point CLU Instructions”

The conventions used in these listings for representing register names, optional items, and choices are covered in detail in “[Register File Registers](#)” on page 2-5. Briefly, these conventions are:

- { } – the curly braces enclose options; these braces are not part of the instruction syntax.
- | – the vertical bars separate choices; these bars are not part of the instruction syntax.
- *Rmd* – the register names in italic represent user-selectable single (*Rs*, *Rm*, or *Rn*), double (*Rsd*, *Rmd*, or *Rnd*) or quad (*Rsq*, *Rmq*, or *Rnq*) register names.



Each instruction presented here occupies one instruction slot in an instruction line. For more information about instruction lines and instruction combination constraints, see “[Instruction Line Syntax and Structure](#)” on page 1-20 and “[Instruction Parallelism Rules](#)” on page 1-24.

Listing 3-2. ALU Fixed-Point Instructions

```

{X|Y|XY}{S|B}Rs = Rm +|- Rn {{(S|SU)}} ;1
{X|Y|XY}{L|S|B}Rsd = Rmd +|- Rnd {{(S|SU)}} ;1
{X|Y|XY}Rs = Rm + CI {-1} ;
{X|Y|XY}LRsd = Rmd + CI {-1} ;
{X|Y|XY}{S|B}Rs = Rm +|- Rn + CI {-1} {{(S|SU)}} ;1
{X|Y|XY}{L|S|B}Rsd = Rmd +|- Rnd + CI {-1} {{(S|SU)}} ;1
{X|Y|XY}{S|B}Rs = (Rm +|- Rn)/2 {{(T){U}}} ;2
{X|Y|XY}{L|S|B}Rsd = (Rmd +|- Rnd)/2 {{(T){U}}} ;2
{X|Y|XY}{S|B}Rs = ABS Rm ;
{X|Y|XY}{L|S|B}Rsd = ABS Rmd ;

```

¹ Options include: (): no saturation, (S): saturation, signed, (SU): saturation, unsigned

² Options include: (): signed, round-to-nearest even, (T): signed, truncate, (U): unsigned, round-to-nearest even, (TU): unsigned, truncate

ALU Instruction Summary

```

{X|Y|XY}{S|B}Rs = ABS (Rm + Rn) {(X)} ;1
{X|Y|XY}{L|S|B}Rsd = ABS (Rmd + Rnd) {(X)} ;1
{X|Y|XY}{S|B}Rs = ABS (Rm - Rn) {(X){U}} ;2
{X|Y|XY}{L|S|B}Rsd = ABS (Rmd - Rnd) {(X){U}} ;2
{X|Y|XY}{S|B}Rs = - Rm ;
{X|Y|XY}{L|S|B}Rsd = - Rmd ;
{X|Y|XY}{S|B}Rs = MAX|MIN (Rm, Rn) {(U){Z}} ;3
{X|Y|XY}{L|S|B}Rsd = MAX|MIN (Rmd, Rnd) {(U){Z}} ;3
{X|Y|XY}S|BRsd = VMAX|VMIN (Rmd, Rnd) ;
{X|Y|XY}{S|B}Rs = INC|DEC Rm {(S|SU)} ;1
{X|Y|XY}{L|S|B}Rsd = INC|DEC Rmd {(S|SU)} ;1
{X|Y|XY}{S|B}COMP(Rm, Rn) {(U)} ;3
{X|Y|XY}{L|S|B}COMP(Rnd,Rnd) {(U)} ;3
{X|Y|XY}{S|B}Rs = CLIP Rm BY Rn ;
{X|Y|XY}{L|S|B}Rsd = CLIP Rmd BY Rnd ;
{X|Y|XY}Rs = SUM S|B Rm {(U)} ;4
{X|Y|XY}Rs = SUM S|B Rmd {(U)} ;4
{X|Y|XY}Rs = ONES Rm|Rmd ;
{X|Y|XY}PR1:0 = Rmd ;
{X|Y|XY}Rsd = PR1:0 ;
{X|Y|XY}Rs = BFOINC Rmd ;
{X|Y|XY}PRO|PR1 += ABS (SRmd - SRnd){(U)} ;4
{X|Y|XY}PRO|PR1 += ABS (BRmd - BRnd){(U)} ;4
{X|Y|XY}PRO|PR1 += SUM SRm {(U)} ;4
{X|Y|XY}PRO|PR1 += SUM SRmd {(U)} ;4
{X|Y|XY}PRO|PR1 += SUM BRm {(U)} ;4
{X|Y|XY}PRO|PR1 += SUM BRmd {(U)} ;4

```

¹ Options include: (X): extend for ABS

² Options include: (X): extend for ABS, (U): unsigned, round-to-nearest even, (XU): unsigned, extend

³ Options include: (): regular signed comparison, (U): comparison between unsigned numbers, (Z): returned result is zero if Rn is selected by MIN/MAX operation; otherwise returned result is Rm, (UZ): unsigned comparison with option (Z) as described above

⁴ Options include: (): signed, (U): unsigned

$\{X|Y|XY\}\{S|B\}Rs = Rm + Rn, Ra = Rm - Rn ; (dual\ operation)$
 $\{X|Y|XY\}\{L|S|B\}Rsd = Rmd + Rnd, Rad = Rmd - Rnd ; (dual\ operation)$

Listing 3-3. ALU Logical Operation Instructions

$\{X|Y|XY\}Rs = PASS Rm ;$
 $\{X|Y|XY\}LRsd = PASS Rmd ;$
 $\{X|Y|XY\}Rs = Rm AND|AND NOT|OR|XOR Rn ;$
 $\{X|Y|XY\}LRsd = Rmd AND|AND NOT|OR|XOR Rnd ;$
 $\{X|Y|XY\}Rs = NOT Rm ;$
 $\{X|Y|XY\}LRsd = NOT Rmd ;$

Listing 3-4. ALU Fixed-Point Miscellaneous

$\{X|Y|XY\}Rsd = EXPAND SRm \{+|- SRn\} \{(\{I|IU\})\} ;^1$
 $\{X|Y|XY\}Rsq = EXPAND SRmd \{+|- SRnd\} \{(\{I|IU\})\} ;^1$
 $\{X|Y|XY\}Rsd = EXPAND BRm \{+|- BRn\} \{(\{I|IU\})\} ;^1$
 $\{X|Y|XY\}Rsq = EXPAND BRmd \{+|- BRnd\} \{(\{I|IU\})\} ;^1$
 $\{X|Y|XY\}SRs = COMPACT Rmd \{+|- Rnd\} \{(\{T|I|IS|ISU\})\} ;^2$
 $\{X|Y|XY\}BRs = COMPACT SRmd \{+|- SRnd\} \{(\{T|I|IS|ISU\})\} ;^2$
 $\{X|Y|XY\}BRsd = MERGE Rm, Rn ;$
 $\{X|Y|XY\}BRsq = MERGE Rmd, Rnd ;$
 $\{X|Y|XY\}SRsd = MERGE Rm, Rn ;$
 $\{X|Y|XY\}SRsq = MERGE Rmd, Rnd ;$

Listing 3-5. Floating-Point ALU Instructions

$\{X|Y|XY\}FRs = Rm +|- Rn \{(T)\} ;^3$
 $\{X|Y|XY\}FRsd = Rmd +|- Rnd \{(T)\} ;^3$

¹ Options include: (): fractional, (I): integer signed, (IU): integer unsigned

² Options include: (): fractional round, (I): integer, no saturate, (T): fractional, truncate, (IS): integer, saturate, signed, (ISU): integer, saturate, unsigned

³ Options include: (): round, (T): truncate

ALU Instruction Summary

$\{X|Y|XY\}FRs = (Rm +|- Rn)/2 \{(T)\} ;^3$
 $\{X|Y|XY\}FRsd = (Rmd +|- Rnd)/2 \{(T)\} ;^3$
 $\{X|Y|XY\}FRs = MAX|MIN (Rm +|- Rn) \{(T)\} ;^1$
 $\{X|Y|XY\}FRsd = MAX|MIN (Rmd +|- Rnd) \{(T)\} ;^3$
 $\{X|Y|XY\}FRs = ABS (Rm) ;$
 $\{X|Y|XY\}FRsd = ABS (Rmd) ;$
 $\{X|Y|XY\}FRs = ABS (Rm +|- Rn) \{(T)\} ;^3$
 $\{X|Y|XY\}FRsd = ABS (Rmd +|- Rnd) \{(T)\} ;^3$
 $\{X|Y|XY\}FRs = - Rm ;$
 $\{X|Y|XY\}FRsd = - Rmd ;$
 $\{X|Y|XY\}FCOMP (Rm, Rn) ;$
 $\{X|Y|XY\}FCOMP (Rmd, Rnd) ;$
 $\{X|Y|XY\}Rs = FIX FRm|FRmd \{BY Rn\} \{(T)\} ;^3$
 $\{X|Y|XY\}FRs|FRsd = FLOAT Rm \{BY Rn\} \{(T)\} ;^3$
 $\{X|Y|XY\}FRsd = EXT D Rm ;$
 $\{X|Y|XY\}FRs = SNGL Rmd ;$
 $\{X|Y|XY\}FRs = CLIP Rm BY Rn ;$
 $\{X|Y|XY\}FRsd = CLIP Rmd BY Rnd ;$
 $\{X|Y|XY\}FRs = Rm COPYSIGN Rn ;$
 $\{X|Y|XY\}FRsd = Rmd COPYSIGN Rnd ;$
 $\{X|Y|XY\}FRs = SCALB FRm BY Rn ;$
 $\{X|Y|XY\}FRsd = SCALB FRmd BY Rn ;$
 $\{X|Y|XY\}FRs = PASS Rm ;$
 $\{X|Y|XY\}FRsd = PASS Rmd ;$
 $\{X|Y|XY\}FRs = RECIPS Rm ;$
 $\{X|Y|XY\}FRsd = RECIPS Rmd ;$
 $\{X|Y|XY\}FRs = RSQRTS Rm ;$
 $\{X|Y|XY\}FRsd = RSQRTS Rmd ;$
 $\{X|Y|XY\}Rs = MANT FRm|FRmd ;$
 $\{X|Y|XY\}Rs = LOGB FRm|FRmd \{(S)\} ;^2$

¹ Options include: (): round, (T): truncate (MIN only)

² Options include: (): do not saturate, (S): saturate

$\{X|Y|XY\}FRs = Rm + Rn, FRa = Rm - Rn ;$ (*dual instruction*)
 $\{X|Y|XY\}FRsd = Rmd + Rnd, FRad = Rmd - Rnd ;$ (*dual instruction*)

Listing 3-6. Fixed-Point CLU Instructions

$\{X|Y|XY\}\{S\}TRsd = TMAX(TRmd + Rmq_h, TRnd + Rmq_l) ;$
 $\{X|Y|XY\}\{S\}TRsd = TMAX(TRmd - Rmq_h, TRnd - Rmq_l) ;$
 $\{X|Y|XY\}\{S\}Rs = TMAX(TRm, TRn) ;$
 $\{X|Y|XY\}\{S\}TRsd = MAX(TRmd + Rmq_h, TRnd + Rmq_l) ;$
 $\{X|Y|XY\}\{S\}TRsd = MAX(TRmd - Rmq_h, TRnd - Rmq_l) ;$
 $\{X|Y|XY\}Rs = TRm ;$
 $\{X|Y|XY\}Rsd = TRmd ;$
 $\{X|Y|XY\}Rs q = TRmq ;$
 $\{X|Y|XY\}TRs = Rm ;$
 $\{X|Y|XY\}TRsd = Rmd ;$
 $\{X|Y|XY\}TRsq = Rmq ;$
 $\{X|Y|XY\}Rs = THRm ;$
 $\{X|Y|XY\}Rsd = THRmd ;$
 $\{X|Y|XY\}Rs q = THRmq ;$ ¹
 $\{X|Y|XY\}THR s = Rm ;$
 $\{X|Y|XY\}THRsd = Rmd \{(i)\} ;$
 $\{X|Y|XY\}THRsq = Rmq ;$ ¹
 $\{X|Y|XY\}TRs = DESPREAD (Rmq, THRd) + TRn ;$
 $\{X|Y|XY\}Rs = TRs, TRs = DESPREAD (Rmq, THRd) ;$ (*dual instruction*)
 $\{X|Y|XY\}Rsd = TRsd, TRsd = DESPREAD (Rmq, THRd) ;$ (*dual instruction*)
 $\{X|Y|XY\}\{S\}TRsq = ACS (TRmd, TRnd, Rm) (TMAX) ;$
 $\{X|Y|XY\}Rs q = TRa q, \{S\}TRsq = ACS (TRmd, TRnd, Rm) (TMAX) ;$ (*dual instr.*)
 $\{X|Y|XY\}Rsd = PERMUTE (Rmd, Rn) ;$
 $\{X|Y|XY\}Rs q = PERMUTE (Rmd, -Rmd, Rn) ;$

¹ Not implemented, but syntax reserved

ALU Instruction Summary

4 MULTIPLIER

The TigerSHARC processor core contains two computation units known as compute blocks. Each compute block contains a register file and three independent computation units—an ALU, a multiplier, and a shifter. The multiplier is highlighted in Figure 4-1. The multiplier takes its inputs from the register file, and returns its outputs to the register file.

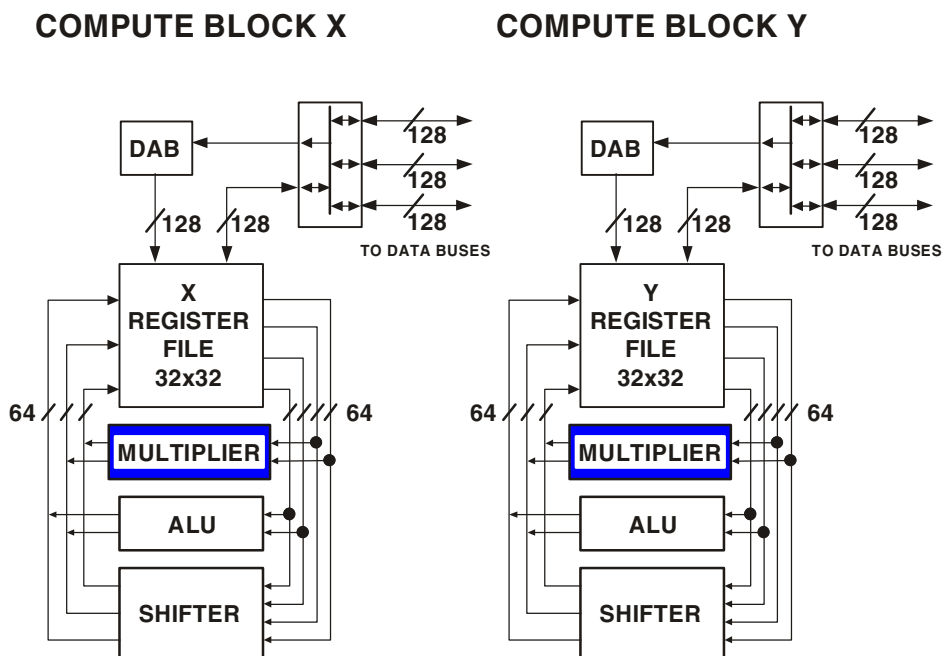


Figure 4-1. Multipliers in Compute Block X and Y

The multiplier performs all *multiply operations* for the processor on fixed- and floating-point data and performs all *multiply-accumulate operations* for the processor on fixed-point data. This unit also performs all *complex multiply operations* for the processor on fixed-point data. The multiplier also executes *data compaction operations* on accumulated results when moving data to the register file in fixed-point formats.

Examining the supported operands for each operation shows that the multiplier operations support these data types:

- Fixed-point fractional and integer *multiply operations* and *multiply-accumulate operations* support:
 - Eight 16-bit (short) input operands with four 16- or 32-bit results
 - Two 32-bit (normal) input operands with a 32- or 64-bit result
- Floating-point fractional *multiply operations* support:
 - Two 32-bit (normal) input operands (IEEE standard) with 32-bit result
 - Two 40-bit (extended) input operands with 40-bit result
- Fixed-point *data compaction operations* support:
 - 16-bit (short) input operands
 - 32-bit (normal) input operands
 - 64-bit (long) input operands
 - output 16- or 32-bit results

Fixed-point formats include these data size distinctions:

- The multiplier can operate on two 32-bit normal words producing either a 64-bit or a 32-bit result or operate on eight 16-bit short words producing either four 32-bit or four 16-bit results. There is no byte word support in the multiplier.
- The result of a multiplier operation (with the exception of the compress instruction) is always either the same size as the operands, or larger.
 - Normal word multiplication results in either a normal word or a long word result.
 - Quad short word multiplication always results in either a quad short-word or a quad-word results.

The TigerSHARC processor supports complex multiply-accumulates. Complex numbers are represented by a pair of short words in a 32-bit register. The least significant bits of the input operands (Rm_L , Rn_L) represent the real part, and the most significant bits of the input operands (Rm_H , Rn_H) represent the imaginary part. The result of a complex multiplication is always stored in a pair of MR registers. The complex multiply-accumulate (indicated with the ** operator) is defined as follows:

$$Real\ Result = (Real_{Rm_L} \times Real_{Rn_L}) - (Imaginary_{Rm_H} \times Imaginary_{Rn_H})$$

$$Imaginary\ Result = (Real_{Rm_L} \times Imaginary_{Rn_H}) + (Imaginary_{Rm_H} \times Real_{Rn_L})$$

Complex multiply-accumulate operations have an option to multiply the first complex operand times the complex conjugate of the second. This complex conjugate operation is defined as:

$$Real\ Result = (Real_{Rm_L} \times Real_{Rn_L}) + (Imaginary_{Rm_H} \times Imaginary_{Rn_H})$$


$$Imaginary\ Result = (Real_{Rm_L} \times Imaginary_{Rn_H}) - (Imaginary_{Rm_H} \times Real_{Rn_L})$$

Multiplier Operations

The complex conjugate option is denoted with a (J) following the instruction. (See “[Complex Conjugate Option](#)” on page 4-16.)

The TigerSHARC processor is compatible with the IEEE 32-bit single-precision floating-point data format with minor exceptions. For more information, see “[IEEE Single-Precision Floating-Point Data Format](#)” on page 2-16.

Within instructions, the register name syntax identifies the input operand and output result data size and type. For information on data type selection for multiplier instructions, see “[Register File Registers](#)” on page 2-5. For information on data size selection for multiplier instructions, see the examples in “[Multiplier Operations](#)” on page 4-4.

 Note that multiplier instruction conventions for selecting input operand and output result data size differ slightly from the conventions for the ALU and shifter.

The remainder of this chapter presents descriptions of multiplier instructions, options, and results using instruction syntax. For an explanation of the instruction syntax conventions used in multiplier and other instructions, see “[Instruction Line Syntax and Structure](#)” on page 1-20. For a list of multiplier instructions and their syntax, see “[Multiplier Instruction Summary](#)” on page 4-23.

Multiplier Operations

The multiplier performs fixed-point or floating point multiplication and fixed-point multiply-accumulate operations. The multiplier supports several data types in fixed- and floating-point. The floating-point formats are float and float-extended. The input operands and output result of most operations is the compute block register file.

The multiplier has one special purpose, five-word register—the MR register—for accumulated results. The DSP uses the MR register to store the results of fixed-point multiply-accumulate operations. Also, the multiplier can transfer the contents of the MR register to the register file before an accumulate operation. The upper 32 bits of the MR register (MR4) store overflow for multiply accumulate operations. For more information on the register files and register naming syntax for selecting data type and width, see “Register File Registers” on page 2-5.

Figure 4-2 on page 4-7 through Figure 4-4 on page 4-8 show the data flow for multiplier operations. The following are multiplier instructions that demonstrate multiply and multiply-accumulate operations.

```
XR2 = R1 * R0 ;;
/* This is a fixed-point multiply of two signed fractional 32-bit
input operands XR1 and XR0; the DSP places the 32-bit result in
XR2. */
```

```
YR1:0 = R3 * R2 ;;
/* This is a fixed-point multiply of two signed fractional 32-bit
input operands YR3 and YR2; the DSP places the 64-bit result in
YR1:0. */
```




For fixed-point multiply operations, single register names (R_m , R_n) for input operands select 32-bit input operands. Single versus double register names output for select 32- versus 64-bit output results.

```
XR3:2 = R5:4 * R1:0 ;;
/* This is a fixed-point multiply of eight signed fractional
16-bit operands:
XR5_upper_half * XR1_upper_half,
XR5_lower_half * XR1_lower_half,
XR4_upper_half * XR0_upper_half,
XR4_lower_half * XR0_lower_half;
The DSP places the four 16-bit results in XR3_upper_half,
XR3_lower_half, XR2_upper_half, and XR2_lower_half. */
```


Multiplier Operations

```
YR3:0 = R7:6 * R5:4 ;;  
/* This is similar to the previous example of a quad 16-bit multiply, but the selection of a quad register for output produces 32-bit (instead of 16-bit) results; the DSP places the four results in YR3, YR2, YR1, and YR0. */
```

 For fixed-point multiply operations, double register names (*Rmd*, *Rnd*) for input operands select 16-bit input operands. Double versus quad register names for output select 16- versus 32-bit output results.

```
XFR2 = R1 * R0 ;;  
/* This is a floating-point multiply of two fractional 32-bit input operands XR1 and XR0 (IEEE format); the DSP places the 32-bit result in XR2. */
```

```
YFR1:0 = R5:4 * R3:2 ::  
/* This is a floating-point multiply of two signed fractional 40-bit input operands YR5:4 and YR3:2; the DSP places the 40-bit result in YR1:0. */
```

 For floating-point multiply operations, single register names (*Rm*, *Rn*) for input operands select 32-bit input operands and 32-bit output result. For floating-point multiply operations, double register names (*Rmd*, *Rnd*, *Rsd*) for input and output operands select 40-bit input operands and 40-bit output result.

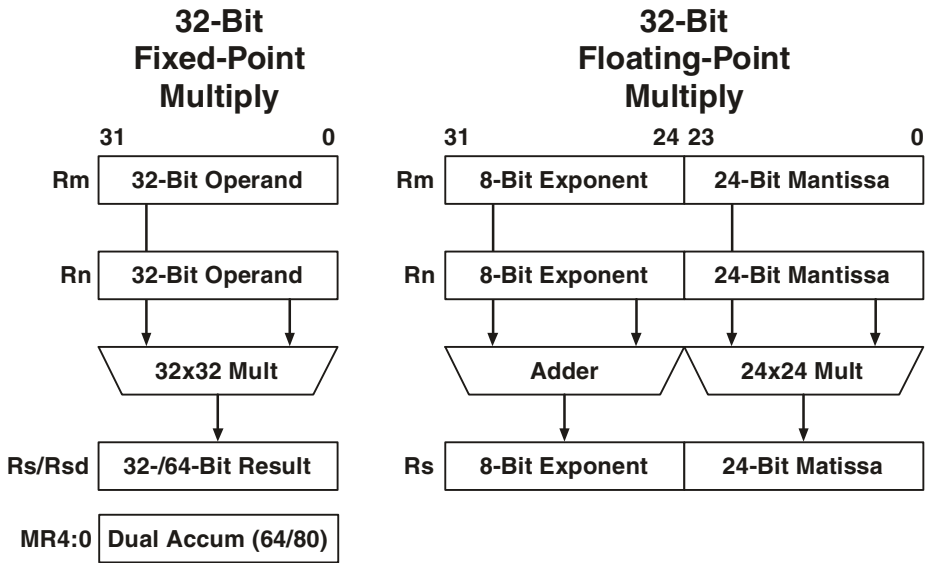


Figure 4-2. 32-Bit Multiplier Operations

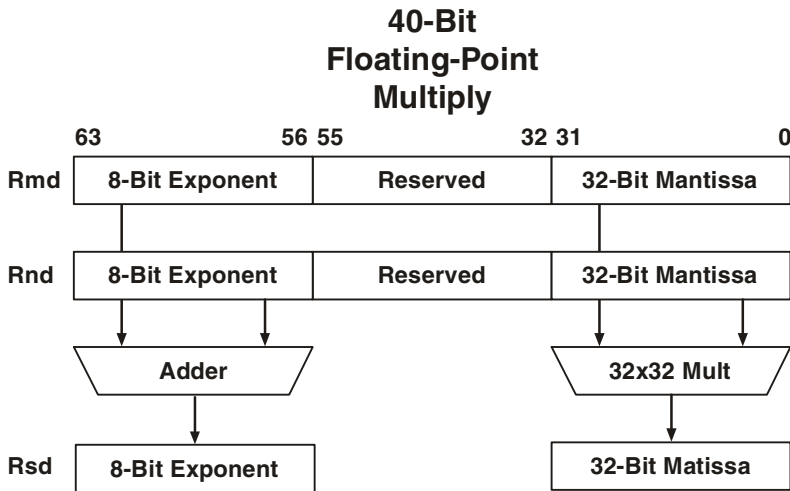


Figure 4-3. 40-Bit Multiplier Operations

Multiplier Operations

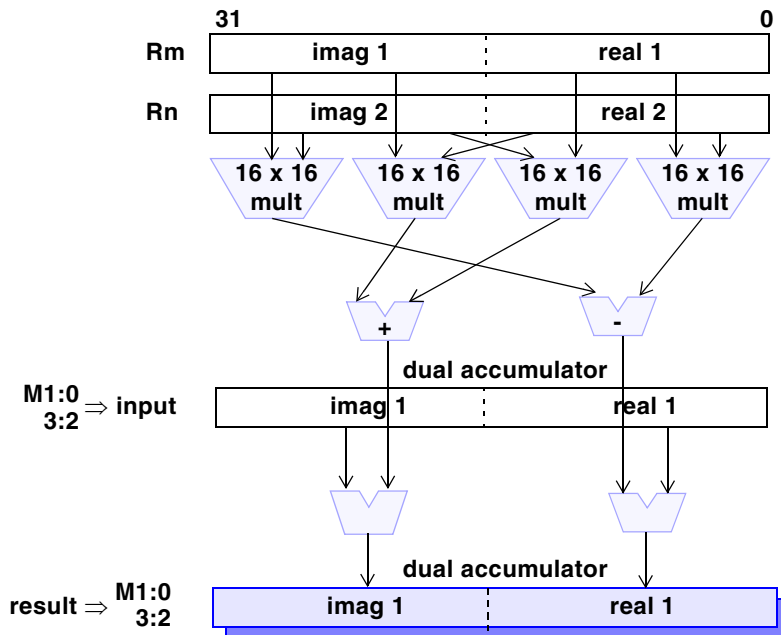


Figure 4-4. 16-Bit Complex Multiplier Operations

Multiplier Instruction Options

Most of the multiplier instructions have options associated with them that permit flexibility in how the instructions execute. It is important to note that these options modify the detailed execution of instructions and options that are particular to a group of instructions—not all options are applicable to all instructions. Instruction options appear in parenthesis at

the end of the instruction's slot. For a list indicating which options apply for particular multiplier instructions, see “Multiplier Instruction Summary” on page 4-23. The multiplier instruction options include:

- () signed operation, no saturation¹, round-to-nearest even², fractional mode³
- (U) unsigned operation, no saturation¹, round-to-nearest even²
- (nU) signed/unsigned input
- (I) signed operation, integer mode⁴
- (S) signed operation, saturation¹
- (T) signed operation, truncation⁴
- (C) clear operation
- (CR) clear/round operation
- (J) complex conjugate operation

The following are multiplier instructions that demonstrate multiply and multiply-accumulate operations with options applied.

```
XR2 = R1 * R0 (U) ;;  
/* This is a fixed-point multiply of two unsigned fractional  
32-bit input operands XR1 and XR0; the DSP places the unsigned  
32-bit result in XR2. */
```

¹ Where saturation applies

² Where rounding applies

³ Where applies for floating-point operations

⁴ Where truncation applies

Multiplier Operations

```
YR1:0 = R3 * R2 (I) ::  
/* This is a fixed-point multiply of two integer 32-bit input  
operands YR3 and YR2; the DSP places the 64-bit result in YR1:0.  
*/  
  
XFR2 = R1 * R0 (T) ;;  
/* This is a floating-point multiply of two fractional 32-bit  
input operands XR1 and XR0 (IEEE format); the DSP places the  

```

Signed/Unsigned Option

The DSP always represents fixed-point numbers in 8, 16, 32, or 64 bits, using up to four 32-bit data registers. Fixed-point data in the multiplier may be unsigned or two's-complement. Floating-point data in the multiplier is always signed (two's-complement). For information on the supported numeric formats, see [“Numeric Formats” on page 2-16](#).

All fixed-point multiplier instructions may use signed or unsigned data types. The options are:

- () Both input operands signed (default)
- (U) Both input operands unsigned
- (nU) Rm is signed, Rn is unsigned; this option is valid only
for $R_s = R_m * R_n$ or $R_{sd} = R_m * R_n$

Fractional/Integer Option

The DSP always represents fixed-point numbers in 8, 16, 32, or 64 bits, using up to four 32-bit data registers. In the multiplier, fractional or integer format is available for the fixed-point multiply, multiply-accumulate and COMPACT instructions. Floating-point multiply operations use fractional format. For information on the supported numeric formats, see [“Numeric Formats” on page 2-16](#).

The integer and fractional option are defined for the fixed-point operations:

- () Data is fractional (default)
- (I) Data is integer

Saturation Option

Saturation is performed on fixed-point operations if option (S) is active when the result *overflows*—crosses the maximum value in which the result format can be represented. In these cases, the returned result is the extreme value that can be represented in the format of the operation following the direction of the correct result. For example, if the format of the result is 16-bit signed and the full result is -0×100000 , the saturated result would be 0×8000 . If the operation is unsigned, the result would be 0×0 . This can occur in three types of operations:

- Multiply operations

When multiplying integer data and the actual result is outside the range of the destination format, the largest representable number in the destination format is returned. When multiplying fractional data, the special case of -1 times -1 (for example, $0 \times 80 \dots 0$ times $0 \times 80 \dots 0$) always returns the largest representable fraction (for example, $0 \times 7F \dots F$).

- Multiply-accumulate operations

Saturation affects both integer and fractional data types. Accumulation values are kept at 80-, 40-, and 20-bit precision and are stored in the combination of MR3:0 and MR4 registers (See “[Multiplier Examples](#)” on page 4-21). When performing saturation in a multiply-accumulate operation, the resulting value out of the multiplier (at 64, 32, or 16 bits) is added to the current accumulation

Multiplier Operations

value. When the accumulation value overflows past 80, 40, or 20 bits, it is substituted by the maximum or minimum possible value. Note that multiply-accumulate operations always saturate.

- MR register transfers

See “Multiplier Examples” on page 4-21.

The final saturated result at 32 --bits for all operations is:

- $0x7F...F$ – if operation is signed and result is positive
- $0x80...0$ – if operation is signed and result is negative
- $0xFF...F$ – if operation is unsigned and result is positive
- $0x00...0$ – if operation is unsigned and result is negative (only in signed $MR = R_m * R_n$)

Saturation option exists for any fixed-point multiplications that may overflow. The following options are available:

- () No saturation (default)
- (S) Saturation is active

Truncation Option

For multiplier instructions that support truncation as the T option, this option permits selection of the results rounding mode. The DSP supports two modes of rounding—round-toward-zero and round-toward-nearest. The rounding modes comply with the IEEE 754 standard and have these definitions:

- Round-Toward-Nearest (not using T option). If the result before rounding is not exactly representable in the destination format, the rounded result is the number that is nearer to the result before

rounding. If the result before rounding is exactly halfway between two numbers in the destination format (differing by an LSB), the rounded result is the number that has an LSB equal to zero.

- Round-Toward-Zero (using T option). If the result before rounding is not exactly representable in the destination format, the rounded result is the number that is nearer to zero. This is equivalent to truncation.

Statistically, rounding up occurs as often as rounding down, so there is no large sample bias. Because the maximum floating-point value is one LSB less than the value that represents Infinity, a result that is halfway between the maximum floating-point value and Infinity rounds to Infinity in this mode.

Though these rounding modes comply with standards set for floating-point data, they also apply for fixed-point multiplier operations on fractional data. The same two rounding modes are supported, but only the round-to-nearest operation is actually performed by the multiplier. Using its local result register for fixed-point operations, the multiplier rounds-to-zero by reading only the upper bits of the result and discarding the lower bits.

Round-to-nearest even is executed by adding the LSB to the truncated result if the first bit under the LSB is set and all bits under are cleared—for example, multiplying two short operands. The real result is a normal word. If the expected result is short, it has to be rounded.

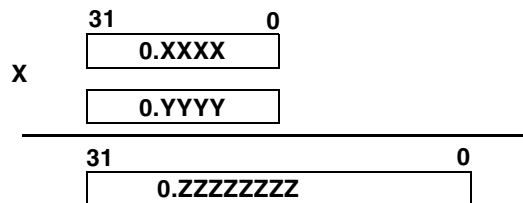


Figure 4-5. Rounding Multiplier Results

Multiplier Operations

Bits 31–16 should be returned, and bits 15–0 should be rounded. The rounding is set according to bit 15 (*round bit*), bit 16 (LSB), and whether bits 14–0 (lower bits) are zero or non-zero:

- If bit 15 is zero, the result is not incremented
- If bit 15 is 1 and bits 14–0 are non-zero, the result is incremented by one.
- If bit 15 is 1 and bits 14–0 are zero, add bit 16 to the result 31–16.

There is no support for round-to-nearest even for multiply-accumulate instructions that also transfer the current MR contents to the register file. If round-to-nearest even is required, transfer the MR registers to the register file as a whole and use the ALU COMPACT instruction. As an alternative, if round-to-nearest (not even) is sufficient, this can be achieved by using the clear and round option in the first multiply-accumulate instruction of the series. [For more information, see “Clear/Round Option” on page 4-14.](#)

Rounding options are:

- | | |
|-----|---------------------------------------------------------------------------------------|
| () | Round-to-nearest even (default) |
| (T) | Truncate—round-to-zero for floating-point and round-to-minus infinity for fixed-point |


Clear/Round Option

Multiply operations and multiply-accumulate with MR register move operations support the clear MR (C) option. Using this option forces the multiplier to clear (=0) the MR register before the accumulate operation.

Multiply-accumulate operations (without an MR move) also support the clear and round (CR) options as an alternative to the clear option. Using the CR option forces the multiplier to clear MR and set the round bit before the accumulate operation. For more information about rounding and the round bit, see [“Truncation Option” on page 4-12.](#)

The clear and clear/round options are:

- () No change of MR register prior to multiply-accumulate operation (default)
- (C) Set target MR to zero prior to multiply-accumulate operation
- (CR) Set target MR to zero and set round bit prior to multiply-accumulate operation

 The C and CR options may be used only for fractional arithmetic, for example when the option (I) is not used.

When this option is set, the MR registers are set to an initial value of 0x00000000 80000000 for 32-bit fractional multiply-accumulate and 0x00008000 in each of the MR registers for quad 16-bit fractional multiply-accumulate. After this initialization, the result is rounded up by storing the upper part of the result in the end of the multiply-accumulate sequence.

For example with the C option, assume a sequence of three quad short fractional multiply-accumulate operations (with quad short result) such that the multiplication results are:

```
Result 1 = 0x0024 0048, 0x0629 4501
Result 2 = 0x0128 0128, 0x2470 2885
Result 3 = 0x1011 fffe, 0x4A30 6d40
Sum      = 0x115d 016e, 0x74c9 dac6
```

In this example, the bottom 16 bits are not to be used if only a short result is expected. Extracting the top 16 bits will give a truncated result, which is 0x115d for the first short and 0x74c9 for the second short. The rounded

Multiplier Operations

result is 0x115d for the first short (no change) and 0x74ca (increment) for the second short. If the MR registers are initialized to 0x00008000, the sum result will be:

```
Sum          = 0x115d 816e, 0x74ca 5ac6
```

The top short is exactly the expected result. Note that this is round-to-nearest, and not round-to-nearest even.

For example with the CR option, assume a sequence of three quad short fractional multiply-accumulate operations, such that the multiplication results are:

```
Result 1 = 0x0024 0048, 0x0629 4501
```

```
Result 2 = 0x0128 0128, 0x2470 2885
```

```
Result 3 = 0x1011 FFFE, 0x4A30 6D40
```

```
Sum       = 0x115D 016E, 0x74C9 DAC6
```

In this example, if a short result is expected, the multiplier does not use the lower 16 bits. Extracting the upper 16 bits produces a truncated result, which is 0x115D for the first short and 0x74C9 for the second short. The rounded result is 0x115D for the first short (no change) and 0x74CA (increment of one) for the second short. If the MR registers are initialized to 0x0000 8000, the sum result is:

```
SUM       = 0x115D 816E, 0x74CA 5AC6
```

The high short is exactly as expected. The rounding method is round-to-nearest, not round-to-nearest even. For information on rounding, see [“Truncation Option” on page 4-12](#).

Complex Conjugate Option

For complex multiply-accumulate operations (** operator), the multiplier supports the complex conjugate (J) option. The J option directs the multiplier to multiply the R_m operand with the complex conjugate of R_n

operand, negating the imaginary part of R_n . For more information, see the discussion on page 4-3. The options are:

- () No conjugate
- (J) Conjugate for complex multiply

Multiplier Result Overflow (MR4) Register

The MR4 register holds the extra bits (overflow) from a multiply-accumulate operation. MR4 register fields are assigned according to the MR register used and the size of the result. See:

- [Figure 4-6](#)—Result is a long word (80-bit accumulation)
- [Figure 4-7](#)—Result is word (40-bit accumulation)
- [Figure 4-8](#)—Result is short (20-bit accumulation)

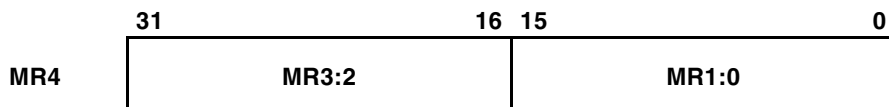


Figure 4-6. MR4 for Long Word Result (80-Bit Accumulation)

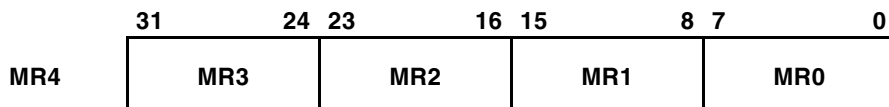
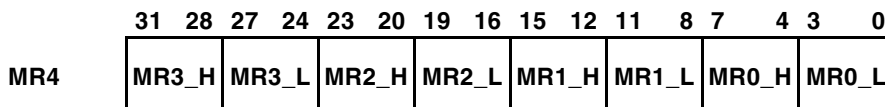


Figure 4-7. MR4 for Normal Word Result (40-Bit Accumulation)

Multiplier Operations



_H indicates High short word field

_L indicates Low short word field

Figure 4-8. MR4 for Short Word Result (20-Bit Accumulation)

These bits are also used as the input to the accumulate step of the multiply-accumulate operation. The bits are cleared together with the clear of the corresponding MR register, and when stored, are used for saturation. The purpose of these bits is to enable the partial result of a multiply-accumulate sequence to go beyond the range assigned by the final result.

Multiplier Execution Status

Multiplier operations update status flags in the compute block's Arithmetic Status (XSTAT and YSTAT) register (see [Figure 2-2 on page 2-4](#) and [Figure 2-3 on page 2-5](#)). Programs can use status flags to control execution of conditional instructions and initiate software exception interrupts. For more information, see [“Multiplier Execution Conditions” on page 4-20](#).

[Table 4-1](#) shows the flags in XSTAT or YSTAT that indicate multiplier status (a 1 indicates the condition) for the most recent multiplier operation.

Table 4-1. Multiplier Status Flags

Flag	Definition	Updated By...
MZ	Multiplier fixed-point zero and floating-point underflow	All fixed- and floating-point multiplier ops
MN	Multiplier result is negative	All fixed- and floating-point multiplier ops

Table 4-1. Multiplier Status Flags (Cont'd)

Flag	Definition	Updated By...
MV	Multiplier overflow	All fixed- and floating-point multiplier ops
MU	Multiplier underflow	All floating-point multiplier ops; cleared by fixed-point ops
MI	Multiplier floating-point invalid operation	All floating-point multiplier ops; cleared by fixed-point ops

Multiplier operations also update sticky status flags in the compute block's Arithmetic Status (XSTAT and YSTAT) register. Table 4-2 shows the flags in XSTAT or YSTAT that indicate multiplier sticky status (a 1 indicates the condition) for the most recent multiplier operation. Once set, a sticky flag remains high until explicitly cleared.

Table 4-2. Multiplier Sticky Status Flags

Flag	Definition	Updated By...
MUS	Multiplier underflow, sticky	All floating-point multiply ops
MVS	Multiplier floating-point overflow, sticky	All floating-point multiply ops
MOS	Multiplier fixed-point overflow, sticky	All fixed-point multiply ops
MIS	Multiplier floating-point invalid operation, sticky	All floating-point multiply ops

Flag update occurs at the end of each operation and is available on the next instruction cycle. A program cannot write the arithmetic status register explicitly in the same cycle that the multiplier is performing an operation.

Multi-operand instructions (for example, $Rsd = Rmd + Rnd$) produce multiple sets of results. In this case, the DSP determines a flag by ORing the result flag values from individual results.

Multiplier Execution Conditions

In a conditional multiplier instruction, the execution of the entire instruction line can depend on the specified condition at the beginning of the instruction line. Conditional multiplier instructions take the form:

```
IF cond; D0, instr.; D0, instr.; D0, instruct. ;;
```

This syntax permits up to three instructions to be controlled by a condition. Omitting the `D0` before the instruction makes the instruction unconditional.

[Table 4-3](#) lists the multiplier conditions. For more information on conditional instructions, see [“Conditional Execution” on page 7-12](#).

Table 4-3. Multiplier Conditions

Condition	Description	Flags Set
MEQ	Multiplier equal to zero	MZ = 1
MLT	Multiplier less than zero	MN and MZ = 1
MLE	Multiplier less than or equal to zero	MN or MZ = 1
NMEQ	NOT (Multiplier equal to zero)	MZ = 0
NMLT	NOT (Multiplier less than zero)	MN and MZ = 0
NMLE	NOT (Multiplier less than or equal to zero)	MN or MZ = 0

Multiplier Static Flags

In the program sequencer, the static flag (SFREG) can store status flag values for later usage in conditional instructions. With SFREG, each compute block has two dedicated static flags X/YSCF0 (condition is SF0) and X/YSCF1 (condition is SF1). The following example shows how to load a compute block condition value into a static flag register.

```
XSCF0 = XMEQ ;; /* Load X-compute block MEQ flag into XSCF0 bit
in static flags (SFREG) register */
IF SF0, XR5 = R4 * R3 ;; /* the SF0 condition tests the XSCF0
static flag */
```

For more information on static flags, see [“Conditional Execution” on page 7-12](#).

Multiplier Examples

[Listing 4-1](#) provides a number of example multiply and multiply-accumulate instructions. The comments with the instructions identify the key features of the instruction, such as fixed- or floating-point format, input operand size, and register usage.

Listing 4-1. Multiplier Instruction Examples

```
XYR4 = R6 * R8 ;;
/* This instruction is a 32-bit multiply that produces a 32-bit
result. */

XYR5:4 = R6 * R8 ;;
/* This instruction is a 32-bit multiply that produces a 64-bit
result. */

XR11:10 = R9:8 * R7:6 ;;
```

Multiplier Examples

```
/* This instruction is a quad 16-bit multiply; the input operands
are XR9_H x XR7_H, XR9_L x XR7_L, XR8_H x XR6_H, and
XR8_L x XR6_L (where _H is high half and _L is low half); the
16-bit results go to XR11_H, XR11_L, XR10_H, and XR10_L (respec-
tively). */
```

```
XMR3:2 += R1 * R0 ;;
```

```
/* This is a multiplication of source operands XR1 and XR0, and
the multiplication result is added to the current contents of the
target XMR registers, overflowing into XMR4. */
```

```
YMR1:0 -= R3 * R2 ;;
```

```
/* This is a multiplication of source operands YR3 and YR2, and
the multiplication result is subtracted from the current contents
of the target YMR registers, overflowing into YMR4. */
```

```
XR7 = MR3:2, MR3:2 += R1 * R0 ;;
```

```
/* This instruction executes a multiply-accumulate and transfers
the MR registers into the register file; the previous value in
the MR registers is transferred to the register file. */
```

```
YMR3:0 += R5:4 * R7:6 ;;
```

```
/* This instruction is four multiplications of four 16-bit shorts
in register pair YR5:4 and four 16-bit shorts in pair YR7:6. The
four results are accumulated in MR3:0 as a word result. The over-
flow bits are written into MR4. */
```

```
XMR3:2 += R9:8 * R7:6 ;;
```

```
/* This instruction is a quad 16-bit multiply-accumulate with
16-bit results; the input operands are XR9_H x XR7_H,
XR9_L x XR7_L, XR8_H x XR6_H, and XR8_L x XR6_L (where _H is high
half and _L is low half); the 16-bit accumulated results go to
XMR3_H, XMR3_L, XMR2_H, and XMR2_L (respectively). */
```

```
MR3:0 += R9:8 * R7:6 ;;
/* This instruction is a quad 16-bit multiply-accumulate with
32-bit results; the input operands are XR9_H x XR7_H,
XR9_L x XR7_L, XR8_H x XR6_H, and XR8_L x XR6_L (where _H is high
half and _L is low half); the 32-bit accumulated results go to
XMR3, XMR2, XMR1, and XMR0 (respectively). */

XMR1:0 += R9 ** R7 ;;
/* This instruction is a multiplication of the complex value in
XR9 and the complex value in XR7. The result is accumulated in
XMR1:0. */

XFR20 = R22 * R23 (T) ;;
/* This is a 32-bit (single precision) floating-point multiply
instruction with 32-bit result; single registers select 32-bit
operation. */

YFR25:24 = R27:26 * R30:29 (T) ;;
/* This is a 40-bit (extended precision) floating-point multiply
instruction with 40-bit result; double registers select 40-bit
operation. */
```

Multiplier Instruction Summary

The following listings show the multiplier instructions' syntax:


- [Listing 4-2](#) “32-Bit Fixed-Point Multiplication Instructions”
- [Listing 4-3](#) “16-Bit Fixed-Point Quad Multiplication Instructions”
- [Listing 4-4](#) “16-Bit Fixed-Point Complex Multiplication Instructions”


Multiplier Instruction Summary

- [Listing 4-5](#) “32- and 40-Bit Floating-Point Multiplication Instructions”
- [Listing 4-6](#) “Multiplier Register Load Instructions”

The conventions used in these listings for representing register names, optional items, and choices are covered in detail in [“Register File Registers” on page 2-5](#). Briefly, these conventions are:

- { } — The curly braces enclose options; these braces are not part of the instruction syntax.
- | — The vertical bars separate choices; these bars are not part of the instruction syntax.
- *Rmd* — The register names in italic represent user selectable single (*Rs*, *Rm*, *Rn*), double (*Rsd*, *Rmd*, *Rnd*) or quad (*Rsq*, *Rmq*, *Rnq*) register names.

 Each instruction presented here occupies one instruction slot in an instruction line. For more information about instruction lines and instruction combination constraints, see [“Instruction Line Syntax and Structure” on page 1-20](#) and [“Instruction Parallelism Rules” on page 1-24](#).

 The MR3:0 registers are four 32-bit accumulation registers. They overflow into MR4, which stores two 16-bit overflows for 32-bit multiples, or four 8-bit overflows for quad 16-bit multiples.

Listing 4-2. 32-Bit Fixed-Point Multiplication Instructions

$$\{X|Y|XY\}Rs = Rm * Rn \{((U|nU)\{I\}\{T\}\{S\})\} ;^1$$
$$\{X|Y|XY\}Rsd = Rm * Rn \{((U|nU)\{I\})\} ;$$
$$\{X|Y|XY\}MRa += Rm * Rn \{((U)\{I\}\{C|CR\})\} ;^2$$

¹ Options include: (): fractional, signed, and no saturation; (S): saturation, signed, (SU): saturation, unsigned


```

{X|Y|XY}MRa -= Rm * Rn {{(I){C|CR}}};
{X|Y|XY}Rs = MRa, MRa += Rm * Rn {{(U){I}{C}}}; dual operation
{X|Y|XY}Rsd = MRa, MRa += Rm * Rn {{(U){I}{C}}}; dual operation
/* where MRa is either MR1:0 or MR3:2 */

```

Listing 4-3. 16-Bit Fixed-Point Quad Multiplication Instructions

```

{X|Y|XY}Rsd = Rmd * Rnd {{(U){I}{T}{S}}};
{X|Y|XY}RsQ = Rmd * Rnd {{(U){I}}};
{X|Y|XY}MRb += Rmd * Rnd {{(U){I}{C}}};
{X|Y|XY}MRb += Rmd * Rnd {{(U){I}{C}}};
/* where MRb is either MR1:0 or MR3:2 */
{X|Y|XY}MR3:0 += Rmd * Rnd {{(U){I}{C|CR}}}; {X|Y|XY}Rsd = MRb,
MRb += Rmd * Rnd {{I}{C}}}; dual operation
/* where MRb is either MR1:0, MR3:2, or MR3:0 */

```

Listing 4-4. 16-Bit Fixed-Point Complex Multiplication Instructions

```

{X|Y|XY}MRa += Rm ** Rn {{(I){C|CR}{J}}};
{X|Y|XY}Rs = MRa, MRa += Rm ** Rn {{(I){C}{J}}}; dual operation
{X|Y|XY}Rsd = MRa, MRa += Rm ** Rn {{(I){C}{J}}}; dual operation
/* where MRa is either MR1:0 or MR3:2 */

```

Listing 4-5. 32- and 40-Bit Floating-Point Multiplication Instructions

```

{X|Y|XY}FRs = Rm * Rn {(T)};
{X|Y|XY}FRsd = Rmd * Rnd {(T)};

```

² Options include: (): signed, round-to-nearest even, (T): signed, truncate, (U): unsigned, round-to-nearest even, (TU): unsigned, truncate

Multiplier Instruction Summary

Listing 4-6. Multiplier Register Load Instructions

```
{X|Y|XY}MRa = Rmd ;  
{X|Y|XY}MR4 = Rm ;  
{X|Y|XY}{S}Rsd = MRa {{{U}{S}}} ;  
{X|Y|XY}Rsq = MR3:0 {{{U}{S}}} ;  
{X|Y|XY}Rs = MR4 ;  
{X|Y|XY}Rs = COMPACT MRa {{{U}{I}{S}}} ;  
{X|Y|XY}SRsd = COMPACT MR3:0 {{{U}{I}{S}{C}}} ;  
/* where MRa is either MR1:0 or MR3:2 */
```

5 SHIFTER

The TigerSHARC processor core contains two computation units known as compute blocks. Each compute block contains a register file and three independent computation units—an ALU, a multiplier, and a shifter. The shifter is highlighted in [Figure 5-1](#). The shifter takes its inputs from the register file, and returns its outputs to the register file.

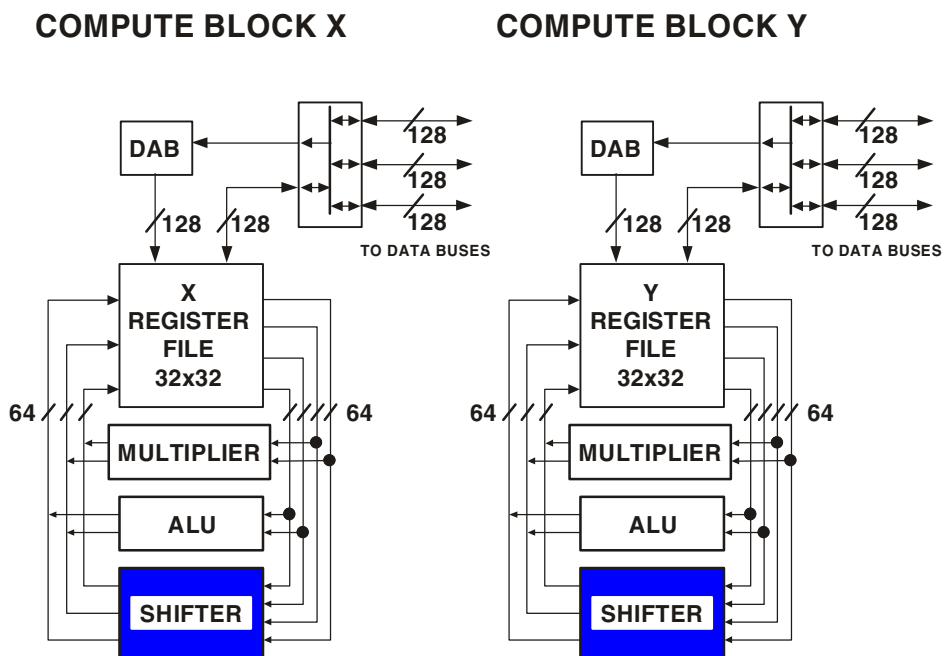


Figure 5-1. Shifters in Compute Block X and Y

This unit performs *bit wise operations* (arithmetic and logical shifts) and performs *bit field operations* (field extraction and deposition) for the processor. The shifter also executes *data conversion operations* such as fixed- and floating-point format conversions. Shifter operations include:


- Shift and rotate bit field, from off-scale left to off-scale right
- Bit manipulation; bit set, clear, toggle, and test
- Bit field manipulation; field extract and deposit
- Scaling factor identification, 16-bit block floating-point
- Extract exponent
- Count number of leading ones or zeros

The shifter operates on fixed-point data and can take the following as input:

- One long word (64-bit) operand
- One or two normal word (32-bit) operands
- Two or four short word (16-bit) operands
- Four or eight byte word (8-bit) operands

As shown in [Figure 5-1](#), the shifter has four inputs and four outputs (unlike the ALU and multiplier, which have two inputs and outputs). The shifter’s I/O paths within the compute block have some implications for instruction parallelism.

- Shifter instructions that use three inputs cannot be executed in parallel with any other compute block operations.
- For `FDEP`, `MASK`, and `PUTBIT` instructions, there are three registers that are passed into the shifter. This operation uses three compute block ports. The output is being placed in the same port.

 For more information on available ports and instruction parallelism, see [“Instruction Parallelism Rules”](#) on page 1-24.

Within instructions, the register name syntax identifies the input operand and output result data size and type. For more information on data size and type selection for shifter instructions, see [“Register File Registers”](#) on page 2-5.

The remainder of this chapter presents descriptions of shifter instructions and results using instruction syntax. For an explanation of the instruction syntax conventions used in shifter and other instructions, see [“Instruction Line Syntax and Structure”](#) on page 1-20. For a list of shifter instructions and their syntax, see [“Shifter Instruction Summary”](#) on page 5-19.

Shifter Operations

The shifter operates on one 64-bit, one or two 32-bit, two or four 16-bit, and four or eight 8-bit fixed-point operands. Shifter operations include:

- Shifts and rotates from off-scale left to off-scale right
- Bit manipulation operations, including bit set, clear, toggle and test

Shifter Operations

- Bit field manipulation operations, including field extract and deposit, using register `BFOTMP` (which is internal to the shifter)
- Bit FIFO operations to support bit streams with fields of varying length
- Support for ADSP-2100 family compatible fixed-point and floating-point conversion operations (such as exponent extract, number of leading 1s or 0s)

The shifter operates on the compute block register files and operates on the shifter register `BFOTMP`—an internal shifter register which is used for the `PUTBITS` instruction. Shifter operations can take their Rm input (data operated on) from the register file and take their Rn input (shift magnitudes) either from the register file or from immediate data provided in the instruction. In cases where the operation involves a third input operand, Rm and Rn inputs are taken from the register file, and the third input, Rs , is a read-modify-write (RMW).

Shift magnitudes for register file-based operations—where the shift magnitude comes from Rn —are held in the right-most bits of Rn . The shift magnitude size (number of bits) varies with the size of the output operand, where Rn is 8 bits for long word output, 7 bits for normal word output, 6 bits for short word output and 6 bits for byte word output. In this way, full-scale right and left shifts can be achieved. Bits of Rn outside of the shift magnitude field are masked.

The following sections describe shifter operation details:

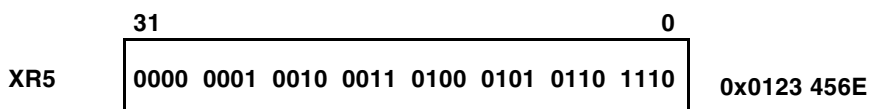
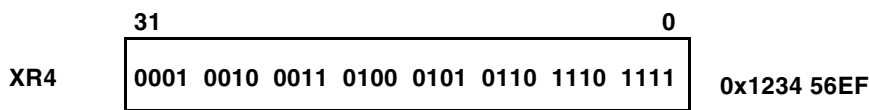
- [“Logical Shift Operation” on page 5-5](#)
- [“Arithmetic Shift Operation” on page 5-6](#)
- [“Bit Manipulation Operations” on page 5-7](#)
- [“Bit Field Manipulation Operations” on page 5-8](#)

- “Bit Field Conversion Operations” on page 5-11
- “Bit Stream Manipulation Operations” on page 5-11

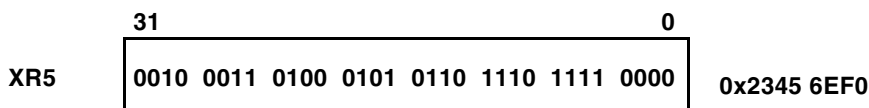
Logical Shift Operation

The following instruction is an example of a logical shift (LSHIFT). The operation shifts the contents of the XR4 register by the shift value (number of bits) specified in the XR3 register. The shifter places the result in the XR5 register. Figure 5-2 shows how the bits in register XR5 are placed for shift values of 4 and -4.

```
XR5 = LSHIFT R4 BY R3;;
```



For a negative LSHIFT value, the shift is to the RIGHT and ZERO-FILLED. Here, the LSHIFT value is -4, so bits 31-28 are zero-filled.



For a positive LSHIFT value, the shift is to the LEFT and ZERO-FILLED. Here, the LSHIFT value is 4, so bits 3-0 are zero-filled.

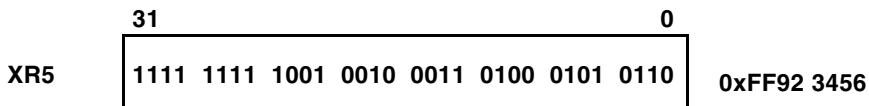
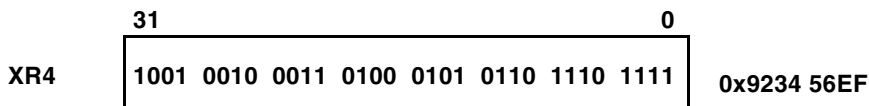
Figure 5-2. LSHIFT Instruction Example

Shifter Operations

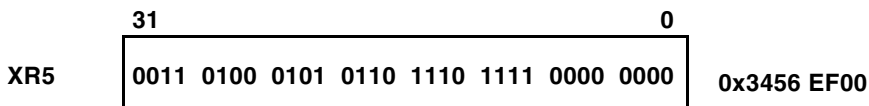
Arithmetic Shift Operation

The following instruction is an example of an arithmetic shift (ASHIFT). The operation shifts the contents of the XR4 register by the shift value (number of bits) specified in the XR3 register. The shifter places the result in the XR5 register. Figure 5-3 shows how the bits in register XR5 are placed for shift values of 8 and -8.

```
XR5 = ASHIFT R4 BY R3 ;;
```



For a negative ASHIFT value, the shift is to the RIGHT and SIGN-EXTENDED. Here, the ASHIFT value is -8, so bits 31-24 are sign-extended.



For a positive ASHIFT value, the shift is to the LEFT and ZERO-FILLED. Here, the ASHIFT value is 8, so bits 7-0 are zero-filled.

Figure 5-3. ASHIFT Instruction Example

Bit Manipulation Operations

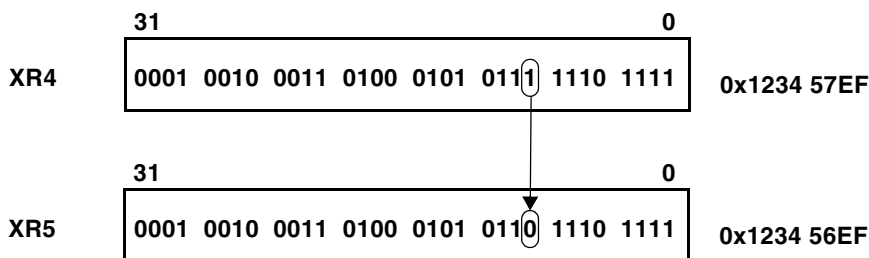
The shifter supports bit manipulation operations including bit clear (BCLR), bit set (BSET), bit toggle (BTGL), and bit test (BITEST). The operand size can be a normal word or a long word. For example:

```
R5 = BCLR R3 By R2 ;; /* 32-bit operand */
R5:4 = BSET R3:2 By R6 ;; /* 64-bit operand */
```

The following instruction is an example of bit manipulation (BCLR). The shifter clears the bit in the XR4 register indicated by the bit number specified in the XR3 register. The shifter places the result in the XR5 register.

Figure 5-4 shows how the bits in register XR5 are affected for bit number 8.

```
XR5 = BCLR R4 By R3 ;;
```



**For a BCLR bit manipulation, the selected bit is CLEARED.
Because XR3=0x8 (the bit number), bit 8 is cleared.**

Figure 5-4. BCLR Instruction Example

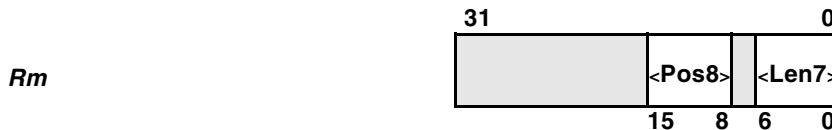
Shifter Operations

Bit Field Manipulation Operations

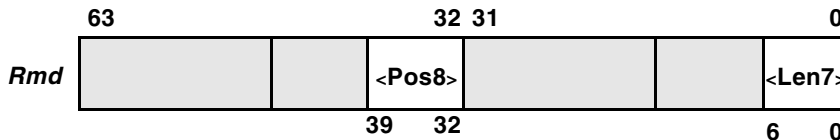
The shifter supports bit field manipulation operations including:

- **FEXT**—Extracts a field from a register according to the length and position specified by another register
- **FDEP**—Deposits a right-justified field into a register according to the length and position specified by another register
- **MASK**—Copies a 32- or 64-bit field created by a mask
- **XSTAT/YSTAT**—Loads or stores all bits or 14 LSBs only of the XSTAT or YSTAT register

For field extract and deposit operations, the Rn operand contains the control information in two fields: $\langle \text{Len7} \rangle$ and $\langle \text{Pos8} \rangle$. These fields select the number of bits to extract (Len7) and the starting position in Rm (Pos8). The location of these fields depends on whether Rn is a single- or dual-register as shown in [Figure 5-5](#).



For single register operands, the Pos8 and Len7 fields are in bits 15–8 and 6–0.



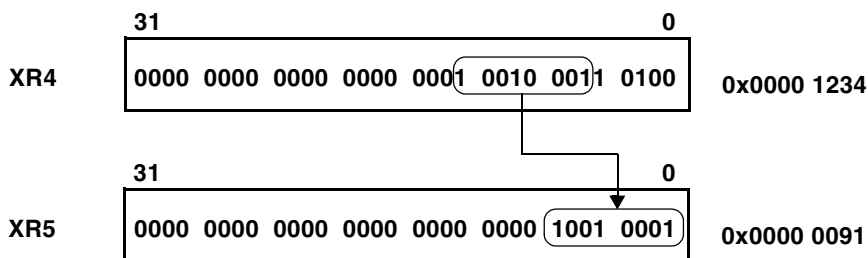
For dual register operands, the Pos8 and Len7 fields are in bits 39–32 and 6–0.

Figure 5-5. FEXT and FDEP Instructions Pos8 and Len7 Fields

There are two versions of the `FEXT` and `FDEP` instructions. One version takes the control information from a register pair. The other version takes control information from a single register. The `FEXT` instruction takes the data from the indicated position in the source register and places right-justified data in the destination register (R_s). The `FDEP` instruction takes the right-justified data from the source register and places data in the indicated position in the destination register (R_s).

The following instruction is an example of bit field extraction (`FEXT`). The shifter extracts the bit field in the `XR4` register indicated by the field position (`Pos8`) and field length (`Len7`) values specified in the `XR3` register. The shifter places the right-justified result in the `XR5` register. The default operation zero-fills the unused bits in the destination register (`XR5` in the example). If the `FEXT` instruction included the sign extend (`SE`) option, the most significant bit of the extracted field is extended. [Figure 5-6](#) shows how the bits in register `XR5` are affected for field position `Pos8=5` and field length `Len7=8`.

```
XR5 = FEXT R4 By R3 ;; /* Pos8=5, Len7=8, XR3=0x0000 0508 */
```



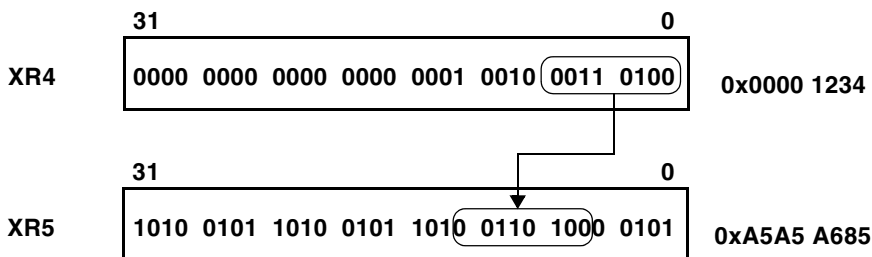
For a `FEXT` field extraction, the unused bits in the destination are CLEARED unless the `SE` option is used. Here, bits 31–8 are cleared.

Figure 5-6. `FEXT` Instruction Example

Shifter Operations

The following instruction is an example of bit field deposit (FDEP). The shifter extracts the right-justified bit field in the XR4 register field length (Len7) value specified in the XR3 register. The shifter places the result in the XR5 register in the location indicated by the field position (Pos8). The default operation does not alter the unused bits in the destination register (XR5 in the example). If the FDEP instruction included the sign extend (SE) option, the most significant bit of the extracted field is extended. If the FDEP instruction included the sign extend (ZF) option, the most significant unused bits of result register are zero filled. Figure 5-7 shows how the bits in register XR5 are affected for field position Pos8=5 and field length Len7=8.

```
XR5 = FDEP R4 By R3 ;; /* Pos8=5, Len7=8, XR3=0x0000 0508, XR5  
value before instruction was 0xA5A5 A5A5 */
```



For a FDEP field deposit, the unused bits in the destination are UNCHANGED unless the SE or ZF option is used. Here, bits 31–13 and 4–0 are unchanged.

Figure 5-7. FDEP Instruction Example

The following instruction is an example of mask (MASK) operation. The shifter takes the bits from XR4 corresponding to the mask XR3 and ORs them into the XR5 register. The bits of XR5 outside the mask remain untouched.

```
XR3 = 0x00007B00;;  
XR4 = 0x50325032;;
```

```
XR5 = 0x85FFFFFF;; /* before mask instruction */
XR5 += MASK R4 BY R3
/* After mask instruction, XR5 = 0x85FFD4FF */
```

Bit Field Conversion Operations

The shifter supports fixed- to floating-point conversion operations including:

- **BKFPT**—Determines scaling factor used in 16-bit block floating-point
- **EXP**—Extracts the exponent
- **LDX**—Extracts leading zeros (0) or ones (1)

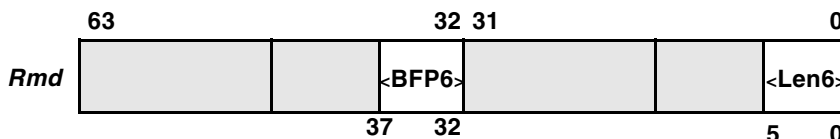
Bit Stream Manipulation Operations

The bit stream manipulation operations, in conjunction with the ALU **BFOINC** instruction, implement a bit FIFO used for modifying the bits in a contiguous bit stream. The shifter supports bit stream manipulation operations including:

- **GETBITS**—Extracts bits from a bit stream
- **PUTBITS**—Deposits bits in a bit stream
- **BFOTMP**—Temporarily stores or returns overflow from **GETBITS** and **PUTBITS** instructions

Shifter Operations

For bit stream extract (GETBITS) and deposit (PUTBITS) operations, the *Rnd* operand contains the control information in two fields: <BFP6> and <Len6>. These fields in the dual register, *Rnd*, appear in Figure 5-8.



The dual register operand provides the BFP6 and Len6 fields (bits 37–32 and 5–0). Note that the BFP must be incremented using the ALU's BFOINC instruction.

Figure 5-8. GETBITS and PUTBITS Instructions BFP6 and Len6 Fields

The GETBITS instruction extracts the number of bits indicated by Len6 starting at BFP6 and places the right-justified data in the output register. The unused bits are cleared unless the sign extend (SE) option is used. With the SE option, the most significant bit of the extract is extended to the most significant bit of the output register.

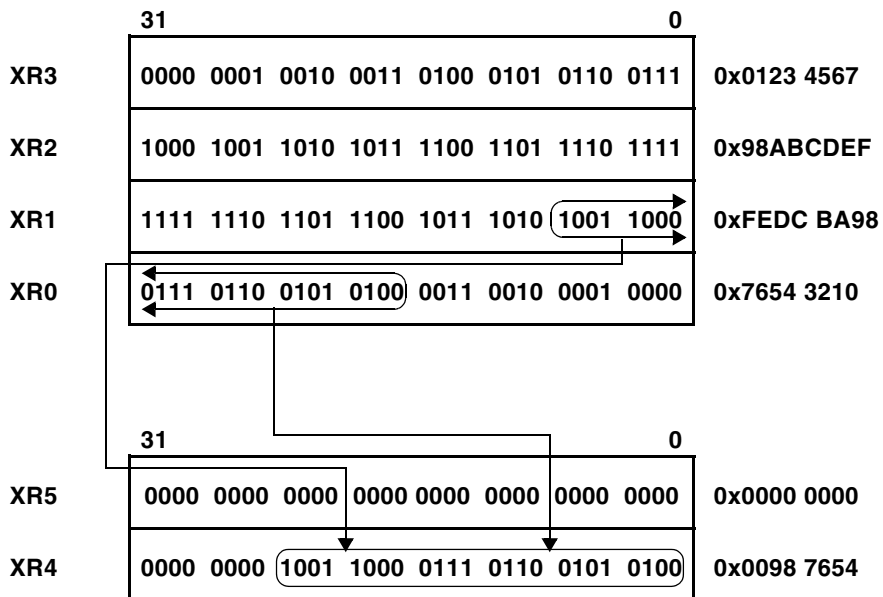
The following instruction is an example of bit stream extraction (GETBITS). The shifter extracts a portion of the bit stream in the XR3:0 quad register indicated by the bit FIFO position (BFP6) and field length (Len6) values specified in the XR7:6 dual register.

i Use the ALU's BFOINC instruction to increment the bit FIFO pointer. Normally, an update of bit FIFO pointer is necessary after executing GETBITS. The ALU instruction BFOINC adds BFP6 and Len6 fields, divides them by 64 and returns the remainder to BFP6 field. If for example, BFP6 is 0x30 and Len6 is 0x18, the new value of BFP6 is 0x08 and the flag AN in XSTAT register is set. This flag may be used to identify this situation and proceed accordingly.

In the example, the shifter places the right-justified result in the XR5:4 dual register. The default operation zero-fills the unused bits in the destination register (XR5:4 in the example). If the GETBITS instruction included

the sign extend (SE) option, the most significant bit of the extracted field is extended. Figure 5-9 shows how the bits in register XR5:4 are affected for field position BFP6=16 and field length Len6=24.

```
XR5:4 = GETBITS R3:0 BY R7:6 ;;
/* BFP6=16, Len6=24, XR7:6=0x0000 0010 0000 0018 */
```



For a GETBITS field extraction, the unused bits in the destination are CLEARED unless the SE option is used. Here, bits XR5 and bits 31–24 of XR4 are cleared.

Figure 5-9. GETBITS Instruction Example

The PUTBITS instruction deposits the 64 bits from *Rmd* registers into a contiguous bit stream held in the quad register composed of *BFOTMP* in the top and *Rsd* in the bottom. In PUTBITS, the BFP field specifies the starting bit where the insertion begins in *Rsd* register, but the Len6 field is ignored. Update of BFP may only be performed by the ALU with the instruction BFOINC.

Shifter Operations

The following instruction is an example of bit stream placement (PUTBITS). The shifter puts the content of the registers XR3:2 into the bit FIFO composed by XR5:4 and BFOTMP beginning with bit 16 of XR4 (specified into BFP field of XR7).

```
XR3 = 0x01234567 ;;
XR2 = 0x89abcdef ;;
XR5 = 0x0 ;
XR4 = 0x0 ;
XR5:4 += PUTBITS R3:2 BY R7:6 ;; /* BFP6=16, XR7:6=0x0000 0010
0000 0018 */
/* After PUTBITS instruction, the registers hold:
  xBFOTMP = 0x0000 0000 0000 0123
  XR5      = 0x4567 89ab
  XR4      = 0xcdef 0000
```

Shifter Instruction Options

Some of the shifter instructions have options associated with them that permit flexibility in how the instructions execute. It is important to note that these options modify the detailed execution of instructions, and the options that are particular to a group of instructions—not all options are applicable to all instructions. Instruction options appear in parenthesis at the end of the instruction's slot. For a list indicating which options apply for particular shifter instructions, see [“Shifter Instruction Summary” on page 5-19](#). The shifter instruction options include:

- () zero filled, right justified
- (SE) sign extended; applies to FEXT, FDEP, and GETBITS instructions
- (ZF) zero filled; applies to FDEP instruction

The following are shifter instructions that demonstrate bit field manipulation operations with options applied.

```
XR5 = FEXT R4 By R3 (SE) ;;
/* The SE option in this instruction sets bits 31–8 to 1 in Figure 5-6 on
page 5-9 */
```

```
XR5 = FDEP R4 By R3 (ZF) ;;
/* The ZF option in this instruction clears bits 31–13 to 0 in Figure 5-7 on
page 5-10 */
```

Sign Extended Option

The sign extend (SE) option is available for the FEXT, FDEP, and GETBITS shifter instructions. If used, this option extends the value of the most significant bit of the placed bit field through the most significant bit of the output register.

Zero Filled Option

The zero filled (ZF) option is available only for the FDEP instruction. If used, this option clears all the unused bits above the most significant bit of the placed bit field in the output register.

Shifter Execution Status

Shifter operations update status flags in the compute block's Arithmetic Status (XSTAT and YSTAT) register (see [Figure 2-2 on page 2-4](#) and [Figure 2-3 on page 2-5](#)). Programs can use status flags to control execution of conditional instructions and initiate software exception interrupts. For more information, see “[Shifter Execution Conditions](#)” on page 5-16.

Shifter Operations

Table 5-1 shows the flags in XSTAT or YSTAT that indicate shifter status (a 1 indicates the condition) for the most recent shifter operation.

Table 5-1. Shifter Status Flags

Flag	Definition	Updated By...
SZ	Shifter fixed-point zero	All shifter ops
SN	Shifter negative	All shifter ops
BF1-0	Shifter block floating-point	BKFPT instruction only
AN	ALU negative	On overflow of quad result in PUTBITS instruction

Flag update occurs at the end of each operation and is available on the next instruction cycle. A program cannot write the arithmetic status register explicitly in the same cycle that the multiplier is performing an operation.

Multi-operand instructions (for example, $BRs = ASHIFT Rn BY Rm;$) produce multiple sets of results. In this case, the DSP determines a flag by ORing the result flag values from individual results.

Shifter Execution Conditions

In a conditional shifter instruction, the execution of the entire instruction line can depend on the specified condition at the beginning of the instruction line. Conditional shifter instructions take the form:

```
IF cond; D0, instr.; D0, instr.; D0, instruct. ; ;
```

This syntax permits up to three instructions to be controlled by a condition. Omitting the D0 before the instruction makes the instruction unconditional.

Table 5-2 lists the shifter conditions. For more information on conditional instructions, see [“Conditional Execution” on page 7-12](#).

Table 5-2. Shifter Conditions

Condition	Description	Flags set
SEQ	Equal to zero	SZ = 1
SLT	Less than zero	SN and SZ = 1
NSEQ	Not equal to zero	SZ = 0
NSLT	Not less than zero	SN and SZ = 0

Shifter Static Flags

In the program sequencer, the static flag (SFREG) can store status flag values for later usage in conditional instructions. With SFREG, each compute block has two dedicated static flags X/YSCF0 (condition is SF0) and X/YSCF1 (condition is SF1). The following example shows how to load a compute block condition value into a static flag register.

```
XSCF0 = XSEQ ;; /* Load X-compute block SEQ flag into XSCF0 bit
in static flags (SFREG) register */
IF SF0, XR5 = LSHIFT R4 BY R3 ;; /* the SF0 condition tests the
XSCF0 static flag */
```

For more information on static flags, see [“Conditional Execution” on page 7-12](#).

Shifter Examples

[Listing 5-1](#) provides a number of shifter instruction examples. The comments with the instructions identify the key features of the instruction, such as input operand size and register usage.

Shifter Examples

Listing 5-1. Shifter Instruction Examples

```
XR5 = LSHIFT R4 BY R3;;
/* This is a logical shift of register XR4 by the value contained
in XR3. */

YR1 = ASHIFT R2 BY R0;;
/* This is an arithmetic shift of register XR2 by the value con-
tained in XR0. */

R1:0 = ROT R3:2 BY 8;;
/* This instruction rotates the content of R3:2 in both the X-
and Y-ALUs by 8 and places the result in XR1:0 and YR1:0. */

XBITEST R1:0 BY R7;;
/* This instruction tests the bit indicated in XR7 of XR1:0 and
sets accordingly the flags XSZ and XSN in XSTAT. */

R9:8 = BTGL R11:10 BY R13;;
/* This instruction toggles the bit indicated in R13 of XR11:10
and YR11:10 and puts the result in xR9:8 and yR9:8. */

XR15 = LD0 R17;;
/* This instruction extracts the leading number of zeros of xR17
and places the result into xR15. */
```

Shifter Instruction Summary

[Listing 5-2](#) shows the shifter instructions' syntax. The conventions used in these listings for representing register names, optional items, and choices are covered in detail in [“Register File Registers” on page 2-5](#). Briefly, these conventions are:

- { } – the curly braces enclose options; these braces are not part of the instruction syntax.
- | – the vertical bars separate choices; these bars are not part of the instruction syntax.
- *Rmd* – the register names in italic represent user selectable single (*Rs*, *Rm*, *Rn*), double (*Rsd*, *Rmd*, *Rnd*) or quad (*Rsq*, *Rmq*, *Rnq*) register names.

In shifter instructions, output register name relates to output operand size as follows:

- The L prefix on an output operand (register name) indicates long word (64-bit) output. For example, the following instruction syntax indicates single, long word output:
`LRsd = ASHIFT Rmd BY Rnd;`
- The absence of a prefix on an output operand (register name) indicates normal word (32-bit) output. For example, the following instruction syntax indicates single, normal word output:
`Rs = ASHIFT Rm BY Rn;`
- A dual register name on an output operand (register name) indicates two normal word (32-bit) outputs. For example, the following instruction syntax indicates two, normal word outputs:
`Rsd = ASHIFT Rmd BY Rnd;`

Shifter Instruction Summary

- The S prefix on an output operand (register name) indicates two or four short word (16-bit) outputs. For example, the following instruction syntax indicates two or four, short word outputs:

```
SRs = ASHIFT Rm BY Rn /* two outputs */;
```

```
SRsd = ASHIFT Rmd BY Rnd; /* four outputs */
```

- The B prefix on an output operand (register name) indicates four or eight byte word (8-bit) outputs. For example, the following instruction syntax indicates four or eight, byte word outputs:

```
BRs = ASHIFT Rm BY Rn /* four outputs */;
```

```
BRsd = ASHIFT Rmd BY Rnd; /* eight outputs */
```



Each instruction presented here occupies one instruction slot in an instruction line. For more information about instruction lines and instruction combination constraints, see [“Instruction Line Syntax and Structure”](#) on page 1-20 and [“Instruction Parallelism Rules”](#) on page 1-24.

Listing 5-2. Shifter Instructions

```

{X|Y|XY}{B|S}Rs = LSHIFT|ASHIFT Rm BY Rn|<Imm> ;1,2
{X|Y|XY}{B|S|L}Rsd = LSHIFT|ASHIFT Rmd BY Rn|<Imm> ;1,2

{X|Y|XY}Rs = ROT Rm BY Rn|<Imm6> ;1
{X|Y|XY}{L}Rsd = ROT Rmd BY Rnd|<Imm> ;1,2

{X|Y|XY}Rs = FEXT Rm BY Rn|Rnd {(SE)} ;3
{X|Y|XY}LRsd = FEXT Rmd BY Rn|Rnd {(SE)} ;3

{X|Y|XY}Rs += FDEP Rm BY Rn|Rnd {(SE|ZF)} ;3
{X|Y|XY}LRsd += FDEP Rmd BY Rn|Rnd {(SE|ZF)} ;3

{X|Y|XY}Rs += MASK Rm BY Rn ;
{X|Y|XY}LRsd += MASK Rmd BY Rnd ;

{X|Y|XY}Rsd = GETBITS Rmq BY Rnd {(SE)} ;

{X|Y|XY}Rsd += PUTBITS Rmd BY Rnd ;

{X|Y|XY}BITEST Rm BY Rn|<Imm5> ;
{X|Y|XY}BITEST Rmd BY Rn|<Imm6> ;

{X|Y|XY}Rs = BCLR|BSET|BTGL Rm BY Rn|<Imm5> ;
{X|Y|XY}Rsd = BCLR|BSET|BTGL Rmd BY Rn|<Imm6> ;

{X|Y|XY}Rs = LDO|LD1 Rm|Rmd ;

{X|Y|XY}Rs = EXP Rm|Rmd ;

{X|Y}STAT = Rm ;

```

¹ The *Rn* data size (bits) for the shift magnitude varies with the output operand: Byte: 5, Short: 6, Normal: 7, Long: 8.

² The size in bits of the *Imm* data varies with the output operand: Byte: 4, Short: 5, Normal: 6, Long: 7.

³ The placement of the Pos8 and Len7 fields varies with the *Rn/Rnd* register, see [Figure 5-5 on page 5-8](#).

Shifter Instruction Summary

$\{X|Y\}STATL = Rm ;$

$\{X|Y\}Rs = \{X|Y\}STAT ;$

$\{X|Y|XY\}BKFPT Rmd, Rnd ;$

$\{X|Y|XY\}Rsd = BFOTMP ;$

$\{X|Y|XY\}BFOTMP = Rmd ;$

6 IALU

The TigerSHARC processor core contains two integer arithmetic logic units known as IALUs. Each IALU contains a register file and dedicated registers for circular buffer addressing. The IALUs can control the Data Alignment Buffers (DABs) for unaligned memory access operations. The IALUs and DABs are highlighted in Figure 6-1.

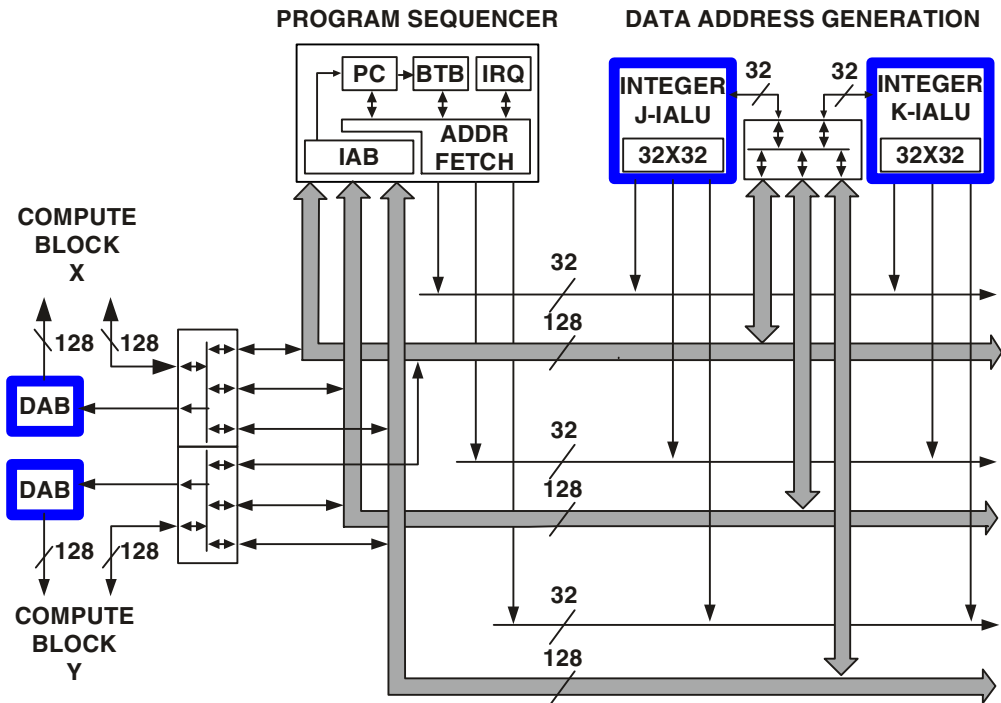


Figure 6-1. IALUs, DABs, and Data Buses

The TigerSHARC processor's two independent IALUs are referred to as the J-IALU and K-IALU. The IALUs support regular ALU operations and data addressing operations. The *arithmetic, logical, and function ALU operations* include:

- Add and subtract, with and without carry/borrow
- Arithmetic right shift, logical right shift, and rotation
- Logical operations: AND, AND NOT, NOT, OR, and XOR
- Functions: absolute value, min, max, compare

The IALUs provide memory addresses when data is transferred between memory and registers. Dual IALUs enable simultaneous addresses for multiple operand reads or writes. The IALU's *data addressing and data movement operations* include:

- Direct and indirect memory addressing
- Circular buffer addressing
- Bit reverse addressing
- Universal register (*Ureg*) moves and loads
- Memory pointer generation

Each move instruction specifies whether a normal, long, or quad word is accessed from each memory block. Because the DSP has two IALUs, two memory blocks can be accessed on each cycle. Long word accesses can be used to supply two aligned normal words to one compute block or one aligned normal word to each compute block. Quad word accesses may be used to supply four aligned normal words to one compute block or two aligned normal words to each compute block. This is useful in applications that use real/imaginary data, or parallel data sets that can be aligned in memory—as are typically found in DSP applications. It is also used for fast save/restore of context during C calls or interrupts.

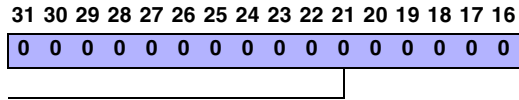
The IALU provides flexibility in moving data as single, dual, or quad words. Every instruction can execute with a throughput of one per cycle. IALU arithmetic instructions execute with a single cycle of latency while computation units have two cycles of latency. Normally, there are no dependency delays between IALU instructions, but if there are, three or four cycles of latency can occur. [For more information, see “Dependency and Resource Effects on Pipeline” on page 7-55.](#)

The IALUs each contain a 32-register data register file and eight dedicated registers for circular buffer addressing. All registers in the IALU are 32-bit wide, memory-mapped, universal registers. Some important points concerning the IALU register files are:

- The J-IALU register file registers are J_{31} – J_0 , and the K-IALU register file registers are K_{31} – K_0 . Except for J_{31} and K_{31} , these registers are general purpose and contain integer data only.
- The J_{31} and K_{31} registers are 32-bit status registers and can also be referred to as $JSTAT$ and $KSTAT$.

The $JSTAT$ (J_{31}) and $KSTAT$ (K_{31}) registers appear in [Figure 6-2](#) and [Figure 6-3](#). These registers have special operations:

- When used as an operand in an IALU arithmetic, logical, or function operation, these registers are referred to as J_{31} and K_{31} registers, and the register’s contents are treated as 0. If J_{31} is used as an output of an operation, it does not retain the result of the instruction, but the flags are set.
- When used for an IALU load, store, or move operation, these registers are referred to as $JSTAT$ and $KSTAT$, and the operation does not clear the register contents.



Reserved

Figure 6-2. JSTAT/KSTAT (Upper Half) Register Bit Descriptions

For fast save and restore operations, load and store instructions can access J31:28 in quad-register format, saving or restoring J30:28 and JSTAT.

The dedicated registers for circular buffer addressing in each IALU select the base address and buffer length for circular buffers. These dedicated registers work with the first four general-purpose registers in each IALU's register file to manage up to eight circular buffers. Some important points concerning the IALU dedicated registers for circular buffer addressing are:

- The circular buffer *index* (current address) is set by a general-purpose register. These are J3–J0 in the J-IALU, and K3–K0 in the K-IALU.
- The circular buffer *base* (starting address) is set by a dedicated register. These are JB3–JB0 in the J-IALU, and KB3–KB0 in the K-IALU.
- The circular buffer *length* (number of memory locations) is set by a dedicated register. These are JL3–JL0 in the J-IALU, and KL3–KL0 in the K-IALU.
- The circular buffer *modifier* (step size between memory locations) is set by either a general-purpose IALU register or an immediate value.
- The index, base, and length registers for controlling circular buffers work as a unit (J0 with JB0 and JL0, J1 with JB1 and JL1, and so on). Any IALU register file register (beside the one serving as index) in the IALU controlling the circular buffer may serve as the modifier.

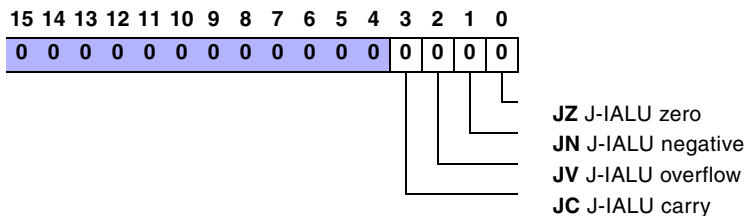


Figure 6-3. JSTAT/KSTAT (Lower Half) Register Bit Descriptions

The IALUs can use add and subtract instructions to generate memory pointers with or without circular buffer or bit reverse addressing. The modified address is stored in an IALU data register and (optionally) can be written to the program sequencer's computed jump (CJMP) register.

IALU Operations

The following sections describe the operation of each type of IALU instruction. These operation descriptions apply to both the J-IALU and K-IALU. The IALU operations are:

- [“IALU Arithmetic, Logical, and Function Operations” on page 6-5](#)
- [“IALU Data Addressing and Transfer Operations” on page 6-13](#)

IALU Arithmetic, Logical, and Function Operations

The IALU performs arithmetic and logical operations on fixed-point (integer) data. The DSP uses IALU register file registers for the input operands and output result from IALU operations. The IALU register file registers are J30 through J0 and K30 through K0. The IALUs each have one special purpose register—the JSTAT/KSTAT register—for status. For more

IALU Operations

information on the register files and register naming syntax for selecting data type and width, see [“Register File Registers” on page 2-5](#). The following are IALU instructions that demonstrate arithmetic operations.

```
J2 = J1 + J0 ;;  
/* This is a fixed-point add of the 32-bit input operands J1 and  
J0; the DSP places the result in J2. */
```

```
K0 = ABS K2 ;;  
/* The DSP places the absolute value of fixed-point 32-bit input  
operand K2 in the result register K0. */
```

```
J2 = ( J1 + J0 ) / 2 ;;  
/* This is a fixed-point add and divide by 2 of the 32-bit input  
operands J1+J0; the DSP places the result in J2. */
```

All IALU arithmetic, logical, and function instructions generate status flags to indicate the status of the result. If for example in the previous add/divide instruction the input was $((-1 + 0) / 2)$, the operation would set the J_N flag (J-IALU, IALU result negative) because operation resulted in a negative value. For more information on IALU status, see [“IALU Execution Status” on page 6-10](#).

IALU Instruction Options

Most of the IALU instructions have options associated with them that permit flexibility in how the instructions execute. It is important to note that these options modify the detailed execution of instructions, and options that are particular to a group of instructions—not all options are applicable to all instructions. Instruction options appear in parenthesis at

the end of the instruction's slot. For a list indicating which options apply for particular IALU instructions, see [“IALU Instruction Summary” on page 6-39](#). The IALU instruction options include:

- () signed operation, round-to-infinity, integer mode
- (U) unsigned operation
- (CB) circular buffer operation for result
- (BR) bit reverse operation for result
- (CJMP) load result into result and computed jump (CJMP) registers

The following are IALU instructions that demonstrate arithmetic operations with options applied.

```
J2 = J1 - J0 (CJMP);;
/* This is a fixed-point subtract of the 32-bit input operands;
the DSP loads the result into J2 and CJMP register. */

K1 = K3 + K4 (BR) ::
/* This is a fixed-point add of the 32-bit input operands with
bit reverse carry operation for the result. */

COMP(J1, J0) (U) ;;
/* This is a comparison of unsigned 32-bit input operands. */
```

Integer Data

The DSP always represents fixed-point numbers in 8, 16, 32, or 64 bits, using up to four 32-bit data registers. In the IALU, all operations use 32-bit integer format data. For information on the supported numeric formats, see [“Numeric Formats” on page 2-16](#).

IALU Operations


Signed/Unsigned Option

The DSP always represents fixed-point numbers in 8, 16, 32, or 64 bits, using up to four 32-bit data registers. Fixed-point 32-bit data in the IALU may be two's-complement or (for the COMP instruction only) unsigned. For information on the supported numeric formats, see [“Numeric Formats” on page 2-16](#).


Circular Buffer Option

The IALU add and subtract instructions support the circular buffer (CB) option. The instructions take the form:

$$Js = Jm + | - Jn | \langle Imm8 \rangle | \langle Imm32 \rangle \{ (\{ CJMP | CB | BR \}) \} ;$$
$$Ks = Km + | - Kn | \langle Imm8 \rangle | \langle Imm32 \rangle \{ (\{ CJMP | CB | BR \}) \} ;$$

 For information on the conventions used in this instruction summary, see [“IALU Instruction Summary” on page 6-39](#).

To use the CB option, the IALU add and subtract instructions require that the related J-IALU or K-IALU base and length registers are previously set up. The IALU add and subtract instructions with the CB option calculate the modified address from the index plus or minus the modifier and also performs circular buffer wrap (if needed) as part of the calculation. The IALU puts the modified value into Js or Ks . (Jm or Km is not modified.)


 For information on circular buffer operations, see [“Circular Buffer Addressing” on page 6-27](#).

Bit Reverse Option


The IALU add and subtract instructions support the bit reverse carry (BR) option. The instructions take the form:

$$Js = Jm + | - Jn | \langle Imm8 \rangle | \langle Imm32 \rangle \{ (\{ CJMP | CB | BR \}) \} ;$$

$$Ks = Km + | - Kn | \langle Imm8 \rangle | \langle Imm32 \rangle \{ (\{ CJMP | CB | BR \}) \} ;$$

 For information on the conventions used in this instruction summary, see [“IALU Instruction Summary” on page 6-39](#).

The IALU add and subtract instructions with the BR option use bit reverse carry to calculate the result of the index plus the modifier (there is no affect on the subtract operation because there is no carry) and put the modified value into J_s or K_s . (J_m or K_m is not modified.)


 For information on bit reverse operations, see [“Bit Reverse Addressing” on page 6-31](#).

Computed Jump Option

The IALU add and subtract instructions support the computed jump (CJMP) option. The instructions take the form:

$$Js = Jm + | - Jn | \langle Imm8 \rangle | \langle Imm32 \rangle \{ (\{ CJMP | CB | BR \}) \} ;$$

$$Ks = Km + | - Kn | \langle Imm8 \rangle | \langle Imm32 \rangle \{ (\{ CJMP | CB | BR \}) \} ;$$

 For information on the conventions used in this instruction summary, see [“IALU Instruction Summary” on page 6-39](#).

The computed jump (CJMP) option directs the IALU to place the result in the program sequencer’s CJMP registers as well as the result (J_s or K_s) register.

IALU Operations

IALU Execution Status

IALU operations update status flags in the IALUs' status (JSTAT and KSTAT) registers. (See [Figure 6-2 on page 6-4](#) and [Figure 6-3 on page 6-5](#).) Programs can use status flags to control execution of conditional instructions and initiate software exception interrupts. For more information, see [“IALU Execution Conditions” on page 6-12](#).

[Table 6-1](#) shows the flags in JSTAT or KSTAT that indicate IALU status (a 1 indicates the condition) for the most recent IALU operation.

Table 6-1. IALU Status Flags

Flag	Definition	Updated By...
JZ	J-IALU zero	All J-IALU arithmetic, logical, and function ops
JN	J-IALU negative	All J-IALU arithmetic, logical, and function ops
JV	J-IALU overflow	All arithmetic ops
JC	J-IALU carry	All arithmetic ops
KZ	K-IALU zero	All K-IALU arithmetic, logical, and function ops
KN	K-IALU negative	All K-IALU arithmetic, logical, and function ops
KV	K-IALU overflow	All arithmetic ops
KC	K-IALU carry	All arithmetic ops

Flag update occurs at the end of each operation and is available on the next instruction cycle. A program cannot write to an IALU status register explicitly in the same cycle that the IALU is performing an operation.

JN/KN–IALU Negative

The JN or KN flag is set whenever the result of a J-IALU or K-IALU operation is negative. The JN or KN flag is set to the most significant bit of the result. An exception is the instructions below, in which the JN or KN flag is set differently:

- $J_s = \text{ABS } J_m$; JN is J_m (input data) sign
- $K_s = \text{ABS } K_m$; KN is K_m (input data) sign

The result sign of the above instructions is not indicated as it is always positive.

JV/KV–IALU Overflow

The JV or KV flag is an overflow indication. In all J-IALU or K-IALU operations, the bit is set when the correct result of the operation is too large to be represented by the result format. The overflow check is done always as signed operands, unless the instruction defines otherwise.

If in the following example J5 and J6 are $0 \times 70 \dots 0$ (large positive numbers), the result of the add instruction (above) will produce a result that is larger than the maximum at the given format.

J10 = J5 + J6 ;;

JC/KC–IALU Carry

The JC or KC flag is used as carry out of add or subtract instructions that can be chained. It can also be used as an indication for unsigned overflow in these operations (JV or KV is set when there is signed overflow). Bit reverse operations do not overflow and do not set the JC or KC flags.

IALU Operations

IALU Execution Conditions

In a conditional IALU instruction, the execution of the entire instruction line can depend on the specified condition at the beginning of the instruction line. Conditional IALU instructions take the form:

```
IF cond; D0, instr.; D0, instr.; D0, instruct. ; ;
```

This syntax permits up to three instructions to be controlled by a condition. Omitting the `D0` before the instruction makes the instruction unconditional.

Table 6-2 lists the IALU conditions. For more information on conditional instructions, see [“Conditional Execution” on page 7-12](#).

Table 6-2. IALU Conditions

Condition	Description	Flags Set
JEQ	J-IALU equal to zero	JZ = 1
JLT	J-IALU less than zero	JN and JZ = 1
JLE	J-IALU less than or equal to zero	JN or JZ = 1
NJEQ	NOT(J-IALU equal to zero)	JZ = 0
NJLT	NOT(J-IALU less than zero)	JN and JZ = 0
NJLE	NOT(J-IALU less than or equal to zero)	JN or JZ = 0
KEQ	K-IALU equal to zero	KZ = 1
KLT	K-IALU less than zero	KN and KZ = 1
KLE	K-IALU less than or equal to zero	KN or KZ = 1
NKEQ	NOT(K-IALU equal to zero)	KZ = 0
NKLT	NOT(K-IALU less than zero)	KN and KZ = 0
NKLE	NOT(K-IALU less than or equal to zero)	KN or KZ = 0

IALU Static Flags

In the program sequencer, the static flag (SFREG) can store status flag values for later usage in conditional instructions. With SFREG, the IALU has two dedicated static flags GSCF0 (condition is SF0) and GSCF1 (condition is SF1). The following example shows how to load a compute block condition value into a static flag register.

```
GSCF0 = JEQ ;; /* Load J-IALU JEQ flag into GSCF0 bit in static
flags (SFREG) register */
IF SF0, J5 = J4 + J3 ;; /* the SF0 condition tests the GSCF0
static flag */
```

For more information on static flags, see [“Conditional Execution” on page 7-12](#).

IALU Data Addressing and Transfer Operations

IALU data addressing instructions provide memory read and write access for loading and storing registers. For memory reads and writes, the IALU provides two types of addressing—direct addressing and indirect addressing. Both types of addressing use an index and a modifier. The index is an address, and the modifier is a value added to the index either before (pre-modify) or after (post-modify) the addressing operation. IALU addressing instruction syntax uses square brackets ([]) to set the address calculation apart from the rest of the instruction.

IALU Operations

Direct and Indirect Addressing

Direct addressing uses an index that is set to zero and uses an immediate value for the modifier. The index is pre-modified, and the modified address is used for the access. The following instruction is a register load (memory read) that uses direct addressing:

```
YR1 = [J31 + 0x00015F00] ;;  
/* This instruction reads a 32-bit word from memory location  
0x00015F00 and loads the word into register YR1. Note that J31  
always contains zero when used as an operand. */
```

Indirect addressing uses a non-zero index and uses either a register or an immediate value for the modifier. As shown in [Figure 6-4](#), there are two types of indirect addressing.

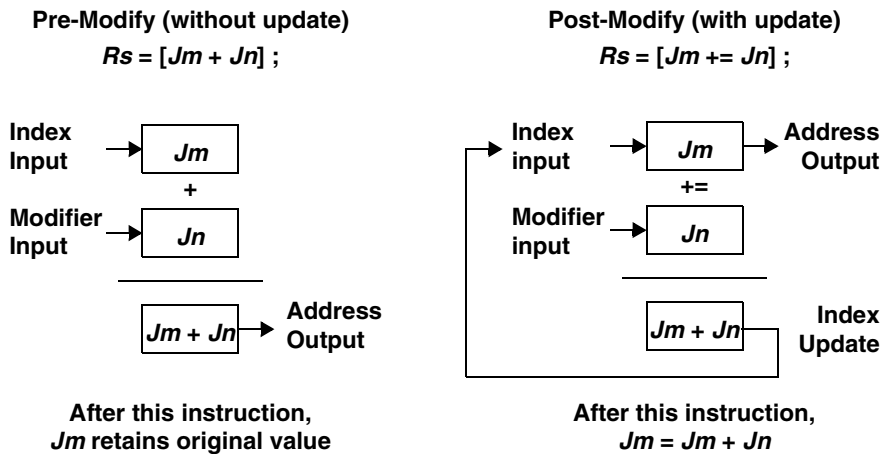



Figure 6-4. Pre- and Post-Modify Indirect Addressing

One type of indirect addressing is *pre-modify without update* (uses the [+] operator). These instructions provide memory access to the index + modifier address without changing the content of the index register. These instructions load the value into a destination register or store the value from a source register. For example:

```
XR0 = [J0 + J1] ;;
/* This instruction reads a 32-bit word from memory location
J0 + J1 (index + modifier) and loads the word into register XR0.
J0 (the index) is not updated with the address. */
```

The other type of indirect addressing is *post-modify with update* (uses the [+=] operator). These instructions provide memory access to the indexed address. These instructions load a value into a destination register or store the value from a source register. After the access, the index register is updated by the modifier value. For example:

```
XR0 = [J0 += J1] ;;
/* This instruction reads a 32-bit word from memory location J0
(the index) and loads the word into register XR0. After the
access, J0 is updated with the value J0 + J1 (index + modifier).
*/
```

 Post-modify indirect addressing is used with circular buffer and bit reversed addressing. For more information, see [“Circular Buffer Addressing” on page 6-27](#) and [“Bit Reverse Addressing” on page 6-31](#).

Normal, Merged, and Broadcast Memory Accesses

The IALU uses direct or indirect addressing to perform read and write memory accesses, loading or storing data in registers. There are three types of memory accesses—normal read/write accesses, merged read/write accesses, and broadcast read accesses. These access types differ as follows:

- Normal Read/Write Accesses –normal read/write memory accesses load or store data in universal registers. Normal accesses read or write the number of 32-bit words needed to load or store the destination or source register indicated in the instruction. Normal accesses occur when the source or destination register size matches the IALU access operator. (See [Figure 6-6](#), [Figure 6-9](#), [Figure 6-12](#), [Figure 6-14](#), [Figure 6-16](#), and [Figure 6-18](#).) Examples of normal accesses (destination \Leftarrow source) are:
 - Single register (R_s) \Leftarrow no operator
 - Dual register (R_{sd}) \Leftarrow L (long) operator
 - Quad register (R_{sq}) \Leftarrow Q (quad) operator
- Broadcast Read Access – broadcast read memory accesses load data in compute block data registers. Broadcast accesses read the number of 32-bit words needed to load the destination registers in both compute blocks with the same data as indicated in the instruction. Broadcast accesses occur when the source register size matches the IALU access operator and the register name uses XY (or no prefix) or YX to indicate both compute blocks; XY (or no prefix) and YX yield the *same* results. (See [Figure 6-7](#), [Figure 6-10](#), and [Figure 6-13](#).) Examples of broadcast accesses are (destination \Leftarrow source) are:
 - Single register (R_s) \Leftarrow no operator
 - Dual register (R_{sd}) \Leftarrow L (long) operator
 - Quad register (R_{sq}) \Leftarrow Q (quad) operator

- Merged Read/Write Accesses – merged read/write memory accesses load or store data in compute block data registers. Merged accesses read or write the number of 32-bit words needed to load or store the destination or source registers in both compute blocks with the different data as indicated in the instruction. Merged accesses occur when the source or destination register size is one-half the size indicated by the IALU access operator and the register name uses XY (or no prefix) or YX to indicate both compute blocks; XY (or no prefix) and YX syntax yield *different* results. (See [Figure 6-8](#), [Figure 6-11](#), [Figure 6-15](#), and [Figure 6-17](#).) Example merged accesses (destination \Leftrightarrow source) are:
 - Single register (R_s) \Leftrightarrow L (long) operator
 - Dual register (R_{sd}) \Leftrightarrow Q (quad) operator.



[Figure 6-6](#) through [Figure 6-17](#) show only a representative sample of memory access types. These figures only show memory accesses for data registers, not universal registers. Also, these figures only show memory accesses for post-modify, indirect addressing. For a complete list of IALU memory access instruction syntax, see “[IALU Instruction Summary](#)” on page 6-39.

Looking at [Figure 6-5](#) (memory contents) and [Figure 6-6](#) through [Figure 6-17](#) (example accesses), it is important to note the relationship between *data size* and *data alignment*. For accesses that load or store a single register, data alignment in memory is not an issue—the index address can be to any address.

For accesses that load or store dual or quad registers, data alignment in memory is important. Dual register-loads or stores must use an index address that is divisible by two (*dual aligned*). Quad-register loads or stores

IALU Operations

must use an index address that is divisible by four (*quad aligned*). Aligning data in memory is possible with assembler directives and linker description file syntax.

i DAB and SDAB accesses do not have these alignment restrictions. For more information on DAB and SDAB accesses, see [“Data Alignment Buffer \(DAB\) Accesses”](#) on page 6-23.

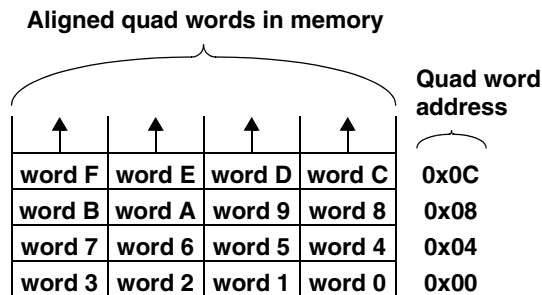
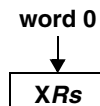


Figure 6-5. Memory Contents for Normal, Broadcast, and Merged Memory Access Examples

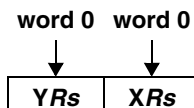
$XRs = [Jm += Jn] ;; /* Jm = address of word 0.*/$



See [Figure 6-5](#) for memory contents.

Figure 6-6. Single Register Normal Read Accesses

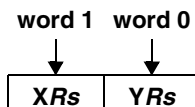
$XYRs = [Jm += Jn] ; ; /* Jm = \text{address of word } 0.* /$



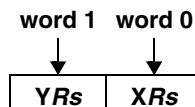
See [Figure 6-5](#) for memory contents.

Figure 6-7. Single Register Broadcast Read Accesses

$XYRs = L [Jm += Jn] ; ; /* Jm = \text{address of word } 0.* /$



$YXRs = L [Jm += Jn] ; ; /* Jm = \text{address of word } 0.* /$

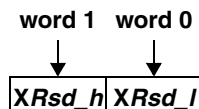


See [Figure 6-5](#) for memory contents.

Note that XY and YX syntax produce *different* results.

Figure 6-8. Single Register Merged Read Accesses

$XRsd = L [Jm += Jn] ; ; /* Jm = \text{address of word } 0.* /$

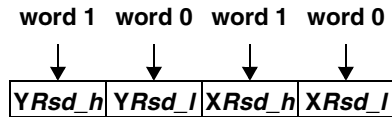


See [Figure 6-5](#) for memory contents.

Figure 6-9. Dual Register Normal Read Accesses

IALU Operations

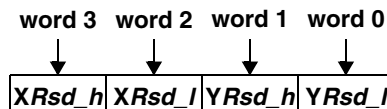
$XYRsd = L [Jm += Jn] ;; /* Jm = address of word 0.*/*$



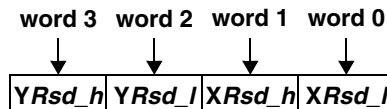
See [Figure 6-5](#) for memory contents.

Figure 6-10. Dual Register Broadcast Read Accesses

$XYRsd = Q [Jm += Jn] ;; /* Jm = address of word 0.*/*$



$YXRsd = Q [Jm += Jn] ;; /* Jm = address of word 0.*/*$

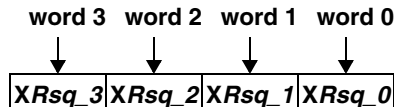


See [Figure 6-5](#) for memory contents.

Note that XY and YX syntax produce *different* results.

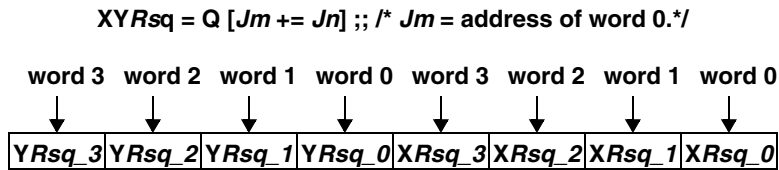
Figure 6-11. Dual Register Merged Read Accesses

$XRsq = Q [Jm += Jn] ;; /* Jm = address of word 0.*/*$



See [Figure 6-5](#) for memory contents.

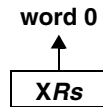
Figure 6-12. Quad Register Normal Read Accesses



See [Figure 6-5](#) for memory contents.

Figure 6-13. Quad Register Broadcast Read Accesses

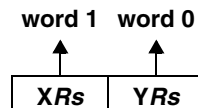
$[Jm += Jn] = XRs ;; /* Jm = address of word 0.*/$



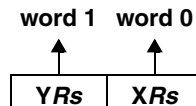
See [Figure 6-5](#) for memory contents.

Figure 6-14. Single Register Normal Write Accesses

$L [Jm += Jn] = XYRs ;; /* Jm = address of word 0.*/$



$L [Jm += Jn] = YXRs ;; /* Jm = address of word 0.*/$



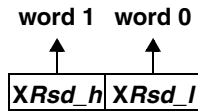
See [Figure 6-5](#) for memory contents.

Note that XY and YX syntax produce *different* results.

Figure 6-15. Single Register Merged Write Accesses

IALU Operations

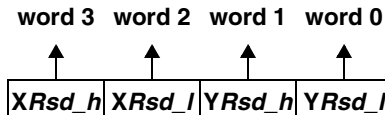
L [Jm += Jn] = XRsd ;; /* Jm = address of word 0.*/



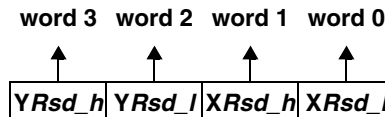
See Figure 6-5 for memory contents.

Figure 6-16. Dual Register Normal Write Accesses

Q [Jm += Jn] = XYRsd ;; /* Jm = address of word 0.*/



Q [Jm += Jn] = YXRsd ;; /* Jm = address of word 0.*/

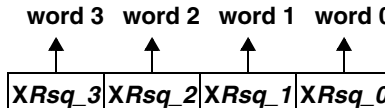


See Figure 6-5 for memory contents.

Note that XY and YX syntax produce *different* results.

Figure 6-17. Dual Register Merged Write Accesses

Q [Jm += Jn] = XRsq ;; /* Jm = address of word 0.*/




See Figure 6-5 for memory contents.

Figure 6-18. Quad Register Normal Write Accesses

Data Alignment Buffer (DAB) Accesses

Each compute block has an associated data alignment buffer (X-DAB and Y-DAB) for accessing non-aligned data. Using the DABs, programs can perform a memory read access of non-aligned quad-word data—either four normal words or eight short words—to load data into quad data registers (Rsq).

 Without using a DAB or SDAB operator, the data for dual- or quad-register load instructions must be aligned. For more information on data alignment, see “Normal, Merged, and Broadcast Memory Accesses” on page 6-16.

The DAB is a single quad-word FIFO. Aligned quad words from memory are input to the DAB, and non-aligned data for the register load is output from the DAB. The DAB uses its single quad-word buffer to hold data that crosses a quad-word boundary and uses data from the FIFO and current quad-word access to load the registers.

One way to understand DAB operation is to compare aligned versus non-aligned data access and compare DAB operations. [Figure 6-19](#) shows how the DAB operates when an IALU instruction reads aligned data from memory. Compare this to the DAB access of non-aligned data in [Figure 6-20](#).

[Figure 6-20](#) demonstrates some important points about DAB accesses. DAB accesses are intended for repeated series of memory accesses, using circular buffer addressing or linear addressing. It takes one read to prime the DAB—clear out previous data and load in the first correct data for the series—before the DAB is ready for repeated access. The DAB automatically determines the nearest quad-word boundary from the index address and reads the correct quad word from memory to load the DAB.

Because DAB accesses automatically perform circular buffer addressing, the circular buffer address registers—index, base, length, and modifier—must be set up before the DAB access begins. For this reason, DAB instructions must only use the IALU registers that support circular buffer

IALU Operations

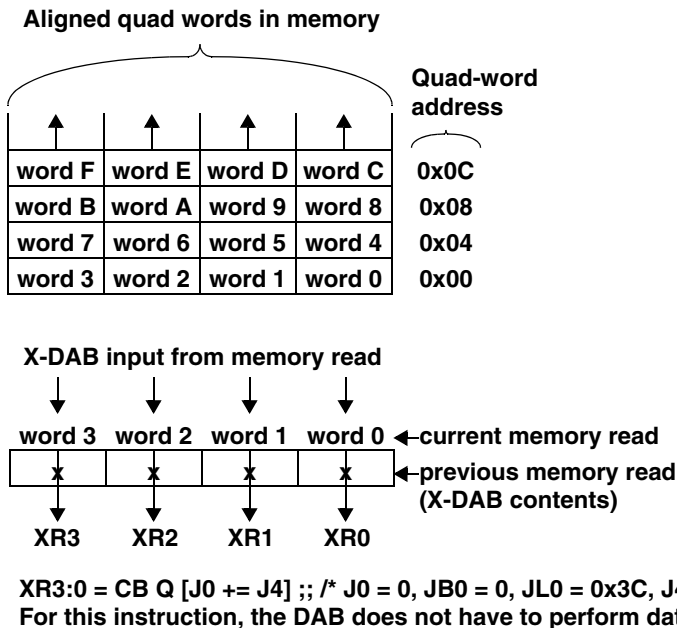
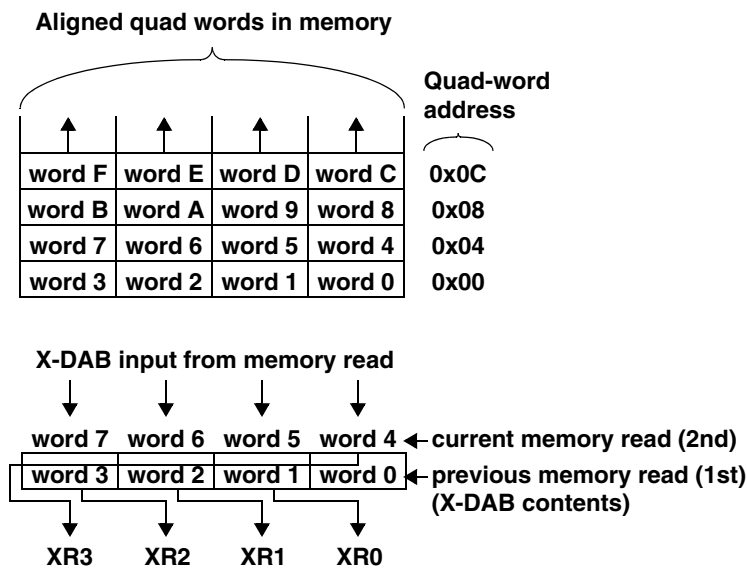


Figure 6-19. DAB Operation for Aligned Data

addressing (J_3 – J_0 , K_3 – K_0). For more information on circular buffer addressing, see “Circular Buffer Addressing” on page 6-27. If circular buffer addressing is used, the modifier value (J_n or K_n) must be equal to four to support correct DAB operation.

i If DAB accesses need to use linear addressing, set the circular buffer length (corresponding JL/KL register) to 0.



XR3:0 = DAB Q [J0 += J4] ;; /* J0 = 1, JB0 = 1, JL0 = 0x3C, J4 = 4 */
For this instruction, the DAB performs data alignment. Two reads are needed to prime the DAB. After that, each repeated read places the needed data in the DAB.

Figure 6-20. DAB Operation for Non-Aligned Data

IALU Operations

The DAB also provides access to non-aligned short word data in memory as shown in Figure 6-21. Short DAB (SDAB) access has the same requirements for setup and access as DAB access, with two exceptions. First, for correct circular buffer addressing operation the modifier value (Jn or Kn) must be equal to eight.

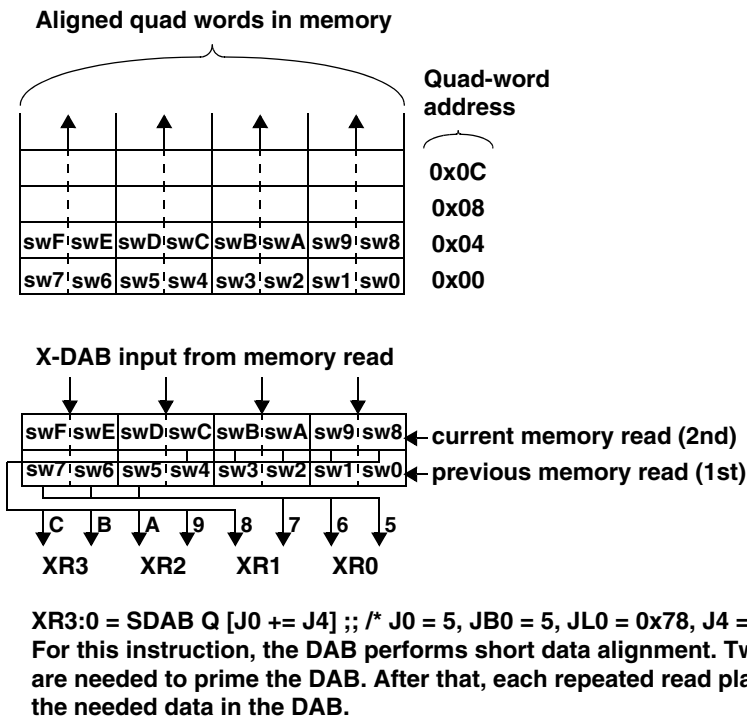


Figure 6-21. SDAB Operation for Non-Aligned Data

Second, the index value for SDAB instructions is either $2x$ (for normal word aligned short words) or $2x+1$ (for non-aligned short words). A comparison of these index values for DAB and SDAB instructions appears in Figure 6-22.

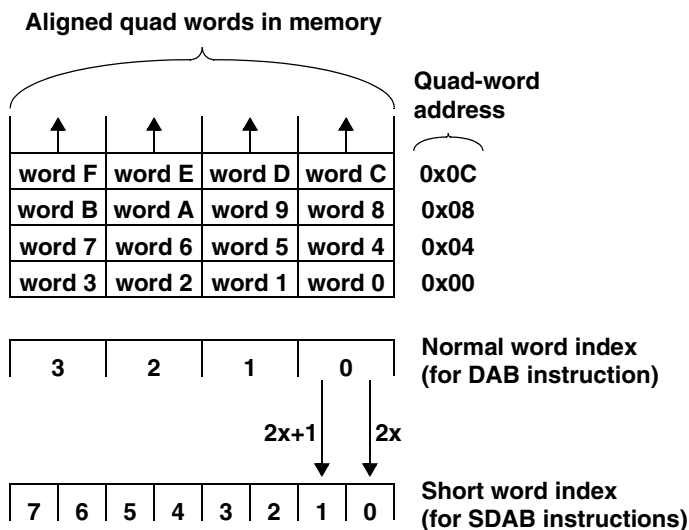


Figure 6-22. DAB Versus SDAB Index Values

Circular Buffer Addressing

The IALUs support addressing *circular buffers*—a range of addresses containing data that IALU memory accesses step through repeatedly, wrapping around to repeat stepping through the range of addresses in a circular pattern. The memory read or write access instruction uses the operator `CB` to select circular buffer addressing.



IALU Operations

To address a circular buffer, the IALU steps the index pointer through the buffer, post-modifying and updating the index on each access with a positive or negative modify value. If the index pointer falls outside the buffer, the IALU subtracts or adds the length of the buffer from or to the value, wrapping the index pointer back to the start of the buffer.

The IALUs use register file and dedicated circular buffering registers for addressing circular buffers. These registers operate as follows for circular buffering:

- The index register contains the value that the IALU outputs on the address bus. In the instruction syntax summary, the index register is represented with Jm or Km . This register can be $J3-J0$ in the J-IALU or $K3-K0$ in the K-IALU.
- The modify value provides the post-modify amount (positive or negative) that the IALU adds to the index register at the end of each memory access. The modify value can be any register file register in the same IALU as the index register. The modify value also can be an immediate value instead of a register. The size of the modify value, whether from a register or immediate, must be less than the length of the circular buffer.
- The length register must correspond to the index register; for example, $J0$ used with $JL0$, $K0$ used with $KL0$, and so on. The length register sets the size of the circular buffer and the address range that the IALU circulates the index register through. If a length register's value is zero, its circular buffer operation is disabled.
- The base register must correspond to the index register; for example, $J0$ used with $JB0$, $K0$ used with $KB0$, and so on. The base register (the buffer's base address) or the base register plus the

length register (the buffer's end address) is the value that the IALU compares the modified index value with after each access to determine buffer wraparound.

-  Circular buffer addressing may only use post-modify addressing. The IALU cannot support pre-modify addressing for circular buffering, because circular buffering requires the index be updated on each access.
-  If the JL/KL register is set to zero, then circular buffering will not be used.

Example code showing the IALU's support for circular buffer addressing appears in [Listing 6-1](#), and a description of the word access pattern for this example code appears in [Figure 6-23](#).

As shown in [Listing 6-1](#), programs use the following steps to set up a circular buffer:

1. Load the starting address within the buffer into an index register in the selected J-IALU or K-IALU. In the J-IALU, $J3-J0$ can be index registers. In the K-IALU, $K3-K0$ can be index registers.
2. Load the buffer's base address into the base register that corresponds to the index register. For example, $JB0$ corresponds to $J0$.
3. Load the buffer's length into the length register that corresponds to the index register. For example, $JL0$ corresponds to $J0$.
4. Load the modify value (step size) into a register file register in the same IALU as the index register. The J-IALU register file is $J30-J0$, and the K-IALU register file is $K30-K0$. Alternatively, an immediate value can supply the modify value.

IALU Operations

Listing 6-1. Circular Buffer Addressing Example

```
.section program ;
    JB0 = 0x100000 ;; /* Set base address */
    JL0 = 11 ;;      /* Set length of buffer */
    J0 = 0x100000 ;; /* Set location of first address */
    XRO = CB [J0 += 4] ;; /* Loads from address 0x100000 */
    XRO = CB [J0 += 4] ;; /* Loads from address 0x100004 */
    XRO = CB [J0 += 4] ;; /* Loads from address 0x100008 */
    XRO = CB [J0 += 4] ;; /* Loads from address 0x100001 */
    XRO = CB [J0 += 4] ;; /* Loads from address 0x100005 */
    XRO = CB [J0 += 4] ;; /* Loads from address 0x100009 */
    XRO = CB [J0 += 4] ;; /* Loads from address 0x100002 */
    XRO = CB [J0 += 4] ;; /* Loads from address 0x100006 */
    XRO = CB [J0 += 4] ;; /* Loads from address 0x10000A */
    XRO = CB [J0 += 4] ;; /* Loads from address 0x100003 */
    XRO = CB [J0 += 4] ;; /* Loads from address 0x100007 */
    XRO = CB [J0 += 4] ;; /* wrap to load from 0x100000 again */
```

Figure 6-23 shows the sequence order in which the IALU code in Listing 6-1 accesses the 11 buffer locations in one pass. On the twelfth access, the circular buffer wrap around occurs.

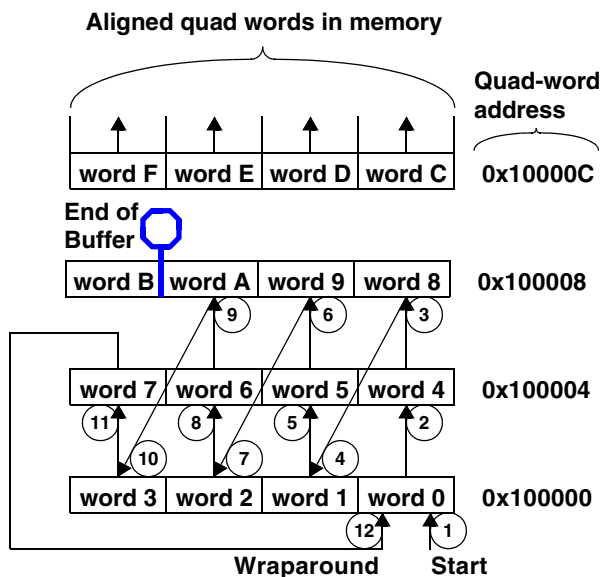


Figure 6-23. Circular Buffer Addressing – Word Access Order

Bit Reverse Addressing

The IALUs support bit reverse addressing through the bit reverse carry operator (BR). When this operator is used with an indirect post-modify read or write access, the bit wise carry moves to the right (instead of left) in the post-modify calculation.

IALU Operations

Figure 6-24 provides an example of the bit reverse carry operation. For a regular add operation $0xA5A5 + 0x2121$, the result is $0xB6B6$. For the same add operation with bit reversed carry, the result is $0x9494$.

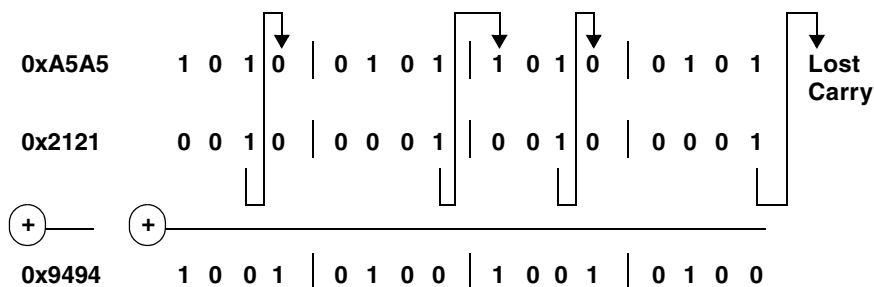


Figure 6-24. Bit Reverse Carry Operation (BR Option)

- i** As with circular buffer operations, bit reverse addressing is only performed using registers J3–J0 or K3–K0. Unlike circular buffers, the related base and length registers in the IALU do not need to be set up for bit reverse addressing.
- i** In bit reverse operations there is no overflow.

Listing 6-2 demonstrates bit reverse addressing. The word access order resulting from the bit reverse carry on the address appears in Figure 6-25. Some important points to remember when reading Listing 6-2 include:

- The number of bit reverse locations (length of buffer) must be a power of 2 (for example, buffer length = 2^n), but the start address for the data buffer to be addressed must be aligned to an address

that is a multiple of the number of locations in the buffer. In this case, the length of the buffer is 8, and the buffer start address is aligned to an address that is a multiple of 8.

- The assembler provides evaluation of expressions. For example, where $N/2$ appears in the code, the assembler evaluates the expression as 4. Also, the assembler `ADDRESS()` operator calculates the address of a symbol.
- For the repeated bit reversed accesses in the loop (`_my_loop`), the data written to the address pointed to by `J4` on each loop iteration is 0, 4, 2, 6, 1, 5, 3, then 7. Note the bit reverse carry operations on each repeated read access and note how they affect the address for each access as shown in [Table 6-3](#).
- [Listing 6-2](#) uses a conditional jump instruction to loop through repeated memory read and write accesses. For more information on conditional execution, see [“Conditional Execution” on page 7-12](#).

IALU Operations

Table 6-3. Post-Modify Operations With BR Addition

Loop Iteration	J0	XR0	BR [J0 += 4] note bit reverse carry (BRC) “...” indicates 1000 0000 0000
0	0x1000 0000	0x0	b#...0000 + b#0100 = b#...0100
1	0x1000 0004	0x4	b#...0100 + b#0100 = b#...0010 (BRC)
2	0x1000 0002	0x2	b#...0010 + b#0100 = b#...0110
3	0x1000 0006	0x6	b#...0110 + b#0100 = b#...0001 (BRCs)
4	0x1000 0001	0x1	b#...0001 + b#0100 = b#...0101
5	0x1000 0005	0x5	b#...0101 + b#0100 = b#...0011 (BRCs)
6	0x1000 0003	0x3	b#...0011 + b#0100 = b#...0111
7	0x1000 0007	0x7	b#...0111 + b#0100 = b#...0000 (BRCs)

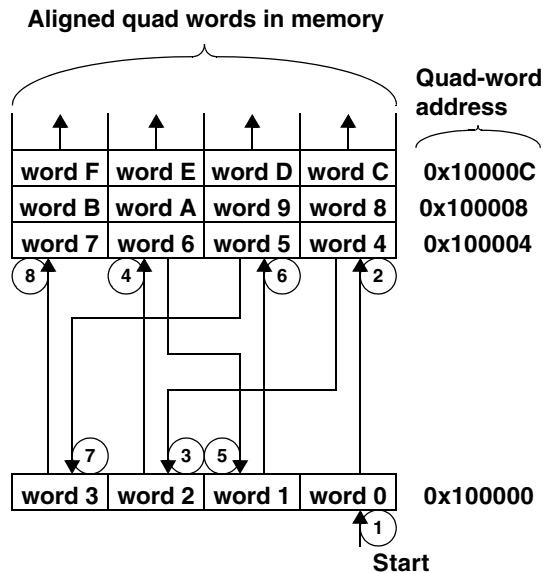


Figure 6-25. Bit Reverse Addressing – Word Access Order

Listing 6-2. Bit Reverse Addressing Example

```

#define N 8 /* N = 8; number of bit reverse locations; N must be
a power of 2 */
.section data1;
    .align N;
/* align the input buffer's start to an address that is a multi-
ple of N; assume for this example that the address is
0x1000 0000 0000 0000 */
    .var input[N]={0,1,2,3,4,5,6,7};
    .var output[N];
.section program;
_main:
    j0 = j31 + ADDRESS(input) ;; /* Input pointer */
    j4 = j31 + ADDRESS(output) ;; /* Output pointer */
    LC0 = N ;; /* Set up loop counter */
_my_loop:
    xr0 = BR [J0 += N/2] ;; /* Data read with bit reverse; modi-
fier must be equal to N/2 */
    if NLC0E, jump _my_loop ; [j4+=1] = xr0 ;; /* Write linear */

```

Universal Register Transfer Operations

The IALUs support data transfers between universal registers. Also, the IALUs support loading universal registers with 15- or 32-bit immediate data. The *Ureg* transfer and load instructions supported by the IALUs include:

```

Ureg_s = <Imm15>|<Imm32> ; /* load a single Ureg with 15- or
32-bit immediate data */

```

```

Ureg_s = Ureg_m ; /* transfer the contents of a single (32-bit)
Ureg to another Ureg */

```

IALU Operations

```
Ureg_sd = Ureg_md ; /* transfer the contents of a dual (64-bit)
Ureg to another Ureg */
/* Numbered registers in compute block or IALU register files may
be treated as dual registers */
```

```
Ureg_sq = Ureg_mq ; /* transfer the contents of a quad (128-bit)
Ureg to another Ureg */
/* Numbered registers in compute block or IALU register files may
be treated as quad registers */
```

Immediate Extension Operations

Many IALU instructions permit immediate data as an operand. When the immediate data is larger than 8 bits for data addressing instructions or larger than 15 bits for universal register load instructions, the data is too large to fit within one 32-bit instruction. To hold this large immediate data, the DSP uses two instruction slots—one for the instruction and one for the extended data. Looking at the IALU instructions listed in the [“IALU Instruction Summary” on page 6-39](#), note the number of instructions that can use 32-bit immediate data (<Imm32>). These instructions all require an immediate extension.

The DSP automatically supports immediate extensions, but the programmer must be aware that an instruction requires an immediate extension and leave an unused slot for the extension. For example, use three (not four) slots on a line in which the DSP must automatically use the second slot for the immediate extension, and place the instruction that needs the extension in the first slot of the instruction line.



Note that only one immediate extension may be in a single instruction line.

IALU Examples

[Listing 6-3](#) and [Listing 6-4](#) provide a number of example IALU arithmetic and data addressing instructions. The comments with the instructions identify the key features of the instruction, such as operands, destination or source addresses, addressing operation, and register usage.

Listing 6-3. IALU Instruction Examples

```
J1 = J0 + 0x81 ; /* Js = Jm + Imm8 data */

K2 = ROTR K0 ; /* Ks = 1 bit right rotate Km */

XSTAT = [J3 + J5] ; /* Load Ureg from addr. Jm + Jn */

J5:4 = L [J2 += 0x81] ; /* Load Ureg_sd from addr. Jm, and
post-modify Jm with Imm8 */

J31:28 = Q [K2 + K5] ; /* Load Ureg_sq from addr. Km + Jn */

XR2 = CB [J0 += 0x8181] ; /* Load Rs from addr. Jm; post-modify Jm
with Imm32 and circular buffer addressing; uses immediate exten-
sion */

YSTAT = 0x8181 ; /* Load Ureg_s from Imm32; */

KSTAT = JSTAT ; /* Load Ureg_s from Ureg_m */

YR3:0 = XR7:4 ; /* Load Ureg_sq from Ureg_mq */
```

IALU Examples

Listing 6-4. DAB Usage Example

```
#define N 8
.SECTION data1 ;
.VAR abuff[5] = 0x0, 0x1, 0x2, 0x3, 0x4 ;
.SECTION external ;
.VAR destx[N] = 0x00000000, 0x11111111,
                0x22222222, 0x33333333,
                0x44444444, 0x55555555,
                0x66666666, 0x77777777;
.VAR desty[N] ;
.SECTION program ;
.GLOBAL _main;
_main:
    J0 = 0x3 ;;
    J1 = J0 + destx ;;
    /* The program needs to load in 0x33333333 through
       0x66666666 with a quad load, which isn't aligned
       correctly. So, it must use DAB. */
    /* xR3:0 = Q[J1 += 4];; This command gives a runtime error
       (not a compiler error), access to misaligned memory */
    XR3:0 = DAB Q[J1 += 4] ;;
    /* For quad access, must have modify value of 4 for next
       DAB access */
    XR3:0 = DAB Q[J1 += 4] ;;
    /* DAB access takes two of the same commands. First loads in
       nearest quad boundary, and the second loads in correct
       value. Now, every access to the same buffer is
       aligned. */
    JL3 = 0x5 ;; /* Use circular buffers */
    JB3 = abuff ;;
    J3 = abuff ;; /* Must set pointer and JB3 */
    NOP ;;
    NOP;;
```

```

/* Need 4-cycle latency between JB and JL setup and
   its use */
/* set up loop to count through 12 steps */
LC0 = 12 ;; /*initialize loop counter*/
start_loop:
XR0 = CB[J3 += 0x1] ;;
/* uses CB to institute circular buffer */
/* once the use of J3 is a circular buffer, any use
   of J3 is a circular buffer*/
/* To rest, reset JL3 and JB3 */
IF NLC0E, JUMP start_loop ;;
/* execute loop while loop counter doesn't equal zero */
__lib_prog_term:
jump __lib_prog_term (NP);;
/* Done. */

```

IALU Instruction Summary

The following listings show the IALU instructions' syntax:

- [Listing 6-5](#) “IALU Integer, Logical, and Function Instructions”
- [Listing 6-6](#) “IALU Ureg Register Load (Data Addressing) Instructions”
- [Listing 6-7](#) “IALU Dreg Register Load Data Addressing (and DAB Operation) Instructions”
- [Listing 6-8](#) “IALU Ureg Register Store (Data Addressing) Instructions”
- [Listing 6-9](#) “IALU Dreg Register Store (Data Addressing) Instructions”
- [Listing 6-10](#) “IALU Universal Register Transfer Instructions”

IALU Instruction Summary

The conventions used in these listings for representing register names, optional items, and choices are covered in detail in “[Register File Registers](#)” on page 2-5. Briefly, these conventions are:

- { } – the curly braces enclose options; these braces are not part of the instruction syntax.
- | – the vertical bars separate choices; these bars are not part of the instruction syntax.
- *Jm* or *Km* – the letter J or K in register names in italic indicate selection of a register in the J-IALU or K-IALU.
- *Jm* – the register names in italic represent user selectable single (*Jm*, *Jn*, *Js*, *Rs*, *Ureg_s*), double (*Rsd*, *Rmd*, *Ureg_sd*, *Ureg_md*) or quad (*Rsq*, *Rmq*, *Ureg_sq*, *Ureg_mq*) register names.

In IALU data addressing instructions, special operators identify the input and output operand size as follows:

- The L operator before an input or output operand (indirect or direct address) indicates a long word (64-bit) memory read or write. For example, the following instruction syntax indicates long word memory read: $Rsd = L [Km += Kn] ;$
- The Q operator before an input or output operand (indirect or direct address) indicates a quad-word (128-bit) memory read or write. For example, the following instruction syntax indicates quad-word memory read: $Rsq = Q [Km += Kn] ;$
- The absence of an L or Q operator before an input or output operand (indirect or direct address) indicates a normal-word (32-bit) memory read or write. For example, the following instruction syntax indicates normal-word memory read: $Rs = [Km += Kn] ;$



Each instruction presented here occupies one instruction slot in an instruction line, except for those using instructions which require immediate extensions. For more information about instruction

lines and instruction combination constraints, see “[Immediate Extension Operations](#)” on page 6-36, “[Instruction Line Syntax and Structure](#)” on page 1-20, and “[Instruction Parallelism Rules](#)” on page 1-24.

IALU Instruction Summary

Listing 6-5. IALU Arithmetic, Logical, and Function Instructions

```
Js = Jm +|- Jn|<Imm8>|<Imm32> {{(CJMP|CB|BR)}} ;  
JB0|JB1|JB2|JB3|JL0|JL1|JL2|JL3 = Jm +|- Jn|<Imm8>|<Imm32> ;  
Js = Jm + Jn|<Imm8>|<Imm32> + JC ;  
Js = Jm - Jn|<Imm8>|<Imm32> + JC - 1 ;  
Js = (Jm +|- Jn|<Imm8>|<Imm32>)/2 ;  
COMP(Jm, Jn|<Imm8>|<Imm32>) {(U)} ;  
Js = MAX|MIN (Jm, Jn|<Imm8>|<Imm32>) ;  
Js = ABS Jm ;  
Js = Jm OR|AND|XOR|AND NOT Jn|<Imm8>|<Imm32> ;  
Js = NOT Jm ;  
Js = ASHIFTR|LSHIFTR Jm ;  
Js = ROTR|ROTL Jm ;  
  
Ks = Km +|- Kn|<Imm8>|<Imm32> {{(CJMP|CB|BR)}} ;  
KB0|KB1|KB2|KB3|KLO|KL1|KL2|KL3 = Km +|- Kn|<Imm8>|<Imm32> ;  
Ks = Km + Kn|<Imm8>|<Imm32> + KC ;  
Ks = Km - Kn|<Imm8>|<Imm32> + KC - 1 ;  
Ks = (Km +|- Kn|<Imm8>|<Imm32>)/2 ;  
COMP(Km, Kn|<Imm8>|<Imm32>) {(U)} ;  
Ks = MAX|MIN (Km, Kn|<Imm8>|<Imm32>) ;  
Ks = ABS Km ;  
Ks = Km OR|AND|XOR|AND NOT Kn|<Imm8>|<Imm32> ;  
Ks = NOT Km ;  
Ks = ASHIFTR|LSHIFTR Km ;  
Ks = ROTR|ROTL Km ;
```

Listing 6-6. IALU Ureg Register Load (Data Addressing) Instructions

```

Ureg_s = [Jm +|+= Jn|<Imm8>|<Imm32>] ;
Ureg_sd = L [Jm +|+= Jn|<Imm8>|<Imm32>] ;
Ureg_sq = Q [Jm +|+= Jn|<Imm8>|<Imm32>] ;

Ureg_s = [Km +|+= Kn|<Imm8>|<Imm32>] ;
Ureg_sd = L [Km +|+= Kn|<Imm8>|<Imm32>] ;
Ureg_sq = Q [Km +|+= Kn|<Imm8>|<Imm32>] ;

/* Ureg suffix indicates: _s=single, _sd=double, _sq=quad */

```

Listing 6-7. IALU Dreg Register Load Data Addressing (and DAB Operation) Instructions

```

{X|Y|XY}Rs = {CB|BR} [Jm += Jn|<Imm8>|<Imm32>] ;
{X|Y|XY}Rsd = {CB|BR} L [Jm += Jn|<Imm8>|<Imm32>] ;
{XY|YX}Rs = {CB|BR} L [Jm += Jn|<Imm8>|<Imm32>] ;
{X|Y|XY}Rsq = {CB|BR|DAB|SDAB} Q [Jm += Jn|<Imm8>|<Imm32>] ;
{XY|YX}Rsd = {CB|BR|DAB|SDAB} Q [Jm += Jn|<Imm8>|<Imm32>] ;

{X|Y|XY}Rs = {CB|BR} [Km += Kn|<Imm8>|<Imm32>] ;
{X|Y|XY}Rsd = {CB|BR} L [Km += Kn|<Imm8>|<Imm32>] ;
{XY|YX}Rs = {CB|BR} L [Km += Kn|<Imm8>|<Imm32>] ;
{X|Y|XY}Rsq = {CB|BR|DAB|SDAB} Q [Km += Kn|<Imm8>|<Imm32>] ;
{XY|YX}Rsd = {CB|BR|DAB|SDAB} Q [Km += Kn|<Imm8>|<Imm32>] ;

/* R suffix indicates: _s=single, _sd=double, _sq=quad */
/* m must be 0,1,2, or 3 for bit reverse or circular buffers */

```

IALU Instruction Summary

Listing 6-8. IALU Ureg Register Store (Data Addressing) Instructions

```
[Jm +|+= Jn|<Imm8>|<Imm32>] = Ureg_s ;  
L [Jm +|+= Jn|<Imm8>|<Imm32>] = Ureg_sd ;  
Q [Jm +|+= Jn|<Imm8>|<Imm32>] = Ureg_sq ;  
  
[Km +|+= Kn|<Imm8>|<Imm32>] = Ureg_s ;  
L [Km +|+= Kn|<Imm8>|<Imm32>] = Ureg_sd ;  
Q [Km +|+= Kn|<Imm8>|<Imm32>] = Ureg_sq ;
```

Listing 6-9. IALU Dreg Register Store (Data Addressing) Instructions

```
{CB|BR} [Jm += Jn|<Imm8>|<Imm32>] = {X|Y}Rs ;  
{CB|BR} L [Jm += Jn|<Imm8>|<Imm32>] = {X|Y}Rsd ;  
{CB|BR} L [Jm += Jn|<Imm8>|<Imm32>] = {XY|YX}Rs ;  
{CB|BR} Q [Jm += Jn|<Imm8>|<Imm32>] = {X|Y}Rsq ;  
{CB|BR} Q [Jm += Jn|<Imm8>|<Imm32>] = {XY|YX}Rsd ;  
  
{CB|BR} [Km += Kn|<Imm8>|<Imm32>] = {X|Y}Rs ;  
{CB|BR} L [Km += Kn|<Imm8>|<Imm32>] = {X|Y}Rsd ;  
{CB|BR} L [Km += Kn|<Imm8>|<Imm32>] = {XY|YX}Rs ;  
{CB|BR} Q [Km += Kn|<Imm8>|<Imm32>] = {X|Y}Rsq ;  
{CB|BR} Q [Km += Kn|<Imm8>|<Imm32>] = {XY|YX}Rsd ;  
  
/* R suffix indicates: _s=single, _sd=double, _sq=quad */  
/* m = 0,1,2 or 3 for bit reverse or circular buffers */
```

Listing 6-10. IALU Universal Register Transfer Instructions

```
Ureg_s = <Imm15>|<Imm32> ;  
Ureg_s = Ureg_m ;  
Ureg_sd = Ureg_md ;  
Ureg_sq = Ureg_mq ;
```

7 PROGRAM SEQUENCER

The TigerSHARC processor core contains a program sequencer. The sequencer contains the instruction alignment buffer (IAB), program counter (PC), branch target buffer (BTB), interrupt manager, and address fetch mechanism. Using these features and the instruction pipeline, the sequencer (highlighted in [Figure 7-1](#)) manages program execution.

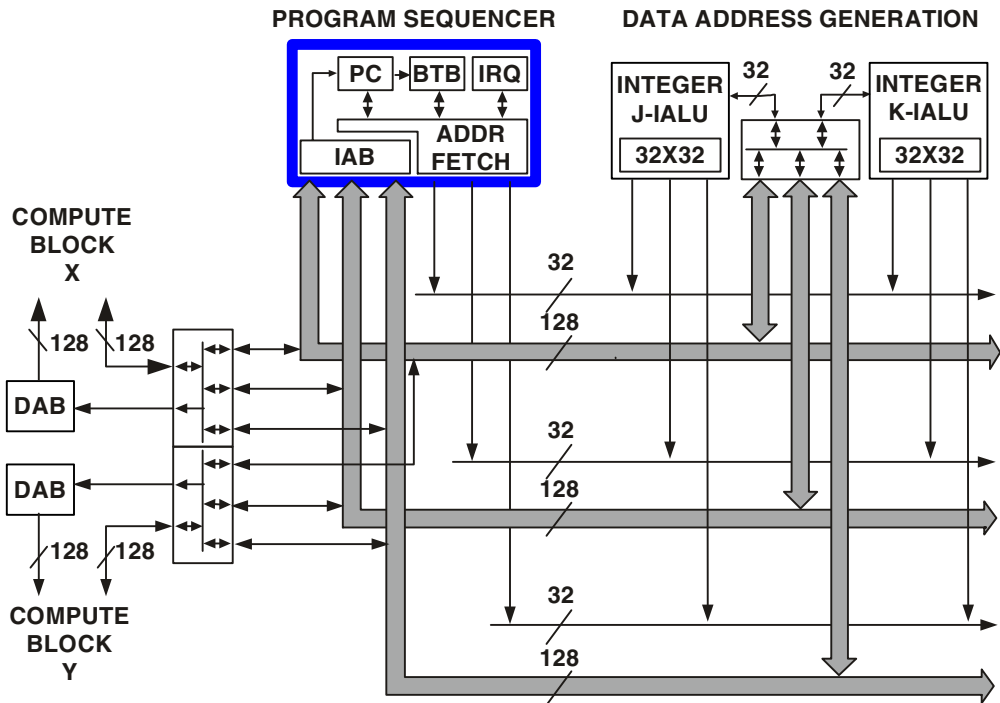


Figure 7-1. Program Sequencer

The sequencer fetches instructions from memory and executes program flow control instructions. The operations that the sequencer supports include:

- Supply address of next instruction to fetch
- Maintain instruction alignment buffer (IAB) by buffering fetched instructions
- Maintain branch target buffer (BTB) by reducing branch delays
- Decrement loop counters
- Evaluate conditions (for conditional instructions)
- Respond to interrupts (with changes to program flow)

[Figure 7-2](#) shows a detailed block diagram of the sequencer. Looking at this diagram, note the following blocks within the sequencer.

- The program counter (PC) increments the fetch address for linear flow or modifies the fetch address as needed for non-linear flow (branches, loops, interrupts, or others).
- The branch target buffer (BTB) caches addresses for branches to reduce pipeline costs on predicted branches. For more information, see [“Branching Execution” on page 7-16](#).
- The fetch unit puts the address on the bus for the next quad word to fetch from memory.
- The instruction alignment buffer (IAB) receives the instruction quad words from memory, buffers them, and distributes the instructions to the compute blocks, IALUs, and program sequencer.

With the functional blocks shown in [Figure 7-2](#), the sequencer can support a number of program flow variations. Program flow in the DSP is mostly linear with the processor executing program instructions sequen-

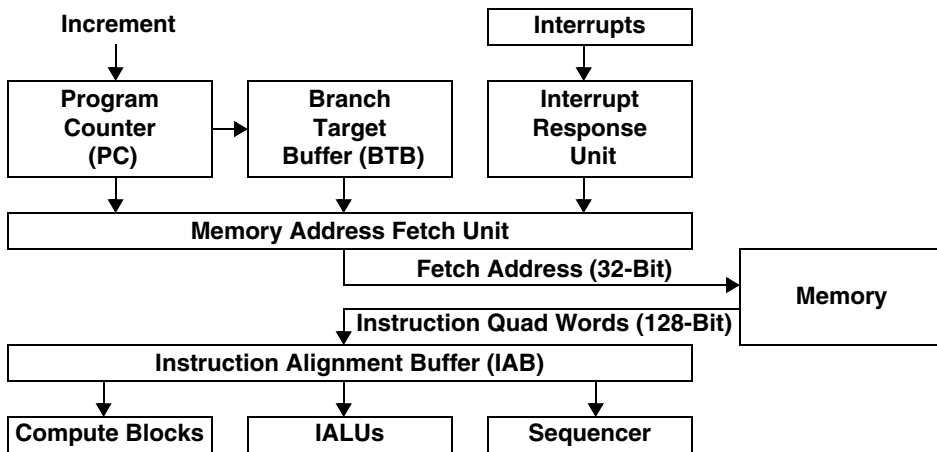


Figure 7-2. Sequencer Detailed Block Diagram

tially. This linear flow varies occasionally when the program uses non-sequential program structures, such as those illustrated in [Figure 7-3](#). Non-sequential structures direct the DSP to execute an instruction that is not at the next sequential address. These structures include:

- **Loops.** One sequence of instructions executes several times with near-zero overhead.
- **Subroutines.** The processor temporarily redirects sequential flow to execute instructions from another part of program memory.
- **Jumps.** Program flow transfers permanently to another part of program memory.
- **Interrupts.** Subroutines in which a runtime event triggers the execution of the routine.
- **Idle.** An instruction that causes the processor to cease operations, holding its current state until an interrupt occurs. Then, the processor services the interrupt and continues normal execution.

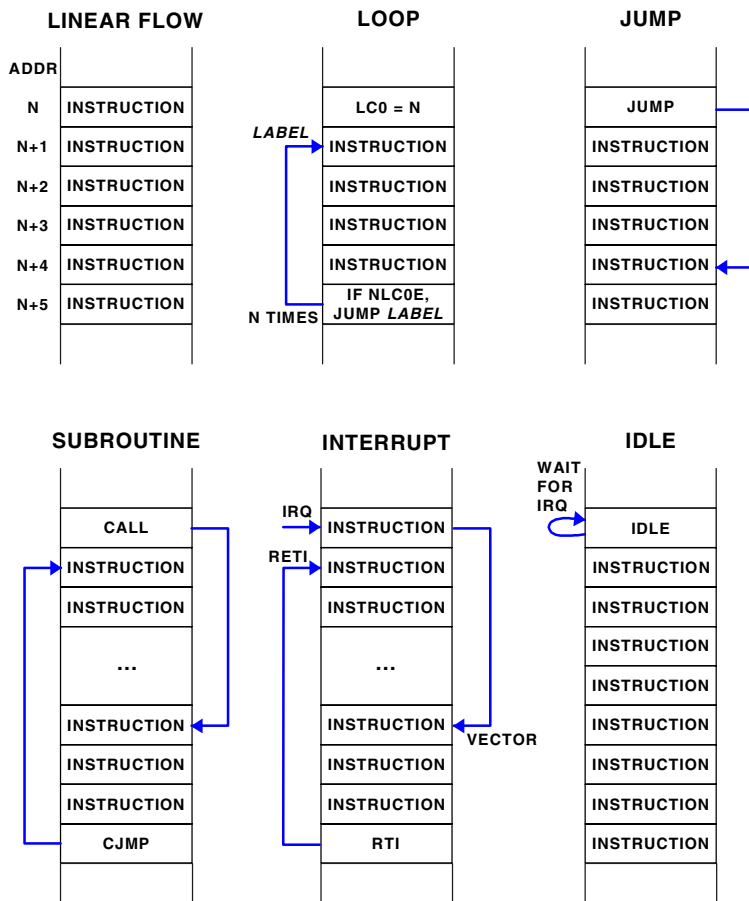


Figure 7-3. Program Flow Variations

For information on using each type of program flow, see [“Sequencer Operations” on page 7-7](#).

To support optimized flow of execution, the TigerSHARC processor uses an instruction pipeline. The DSP fetches quad words from memory, parses the quad words into instruction lines (consisting of one to four instructions), decodes the instructions, and executes them. [Figure 7-4](#)

shows the instruction pipeline stages and shows how the pipeline interacts with the branch target buffer (BTB) and instruction alignment buffer (IAB).

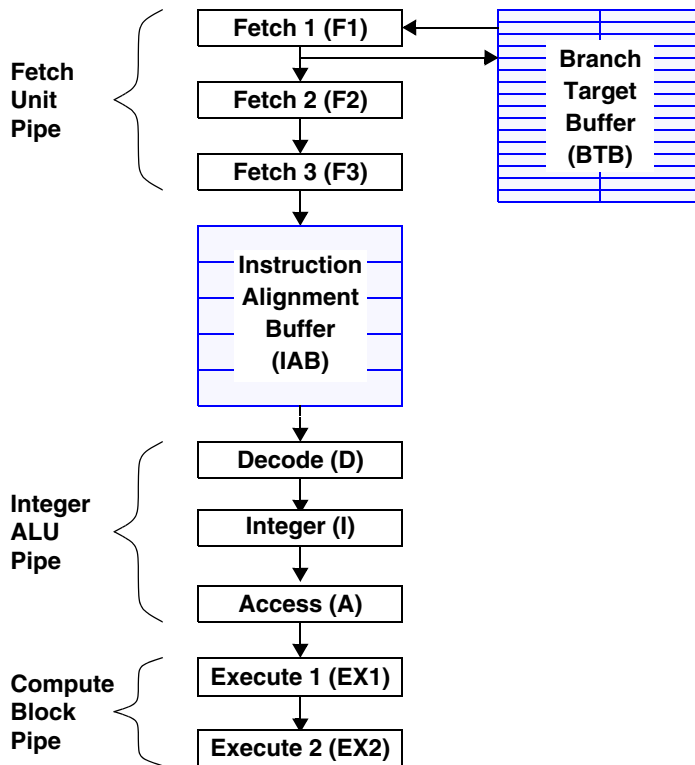


Figure 7-4. Instruction Pipeline, IAB, and BTB

From start to finish, an instruction line requires eight cycles to traverse the pipeline. The execution throughput is one instruction line every DSP core clock (CCLK) cycle. Due to the execution of IALU instructions at the integer (I) stage, execution of compute block instructions at the execute 2 (EX2) stage, and memory access effects, the full flow cannot be analyzed as a single eight-stage pipeline, but must be viewed as sequential unit pipes.

The instruction fetch (F1, F2, and F3) pipe stages are common to all instructions and are driven by memory accesses. The F1 stage interacts with the BTB to minimize overhead cycles when branching. For more information, see [“Branching Execution” on page 7-16](#) and [“Branch Target Buffer \(BTB\)” on page 7-34](#).

The remaining pipe stages are instruction driven. The execution differs between the IALU, compute block, and sequencer. The instruction driven pipe stages are Decode (D), Integer (I), operand Access (A), Execute 1 (EX1), and Execute 2 (EX2).

In the instruction driven pipeline stages, instruction pipe details differ according to the unit executing the instruction:

Decode	<p>The sequencer extracts the next instruction line, uses the IAB to distribute it to the respective execution units (compute blocks, IALUs, and sequencer), and updates the program counter. For more information, see “Instruction Alignment Buffer (IAB)” on page 7-31.</p> <p>The IALUs decode instructions.</p> <p>The compute blocks transfer instructions to the computation units (ALU, multiplier, and shifter).</p>
Integer	<p>The IALUs execute arithmetic instruction, return results, and update flags.</p> <p>The computational units decode instruction and check for dependencies.</p>
Access	<p>The IALUs begin memory access (for applicable instructions).</p> <p>The computational units select source registers in the register files.</p>

Execute 1 and 2 The IALUs complete execution of memory access instruction.


The computational units complete execution, return results, and update flags.

These differences in pipe operation between IALUs and computational units cause different pipeline effects when branching. For more information, see [“Instruction Pipeline Operations” on page 7-26](#).

Sequencer Operations

Sequencer operations support linear execution (one sequential address after another) and non-linear execution (transferring execution to a non-sequential address). These operations are described in these sections:

- [“Conditional Execution” on page 7-12](#)
- [“Branching Execution” on page 7-16](#)
- [“Looping Execution” on page 7-19](#)
- [“Interrupting Execution” on page 7-20](#)

 Depending on programming and pipeline effects, some branching variations require the DSP to automatically insert stall cycles. For more information on pipeline effects and stalls, see [“Instruction Pipeline Operations” on page 7-26](#).

Conditional instructions begin with the syntax `IF Condition`. The *Condition* can be any status flag from a compute block or IALU operation (for example, AZ, JZ, MV, and others), any static flag from the SFREG register, or a negated status/static flag (prefixed with N).

Sequencer Operations

Almost all TigerSHARC processor instructions can be conditional. The exceptions are a small set of sequencer instructions that may not be conditional. The always unconditional instructions are NOP, IDLE, BTBINV, TRAP, and EMUTRAP.

In the sequencer, there are two registers that provide control and status. The sequencer control (SQCTL) and sequencer status (SQSTAT) registers appear in [Figure 7-5](#), [Figure 7-7](#), [Figure 7-6](#), and [Figure 7-8](#). These registers support:

- **Normal mode.** Using the normal mode (NMOD) bit, programs select user mode (=0) in which programs can only access the compute block and IALU registers or supervisor mode (=1) in which programs have unlimited register access. After booting and when responding to an interrupt, the DSP automatically goes into supervisor mode. There is a three-cycle latency on explicitly switching between these modes.
- **Branch target buffer control.** Using the BTB enable (BTBEN) and BTB lock (BTBLK) bits, programs control the BTB operation.
- **Timer control.** Using the timer run (TMR0RN and TMR1RN) bits, programs can turn on the two timers independently.
- **Flag control and status.** Using the flag enable (FLAG_x_EN), flag output (FLAG_x_OUT), and flag input (FLG_x) bits, programs can select whether a flag pin (FLAG₃₋₀) is an input or an output. If an output, select 1 or 0 for the output value. If an input, observe the input value.¹

¹ The flag pin bit updates the corresponding flag pin after a delay of between 1 and 3 SCLK cycles. Setting and clearing a flag bit might not affect the pin if both operations occur during that delay. The recommended way to set and clear the flag pin bit is to get an external indication that the bit has been set before clearing it. Otherwise, insert (2 x LCLKRAT) instruction lines between the set and the clear to compensate for the delay.

- **Interrupt sensitivity.** Using the interrupt edge (`IRQ_EDGE`) bit, programs select edge or level sensitivity for the `IRQ3-0` pins independently.
- **Software reset.** Using the software reset (`SWRST`) bit, programs can reset the DSP core.



Other status information is also available. See [Figure 7-5](#), [Figure 7-7](#), [Figure 7-6](#), and [Figure 7-8](#).

Sequencer Operations

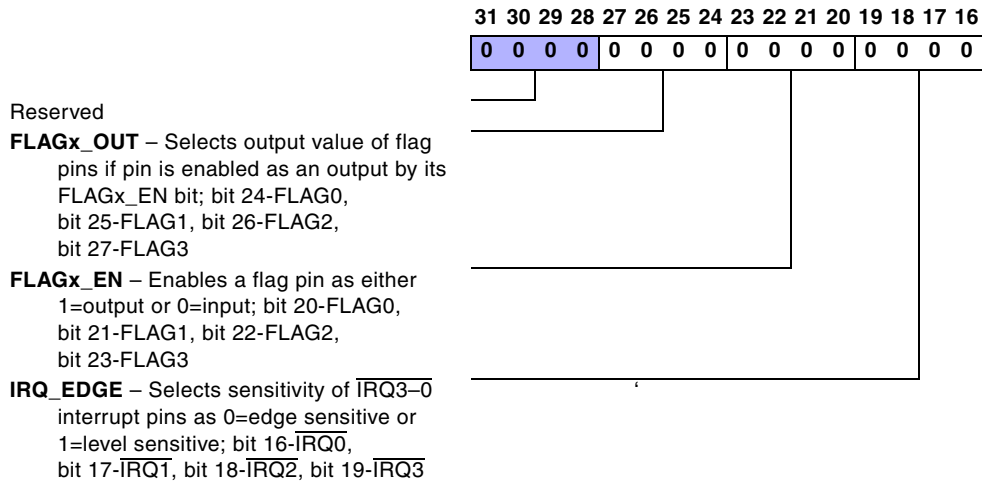


Figure 7-5. SQCTL (Upper) Register Bit Descriptions

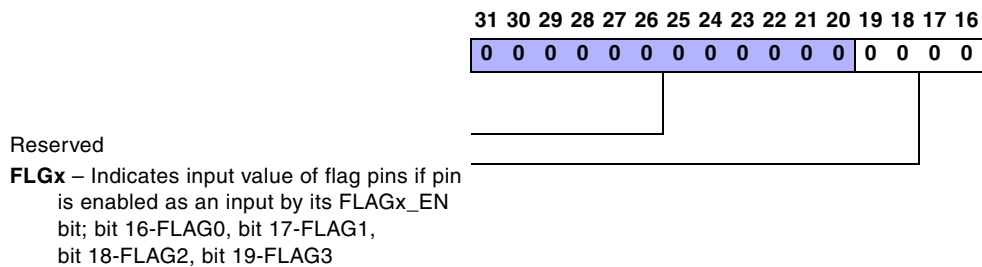


Figure 7-6. SQSTAT (Upper) Register Bit Descriptions

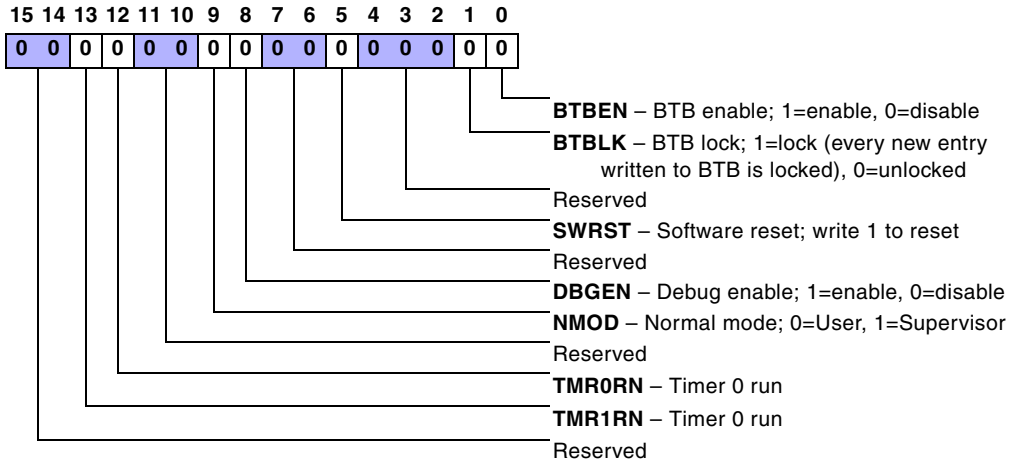


Figure 7-7. SQCTL (Lower) Register Bit Descriptions

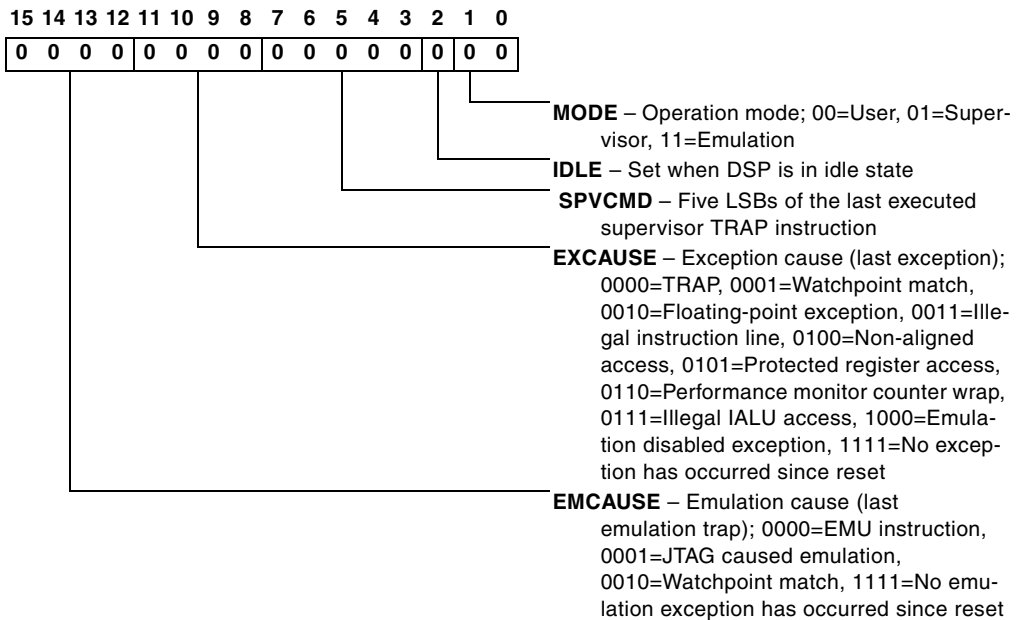


Figure 7-8. SQSTAT (Lower) Register Bit Descriptions

Conditional Execution

All TigerSHARC processor instructions¹ can be executed conditionally (a mechanism also known as predicated execution). The condition field exists in one instruction in an instruction line, and all the remaining instructions in that line either execute or not, depending on the outcome of the condition.

In a conditional computational instruction, the execution of the entire instruction line can depend on the specified condition at the beginning of the instruction line. Conditional computational instructions take the form:

```
IF Condition;  
    D0, Instruction; D0, Instruction; D0, Instruction ;;
```

This syntax permits up to three instructions to be controlled by a condition. Omitting the `D0` before the instruction makes the instruction unconditional.

[Listing 7-1](#) shows some example conditional ALU instructions. For a description of the ALU conditions used in these examples, see “[ALU Execution Conditions](#)” on page 3-14.

Listing 7-1. Conditional Compute and IALU Instructions

```
IF XALT; D0, R3 = R1 + R2 ;;  
/* conditional execution of the add in compute blocks X and Y is  
based on the ALT condition in compute block X */  
  
IF YAEQ; D0, XR0 = R1 + R2 ;;  
/* conditional execution of the add in compute block X is based  
on the AEQ condition in compute block Y */
```

¹ Except for NOP, IDLE, BTBINV, TRAP, and EMUTRAP


```
IF ALE; D0, R0 = R1 + R2 ;;
/* conditional execution of the add in compute blocks X and Y is
based on the ALE condition in compute block X for execution in X
and is based on the ALE condition in compute block Y for execu-
tion in Y */

IF ALE; D0, XR0 = R1 + R2 ;;
/* conditional execution of the add in compute block X is based
on the ALE condition in compute block X */

IF ALE; D0, R0 = [J0 + J1] ;;
/* conditional execution of load is based on ORing the ALE condi-
tion in compute blocks X and Y */
```

In a conditional program sequencer instruction that is based on an ALU condition, the execution of the program sequencer instruction line depends on the specified ALU condition at the beginning of the instruction line. These conditional program sequencer instructions take the form:

```
IF Condition, Sequencer_Instruction;
   ELSE, Instruction; ELSE, Instruction; ELSE, Instruction ;;
```

This syntax permits up to three instructions to be controlled by a condition. Omitting the `ELSE` before the instruction makes the instruction unconditional.

This syntax permits program sequencer instructions to be based on computational conditions. [Listing 7-2](#) shows some example conditional program sequencer instructions based on ALU conditions.

Sequencer Operations

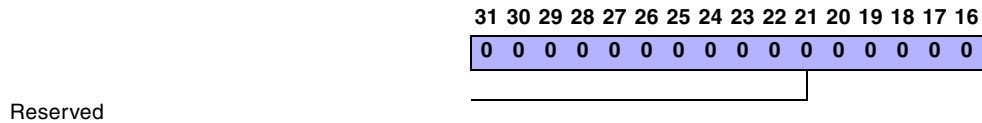


Figure 7-9. SFREG (Upper) Register Bit Descriptions

Listing 7-2. Conditional Sequencer Instructions

```
IF ALE, JUMP label;;  
/* conditional execution of the jump is based on ORing the ALE  
condition in compute blocks X and Y */  
  
IF XALE, JUMP label;;  
/* conditional execution of the jump is based on ALE condition in  
compute block X */  
  
IF AEQ, JUMP label; ELSE, XR0 = R5 + R6; YR8 = R9 - R10;;  
/* conditional execution of the jump is based on ORing the AEQ  
condition in compute blocks X and Y; the add in compute block X  
only executes if the jump does not execute; the subtract in com-  
pute block Y is unconditional (always executes) */
```

Besides the requirement that any sequencer instruction must use the first instruction slot (each instruction line contains four slots), it is important to note that the address used in a sequencer branch instruction (for example, `JUMP Address`) determines whether the instruction uses one slot or two. If a sequencer instruction specifies a relative or absolute address greater than 16 bits, the DSP automatically uses an immediate extension (the second instruction slot) to hold the extra address bits.

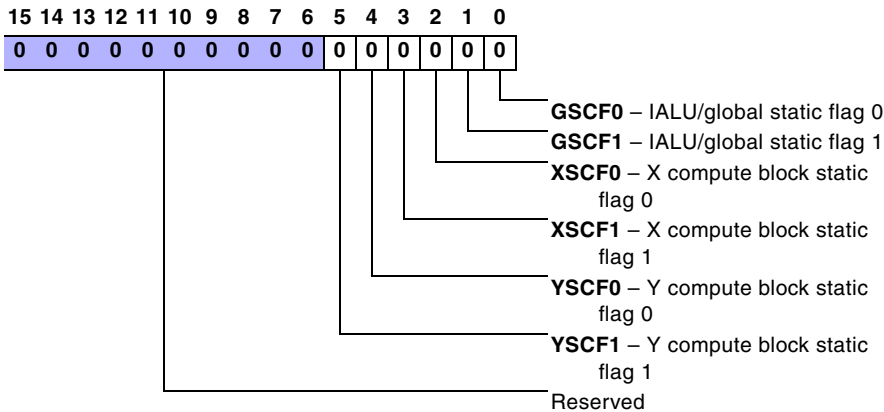


Figure 7-10. SFREG (Lower) Register Bit Descriptions


When a program *Label* is used instead of an address, the assembler converts the *Label* to a PC-relative address. When the *Label* is contained in the same program `.SECTION` as the branch instruction, the assembler uses a 16-bit address. When the *Label* is *not* contained in the same program `.SECTION` as the branch instruction, the assembler uses a 32-bit address. For more information on PC-relative and absolute addresses and branching, see [“Branching Execution” on page 7-16](#).

To provide conditional instruction support for static conditions, the sequencer has a static flag (SFREG) register (appears in [Figure 7-9](#) and [Figure 7-10](#)). The sequencer uses this register to store status flag values from the compute blocks and IALUs for later usage in conditional instructions. For examples using the SFREG conditions in the compute blocks and IALUs, see [“ALU Static Flags” on page 3-15](#), [“Multiplier Static Flags” on page 4-21](#), [“Shifter Static Flags” on page 5-17](#), [“ALU Static Flags” on page 3-15](#), and [“IALU Static Flags” on page 6-13](#).

A `CJMP_CALL` instruction transfers execution to a subroutine using a computed jump address (`CJMP` register). One way to load the computed jump address is to use the (`CJMP`) option on the IALU add/subtract instruction. The `CJMP_CALL` transfers execution to the address indicated by the `CJMP` register, then loads the return address into `CJMP`. The `CJMP` instruction at the end of the subroutine causes the sequencer to jump to the address in `CJMP`.

A `JUMP` instruction transfers execution to address *Label* (or an immediate 16- or 32-bit address).

Branching sequencer instructions use PC-relative or absolute addresses. For relative addressing, the program sequencer bases the address for relative branch on the 32-bit Program Counter (PC). For absolute addressing instructions, the branch instruction provides an immediate 32-bit address value. The PC allows linear addressing of the full 32-bit address range.

 If a 32-bit PC-relative or absolute address is used with a `CALL` or `JUMP` instruction (or if the *Label* is located outside the program `.SECTION` containing the `CALL` or `JUMP` instruction), the `CALL` or `JUMP` instruction requires the immediate extension instruction slot to hold the 16-bit address extension.

The default operation of `CALL` and `JUMP` instructions is to assume the address in the instruction is *PC-relative*. To use an *absolute address*, branch instructions use the absolute (`ABS`) option. The following example shows a PC-relative address branch instruction and an absolute address branch instruction.

```
JUMP fft_routine ;  
/* fft_routine is a program label that appears within this .sec-  
tion; the assembler converts the label into a 16-bit PC-relative  
address */
```


Sequencer Operations

```
CALL far_iir_routine (ABS) ;;  
/* far_iir_routine is a program label from outside this .section;  
the assembler converts the label into an absolute address  
(because the ABS option is used); also the label converts to a  
32-bit address (because it is outside this section) */
```

For more examples of branch instructions, see [“Sequencer Examples” on page 7-72](#).

Another default operation that applies to all conditional branch instructions is that the sequencer assumes the conditional test is `TRUE` and predicts the branch is taken—a *predicted branch*. When for programming reasons the programmer can predict that most of the time the branch is not taken, the branch is called not predicted and is indicated as such using the not predicted (`NP`) option. In the following example, the first conditional branch is a predicted branch and the second is a *not predicted branch*:

```
IF AEQ, JUMP fft_routine ;;  
/* this conditional branching instruction is a predicted branch  
*/  
IF MEQ CALL far_iir_routine (ABS) (NP) ;;  
/* this conditional branching instruction is a not predicted  
branch */
```

 Unconditional branches are treated as predicted branches by the sequencer. The sequencer treats unconditional branches as though they are prefixed with the condition `IF TRUE`.

Correctly predicting a branch as taken or not taken is extremely important for pipeline performance. The sequencer writes information about every predicted branch into BTB. The BTB examines the flow of addresses during the pipeline stage Fetch 1. When a BTB hit occurs (the BTB recognizes the address of an instruction that caused a jump on a previous pass of the program code), the BTB substitutes the corresponding destination address as the fetch address for the following instruction. When a

branch is currently cached and correctly predicted, the performance loss due to branching is reduced from either six or three stall cycles to zero. For more information, see “[Branch Target Buffer \(BTB\)](#)” on page 7-34.

Looping Execution

The sequencer supports zero-overhead and near-zero-overhead looping execution. As shown in [Figure 7-3 on page 7-4](#), a loop lets a program execute a group of instructions repetitively, until a particular condition is met. When the condition is met, execution continues with the next sequential instruction after the loop.

To set up a loop, a program uses a loop counter register, an instruction to decrement the counter, and a conditional instruction jump instruction that tests whether the condition is met and (if not met) jumps to the beginning of the loop.


For zero-overhead loops (no lost cycles for loop test and counter decrement), the sequencer has two dedicated loop counter (LC0 and LC1) registers and special loop counter conditions (counter not expired: IF NLCOE and IF NLC1E, and counter expired: IF LCOE and IF LC1E) for the conditional jump at the end of the loop. Also, the sequencer automatically decrements the counter when executing the special loop counter test and conditional jump. For an example, see [Listing 7-3](#).

Listing 7-3. Zero-Overhead Loop Example

```
LC0 = N ;; /* N = 10, sets up loop counter */
_start_loop:
  NOP ;; /* any instruction */
  NOP ;; /* any instruction */
  NOP ;; /* any instruction */
  IF NLCOE, jump _start_loop ;; /* condition test at loop end */
```

Sequencer Operations

Beside the two zero-overhead loops, the sequencer supports any number of near-zero-overhead loops. A near-zero-overhead loop uses an IALU register for the loop counter, includes an instruction to decrement the counter, and uses a condition to test the decrement operation in the conditional jump at the end of the loop. For an example containing zero-overhead and near-zero-overhead loops, see [Listing 7-8 on page 7-73](#).

 For near-zero-overhead loops, programs get better performance through using an IALU register (rather than a compute block register) for the counter. The reason for this performance difference stems from the difference between compute block and IALU instruction execution in the instruction pipeline. For more information, see [“Instruction Pipeline Operations” on page 7-26](#).

Interrupting Execution

Interrupts are events that cause the core to pause its current process, branch, and execute another process. These events can occur at any time and are:

- Internal to the processor
- External to the processor

Interrupts are intended for:

- Synchronizing core and non-core operation
- Error detection
- Debug features
- Control by applications

Each interrupt has a vector register in the Interrupt Vector Table (IVT) and an assigned bit in the interrupt flags and masks registers (ILAT, IMASK and PMASK). The vector register contains the user-definable address of the interrupt routine that services the interrupt. Some important points about the interrupt vector table include:

- 31 interrupts
- Most interrupts are dedicated
- Four general-purpose interrupts associated with $\overline{IRQ3-0}$ pins



In the IMASK register, a “1” means the interrupt is *unmasked* (DSP recognizes and services interrupt). A “0” means the interrupt is *masked* (DSP does not recognize interrupt). The IMASK, ILAT and PMASK registers all have the same bit definitions

Interrupts are classified as either edge or level sensitive. Edge sensitive interrupts are latched when they occur and remain latched until serviced or reset by an instruction. Level sensitive interrupts differ in that if the interrupt is not serviced before the request is removed, the interrupt is forgotten, and if the request remains after the service routine executes, it is considered a new interrupt.

The sequencer manages interrupts using the ILAT, IMASK, and PMASK control registers. These registers appear in [Figure 7-12](#), [Figure 7-13](#), [Figure 7-14](#), and [Figure 7-15](#) and have the following usage:

- ILAT – Interrupt Latch Register, latches edge sensitive interrupts (interrupt’s bit =1) and displays active level sensitive interrupts (interrupt’s =1)
- IMASK – Interrupt Mask Register, masks each individually (interrupt’s bit =0) or all interrupts (GIE bit =0)
- PMASK – Interrupt Mask Pointer Register, indicates each interrupt being serviced (interrupt’s bit =1)

Sequencer Operations

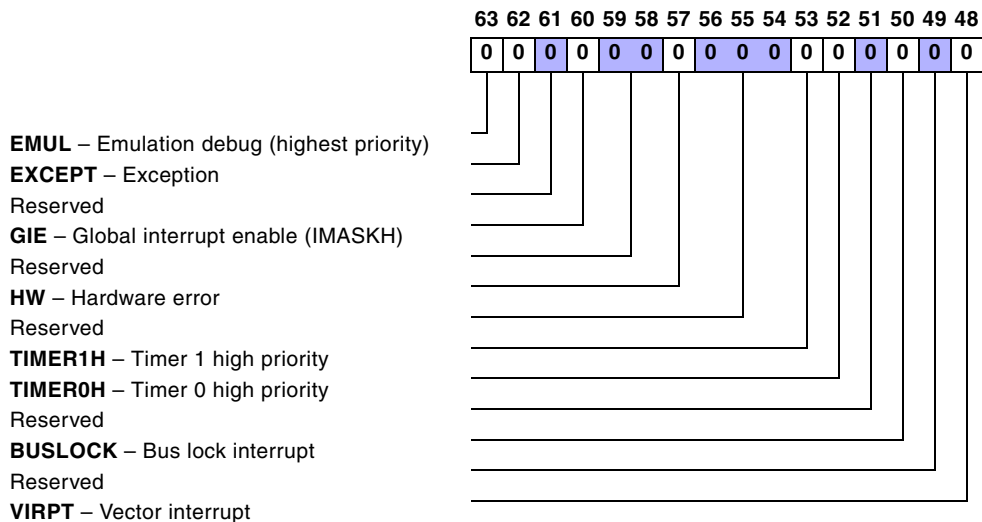


Figure 7-12. IMASKH, ILATH, PMASKH (Upper) Register Bit Descriptions

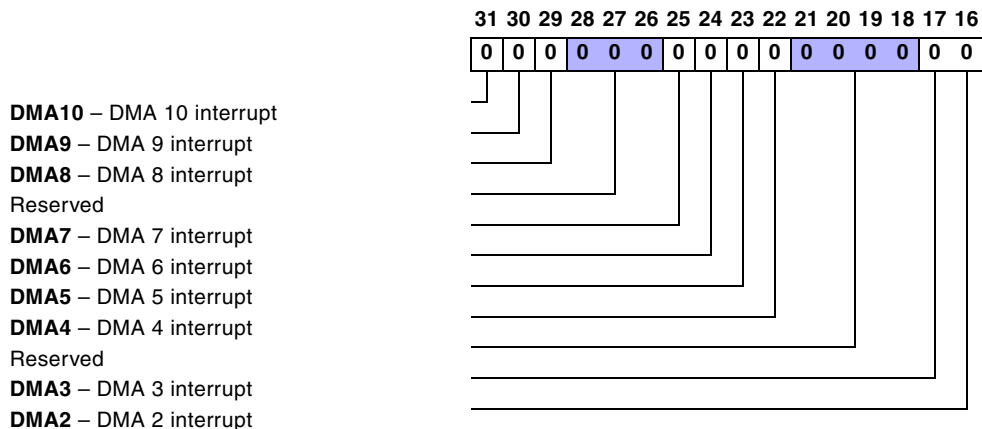


Figure 7-13. IMASKL, ILATL, PMASKL (Upper) Register Bit Descriptions

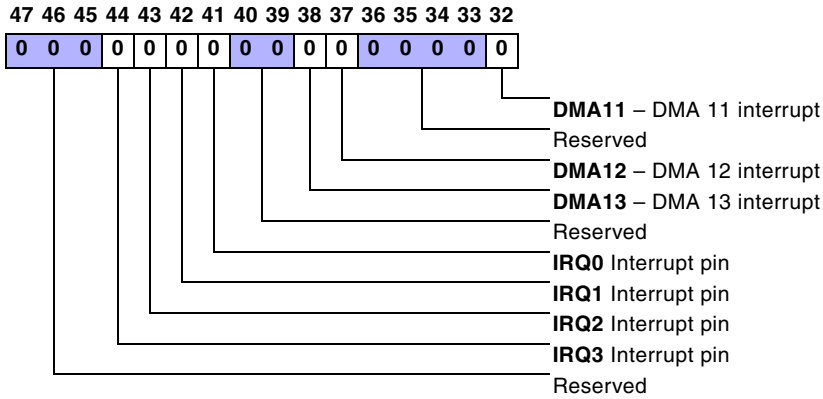


Figure 7-14. IMASKH, ILATH, PMASKH (Lower) Register Bit Descriptions

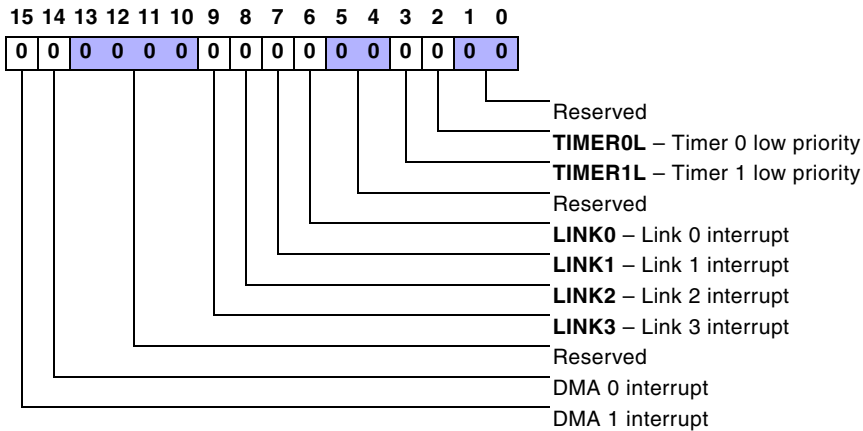


Figure 7-15. IMASKL, ILATL, PMASKL (Lower) Register Bit Descriptions

Sequencer Operations

The priority of interrupts high to low matches their order of appearance in `IMASK`, `ILAT`, and `PMASK`. For more information on interrupt sensitivity, interrupt latching, the Interrupt Vector Table, and other interrupt issues, see the *ADSP-TS101 TigerSHARC Processor Hardware Reference*.

The sequencer supports interrupting execution through hardware interrupts (external `TRQ3-0` pins and internal process conditions) and software interrupts (program sets an interrupt's latch bit). [Figure 7-3 on page 7-4](#) provides a high-level comparison of branching variations.

Looking at [Figure 7-16](#), note some additional details about interrupt service routines for non-reusable interrupts and reusable interrupts. The difference between these two types of interrupt service is that a non-reusable interrupt service routine cannot re-latch the same interrupt again until it has been serviced, and a reusable interrupt service routine (because it has been reduced to subroutine status with the `RDS` instruction) can re-latch the same interrupt while it is still being serviced. To understand the difference better, note the steps for interrupt processing of these two types of interrupt service.

For non-reusable interrupt service, the steps are:

1. Sequencer recognizes interrupt when the interrupt's bit is latched in the `ILAT` register, the interrupt is unmasked by the interrupt's bit in the `IMASK` and `PMASK` registers, and interrupts are globally enabled by the `GIE` bit in the `IMASK` register.
2. Sequencer places the DSP in supervisor mode, jumps execution to the interrupt's vector address set in the interrupt's vector register, loads the return address (next sequential address after the interrupt) into the `RETI` register, and sets the interrupt's `PMASK` bits.
3. DSP executes interrupt service routine.
4. Sequencer jumps execution to the return address on reaching the `RTI` instruction at the end of the interrupt service routine, clears the interrupt's `ILAT` bit, and clears the interrupt's `PMASK` bits.

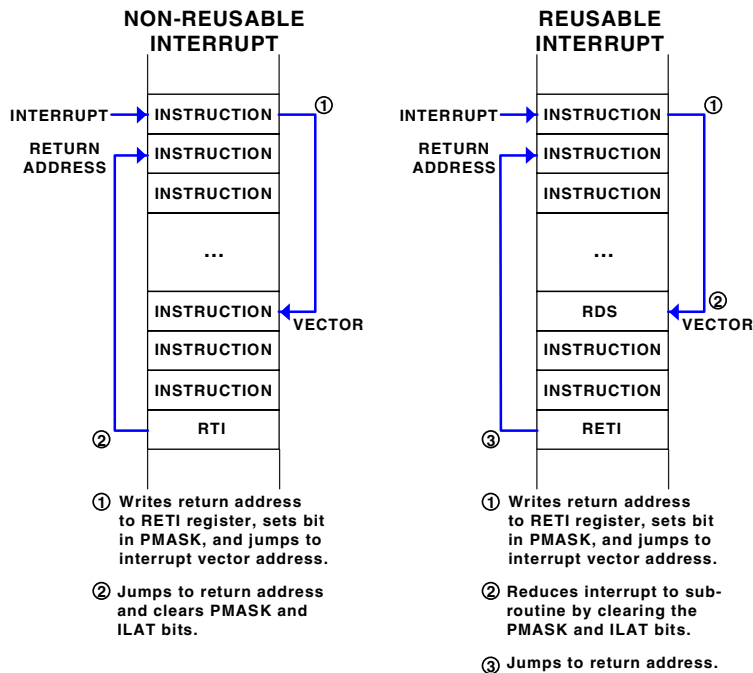


Figure 7-16. Non-Reusable Versus Reusable Interrupt Service

For reusable interrupt service, the steps are:

1. Sequencer recognizes interrupt when the interrupt's bit is latched in the ILAT register, the interrupt is unmasked by the interrupt's bit in the IMASK and PMASK registers, and interrupts are globally enabled by the GIE bit in the IMASK register.
2. Sequencer places the DSP in supervisor mode, jumps execution to the interrupt's vector address set in the interrupt's vector register, loads the return address (next sequential address after the interrupt) into the RETI register, and sets the interrupt's PMASK bits.

Instruction Pipeline Operations

3. Sequencer reduces the interrupt to a subroutine on reaching the RDS instruction at the beginning of the interrupt service routine, clears the interrupt's ILAT bit, and clears the interrupt's PMASK bits.

Because the interrupt's ILAT and PMASK bits are cleared, the interrupt may be latched again before the interrupt service is completed.

4. DSP executes interrupt service routine.
5. Sequencer jumps execution to the return address on reaching the RETI instruction at the end of the interrupt service routine.



Depending on the instruction that is being executed when the interrupt is recognized, different pipeline effects may occur. For more information, see [“Instruction Pipeline Operations” on page 7-26](#).

Instruction Pipeline Operations

As introduced in the instruction pipeline discussion [on page 7-4](#), the TigerSHARC processor has an eight-stage instruction pipeline. The pipeline stages and their relationship to the branch target buffer and instruction alignment buffer appears in [Figure 7-4 on page 7-5](#). To better understand the flow of instructions through the pipeline and the time required for each stage, this section uses a series of diagrams similar to [Figure 7-17](#).

Looking at [Figure 7-17](#), note that the instruction pipeline stages appear on the vertical axis and CCLK (DSP core clock) cycles appear on the horizontal axis. Usually, each pipeline stage requires one cycle to process an instruction line of up to four instructions. This section describes the situations in which a pipeline stage may require more time to complete its operation.

#1: $XR0 = R1 + R2$;; /* X compute block */
 #2: $YR3 = R4 * R5$;; /* Y compute block */
 #3: $J0 = J1 + J2$;; /* J-IALU */
 #4: $K0 = K1 - K2$;; /* K-IALU */

Because IALU instructions execute at I and compute block instructions execute at EX2, results are available earlier for IALU instructions.

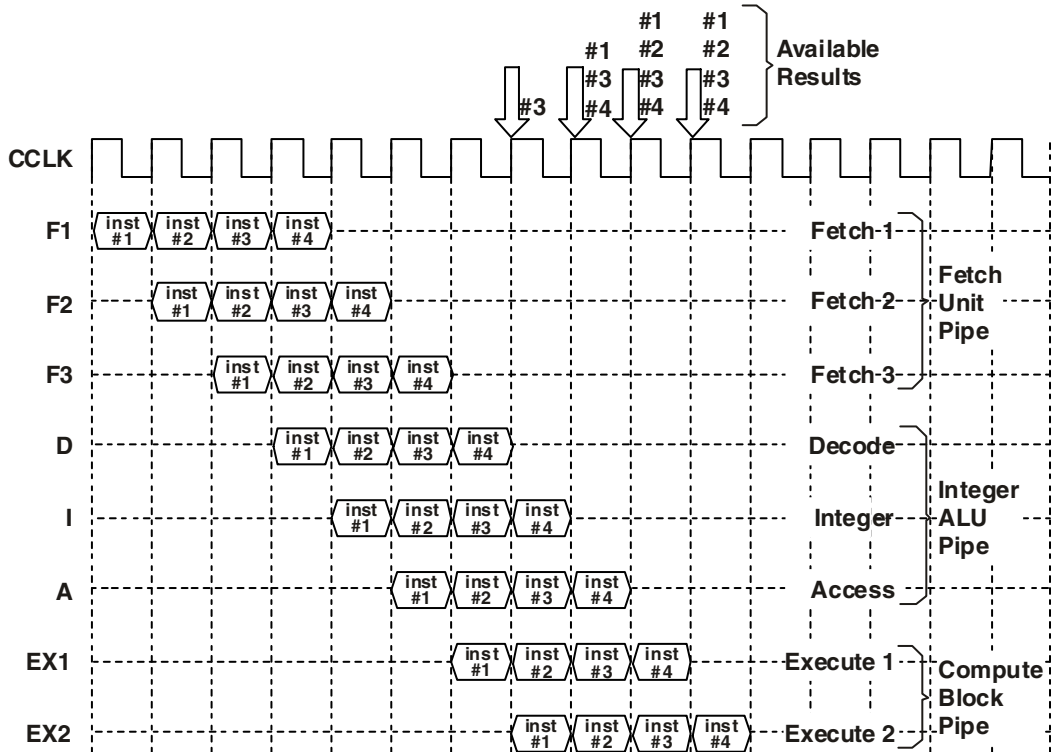


Figure 7-17. Timing for Stages of Instruction Pipeline

Also note from [Figure 7-17](#) the order in which results from an operation become available. Because IALU arithmetic operations execute at the integer stage and compute block operations execute at the execute 2 stage, IALU results may be available before compute block results, depending on the order and type of instructions.

Instruction Pipeline Operations

The first three stages of the instruction pipeline (F1, F2, and F3) are called the *fetch unit pipe*. The fetch cycles are the first pipeline and are tied to the memory accesses. The progress in this pipeline is memory driven and not instruction driven. The fetch unit fills up the instruction alignment buffer (IAB) whenever the IAB has less than three quad words. Since the execution units can pull in instructions in throughput lower or equal to the fetch throughput of four words every cycle, it is probable that the fetch unit will fill the IAB faster than the execution units pull the instructions out of the IAB. The IAB can be filled with up to five quad words of instructions.

When the fetch is from external memory the flow is similar although much slower. The maximum fetch throughput is one instruction line to every two SCLK cycles and the latency is according to the system design (external memory pipeline depth, wait cycles).

The next three instruction pipeline stages (D, I, and A) are called the *integer ALU pipe*. The Decode stage is the first stage in this pipe. In this cycle the next full instruction line is extracted from the instruction alignment buffer and the different instructions are distributed to the execution units. The units include:

- J-IALU or K-IALU – integer instructions, load/store and register transfers
- Compute block X or Y or both – two instructions (the switching within the compute block is done by the register file)
- Sequencer – branch and condition instructions, and others

The instruction alignment buffer (IAB) also calculates the program counter of a sequential line and some of the non-sequential instructions. The IAB does not perform any decoding.

After the Decode stage, instructions enter the integer stage of the integer ALU pipe. IALU instructions include address or data calculation and, optionally, memory access. [Figure 7-17 on page 7-27](#) shows the instruc-

tion flow in the IALU pipeline. The IALU instruction's calculation is executed at the Integer stage. If the IALU instruction includes a memory access, the bus is requested on the Integer stage. In this case, the memory access begins at the Access stage as long as the bus is available for the IALU.

The result of the address calculation is ready at the Integer stage. Since the execution of the IALU instruction may be aborted (either because of a condition or branch prediction), the operand is returned to the destination register only at the end of EX2. The result is passed through the pipeline, where it may be extracted by a new instruction should it be required as a source operand. Dependency between IALU calculations normally do not cause any delay, but there are some exceptions. The data that is loaded, however, is only ready in the register at pipe stage EX2.

The final pipeline stages (EX1 and EX2) are called the *compute block pipe*. The compute block pipe is relatively simple. At the decode cycle, the compute block gets the instruction and transfers it to the execution unit (ALU, multiplier or shifter). At the Integer stage, the instruction is decoded in the execution unit (ALU, multiplier or shifter), and dependency is checked. At the Access stage, the source registers are selected in the register file. At the execution stages EX1 and EX2, the result and flag updates are calculated by the appropriate compute block. The execution is always two cycles, and the result is written into the target register on the rising edge after pipe stage EX2. See [Figure 7-17 on page 7-27](#).

All results are written into the target registers and status flags at pipe stage EX2. There are two exceptions to this rule:

- External memory access, in which the delay is determined by the system
- Multiply-accumulate instructions, which write into MR registers and sticky flags one cycle after EX2 (This write is important to retain coherency in case of a pipeline break.)


Instruction Pipeline Operations

When executed either at the Integer stage (IALU arithmetic) or execute 2 stage (all other instructions), the instructions in a single line are executed in parallel. When there are two instructions in the same line which use the same register (one as operand and the other as result), the operand is determined as the value of the register *prior* to the execution of this line. For example:

```
/* Initial values are: R0 = 2, R1 = 3, R2 = 3, R3 = 8 */  
R2 = R0 + R1 ; R6 = R2 * R3 (I) ; ;
```

In the previous example, R2 is modified by the first instruction, and the result is 5. Still the second instruction sees input to R2 as 3, and the result written to R6 is 24. This rule is not guaranteed for memory store instructions. In this next example with the same initial values, the result of the first slot is used in the second slot with unpredictable results.

```
R2 = R0 + R1 ; [Address] = R2 ; ;  
/* The results of using R2 as input for the memory store instruction here are unpredictable due to possible memory access stalls. The assembler flags this instruction as illegal. */
```

 For best results, do not use the results from one instruction as an operand for another instruction within the same instruction line.

The pipeline creates complications because of the overlap between the execution time of instructions of different lines. For example, take a sequence of two instruction lines where the second instruction line uses the result of the first instruction line as an input operand. Because of the pipeline length, the result may not be ready when the second instruction fetches its operands. In such a case, a *stall* is issued between the first and second instruction line. Since this may cause performance loss, the programmer or compiler should strive to create as few of these cases as possible. These combinations are legal however, and the result will be correct. This type of problem is discussed in detail in [“Dependency and Resource Effects on Pipeline” on page 7-55](#).

Instruction Alignment Buffer (IAB)

The IAB is a five quad-word FIFO as shown in [Figure 7-18](#). When the sequencer fetches an instruction quad word from memory, the quad word is written into the next entry in the IAB. If there is at least one full instruction line in the IAB, the sequencer can pull it for execution.

The IAB provides these services:

- Buffer the input (fetched instructions) from the fetch unit pipe, keeping the fetch unit independent from the rest of the instruction pipeline. This independence lets the fetch unit continue to run even when other parts of the pipeline stall.
- Align the input (unaligned quad words) to prepare complete instruction lines for execution. This alignment guarantees that complete instruction lines with one to four instructions are able to execute in parallel.

Instructions are 32-bit words that are stored in memory without regard to quad-word alignment (128-bit) or instruction line length (1–4 instructions). There is no wasted memory space. Instructions are executed in parallel as determined by the MSB of the opcode for each instruction.

- Distribute the instruction lines to the execution units—IALUs, compute blocks, and sequencer.

Instruction Pipeline Operations

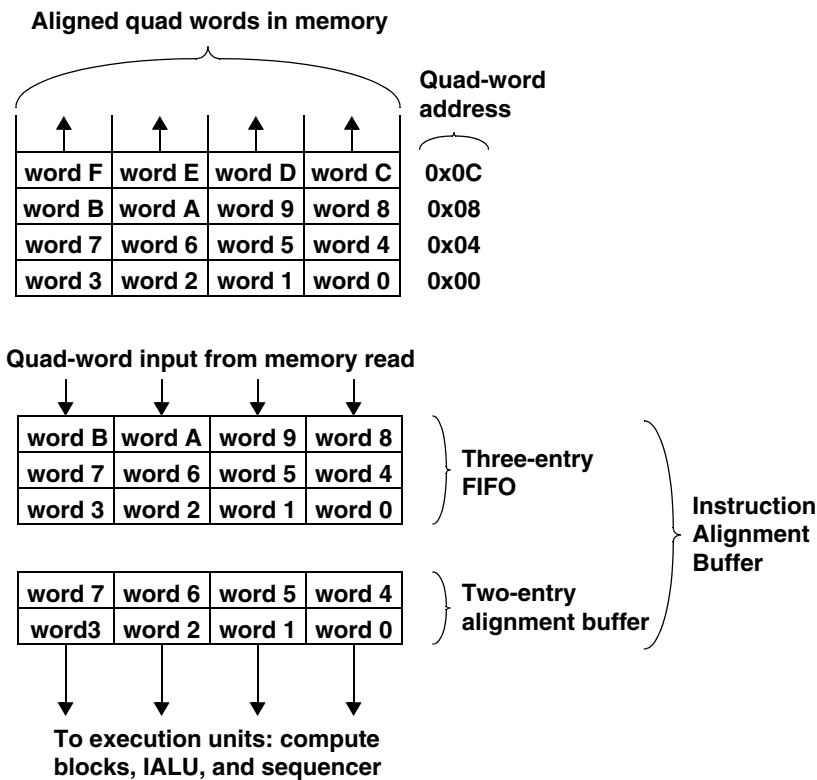


Figure 7-18. Instruction Alignment Buffer (IAB) Structure

Through these services, the IAB insures execution of an entire instruction line without inserting additional stall cycles or forcing memory quad-word alignment on instruction lines.

To understand the value of the buffer, align, and distribute service that the IAB provides, see [Figure 7-19](#). This figure provides an example of how normal (32-bit) instruction words are stored unaligned in memory and how the IAB buffers, aligns, and distributes these words.

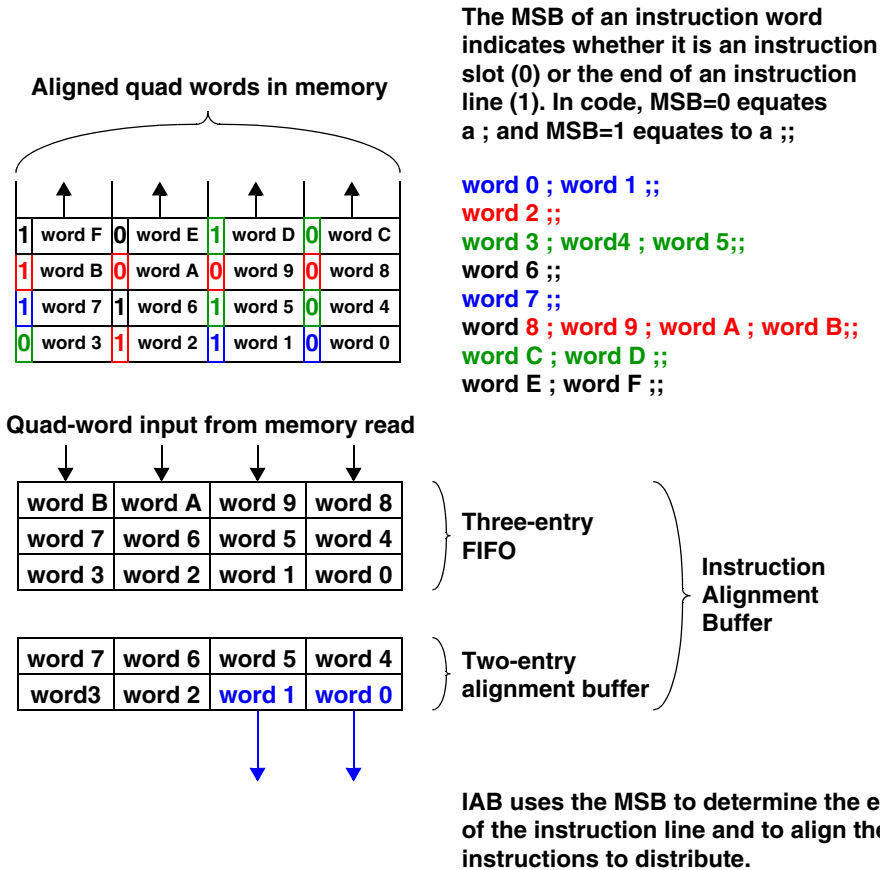


Figure 7-19. IAB Aligns Instruction Lines for Distribution/Execution

i These descriptions of IAB operation apply to internal memory fetches only. Instruction fetches from external memory result in a

Instruction Pipeline Operations

much slower instruction flow. Fetch throughput is one instruction for every SCLK cycle—at best, 25% of the rate for internal memory fetches. Latency depends on system design (external memory pipeline depth, wait cycles, and other issues).

Branch Target Buffer (BTB)

The sequencer uses the branch target buffer (BTB) to reduce or eliminate the branch costs (lost cycles as pipeline accommodates the branch) that result from branching execution in the instruction pipeline. The BTB has 32 entries of 4-way set-associative cache (total of 128 entries) that store branch target addresses and has a Least Recently Used (LRU) replacement policy.

The BTB is active while the BTB enable (*BTBEN*) bit in sequencer control (*SQCTL*) register is set. For more information, see the *SQCTL* register in [Figure 7-7 on page 7-11](#).

For permanent buffered program sections, program must lock the BTB using the *BTBLK* bit in *SQCTL*. While the *BTBLK* bit is set, the BTB puts every new entry into the BTB in locked status. When this happens the BTB entry is not replaced until the whole BTB is flushed, in order to keep performance-critical jumps in BTB.

Whenever program overlays are used to swap program segments into and out of internal memory, the BTB must be cleared using instruction *BTBINV* in order to invalidate the BTB.



The BTB contents can be accessed directly for debug and diagnostic purposes only, but it must be disabled prior to access by clearing the *BTBEN* bit in the *SQCTL* register. Do not directly access BTB contents during normal operations because this access may result in multi-hit and coherency problems.

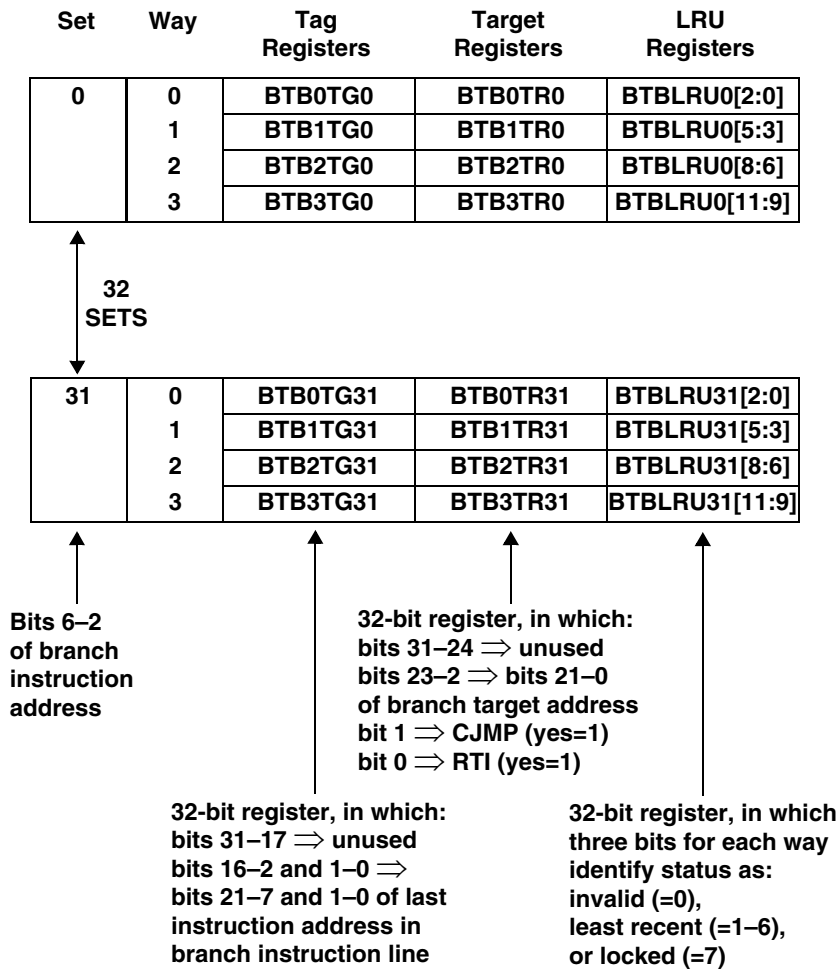


Figure 7-20. Branch Target Buffer (BTB) Structure

The BTB structure appears in [Figure 7-20](#). This structure consists of three types of registers that make up each of the 128 BTB entries. The entries are divided into 32 sets and within each set there are four ways. The parts of the BTB include set, way, tag, target, and LRU.


Instruction Pipeline Operations

Index	The index determines the set for the BTB entry. The index is bits 6–2 of the branch instruction’s address. Using five bits provides 32 <i>sets</i> . Each set contains four <i>ways</i> (branch instruction addresses with the same 6–2 bits).
Tag	The tag determines whether a fetched instruction is a BTB hit (address is in BTB) or a BTB miss (address is not in BTB) when comparing the fetched instruction with contents of ways in the appropriate set. Tags are recorded in the $BTBiTG_{xx}$ registers where i represents the way and xx represents the set. Each register is 32-bits wide, but only the 17 LSBs are valid. These 17 bits are comprised of bits 21–7 and 1–0 of the address of the last instruction in the branch instruction’s instruction line.
Target	The target is the target address of the branch. The PC begins fetching at the target address on a BTB hit. Targets are recorded in the $BTBiTR_{xx}$ registers where i represents the way and xx represents the set. Each register is 32-bits wide, but only the 24 LSBs are valid. These 24 bits are comprised of bits 21–0 of the address of the instruction that is the target of the branch, a $CJMP$ bit (0 = branch is not $CJMP$, 1 = branch is $CJMP$), and an RTI bit (0 = branch is not RTI , 1 = branch is RTI),
LRU	The LRU is the Least Recently Used field, which determines whether a way is locked or valid and whether a way should be overwritten in the event of a BTB miss. LRU bits are recorded in the $BTBLRU_{xx}$ registers where xx represents the set. Each register is 32-bits wide, but only the 12 LSBs are valid. Each

set has an LRU register that contains a three-bit field for the LRU value for each of the ways in the set. The LRU values represent: 0 = invalid, 1 through 6 = regular LRU value with 6 being most recently used, and 7 = locked.

Now that the controls for the BTB (bits in the `SQCTL` register) and the structure of the BTB (set, way, tag, target, LRU) are understood, it is important to look at how the BTB and branch prediction relate to the instruction pipeline. A flow chart of the sequencer's BTB and branch prediction operations with related branch costs (penalty cycles) appears in [Figure 7-21](#).

As shown in [Figure 7-21](#), BTB usage and branch prediction are independent. The sequencer always compares the fetched instruction against the BTB contents when the instruction is at pipeline Fetch 1 stage. If there is a BTB miss, the branch prediction tells the sequencer at later stages what to assume and when to make its decision. At pipeline Decode stage, the sequencer knows that the instruction is a branch and whether it is a predicted branch or a not predicted (NP) branch. This stage is where the pipeline makes the decision on where to fetch from next. This prediction also affects whether or not the branch is entered into the BTB, so that a hit occurs for the next iteration.

-  By default, all conditional branches are predicted branches (predicted as taken) and are unconditional branches (treated as though prefixed with the condition `IF TRUE`). Only conditional branches with the (NP) (not predicted) option are predicted a not taken.

Instruction Pipeline Operations

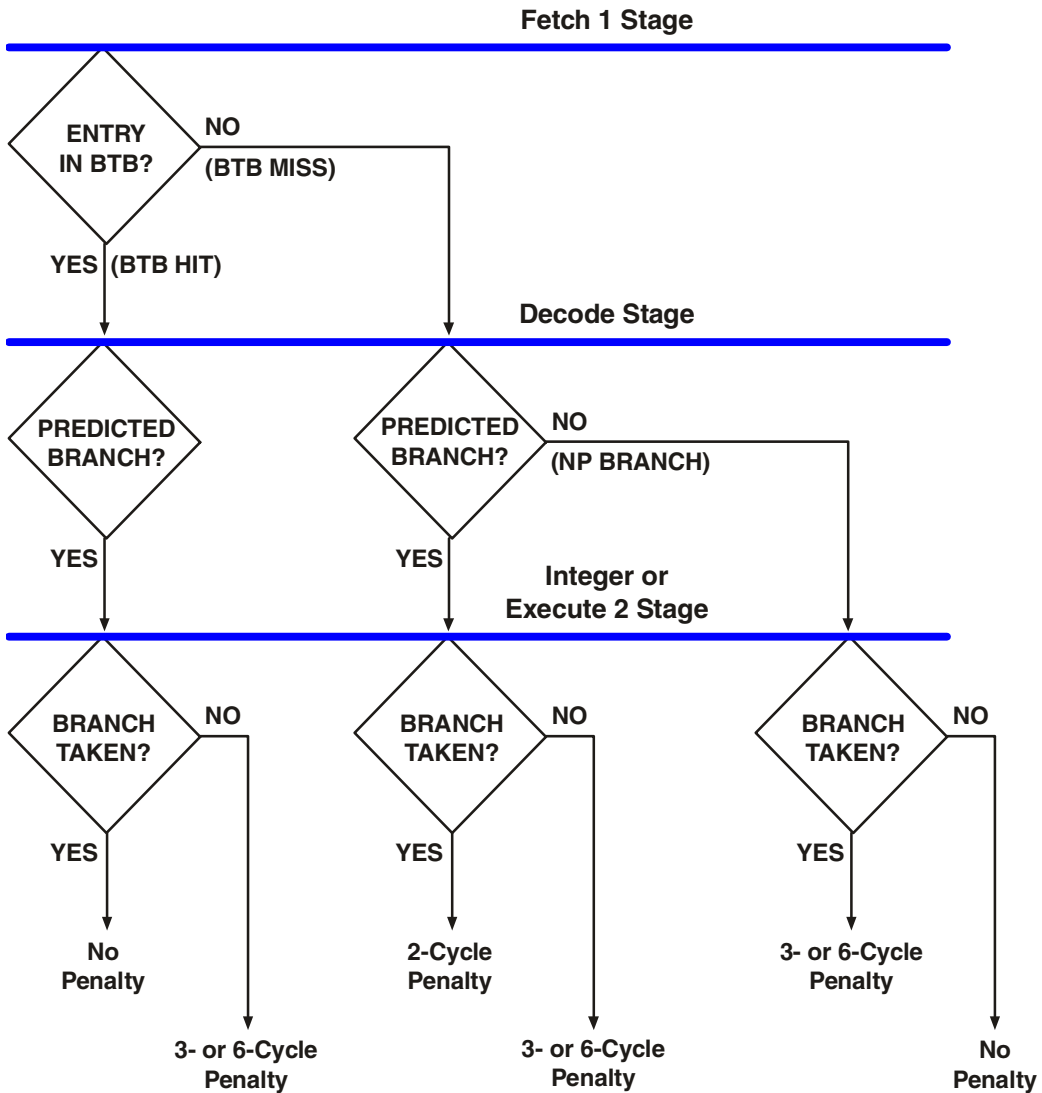



Figure 7-21. Branch Prediction Penalty Tree¹

¹ When the fetched instruction line crosses a quad-word boundary, add one penalty cycle in all cases.

The penalty cycles (stalls) for incorrectly predicted branches include:

- When a branch is predicted and there is a BTB hit, the sequencer assumes that the branch is taken and begins fetching from the branch target address when the branch goes from pipeline stage Fetch 1 to Fetch 2. If predicted correctly, this case has zero penalty cycles.
 - When a branch is predicted but there is a BTB miss (BTB disabled or no entry match), the sequencer assumes that the branch is taken and begins fetching from the branch target when the branch goes from pipeline stage Fetch 3 to Decode. If predicted correctly, this case has two penalty cycles.
 - When a conditional branch is not predicted (NP) and there is a BTB miss (this is always the case with (NP) branches), the sequencer assumes that the branch is not taken and does not begin fetching from the branch target address until the condition is evaluated. This evaluation occurs when the branch goes from pipeline stage Decode to Integer for IALU conditions or from pipeline stage Execute 1 to Execute 2 for compute conditions. If predicted correctly, this case has zero penalty cycles.
 - When a conditional branch is not correctly predicted (a predicted branch is not taken or a not predicted (NP) is taken), the sequencer does not catch the incorrect prediction until the condition is evaluated. This evaluation occurs when the branch goes from pipeline stage Decode to Integer for IALU conditions or from pipeline stage Execute 1 to Execute 2 for compute conditions. This case has three (for IALU condition) or six (for compute condition) penalty cycles.
-  When the fetched instruction line crosses a quad-word boundary, add one penalty cycle in all cases. Using the `.align_code 4` assembler directive to quad-word align *labels* for JUMP and CALL instructions eliminates this penalty cycle for these branch instructions.

Instruction Pipeline Operations

Even assuming that the BTB were disabled (BTBEN bit =0) which would always cause a BTB miss, branch prediction still has an effect on the instruction pipeline's decision making (see all the BTB miss cases above).

Besides understanding the limit of the BTB's affect on the pipeline, it is important to understand the limits of BTB usage regarding branch instruction placement in memory.

Only internal memory branches are cached in the BTB. The width of the cached target addresses is 22 bits. The BTB stores only one tag entry per aligned quad word of program instructions and, consequently, only one branch may be predicted per aligned quad word. If a programmer requires more than one adjacent branch be predicted, then one to three NOP instructions must be inserted between the branches to insure that both branches do not fall into the same aligned quad word.

To avoid the possibility of placing more than one instruction containing a predicted branch within the same quad-word boundary in memory and causing unexpected BTB function, this combination of instructions and placement causes an assembler warning. The assembler warns that it has detected two predicted jumps within instruction lines whose line endings are within four words of each other. Further, the assembler states that depending on section alignment, this combination of predicted branch instructions and the instructions placement in memory may violate the constraint that they cannot end in the same quad word.

It's useful to examine how different placement of words in memory results in different contents in the BTB. For example, the code example in [Listing 7-4](#) contains a predicted branch.

Listing 7-4. Predicted Branches, Aligned Quad Words, and the BTB

```
nop; nop; nop; nop;;  
jump HERE; nop;;  
nop; nop; nop; nop;;
```

In memory, each instruction occupies an address, and sets of four locations make up a quad word. The placement of quad words in memory is shown in [Figure 7-18 on page 7-32](#) and discussed in “[Instruction Alignment Buffer \(IAB\)](#)” on page 7-31. The quad-word address is the address of the first instruction in the quad word.

Depending on how the code in [Listing 7-4](#) aligns in memory, quad-word address `0x00000004` could contain:

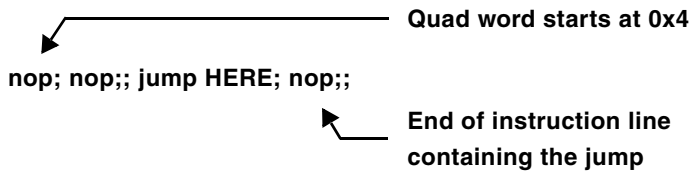


Figure 7-22. Quad Words and Jump Instructions (1)

If so, the BTB entry for the branch would contain:

Tag = `0x00000004`, Target Address = `HERE`

But, the code in [Listing 7-4](#) could align in memory differently. For example, this code could align such that quad-word address `0x00000004` (first line) and `0x00000008` (second line) contain:

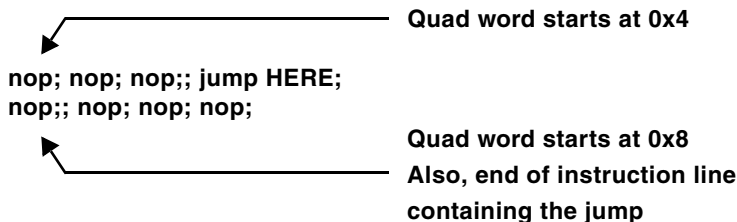


Figure 7-23. Quad Words and Jump Instructions (2)

Instruction Pipeline Operations

If so, the BTB entry for the branch would contain:

Tag = 0x00000008, Target Address = HERE

If prediction is enabled, the current PC is compared to the BTB tag values at the F1 stage of the pipeline. If there is a match, the DSP modifies the PC to reflect the branch target address stored in the BTB, and the sequencer continues to fetch subsequent quad words at the modified PC. If there is no match, the DSP does not modify the PC, and the sequencer continues to fetch subsequent quad words at the unmodified PC.

When the same instruction reaches the Decode stage of the pipeline, the instruction is identified as a branch instruction. If there was a BTB match, no branch-exception action is taken. The PC has already been modified, and the sequencer has already fetched from the branch target address. If there is no BTB match, the sequencer aborts the two instructions fetched prior to reaching the Decode stage (two stall cycles), and the DSP modifies the PC to reflect the branch target address and begins fetching quad words at the modified PC. The sequencer updates the BTB with the branch target address such that the next time the branch instruction is encountered, it is likely that there will be a BTB match.

The BTB contents vary with the instruction placement in memory, because:

- The sequencer fetches instructions a full quad word at a time.
- An instruction line may occupy less than a full quad word, occupy a full quad word, or span two quad words.
- An instruction line may start at a location other than a quad-word aligned address.

Because the BTB can store only a single branch target address for each aligned quad word of instruction code, its important to examine coding techniques that work with this BTB feature. The following code example

produces unpredictable results in the hardware, because this code (depending on memory placement) may attempt to force the BTB to store multiple branch target addresses from a single aligned quad word.

```

jump FIRST_JUMP; LC1 = yR16;;
jump SECOND_JUMP; R29 = R27;;

```

Illegal, the line ends of instruction lines that contain JUMPs are within four instructions of each other.

Figure 7-24. Quad Words and Jump Instructions (3) Illegal Proximity

The situation can be remedied by using NOP instructions to force the branch instructions to exhibit at least four words of separation.

```

jump FIRST_JUMP; LC1 = yR16;;
jump SECOND_JUMP; R29 = R27; nop; nop;;
/* Adding NOPs moves the line ending of 2nd instruction */

```

While adding these NOP instructions increases the size of the code, these NOP instructions do not affect the performance of the code.

Another way to control the relationship between alignment of code within quad words and BTB contents is to use the `.align_code 4` assembler directive. This directive forces the immediately subsequent code to be quad-word aligned as follows:

```

jump FIRST_JUMP; LC1 = yR16;;
.align_code 4;
/* Forcing quad alignment shifts the line ending of the next
instruction */
jump SECOND_JUMP; R29 = R27;;

```

Instruction Pipeline Operations

If the BTB hit is a computed jump, the `RETI` or `CJMP` register is used (according to the instruction) as a target address. In this case, any change in this register's value until the jump takes place will cause the TigerSHARC processor to abort the fetched instructions and repeat the flow as if there were no hit.

Conditional Branch Effects on Pipeline

Correct prediction of conditional branches is critical to pipeline performance. Prediction affects, and is affected by, all the pipes (fetch, IALU, compute) in the pipeline. Each branch flow differs from every other and is derived by the following criteria:

- Jump prediction (See [“Conditional Execution” on page 7-12.](#))
- BTB hit or miss (See [“Branch Target Buffer \(BTB\)” on page 7-34.](#))
- Condition pipe stage—pipeline stage Integer or Execute 2—when is it resolved (See the Branch Prediction Penalty Tree in [Figure 7-21 on page 7-38.](#))

The prediction is set by the programmer or compiler. The prediction is normally `TRUE` or ‘branch taken’. When the programmer uses option `(NP)` in a control flow instruction, the prediction is ‘branch not taken’. [For more information, see “Branch Target Buffer \(BTB\)” on page 7-34.](#) In general, prediction indicates if the default assumption for this branch will or will not be taken. Take, for example, a loop that is executed n times, where the branch is taken $n-1$ times, and always more than once. Setting this bit has two consequences:

- The branch goes into BTB.
- At stage Decode, the TigerSHARC processor identifies the instruction as a jump instruction and continues fetching from the target of the jump, regardless of the condition.

If a branch instruction is a BTB hit, the TigerSHARC processor fetches, in sequence, the target of the branch after fetching the branch. In this case there is no overhead for correct prediction. For a detailed description of BTB behavior see [“Branch Target Buffer \(BTB\)” on page 7-34](#).

The various condition codes are resolved in different stages. IALU conditions are resolved in stage Integer of the instruction that updates the condition flags. Compute block flags are updated in pipe stage EX2. The other flags (`BM`, `FLGO-3`, and `TRUE`) are asynchronous to the pipeline because they are created by external events. These are used in the same fashion as IALU conditions and are resolved at pipe stage Integer, except for the condition `BM`, which is resolved at pipe stage EX2.

Different situations produce different flows and, as a result, different performance results. The parameters for the branch cost are:

- Prediction – branch is taken or not taken
- Branch on IALU or compute block
- BTB hit – miss
- Branch real result – taken or not


There are 16 combinations. The following combinations are ignored.

- If the prediction is ‘not taken’, the BTB cannot give a hit.
- If the prediction is ‘not taken’ and the branch is not taken, the flow is as if no branch exists.
- If the prediction is ‘taken’ and the branch is taken, the flow is identical for IALU and compute block instructions.

Instruction Pipeline Operations

The different flows are shown in [Figure 7-25 on page 7-47](#) through [Figure 7-36 on page 7-65](#). Each diagram shows the flow of each combination and its cost. The cost of a branch can be summarized as:

- Prediction not taken, branch not taken – no cost
- BTB hit, branch taken – no cost
- BTB miss, prediction taken, branch taken – two cycles
- Prediction taken, branch not taken (either BTB hit or miss); or prediction not taken, branch taken: IALU condition (three cycles) or Compute block (six cycles)

 If the prediction is ‘not taken’, there cannot be a BTB hit since the ‘prediction taken’ is a condition for adding an entry to BTB.

One cycle should be added to the above branch costs if one of the following applies:

- The jump is taken and the target instruction line crosses a quad-word boundary.
- The branch was predicted to be taken and was not, and the sequential instruction line crosses a quad-word boundary.

Figure 7-25 shows a predicted branch that is based on an IALU condition. Because the branch is correctly predicted as taken and the BTB contains the branch instruction (BTB hit), the pipeline contents are continuously executable (no pipeline stages voided), and there are no lost cycles (branch cost). Note that the pipeline evaluates the IALU condition (JEQ flag set by instruction #1) when the conditional instruction reaches the Integer (I) pipeline stage.

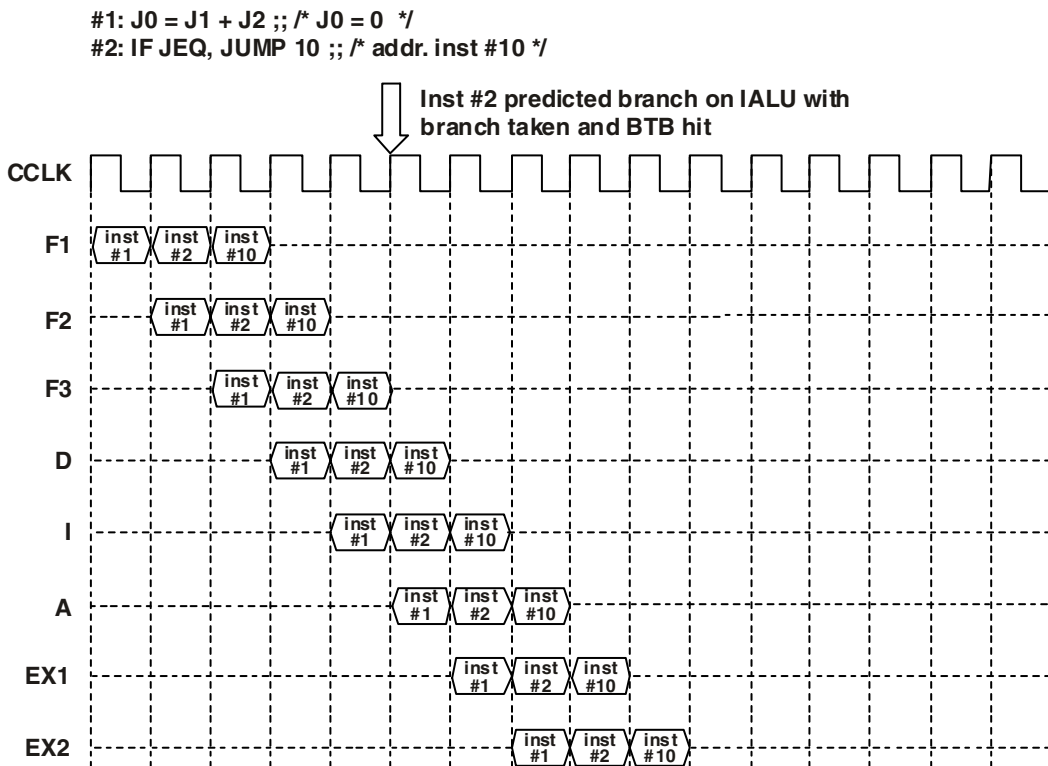


Figure 7-25. Predicted Branch Based on IALU Condition—Branch Taken—With BTB Hit

Instruction Pipeline Operations

Figure 7-26 shows a predicted branch that is based on an IALU condition. Because the branch is correctly predicted as taken and the BTB does not contain the branch instruction (BTB miss), the pipeline contents are not continuously executable (two pipeline stages voided), and there are two lost cycles (branch cost). Note that the pipeline evaluates the IALU condition (JEQ flag set by instruction #1) when the conditional instruction reaches the Integer (I) pipeline stage. From this evaluation, the pipeline determines how many pipeline stages to void.

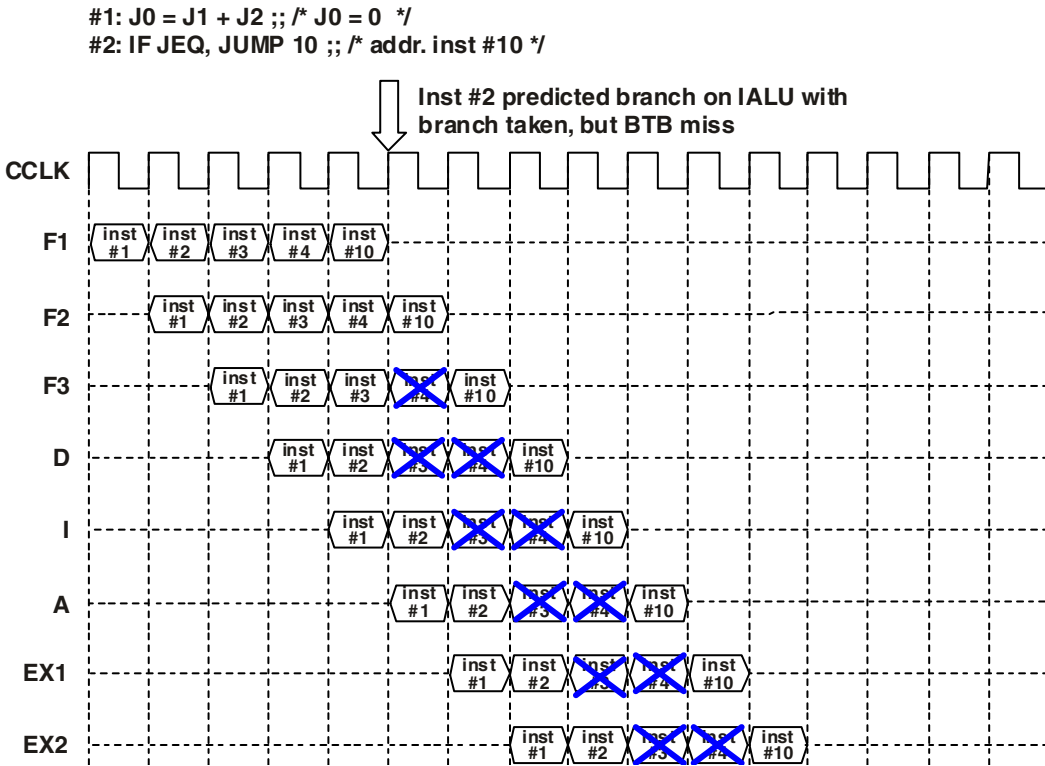


Figure 7-26. Predicted Branch Based on IALU Condition—Branch Taken—With BTB Miss

Figure 7-27 shows a not predicted (NP) branch that is based on a compute condition. Because the branch is incorrectly predicted as not taken, the pipeline contents are not continuously executable (six pipeline stages voided), and there are six lost cycles (branch cost). Note that the pipeline evaluates the compute condition (AEQ flag set by instruction #1) when the conditional instruction reaches the Execute 2 (EX2) pipeline stage. From this evaluation, the pipeline determines how many pipeline stages to void.

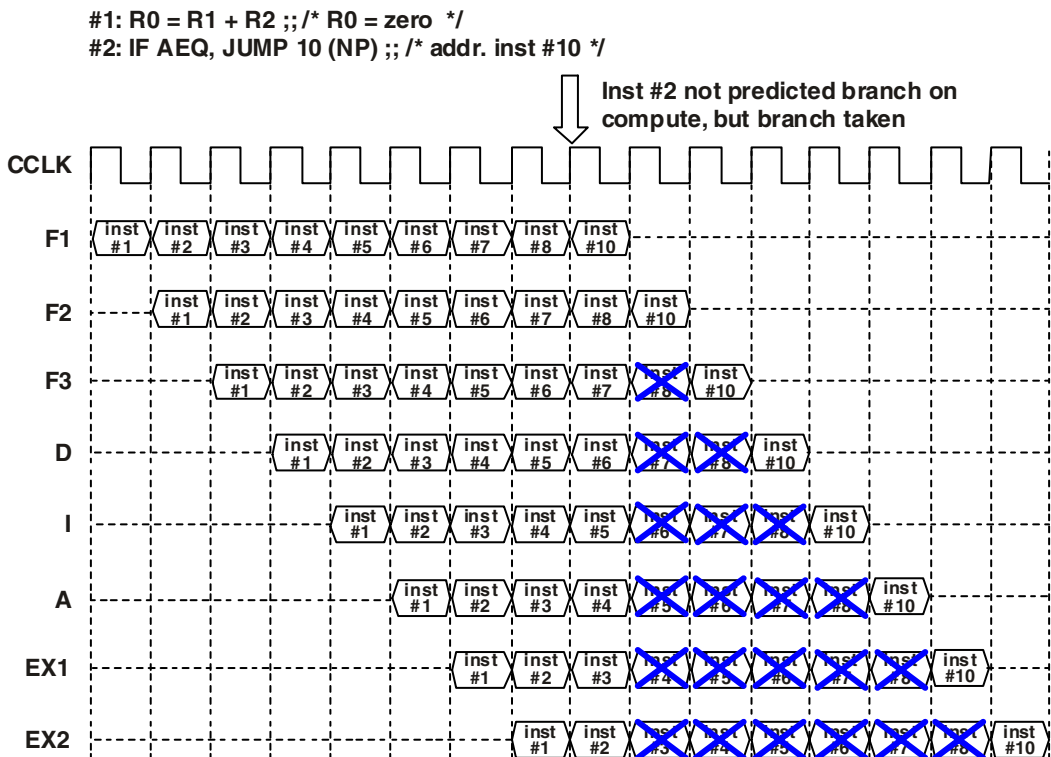


Figure 7-27. Not Predicted (NP) Branch Based on Compute Condition—Branch Taken

Instruction Pipeline Operations

Figure 7-28 shows a not predicted (NP) branch that is based on an IALU condition. Because the branch is incorrectly predicted as not taken, the pipeline contents are not continuously executable (three pipeline stages voided), and there are three lost cycles (branch cost). Note that the pipeline evaluates the IALU condition (JEQ flag set by instruction #1) when the conditional instruction reaches the Integer (I) pipeline stage. From this evaluation, the pipeline determines how many pipeline stages to void.

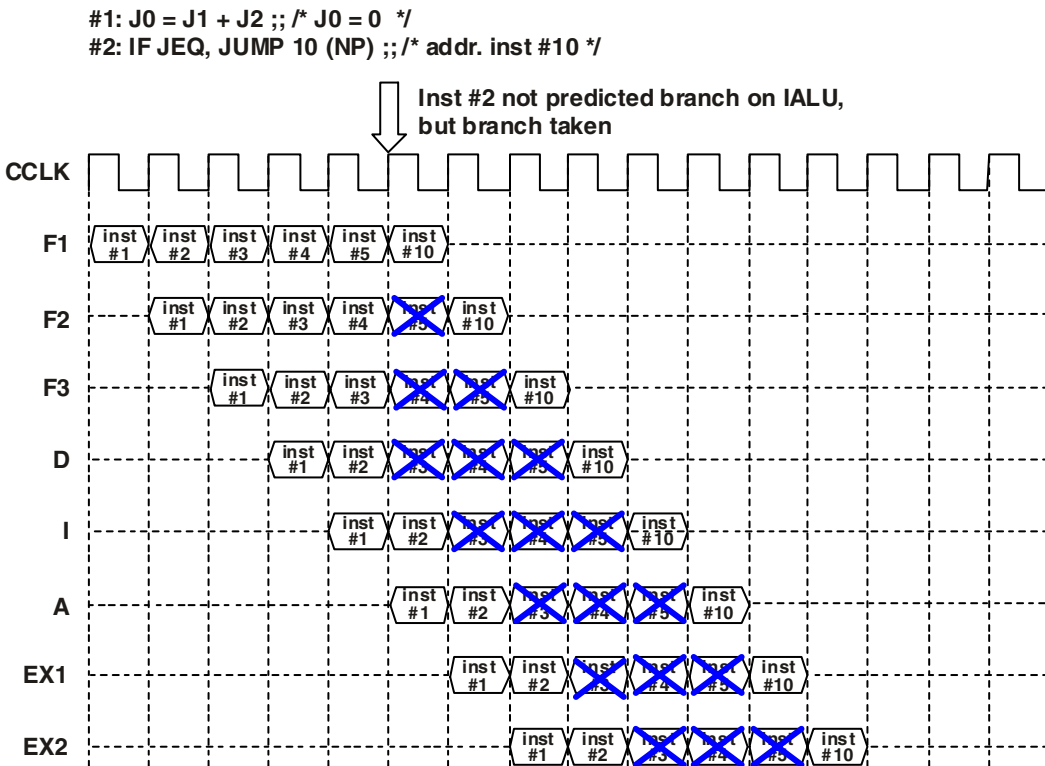


Figure 7-28. Not Predicted (NP) Branch Based on IALU Condition—Branch Taken

Figure 7-29 shows a predicted branch that is based on a compute condition. Because the branch is incorrectly predicted as taken and the BTB contains the branch instruction (BTB hit), the pipeline contents are not continuously executable (six pipeline stages voided), and there are six lost cycles (branch cost). Note that the pipeline evaluates the compute condition (AEQ flag set by instruction #1) when the conditional instruction reaches the Execute 2 (EX2) pipeline stage. From this evaluation, the pipeline determines how many pipeline stages to void.

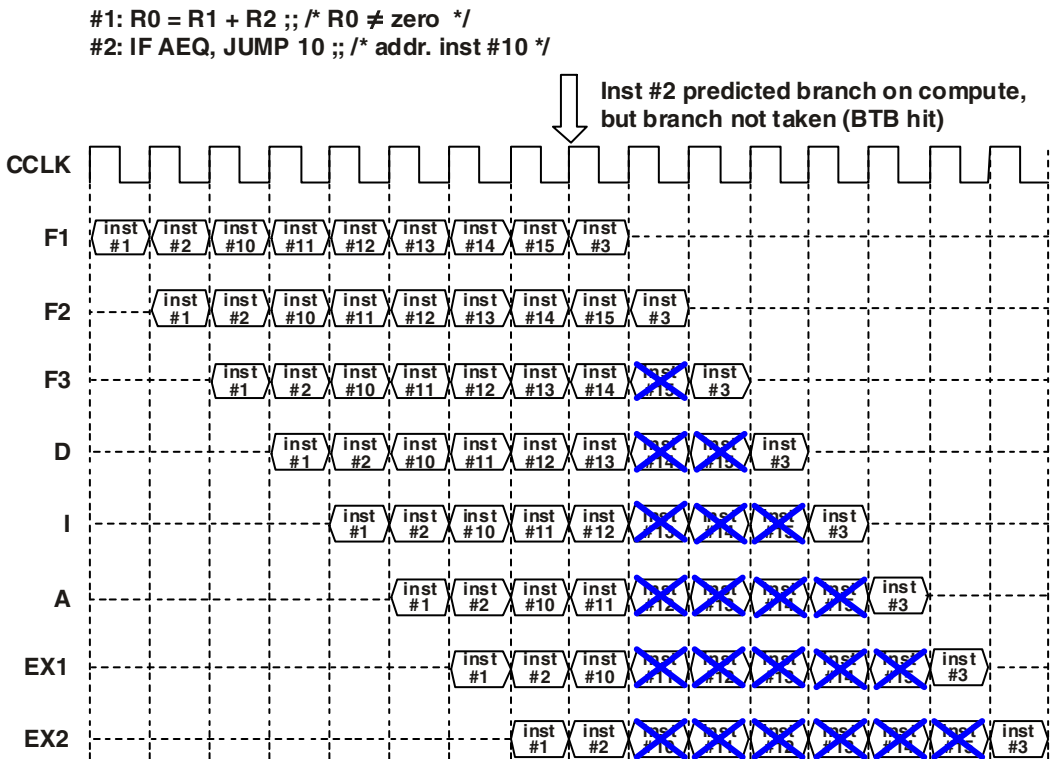


Figure 7-29. Predicted Branch Based on Compute Block Condition—Branch Not Taken—With BTB Hit

Instruction Pipeline Operations

Figure 7-30 shows a predicted branch that is based on an IALU condition. Because the branch is incorrectly predicted as taken and the BTB contains the branch instruction (BTB hit), the pipeline contents are not continuously executable (three pipeline stages voided), and there are three lost cycles (branch cost). Note that the pipeline evaluates the compute condition (JEQ flag set by instruction #1) when the conditional instruction reaches the Integer (I) pipeline stage. From this evaluation, the pipeline determines how many pipeline stages to void.

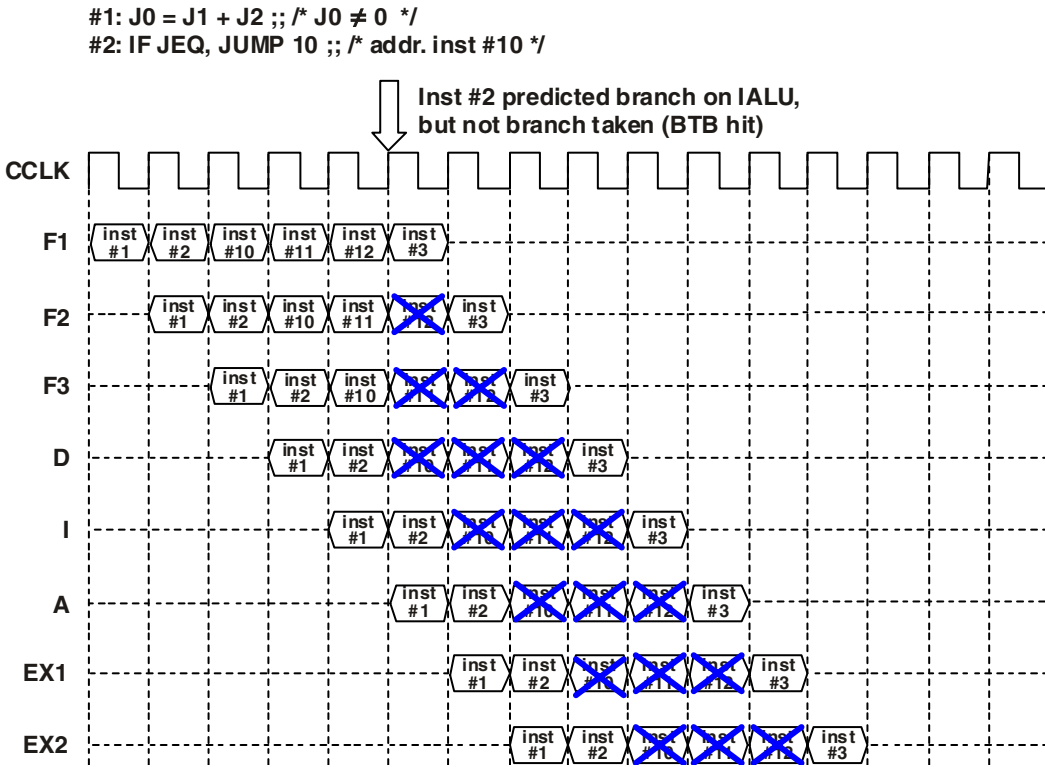


Figure 7-30. Predicted Branch Based on IALU Condition—Branch Not Taken—With BTB Hit

Figure 7-31 shows a predicted branch that is based on a compute condition. Because the branch is incorrectly predicted as taken and the BTB does not contain the branch instruction (BTB miss), the pipeline contents are not continuously executable (six pipeline stages voided), and there are six lost cycles (branch cost). Note that the pipeline evaluates the compute condition (AEQ flag set by instruction #1) when the conditional instruction reaches the Execute 2 (EX2) pipeline stage. From this evaluation, the pipeline determines how many pipeline stages to void.

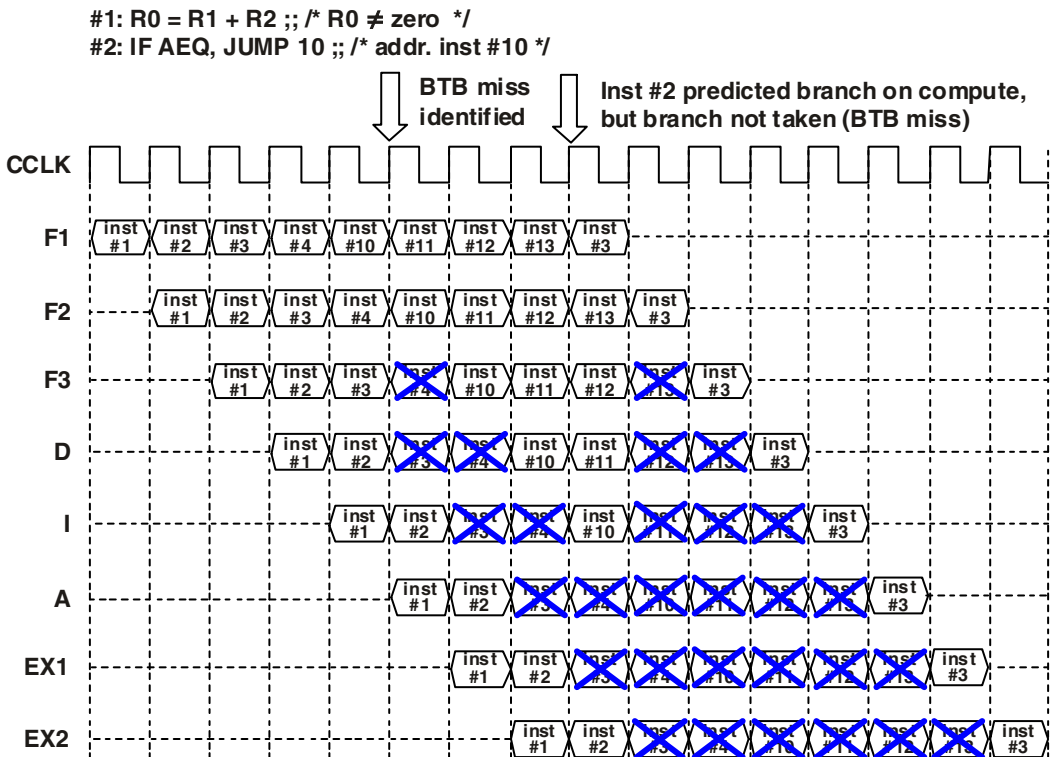


Figure 7-31. Predicted Branch Based on Compute Block Condition—Branch Not Taken—With BTB Miss

Instruction Pipeline Operations

Figure 7-32 shows a predicted branch that is based on an IALU condition. Because the branch is incorrectly predicted as taken and the BTB does not contain the branch instruction (BTB miss), the pipeline contents are not continuously executable (three pipeline stages voided), and there are three lost cycles (branch cost). Note that the pipeline evaluates the compute condition (JEQ flag set by instruction #1) when the conditional instruction reaches the Integer (I) pipeline stage. From this evaluation, the pipeline determines how many pipeline stages to void.

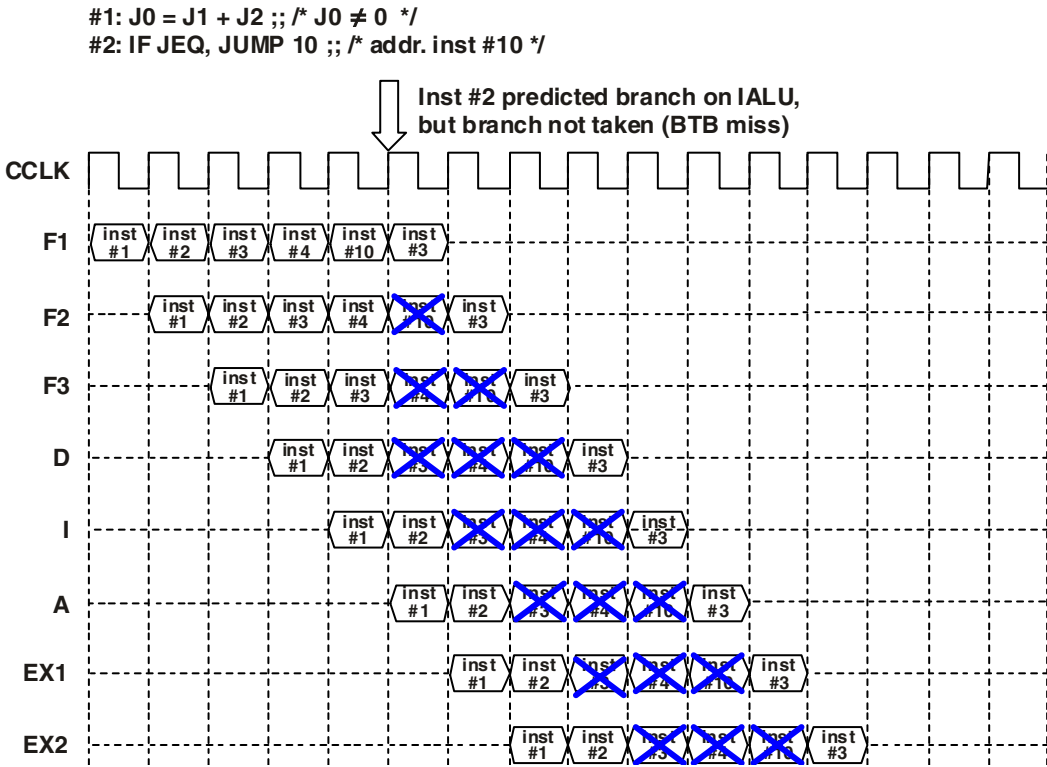


Figure 7-32. Predicted Branch Based on IALU Condition—Branch Not Taken—With BTB miss

Dependency and Resource Effects on Pipeline

The TigerSHARC processor supports any sequence of instruction lines, as long as each separate line is legal. The pipelined instruction execution causes overlap between the execution of different lines. Two problems may arise from this overlap.

- Dependency
- Resource conflict

A *dependency condition* is caused by any instruction that uses as an input the result of a previous instruction, if the previous instruction data is not ready when the current instruction needs the operand.

Resource conflicts only occur in the internal memory. The following instructions cause a bus request conflicts.

- `Load/store` – request an internal bus according to internal memory block. If the address is external, the virtual bus is used.
- `Immediate load, move reg to reg, and add or sub with option CJMP` all request the virtual bus.

If any of the above two instructions use the same internal bus, or if another resource (DMA or bus interface) requests the same bus on the same cycle that the IALU requests the bus, the bus might not be granted to the IALU. This in turn could cause a delay in the execution.

This section details the different cases of stalls. A *stall* is any delay that is caused by one of the two conditions previously described. Although the information in this manual is detailed, there may be some cases that are not defined here or conditions that are not always perceivable to the system designer. Exact behavior can only be reproduced through the TigerSHARC processor simulator.

Instruction Pipeline Operations

Stall From Compute Block Dependency

This is the most common dependency in applications and occurs on compute block operations on the compute block register file. The compute block accesses the register file for operand fetch on pipe stage Access, uses the operand on EX1, and writes the result back on EX2. The delay is comprised of basically two cycles—however, a bypass transfers the result (which is written at the end of pipe stage EX2) directly into the compute unit that is using it in the beginning of pipe stage EX1. As a result, one stall cycle is inserted in the dependent operation.

There is also a one-cycle stall when the MR is loaded immediately after multiply-accumulate. Example:

```
MR2 += R3 * R2;;  
MR2 = R4;;
```

Figure 7-33 shows two compute block instructions. Execution of instruction #2 is dependent on the result from instruction #1. The pipeline inserts a one-cycle stall at the Integer stage (I) when the register dependency is recognized. Execution of instruction #2 is stalled for one cycle.

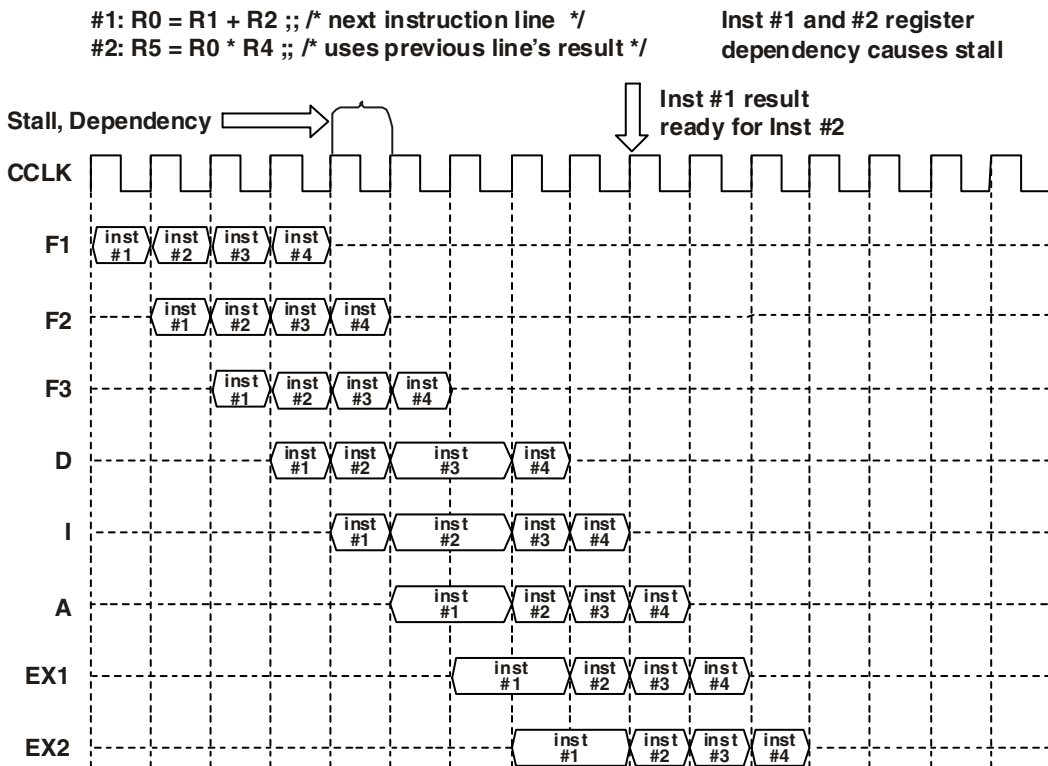


Figure 7-33. Compute Block Operations—
Result Dependency Stalls Following Instruction

As with other ALU instructions, all communications logic unit (CLU) instructions are executed in the compute pipeline. Similar to other compute instructions, all CLU instructions have a dependency check. Every use of a result of the previous line causes a stall for one cycle. In some spe-

Instruction Pipeline Operations

cial cases the stall is eliminated by using special forwarding logic in order to improve the performance of certain algorithms executions. The forwarding logic can function (and the stall can be eliminated) only when the first instruction is not predicated (for example, `if <cond>; do, <inst>;`). The exceptions cases are:

- Load `TR` or `THR` register and any instruction that uses it on the next line.
- Although the `THR` register is a hidden operand and/or result of the instructions `ACS` and `DESPREAD`, there is no dependency on it.
- The instruction `ACS`, which can use previous result of `ACS` instruction as `TRmd` with no stall. For example the following sequence will cause no stall:

```
TR3:0 = ACS (...);;  
TR7:4 = ACS (TR1:0, TR5:4, R8);;
```

Or the sequence:

```
TR3:0 = ACS (...);;  
TR7:4 = ACS (TR3:2, TR5:4, R8);;
```

However, there are a few cases that cause stalls. The first case is when the dependency is on `TRN`. For example:

```
TR3:0 = ACS (...);;  
TR7:4 = ACS (TR11:10, TR1:0, R8);;
```

The second case is when two different formats are used in the two instructions. For example:

```
XTR3:0 = ACS (TR5:4, TR7:6, R1);;
XSTR11:8 = ACS (TR15:14, TR13:12, R2);;
```

ACS of short operands has the identical flow.

- Data transfer from a CLU register to a compute register file has no dependency. The data transfer is executed in EX2.

The CLU register load can be executed in parallel to other CLU instructions. CLU register load code is similar to the code of a shifter instruction, while the code of the other CLU instructions is similar to the code of ALU instructions.

No exceptions are caused by the CLU instructions.

Stall from Bus Conflict

The execution of the following instructions uses the internal bus:

- $Ureg = [Jm + Jn | imm]$, $Ureg = [Km + Kn | imm]$
all data types and options
- $[Jm + Jn | imm] = Ureg$, $[Km + Kn | imm] = Ureg$
all data types and options
- $Ureg = Ureg$
even if both are in the same register file
- $Ureg = imm$
- $Js = Jm + | - Jn$ (CJ)

Instruction Pipeline Operations

The first two instruction types select a bus according to the memory block:

- Address 0x00000 – 0xFFFF: bus #0
- Address 0x80000 – 0x8FFFF: bus #1
- Address 0x100000 – 0x10FFFF: bus #2
- Address 0x1C00000 – 0xFFFFFFFF: External address

The other three instructions use the virtual bus. The arbitration between the masters on the bus is detailed in “Bus Arbitration Protocol” in the *ADSP-TS101 TigerSHARC Processor Hardware Reference*.

The IALU always requests the bus on pipe stage Integer. If it doesn't receive the bus, the execution of the bus transaction is delayed until the bus is granted. The rest of the line, however, including the other IALU operations (for example, post-modify of address), are continued. This is to prevent deadlock in case of two memory accesses in the same cycle to the same bus. The following instruction lines are stalled until this line can continue executing the transaction (or transactions, if more than one of the transaction's instructions are in execution).

Figure 7-34 shows a load instruction, using an IALU post-modify addressing memory access. The memory access stalls for two cycles due to a bus conflict over access to the memory block containing the address in J0. Execution of instruction #1 is extended over three cycles—two for the stall and one for the access.

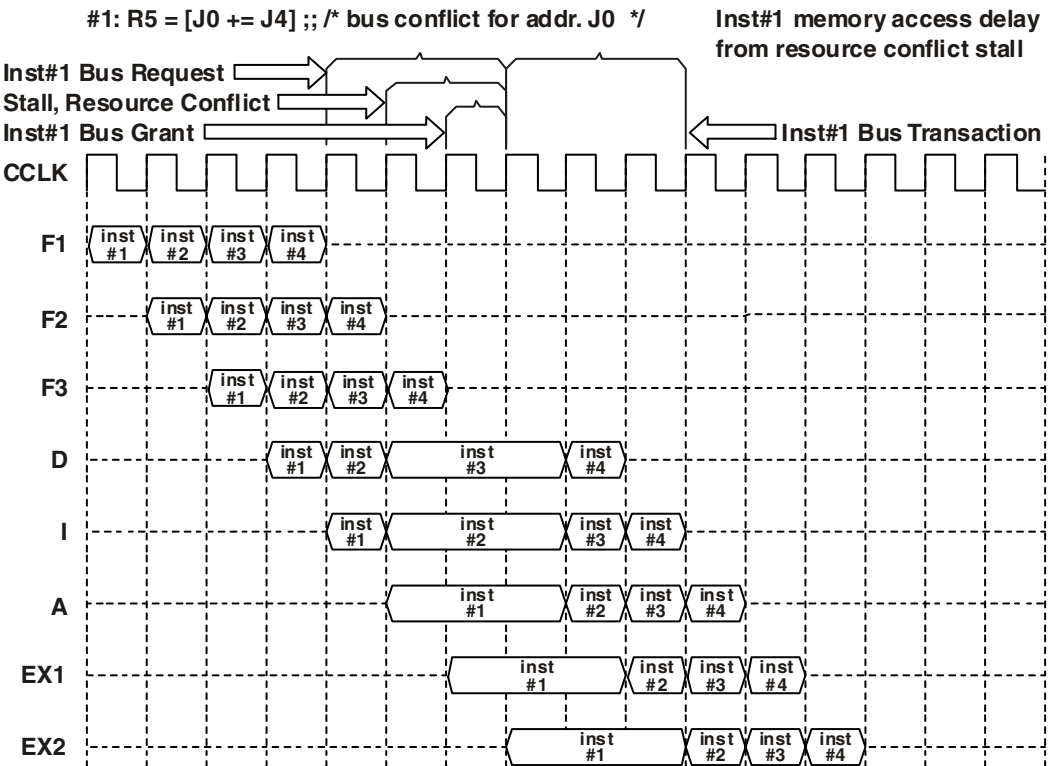


Figure 7-34. Register Load Using Post-Modify With Update—Resource Conflict Stalls Bus Access

Instruction Pipeline Operations

Stall From Compute Block Load Dependency

Data in load instructions is transferred at pipe stage EX2, exactly as in compute block operations. In case of dependency between a load instruction and compute operation that uses this data, the behavior is similar to that of compute block dependency (see [Figure 7-35 on page 7-63](#)). For example, the following sequence:

```
XR0 = [J31 + memory_access] ;;  
XR5 = R0 * R4 ;; /* One-cycle Stall */
```

This would cause a one-cycle delay, occurring when the load instruction comes from internal memory and the bus was accepted by the IALU that executes the transaction.

If the load is from external memory or the bus request was delayed, the second instruction is executed two cycles after the completion of the load—that is, after the data is returned.

Stall From IALU Load Dependency

The dependency between load instructions and IALU instructions is more problematic than in the previous cases because data is loaded at pipe stage EX2 and is used in stage Decode. To overcome this gap, four stalls are inserted before the instruction that is using the loaded data, as shown in Figure 7-36 on page 7-65.

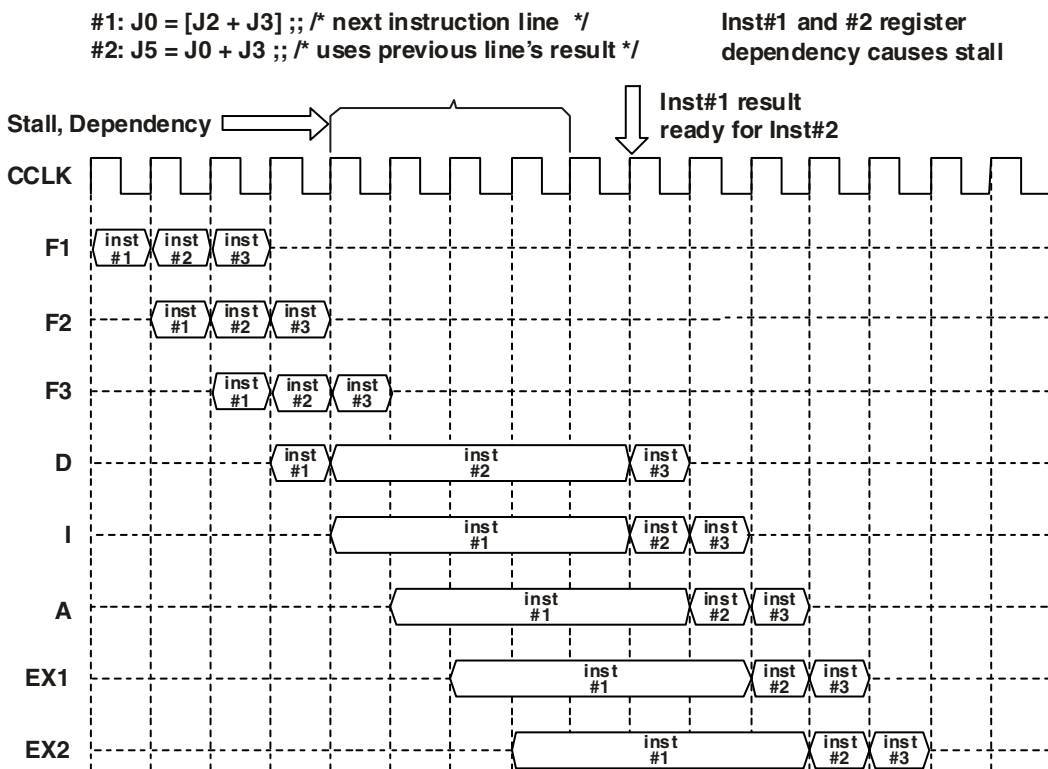


Figure 7-35. IALU Operations—
Result Dependency Stalls Following Instruction

Instruction Pipeline Operations

Stall From Load (From External Memory) Dependency

The combination of any execution instruction followed by a store instruction is dependency free, because the data is transferred by the store at pipe stage EX2. The only exception to this rule is the store of data that has been loaded from external memory. For example:

```
XR0 = [J31 + external_address] ;;  
[J0+ = 0] = XR0 ;; /* stall until XR0 is ready */
```

In a case like this, there is a stall until XR0 is actually loaded.

Stall From Conditional IALU Load Dependency

Normally IALU instructions are executed in a single cycle at pipe stage Integer. The result is pipelined and written into the result register at pipe stage EX2. If the following instruction uses the result of this instruction (either the result is used or a condition is used), the sequential instruction extracts the result from the pipeline. In one exceptional instance the bypass cannot be used, as shown in [Figure 7-36 on page 7-65](#). This occurs when the first instruction is conditional, the bypass usage is conditional, and the condition value is not known yet. The result of inst#1 in the example cannot be extracted from the bypass and must be taken from the

J0 register after the completion of the execution—after pipe stage EX2. In this case, three stall cycles are inserted if the condition is compute block, and one cycle is inserted for other types of conditions.

```
#1: IF AEQ; DO, J0 = [J2 + J3] ;; /* next instruction line */ Inst#1 and #2 register
#2: J5 = J0 + J3 ;; /* uses previous line's result */ dependency causes stall
```

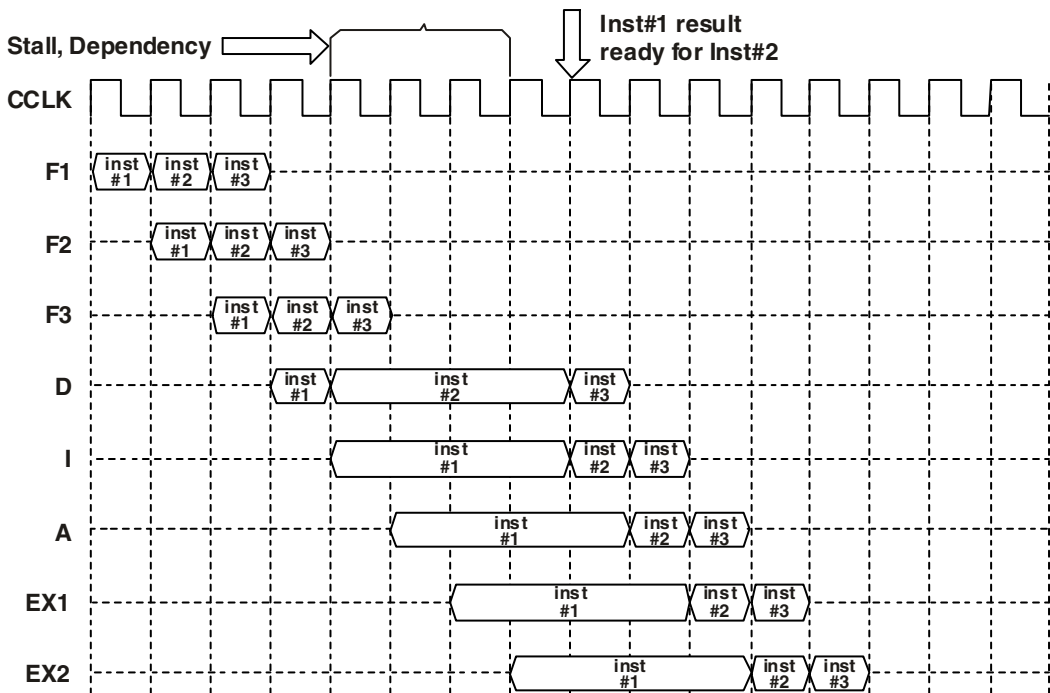


Figure 7-36. IALU Conditional Operations—
Result Dependency Stalls Following Instruction

Interrupt Effects on Pipeline

Interrupts (and exceptions) break the flow of execution and cause pipeline effects similar to other types of branching execution. The interrupt types are described in the “Interrupts” chapter of the *ADSP-TS101 TigerSHARC Processor Hardware Reference*.

The interrupts in some applications are performance critical, and the TigerSHARC processor executes them (in most cases) in the same pipeline in the optimal flow. The next sections describe the different flows of interrupts.

The most common case of a hardware interrupt is shown in [Figure 7-37 on page 7-67](#). When an interrupt is identified by the core (when the interrupt bit in `ILAT` register is set) or when the interrupt becomes enabled (the interrupt bit in `IMASK` register is set), the TigerSHARC processor starts fetching from the interrupt routine address. The execution of the instruc-

tions of the regular flow continues, except for the last instruction before the interrupt (Inst #2 in the previous example). The return address saved in RETI would be the address of instruction 2.

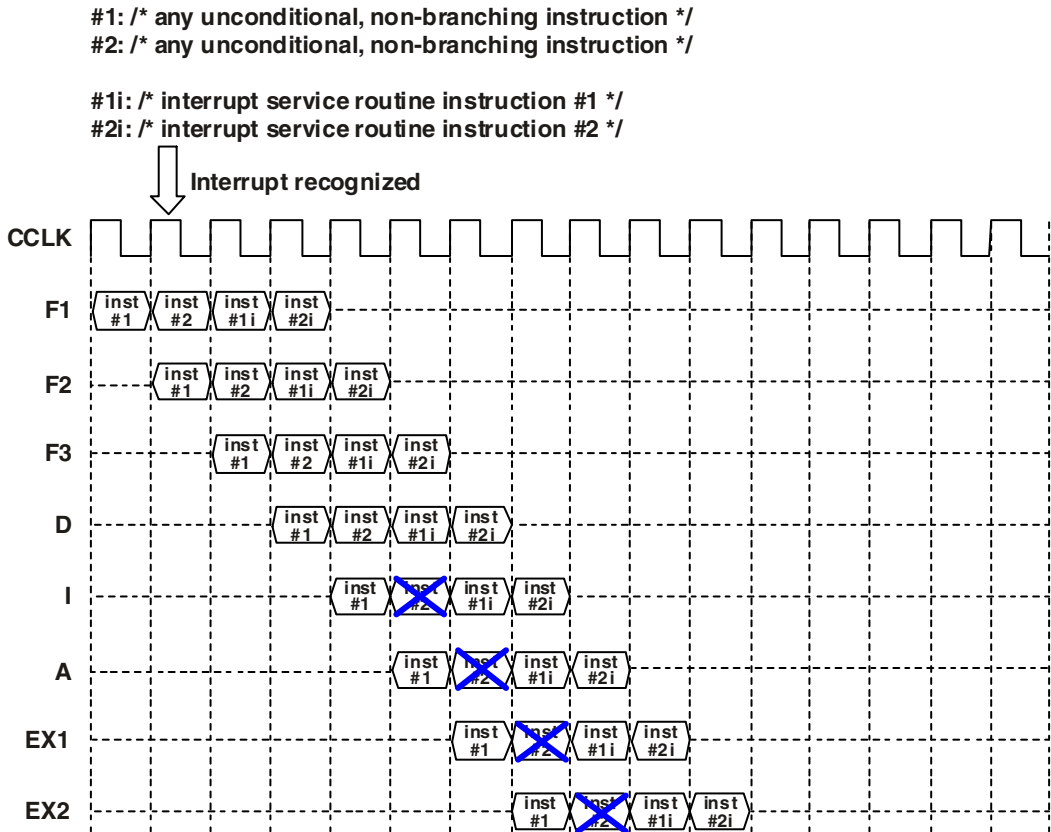


Figure 7-37. Interrupting Linear Execution (Interrupts Enabled, Interrupting Unconditional, Non-Branching Instructions)

Instruction Pipeline Operations

Interrupt During Conditional Instruction

When a predicted branch or not predicted branch instruction is fetched, the TigerSHARC processor cannot decide immediately if the branch is to be taken (as discussed in “[Branch Target Buffer \(BTB\)](#)” on page 7-34).

When an interrupt occurs during a predicted branch or not predicted branch instruction, if the prediction is found incorrect, the predicted part is aborted while the interrupt instructions that follow are not aborted.

This case is illustrated in [Figure 7-38 on page 7-69](#). When the interrupt is inserted into the flow, instructions #3 and #4 are in the pipeline speculatively. When the jump instruction is finalized (EX2) and if the speculative is found wrong, instructions #3 and #4 are aborted (similar to the flow described in [Figure 7-39 on page 7-71](#)). The instructions that belong to the interrupt flow, however, are not part of the speculative flow, and they are not aborted. The return address in this case is the correct target of the

jump instruction #2. Similar flows happen in all cases of aborted predicted or not predicted flows, when interrupt routine instructions are already in the pipeline.

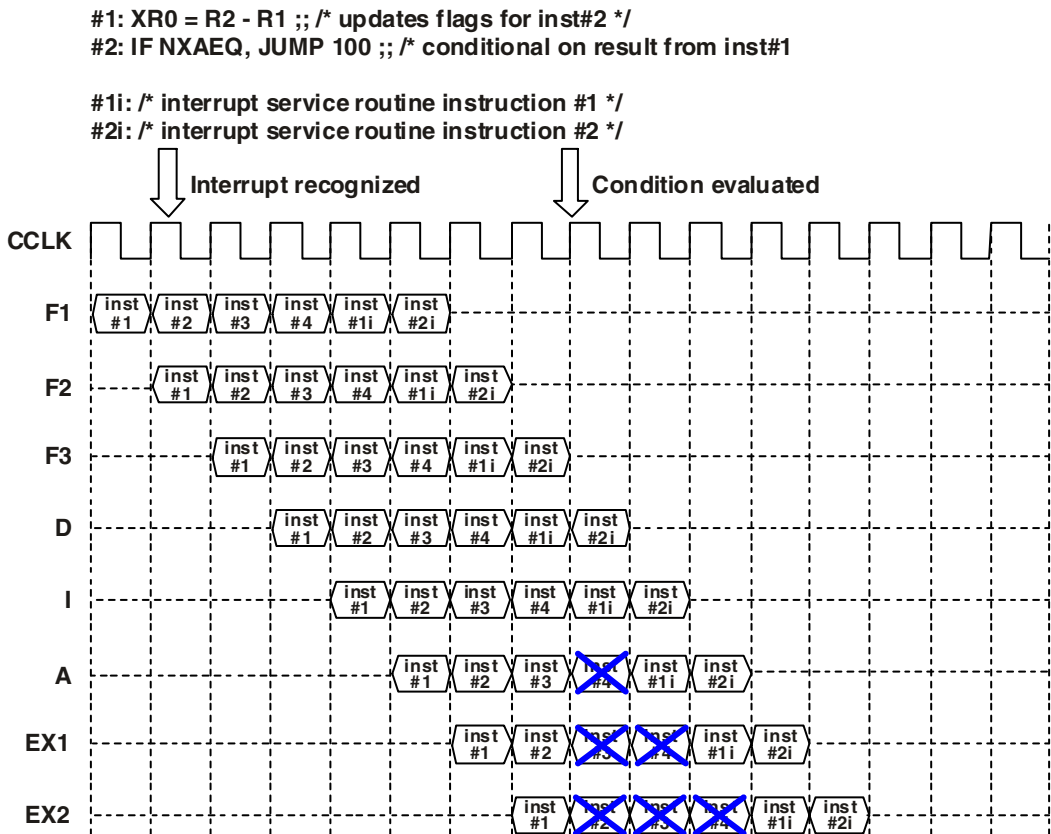


Figure 7-38. Interrupting Conditional Execution (Interrupts Enabled, Interrupting Conditional Instructions)

Instruction Pipeline Operations

Interrupt During Interrupt Disable Instruction

Sometimes the programmer needs a certain part of the code to be executed free of interrupts. In this case, disabling all hardware interrupts by clearing the global interrupt enable `GIE` bit of `IMASK` is effective immediately (contrary to clearing a specific interrupt enable). Be aware that there is a performance cost to using this feature. If the interrupt is already in the pipeline when `GIE` is cleared, it will continue execution until reaching `EX1`, and only then will it be aborted and the flow returned to a normal flow.

An example to this flow is shown in [Figure 7-39 on page 7-71](#). The interrupt is identified by the TigerSHARC processor on the second cycle (when `inst #1` is fetched). `Inst #1`—which clears `GIE`—is only completed five cycles after the interrupt occurs. When the first interrupt routine

instruction reaches EX1, GIE is checked again, and if it is cleared, the whole interrupt flow is aborted and the TigerSHARC processor returns to its original flow.

```

# : XR0 = IMASKH ;; /* load XR0 with current interrupt mask */
# : XR0 = BCLR R0 BY INT_GIE_P;;
    /* where INT_GIE_P = 0xEFFFFFFF from depts101.h file;
    this clears the GIE bit, but retains other bit values */
#1: IMASKH = XR0 ;; /* load IMASKH with data globally disabling interrupts */
#2: /* any unconditional instruction */

#1i: /* interrupt service routine instruction #1 */
#2i: /* interrupt service routine instruction #2 */
    
```

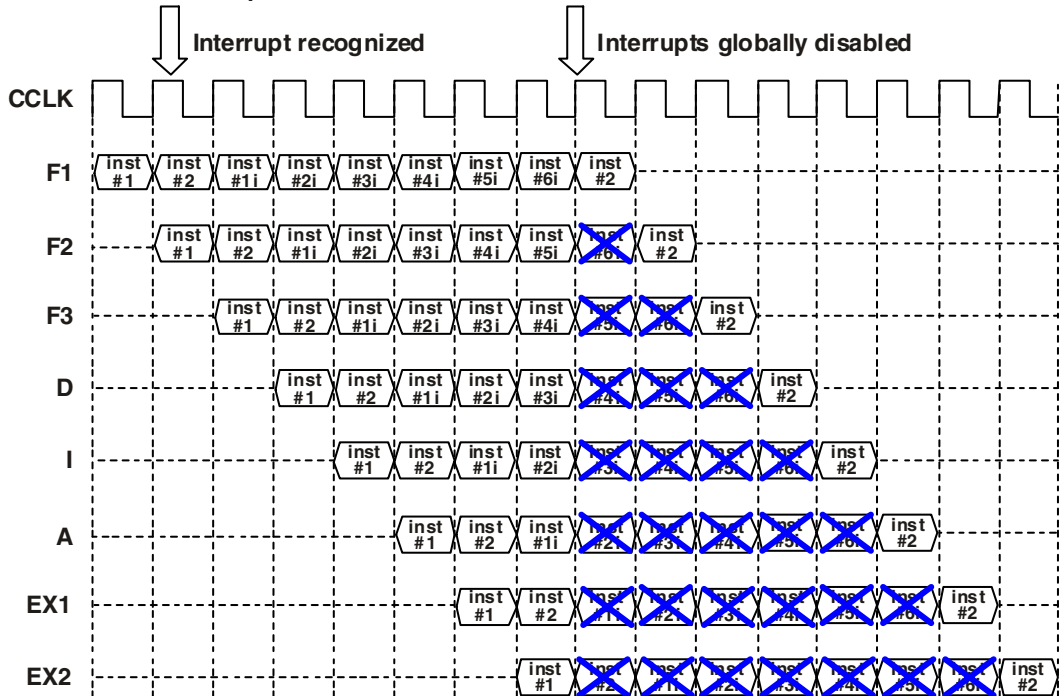


Figure 7-39. Interrupting Linear Execution (Interrupts Enabled, Interrupting Unconditional, Non-Branching Instructions)—Interrupt Disabled While In Pipeline

Sequencer Examples

Exception Effects on Pipeline

An exception is normally caused by using a specific instruction line. The exception routine's first instruction is the next instruction executed after the instruction that caused it. In order to make this happen, when the instruction line that caused the exception reaches EX2, all the instructions in the pipeline are aborted, and the TigerSHARC processor starts fetching from the exception routine. This flow is similar to the flow of unpredicted and taken jumps conditioned by an EX2 condition (see [Figure 7-27 on page 7-49](#)).

Sequencer Examples

The listings in this section provide examples of sequencer instruction usage. The comments with the instructions identify the key features of the instruction, such as predicted or not predicted branches, loop setup, and others.

Listing 7-5. Sequencer Instructions, Slots, and Lines

```
IF XAEQ, JUMP Label_1NP;;
/* This is a single instruction line and occupies one "slot" or
32-bit word */
NOP; NOP; NOP; NOP;;
IF XMEQ; D0, XR3:2 = R5:4 + R7:6;;
/* This is a two instruction line, the first slot is the condi-
tional instruction XMEQ, the second instruction is the addition
in the X computation block */
```

Listing 7-6. JUMP Instruction Example

```
.SECTION program ;
CALL test ;; /* Addr:0x0; */
```

```
/* Jumps to 0x3. Stores 0x1 in CJMP register */
NOP ;; /* Addr: 0x1 */
endhere:
JUMP endhere ;; /* Addr: 0x2 */
test:
    NOP ;; /* Addr: 0x3 */
    CJMP (ABS) ;; /* Addr: 0x4; */
    /* End of subroutine. Jumps back to Addr 0x1. */
```

Listing 7-7. CJMP_CALL Instruction Example

```
.SECTION program;
J0 = ADDRESS(endhere) ;; /* Addr:0x0 */
CJMP = ADDRESS(test) ;; /* Addr: 0x1 */
/* Preload cjmp register with 0x6 */
CJMP_CALL (ABS) ;; /* 0x2 */
/* Jumps to value stored in CJMP register (0x6).
/* Puts new value of 0x3 in CJMP register. */
NOP ;; /* Addr: 0x3 */
endhere:
    NOP ;; /* Addr: 0x4 */
    JUMP endhere ;; /* Addr: 0x5 */
test:
    J31 = J0 + J31 (CJMP) ;; /* Addr: 0x6 */
    /* Uses IALU add (CJMP) option */
    /* Loads result (0x4) into CJMP register. */
    CJMP (ABS) ;; /* Addr: 0x7 */
    /* End of subroutine. Jumps back to 0x4. */
```

Listing 7-8. Zero-Overhead and Near-Zero-Overhead Loops Example

```
J6 = J31 + 10;; /*initialize counter for outermost loop */
Outer_loop:
    instr0;;
```

Sequencer Examples

```
LC1 = 5; /* Counter for middle loop. */
/* Use zero overhead in loop counter number one */
instr1;;
Middle_loop:
    instr2;;
    LC0 = 6; instr3;; /* Counter for inner loop. */
    /* Use loop counter zero */
    Inner_loop:
        instr4;;
        instr5;;
    If NLC0E, JUMP Inner_loop; instr6;; /* End inner loop */
    instr7;;
    If NLC1E, JUMP Middle_loop; instr8;; /*End middle loop */
    instr9; J6 = J6 - 1;; /* Decrement counter for outer loop */
    instr10;;
    If NJEQ, JUMP Outer_loop; instr11;; /* End outer loop*/
```

Listing 7-9. Branch Target Buffer (BTB) Usage (No Branch Prediction)

```
LC0 = 100;; /* Initialize loop count */
instr2;;
instr3;;
_loop:
    instr4;;
    instr5;;
    If NLC0E, JUMP _loop (NP); instr6;; /* End loop */
```

Some notes on [Listing 7-9](#):

- “NP” denotes no predict
- First 99 times through the loop there is a three cycle penalty
- Last time though the loop there is no penalty
- Total penalty cycles => 297

Listing 7-10. Branch Target Buffer (BTB) Usage (Branch Prediction)

```
SQCTL=0xF0301;; /* Make sure BTB is enabled */
LC0 = 100;; /* Initialize loop count */
instr2;;
instr3;;
_loop:
    instr4;;
    instr5;;
If NLC0E, JUMP _loop; instr6;; /* End loop */
```

Some notes on [Listing 7-10](#):

- First time through (and second time through if the branch is not updated in the BTB) the loop there is a two cycle penalty
- Next 98 times through the loop there is no penalty
- Last time through the loop there is a three cycle penalty
- Total penalty cycles => five (seven if no BTB update)

Sequencer Instruction Summary

[Listing 7-11](#) shows the sequencer instructions' syntax. The conventions used in these listings for representing register names, optional items, and choices are covered in detail in [“Register File Registers” on page 2-5](#).

Briefly, these conventions are:

- { } – the curly braces enclose options; these braces are not part of the instruction syntax.
- | – the vertical bars separate choices; these bars are not part of the instruction syntax.
- *Label* – the program label in italic represents a user-selectable program label, a PC-relative 16- or 32-bit address, or a 32-bit absolute address. When a program *Label* is used instead of an address, the assembler converts the *Label* to an address, using a 16-bit address when the *Label* is contained in the same program `.SECTION` as the branch instruction and using a 32-bit address when the *Label* is not in the same program `.SECTION` as the branch instruction. For more information on relative and absolute addresses and branching, see [“Branching Execution” on page 7-16](#).



Each instruction presented here occupies one instruction slot in an instruction line. For more information about instruction lines and instruction combination constraints, see [“Instruction Line Syntax and Structure” on page 1-20](#) and [“Instruction Parallelism Rules” on page 1-24](#).

PC-relative addressing and predicted branch for jumps and calls are the default operation. To use absolute addressing, use the (ABS) option. To indicate a not predicted branch, use the (NP) option.

In conditional instructions, the *Condition* is one of the following:

- {N}{X|Y|XY} AEQ|ALT|ALE|MEQ|MLT|MLE|SEQ|SLT|SF0|SF1
- {N}{J|K} EQ|LT|LE
- {N}ISF0|ISF1|BM|LC0E|LC1E|FLG0|FLG1|FLG2|FLG3.

The AEQ, ALT, and ALE conditions are the ALU equal, less than, and less than or equal to zero conditions. The MEQ, MLT, and MLE conditions are the multiplier equal, less than, and less than or equal to zero conditions. The SEQ and SLT conditions are the Shifter equal and less than zero conditions. SF0 and SF1 are the compute block static flag 0 and 1 conditions.

The JEQ, JLT, and JLE conditions are the J-ALU equal, less than, and less than or equal to zero conditions and, similarly, for K-ALU.

The ISF0 and ISF1 conditions are the general integer static flag conditions. BM is bus master. The LC0E and LC1E conditions are the loop counter 0 and 1 equal to zero conditions. The FLG0 through FLG3 conditions are the flag pin conditions.

An N prefix on a condition negates the condition in the instruction. Thus, NJLT is “J-IALU greater than or equal to zero.” For example, see how N is used in the following instruction:

```
IF NXMLT, JUMP label_10;;
```

executes a jump to the address that corresponds to `label_10` if the condition MLT (multiplier less than zero) in compute block X evaluates false.

JUMP|...|CJMP denotes any valid branch instruction. This is a line with more than one instruction.

Sequencer Instruction Summary

Instructions that follow after a conditional instruction or after a jump/call in a line can have keyword `DO|ELSE` or have no prefix. Absence of prefix indicates execution regardless of the condition value. Keyword `DO` relates only to condition instruction, while `ELSE` relates only to conditional branch.



The `NOP`, `IDLE (lp)`, `BTBINV`, `TRAP (<imm>)`, and `EMUTRAP` instructions may not be conditional. The following instruction line for example is not legal: `if aeq; do idle;;`

Listing 7-11. Sequencer Instructions

```
{IF Condition,} JUMP|CALL <Label> {(NP)} {(ABS)} ;

{IF Condition,} CJMP|CJMP_CALL {(NP)} {(ABS)} ;

{IF Condition,} RETI|RTI {(NP)} {(ABS)} ;

{IF Condition,} RDS ;

IF Condition;
    DO, instruction; DO, instruction; DO, instruction ;;
/* This syntax permits up to three instructions to be controlled
by a condition. Omitting the DO before the instruction makes the
instruction unconditional. */

IF Condition, JUMP|CALL|CJMP|CJMP_CALL ;
    ELSE, instruction; ELSE, instruction; ELSE, instruction ;;
/* This syntax permits up to three instructions to be controlled
by a condition. Omitting the ELSE before the instruction makes
the instruction unconditional. */

SF1|SF0 = Condition ;

SF1|SF0 += AND|OR|XOR Condition ;
```

IDLE {(LP)} ;

BTBINV

TRAP (<Imm5>) ;;

EMUTRAP ;;

NOP ;

Sequencer Instruction Summary

8 INSTRUCTION SET

This chapter describes the ADSP-TS101 TigerSHARC processor instruction set in detail. The instruction set reference pages are split into groups according to the units that execute the instruction.

- [“ALU Instructions”](#) on page 8-2
- [“CLU Instructions”](#) on page 8-91
- [“Multiplier Instructions”](#) on page 8-121
- [“Shifter Instructions”](#) on page 8-175
- [“IALU \(Integer\) Instructions”](#) on page 8-200
- [“IALU \(Load/Store/Transfer\) Instructions”](#) on page 8-218
- [“Sequencer Instructions”](#) on page 8-228

For information on these architectural units of the TigerSHARC processor core, see [“ALU”](#) on page 3-1, [“Multiplier”](#) on page 4-1, [“Shifter”](#) on page 5-1, [“IALU”](#) on page 6-1, and [“Program Sequencer”](#) on page 7-1. For more information on registers (and register naming syntax) used in the instructions, see [“Compute Block Registers”](#) on page 2-1, [“ALU Instruction Summary”](#) on page 3-28, [“Multiplier Instruction Summary”](#) on page 4-23, [“Shifter Instruction Summary”](#) on page 5-19, [“IALU Instruction Summary”](#) on page 6-39, and [“Sequencer Instruction Summary”](#) on page 7-76.

The instructions within each group are described in detail in this chapter. Full details about instruction decoding can be found in [“Instruction Decode”](#) on page C-1.

ALU Instructions

The ALU performs all *arithmetic operations* (addition/subtraction) for the processor on data in fixed-point and floating-point formats and performs *logical operations* for the processor on data in fixed-point formats. The ALU also executes *data conversion operations* such as expand/compact on data in fixed-point formats. For a description of ALU operations, status flags, conditions, and examples, see [“ALU” on page 3-1](#).

The conventions used in these reference pages for representing register names, optional items, and choices are covered in detail in [“Register File Registers” on page 2-5](#). Briefly, these conventions are:

- { } – the curly braces enclose options; these braces are not part of the instruction syntax.
- | – the vertical bars separate choices; these bars are not part of the instruction syntax.
- *Rmd* – the register names in italic represent user-selectable single (*Rs*, *Rm*, *Rn*), double (*Rsd*, *Rmd*, *Rnd*) or quad (*Rsq*, *Rmq*, *Rnq*) register names.



Each instruction presented on these reference pages occupies one instruction slot in an instruction line. For more information about instruction lines and instruction combination constraints, see [“Instruction Line Syntax and Structure” on page 1-20](#) and [“Instruction Parallelism Rules” on page 1-24](#).

Add/Subtract

Syntax

$$\{X|Y|XY\}\{S|B\}Rs = Rm +|- Rn \{(\{S|SU\})\} ;$$

$$\{X|Y|XY\}\{L|S|B\}Rsd = Rmd +|- Rnd \{(\{S|SU\})\} ;$$

Function

These instructions add or subtract the operands in registers Rm and Rn . The result is placed in register Rs . The L, S, and B prefixes denote the operand type and d denotes operand size—see “[Instruction Line Syntax and Structure](#)” on page 1-20.

The MAX_SN, MIN_SN, and MAX_UN are the maximum signed, minimum signed, and maximum unsigned numbers representable in the output format. For instance, if the output format is 16-bit short words, then MAX_SN=0x7fff, MIN_SN=0x8000, and MAX_UN=0xffff.

For saturation on add:

- Signed saturation—option (S):
 - If $Rm+Rn$ overflows the MAX_SN, then $Rs=MAX_SN$
 - If $Rm+Rn$ underflows the MIN_SN, then $Rs=MIN_SN$
- Unsigned saturation—option (SU):
 - If $Rm+Rn$ overflows the MAX_UN, then $Rs=MAX_UN$

For saturation on subtract:

- Signed saturation—option (S):
 - If $Rm-Rn$ overflows the MAX_SN, then $Rs=MAX_SN$
 - If $Rm-Rn$ underflows the MIN_SN, then $Rs=MIN_SN$

ALU Instructions

- Unsigned saturation—option (SU):
 - If $R_m - R_n$ underflows zero, then $R_S=0$

Status Flags

AZ	Set if all bits in result are zero
AN	Set to the most significant bit of result
AV (AOS)	Signed overflow
AC	Set to carry out; can be used to indicate unsigned overflow (inverted on subtract)

Options

()	Saturation off
(S)	Saturation active, and signed
(SU)	Saturation active, and unsigned

Example

YBR9 = R2 + R8 (S);; *see Figure 8-1*

YSR2 = R1 - R0 (SU);; *see Figure 8-2*

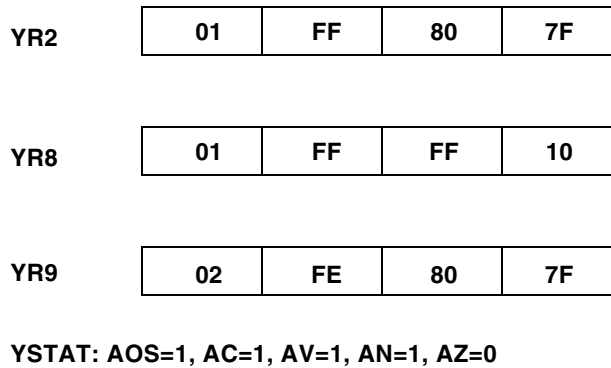


Figure 8-1. Quad Byte, Single Register, Signed Saturating Addition in Compute Block Y

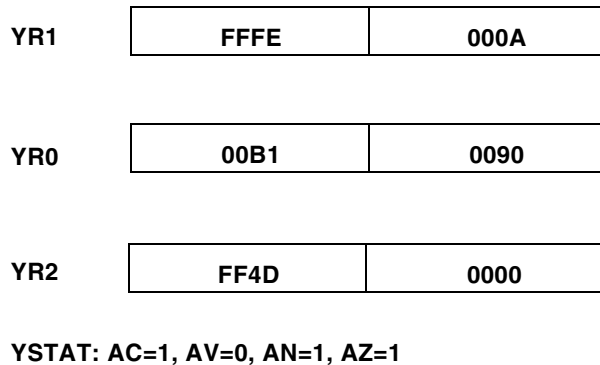


Figure 8-2. Dual Short, Single Register, Unsigned Saturating Subtraction in Compute Block Y

ALU Instructions

Add/Subtract With Carry/Borrow

Syntax

$\{X|Y|XY\}Rs = Rm + CI \{-1\} ;$

$\{X|Y|XY\}LRsd = Rmd + CI \{-1\} ;$

$\{X|Y|XY\}\{S|B\}Rs = Rm +|- Rn + CI \{-1\} \{(\{S|SU\})\} ;$

$\{X|Y|XY\}\{L|S|B\}Rsd = Rmd +|- Rnd + CI \{-1\} \{(\{S|SU\})\} ;$

Function

These instructions add with carry or subtract with borrow the operands in registers Rm and Rn . The carry (CI) is indicated by the AC flag in $X/YSTAT$. The Rn operand may be omitted, performing an add or subtract with carry or borrow with Rm register and the CI. The result is placed in register Rs . The prefix L denotes long-word operand type and the suffix d denotes dual register size.

For add with carry:

- Signed saturation—option (S):
 - If $Rm+CI>MAX_SN$, then $Rs=MAX_SN$
 - If $Rm+Rn+CI<MIN_SN$, then $Rs=MIN_SN$
- Unsigned saturation—option (SU):
 - If $Rm+Rn+CI>MAX_UN$, then $Rs=MAX_UN$

For subtract with borrow:

- Signed saturation—option (S):
 - If $Rm-Rn+CI-1>MAX_SN$, then $Rs=MAX_SN$
 - If $Rm-Rn+CI-1<MIN_SN$, then $Rs=MIN_SN$

- Unsigned saturation—option (SU):

- If $R_m - R_n + CI - 1 < 0$, then $R_s = 0$

MAX_SN, MIN_SN, and MAX_UN are the maximum signed, minimum signed, and maximum unsigned numbers representable in the output format.

Status Flags

AZ	Set if all bits in result are zero
AN	Set to the most significant bit of result
AV (AOS)	Signed overflow
AC	Set to carry out; can be used to indicate unsigned overflow (inverted on subtract)

Options

()	Saturation off
(S)	Saturation active, and signed
(SU)	Saturation active, and unsigned

Example

```
xR2 = 0x00000001;;
xR0 = 0xFFFFFFFF;;
xR1 = R0 + R0;; /* generates carry */
xR3 = R2 + R2 + CI;;
```

The result in xR3 is 0x00000003, and XSTAT = 0.

```
R9 = R3 - R10 + CI - 1;;
```

If R3 = 0x17 and R10 = 0x4 and the carry from the previous ALU operation = 1 then R9 = 0x13

ALU Instructions

Average

Syntax

$$\{X|Y|XY\}\{S|B\}Rs = (Rm +|- Rn)/2 \{(\{T\}\{U\})\} ;$$
$$\{X|Y|XY\}\{L|S|B\}Rsd = (Rmd +|- Rnd)/2 \{(\{T\}\{U\})\} ;$$

Function

These instructions add or subtract the operands in registers Rm and Rn and divide the result by two. Because the carry resulting from the addition is right-shifted into the MSB position of the output, no overflow occurs. Rounding is to nearest even (unbiased round) or by truncation. In case of negative result in unsigned, the result should be zero, and overflow bit is set.

The L, S, and B prefixes denote the operand type and d denotes operand size—see [“Instruction Line Syntax and Structure”](#) on page 1-20.

Status Flags

AZ	Set if all bits in result are zero
AN	Set to the most significant bit of result
AV (AOS)	For add, cleared (AOS not changed); for subtract, set for negative result if option (U) or (TU) is active, otherwise AV cleared
AC	Cleared

Options

()	Signed round-to-nearest even
(T)	Signed truncate
(U)	Unsigned round-to-nearest even
(TU)	Unsigned truncate

The carry resulting from the addition is right-shifted into the MSB position according to:

- Signed inputs—options () and (T): If AV=1 then shift c_n into MSB, else arithmetic right-shift result
- Unsigned inputs—options (U) and (TU): For add, shift c_n into MSB; for subtract, shift 0 into MSB

Examples

$R9 = (R3 + R5)/2$; ; *round to nearest even*

If $R3 = 0x1$ *and* $R5 = 0x2$

then $R9 = 0x2$

If $R3 = 0x3$ *and* $R5 = 0x2$

then $R9 = 0x2$

If $R3 = 0x7FFFFFFF$ *and* $R5 = 0x1$

then , *carry shifted in* $R9 = 0x40000000$

If $R3 = 0x80000001$ *and* $R5 = 0x1$

then , *arithmetic right-shift* $R9 = 0xC0000001$

AN	Set to the most significant bit of the result prior to ABS
AV (AOS)	<p>For ABS, set when Rm is the most negative number; otherwise AV cleared</p> <p>For ABS (add), overflow, when option (X) is not used, if the result is more than the maximum signed ($0x7F...F$), or if option (X) is used, and both operands are $0x80...0$</p> <p>For ABS (subtract), overflow, when option (X) is not used, if the result is more than the maximum signed; otherwise AV cleared</p>
AC	Cleared
Options	
()	Signed outputs in range $[0x0, 0x7F...F]$
(X)	Unsigned outputs in extended range $[0x0, 0xF...F]$

ALU Instructions

Examples

R5 = ABS R4;;

If R4 = 0x8000 0000
then R5 = 0x7FFF FFFF *and* AV, AN *are set*

If R4 = 0x7FFF FFFF
then R5 = 0x7FFF FFFF

If R4 = 0xF000 0000
then R5 = 0x1000 0000 *and* AN *is set*

XR3 = ABS (R0 + R1);;

If XR0 = 0x80000001 *and* XR1 = 0x80000001
then XR3 = 0x7FFFFFFF *and* XSTAT: AOS=1, AC=0, AV=1, AN=0, AZ=0

XR3 = ABS (R0 + R1) (X);;

If XR0 = 0x80000001 *and* XR1 = 0x80000001
then XR3 = 0xFFFFFFFF *and* XSTAT: AOS=0, AC=0, AV=0, AN=0, AZ=0

R5 = ABS (R4 - R3);;

If R4 = 0x6 *and* R3 = 0xA
then R5 = 0x4

If R4 = 0xA *and* R3 = 0x6
then R5 = 0x4

If R4 = 0x7FFFFFFF8 *and* R3 = 0xFFFFFFFF0
then R5 = 0x7FFFFFFF
If option X is active, then R5 = 0x80000008

Negate

Syntax

```
{X|Y|XY}{S|B}Rs = - Rm ;
{X|Y|XY}{L|S|B}Rsd = - Rmd ;
```

Function

This instruction returns the two's-complement of the operand in register *Rm*. The result is placed in register *Rs*. If the input is the minimum negative number representable in the specified format, the result is the maximum positive number.

The L, S, and B prefixes denote the operand type and d denotes operand size—see [“Instruction Line Syntax and Structure”](#) on page 1-20.

Status Flags

AZ	Set if all bits in result are zero
AN	Set to the most significant bit of the result
AV (AOS)	Set for the maximum negative (0x80...0); otherwise AV cleared
AC	Not cleared

Example

```
XBR6 = -R3;;
```

If XR3 = 0x80 81 50 FF

then XR6 = 0x7F 7F B0 01 *and* XSTAT: AOS=1, AC=0, AV=1, AN=1, AZ=0

ALU Instructions

Maximum/Minimum

Syntax

$\{X|Y|XY\}\{S|B\}Rs = \text{MAX|MIN} (Rm, Rn) \{(\{U\}\{Z\})\} ;$
 $\{X|Y|XY\}\{L|S|B\}Rsd = \text{MAX|MIN} (Rmd, Rnd) \{(\{U\}\{Z\})\} ;$

Function

These instructions return the maximum (larger of) or minimum (smaller of) the two operands in registers *Rm* and *Rn*. The result is placed in register *Rs*. The comparison is performed byte-by-byte, short-by-short, or word-by-word—depending on the data size, where the maximum or minimum value of each comparison is passed to the corresponding byte/short/word in the result register.

If the Zero (Z) option is included in the instruction the result register *Rs* receives the operand *Rm* only if $Rm \geq Rn$ for maximum or $Rm \leq Rn$ for minimum; otherwise $Rs=0$.

The L, S, and B prefixes denote the operand type and d denotes operand size—see [“Instruction Line Syntax and Structure”](#) on page 1-20.

Status Flags

AZ	Set if all bits in result are zero
AN	Set to the most significant bit of result
AV (AOS)	Cleared (not changed)
AC	Cleared

Options

()	Signed
(U)	Unsigned
(Z)	Signed Zero option

(UZ) Unsigned Zero option

Example

SR9:8 = MAX (R3:2, R1:0);; see *Figure 8-3*

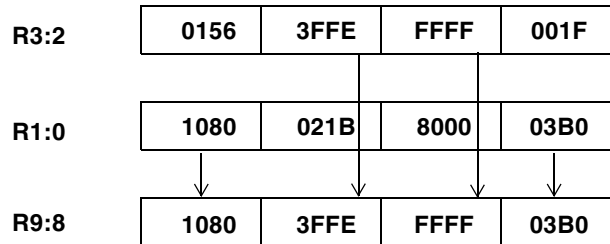


Figure 8-3. Example in Quad Short

SR9:8 = MAX (R3:2, R1:0) (Z);; see *Figure 8-4*

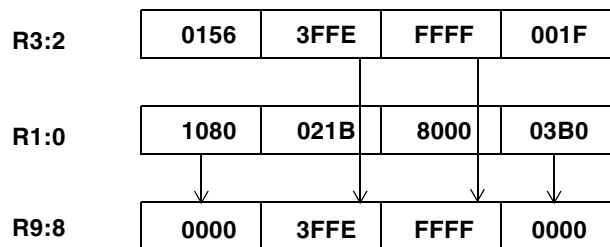


Figure 8-4. Example in Quad Short

SR9:8 = MIN (R3:2, R1:0) ;; see *Figure 8-5*

SR9:8 = MIN (R3:2, R1:0) (Z);; see *Figure 8-6*

ALU Instructions

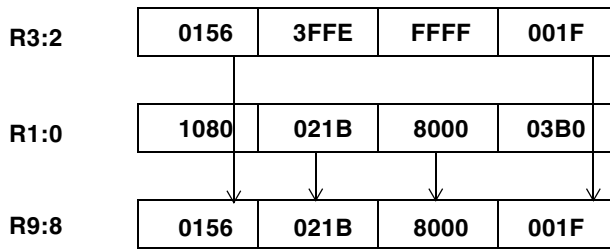


Figure 8-5. Example in Quad Short

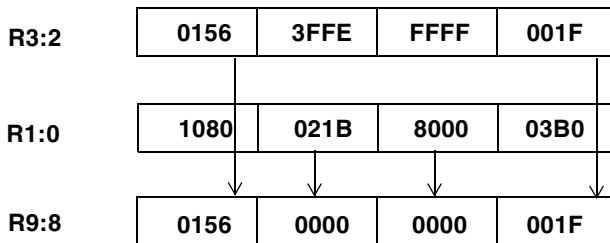


Figure 8-6. Example in Quad Short

Viterbi Maximum/Minimum

Syntax

```
{X|Y|XY}S|BRsd = VMAX|VMIN (Rmd, Rnd) ;
```

Function

These instructions return the Viterbi maximum (larger of) or Viterbi minimum (smaller of) the two operands in registers Rm and Rn . The result is placed in register Rs . The comparison is performed byte-by-byte or short-by-short—depending on the data size, where the maximum value of each comparison is passed to the corresponding byte/short in the result register. Indication of the selection bits is placed into the top 4 or 8 bits of $PR1:0$ after the old content of $PR1:0$ was shifted right by the same number of bits.

The algorithm for Viterbi maximum is:

```
for i = 0 to n-1 (n is 8 for octal byte and 4 for quad short)
  if  $Rm(i) \geq Rn(i)$  then
     $Rs(i) = Rm(i)$ 
     $PR1:0 = \{1, PR1:0[63:1]\}$ 
  else
     $Rs(i) = Rn(i)$ 
     $PR1:0 = \{0, PR1:0[63:1]\}$ 
  end (if)
end (for)
```

ALU Instructions

The algorithm for Viterbi minimum is:

```
for i = 0 to n-1 (n is 8 for octal byte and 4 for quad short)
  if Rm(i) < Rn(i) then
    Rs(i) = Rm(i)
    PR1:0 = {1, PR1:0[63:1]}
  else
    Rs(i) = Rn(i)
    PR1:0 = {0, PR1:0[63:1]}
  end (if)
end (for)
```

The S and B prefixes denote the operand type and d denotes operand size—see “[Instruction Line Syntax and Structure](#)” on page 1-20. The comparison is done in signed format always.

Status Flags

AZ	Set if all bits in result are zero
AN	Set to the most significant bit of result
AV (AOS)	Cleared (not changed)
AC	Cleared

Example

```
Initial PR1:0 = 0xabc01281 987fed35
SR9:8= VMAX (R3:2, R1:0);; see Figure 8-7
```

```
Initial PR1:0 = 0xabc01281 987fed35
SR9:8= VMIN (R1:0, R3:2);; see Figure 8-8
```

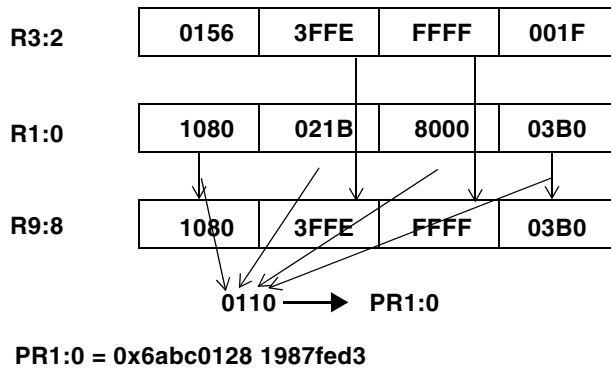


Figure 8-7. Example in Quad Short

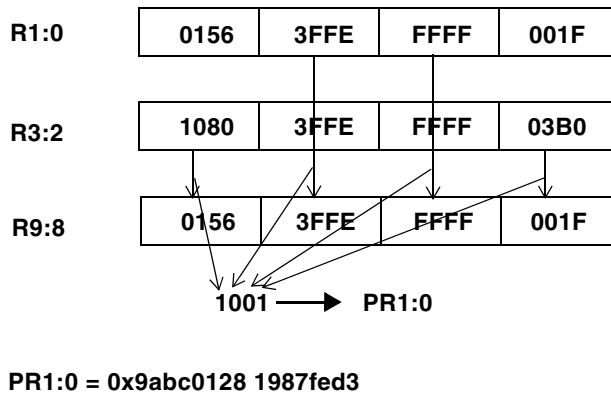


Figure 8-8. Example in Quad Short

ALU Instructions

Increment/Decrement

Syntax

```
{X|Y|XY}{S|B}Rs = INC|DEC Rm {{(S|SU)}} ;  
{X|Y|XY}{L|S|B}Rsd = INC|DEC Rmd {{(S|SU)}} ;
```

Function

These instructions add one to or subtract one from the operand in register *Rm*. The result is placed in register *Rs*.

The L, S, and B prefixes denote the operand type and d denotes operand size—see [“Instruction Line Syntax and Structure” on page 1-20](#).

Signed saturation—option (S):

- If $(R_{m+1} \text{ or } R_{m-1}) > \text{MAX_SN}$, then $R_s = \text{MAX_SN}$
- If $(R_{m+1} \text{ or } R_{m-1}) < \text{MIN_SN}$, then $R_s = \text{MIN_SN}$

Unsigned saturation—option (SU):

- If $(R_{m+1} \text{ or } R_{m-1}) < 0$, then $R_s = 0$

MAX_SN and MIN_SN are the maximum and minimum signed numbers representable in the output format. For instance, if the output format is 16-bit short words, then MAX_SN=0x7fff, MIN_SN=0x8000.

Status Flag

AZ	Set if all bits in result are zero
AN	Set to the most significant bit of result
AV (AOS)	Signed overflow
AC	Set to carry out; can be used to indicate unsigned overflow (inverted for decrement)

Options

- () Saturation off
- (S) Saturation active, and signed
- (SU) Saturation active, and unsigned

Example

```
R6 = INC R3 ;;  
If R3 = 0x00...01D  
then R6 = 0x00...01E
```

```
R6 = DEC R3 ;;  
If R3 = 0x00...01D  
then R6 = 0x00...01C
```

ALU Instructions

Compare

Syntax

```
{X|Y|XY}{S|B}COMP(Rm, Rn) {(U)} ;  
{X|Y|XY}{L|S|B}COMP(Rnd,Rnd) {(U)} ;
```

Function

This instruction compares the operand in register *Rm* with the operand in register *Rn*. The instruction sets the *AZ* flag if the two operands are equal, and the *AN* flag if the operand in register *Rm* is smaller than the operand in *Rn*. The comparison is performed byte-by-byte, short-by-short or word-by-word, depending on the data size. Note that as in all compute block instructions, in multiple data elements (for example, the instruction `BCOMP (Rm, Rn)`; quad byte compare), the flags are determined by ORing the result flag values from individual results.

The L, S, and B prefixes denote the operand type and d denotes operand size—see [“Instruction Line Syntax and Structure” on page 1-20](#).

Status Flags

AZ	Set if <i>Rm</i> and <i>Rn</i> are equal
AN	Set if <i>Rm</i> is less than <i>Rn</i>
AV (AOS)	Cleared (not changed)
AC	Cleared

Options

()	Signed
(U)	Unsigned

Examples

```
COMP (R3, R5) ;;
```

If R3 = 0xFFFFFFFF *and* R5 = 0x0
then AN = 1 *and* .AZ = 0

```
COMP (R3, R5) (U) ;;
```

If R3 = 0xFFFFFFFF *and* R5 = 0x0
then AN = 0 *and* .AZ = 0

ALU Instructions

Clip

Syntax

```
{X|Y|XY}{S|B}Rs = CLIP Rm BY Rn ;  
{X|Y|XY}{L|S|B}Rsd = CLIP Rmd BY Rnd ;
```

Function

This instruction returns the signed operand in register *Rm* if the absolute value of the operand in *Rm* is less than the absolute value of the operand in *Rn*. Otherwise, returns $|Rn|$ if *Rm* is positive, and $-|Rn|$ if *Rm* is negative. The result is placed in register *Rs*.

The L, S, and B prefixes denote the operand type and d denotes operand size—see “[Instruction Line Syntax and Structure](#)” on page 1-20.

Status Flags

AZ	Set if all bits in result are zero
AN	Set to the most significant bit of result
AV (AOS)	Cleared (not changed)
AC	Cleared

Example

```
XR3:2 = CLIP R9:8 BY R1:0;; see Figure 8-9
```

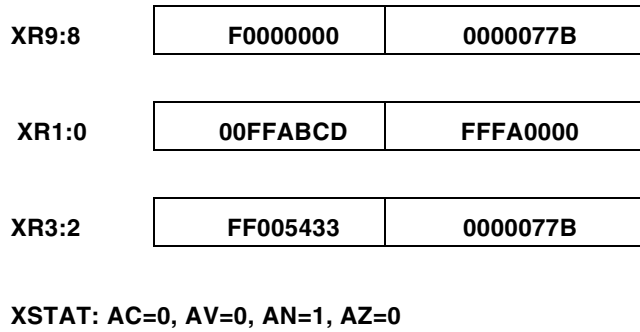


Figure 8-9. Dual Normal Word In Compute Block X

ALU Instructions

Sum

Syntax

$$\{X|Y|XY\}Rs = \text{SUM } S|B \ Rm \ \{(U)\} ;$$
$$\{X|Y|XY\}Rs = \text{SUM } S|B \ Rmd \ \{(U)\} ;$$

Function

This instruction adds the bytes/shorts in register *Rm* into the result register *Rs*. If the bytes/shorts are signed, they are sign extended before being added. The result is always right-justified—for example, the binary point of the result is always to the right of the LSB.

The B and S prefixes denote byte- and short-word operand types respectively, and the d suffix denotes operand size—see [“Instruction Line Syntax and Structure” on page 1-20](#).

Status Flags

AZ	Set if all bits in result are zero
AN	Set to the most significant bit of result
AV (A0S)	Cleared (not changed)
AC	Cleared

Options

()	Signed integer
(U)	Unsigned integer

Examples

XR4 = SUM SR3:2;; *see Figure 8-10*

XR4 = SUM SR3:2 (U);; *see Figure 8-11*

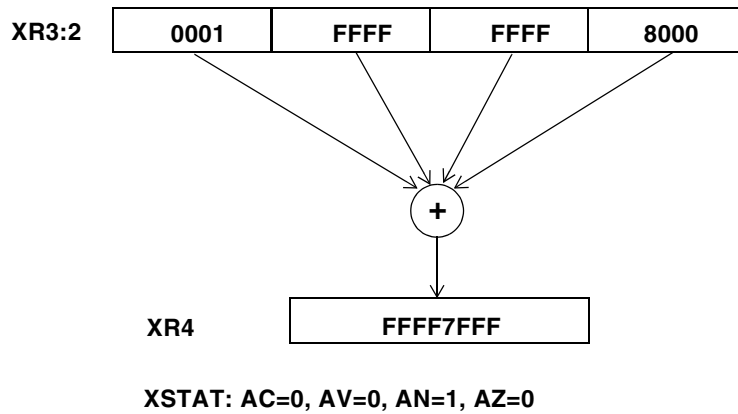


Figure 8-10. Signed SUM

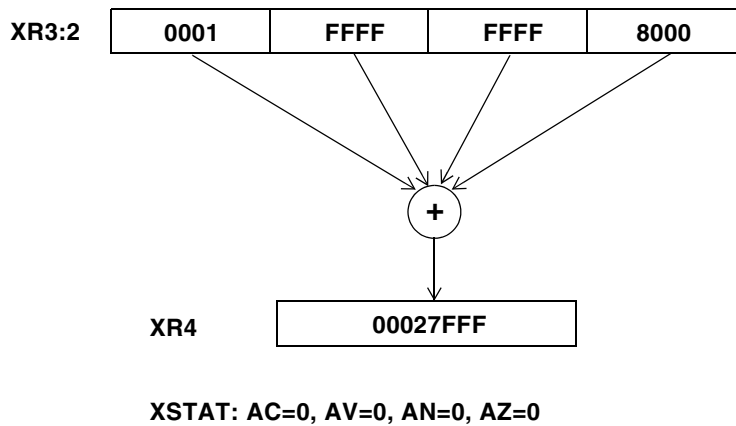


Figure 8-11. Unsigned SUM

ALU Instructions

Ones Counting

Syntax

$\{X|Y|XY\}Rs = \text{ONES } Rm|Rmd ;$

Function

This instruction counts the number of ones in the operand in register Rm . The result is placed in register Rs .

The d suffix denotes operand size—see [“Instruction Line Syntax and Structure” on page 1-20](#).

Status Flags

AZ	Set if all bits in result are zero
AN	Cleared
AV (AOS)	Cleared (not changed)
AC	Cleared

Example

```
R6 = ONES R3;;  
If R3 = b#00...011101  
then R6 = 0x00...04
```


Parallel Result Register

Syntax

$$\{X|Y|XY\}PR1:0 = Rmd ;$$

$$\{X|Y|XY\}Rsd = PR1:0 ;$$

Function

These instructions load register $PR1:0$ with the operand in Rmd or load register Rsd with the operand in $PR1:0$.

The d suffix denotes operand size—see [“Instruction Line Syntax and Structure” on page 1-20](#).

Status Flags

AZ	Set if all bits in result are zero
AN	Set to the most significant bit of result
AV (AOS)	Cleared (not changed)
AC	Cleared

ALU Instructions

Bit FIFO Increment

Syntax

$\{X|Y|XY\}Rs = \text{BFOINC } Rmd ;$

Function

This instruction adds the seven LSBs in each operand in dual register Rmd , divides them by 64 and returns the remainder to the six LSBs of the second operand, represented here by Rs .

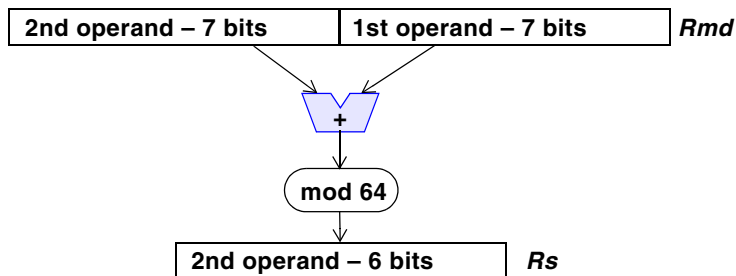


Figure 8-12. BFOINC Remainder

Status Flags

AZ	Set if all bits in result are zero
AN	Set when addition passes 63
AV (AOS)	Cleared (not changed)
AC	Cleared

Example

$\{X|Y|XY\}Rs = \text{BFOINC } Rmd;$

If $Rmd = 0x0\dots030 \ 0\dots020$

then $Rs = 0x0\dots010$ *and* AN *is set*

ALU Instructions

Parallel Absolute Value of Difference

Syntax

```
{X|Y|XY}PR0|PR1 += ABS (SRmd - SRnd){(U)} ;  
{X|Y|XY}PR0|PR1 += ABS (BRmd - BRnd){(U)} ;
```

Function

This instruction subtracts the bytes/shorts in register pair *Rnd* from those in register pair *Rmd*; performs a short-/byte-wise absolute value on the results; sums sideways these positive results and adds this single quantity to the contents of one of the PR registers. The final result is stored back into the PR register. The values in the PR registers are always right-justified—for example, the binary point is always to the right of the LSB. The values in the PR registers are also signed. Saturation is always active.

The S and B prefixes denote the operand type and the d suffix denotes operand size—see [“Instruction Line Syntax and Structure” on page 1-20](#).

Status Flags

The following flags are affected by the last addition into PR:

AZ	Set if all bits in result are zero
AN	Set to the most significant bit of final result
AV (AOS)	Set when signed result overflows; otherwise AV cleared
AC	Cleared

Options

()	Signed inputs
(U)	Unsigned inputs

Examples

XPR0 += ABS (SR3:2 - SR1:0);; see *Figure 8-13*

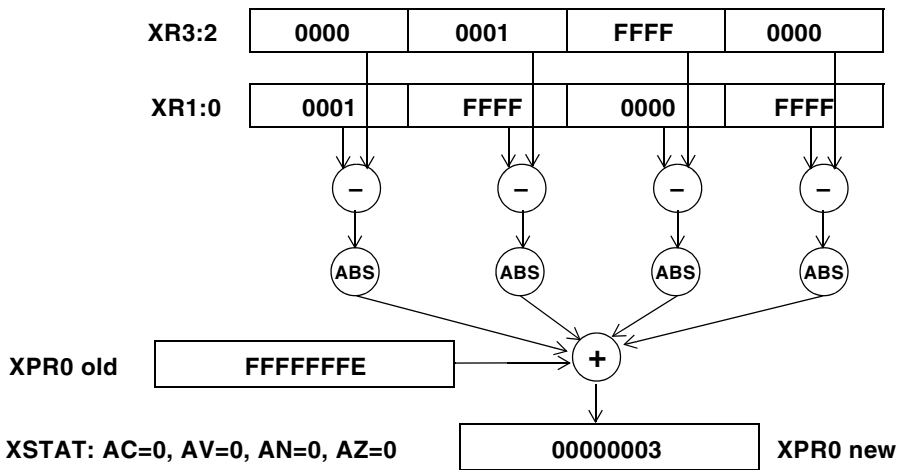


Figure 8-13. Parallel ABS

ALU Instructions

Sideways Sum

Syntax

```
{X|Y|XY}PR0|PR1 += SUM SRm {(U)} ;  
{X|Y|XY}PR0|PR1 += SUM SRmd {(U)} ;  
{X|Y|XY}PR0|PR1 += SUM BRm {(U)} ;  
{X|Y|XY}PR0|PR1 += SUM BRmd {(U)} ;
```

Function

This instruction performs a short- or byte-wise addition on the contents of *Rm* and adds this quantity to the contents of one of the *PR* registers. If the bytes/shorts are signed, they are sign extended before being added. The final result is stored back into the *PR* register. The values in the *PR* registers are always right-justified—for example, the binary point is always to the right of the LSB. The values in the *PR* registers can be signed or unsigned. The addition into *PR* is always saturating. The *S* and *B* prefixes denote the operand type and the *d* suffix denotes operand size—see [“Instruction Line Syntax and Structure” on page 1-20](#).

Status Flags

The following flags are affected by the last addition into *PR*:

AZ	Set if all bits in result are zero
AN	Set to the most significant bit of result
AV (AOS)	Signed overflow for normal, unsigned overflow if option (U) is set; otherwise AV cleared
AC	Cleared

Options

- () Signed inputs
- (U) Unsigned inputs and result

ALU Instructions

Add/Subtract (Dual Operation)

Syntax

$\{X|Y|XY\}\{S|B\}Rs = Rm + Rn, Ra = Rm - Rn ;$ (*dual operation*)
 $\{X|Y|XY\}\{L|S|B\}Rsd = Rmd + Rnd, Rad = Rmd - Rnd ;$ (*dual operation*)

Function

This instruction simultaneously adds and subtracts the operands in registers Rm and Rn . The results are placed in registers Rs and Ra . Saturation is always active and the result is signed.

The L, S, and B prefixes denote the operand type and d denotes operand size—see “[Instruction Line Syntax and Structure](#)” on page 1-20.

Status Flags

AZ	Set if all bits in one of the results are zero
AN	Set to the or between the most significant bit of two results
AV (AOS)	Signed overflow in one of the results
AC	Not cleared

Example

$R9 = R4 + R8, R2 = R4 - R8;;$

If $R4 = 8$ *and* $R8 = 2$

then $R9 = 10$ *and* $R2 = 6$

Pass

Syntax

```
{X|Y|XY}Rs = PASS Rm ;
{X|Y|XY}LRsd = PASS Rmd ;
```

Function

This instruction passes the operand in register *Rm* through the ALU to register *Rs*.

The L prefix denotes the operand type and d denotes operand size—see [“Instruction Line Syntax and Structure” on page 1-20](#).

Status Flags

AZ	Set if all bits in result are zero
AN	Set to the most significant bit <i>Rm</i>
AV (AOS)	Cleared (not changed)
AC	Cleared

Example

```
XR6 = PASS R3;;
If R3 = 0x80000000
then R6 = 0x80000000 and XSTAT: AC=0, AV=0, AN=1, AZ=0
```

ALU Instructions

Logical AND/AND NOT/OR/XOR/NOT

Syntax

$$\{X|Y|XY\}Rs = Rm \text{ AND|AND NOT|OR|XOR } Rn ;$$
$$\{X|Y|XY\}LRsd = Rmd \text{ AND|AND NOT|OR|XOR } Rnd ;$$
$$\{X|Y|XY\}Rs = \text{NOT } Rm ;$$
$$\{X|Y|XY\}LRsd = \text{NOT } Rmd ;$$

Function

These instructions logically AND, AND NOT, OR, or XOR the operands in registers Rm and Rn . The NOT instruction logically complements the operand in the Rm register. The result is placed in register Rs .

The L prefix denotes the operand type and d denotes operand size—see [“Instruction Line Syntax and Structure”](#) on page 1-20.

Status Flags

AZ	Set if all bits in result are zero
AN	Set to the most significant bit of result
AV (AOS)	Cleared (not changed)
AC	Cleared

Example

R5 = R4 AND R8;;

If R4 = b#...1001 *and* R8 = b#...1100
then R5 = b#...1000

R5 = R4 AND NOT R8;;

If R4 = b#...1001 *and* R8 = b#...1100
then R5 = b#...0001

R5 = R3 OR R4;;

If R3 = b#...1001 *and* R4 = b#...1100
then R5 = b#...1101

R3 = R2 XOR R7;;

If R2 = b#...1001 *and* R7 = b#...1100
then R3 = b#...0101

R6 = NOT R3;;

If R3 = b#00...011101
then R6 = b#11...100010



The b# prefix denotes binary.

ALU Instructions

Expand

Syntax

```
{X|Y|XY}Rsd = EXPAND SRm {+|- SRn} {{{I|IU}}} ;  
{X|Y|XY}Rsq = EXPAND SRmd {+|- SRnd} {{{I|IU}}} ;  
{X|Y|XY}Rsd = EXPAND BRm {+|- BRn} {{{I|IU}}} ;  
{X|Y|XY}Rsq = EXPAND BRmd {+|- BRnd} {{{I|IU}}} ;
```

Function

These instructions add or subtract the operands in the *Rm* and *Rn* registers then cast the results, expanding 8-bit values to 16-bit values or 16-bit values to 32-bit values. The *Rn* operand may be omitted, performing the expand on *Rm*. If the format is fractional, the DSP appends zeros at the end. If the type is integer, the DSP prepends zeros (or ones if sign-extending a signed integer) at the beginning. These instructions expand the result in the following manner:

- *Rsd*=EXPAND *SRm* expands two shorts to two normals
- *Rsq*=EXPAND *SRmd* expands four shorts to four normals
- *RsdS*=EXPAND *BRm* expands four bytes to four shorts
- *RsqS*=EXPAND *BRmd* expands eight bytes to eight shorts

The result is placed in the fixed-point register *Rs*.

The **S** prefix denotes short-word operand type and the **B** suffix denotes byte-word operands. The suffices **d** and **q** denote operand size—see [“Instruction Line Syntax and Structure” on page 1-20](#).

Status Flags

AZ	Set if all bits in result are zero
AN	Set to the most significant bit of result

AV (AOS) Cleared (not changed)

AC Cleared

Options

() Fractional

(I) Signed integer

(IU) Unsigned integer

Examples

$\{X|Y|XY\}Rsd = \text{EXPAND } SRm$; see *Figure 8-14*

$\{X|Y|XY\}Rsq = \text{EXPAND } SRmd$;

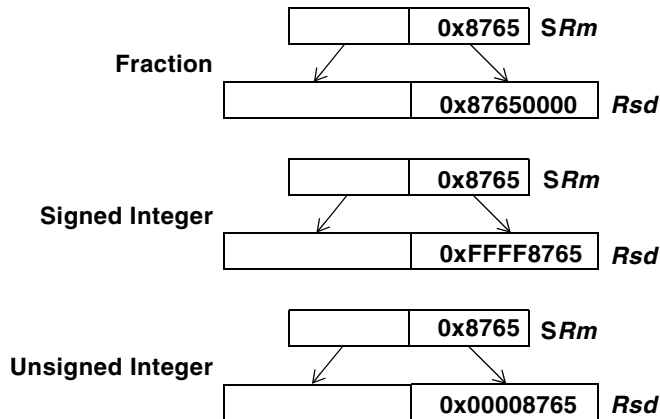


Figure 8-14. Fractional, Signed Integer, and Unsigned Integer EXPAND

ALU Instructions

$\{X|Y|XY\}SRsd = \text{EXPAND } BRm$; see *Figure 8-15*

$\{X|Y|XY\}SRsq = \text{EXPAND } BRmd$;

**Byte to short expand
in fraction, signed int,
and unsigned int modes**

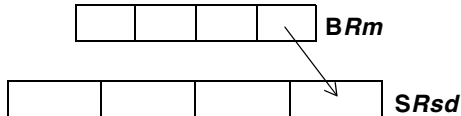


Figure 8-15. Expand Byte Words to Short Words

$\{X|Y|XY\}Rsd = \text{EXPAND } SRm + SRn$;

$\{X|Y|XY\}Rsq = \text{EXPAND } SRmd + SRnd$; see *Figure 8-16*

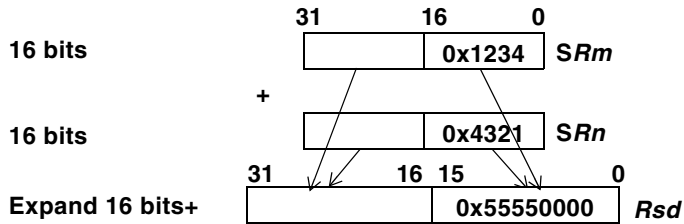


Figure 8-16. Expand Short Words to Normal Words

$\{X|Y|XY\}SRsd = \text{EXPAND } BRm + BRm;$
 $\{X|Y|XY\}SRsq = \text{EXPAND } BRmd + BRmd;$ *see Figure 8-17*

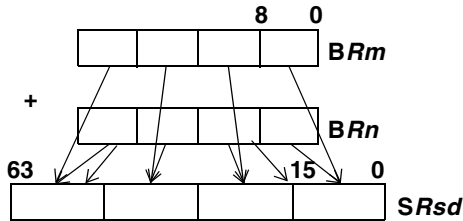


Figure 8-17. Add and Expand Byte Words to Short Words

$\{X|Y|XY\}Rsd = \text{EXPAND } SRm - SRn;$ *see Figure 8-18*
 $\{X|Y|XY\}Rsq = \text{EXPAND } SRmd - SRnd;$

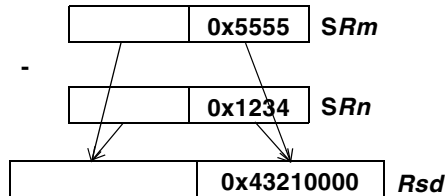


Figure 8-18. Subtract and Expand Short Words to Normal Words

ALU Instructions

$\{X|Y|XY\}SRsd = \text{EXPAND } BRm - BRm$; see *Figure 8-19*

$\{X|Y|XY\}SRsq = \text{EXPAND } BRmd - BRmd$;

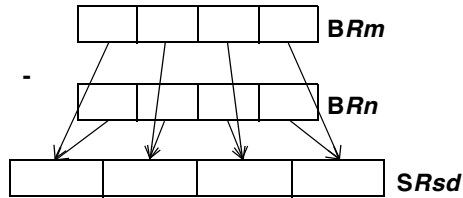


Figure 8-19. Subtract and Expand Byte Words to Short Words

Compact

Syntax

```
{X|Y|XY}SRs = COMPACT Rmd {+|- Rnd} {(T|I|IS|ISU)} ;
{X|Y|XY}BRs = COMPACT SRmd {+|- SRnd} {(T|I|IS|ISU)} ;
```

Function

These instructions add or subtract the operands in the *Rmd* and *Rnd* registers, compact the operands in source register pair *Rmd* into a data type of lower precision, and place the result in destination register *Rs*. The *Rnd* operand may be omitted, performing the compact on *Rm*. Compaction is performed either from two normal words into two shorts, or from four shorts into four bytes.

Two data types are supported—fractional (default) or integer—using options (I), (IS) or (ISU).

Fractional compact transfers the upper half of the result into the destination register, and either rounds it to nearest even (default) or truncates the lower half—option (T). Overflow may occur only when rounding up the maximum positive number, and in this case the result is saturated (assuming that the data is signed).

Integer compaction transfers the lower half of the result to the destination register. When compacting an integer, saturation can be selected as an option. When saturation is not selected (default), the upper bits of the input operands are used only for overflow decisions—for signed. If at least one of the upper bits is different from the result MSB (sign bit), the overflow bit is set. If saturation is enabled on signed data—option (IS), in case of overflow (same detection as in option (I)), the result is 0x7FF...F for a positive input operand, and 0x800...0 for a negative input operand. If saturation is unsigned—option (ISU) is set—the overflow criteria are that not all upper bits are zero, and the saturated result is always 0xFF...F.

ALU Instructions

The *S* and *B* prefixes denote short-word and byte-word operand types, respectively, and the *d* and *q* suffices denote operand size—see “[Instruction Line Syntax and Structure](#)” on page 1-20.

Status Flags

AZ	Set if all bits in result are zero
AN	Set to the most significant bit of result
AV (AOS)	Set when result is larger than maximum compact (or for negative results on option (ISU)); otherwise AV cleared
AC	Cleared

Options

()	Fraction round
(T)	Fraction truncated
(I)	Integer signed, no saturation
(IS)	Signed integer, saturation
(ISU)	Unsigned integer, saturation

Examples

$\{X|Y|XY\}SRs = \text{COMPACT } Rmd$; see *Figure 8-20*

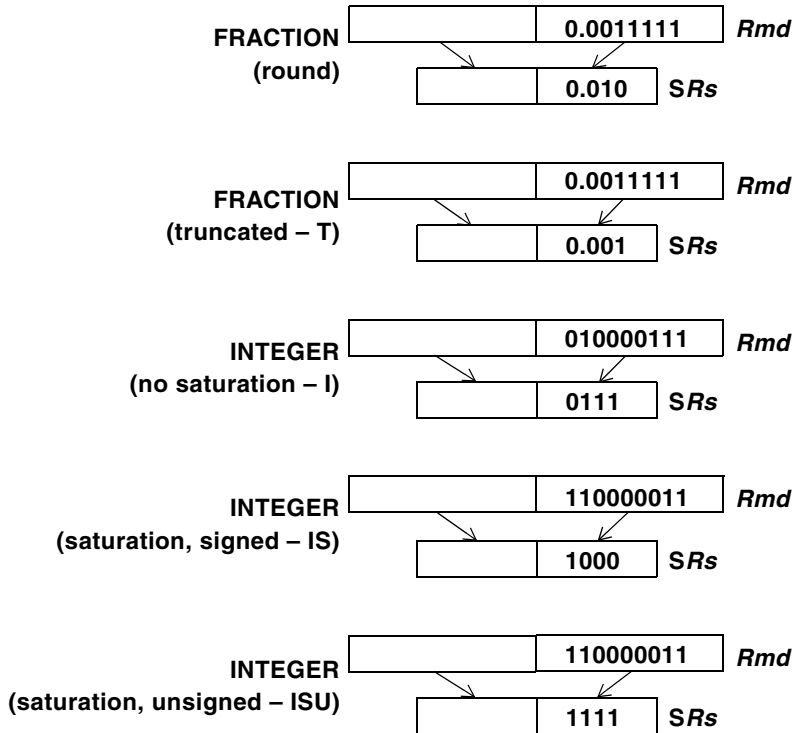


Figure 8-20. COMPACT Instruction Options

ALU Instructions

$\{X|Y|XY\}BRs = \text{COMPACT } SRmd$; see *Figure 8-21*

**Short to byte compact
in fraction round,
fraction trunc, int no sat,
signed int and unsigned
int modes**



Figure 8-21. Compact Short Words to Byte Words

$\{X|Y|XY\}SRs = \text{COMPACT } Rmd + Rnd$; see *Figure 8-22*

$\{X|Y|XY\}BRsq = \text{COMPACT } SRmd + SRnd$;

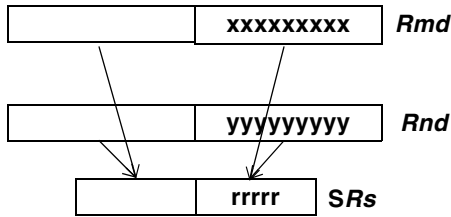


Figure 8-22. Add and Compact Normal Words to Short Words

$\{X|Y|XY\}BRs = \text{COMPACT } SRmd + SRnd$; see *Figure 8-23*

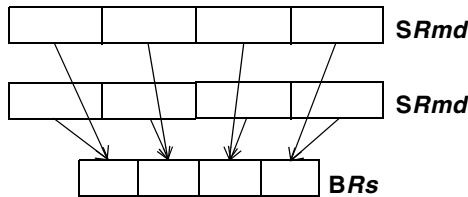


Figure 8-23. Add and Compact Short Words to Byte Words

Merge

Syntax

```
{X|Y|XY}BRsd = MERGE Rm, Rn ;
{X|Y|XY}BRsq = MERGE Rmd, Rnd ;
{X|Y|XY}SRsd = MERGE Rm, Rn ;
{X|Y|XY}SRsq = MERGE Rmd, Rnd ;
```

Function

This instruction merges (transposes) the operands in registers *Rm* and *Rn*. The result is placed in register *Rs*.

The B and S prefixes denote byte- and short-word operand types respectively, and the suffices d and q denote operand size—see [“Instruction Line Syntax and Structure” on page 1-20](#).

Status Flags

AZ	Set if all bits in result are zero
AN	Set to the most significant bit of result
AV (AOS)	Cleared (not changed)
AC	Cleared

ALU Instructions

Example

XBR3:2 = MERGE R0, R1;;

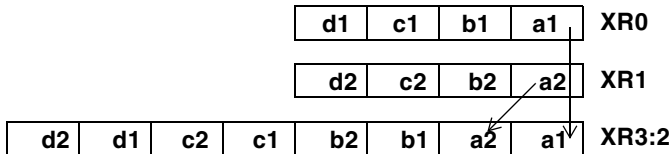


Figure 8-24. Merge Function Data Flow

Instruction `MERGE` may be used to transpose shorts and bytes by repeatedly merging the results.

Add/Subtract (Floating-Point)

Syntax

$$\{X|Y|XY\}FRs = Rm +|- Rn \{(T)\} ;$$

$$\{X|Y|XY\}FRsd = Rmd +|- Rnd \{(T)\} ;$$

Function

These instructions add or subtract the floating-point operands in registers *Rm* and *Rn*. The normalized result is placed in register *FRs*. The *d* suffix denotes extended operand size—see [“Instruction Line Syntax and Structure” on page 1-20](#).

Rounding is to nearest (IEEE), or by truncation, to a 32-bit or 40-bit boundary as defined by the *(T)* option. Post-rounded overflow returns \pm infinity (round-to-nearest) or \pm NORM.MAX¹ (round-to-zero). Post-rounded denormal returns \pm zero. Denormal inputs are flushed to \pm zero. A NAN input returns as all ones result.

Status Flags

AZ	Set if post-rounded result is denormal (unbiased exponent ≤ 126) or \pm zero
AUS	Set if post-rounded result is denormal; otherwise not cleared ²
AN	Set if result is negative
AV	Set if post-rounded result overflows

¹ Maximum normal value — mantissa: all 1 (0x7FFFFFFF), exponent: FE Bit 31 is the sign bit; bits 30–23 are the eight exponent bits biased by 127 (thus, 127 must be subtracted from the unsigned value given by the bits 30–23 to obtain the actual exponent); bits 22–0 are the fractional part of the mantissa bits (1.0 is always assumed to be the fixed part of the mantissa, thus 1. (Bits 22–0 makes the actual mantissa). For more information, see [“Numeric Formats” on page 2-16](#).

² See [“ALU Execution Status” on page 3-11](#).

ALU Instructions

AVS	Set if post-rounded result overflows; otherwise not cleared ¹
AC	Cleared
AI	Set if either input operand is a NAN, or if they are opposite (on add) or same sign (on subtract) infinities
AIS	Set if either input operand is a NAN, or if they are opposite (on add) or same sign (on subtract) infinities; otherwise not cleared ²

Options

()	Round
(T)	Truncate

Examples

```
YFR0 = R1 + R2 (T) ;; /* Y compute block with truncation */  
FR0 = R1 + R2 ;; /* SIMD with rounding (by default) */  
XFR1:0 = R3:2 + R5:4 ;; /*Extended precision 40-bit float-  
ing-point add */
```

```
XFR0 = R1 - R2 ;; /* X compute block with rounding (by default) */  
FR0 = R1 - R2 (T) ;; /* SIMD with truncation */  
XFR1:0 = R3:2 - R5:4 ;; /* Extended precision 40-bit float-  
ing-point subtract */
```

¹ See “ALU Execution Status” on page 3-11.

² See “ALU Execution Status” on page 3-11.

Average (Floating-Point)

Syntax

$$\{X|Y|XY\}FRs = (Rm +|- Rn)/2 \{(T)\} ;$$

$$\{X|Y|XY\}FRsd = (Rmd +|- Rnd)/2 \{(T)\} ;$$

Function

These instructions add or subtract the floating-point operands in registers Rm and Rn , then divide the result by two, decrementing the exponent of the sum before rounding. The normalized result is placed in register FRs . The d suffix denotes extended operand size—see “[Instruction Line Syntax and Structure](#)” on page 1-20.

Rounding is to nearest (IEEE), or by truncation, to a 32-bit or 40-bit boundary as defined by the (T) option. Post-rounded overflow returns \pm infinity (round-to-nearest) or \pm NORM.MAX¹ (round-to-zero). Post-rounded denormal returns \pm zero. Denormal inputs are flushed to \pm zero. A NAN input returns as all ones result.

Status Flags

AZ	Set if post-rounded result is denormal (unbiased exponent ≤ 126) or \pm zero
AUS	Set if post-rounded result is denormal; otherwise not cleared ²
AN	Set if result is negative
AC	Cleared
AV	Set if post-rounded result overflows

¹ Maximum normal value - mantissa - all 1 (0x7F...0), exponent - 0x1E

² Maximum normal value - mantissa - all 1 (0x7F...0), exponent - 0x1E

ALU Instructions

AVS	Set if post-rounded result overflows; otherwise not cleared ¹
AI	Set if either input operand is a NAN, or if they are opposite (on add) or same sign (on subtract) infinities
AIS	Set if either input operand is a NAN, or if they are opposite (on add) or same sign (on subtract) infinities; otherwise not cleared ²

Options

()	Round
(T)	Truncate

Examples

```
XFR0 = (R1 + R2)/2 ;; /* X compute block with rounding (by
default) */
FR0 = (R1 + R2)/2 (T) ;; /* SIMD with truncation */
XFR1:0 = (R3:2 + R5:4)/2 ;; /* Extended precision 40-bit float-
ing-point add divide by 2 */

YFR0 = (R1 - R2)/2 (T) ;; /* Y compute block with truncation */
FR0 = (R1 - R2)/2 ;; /* SIMD with rounding (by default) */
XFR1:0 = (R3:2 - R5:4)/2;; /* Extended precision 40-bit float-
ing-point subtract divide by 2 */
```

¹ See “ALU Execution Status” on page 3-11

² See “ALU Execution Status” on page 3-11

Maximum/Minimum (Floating-Point)

Syntax

$$\{X|Y|XY\}_{FRS} = \text{MAX}|\text{MIN} (Rm +|- Rn) \{(T)\} ;$$

$$\{X|Y|XY\}_{FRSd} = \text{MAX}|\text{MIN} (Rmd +|- Rnd) \{(T)\} ;$$

Function

These instructions return the maximum (larger of) or minimum (smaller of) the floating-point operands in registers Rm and Rn . The result is placed in register FRS . The d suffix denotes extended operand size—see “[Instruction Line Syntax and Structure](#)” on page 1-20.

A NAN input returns a floating-point all ones result. MAX of (+zero and -zero) returns +zero. MIN of (+zero and -zero) returns -zero. Denormal inputs are flushed to \pm zero.

Status Flags

AZ	Set if result is \pm zero
AUS	Not cleared ¹
AN	Set if result is negative
AV	Cleared
AVS	Not cleared ² ; retains value from previous event
AC	Cleared
AI	Set if either input is a NAN
AIS	Set if either input is a NAN; otherwise not cleared ³

¹ See “[ALU Execution Status](#)” on page 3-11.

² See “[ALU Execution Status](#)” on page 3-11.

³ See “[ALU Execution Status](#)” on page 3-11.

ALU Instructions

Options

- () Round (MIN only)
- (T) Truncate (MIN only)

Examples

```
XFR0 = MAX (R1, R2) (T) ;; /* x compute with truncate option */  
YFR1:0 = MAX (R3:2, R5:4) ;; /* y compute block with extended  
40-bit precision with rounding (by default) */
```

```
XFR0 = MIN (R1, R2) (T) ;; /* x compute with truncate option */  
YFR1:0 = MIN (R3:2, R5:4) ;; /* y compute block with extended  
40-bit precision with rounding (by default) */
```

Absolute Value (Floating-Point)

Syntax

```
{X|Y|XY}FRs = ABS (Rm) ;
{X|Y|XY}FRsd = ABS (Rmd) ;
{X|Y|XY}FRs = ABS (Rm +|- Rn) {(T)} ;
{X|Y|XY}FRsd = ABS (Rmd +|- Rnd) {(T)} ;
```

Function

These instructions add or subtract the floating-point operands in registers *Rm* and *Rn*, then place the absolute value of the normalized result in register *FRs*. The *Rn* operand may be omitted, performing an absolute value on *Rm*. The *d* suffix denotes extended operand size—see “[Instruction Line Syntax and Structure](#)” on page 1-20.

Rounding is to nearest (IEEE), or by truncation, to a 32-bit or 40-bit boundary as defined by the (T) option. Post-rounded overflow returns +infinity (round-to-nearest) or +NORM.MAX¹ (round-to-zero). Post-rounded denormal returns +zero. Denormal inputs are flushed to +zero. A NAN input returns as all ones result.

Status Flags

AZ	Set if operand (post-rounded result for add or subtract) is denormal (unbiased exponent <=126) or ±zero
AUS	Set if operand (post-rounded result for add or subtract) is denormal; otherwise not cleared ²
AN	Set if result of addition operation prior to ABS is negative

¹ Maximum normal value - mantissa - all 1 (0x7F...0), exponent - 0x1E

² Maximum normal value - mantissa - all 1 (0x7F...0), exponent - 0x1E

ALU Instructions

AV	Set if operand (post-rounded result for add or subtract) overflows
AVS	Set if operand (post-rounded result for add or subtract) overflows; otherwise not cleared ¹
AC	Cleared
AI	Set if either input operand is a NAN, or if they are opposite (on add) or same sign (on subtract) infinities
AIS	Set if either input operand is a NAN, or if they are opposite (on add) or same sign (on subtract) infinities; otherwise not cleared ²
Options	
()	Round
(T)	Truncate

¹ See “ALU Execution Status” on page 3-11.

² See “ALU Execution Status” on page 3-11.

Examples

```
FRO = ABS R1 ;; /* SIMD, abs of xr1 and yr1 */
XFR1:0 = ABS R1:0 ;; /* Extended 40 bit precision

XFR0 = ABS(R1 + R2) ;; /* x compute block with rounding (by
default) */
FRO = ABS(R1 + R2) (T) ;; /* SIMD with truncation */
XFR1:0 = ABS(R3:2 + R5:4) ;; /* Extended precision 40-bit float-
ing-point absolute value of add */

XFR0 = ABS(R1 - R2) ;; /* x compute block with rounding (by
default) */
FRO = ABS(R1 - R2) (T) ;; /* SIMD with truncation */
XFR1:0 = ABS(R3:2 - R5:4) ;; /* Extended precision 40-bit float-
ing-point absolute value of subtract */
```

ALU Instructions

Negate (Floating-Point)

Syntax

$\{X|Y|XY\}FRs = - Rm ;$

$\{X|Y|XY\}FRsd = - Rmd ;$

Function

This instruction complements the sign bit of the floating-point operand in register Rm . The complemented result is placed in register FRs . The d suffix denotes extended operand size—see “[Instruction Line Syntax and Structure](#)” on page 1-20.

Denormal inputs are flushed to \pm zero (sign is the inverse of Rm 's sign). A NAN input returns an all ones result.

Status Flags

AZ	Set if result is \pm zero or denormal
AUS	Not cleared ¹
AN	Set if result is negative
AV	Cleared
AVS	Not cleared ²
AC	Cleared
AI	Set if input is NAN
AIS	Set if input is NAN; otherwise not cleared ³

¹ See “[ALU Execution Status](#)” on page 3-11.

² See “[ALU Execution Status](#)” on page 3-11.

³ See “[ALU Execution Status](#)” on page 3-11.

Examples

```
XFR0 = -R1 ;; /* x compute block complement */  
YFR1:0 = -R3:2;; /* extended 40-bit precision */
```

ALU Instructions

Compare (Floating-Point)

Syntax

```
{X|Y|XY}FCOMP (Rm, Rn) ;  
{X|Y|XY}FCOMP (Rmd, Rnd) ;
```

Function

This instruction compares the floating-point operand in register *Rm* with the floating-point operand in register *Rn*. Denormal inputs are flushed to zero. The *d* suffix denotes extended operand size—see “[Instruction Line Syntax and Structure](#)” on page 1-20.

This instruction sets the *AZ* flag if the two operands are equal and the *AN* flag if the operand in register *Rm* is smaller than the operand in register *Rn*.

Status Flags

<i>AZ</i>	Set if $R_m = R_n$ and neither R_m or R_n are NaNs
<i>AUS</i>	Not cleared ¹
<i>AN</i>	Set if $R_m < R_n$ and neither R_m or R_n are NaNs
<i>AV</i>	Cleared
<i>AVS</i>	Not cleared ²
<i>AC</i>	Cleared
<i>AI</i>	Set if either input is a NaN
<i>AIS</i>	Set if either input is a NaN; otherwise not cleared ³

¹ See “[ALU Execution Status](#)” on page 3-11.

² See “[ALU Execution Status](#)” on page 3-11.

³ See “[ALU Execution Status](#)” on page 3-11.

For IEEE compatibility: After using `FCOMP` the programmer may use conditions `ALT` (ALU result is less than) or `ALE` (ALU result is less than or equal) – see “[ALU Execution Conditions](#)” on page 3-14. In all these cases the condition will be false if one of the operands is NAN. In some cases the inverse condition is also expected to give false result if one of the operands is NAN. In this case the programmer should *not* use the inverse condition (`NALT` or `NALE`). Instead, the programmer should flip between the operands, and use the condition `ALT` instead of `NALE` or `ALE` instead of `NALT`.

Example

```
FCOMP (R5,R8) ;;
IF ALT, JUMP my_label ;;
```

The condition is true if $R5 < R8$. If they are equal, $R5 > R8$ or one of the numbers is NAN the condition is false.

Using the condition `NALT` will give a false result if $R5 < R8$. In any other case, including the case of NAN input, the condition will be true. To get the same result, but false results in case of NAN input, switch between the operands of the `FCOMP` and change the `ALT` to `ALE` (or vice versa):

```
FCOMP (R8,R5) ;;
IF ALE, JUMP my_other_label ;;
```

ALU Instructions

Floating- to Fixed-Point Conversion

Syntax

$\{X|Y|XY\}Rs = \text{FIX } FRm|FRmd \{BY Rn\} \{(T)\} ;$

Function

These instructions add the two's-complement operand in register Rn to the exponent of floating-point operand in register Rm , then convert the result to a two's-complement 32-bit fixed-point integer result. If the Rn operand is omitted, Rm is converted. The floating-point operand is rounded to the nearest even integer. The result is placed in register Rs . The d suffix denotes extended operand size—see [“Instruction Line Syntax and Structure” on page 1-20](#).

Rounding is to nearest (IEEE) or to zero, as defined by the (T) option. A NAN input returns a floating-point all ones result. All underflow results, or input which are zero or denormal, return zero. Overflow result always returns a signed saturated result— $0x7FFFFFFF$ for positive, and $0x80000000$ for negative.

Status Flags

AZ	Set if fixed-point result is zero
AUS	Set if pre-rounded result absolute value is less than 0.5 and not zero; otherwise not cleared ¹
AN	Set if fixed-point result is negative
AV	Set if the exponent of the floating-point result is larger or equal to 157; unless the operand is negative, the sum is equal to 157, and the mantissa is all zero

¹ See [“ALU Execution Status” on page 3-11](#).

AVS	Set if the exponent of the floating-point result is larger or equal to 157; unless the operand is negative, the sum is equal to 157, and the mantissa is all zero; otherwise not cleared ¹
AC	Cleared
AI	Set if input is a NAN
AIS	Set if input is a NAN; otherwise not cleared ²

Options

()	Round
(T)	Truncate

Examples

```
YR0 = FIX YFR1 (T) ;; /* y compute block, truncated option */
XR0 = FIX XFR3:2 ;; /* Extended 40-bit precision with rounding
(by default) */
```

```
YR0 = FIX YFR1 BY R2 (T) ;; /* Y compute block, truncated option
*/
XR0 = FIX XFR3:2 BY R4 ;; /* Extended 40-bit precision with
rounding (by default) */
```

¹ See “ALU Execution Status” on page 3-11.

² See “ALU Execution Status” on page 3-11.

ALU Instructions

Fixed- to Floating-Point Conversion

Syntax

$\{X|Y|XY\}FRs|FRsd = \text{FLOAT } Rm \{BY Rn\} \{(T)\} ;$

Function

These instructions convert the fixed-point operand in Rm to a floating-point result. If used, Rn denotes the scaling factor, where the fixed-point two's-complement integer in Rn is added to the exponent of the floating-point result. The final result is placed in register FRs . The d suffix denotes extended result size—see “[Instruction Line Syntax and Structure](#)” on page 1-20.

Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the (T) option. The exponent scale bias may cause a floating-point overflow or a floating-point underflow: overflow returns $\pm\text{infinity}$ (round-to-nearest) or $\pm\text{NORM.MAX}^1$ (round-to-zero); underflow returns $\pm\text{zero}$.

Status Flags

AZ	For not scaled, set if post-rounded result is $\pm\text{zero}$; for scaled, set if post-rounded result is denormal (unbiased exponent ≤ 126) or $\pm\text{zero}$
AUS	For not scaled, not cleared; for scaled, set if post-rounded result is denormal—otherwise not cleared ²
AN	Set if result is negative
AV	For not scaled, cleared; for scaled, set if post-rounded result overflows

¹ Maximum normal value - mantissa - all 1 (0x7F...0), exponent - 0x1E

² Maximum normal value - mantissa - all 1 (0x7F...0), exponent - 0x1E

AVS	For not scaled, not cleared; for scaled, set if post-rounded result overflows; otherwise not cleared ¹
AC	Cleared
AI	Cleared
AIS	Not cleared ²

Options

()	Round
(T)	Truncate

Examples

```
XFR0 = FLOAT R1 (T) ;; /* x compute block, truncated */  
YFR1:0 = FLOAT R1 ;; /* Extended 40-bit precision with rounding  
(by default) */
```

```
XFR0 = FLOAT R1 BY R2 (T) ;; /* x compute block, truncated */  
YFR1:0 = FLOAT R1 BY R2 ;; /* Extended 40-bit precision with  
rounding (by default) */
```

¹ See “ALU Execution Status” on page 3-11.

² See “ALU Execution Status” on page 3-11.

ALU Instructions

Floating-Point Normal to Extended Word Conversion

Syntax

$\{X|Y|XY\}FRsd = EXT D Rm ;$

Function

This instruction extends the floating-point operand in register Rm . The extended result is placed in register $FRsd$. The d suffix denotes extended operand size—see “[Instruction Line Syntax and Structure](#)” on page 1-20.

Denormal inputs are flushed to \pm zero. A NAN input returns an all ones result.

Status Flags

AZ	Set if result is \pm zero or denormal
AUS	Not cleared ¹
AN	Set if result is negative
AV	Cleared
AVS	Not cleared ²
AC	Cleared
AI	Set if input is NAN
AIS	Set if input is NAN; otherwise not cleared ³

¹ See “[ALU Execution Status](#)” on page 3-11.

² See “[ALU Execution Status](#)” on page 3-11.

³ See “[ALU Execution Status](#)” on page 3-11.

Examples

```
XFR1:0 = EXT D R2 ;; /* x compute block extend to 40 bit */
```

ALU Instructions

Floating-Point Extended to Normal Word Conversion

Syntax

$\{X|Y|XY\}FRs = SNGL Rmd ;$

Function

This instruction translates the extended floating-point operand in (dual) register *Rmd* into a single floating-point operand. The result is placed in register *FRs*. The *d* suffix denotes dual operand size—see [“Instruction Line Syntax and Structure” on page 1-20](#).

Rounding is to nearest (IEEE) or by truncation, as defined by the (T) option. A NAN input returns a floating-point all ones result. Overflow returns $\pm NORM.MAX$ ¹ if (T) option is set, and \pm infinity otherwise.

Status Flags

AZ	Set if result is \pm zero
AUS	Not cleared ²
AN	Set if result is negative
AV	Set if the exponent is 0xFE and the mantissa is more than 0x7FFFFFF80 and option (T) is not used
AVS	Set if the exponent is 0xFE and the mantissa is more than 0x7FFFFFF80, otherwise not cleared ³
AC	Cleared
AI	Set if input is a NAN

¹ Maximum normal value - mantissa - all 1 (0x7F...0), exponent - 0x1E

² See [“ALU Execution Status” on page 3-11](#).

³ See [“ALU Execution Status” on page 3-11](#).

AIS Set if input is a NAN; otherwise not cleared¹

Options

() Round

(T) Truncate

Examples

```
XFR0 = SNGL R3:2 (T) ;; /* x compute block with truncate option
*/
```

¹ See “[ALU Execution Status](#)” on page 3-11.

ALU Instructions

Clip (Floating-Point)

Syntax

$\{X|Y|XY\}FRs = CLIP Rm BY Rn ;$
 $\{X|Y|XY\}FRsd = CLIP Rmd BY Rnd ;$

Function

This instruction returns the floating-point operand in Rm if the absolute value of the operand in Rm is less than the absolute value of the floating-point operand in Rn . Else, returns $|Rn|$ if Rm is positive, and $-|Rn|$ if Rm is negative. The result is placed in register FRs . The d suffix denotes extended operand size—see [“Instruction Line Syntax and Structure” on page 1-20](#).

A NAN input returns an all ones result. Denormal inputs are flushed to $\pm zero$.

Status Flags

AZ	Set if result is $\pm zero$ or denormal
AUS	Not cleared ¹
AN	Set if result is negative
AV	Cleared
AVS	Not cleared ²
AC	Cleared
AI	Set if either input operand is a NAN

¹ See [“ALU Execution Status” on page 3-11](#).

² See [“ALU Execution Status” on page 3-11](#).

AIS Set if either input operand is a NAN; otherwise not cleared¹

Examples

FR3 = CLIP R9 BY R0;

	Compute X	Compute Y
FR9	995	-35
FR0	-57.3	89
FR3	57.3	-35

XSTAT: AC=0, AV=0, AN=0, AZ=0

YSTAT: AC=0, AV=0, AN=1, AZ=0

Figure 8-25. Dual Normal Word in Compute Blocks X and Y

```
XFR0 = CLIP R1 BY R2 ;; /* X compute block */
YFR1:0 = CLIP R5:4 BY R3:2 ;; /* Y compute block, extended 40-bit
precision */
```

¹ See “ALU Execution Status” on page 3-11.

ALU Instructions

Copysign (Floating-Point)

Syntax

```
{X|Y|XY}FRs = Rm COPYSIGN Rn ;  
{X|Y|XY}FRsd = Rmd COPYSIGN Rnd ;
```

Function

This instruction copies the sign of the floating-point operand in register *Rn* to the floating-point operand from register *Rm* without changing the exponent or the mantissa. The result is placed in register *FRs*. The *d* suffix denotes extended operand size—see [“Instruction Line Syntax and Structure”](#) on page 1-20.

A NAN input returns an all ones result. Denormal *Rm* returns zero, with the sign copied from *Rn*.

Status Flags

AZ	Set if result is \pm zero or denormal
AUS	Not cleared ¹
AN	Set if result is negative
AV	Cleared
AVS	Not cleared ²
AC	Cleared
AI	Set if either input operand is a NAN

¹ See [“ALU Execution Status”](#) on page 3-11.

² See [“ALU Execution Status”](#) on page 3-11.

AIS Set if either input operand is a NAN; otherwise not cleared¹

Examples

```
YFRO = R1 COPYSIGN R2 ;; /* Y compute block */  
XFR1:0 = R3:2 COPYSIGN R5:4 ;; /* X compute block, extended  
40-bit precision */
```

¹ See “ALU Execution Status” on page 3-11.

ALU Instructions

Scale (Floating-Point)

Syntax

$\{X|Y|XY\}FRs = \text{SCALB } FRm \text{ BY } Rn ;$
 $\{X|Y|XY\}FRsd = \text{SCALB } FRmd \text{ BY } Rn ;$

Function

This instruction scales the exponent of the floating-point operand in Rm by adding to it the fixed-point, two's-complement integer in Rn . The scaled floating-point result is placed in register FRs . The d suffix denotes extended operand size—see “[Instruction Line Syntax and Structure](#)” on [page 1-20](#). If the sum of FRm exponent and Rn is zero, the result is rounded to nearest: minimum normal if the FRm mantissa is all ones; else zero.

Overflow returns \pm infinity. Denormal returns \pm zero. Denormal inputs are flushed to \pm zero. A NAN input returns an all ones result.

Status Flags

AZ	Set if post-rounded result is denormal (unbiased exponent ≤ 126) or \pm zero
AUS	Set if post-rounded result is denormal; otherwise not cleared ¹
AN	Set if result is negative
AV	Set if post-rounded result overflows
AVS	Set if post-rounded result overflows; otherwise not cleared ²
AC	Cleared

¹ See “[ALU Execution Status](#)” on [page 3-11](#).

² See “[ALU Execution Status](#)” on [page 3-11](#).

AI	Set if either input operand is a NAN
AIS	Set if either input operand is a NAN; otherwise not cleared ¹

Examples

FR5 = scalb R3 by R2

If R3 = 5 (*floating-point value; $5=(2^3 \times 0.625)$*)

and R2 = 2 (*fixed-point value*)

then R5 = 20 (*floating-point value; $20=(2^{(3+2)} \times 0.625)$*)

```
YFR1:0 SCALB FR3:R2 BY R5 ;; /* 40-bit extended precision */
```

¹ See “ALU Execution Status” on page 3-11.

ALU Instructions

Pass (Floating-Point)

Syntax

$\{X|Y|XY\}FRs = PASS Rm ;$
 $\{X|Y|XY\}FRsd = PASS Rmd ;$

Function

This instruction passes the floating-point operand in register Rm through the ALU to the floating-point field in register FRs . The d suffix denotes extended operand size—see [“Instruction Line Syntax and Structure” on page 1-20](#).

Denormal inputs are flushed to $\pm zero$. A NAN input returns an all ones result.

Status Flags

AZ	Set if result is $\pm zero$ or denormal
AUS	Not cleared ¹
AN	Set if result is negative
AV	Cleared;
AVS	Not cleared; retains value from previous event ²
AC	Cleared
AI	Set if input is NAN
AIS	Set if input is NAN; otherwise not cleared ³

¹ See [“ALU Execution Status” on page 3-11](#).

² See [“ALU Execution Status” on page 3-11](#).

³ See [“ALU Execution Status” on page 3-11](#).

Examples

```
FRO = PASS R1 ;; /* SIMD pass */  
XFR1:0 = PASS R3:2 ;; /* 40-bit extended precision */
```

ALU Instructions

Reciprocal (Floating-Point)

Syntax

```
{X|Y|XY}FRs = RECIPS Rm ;  
{X|Y|XY}FRsd = RECIPS Rmd ;
```

Function

This instruction creates an 8-bit accurate approximation for $1/R_m$, the reciprocal of R_m . The *d* suffix denotes the operand size—see “[Instruction Line Syntax and Structure](#)” on page 1-20.

The mantissa of the approximation is determined from a table using the seven MSBs (excluding the hidden bit) of the R_m mantissa as an index. The unbiased exponent of the seed is calculated as the two’s-complement of the unbiased R_m exponent, decremented by one. If e is the unbiased exponent of R_m , then the unbiased exponent of $FRs = -e - 1$. The sign of the seed is the sign of the input. A $\pm zero$ returns $\pm infinity$ and sets the overflow flag. If the unbiased exponent of R_m is greater than +125, the result is $\pm zero$. A NAN input returns an all ones result.

Status Flags

AZ	Set if result is $\pm zero$ or denormal
AUS	Set if result is denormal; otherwise not cleared ¹
AN	Set if result is negative
AV	Set if input operand is $\pm zero$
AVS	Set if input operand is $\pm zero$; otherwise not cleared ²
AC	Cleared

¹ See “[ALU Execution Status](#)” on page 3-11.

² See “[ALU Execution Status](#)” on page 3-11.

AI	Set if input is NAN
AIS	Set if input is NAN; otherwise not cleared ¹

Examples

```

/* This code provides a floating-point division using an iterative convergence algorithm. The result is accurate to one LSB */
.SECTION program;
_main:
  YR0 = 10.0 ;; /* Random Numerator */
  YR12 = 3.0 ;; /* Random Denominator */
  YR11 = 2.0 ;;
  R7 = R0 ;;
  YFR0 = RECIPS R12 ;; /* Get 8 bit 1/r12 */
  YFR12 = R0 * R12 ;;
  YFR7 = R0 * R7 ; YFR0=R11-R12 ;;
  YFR12 = R0 * R12 ;;
  YFR7 = R0 * R7 ; YFR0=R11-R12 ;;
/* single precision. You can eliminate the three lines below if only a +/-1 LSB accurate single precision result is necessary */
  YFR12 = R0 * R12 ;;
  YFR7 = R0 * R7 ;;
  YFR0 = R11 - R12 ;;
/* Above three lines are for results accurate to one LSB */
  YFR0 = R0 * R7 ;; /* Output is in YR0 */
endhere:
  NOP ;;
  IDLE ;;
  JUMP endhere ;;

```

¹ See “ALU Execution Status” on page 3-11.

ALU Instructions

Reciprocal Square Root (Floating-Point)

Syntax

$\{X|Y|XY\}FRs = RSQRTS Rm ;$
 $\{X|Y|XY\}FRsd = RSQRTS Rmd ;$

Function

This instruction creates an 8-bit accurate approximation for $1/\sqrt{Rm}$, the reciprocal square root of Rm . The *d* suffix denotes the operand size—see “[Instruction Line Syntax and Structure](#)” on page 1-20.

The mantissa of the approximation is determined from a table using the LSB of the biased exponent of Rm conjugating with the six MSBs (excluding the hidden bit) of the mantissa of Rm as an index. The unbiased exponent of the seed is calculated as the two’s-complement of the unbiased Rm exponent, shifted right by one bit and decremented by one—that is, if e is the unbiased exponent of Rm , then the unbiased exponent of $FRs = -NT[e/2] - 1$. The sign of the seed is the sign of the input. A \pm zero returns \pm infinity and sets the overflow flag. A $+$ infinity returns $+$ zero. A NAN input or a negative non-zero (including $-$ infinity) returns an all ones result.

Status Flags

AZ	Set if result is zero
AUS	Not cleared ¹
AN	Set if input operand is $-$ zero
AV	Set if input operand is \pm zero
AVS	Set if input operand is \pm zero; otherwise not cleared ²

¹ See “[ALU Execution Status](#)” on page 3-11.

² See “[ALU Execution Status](#)” on page 3-11.

AC	Cleared
AI	Set if input is negative and non-zero, or NAN
AIS	Set if input is negative and non-zero, or NAN; otherwise not cleared ¹

Examples

```

/* This code calculates a floating point reciprocal square root
using a Newton Raphson iteration algorithm. The result is accurate
to one LSB */
XR0 = 9.0 ;; // random input ;;
XR8 = 3.0 ;;
XR1 = 0.5 ;;
XFR4 = RSQRTS R0 ;;
XFR12 = R4 * R4 ;;
XFR12 = R12 * R0 ;;
XFR4 = R1 * R4 ; XFR12 = R8 - R12 ;;
XFR4 = R4 * R12 ;;
XFR12 = R4 * R4 ;;
XFR12 = R12 * R0 ;;
XFR4 = R1 * R4 ; XFR12 = R8 - R12 ;;
/* Single precision. You can eliminate the four lines below if
only a +/-1 LSB accurate single precision result is necessary */
XFR4 = R4 * R12 ;;
XFR12 = R4 * R4 ;;
XFR12 = R12 * R0 ;;
XFR4 = R1 * R4 ; XFR12 = R8 - R12 ;;
XFR4 = R4 * R12 ;; /* reciprocal square root of xr0 is output is
in r4 */
endhere:

```

¹ See “ALU Execution Status” on page 3-11.

ALU Instructions

```
NOP ;;  
IDLE ;;  
JUMP endhere ;;
```


Mantissa (Floating-Point)

Syntax

$$\{X|Y|XY\}Rs = \text{MANT } FRm|FRmd ;$$

Function

This instruction extracts the mantissa (fraction bits with explicit hidden bit, excluding the sign bit) from the floating-point operand in register *Rm*. The unsigned-magnitude result is left-adjusted in the fixed-point field and placed in register *Rs*. The *d* suffix denotes extended operand size—see “[Instruction Line Syntax and Structure](#)” on page 1-20.

Note that the *AN* flag is set to the sign bit of the input operand.

Rounding modes are ignored and no rounding is performed, because all results are inherently exact. Denormal inputs are flushed to zero. A NAN or an infinity input returns an all ones result.

Status Flags

AZ	Set if result is zero
AUS	Not cleared ¹
AN	Set if input operand (<i>FRm</i>) is negative
AV	Cleared
AVS	Not cleared ²
AC	Cleared
AI	Set if input is NAN or infinity

¹ See “[ALU Execution Status](#)” on page 3-11.

² See “[ALU Execution Status](#)” on page 3-11.

ALU Instructions

AIS Set if input is NAN or infinity; otherwise not cleared¹

Examples

```
XR0 = MANT FR1 ;; /* x compute block */
YR0 = MANT FR3:2 ;; /* y compute block, 40-bit double precision
*/
```

¹ See [“ALU Execution Status”](#) on page 3-11.

Logarithm (Floating-Point)

Syntax

$$\{X|Y|XY\}Rs = \text{LOGB } FRm|FRmd \{(S)\} ;^1$$

Function

This instruction converts the exponent of the floating-point operand in register *Rm* to an unbiased two's-complement, fixed-point integer result. The result is placed in register *Rs* as an integer. The *d* suffix denotes extended operand size—see “[Instruction Line Syntax and Structure](#)” on [page 1-20](#).

Unbiasing is performed by subtracting 127 from the floating-point exponent in *Rm*. If saturation mode is not set, a \pm infinity input returns a floating-point \pm infinity; otherwise it returns the maximum positive value (0x7FFF FFFF). A \pm zero input returns a floating-point \pm infinity if saturation is not set, and maximum negative value (0x8000 0000) if saturation is set. Denormal inputs are flushed to \pm zero. A NAN input returns an all ones result.

Status Flags

AZ	Set if fixed-point result is zero
AUS	Not cleared ²
AN	Set if fixed-point result is negative
AV	Set if input is \pm infinity or \pm zero
AVS	Set if input is \pm infinity or \pm zero; otherwise not cleared ³

¹ Options include: (): Do not saturate, (S): Saturate

² See “[ALU Execution Status](#)” on [page 3-11](#).

³ See “[ALU Execution Status](#)” on [page 3-11](#).

ALU Instructions

AC	Cleared
AI	Set if input is a NAN
AIS	Set if input is a NAN; otherwise not cleared ¹

Options

()	Saturation inactive
(S)	Saturation active

Examples

```
XR0 = LOGB XFR1 (S) ;; /* x compute block with saturate active
option */
YR0 = LOGB YFR3:2 ;; /* y compute blocks 40 bit extended preci-
sion */
```

¹ See [“ALU Execution Status”](#) on page 3-11.

Add/Subtract (Dual Operation, Floating-Point)

Syntax

$\{X|Y|XY\}FRs = Rm + Rn, FRa = Rm - Rn ;$ (*dual instruction*)

$\{X|Y|XY\}FRsd = Rmd + Rnd, FRad = Rmd - Rnd ;$ (*dual instruction*)

Function

This instruction simultaneously adds and subtracts the floating-point operands in registers Rm and Rn . The results are placed in registers FRs and FRa .

The d suffix denotes extended operand size—see “[Instruction Line Syntax and Structure](#)” on page 1-20.

Status Flags

AZ	Set if one of the post-rounded results is denormal or zero (unbiased exponent ≤ 126) or \pm zero
AUS	Set if one of the post-rounded results is denormal; otherwise not cleared ¹
AN	Set if one of the post-rounded results is negative
AV	Set if one of the post-rounded results overflows
AVS	Set if one of the post-rounded results overflows; otherwise not cleared ²
AC	Not cleared
AI	Set if either input operand is a NAN, or if both operands are Infinities

¹ See “[ALU Execution Status](#)” on page 3-11.

² See “[ALU Execution Status](#)” on page 3-11.

ALU Instructions

AIS

Set if either input operand is a NAN, or if both operands are Infinities; otherwise not cleared¹

Examples

```
XFR0 = R1 + R2 , XFR3 = R1 - R2 ;; /* x compute block add and subtract */
```

```
XFR1:0 = R3:2 + R5:4 , XFR7:6 = R3:2 - R5:4 ;; /* extended 40 bit precision */
```

¹ See [“ALU Execution Status”](#) on page 3-11.

CLU Instructions

The communications logic unit (CLU) performs all communications algorithm specific *arithmetic operations* (addition/subtraction) and *logical operations*. For a description of CLU operations, status flags, conditions, and examples, see [“CLU Data Types and Sizes” on page 3-22](#), [“TMAX Function” on page 3-23](#), [“Trellis Function” on page 3-24](#), and [“Despread Function” on page 3-26](#).

The conventions used in these reference pages for representing register names, optional items, and choices are covered in detail in [“Register File Registers” on page 2-5](#). Briefly, these conventions are:

- { } – the curly braces enclose options; these braces are not part of the instruction syntax.
- | – the vertical bars separate choices; these bars are not part of the instruction syntax.
- *Rmd* – the register names in italic represent user-selectable single (*Rs*, *Rm*, *Rn*), double (*Rsd*, *Rmd*, *Rnd*) or quad (*Rsq*, *Rmq*, *Rnq*) register names.



Each instruction presented on these reference pages occupies one instruction slot in an instruction line. For more information about instruction lines and instruction combination constraints, see [“Instruction Line Syntax and Structure” on page 1-20](#) and [“Instruction Parallelism Rules” on page 1-24](#).

CLU Instructions

Trellis Maximum (CLU)

Syntax

```
{X|Y|XY}{S}TRsd = TMAX(TRmd + Rmq_h, TRnd + Rmq_l) ;  
{X|Y|XY}{S}TRsd = TMAX(TRmd - Rmq_h, TRnd - Rmq_l) ;  
{X|Y|XY}{S}Rs = TMAX(TRm, TRn) ;  
{X|Y|XY}{S}TRsd = MAX(TRmd + Rmq_h, TRnd + Rmq_l) ;  
{X|Y|XY}{S}TRsd = MAX(TRmd - Rmq_h, TRnd - Rmq_l) ;
```

Function:

$STRsd = TMAX(TRmd + Rmq_h, TRnd + Rmq_l) ;$

High part of Rmq is added to $TRmd$, low part of Rmq is added to $TRnd$, and $TMAX$ function is executed between both add results, as illustrated in [Figure 8-27](#) and [Figure 8-28](#).

This instruction can be executed in parallel to shifter instructions, multiplier instructions and CLU register load. It can not be executed in parallel to other CLU instructions and ALU instructions of the same compute block. Saturation is supported in this instruction. For more details, see “[Saturation Option](#)” on page 3-8.

Function:

$STRsd = TMAX(TRmd - Rmq_h, TRnd - Rmq_l) ;$

The high part of Rmq is subtracted from $TRmd$, low part of Rmq is subtracted from $TRnd$, and $TMAX$ function is executed between both subtract results, as illustrated in [Figure 8-29](#) and [Figure 8-30](#). For subtraction, the order of operands appears in [Figure 8-26](#).

This instruction can be executed in parallel to shifter instructions, multiplier instructions and CLU register load. It can not be executed in parallel to other CLU instructions and ALU instructions of the same compute block. Saturation is supported in this instruction. For more details, see “[Saturation Option](#)” on page 3-8.

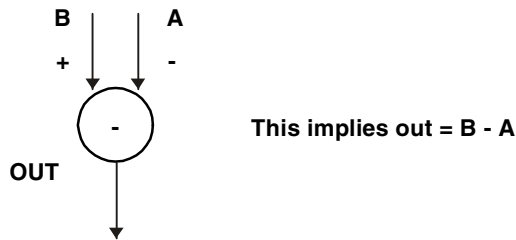


Figure 8-26. Order of Operands for Subtract Diagrams

Function:

$$SRs = TMAX(TRm, TRn) ;$$

TMAX function is executed between TRm and TRn , as illustrated in [Figure 8-31](#) and [Figure 8-32](#).

This instruction can be executed in parallel to shifter instructions, multiplier instructions and CLU register load. It cannot be executed in parallel to other CLU instructions and ALU instructions of the same compute block.

Status Flags

TROV	Overflow (TROV) is calculated every cycle
TRSOV	Set whenever an overflow occurs and cleared only by X/Ystat load

CLU Instructions

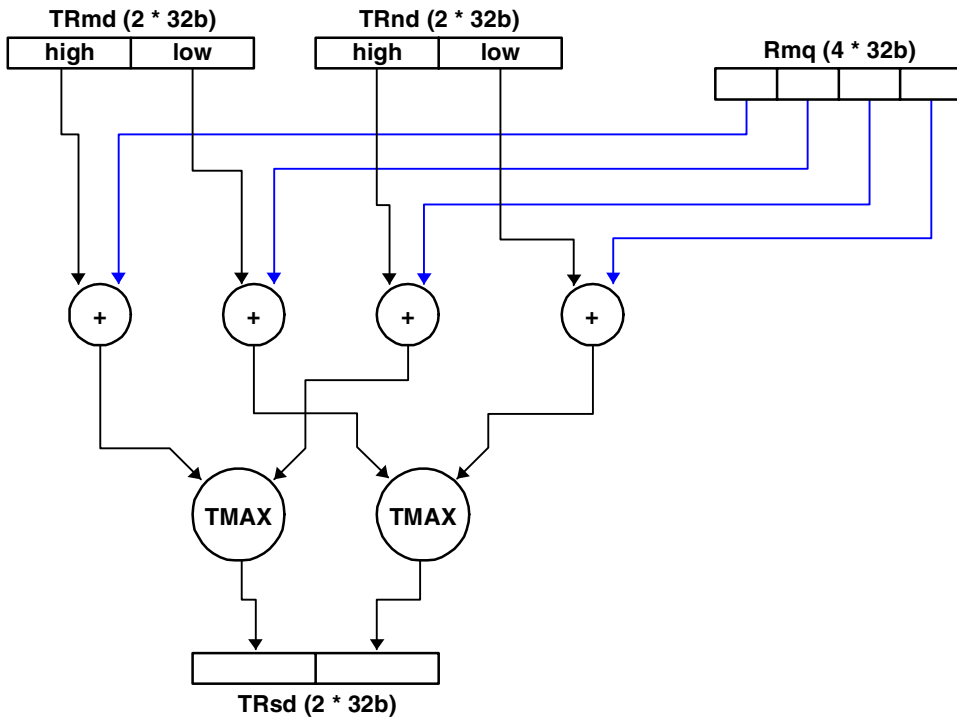


Figure 8-27. $TRsd = TMAX(TRmd + Rmq_h, TRnd + Rmq_l)$

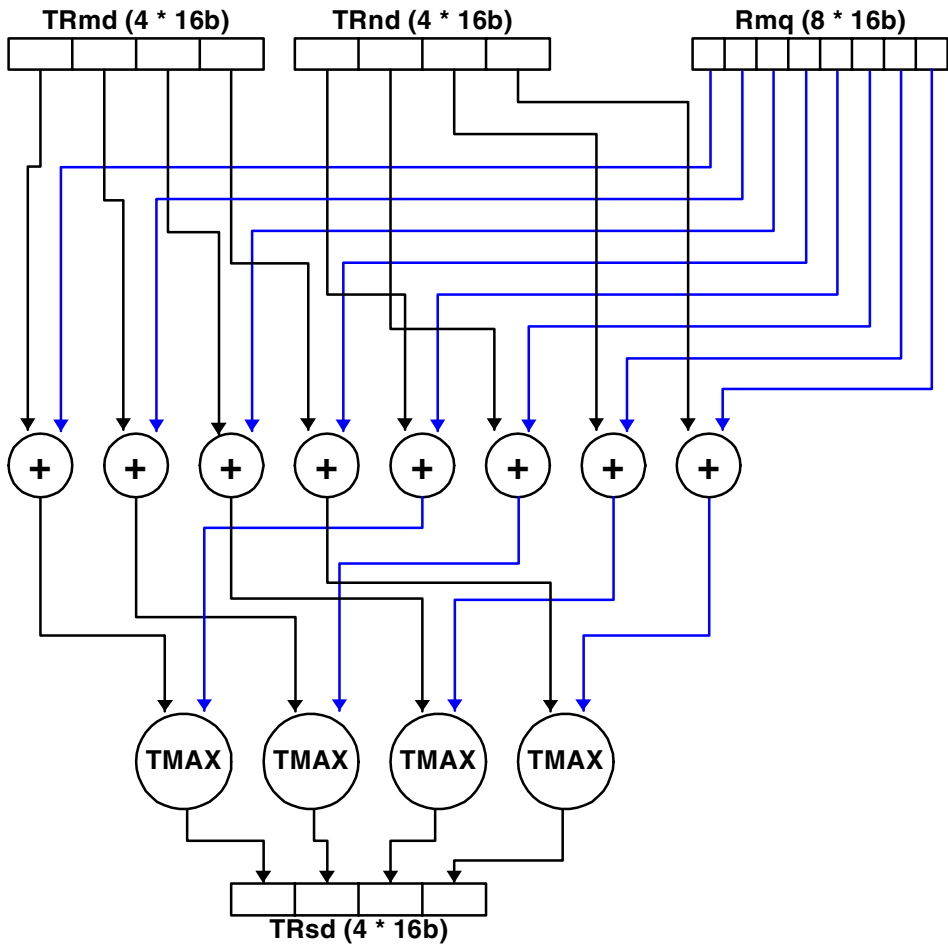


Figure 8-28. $STRsd = TMAX(TRmd + Rmq_h, TRnd + Rmq_l)$

CLU Instructions

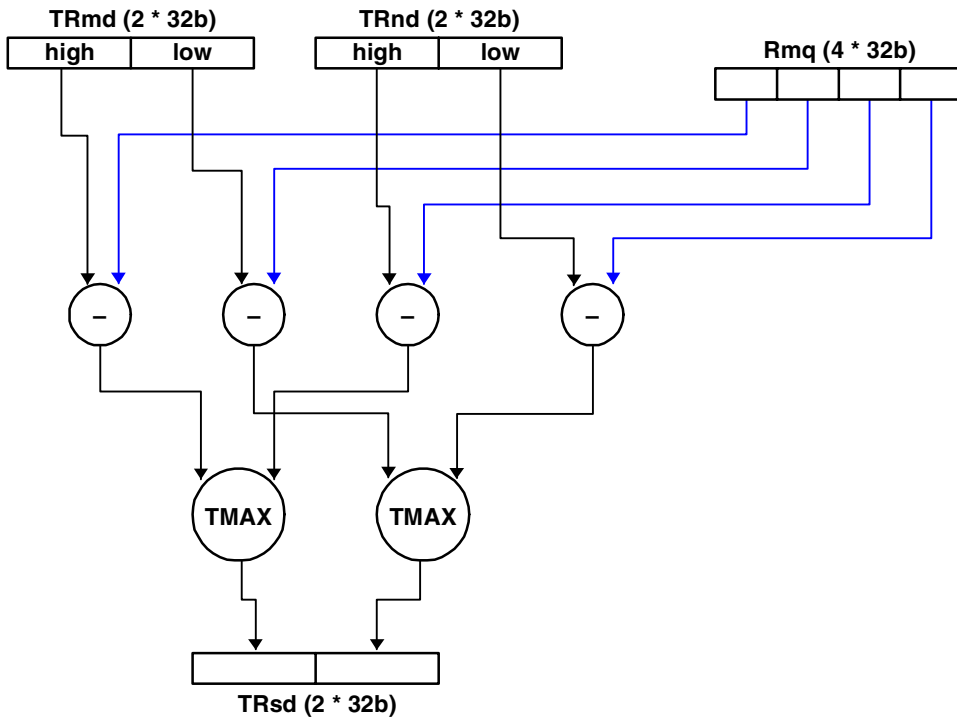


Figure 8-29. $TRsd = TMAX(TRmd - Rmq_h, TRnd - Rmq_l)$

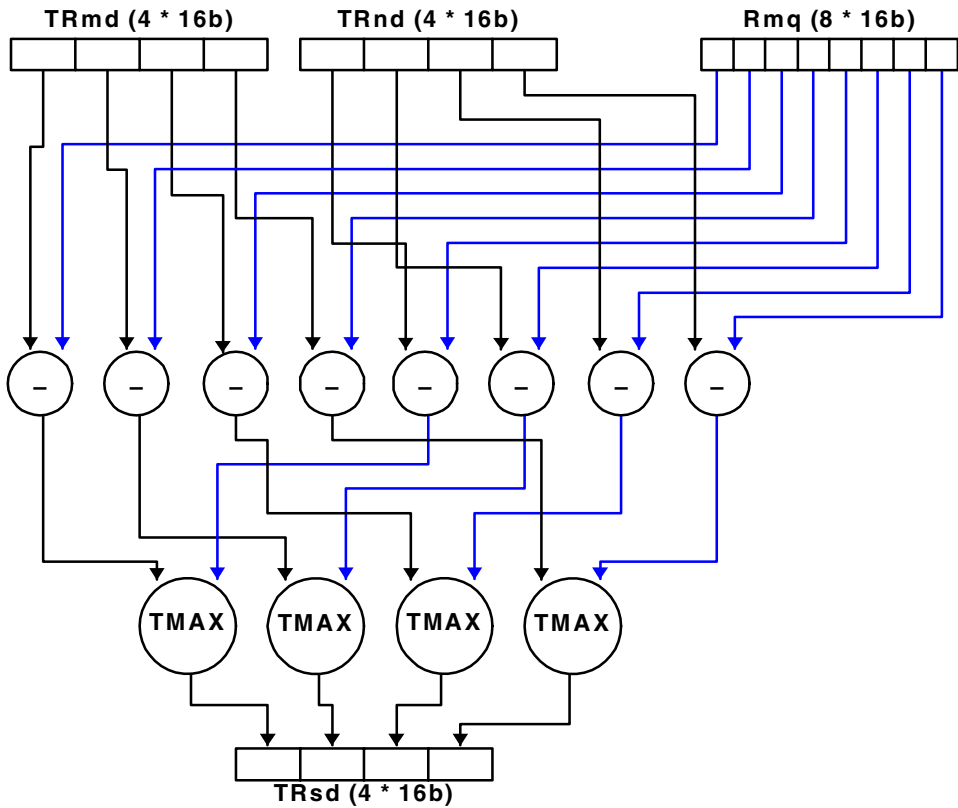


Figure 8-30. $STRsd = TMAX(TRmd - Rmq_h, TRnd - Rmq_l)$

CLU Instructions

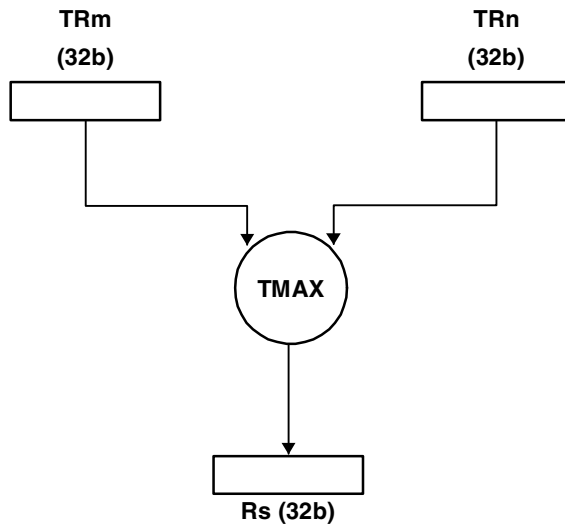


Figure 8-31. $R_s = TMAX(TR_m, TR_n)$ instruction processing

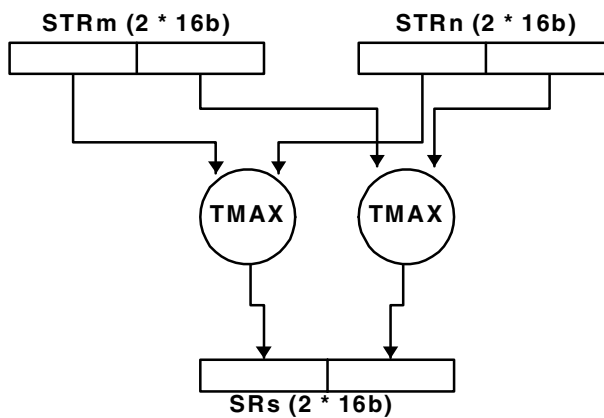


Figure 8-32. $SR_s = TMAX(TR_m, TR_n)$

Maximum (CLU)

Syntax

$$\{X|Y|XY\}\{S\}TRsd = \text{MAX}(TRmd + Rmq_h, TRnd + Rmq_l) ;$$

$$\{X|Y|XY\}\{S\}TRsd = \text{MAX}(TRmd - Rmq_h, TRnd - Rmq_l) ;$$

Function:

$$STRsd = \text{MAX}(TRmd + Rmq_h, TRnd + Rmq_l) ;$$

High part of Rmq is added to $TRmd$, low part of Rmq is added to $TRnd$, and selects the MAX between both add results, as illustrated in [Figure 8-33](#) and [Figure 8-34](#).

This instruction can be executed in parallel to shifter instructions, multiplier instructions and CLU register load. It can not be executed in parallel to other CLU instructions and ALU instructions of the same compute block. Saturation is supported in this instruction. For more details, see “[Saturation Option](#)” on page 3-8.

Function:

$$STRsd = \text{MAX}(TRmd - Rmq_h, TRnd - Rmq_l) ;$$

High part of Rmq is subtracted from $TRmd$, low part of Rmq is subtracted from $TRnd$, and selects the MAX between both subtract results, as illustrated in [Figure 8-35](#) and [Figure 8-36](#).

This instruction can be executed in parallel to shifter instructions, multiplier instructions and CLU register load. It can not be executed in parallel to other CLU instructions and ALU instructions of the same compute block. Saturation is supported in this instruction. For more details, see “[Saturation Option](#)” on page 3-8.

CLU Instructions

Status Flags

TR0V Overflow (TR0V) is calculated every cycle

TRSOV Set whenever an overflow occurs and cleared only by X/Ystat load

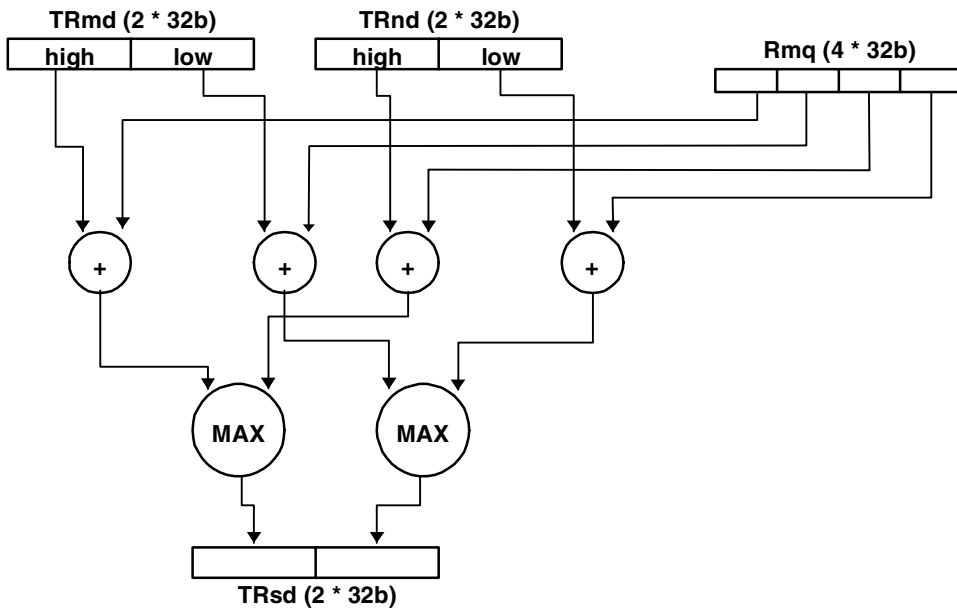


Figure 8-33. $TRsd = \text{MAX}(TRmd + Rmq_h, TRnd + Rmq_l)$

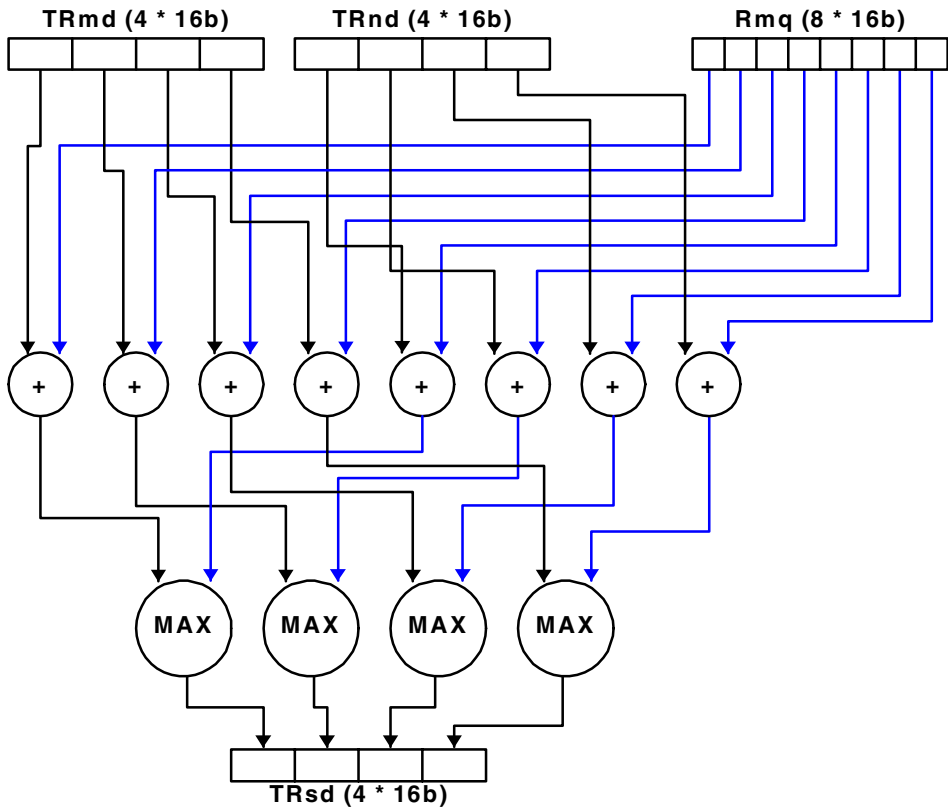


Figure 8-34. $STRsd = MAX(TRmd + Rmq_h, TRnd + Rmq_l)$

CLU Instructions

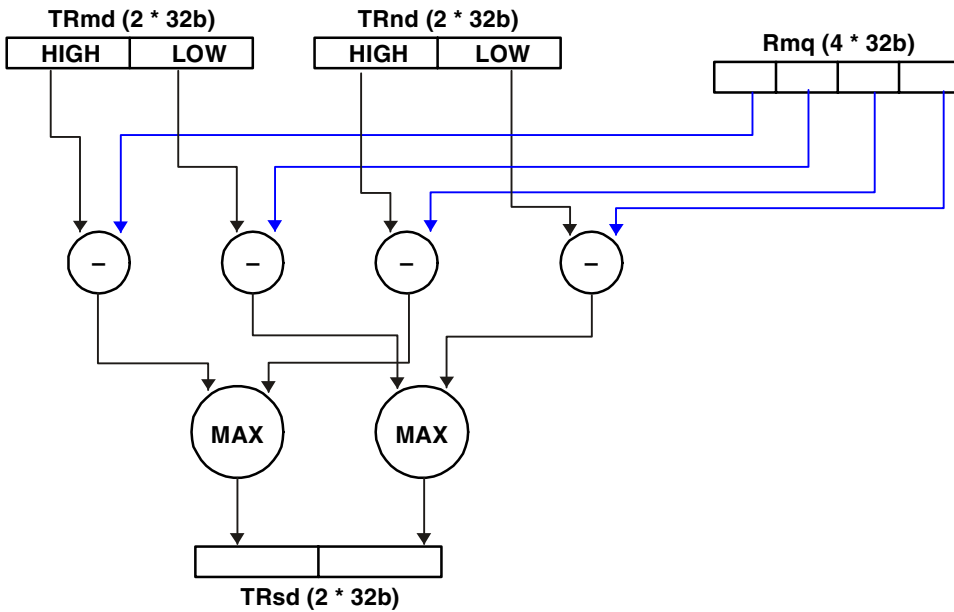


Figure 8-35. $TRsd = \text{MAX}(TRmd - Rmq_h, TRnd - Rmq_l)$

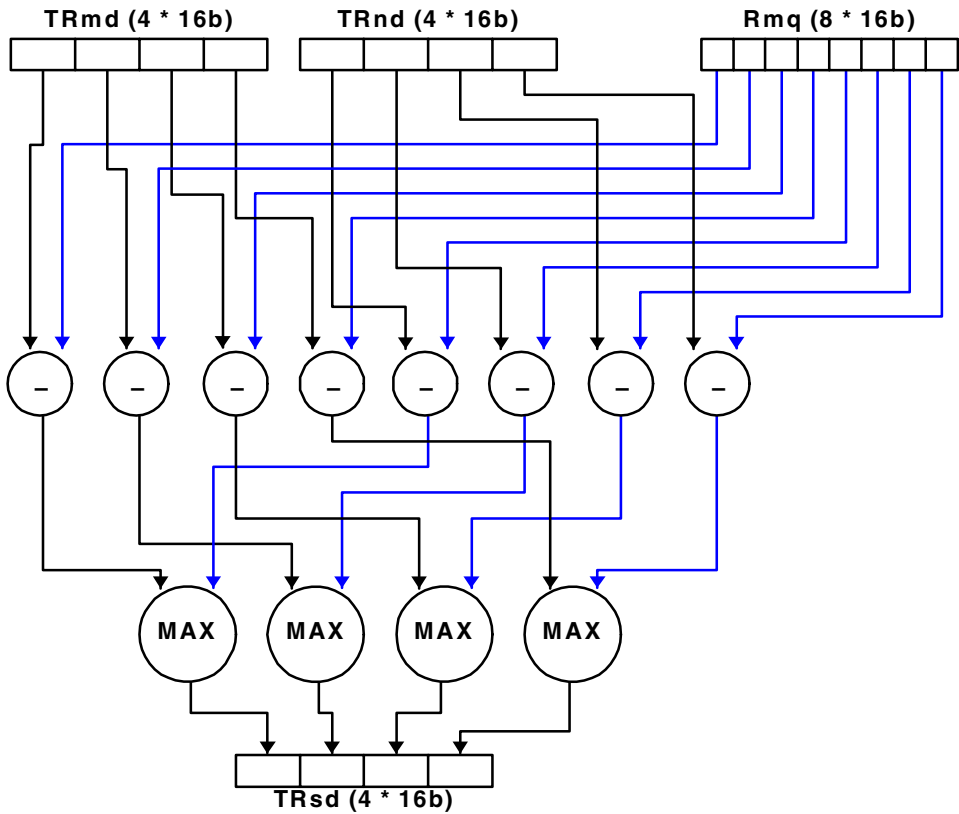


Figure 8-36. $STRsd = MAX(TRmd - Rmq_h, TRnd - Rmq_l)$

CLU Instructions

Trellis Registers (CLU)

Syntax

```
{X|Y|XY}Rs = TRm ;  
{X|Y|XY}Rsd = TRmd ;  
{X|Y|XY}Rsq = TRmq ;  
{X|Y|XY}TRs = Rm ;  
{X|Y|XY}TRsd = Rmd ;  
{X|Y|XY}TRsq = Rmq ;  
{X|Y|XY}Rs = THRm ;  
{X|Y|XY}Rsd = THRmd ;  
{X|Y|XY}Rsq = THRmq ;1  
{X|Y|XY}THRs = Rm ;  
{X|Y|XY}THRsd = Rmd {(i)} ;  
{X|Y|XY}THRsq = Rmq ;1
```

Function

The data in the source register (to right of =) are transferred to the destination register (to left of =). Data are single (32 bits), double (64 bits) or quad word (128 bits). The register number must be aligned to the data size.

Data are transferred on EX2.

¹ Not implemented, but syntax reserved

This instruction may be executed together with Shifter or Multiplier instructions. It can not be executed in parallel to ALU or CLU instructions in the same compute unit.

- i** The quad option ($Rsq = THRs$; and $THRs = Rmq$;) is not implemented because there are only two registers; it is reserved for future use.
- i** In case of executing the $THRs = Rm$; or $THRs = Rmd \{(i)\}$; instructions in parallel to an instruction that shifts the THR register (ACS, DESPREAD), the THR load instruction takes priority on the THR shift in this instruction

Status Flags

None

Options

The interleave option (I) is valid only for double registers. It interleaves the bits of the low word of Rmd with the bits of the high word of Rmd , and loads the interleaved data into the THR register, as illustrated in [Figure 8-37](#).

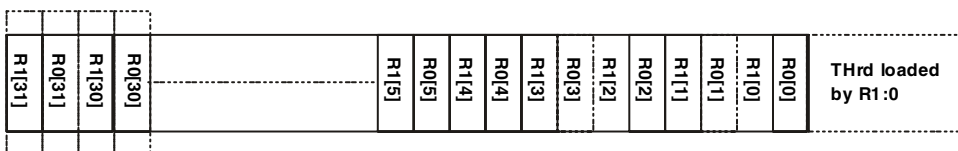


Figure 8-37. Interleave Option for Double Registers

CLU Instructions

Despread (CLU)

Syntax

$\{X|Y|XY\}TRs = \text{DESPREAD} (Rmq, \text{THRd}) + TRn ;$
 $\{X|Y|XY\}Rs = TRs, TRs = \text{DESPREAD} (Rmq, \text{THRd}) ;$ *(dual instruction)*
 $\{X|Y|XY\}Rsd = TRsd, TRsd = \text{DESPREAD} (Rmq, \text{THRd}) ;$ *(dual instruction)*

Function

The input register Rmq is composed of 8 complex shorts - D7 to D0, in which each complex number is composed of 2 bytes. The most significant byte is the imaginary, and the least significant is the real. Semantics - Dn is composed of DnI and DnQ, where I denotes the real part and Q denotes the imaginary part.

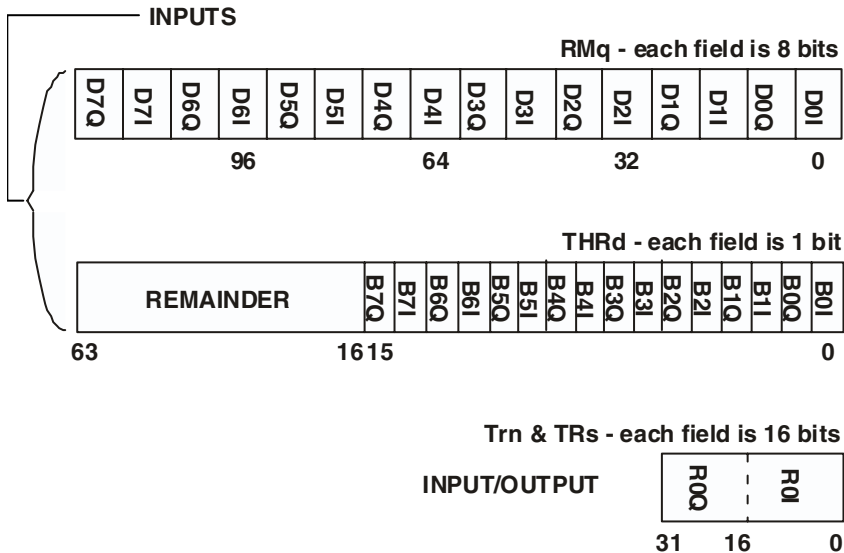


Figure 8-38. Bit field in registers for $TRs = \text{DESPREAD} (Rmq, \text{THRd}) + TRn ;$

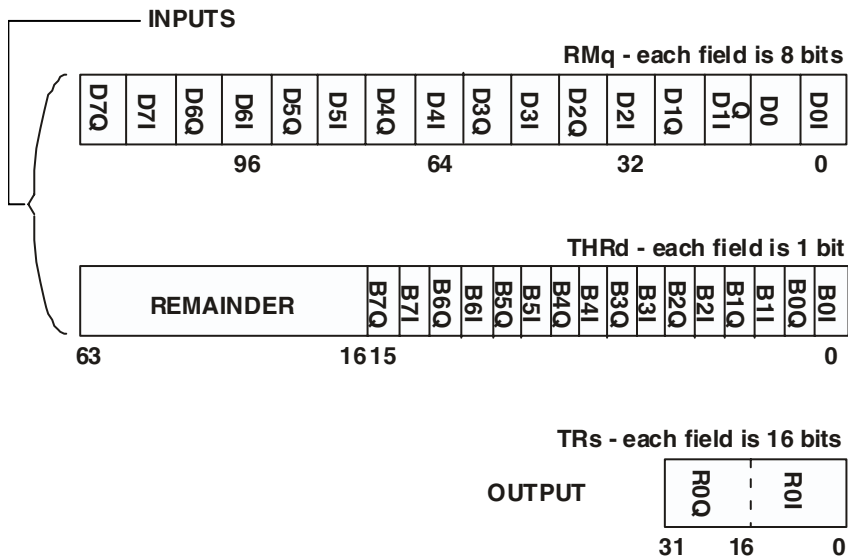





Figure 8-39. Bit field in registers for:
 $R_s = TR_s$, $TR_s = DESPREAD (RM_q, THR_d)$;

The THR_d register is composed of 8 complex numbers $\{B7...B0\}$ and 48 Remainder bits. Each complex number is composed of one real bit (least significant) and one imaginary bit. Each bit represents the value of +1

CLU Instructions

(when clear) or -1 (when set). $THRd$ is post-shifted right by 16 bits so that the lowest 16 bits of the remainder may be used for a despread on the next cycle.

-  Saturation is supported in this instruction. For more details, see “[Saturation Option](#)” on page 3-8.
-  This instruction can be executed in parallel to multiplier instructions and CLU register load. It can not be executed in parallel to other CLU instructions and ALU instructions of the same compute block.
-  When you execute this instruction in parallel to THR register load the THR load instruction takes priority on the THR shift in this instruction.

Function: $TRs = DESPREAD (Rmq, THRd) + TRn$;

TRn and TRs are complex words, each composed of two shorts - real (least significant) and imaginary (most significant).

The function (illustrated in [Figure 8-41](#)) is:

$$TRsI = (\text{sum } (n = 0 \text{ to } 7) (BnI * DnI - BnQ * DnQ)) + TRnI$$
$$TRsQ = (\text{sum } (n = 0 \text{ to } 7) (BnI * DnQ + BnQ * DnI)) + TRnQ$$

The multiplication is by integer (± 1) for example the result alignment is to least significant, and the sum is sign extended.

Function: $Rs = TRs, TRs = DESPREAD (Rmq, THRd)$;

TRs is a complex word, composed of two shorts - real (least significant) and imaginary (most significant).

The function (illustrated in [Figure 8-42](#)) is:

$$TRsI = (\text{sum } (n = 0 \text{ to } 3) (BnI * DnI - BnQ * DnQ))$$
$$TRsQ = (\text{sum } (n = 0 \text{ to } 3) (BnI * DnQ + BnQ * DnI))$$

The value of TRs before the operation is stored in Rs .

Function: $Rsd = TRsd, TRsd = DESPREAD (Rmq, THRd) ;$

The function (illustrated in Figure 8-43) is:

$$TR0I = (\text{sum } (n = 0 \text{ to } 3) (BnI * DnI - BnQ * DnQ))$$

$$TR0Q = (\text{sum } (n = 0 \text{ to } 3) (BnI * DnQ + BnQ * DnI))$$

$$TR1I = (\text{sum } (n = 4 \text{ to } 7) (BnI * DnI - BnQ * DnQ))$$

$$TR1Q = (\text{sum } (n = 4 \text{ to } 7) (BnI * DnQ + BnQ * DnI))$$

Status Flags

TROV Overflow (TROV) is calculated every cycle

TRSOV Set whenever an overflow occurs and cleared only by X/Ystat load

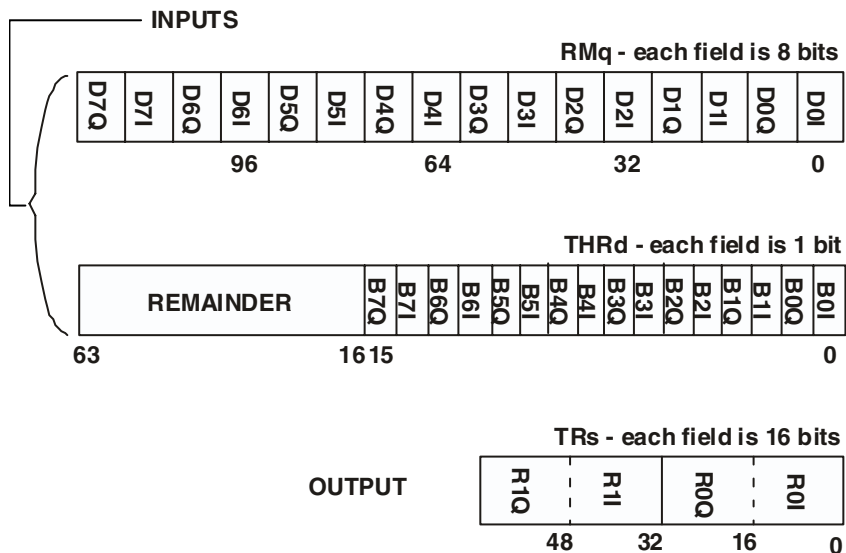


Figure 8-40. Bit fields in registers for:
 $Rsd = TRsd, TRsd = DESPREAD (Rmq, THRd) ;$

CLU Instructions

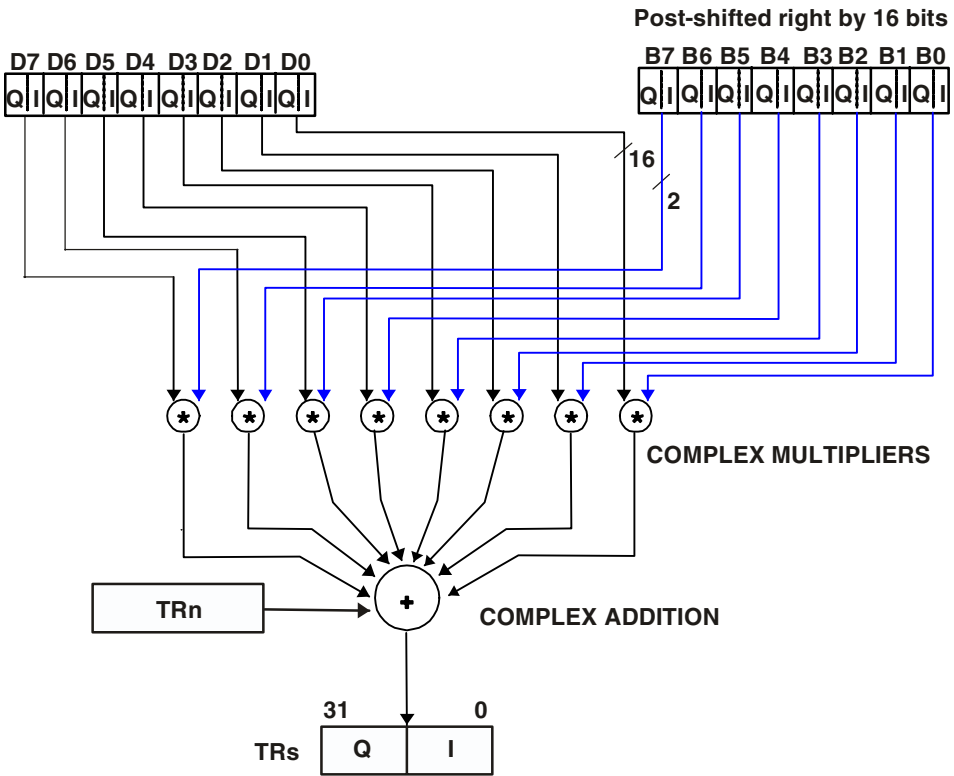


Figure 8-41. $TRs = \text{DESPREAD}(Rmq, THRd) + TRn$;

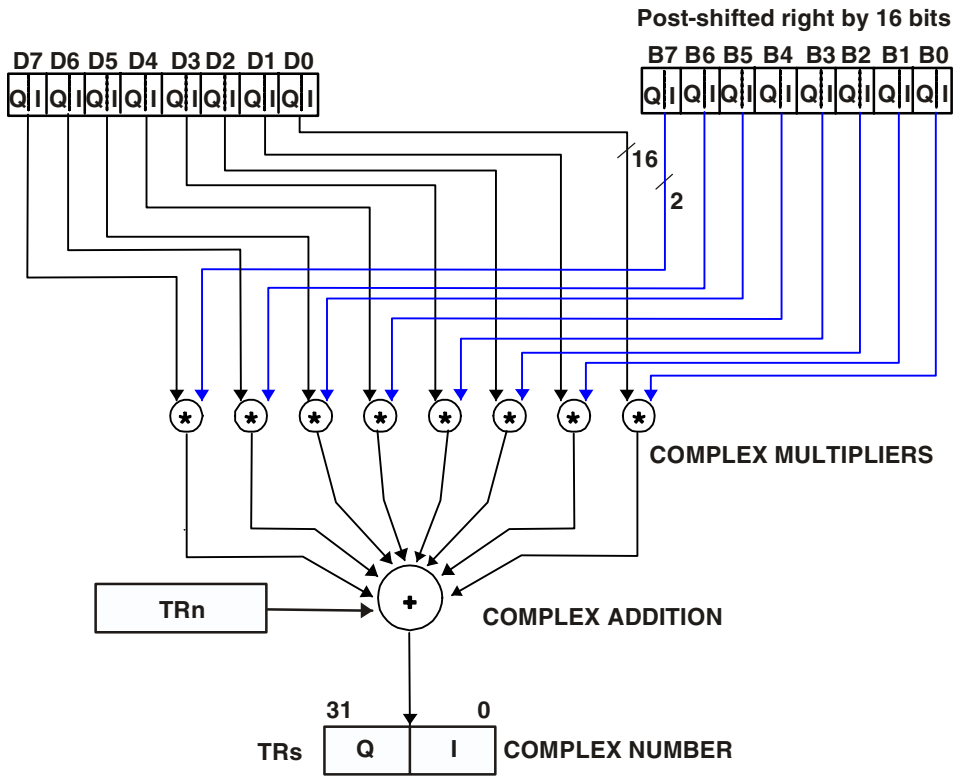


Figure 8-42. $R_s = TR_s$, $TR_s = \text{DESPREAD}(R_{mq}, \text{THRd})$;

CLU Instructions

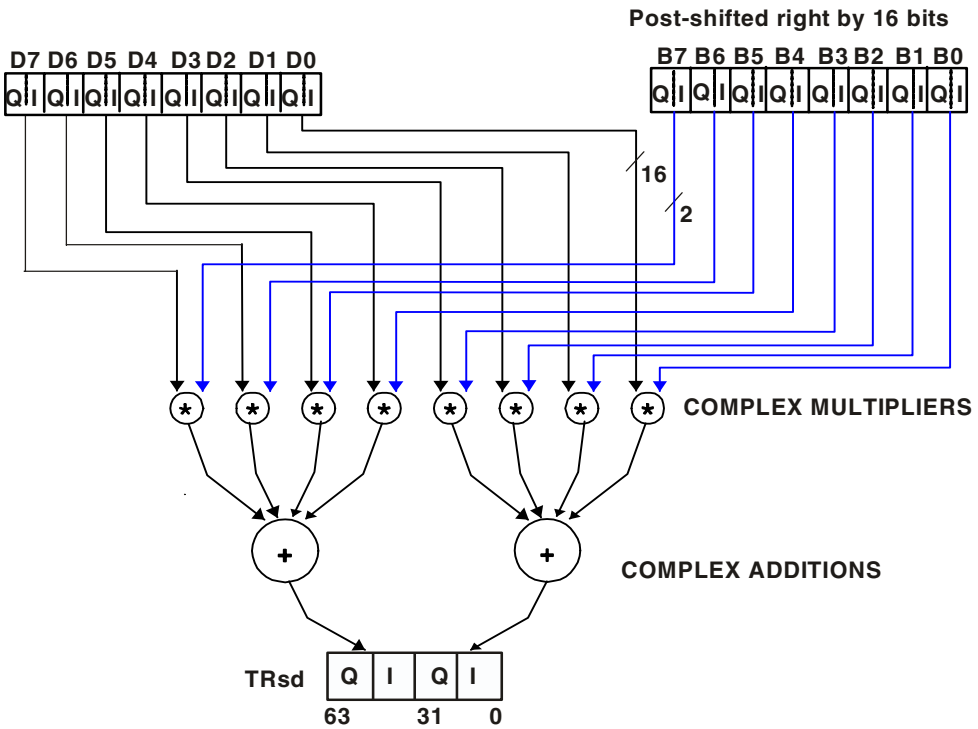


Figure 8-43. $Rsd = TRsd$, $TRsd = DESPREAD(Rmq, THRd)$;

Add/Compare/Select (CLU)

Syntax

$\{X|Y|XY\}\{S\}TRsq = ACS (TRmd, TRnd, Rm) (TMAX) ;$
 $\{X|Y|XY\}Rsq = TRaq, \{S\}TRsq = ACS (TRmd, TRnd, Rm) (TMAX) ;$ (*dual instr.*)

Function

For $TRsq = ACS (TRmd, TRnd, Rm) ;$, each *short* in Rm is added to and subtracted from the corresponding short word in $TRmd$ and $TRnd$. The *four* results of the add and subtract are compared in trellis order, as illustrated in [Figure 8-44](#).

Trellis history register $THR[1:0]$ is updated with the selection of the $MAX / TMAX$. After every ACS operation the *four* selection decisions are loaded into bits 31:28 of register $THR1$, while the content of $THR1:0$ is shifted right *4 bits*. Decision bit indicates which input the MAX or $TMAX$ has selected. For selection of $TRm +/- Rm$ the decision bit is 1, and for $TRn +/- Rm$ the decision is 0.

For $STRsq = ACS (TRmd, TRnd, Rm) ;$, each *byte* in Rm is added to and subtracted from the corresponding byte word in $TRmd$ and $TRnd$. The *eight* results of the add and subtract are compared in trellis order, as illustrated in [Figure 8-45](#).

Trellis history register $THR[1:0]$ is updated with the selection of the $MAX / TMAX$. After every ACS operation the *eight* selection decisions are loaded into bits 31:28 of register $THR1$, while the content of $THR1:0$ is shifted right *8 bits*. Decision bit indicates which input the MAX or $TMAX$ has selected. For selection of $TRm +/- Rm$ the decision bit is 1, and for $TRn +/- Rm$ the decision is 0.

CLU Instructions

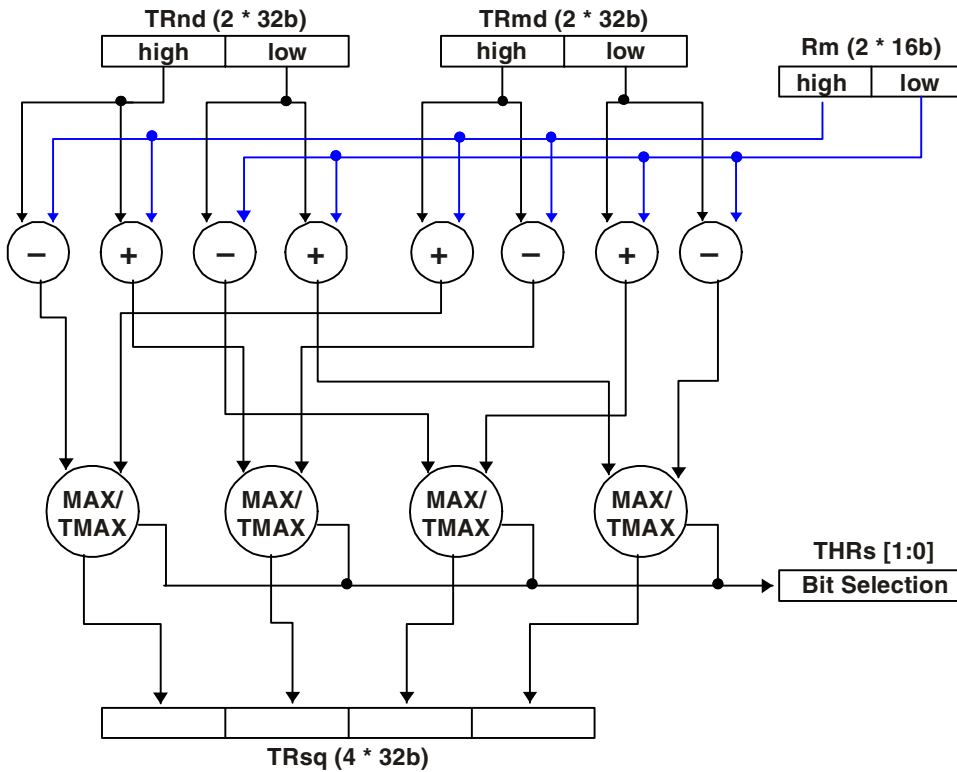




Figure 8-44. $TRsq = ACS (TRmd, TRnd, Rm)$;

Optionally, a trellis register transfer can be added for dual operation, as in:
 $Rsq = TRsq, \{S\}TRsq = ACS (TRmd, TRnd, Rm)$;

-  Saturation is supported in this instruction. For more details, see “[Saturation Option](#)” on page 3-8.
-  This instruction can be executed in parallel to shifter instructions, multiplier instructions and CLU register load. It can not be executed in parallel to other CLU instructions and ALU instructions of the same compute block.

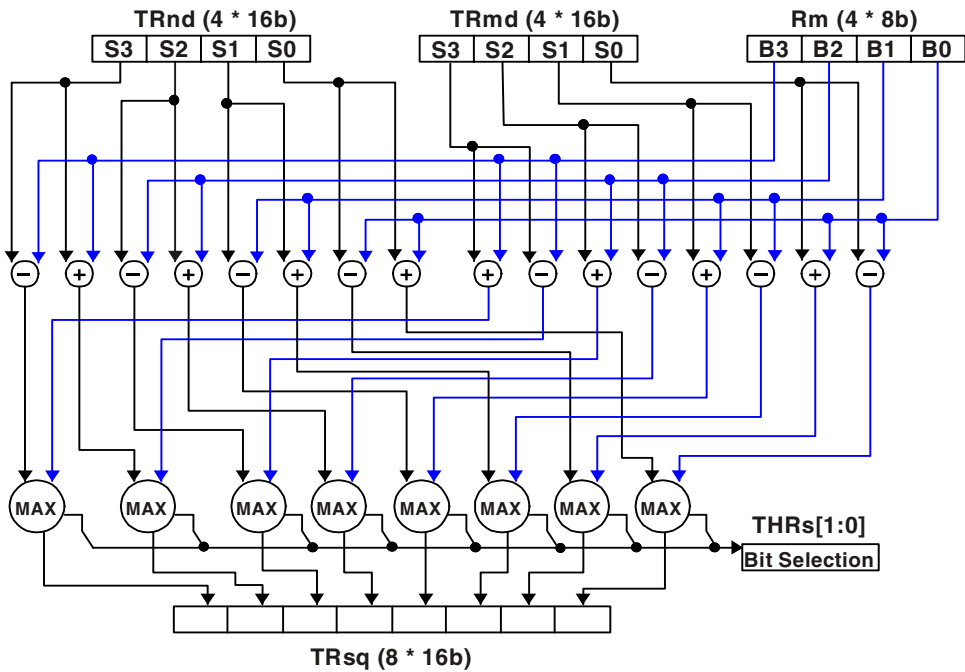


Figure 8-45. STRsq = ACS (TRmd, TRnd, Rm);

Status Flags

TROV	Overflow (TROV) is calculated every cycle
TRSOV	Set whenever an overflow occurs and cleared only by X/Ystat load

CLU Instructions

Options

TMAX

replaces the MAX function with TMAX for example added to the value from table:

$\text{Ln}(1 + e^{(-|A - B|)})$ where A and B are the two compared values

Example

The way to use the function in a trellis operation is as follows:

```
Loop: TR11:8 = ACS (TR1:0, TR5:4, R0);;
```

```
TR15:12 = ACS (TR3:2, TR7:6, R0);;
```

```
TR3:0 = ACS (TR9:8, TR13:12, R1);;
```

```
TR15:4 = ACS (TR11:10, TR15:14, R1); if nLCOE. jump Loop;;
```


Permute (Byte Word, CLU)

Syntax

```
{X|Y|XY}Rsd = PERMUTE (Rmd, Rn) ;
```

Function

The result, *Rsd*, is composed of bytes from first operand *Rmd*, which are selected by the control word *Rn*.

Rn is broken into 8 nibbles (4 bit fields), and *Rmd* is broken into 8 bytes. Each nibble in *Rn* is the control of the corresponding byte in *Rsd*. For example, nibble 2 in *Rn* (bits 11:8) corresponds to byte 2 in *Rsd* (bits 23:16). The control word selects which byte in *Rmd* is written to the corresponding byte in *Rsd*. The decode of a nibble in *Rn* is:

```
b#0000:  Select byte 0 - bits 7:0 of low word
b#0001:  Select byte 1 - bits 15:8 of low word
b#0010:  Select byte 2 - bits 23:16 of low word
b#0011:  Select byte 3 - bits 31:24 of low word
b#0100:  Select byte 4 - bits 7:0 of high word
b#0101:  Select byte 5 - bits 15:8 of high word
b#0110:  Select byte 6 - bits 23:16 of high word
b#0111:  Select byte 7 - bits 31:24 of high word
b#1XXX:  Reserved
```

Different nibbles in *Rn* may point to the same byte in *Rm*. In this case the same byte in *Rmd* is duplicated in *Rsd*.

Status Flags

Affected flags: None

CLU Instructions

This instruction can be executed in parallel to shifter instructions, multiplier instructions and CLU register load. It can not be executed in parallel to other CLU instructions and ALU instructions of the same compute block.

Example

```
R1:0 = 0x01234567_89abcdef  
R4 = 0x03172454  
R7:6 = permute (R1:0, R4)  
Result R7:6 is 0xef89cd01_ab674567
```

Permute (Short Word, CLU)

Syntax

$\{X|Y|XY\}Rsq = \text{PERMUTE} (Rmd, -Rmd, Rn) ;$

Function

The result, Rsq , is composed of shorts from first operand Rmd , and the short-wise two's complement values: $-Rmd$. The shorts are selected by the control word Rn .

Rn is broken into 8 nibbles (4 bit fields), and Rsq is broken into 8 shorts. Each nibble in Rn is the control of the corresponding short in Rmd (for example, nibble 2 in Rn , which is bits 11:8, corresponds to short 2 in Rsd , which is bits 47:32). The control word selects which short in Rmd is written to the corresponding short in Rsd . The decode of a nibble in Rn is:

b#0000: Select short 0 - bits 15:0 of Rmd
 b#0001: Select short 1 - bits 31:16 of Rmd
 b#0010: Select short 2 - bits 47:32 of Rmd
 b#0011: Select short 3 - bits 63:48 of Rmd

b#0100: Select short 0 - negated value of bits 15:0 of Rmd
 b#0101: Select short 1 - negated value of bits 31:16 of Rmd
 b#0110: Select short 2 - negated value of bits 47:32 of Rmd
 b#0111: Select short 3 - negated value of bits 63:48 of Rmd

-

b#1XXX: Reserved

Different nibbles in Rn may point to the same short in Rmd . In this case the same short in Rmd is duplicated in Rsd .

This instruction can not be executed in parallel to shifter instructions (because a quad result doesn't work with a shifter instruction) or ALU instructions. It can be executed in parallel with multiplier instructions or with load TR or load THR instructions.

CLU Instructions

Status Flags

Affected flags - Overflow (TR0V) is calculated every cycle, set if one of the shorts in Rmd is maximum negative ($0x8000$) and it is selected at least one of the nibbles of Rn to the result.

TRSOV is set whenever an overflow occurs and cleared only by X/Ystat load.

Example

```
R1:0 = 0x0123456789ABCDEF
```

```
-R1:0 = 0xFEDDBA9976553211
```

```
R4 = 0x03147623
```

```
R11:8 = permute (R1:0, -R1:0, R4)
```

```
Result R11:8 is 0xCDEF0123_89AB3211_FEDDBA99_45670123
```

Multiplier Instructions

The multiplier performs all *multiply operations* for the processor on fixed- and floating-point data and performs all *multiply-accumulate operations* for the processor on fixed-point data. This unit also performs all *complex multiply operations* for the processor on fixed-point data. The multiplier also executes *data compaction operations* on accumulated results when moving data to the register file in fixed-point formats. For a description of ALU operations, status flags, conditions, and examples, see [“Multiplier” on page 4-1](#).

The conventions used in these reference pages for representing register names, optional items, and choices are covered in detail in [“Register File Registers” on page 2-5](#). Briefly, these conventions are:

- { } – the curly braces enclose options; these braces are not part of the instruction syntax.
- | – the vertical bars separate choices; these bars are not part of the instruction syntax.
- *Rmd* – the register names in italic represent user-selectable single (*Rs*, *Rm*, *Rn*), double (*Rsd*, *Rmd*, *Rnd*) or quad (*Rsq*, *Rmq*, *Rnq*) register names.



Each instruction presented on these reference pages occupies one instruction slot in an instruction line. For more information about instruction lines and instruction combination constraints, see [“Instruction Line Syntax and Structure” on page 1-20](#) and [“Instruction Parallelism Rules” on page 1-24](#).

Multiplier Instructions

Multiply (Normal Word)

Syntax

$\{X|Y|XY\}Rs = Rm * Rn \{(\{U|nU\}\{I\}\{T\}\{S\})\} ;$

$\{X|Y|XY\}Rsd = Rm * Rn \{(\{U|nU\}\{I\})\} ;$

Function

This is a 32-bit multiplication of the normal-word value in register Rm with the value in Rn . For fractional operands, if rounding is specified by the absence of T , the result is rounded. (Note that option T does not apply to integer data). The result is placed in register Rs .

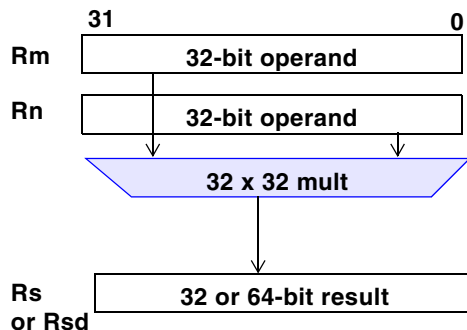


Figure 8-46. Multiply (Normal Word) Data Flow

Status Flags

MZ Set if all bits in result are zero

MN Set if result is negative

MV (MOS) Set according to the data format, under the following conditions (MOS unchanged if MV is cleared):

$Rsd = Rm * Rn; \Rightarrow$ no overflow

Fractional \Rightarrow No overflow. ¹

Signed integer \Rightarrow Upper 33 bits of M are not all zeros or all ones

Unsigned integer \Rightarrow Upper 32 bits of M are not all zeros

MU (MUS)

Cleared (unchanged)

Options

- () *Rm, Rn* signed fractional, result is rounded (if word)
- (U) *Rm, Rn* unsigned
- (nU) *Rm* signed, *Rn* unsigned
- (I) Operands are integer
- (T) Result is truncated (only for fractional if result is word)
- (S) Saturate (only for integer)

Note that not all allowed combinations are meaningful. For instance, rounding and truncation only apply to fractions, but do not apply to integers. See [“Multiplier Instruction Options” on page 4-8](#).

Example

```
XR1 = 0xFF46AC0C ;; /* xR1 = -0xB953F4 */
XR2 = 0xAF305216 ;;
XR0 = R1 * R2 (nUT) ;; /* xR0 = 0xFF812CA1 */
```

¹ Except when multiplying the most negative fraction times itself, in which case MV and MOS are set.

Multiplier Instructions

This multiply instruction specifies a 32-bit multiply of two fractions. $R1$ is signed, and $R2$ is unsigned. The multiplication produces a 32-bit unsigned and truncated result, and the MN flag is set indicating the result is negative.

```
XR0 = 0x0046AC0C ;;  
XR1 = 0x00005216 ;;  
XR3:2 = R0 * R1(UI) ;; /* xR3 = 0x00000016, xR2 = 0xA92EA108 */
```

This multiply instruction specifies a 32-bit multiply of two unsigned integers stored in $R0$ and $R1$. The result is a 64-bit unsigned integer. The upper 32 bits are stored in $R3$, and the lower 32 bits in $R2$. No flags are set for this example.

Multiply-Accumulate (Normal Word)

Syntax

```
{X|Y|XY}MRa += Rm * Rn {{{U}{I}{C|CR}}};
{X|Y|XY}MRa -= Rm * Rn {{{I}{C|CR}}};
/* where MRa is either MR1:0 or MR3:2 */
```

Function

These instructions provide a 32-bit multiply-accumulate operation that multiplies the fields in registers *Rm* and *Rn*, then adds the result to (or subtracts the result from) the specified MR register value. The result is 80 bits and is placed in the MR accumulation register, which must be the same MR register that provided the input. Since MR1:0 and MR3:2 are only 64 bits, the extra 16 bits (for the 80-bit accumulation) are stored in the MR4 register. MR4[15:0] holds the extra bits when the destination of the multiply-accumulate is MR1:0, and MR4[31:16] holds the extra bits when the destination of the multiply-accumulate is MR3:2. Saturation is always active.

The previous diagram is for MR3:2 += *Rm* * *Rn* (I) (U) (C) (CR). If MR1:0 += *Rm* * *Rn* (I) (U) (C) (CR) were used, the additional 16 bits for the accumulation would be placed in the lower half of the MR4 register.

Status Flags

MZ	Unaffected
MN	Unaffected
MV	Unaffected
MU (MUS)	Unaffected
MOS	Set according to the final result of the sum and data type. (see following)

Multiplier Instructions

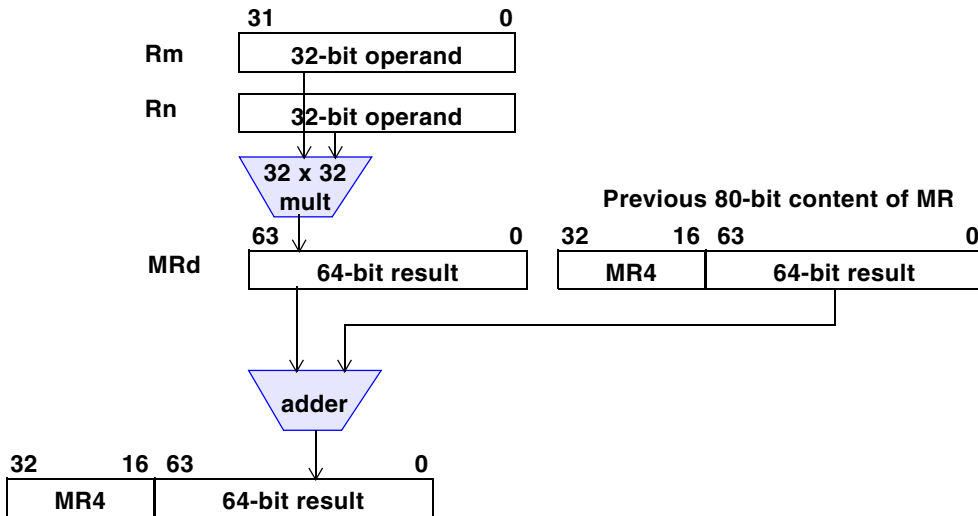


Figure 8-47. Multiply Accumulate (Normal Word) Data Flow

For result, MOS is set according to:

- Signed fractional – if final result is equal to or larger than 2^{15} , or if less than -2^{15}
- Signed integer – if final result is equal to or larger than 2^{79} , or if less than -2^{79}
- Unsigned fractional (add only) – if final result is equal to or large than 2^{16}
- Unsigned integer (add only) – if final result is equal to or larger than 2^{80}

Options

- (U) *Rm, Rn* unsigned (add only)
- (I) Integer

(C) Clear MR prior to accumulation

(CR) Clear and Round

See “Multiplier Instruction Options” on page 4-8 for more details about available options.

Example

```
/*{X|Y|XY}MRa += Rm * Rn {{U}{I}{C|CR}} ; */
R2 = -2 ;;
R1 = 5 ;;
MR3:2 += R1 * R2 (I) ;;
```

In this example, if the previous contents of the MR3:2 register was 7, the new contents would be -3. No flags would be set.

```
R0 = 0xF0060054 ;;
R1 = 0xF0356820 ;;
R2 = 0xF036AC42 ;;
R3 = 0xD721C843 ;;
MR3:2 += R0 * R2 (UI) ;;
MR3:2 += R0 * R2 (UI) ;;
```

This example shows how the MR4 register is used to store the extra bits for the 80 bit accumulation. The contents of the MR4:0 registers after the second multiply-accumulate are:

```
MR0 = 0x00000000
MR1 = 0x00000000
MR2 = 0x76F90B50
MR3 = 0xC271C629
MR4 = 0x00010000

R0 = 0xF0060054 ;;
R1 = 0xF0356820 ;;
R2 = 0xF036AC42 ;;
R3 = 0xD721C843 ;;
```

Multiplier Instructions

```
MR1:0 += R0 * R2 (UI) ;;  
MR1:0 += R0 * R2 (UI) ;;
```

This example is identical to the previous, except the MR1:0 registers are used instead of the MR3:2 registers. The contents of the MR4:0 registers after the second multiply-accumulate are:

```
MR0 = 0x76F90B50  
MR1 = 0xC271C629  
MR2 = 0x00000000  
MR3 = 0x00000000  
MR4 = 0x00000001
```

Notice the difference in placement of the additional 16 bits in the MR4 register from the previous example.

```
/* {X|Y|XY}MRa -= Rm * Rn {{I}{C|CR}} ; */  
R0 = 2;;  
R1 = 5;;  
R2 = -10;;  
R3 = 6;;  
MR3:2 -= R0 * R1 (I);;  
MR3:2 -= R2 * R3 (I);;
```

The contents of the MR4:0 registers after the first multiply-accumulate instruction are:

```
MR0 = 0x00000000  
MR1 = 0x00000000  
MR2 = 0xFFFFFFFF6  
MR3 = 0xFFFFFFFFF  
MR4 = 0xFFFF0000
```

After the second multiply-accumulate, the results are:

```
MR0 = 0x00000000  
MR1 = 0x00000000
```

```
MR2 = 0x00000032
MR3 = 0x00000000
MR4 = 0x00000000

R0 = 2;;
R1 = 5;;
R2 = -10;;
R3 = 6;;
MR1:0 -= R0 * R1 (I);;
MR1:0 -= R2 * R3 (I);;
```

The contents of the MR4:0 registers after the first multiply-accumulate instruction are:

```
MR0 = 0xFFFFFFFF6
MR1 = 0xFFFFFFFF
MR2 = 0x00000000
MR3 = 0x00000000
MR4 = 0x0000FFFF
```

After the second multiply-accumulate, the results are:

```
MR0 = 0x00000032
MR1 = 0x00000000
MR2 = 0x00000000
MR3 = 0x00000000
MR4 = 0x00000000
```

Multiplier Instructions

Multiply-Accumulate/Move (Dual Operation, Normal Word)

Syntax

```
{X|Y|XY}Rs = MRa, MRa += Rm * Rn {{U}{I}{C}} ; dual operation  
{X|Y|XY}Rsd = MRa, MRa += Rm * Rn {{U}{I}{C}} ; dual operation  
/* where MRa is either MR1:0 or MR3:2 */
```

Function

This is a 32-bit multiply-accumulate that multiplies the 32-bit value in register R_m with the value in R_n , adds this result to MRa and transfers the previous contents of MRa to Rs . The clear option (C) clears MRa register after writing its value to Rs or Rsd , and before adding it to the new multiplication result. The MR to register file transfer is always truncated.

When the register that serves as destination in the transfer is a register pair Rsd , a 64-bit accumulation value is transferred to Rsd (see “[Multiplier Operations](#)” on page 4-4). When the destination register is a single register Rs , the portion of the 64-bit accumulation value that is transferred depends on the data type—when integer, the lower portion of the value is transferred; when fraction, the upper.

Regarding the multiply-accumulate operation, the extra 16 bits (for the 80-bit accumulation) are stored in the MR4 register. $MR4[15:0]$ holds the extra bits when the destination of the multiply-accumulate is MR1:0, and $MR4[31:16]$ holds the extra bits when the destination of the multiply-accumulate is MR3:2.

Status Flags

MUS	Unaffected
MZ	Unaffected
MN	Unaffected
MV	Unaffected

MOS Set according to the final result of the sum and data type. (see following)

For result, MOS is set according to:

- Signed fractional – if final result is equal to or larger than 2^{15} , or if less than -2^{15}
- Signed integer – if final result is equal to or larger than 2^{79} , or if less than -2^{79} . Also set for $Rs = MRa$, $MRa += Rm * Rn$ (Rs single, MRa double) if the old value of MRa is equal to or larger than 2^{31} or smaller than -2^{31}
- Unsigned fractional – if final result is equal to or larger than 2^{16}
- Unsigned integer – if final result is equal to or larger than 2^{80} . Also set for $Rs = MRa$, $MRa += Rm * Rn$ (Rs single, MRa double) if the old value of MRa is equal to or larger than 2^{32} or smaller than -2^{31}

Other flags are unaffected.

Options

- | | |
|-----|---------------------------------------------------------------------|
| (U) | Rm, Rn unsigned |
| (I) | Integer |
| (C) | Clear MR after transfer to register file, and prior to accumulation |

See “Multiplier Instruction Options” on page 4-8 for more details about available options.

Multiplier Instructions

Example

Listing 8-1. Example 1

```
R0 = 2 ;;  
R1 = 5 ;;  
R2 = 10 ;;  
R3 = 6 ;;  
R4 = MR1:0, MR1:0 += R0 * R1 (UIC) ;;  
R4 = MR1:0, MR1:0 += R2 * R3 (UIC) ;;
```

With the example in [Listing 8-1](#), after the execution of the first instruction containing the register move and multiply, the results of the MR4:0 register are (assuming previous contents of MR4:0 were zero):

```
MR0 = 0x0000000A  
MR1 = 0x00000000  
MR2 = 0x00000000  
MR3 = 0x00000000  
MR4 = 0x00000000
```

The R4 register is loaded with the value 0x00000000.

After execution of the second instruction, the MR4:0 registers contain:

```
MR0 = 0x0000003C  
MR1 = 0x00000000  
MR2 = 0x00000000  
MR3 = 0x00000000  
MR4 = 0x00000000
```

The R4 register is loaded with the value 0x0000000A.

Listing 8-2. Example 2

```
R0 = 0x12345678 ;;
R1 = 0x87654321 ;;
R2 = 0xA74EF254 ;;
R3 = 0xB4ED9032 ;;
R4 = MR3:2, MR3:2 += R0 * R1 ;;
R4 = MR3:2, MR3:2 += R2 * R3 ;;
```

With the example in [Listing 8-2](#), after the execution of the first instruction containing the register move and multiply, the results of the MR4:0 register are (assuming previous contents of MR4:0 were zero):

```
MR0 = 0x00000000
MR1 = 0x00000000
MR2 = 0xE1711AF0
MR3 = 0xEED8ED1A
MR4 = 0xFFFF0000
```

The R4 register is loaded with the value 0x00000000.

After execution of the second instruction, the MR4:0 registers contain:

```
MR0 = 0x00000000
MR1 = 0x00000000
MR2 = 0xDC6E43C0
MR3 = 0x22DD717B
MR4 = 0x00000000
```

The R4 register is loaded with the value 0xEED8ED1A.

Listing 8-3. Example 3

```
R0 = 2000;;
R1 = 4000;;
R2 = 3000;;
```

Multiplier Instructions

```
R3 = -5000;;  
R5:4 = MR1:0, MR1:0 += R0 * R1 (I);;  
R5:4 = MR1:0, MR1:0 += R2 * R3 (I);;  
R5:4 = MR1:0, MR1:0 += R0 * R1 (I);;
```

With the example in [Listing 8-3](#), after the execution of the first instruction containing the register move and multiply, the results of the MR4:0 register are (assuming previous contents of MR4:0 were zero):

```
MR0 = 0x007A1200  
MR1 = 0x00000000  
MR2 = 0x00000000  
MR3 = 0x00000000  
MR4 = 0x00000000
```

The R5 and R4 registers are loaded with the value 0x00000000.

After execution of the second instruction, the MR4:0 registers contain:

```
MR0 = 0xFF953040  
MR1 = 0xFFFFFFFF  
MR2 = 0x00000000  
MR3 = 0x00000000  
MR4 = 0x0000FFFF
```

The R5 register is loaded with the value 0x00000000, and the R4 register is loaded with 0x007A1200.

After execution of the third instruction, the MR4:0 registers contain:

```
MR0 = 0x000F4240  
MR1 = 0x00000000  
MR2 = 0x00000000  
MR3 = 0x00000000  
MR4 = 0x00000000
```

The R5 register is loaded with the value 0xFFFFFFFF, and the R4 register is loaded with 0xFF953040.

Listing 8-4. Example 4

```

R0 = 100 ;;
R1 = 50 ;;
R2 = 300 ;;
R3 = 70 ;;
R5:4 = MR1:0, MR1:0 += R0 * R1 ;;
R5:4 = MR1:0, MR1:0 += R2 * R3 ;;

```

Assume the previous contents of the MR4:0 registers are as follows:

```

MR0 = 0xFFFFF500
MR1 = 0xFFFFFFFF
MR2 = 0x00000000
MR3 = 0x00000000
MR4 = 0x00000000

```

With the example in [Listing 8-4](#), after the execution of the first instruction containing the register move and multiply, the results of the MR4:0 register are:

```

MR0 = 0x00000888
MR1 = 0x00000000
MR2 = 0x00000000
MR3 = 0x00000000
MR4 = 0x00000001

```

The R5 register is loaded with the value 0xFFFFFFFF, and the R4 register is loaded with the value 0xFFFFF500.

After execution of the second instruction, the MR4:0 registers contain:

```

MR0 = 0x00005A90
MR1 = 0x00000000
MR2 = 0x00000000
MR3 = 0x00000000
MR4 = 0x00000001

```

Multiplier Instructions

The R5 register is loaded with the value 0xFFFFFFFF, and the R4 register is loaded with the value 0xFFFFFFFF. As the contents of the MR4 register have saturated, the maximum 64 bit resolution that can be represented in the double R5:4 double register. The MOS flag is not set.

Listing 8-5. Example 5

```
R0 = 0x4236745D ;;
R1 = 0x53ACBE34 ;;
R2 = 0xF38D153C ;;
R3 = 0x4129EDA1 ;;
R5:4 = MR3:2, MR3:2 += R0 * R1 ;;
R5:4 = MR3:2, MR3:2 += R2 * R3 ;;
```

Assume the previous contents of the MR4:0 registers are as follows:

```
MR0 = 0x00000000
MR1 = 0x00000000
MR2 = 0x12345678
MR3 = 0xFFFFFFFF
MR4 = 0x7FFF0000
```

With the example in [Listing 8-5](#), after the execution of the first instruction containing the register move and multiply, the results of the MR4:0 register are:

```
MR0 = 0x00000000
MR1 = 0x00000000
MR2 = 0xFFFFFFFF
MR3 = 0xFFFFFFFF
MR4 = 0x7FFF0000
```

The R5 register is loaded with the value 0x7FFFFFFF, and the R4 register is loaded with the value 0xFFFFFFFF. This is due to the additional 16 bit overflow bits being set so saturation occurs on the 64 bit result written to registers R5:4. The MOS bit is also set.

After execution of the second instruction, the MR4:0 registers contain:

MR0 = 0x00000000

MR1 = 0x00000000

MR2 = 0xD5FDCD77

MR3 = 0xF9A990DC

MR4 = 0x7FFF0000

The R5 register is loaded with the value 0x7FFFFFFF, and the R4 register is loaded with 0xFFFFFFFF. The MOS flag remains set. Although this instruction did not result in the MOS flag being set, this flag is a sticky flag and must be explicitly cleared.

Multiplier Instructions

Multiply (Quad-Short Word)

Syntax

$\{X|Y|XY\}Rsd = Rmd * Rnd \{(\{U\}\{I\}\{T\}\{S\})\} ;$

$\{X|Y|XY\}Rsq = Rmd * Rnd \{(\{U\}\{I\})\} ;$

Function

This is a 16-bit quad fixed-point multiplication of the four shorts in register Rm with the four shorts in Rn . For fractional operands, if rounding is specified by the absence of T , the result is rounded. (Note that option T does not apply to integer data). The result is placed in register Rs .

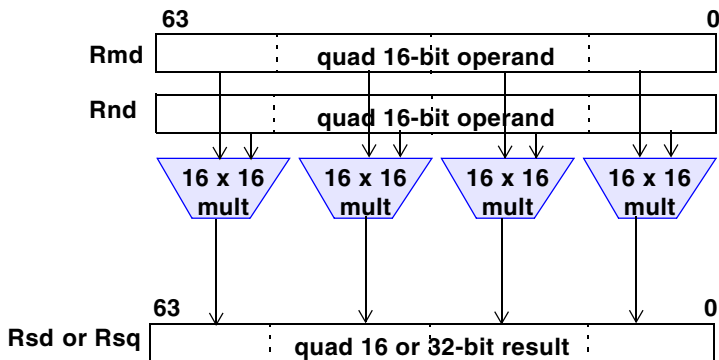


Figure 8-48. Multiply (Quad Short Word) Data Flow

Status Flags

MZ Set if all bits in result are zero in one of the results

MN Set if result is negative in one of the results

MV (MOS) Set according to the data format, under the following conditions (MOS unchanged if MV is cleared):

$Rsq = Rmd * Rnd; \Rightarrow$ no overflow

Fractional \Rightarrow No overflow¹

Signed integer \Rightarrow Upper 17 bits of M are not all zeros or all ones

Unsigned integer \Rightarrow Upper 16 bits of M are not all zeros

MU (MUS) Cleared

Options

() Rm, Rn signed fractional, result is rounded (if result is quad short)

(U) Rm, Rn unsigned

(I) Operands are integer

(T) Result is truncated (only for fractional if result is quad short)

(S) Saturate (only for integer)

See “Multiplier Instruction Options” on page 4-8 for more details of the available options.

¹ Except when multiplying the most negative fraction times itself, in which case MV and MOS are set.

Multiplier Instructions

Example

```
XR0 = 0x42861234 ;;
XR1 = 0x782F4217 ;;
XR2 = 0x8AC90FEA ;;
XR3 = 0x724D76EB ;;
XR5:4 = R1:0 * R3:2 (T) ;; /* xR5 = 6B52 3D66 , xR4 = C314 0243 */
```

This example specifies four 16-bit multiplications of signed fractional data resulting in four 16-bit truncated signed fractional results. The `xMN` flag is set in this example as the 16-bit result stored in the upper 16 bits of the `xR4` register is negative. When flags are set in this manner, there is no indication as to which operand multiply was the cause of the flag being set. The flag indicates that at least one of the results produced a negative result.

```
XR0 = 0x00060004 ;;
XR1 = 0x00030007 ;;
XR2 = 0x00090000 ;;
XR3 = 0x00050008 ;;
XR7:4 = R1:0 * R3:2 (I) ;;
/* xR7 = 0x0000000F, xR6 = 0x00000038, xR5 = 0x00000036
xR4 = 0x00000000 */
```

This example specifies four 16-bit multiplies of signed integer 16-bit values and results in four 32-bit signed integer values. The `xMZ` flag is set in this case, as one of the results is equal to zero.

Multiply-Accumulate (Quad-Short Word)

Syntax

```
{X|Y|XY}MRb += Rmd * Rnd {{{U}{I}{C}}};
/* where MRb is either MR1:0, MR3:2 */
{X|Y|XY}MR3:0 += Rmd * Rnd {{{U}{I}{C|CR}}}; {X|Y|XY}Rsd = MRb,
MRb += Rmd * Rnd {{{I}{C}}}; dual operation
/* where MRb is either MR1:0, MR3:2, or MR3:0 */
```

Function

This is a 16-bit quad fixed-point multiply-accumulate operation that multiplies the four short words in register pair *Rmd* with the four shorts in *Rnd*, and adds the four results element-wise to the four values in the specified MR register. The results are placed in the MR accumulation register, which must be the same MR register that provided the input.

When using MR3:0, the four multiplication results are accumulated in the four MR registers at full 32-bit precision. When using either MR3:2 or MR1:0, the four multiplier results are accumulated in two MR registers at 16-bit precision.

The extra bits from the multiplications are stored in MR4. Saturation is always active. See [Figure 8-49](#).

Refer to “[Multiplier Result Overflow \(MR4\) Register](#)” on page 4-17 for a description of the fields in MR4.

Status Flags

MZ	Unaffected
MN	Unaffected
MV	Unaffected
MU (MUS)	Unaffected

Multiplier Instructions

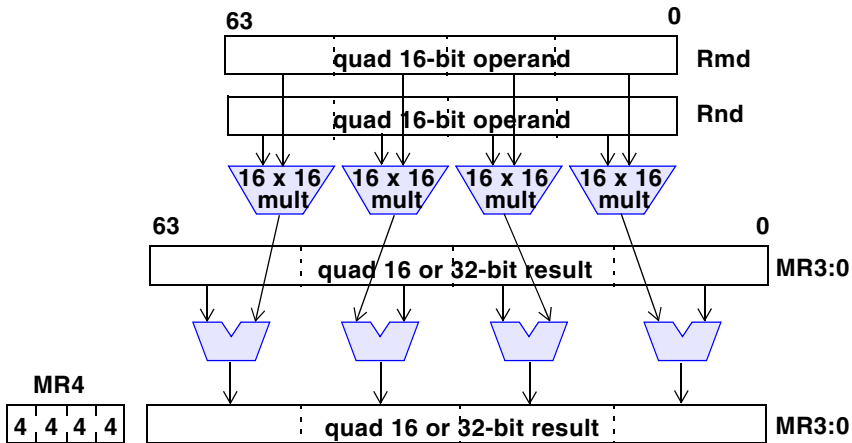


Figure 8-49. Quad 16-Bit Result

MOS Set according to the final result of the sum and data type and size. (see following)

For word result ($MR3:0 = Rmd * Rnd$), MOS is set according to:

- Signed fractional – if final result is equal to or larger than 2^7 , or if less than -2^7
- Signed integer – if final result is equal to or larger than 2^{39} , or if less than -2^{39}
- Unsigned fractional – if final result is equal to or larger than 2^8
- Unsigned integer – if final result is equal to or larger than 2^{40}

For short result $MR1:0/3:2 = Rmd * Rnd$, MOS is set according to:

- Signed fractional – if final result is equal to or larger than 2^3 , or if less than -2^3
- Signed integer – if final result is equal to or larger than 2^{19} , or if less than -2^{19} , or if the multiplication intermediate result is equal to or larger than 2^{15} or smaller than -2^{15}
- Unsigned fractional – if final result is equal to or larger than 2^4
- Unsigned integer – if final result is equal to or larger than 2^{20} or if the multiplication intermediate result is equal to or larger than 2^{16}

Options

(U)	<i>Rm, Rn</i> unsigned
(I)	Integer
(C)	Clear MR prior to accumulation
(CR)	Clear and Round

See “Multiplier Instruction Options” on page 4-8 for more details about available options.

Example

```
R0 = 0x70000004 ;;
R1 = 0x06000002 ;;
R2 = 0xF0000005 ;;
R3 = 0xD0000006 ;;
MR3:0 += R1:0 * R3:2 (UI) ;;
MR3:0 += R1:0 * R3:2 (UI) ;;
```

Multiplier Instructions

This example shows a quad 16-bit multiply-accumulate where the result of each multiply is stored at full 32-bit precision. After execution of the second multiply-accumulate, the 32-bit precision is not enough and the MR4 register is used for the overflow.

The results of the MR4:0 registers after the second multiply-accumulate are:

```
MR0 = 0x00000028
MR1 = 0x4A000000
MR2 = 0x00000018
MR3 = 0x09C00000
MR4 = 0x00000100
```

No flags were set during this example.

```
R0 = 0x70000004 ;;
R1 = 0x06000002 ;;
R2 = 0xF0000005 ;;
R3 = 0xD0000006 ;;
MR3:2 += R1:0 * R3:2 (UI) ;;
MR3:2 += R1:0 * R3:2 (UI) ;;
```

The results of the MR4:0 registers after the second multiply-accumulate are as follows:

```
MR0 = 0x00000000
MR1 = 0x00000000
MR2 = 0xFFFFE028
MR3 = 0xFFFFE018
MR4 = 0x10100000
```

The MOS flag is set on both multiply-accumulate instructions in this example due to the saturation of two of the 16-bit multiplies.

The next example is identical to the previous, except it uses the MR1:0 registers for the result instead of MR3:2.

```
R0 = 0x70000004;;
R1 = 0x06000002;;
R2 = 0xF0000005;;
R3 = 0xD0000006;;
MR1:0 += R1:0 * R3:2 (UI);;
MR1:0 += R1:0 * R3:2 (UI);;
```

The results of the MR4:0 registers after the second multiply-accumulate are as follows:

```
MR0 = 0xFFFFE028
MR1 = 0xFFFFE018
MR2 = 0x00000000
MR3 = 0x00000000
MR4 = 0x00001010
```

The MOS flag is set on both multiply-accumulate instructions in this example due to the saturation of two of the 16-bit multiplies.

Notice this time how the positioning of the set bits in the MR4 register differs from previous examples due to the MR4 register only containing 4 bits worth of overflow for the accumulation (20 bits in total).

Multiplier Instructions

Multiply-Accumulate (Dual Operation, Quad-Short Word)

Syntax

```
{X|Y|XY}Rsd = MRb, MRb += Rmd * Rnd {{I}{C}} ; dual operation  
/* where MRb is either MR1:0, MR3:2, or MR3:0 */
```

Function

This is a 16-bit quad fixed-point multiply-accumulate operation that multiplies the four short words in register pair *Rmd* with the four shorts in *Rnd*, adds the four results element-wise to the four values in MR, and transfers the contents of *MRa* into *Rsd*.

When using MR[3:0], the four multiplication results are accumulated in the four MR registers at full 32-bit precision. When using either MR[3:2] or MR[1:0], the four multiplier results are accumulated in two MR registers at 16-bit precision.

The clear option (C) prevents the old value of *MRa* from being added to the new multiplication result. The operations are always saturated. The MR to register file transfer is always truncated.

For the dual result (uses MR3:2, or MR1:0), the extra 4-bit bits are stored in MR4, according to the specification in [“Multiplier Result Overflow \(MR4\) Register” on page 4-17](#). For the quad result (uses MR3:0), the extra 8-bit bits are stored in MR4, according to the specification in [“Multiplier Result Overflow \(MR4\) Register” on page 4-17](#). Saturation is always active.

When the register that serves as destination in the transfer is a register quad (MR3:0), four 40-bit accumulation values are truncated according to data type (integer or fractional) and transferred to *Rsd*.

When the MR registers that serve as a source for the transfer are an MR register pair (MR1:0, MR3:2), the four 20-bit accumulation values are transferred to *Rsd*.

Status Flags

MZ	Unaffected
MN	Unaffected
MV	Unaffected
MU	Unaffected
MOS	Set according to the final result of the sum and data type and size. (see following)

For word result ($MR3:0 \text{ += } Rmd * Rnd$), MOS is set according to:

- Signed fractional – if final result is equal to or larger than 2^7 , or if less than -2^7
- Signed integer – if final result is equal to or larger than 2^{39} , or if less than -2^{39}
- Unsigned fractional – if final result is equal to or larger than 2^8
- Unsigned integer – if final result is equal to or larger than 2^{40}

For short result ($MR1:0/3:2 \text{ += } Rmd * Rnd$), MOS is set according to:

- Signed fractional – if final result is equal to or larger than 2^3 , or if less than -2^3
- Signed integer – if final result is equal to or larger than 2^{19} , or if less than -2^{19} , or if the multiplication intermediate result is equal to or larger than 2^{15} or smaller than -2^{15}
- Unsigned fractional – if final result is equal to or larger than 2^4
- Unsigned integer – if final result is equal to or larger than 2^{20} or if the multiplication intermediate result is equal to or larger than 2^{16}

Multiplier Instructions

Options

- (U) *Rm, Rn* unsigned
- (I) Integer
- (C) Clear MR after transfer to register file, and prior to accumulation

See “Multiplier Instruction Options” on page 4-8 for more details about available options.

Example

```
R0 = 0x00080007 ;;
R1 = 0x00060005 ;;
R2 = 0x00040003 ;;
R3 = 0x00020001 ;;
R5:4 = MR1:0, MR1:0 += R1:0 * R3:2 (UIC) ;;
R5:4 = MR1:0, MR1:0 += R1:0 * R3:2 (UI) ;;
```

If the previous contents of the MR register were:

```
MR0 = 0x00800050
MR1 = 0x01000200
MR2 = 0x00300040
MR3 = 0x00500060
MR4 = 0x00000001
```

After execution of the first register transfer and multiply instruction, the values of all modified registers are:

```
R4 = 0x0080FFFF
R5 = 0x01000200
MR0 = 0x00200015
MR1 = 0x000C0005
MR2 = 0x00300040
```



```
MR3 = 0x00500060
MR4 = 0x00000000
```

After execution of the next instruction, the above registers are set to the following:

```
R4 = 0x00200015
R5 = 0x000C0005
MR0 = 0x0040002A
MR1 = 0x0018000A
MR2 = 0x00300040
MR3 = 0x00500060
MR4 = 0x00000000
```

No flags are set during the process.

```
R0 = 0x00080007 ;;
R1 = 0x00060005 ;;
R2 = 0x00040003 ;;
R3 = 0x00020001 ;;
R5:4 = MR3:2, MR3:2 += R1:0 * R3:2 (UIC) ;;
R5:4 = MR3:2, MR3:2 += R1:0 * R3:2 (UI) ;;
```

If the previous contents of the MR register were:

```
MR0 = 0x00800050
MR1 = 0x01000200
MR2 = 0x00300040
MR3 = 0x00500060
MR4 = 0x00000001
```

After execution of the first register transfer and multiply instruction, the values of all modified registers are:

```
R4 = 0x00300040
R5 = 0x00500060
MR0 = 0x00800050
```

Multiplier Instructions

```
MR1 = 0x01000200
MR2 = 0x00200015
MR3 = 0x000C0005
MR4 = 0x00000000
```

After execution of the next instruction, the above registers are set to the following:

```
R4 = 0x00200015
R5 = 0x000C0005
MR0 = 0x00800050
MR1 = 0x01000200
MR2 = 0x0040002A
MR3 = 0x0018000A
MR4 = 0x00000000
```

No flags are set during the process.

```
R0 = 0x85006300 ;;
R1 = 0x00060005 ;;
R2 = 0x00100020 ;;
R3 = 0x00020001 ;;
R5:4 = MR3:0, MR3:0 += R1:0 * R3:2 (UIC) ;;
R5:4 = MR3:0, MR3:0 += R1:0 * R3:2 (UI) ;;
```

If the previous contents of the MR register were:

```
MR0 = 0x00800050
MR1 = 0x01000200
MR2 = 0x00000040
MR3 = 0x00000060
MR4 = 0x00000001
```

After execution of the first register transfer and multiply instruction, the values of all modified registers are:

```
R4 = 0xFFFFFFFF
R5 = 0x00600040
MR0 = 0x00000015
MR1 = 0x00000020
MR2 = 0x00000005
MR3 = 0x0000000C
MR4 = 0x00000000
```

After execution of the next instruction, the above registers are set to the following:

```
R4 = 0x00200015
R5 = 0x000C0005
MR0 = 0x0000002A
MR1 = 0x00000040
MR2 = 0x0000000A
MR3 = 0x00000018
MR4 = 0x00000000
```

No flags are set during the process.

Multiplier Instructions

Complex Multiply-Accumulate (Short Word)

Syntax

```
{X|Y|XY}MRa += Rm ** Rn {{(I){C|CR}{J}}};  
/* where MRa is either MR1:0 or MR3:2 */
```

Function

This is a 16-bit complex multiply-accumulate operation that multiplies the complex value in register *Rm* with the complex value in *Rn* and adds the result to the specified MR registers. The result is placed in the MR accumulation register, which must be the same MR register that provided the input. Saturation is always active. See “Complex Conjugate Option” on page 4-16 and Figure 8-50.

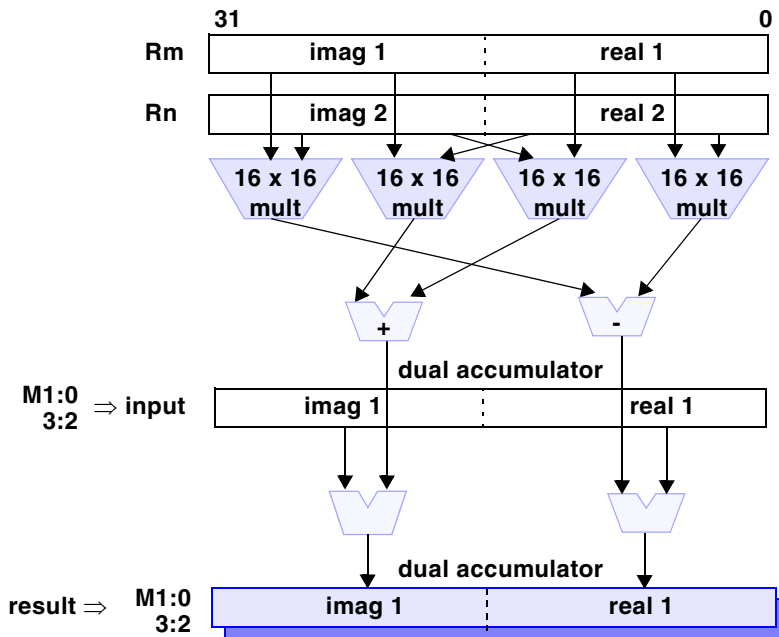


Figure 8-50. Complex Multiply-Accumulate (Short Word) Data Flow

There are eight overflow bits for each accumulated real and imaginary number. With $MR1:0 += Rm ** Rn$, bits 7:0 of the MR4 register contain the overflow bits for the real part of the accumulation, and bits 15:8 contain the overflow for the imaginary part of the accumulation. With $MR3:2 += Rm ** Rn$, bits 23:16 of the MR4 register contain the overflow bits for the real part of the accumulation, and bits 31:24 contain the overflow for the imaginary part of the accumulation.

It is important to note that the conjugate option (J) does not perform the conjugate of the result. Looking at the complex multiply $(A + jB) * (C + jD)$, normally the conjugate would be the result of $(A - jB) * (C - jD)$. On the TigerSHARC processor, the conjugate option provides $(A + jB) * (C - jD)$. The result of a conjugate complex multiply is the original first value multiplied by the complex conjugate of the second value.

Status Flags

MZ	Unaffected
MN	Unaffected
MU	Unaffected
MUS	Unaffected
MOS	Set according to the final result of the sum and data type and size. (see following)

For word result ($MR3:0 = Rmd * Rnd$):

- Signed fractional – if any part of the final result (real or imaginary) is equal to or larger than 2^7 , or if less than -2^7
- Signed integer – if any part of the final result (real or imaginary) is equal to or larger than 2^{39} , or if less than -2^{39} .

Multiplier Instructions

- Unsigned fractional – if any part of the final result (real or imaginary) is equal to or larger than 2^8
- Unsigned integer – if any part of the final result (real or imaginary) is equal to or larger than 2^{40}

Options

(I)	Integer
(C)	Clear MR
(CR)	Clear and Round
(J)	Multiplication of value in Rm times the complex conjugate of the value in Rn

See [“Multiplier Instruction Options” on page 4-8](#) for more details about available options.

Example

```
R0 = 0xFFFF50009;;  
R1 = 0x00060004;;  
MR1:0 += R0 ** R1 (IC);;
```

If the previous contents of the MR register were:

```
MR0 = 0x00800050  
MR1 = 0x01000200  
MR2 = 0x00000040  
MR3 = 0x00000060  
MR4 = 0x00000001
```

After execution of the complex multiply-accumulate instruction, the values of all modified registers are:

```
MR0 = 0x00000066
MR1 = 0x0000000A
MR2 = 0x00000040
MR3 = 0x00000060
MR4 = 0x00000000
```

No flags are set.

```
R0 = 0xFFF50009;;
R1 = 0x00060004;;
MR3:2 += R0 ** R1 (ICJ);;
```

If the previous contents of the MR register were:

```
MR0 = 0x00800050
MR1 = 0x01000200
MR2 = 0x00000040
MR3 = 0x00000060
MR4 = 0x00000001
```

After execution of the complex multiply-accumulate instruction, the values of all modified registers are:

```
MR0 = 0x00800050
MR1 = 0x01000200
MR2 = 0xFFFFFFFFE2
MR3 = 0xFFFFFFFF9E
MR4 = 0xFFFFF0001
```

No flags are set during the operation.

Multiplier Instructions

Complex Multiply-Accumulate/Move (Dual Operation, Short Word)

Syntax

```
{X|Y|XY}Rs = MRa, MRa += Rm ** Rn {{I}{C}{J}} ; dual operation  
{X|Y|XY}Rsd = MRa, MRa += Rm ** Rn {{I}{C}{J}} ; dual operation  
/* where MRa is either MR1:0 or MR3:2 */
```

Function

This is a 16-bit complex multiply-accumulate operation that multiplies the complex value in register Rm with the complex value in Rn , adds the result to the specified MR registers, and transfers the previous contents of MRa in Rs . The `Clear` option prevents the old value of MRa from being added to the new multiplication result. The operations are always truncated.

The MR to register file transfer moves a 64-bit complex value stored in register pair MR3:2 or MR1:0 (32-bit real, and 32-bit imaginary components) into destination register Rs or into register pair Rsd . In the case of destination register Rs , the 32-bit real component in MR2 or MR0 is transferred to the 16 LSBs of Rs , and the imaginary component in MR3 or MR1 to the 16 MSBs of Rs . In the case of destination register pair Rsd , the 32-bit real component in MR2 or MR0 is transferred to the lower register in pair Rsd , and the imaginary component in MR3 or MR1 to the upper register in pair Rsd .

M is one of the 40-bit multiplication results that is written into MR3:0. The extra 8-bit bits of the four results are stored in MR4. Saturation is always active.

For an integer transfer to the single Rs register, the lower 16 bits of the real and imaginary parts are transferred to the Rs register. If the 32-bit real signed integer component stored in MR2 or MR0 exceeds 2^{15} or -2^{16} , the value to be transferred has exceeded the value that can be represented by 16 bits, and saturation occurs. The same saturation rule applies for

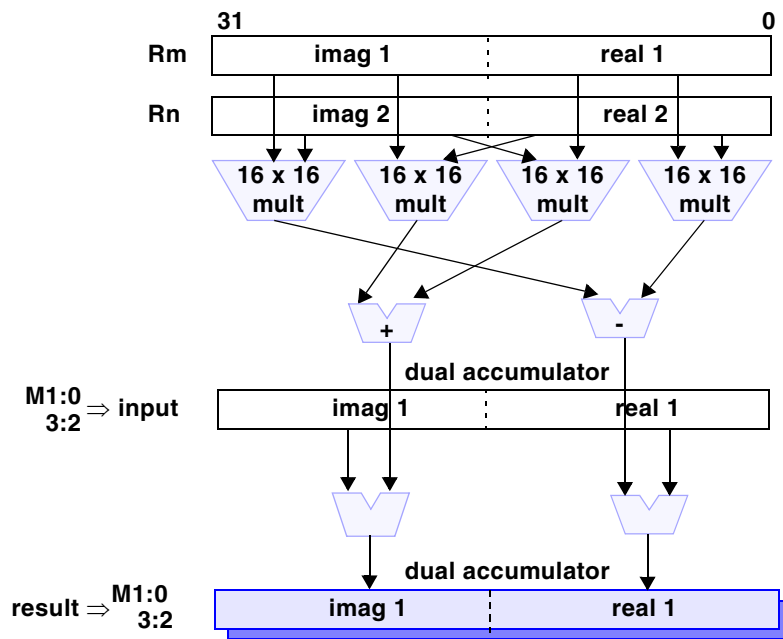


Figure 8-51. Complex Multiply-Accumulate With Transfer MR Register (Dual Operation, Short Word) Data Flow

unsigned integer data, signed/unsigned fractional data, and the imaginary part stored in MR3 and MR1. For fractional data transfers, the upper 16 bits of both the real and imaginary data are transferred to the register.

Status Flags

MZ	Unaffected
MN	Unaffected
MV	Unaffected
MU	Unaffected

Multiplier Instructions

MUS	Unaffected
MV (MOS)	Set according to the final result of the sum and data type and size. (see following)

For word result ($MR3:0 = Rmd * Rnd$), MOS set according to:

- Signed fractional – if any part of the final result (real or imaginary) is equal to or larger than 2^7 , or if less than -2^7
- Signed integer – if any part of the final result (real or imaginary) is equal to or larger than 2^{39} , or if less than -2^{39}
- Unsigned fractional – if any part of the final result (real or imaginary) is equal to or larger than 2^8
- Unsigned integer – if any part of the final result (real or imaginary) is equal to or larger than 2^{40}

Options

(I)	Integer
(C)	Clear MR after transfer to register file, and prior to accumulation
(J)	Conjugate

See “[Multiplier Instruction Options](#)” on page 4-8 for more details about available options.

Example

```
R0 = 0xFFFF50009 ;;
R1 = 0x00060004 ;;
R2 = 0x00100007 ;;
R3 = 0x001AFFF6 ;;
R4 = MR1:0, MR1:0 += R0 ** R1 (IC) ;;
R4 = MR1:0, MR1:0 += R0 ** R1 (I) ;;
```

If the previous contents of the MR register were:

```
MR0 = 0x00800050
MR1 = 0x01000200
MR2 = 0x00000040
MR3 = 0x00000060
MR4 = 0x00000001
```

After execution of the first register transfer and complex multiply-accumulate instruction, the values of all modified registers are:

```
R4 = 0x7FFF7FFF
MR0 = 0x00000066
MR1 = 0x0000000A
MR2 = 0x00000040
MR3 = 0x00000060
MR4 = 0x00000000
```

No flags are set during the operation.

After the next instruction is executed, the contents become:

```
R4 = 0x000A0066
MR0 = 0x000000CC
MR1 = 0x00000014
MR2 = 0x00000040
MR3 = 0x00000060
MR4 = 0x00000000
```

No flags are set.

```
R0 = 0xFFFF50009;;
R1 = 0x00060004;;
R2 = 0x00100007;;
R3 = 0x001AFFF6;;
R4 = MR3:2, MR3:2 += R0 ** R1 (ICJ);;
R4 = MR3:2, MR3:2 += R0 ** R1 (IJ);;
```

Multiplier Instructions

If the previous contents of the MR register were:

```
MR0 = 0x00800050
MR1 = 0x01000200
MR2 = 0x00000040
MR3 = 0x00000060
MR4 = 0x00000001
```

After execution of the first register transfer and complex multiply-accumulate instruction, the values of all modified registers are:

```
R4 = 0x00600040
MR0 = 0x00800050
MR1 = 0x01000200
MR2 = 0xFFFFFFFFE2
MR3 = 0xFFFFFFFF9E
MR4 = 0xFFFF0001
```

No flags are set during the operation.

After the next instruction is executed, the contents become:

```
R4 = 0xFF9EFFE2
MR0 = 0x00800050
MR1 = 0x01000200
MR2 = 0xFFFFF4C
MR3 = 0xFFFFF3C
MR4 = 0xFFFF0001
```

No flags are set.

```
R0 = 0xFFF50009;;
R1 = 0x00060004;;
R2 = 0x00100007;;
R3 = 0x001AFFF6;;
R5:4 = MR1:0, MR1:0 += R0 ** R1 (IC);;
R5:4 = MR1:0, MR1:0 += R0 ** R1 (I);;
```

If the previous contents of the MR register were:

```
MR0 = 0x00800050
MR1 = 0x01000200
MR2 = 0x00000040
MR3 = 0x00000060
MR4 = 0x00000001
```

After execution of the first register transfer and complex multiply-accumulate instruction, the values of all modified registers are:

```
R4 = 0x7FFFFFFF
R5 = 0x01000200
MR0 = 0x00000066
MR1 = 0x0000000A
MR2 = 0x00000040
MR3 = 0x00000060
MR4 = 0x00000000
```

No flags are set during the operation.

After the next instruction is executed, the contents become:

```
R4 = 0x00000066
R5 = 0x0000000A
MR0 = 0x000000CC
MR1 = 0x00000014
MR2 = 0x00000040
MR3 = 0x00000060
MR4 = 0x00000000
```

No flags are set.

```
R0 = 0xFFF50009;;
R1 = 0x00060004;;
R2 = 0x00100007;;
R3 = 0x001AFFF6;;
```

Multiplier Instructions

```
R5:4 = MR3:2, MR3:2 += R0 ** R1 (ICJ);;
```

```
R5:4 = MR3:2, MR3:2 += R0 ** R1 (IJ);;
```

If the previous contents of the MR register were:

```
MR0 = 0x00800050
```

```
MR1 = 0x01000200
```

```
MR2 = 0x00000040
```

```
MR3 = 0x00000060
```

```
MR4 = 0x00000001
```

After execution of the first register transfer and complex multiply-accumulate instruction, the values of all modified registers are:

```
R4 = 0x00000040
```

```
R5 = 0x00000060
```

```
MR0 = 0x00800050
```

```
MR1 = 0x01000200
```

```
MR2 = 0xFFFFFFFFE2
```

```
MR3 = 0xFFFFFFFF9E
```

```
MR4 = 0xFFFFF0001
```

No flags are set during the operation.

After the next instruction is executed, the contents become:

```
R4 = 0xFFFFFFFFE2
```

```
R5 = 0xFFFFFFFF9E
```

```
MR0 = 0x00800050
```

```
MR1 = 0x01000200
```

```
MR2 = 0xFFFFFFF0C4
```

```
MR3 = 0xFFFFFFF03C
```

```
MR4 = 0xFFFFF0001
```

No flags are set.

Multiply (Floating-Point, Normal/Extended Word)

Syntax

```
{X|Y|XY}FRs = Rm * Rn {(T)} ; /* Normal (32-bit) Word */
```

```
{X|Y|XY}FRsd = Rmd * Rnd {(T)} ; /* Extended (40-bit) Word */
```

Function

This is a 32/40-bit floating-point multiplication of the value in register *Rm* with the value in *Rn*. The result is placed in register *Rs* and rounded unless the *T* option is specified. Single-precision multiplication is 32-bit IEEE floating-point. Dual registers in the instruction denote 40-bit extended-precision floating-point multiplication.

Status Flags

MZ	Set if floating-point result is \pm zero
MN	Set if result is negative
MV	Set if the unbiased exponent of the result is greater than 127
MOS	Set if the unbiased exponent of the result is greater than 127; otherwise unaffected
MU	Set if the unbiased exponent of the result is less than -126
MUS	Set if the unbiased exponent of the result is less than -126 ; otherwise unaffected
MI	Set if either input is a NAN, or if the inputs are \pm infinity and \pm zero
MIS	Set if either input is a NAN, or if the inputs are \pm infinity and \pm zero; otherwise unaffected

Multiplier Instructions

Example

```
R0 = 0x3AF5EC81 ;; /* 0.00187625 */  
R1 = 0x3AF58CDF ;; /* 0.0018734 */  
FR2 = R0 * R1 ;;
```

After executing the floating-point multiply, the result stored in R2 is 0x366BE2AB (3.51497E-006).

Multiplier Result Register

Syntax

```

{X|Y|XY}MRa = Rmd ;
{X|Y|XY}MR4 = Rm ;
{X|Y|XY}{S}Rsd = MRa {{(U){S}}};
{X|Y|XY}Rsq = MR3:0 {{(U){S}}};
{X|Y|XY}Rs = MR4 ;
/* where MRa is either MR1:0 or MR3:2 */

```

Function

These instructions transfer the value of the source (to right of =) register to the destination (to left of =) register.

Because the source MR register can hold a saturated result that is larger than the destination *Rs*, the DSP has a protocol for handling the data size mismatch. When the option to saturate is specified, and if the result is too large for the *Rs* register, saturation is according to M4.

- *Rsd* denotes the transfer of one 80-bit result into a 64-bit register
- *Rsq* denotes the transfer of four 40-bit results into four 32-bit registers
- *SRsd* denotes the transfer of four 20-bit results into four 16-bit shorts

See the illustrations in [Figure 8-52](#), [Figure 8-53](#), and [Figure 8-54](#).

Rsd = MR1:0; see [Figure 8-52](#)

SRsd = MR1:0; see [Figure 8-53](#)

Rsq = MR3:0; see [Figure 8-54](#)

Multiplier Instructions

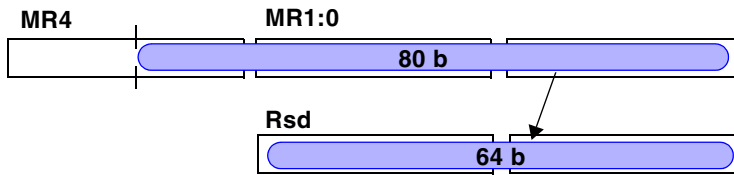


Figure 8-52. Move One 80-bit Result Into One 64-bit Register

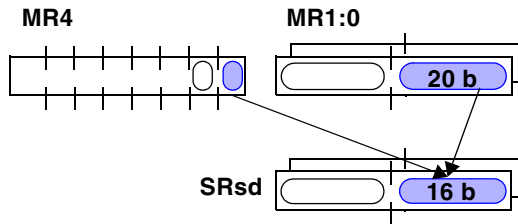


Figure 8-53. Move Four 20-bit Results Into Four 16-bit Shorts

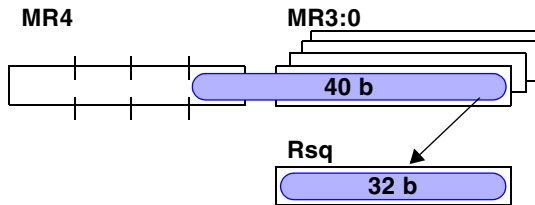


Figure 8-54. Move Four 40-bit Results Into Four 32-bit Registers

Status Flags

MZ	For $R_S=MR$, set if result = zero; for $MR=R_S$, cleared
MN	For $R_S=MR$, set if result is negative; for $MR=R_S$, cleared
MV	For $R_S=MR$, set if MR and MR4 is extended overflow; for $MR=R_S$, cleared

MOS For $R_S=MR$, set if MR and MR4 is extended overflow—otherwise unaffected; for $MR=R_S$, unchanged

MU (MUS) Cleared (unchanged)

Options

(S) Saturate

(U) Unsigned

See “Multiplier Instruction Options” on page 4-8 for more details about available options.

Example

Current contents of MR register are:

```
MR0 = 0x00000000
MR1 = 0x11111111
MR2 = 0x22222222
MR3 = 0xFFFFFFFF
MR4 = 0x0000FF00
```

After executing the following command:

```
R1:0 = MR3:2 (S);;
```

The contents of R1:0 are:

```
R0 = 0x22222222
R1 = 0x0FFFFFF5
```

The MV flag and MOS flag are both set.

Multiplier Instructions

If the following instruction was executed:

```
R3:2 = MR3:2;;
```

The contents of R3:2 would be:

```
R2 = 0x22222222
```

```
R3 = 0x0FFFFFF5
```

The MN, MV, and MOS flags would be set.

After executing the following instruction:

```
R1:0 = MR1:0 (S);;
```

The contents of R1:0 are:

```
R0 = 0x00000000
```

```
R1 = 0x80000000
```

The MV, MN, and MOS flag are all set.

If the following instruction was executed:

```
R3:2 = MR1:0 (U);;
```

The contents of R3:2 would be:

```
R2 = 0x00000000
```

```
R3 = 0x11111111
```

The MV and MOS flags would be set.

After executing the following instruction:

```
SR1:0 = MR3:2 (S);;
```

The contents of R1:0 are:

```
R0 = 0x22222222
```

```
R1 = 0x0FFF7FFF
```

The MV and MOS flags are set.

If the following instruction was executed:

```
SR3:2 = MR3:2 (U);;
```

The contents of R3:2 would be :

```
R2 = 0x22222222
```

```
R3 = 0x0FFFFFF5
```

The MV and MOS flags would be set.

After executing the following instruction:

```
SR1:0 = MR1:0 (S);;
```

The contents of R1:0 are:

```
R0 = 0x00000000
```

```
R1 = 0x80008000
```

The MV, MN, MZ, and MOS flags are all set.

If the following instruction was executed:

```
SR3:2 = MR1:0 (U);;
```

The contents of R3:2 would be:

```
R2 = 0x00000000
```

```
R3 = 0x11111111
```

The MZ, MV, and MOS flags are set.

Multiplier Instructions

After executing the following instruction:

```
R3:0 = MR3:0 (S);;
```

The contents of R1:0 are:

```
R0 = 0x00000000
```

```
R1 = 0x80000000
```

```
R2 = 0x22222222
```

```
R3 = 0x0FFFFFF5
```

The MV, MN, MZ, and MOS flags are all set.

If the following instruction was executed:

```
R3:0 = MR3:0 (U);;
```

The contents of R3:2 would be:

```
R0 = 0x00000000
```

```
R1 = 0x11111111
```

```
R2 = 0x22222222
```

```
R3 = 0x0FFFFFF5
```

The MV, MN, MZ, and MOS flags are all set.

```
R0 = 0x11111111 ;;
```

```
R1 = 0x22222222 ;;
```

```
R2 = 0x33333333 ;;
```

```
R3 = 0x44444444 ;;
```

```
R5 = 0x55555555 ;;
```

```
MR3:2 = R1:0 ;;
```

```
MR1:0 = R3:2 ;;
```

```
MR4 = R5 ;;
```

All set multiplier flags (including sticky) are cleared.

Compact Multiplier Result

Syntax

```
{X|Y|XY}Rs = COMPACT MRa {{(U){I}{S}}};
{X|Y|XY}SRsd = COMPACT MR3:0 {{(U){I}{S}{C}}};
/* where MRa is either MR1:0 or MR3:2 */
```

Function

The `COMPACT MRa` instruction compresses one 80-bit result in MR and MR4 into one 32-bit output. The output is always truncated. The compact `MRa` operation appears in [Figure 8-55](#).

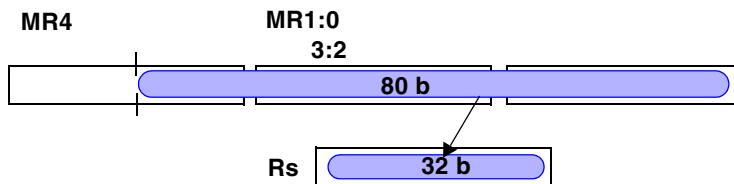


Figure 8-55. Compact MRa

The `COMPACT MR3:0` instruction compresses four 40-bit results in MR3:0 and MR4 into four 16-bit outputs. The result is always truncated. The compact `MR3:0` operation appears in [Figure 8-56](#).

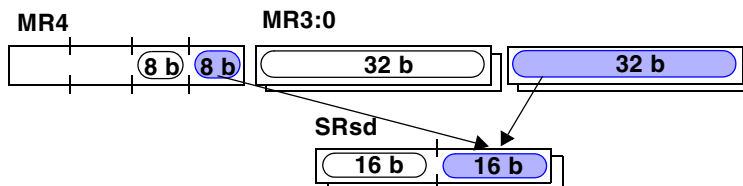


Figure 8-56. Compact MR3:0

Multiplier Instructions

Status Flags

MZ	Set if all bits in result are zero
MN	Set if result is negative
MU (MUS)	Cleared (unchanged)
MV (MOS)	Set according to the data format (see following conditions; (MOS unchanged if MV is cleared)

For word result, MV set according to:

- Signed fractional – Upper 17 bits of M are not all zeros or all ones
- Signed integer – Upper 49 bits of M are not all zeros or all ones
- Unsigned fractional – Upper 16 bits of M are not all zeros
- Unsigned integer – Upper 48 bits of M are not all zeros

Options

(I)	Integer
(S)	Saturate
(C)	Clear MR
(U)	Unsigned

See [“Multiplier Instruction Options” on page 4-8](#) for more details about available options.

Example

Current contents of MR register are:

```
MR0 = 0x00000000  
MR1 = 0x11111111
```


MR2 = 0x22222222

MR3 = 0xFFFFFFFF5

MR4 = 0x0000FF00

After executing:

R0 = COMPACT MR3:2 (UI) ;;

R0 = 0x22222222

The MV and MOS flags are set.

After executing:

R0 = COMPACT MR3:2 (U) ;;

R0 = 0xFFFFFFFF5

The MN and MOS flags are set.

Current contents of MR register are:

MR0 = 0x00000000

MR1 = 0x11111111

MR2 = 0x22222222

MR3 = 0xFFFFFFFF5

MR4 = 0x0000FF00

After executing:

R0 = COMPACT MR3:2 (UI) ;;

R0 = 0xFFFFFFFF

The MN, MV, and MOS flags are set.

After executing:

R0 = COMPACT MR3:2 (U) ;;

R0 = 0x80000000

The MN, MV, and MOS flags are set.

Multiplier Instructions

Current contents of MR register are:

MR0 = 0x00000000

MR1 = 0x11111111

MR2 = 0x22222222

MR3 = 0xFFFFFFFF

MR4 = 0x0000FF00

After executing:

SR1:0 = COMPACT MR3:0 (UI) ;;

R0 = 0x11110000

R1 = 0xFFFF52222

The MZ, MN, MV, and MOS flags are set.

After executing:

R1:0 = COMPACT MR3:0 (S) ;;

R0 = 0x80000000

R1 = 0x7FFF2222

The MZ, MN, MV, and MOS flags are set.

Shifter Instructions

The shifter performs *bit wise operations* (arithmetic and logical shifts) and performs *bit field operations* (field extraction and deposition) for the processor. The shifter also executes *data conversion operations* such as fixed-/floating-point format conversions. For a description of ALU operations, status flags, conditions, and examples, see [“Shifter” on page 5-1](#).

The conventions used in these reference pages for representing register names, optional items, and choices are covered in detail in [“Register File Registers” on page 2-5](#). Briefly, these conventions are:

- { } – the curly braces enclose options; these braces are not part of the instruction syntax.
- | – the vertical bars separate choices; these bars are not part of the instruction syntax.
- *Rmd* – the register names in italic represent user selectable single (*Rs*, *Rm*, *Rn*), double (*Rsd*, *Rmd*, *Rnd*) or quad (*Rsq*, *Rmq*, *Rnq*) register names.



Each instruction presented on these reference pages occupies one instruction slot in an instruction line. For more information about instruction lines and instruction combination constraints, see [“Instruction Line Syntax and Structure” on page 1-20](#) and [“Instruction Parallelism Rules” on page 1-24](#).

Shifter Instructions

Arithmetic/Logical Shift

Syntax

$$\{X|Y|XY\}\{B|S\}Rs = \text{LSHIFT|ASHIFT } Rm \text{ BY } Rn | \langle Imm \rangle ;^{1,2}$$
$$\{X|Y|XY\}\{B|S|L\}Rsd = \text{LSHIFT|ASHIFT } Rmd \text{ BY } Rn | \langle Imm \rangle ;$$

Function

These instructions arithmetically shift (extends sign on right shifts) or logically shift (no sign extension) the operand in register Rm by the value in register Rn or shifts it by the instruction's immediate value. A positive value of Rn denotes a shift to the left and a negative value of Rn denotes a shift to the right. The shifted result is placed in the result register Rs . The L, S, and B prefixes denote the operand type and d denotes operand size—see “[Instruction Line Syntax and Structure](#)” on page 1-20.

All shift values are two's-complement numbers. Positive values result in a left shift, and negative values result in a right shift. Shift magnitudes for register file-based operations are computed by using the right-most B bits of Rn , and masking the remaining bits in Rn , where $B = 8$ for long words, 7 for normals, 6 for shorts, and 5 for bytes—thereby achieving full-scale right and left shifts. Shift magnitudes for immediate-based shifts only require $B = 7$ for long words, 6 for normals, 5 for shorts, and 4 for bytes—see “[Shifter](#)” on page 5-1.

Status Flags

SZ	Set if shifter result is zero
SN	Equals the MSB of the result; for LSHIFT, cleared for all right shifts by more than zero

¹ The Rn data size (bits) for the shift magnitude varies with the output operand: Byte: 5, Short: 6, Normal: 7, Long: 8.

² The size in bits of the Imm data varies with the output operand: Byte: 4, Short: 5, Normal: 6, Long: 7.

Examples

R5 = lshift R3 by -4;

If R3 = 0x140056A3

then R5 = 0x0140056A

R5 = lshift R3 by 4;

If R3 = 0x140056A3

then R5 = 0x40056A30

R5:4=lshift R3:2 by -4;

If R3=0x140056A3 *and* R2=0x87654321

then R5=0x0140056A *and* R4=0x08765432;

R5:4=lshift R3:2 by 4;

If R3=0x140056A3 *and* R2=0x87654321

then R5=0x40056A0 *and* R4=0x76543210;

SR5=lshift R3 by -4;

If R3=0x140056A3

then R5=0x01400056A

SR5=lshift R3 by 4;

If R3=0x140056A3

then R5=0x40006A30

BR5=lshift R3 by -4;

If R3=0x140056A3

then R5=0x0100050A

BR5=lshift R3 by 4;

If R3=0x140056A3

then R5=0x40006030

R5 = ashift R3 by -4;

If R3 = 0x840056A3

then R5 = 0xF840056A

Shifter Instructions

R5 = ashift R3 by -4;

If R3 = 0x140056A3

then R5 = 0x0140056A

R5:4=ashift R3:2 by -4;

If R3=0x840056A3 *and* R2=0x87654321

then R5=0xF840056A *and* R4=0xF8765432;

R5:4=ashift R3:2 by 4;

If R3=0x840056A3 *and* R2=0x87654321

then R5=0x40056A0 *and* R4=0x76543210;

SR5=ashift R3 by -4;

If R3=0x840056A3

then R5=0xF8400056A

SR5=ashift R3 by 4;

If R3=0x840056A3

then R5=0x40006A30

BR5=ashift R3 by -4;

If R3=0x840056A3

then R5=0xF80005FA

BR5=ashift R3 by 4;

If R3=0x840056A3

then R5=0x40006030

Rotate

Syntax

```
{X|Y|XY}Rs = ROT Rm BY Rn|<Imm6> ;
{X|Y|XY}{L}Rsd = ROT Rmd BY Rnd|<Imm> ;
```

Function

This instruction rotates the operand in register Rm by a value determined by the operand in register Rn or by the bit immediate value in the instruction. The rotated result is placed in Rs . The L prefix denotes long operand type and d denotes operand size—see “[Instruction Line Syntax and Structure](#)” on page 1-20.

Rotate values are two’s-complement numbers. Positive values result in a left rotate, negative values in a right rotate. Rotate magnitudes for register file-based operations are computed by using the right-most B bits of Rn , and masking the remaining bits in Rn , where $B = 8$ for long words, and 7 for normals—thereby achieving full-scale right and left rotates. Rotate magnitudes for immediate-based rotates only require $B = 7$ for long words and 6 for normals—see “[Shifter](#)” on page 5-1.

Status Flags

SZ	Set if result is zero
SN	Equals the MSB of the result

Shifter Instructions

Examples

R5 = rot R3 by -4;

If R3 = 0x140056A3

then R5 = 0x3140056A

R5 = rot R3 by 4;

If R3 = 0x140056A3

then R5 = 0x40056A31

R5:4=rot R3:2 by -4;

If R3=0x140056A3 *and* R2=0x87654321

then R5=0x3140056A *and* R4=0x18765432;

R5:4=rot R3:2 by 4;

If R3=0x140056A3 *and* R2=0x87654321

then R5=0x40056A31 *and* R4=0x76543218;

LR5:4=rot R3:2 by -4;

If R3=0x140056A3 *and* R2=0x87654321

then R5=0x1140056A *and* R4=0x38765432;

LR5:4=rot R3:2 by 4;

If R3=0x140056A3 *and* R2=0x87654321

then R5=0x40056A38 *and* R4=0x76543211;

Field Extract

Syntax

```
{X|Y|XY}Rs = FEXT Rm BY Rn|Rnd {(SE)} ;
{X|Y|XY}LRsd = FEXT Rmd BY Rn|Rnd {(SE)} ;
```

Function

This instruction extracts a field from register *Rm* into register *Rs* that is specified by the control information in register *Rn*.

There are two versions of this instruction. One takes the control information (field’s target position and length) from a register pair—*Rnd*. The other takes the control information from a single register—*Rn*.

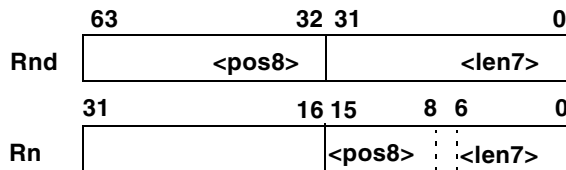


Figure 8-57. Len7 and Pos8 Fields for Dual- and Single-Word Registers

The length field is right-justified in *Rn* and its length is 7 bits, allowing lengths of 64 bits and 0 bits inclusive. The position field is 8 bits, allowing off-scale left extracts.

If the SE option is set, the bits to the left of the extracted field in the destination register *Rs* are set equal to the MSB of the extracted field; otherwise the bits to the left of the extracted field in *Rs* are set to zero.

The L prefix denotes long operand type and d denotes operand size—see [“Instruction Line Syntax and Structure” on page 1-20](#).

Field Deposit

Syntax

```
{X|Y|XY}Rs += FDEP Rm BY Rn|Rnd {(SE|ZF)} ;
{X|Y|XY}LRsd += FDEP Rmd BY Rn|Rnd {(SE|ZF)} ;
```

Function

This instruction deposits a right-justified field from register *Rm* into register *Rs*, where the position and length in the destination register *Rs* are determined by the control information in register *Rn*.

If the SE option is set, the bits to the left of the deposited field in the destination register *Rs* are set equal to the MSB of the deposited field; otherwise the original bits in *Rs* are unaffected. If the ZF option is set, the bits to the left of the deposited field in *Rs* are set to zero; otherwise the original bits in *Rs* are unaffected.

There are two versions of this instruction. One takes the control information (field’s target position and length) from a register pair—*Rnd*. The other takes the control information from a single register—*Rn*.

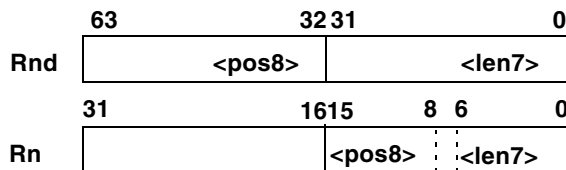


Figure 8-59. Len7 and Pos8 Fields for Dual- and Single-Word Registers

The length field is right-justified in *Rn* and its length is 7 bits, allowing lengths of 64 bits and 0 bits inclusive. The position field is 8 bits, allowing off-scale left deposits.

Shifter Instructions

The *L* prefix denotes long operand type and *d* denotes operand size—see “Instruction Line Syntax and Structure” on page 1-20.

Status Flags

- SZ Set if result is zero
- SN Equals the MSB of the result

Examples

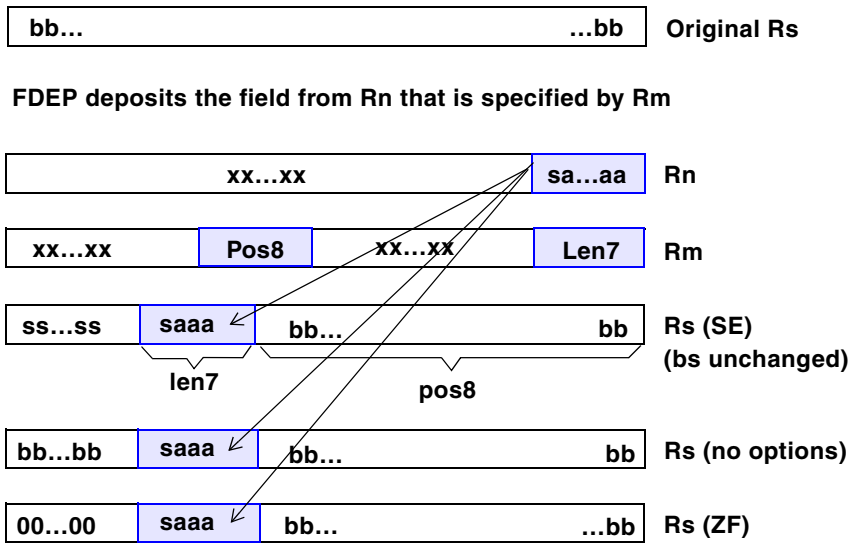


Figure 8-60. FDEP Instruction Performed on Single-Word Registers

Field/Bit Mask

Syntax

```
{X|Y|XY}Rs += MASK Rm BY Rn ;
{X|Y|XY}LRsd += MASK Rmd BY Rnd ;
```

Function

This instruction substitutes the field in register R_s by a 32-bit (or 64-bit, for long words) field in register R_m , which is determined by the set bits in register R_n . R_s is a read-modify-write register. Logically, the operation is:

$$R_s = (R_s \text{ AND } \text{NOT}(R_n)) \text{ OR } (R_m \text{ AND } R_n)$$

AND, OR, and NOT are bit-wise logical operators.

The L prefix denotes long operand type and d denotes operand size—see [“Instruction Line Syntax and Structure” on page 1-20](#).

Status Flags

SZ	Set if result is zero
SN	Equals the MSB of the result

Shifter Instructions

Example

```
Rs += MASK Rm BY Rn;;
```

The '1' mask bits determine which bits in *Rn* are to be copied—the bits need not be contiguous

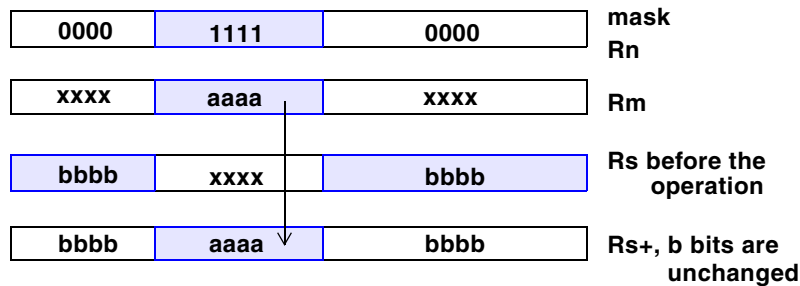


Figure 8-61. Field/Bit Mask Data Flow

```
XR0 = IMASKH ;; /* load upper half of IMASK into R0 */  
XR1 = 0xEFFF ;; /* load mask value (GIE bit cleared) into R1 */  
XR2 += MASK XR0 BY XR1 ;;  
/* use MASK to clear bit, but retain other values */  
IMASKH = XR2 ;;  
/* load IMASKH with data globally disabling interrupts */
```

Get Bits

Syntax

```
{X|Y|XY}Rsd = GETBITS Rmq BY Rnd {(SE)} ;
```

Function

This instruction extracts a bit field from a contiguous bit stream held in the quad register *Rmq* and stores the extracted field in *Rsd*, according to the control information in *Rnd*. This instruction (in conjunction with shifter instructions `PUTBITS` and `BFOTMP`, and ALU instruction `BFOINC`) is used to implement a bit FIFO.

If the `SE` option is set, the bits to the left of the deposited field in the destination register pair *Rsd* are set equal to the MSB of the deposited field; otherwise those bits in *Rsd* are set to zero.

The control information (current bit FIFO pointer, or BFP, and length of extracted field) form a register pair—*Rnd*. Instruction `GETBITS` uses the BFP and length information (stored in *Rnd*) to perform the bit field extraction, but does not update the BFP. Update to the BFP should be performed by the ALU with instruction `BFOINC`. See example below for a suggested code sequence.

The `q` suffix denotes operand size—see [“Instruction Line Syntax and Structure” on page 1-20](#).

Status Flags

SZ	Set if bits in extracted field are all zero
SN	MSB of extracted field

Shifter Instructions

Example

```
R1:0 = GETBITS R7:4 BY R17:16;;  
R17 = BFOINC R17:16;;
```

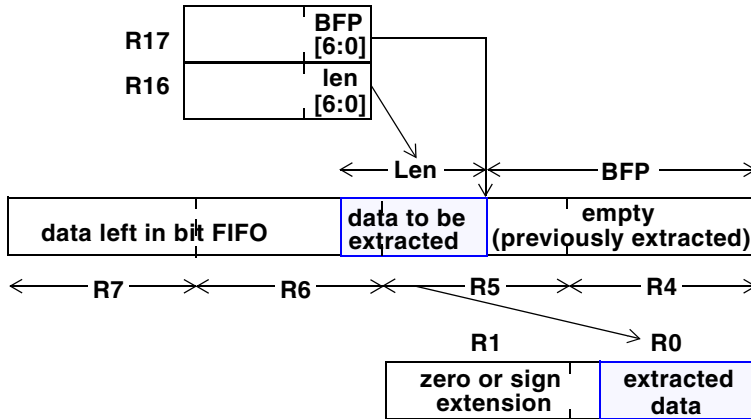


Figure 8-62. Get Bits Data Flow

Put Bits

Syntax

```
{X|Y|XY}Rsd += PUTBITS Rnd BY Rnd ;
```

Function

This instruction deposits the 64 bits in *Rnd* into a contiguous bit stream held in the quad register composed of *BFOTMP* in the top and *Rsd* in the bottom. The data is inserted, beginning from the bit pointed to by *BFP* field in *Rnd*. The *LEN* field of *Rnd* is ignored. This instruction (in conjunction with shifter instructions *GETBITS* and *BFOTMP*, and ALU instruction *BFOINC*) is used to implement a bit *FIFO*.

The control information (current bit *FIFO* pointer, or *BFP*, and length of extracted field) form a register pair—*Rnd*. Instruction *PUTBITS* uses only the pointer field *BFP* (stored in *Rnd*) to perform the bit insertion; it does not use the length field in *Rnd*. *PUTBITS* also does not update the *BFP*. Update to the *BFP* should be performed by the ALU with instruction *BFOINC*.

Whenever a bit insertion is performed, the entire contents of register pair *Rnd* is placed in the register quad formed by *BFOTMP* and *Rsd*. Generally, the bit field to be inserted into the bit *FIFO* is less than 64 bits long and, in this case the field of interest, should be placed right-justified in *Rnd*.

Note that the remaining bits to the left of the field of interest are irrelevant. When the *BFP* overflows past bit 64 (an event performed by the ALU and recorded in flag *AN*), the contents of *Rsd* should be moved out, and the *BFOTMP* register should be moved into *Rsd*. See [Figure 8-63](#) for a suggested code sequence used to implement a bit *FIFO* insertion.

The *d* suffix denotes operand size—see [“Instruction Line Syntax and Structure”](#) on page 1-20.

Shifter Instructions

Status Flags

SZ Cleared

SN Cleared

Example

```
R1:0 += PUTBITS R5:4 BY R7:6;;
```

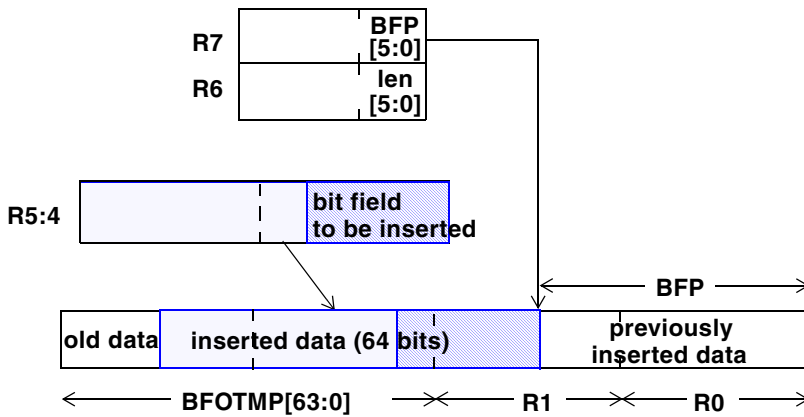


Figure 8-63. Data Flow for Instruction:
`R1:0 += PUTBITS R5:4 BY R7:6;;`

Bit Test

Syntax

```
{X|Y|XY}BITEST Rm BY Rn<Imm5> ;
{X|Y|XY}BITEST Rmd BY Rn<Imm6> ;
```

Function

This instruction tests bit #*n* in register *Rm*, as indicated by the operand in *Rn* or by the bit immediate value in the instruction. The *SZ* flag is set if the bit is a zero and cleared if the bit is one. *SZ* is also set when the tested bit position is greater than 31 and 63 for normal and long operands, respectively. The position of the bit is the 6- or 5-bit value in register *Rn* (for long- or normal-words respectively) or the bit immediate value in the instruction. For instance, in a normal word operation the value 0x00000000 in *Rn* tests the LSB of *Rm*, and the value 0x0000001F tests the MSB of *Rm*. Different from the normal concept, the *d* suffix denotes a 64-bit (long) single operand.

Status Flags

<i>SZ</i>	Cleared if tested bit is 1 Set if tested bit is zero, or if bit position is greater than 31
<i>SN</i>	Equals the MSB of the input

Shifter Instructions

Bit Clear/Set/Toggle

Syntax

```
{X|Y|XY}Rs = BCLR|BSET|BTGL Rm BY Rn|<Imm5> ;  
{X|Y|XY}Rsd = BCLR|BSET|BTGL Rmd BY Rn|<Imm6> ;
```

Function

These instructions clear, set, or toggle bit #n in register *Rm* as indicated by the operand in *Rn* or by the bit immediate value in the instruction. The result is placed in register *Rs*. The position of the bit is the 6- or 5-bit value in register *Rn* (for long- or normal-words respectively), or the bit immediate value in the instruction. For example, in a normal word BCLR operation the value 0x00000000 in *Rn* clears the LSB of *Rm* and the value 0x0000001F clears the MSB of *Rm*. The *d* suffix denotes operand size—see “[Instruction Line Syntax and Structure](#)” on page 1-20.

Status Flags

SZ	Set if result is zero
SN	Equals the MSB of the result

Example

R5 = bclr R3 by 5;

If R3=0x140056A3

then R5=0x14005683

R5:4=bclr R3:2 by 5;

If R3=0x140056A3 *and* R2=0x87654321

then R5=0x140056A3 *and* R4=0x87654301;

R5 = bset R3 by 6;

If R3=0x140056A3

then R5=0x140056E3

R5:4=bset R3:2 by 6;

If R3=0x140056A3 *and* R2=0x87654321

then R5=0x140056A3 *and* R4=0x87654361

R5 = btgl R3 by 6;

If R3=0x140056A3

then R5=0x140056E3

R5:4=btgl R3:2 by 6;

If R3=0x140056A3 *and* R2=0x87654321

then R5=0x140056A3 *and* R4=0x87654361;

Shifter Instructions

Extract Leading Zeros

Syntax

$\{X|Y|XY\}Rs = LD0|LD1 Rm|Rmd ;$

Function

This instruction extracts the number of leading zeros or leading ones from the operand in register *Rm*. The extracted number is placed in the six LSBs of *Rs*.

Status Flags

SZ Set if the most significant bit of *Rm* is one with option of leading zeros (LD0); also set if the MSB of *Rm* is zero with option of leading ones (LD1)

SN Cleared

Example

$R5=1d0 R3;$

If $R3=0x140056A3$

then $R5=0x3;$

$R5=1d1 R3:2;$

If $R3=0xE40056A3$ *and* $R2=0x87654321$

then $R5=3;$

Extract Exponent

Syntax

$\{X|Y|XY\}Rs = EXP Rm|Rmd ;$

Function

This instruction extracts the exponent of the operand in register Rm . The exponent is calculated as the two's-complement of the leading sign bits in $Rm - 1$.

Status Flags

SZ	Set if extracted exponent is zero
SN	Set if fixed-point operand in Rm is negative (MSB is a one)



SN is used to return the sign of the input. This is required for non-IEEE floating-point and for double precision float.

Examples

$R5 = EXP R3 ;$

If $R3 = b\#111010\dots$
then $R5 = -2$

If $R3 = 0x0000101$
then $R5 = -3$

$R5 = \text{exp } R3 : 2 ;$

If $R3 = 0x140056A3$ *and* $R2 = 0x87654321$
then $R5 = -2$ (in hexadecimal, $R5 = 0xFFFFFFFFE$)

Shifter Instructions

XSTAT/YSTAT Register

Syntax

```
{X|Y}STAT = Rm ;  
{X|Y}STATL = Rm ;  
{X|Y}Rs = {X|Y}STAT ;
```

Function

These instructions load the operand in register Rm into the X/Y status register or store the operand in the X/Y status register into Rs . There are two $X/YSTAT$ load instructions. One loads the entire register Rm into $X/YSTAT$, while the other only loads the 15 LSBs of Rm into $X/YSTAT$. The latter form of this instruction is used when restoring only the dynamic status flags, without affecting the sticky flags and mode bits.

Status Flags

For $X/YSTAT=Rm$, the status flags register ($X/YSTAT$) receives the value of Rm and updates the status flags according to the Rm value.

For $Rm=X/YSTAT$, the status flags are unaffected.

Example

```
XSTATL = R6;;
```

```
If R6 = 0x910  
then XSTAT = 0x910
```

```
If R6 = 0x90010  
then XSTAT = 0x10
```

```
XR6 = STAT;;
```

```
If XSTAT = 0x10  
then R6 = 0x10
```


Block Floating-Point

Syntax

```
{X|Y|XY}BKFPT Rmd, Rnd ;
```

Function

This instruction is used for determining the scaling factor used in 16-bit block floating-point. The two input registers, *Rmd* and *Rnd*, hold eight short words and, after execution, the two block floating-point flags BF1:0 in X/YSTAT are set according to the largest of (a) the number of redundant sign bits in the eight input short words and (b) the previous state of BF1:0.

The BKFPT instruction maps each one of the eight input short words into two bits, which are equal to three minus the number of sign bits. The value depends on the three MSBs of the input short:

- If the three MSBs of operand are 000 or 111 – value is 00
- If the three MSBs of operand are 001 or 110 – value is 01
- If the two MSBs of operand are 01 or 10 – value is 10

After computing these two bits for each short word, the final output is determined by finding the maximum among the set eight values plus the current value of BF1:0. Finally, BF1:0 is updated with the result. This instruction records up to two redundant sign bits.

For example, if BF1:0 = b#00, and if the three MSBs of all eight input numbers are either all 0s or all 1s, then BF1:0 = b#00.

If at least one of the eight input numbers has the three MSBs as b#001 or b#110, then BF1:0 = b#01.

If at least one of the eight input numbers has the three MSBs as b#011 or b#100, then BF1:0 = b#10.

Shifter Instructions

Status Flags

Updates flags BF0 and BF1. Status flags are unaffected.

Examples

Assume that initially BF1:0 = b#00:

XR0 = b#0010 0000 ... 0000

XR1 = 0

Execution of XBKFPT R1:0, R1:0;; *causes the flags to be set to* BF1:0 = b#01

XR0=0

XR1=0

With these inputs, second execution of XBKFPT R1:0, R1:0;;
results in BF1:0 = b#01

BFOTMP Register

Syntax

```
{X|Y|XY}Rsd = BFOTMP ;
{X|Y|XY}BFOTMP = Rmd ;
```

Function

These instructions load the value of the BFOTMP register into the *Rsd* register or loads the operand in register *Rmd* into the BFOTMP register. BFOTMP is a register internal to the shifter. This function is used to temporarily hold the overflow bits after a PUTBITS instruction.

Status Flags

The status flags are unaffected by this operation

Example

```
R9:8 = BFOTMP;;
```

```
If BFOTMP = 0x17000016
then R9:8 = 0x17000016
```

```
BFOTMP = R9:8;;
```

```
If R9:8 = 0x40000005
then BFOTMP = 0x40000005
```

IALU (Integer) Instructions

The TigerSHARC processor's two independent IALUs are referred to as the J-IALU and K-IALU. The IALUs support regular ALU operations and data addressing operations. The *arithmetic, logical, and function ALU operations* include:

- Add and subtract, with and without carry/borrow
- Arithmetic right shift, logical right shift, and rotation
- Logical operations: AND, AND NOT, NOT, OR, and XOR
- Functions: absolute value, min, max, compare

For a description of IALU operations, status flags, conditions, and examples, see [“IALU” on page 6-1](#). The IALUs also provide data addressing support instructions. For instruction reference pages on IALU data addressing instructions, see [“IALU \(Load/Store/Transfer\) Instructions” on page 8-218](#).

The conventions used in these reference pages for representing register names, optional items, and choices are covered in detail in [“IALU” on page 6-1](#). Briefly, these conventions are:

- { } – the curly braces enclose options; these braces are not part of the instruction syntax.
- | – the vertical bars separate choices; these bars are not part of the instruction syntax.

- *Jm* or *Km* – the letter J or K in register names in italic indicate selection of a register in the J-IALU or K-IALU.
- *Jm* – the register names in italic represent user selectable single (*Jm*, *Jn*, *Js*, *Rs*, *Ureg_s*), double (*Rsd*, *Rmd*, *Ureg_sd*, *Ureg_md*) or quad (*Rsq*, *Rmq*, *Ureg_sq*, *Ureg_mq*) register names.



Each instruction presented on these reference pages occupies one instruction slot in an instruction line. For more information about instruction lines and instruction combination constraints, see [“Instruction Line Syntax and Structure”](#) on page 1-20 and [“Instruction Parallelism Rules”](#) on page 1-24.

IALU (Integer) Instructions

Add/Subtract (Integer)

Syntax

$$J_s = J_m + | - J_n | \langle Imm8 \rangle | \langle Imm32 \rangle \{ (\{ CJMP | CB | BR \}) \} ;$$
$$JB0 | JB1 | JB2 | JB3 | JL0 | JL1 | JL2 | JL3 = J_m + | - J_n | \langle Imm8 \rangle | \langle Imm32 \rangle ;$$
$$K_s = K_m + | - K_n | \langle Imm8 \rangle | \langle Imm32 \rangle \{ (\{ CJMP | CB | BR \}) \} ;$$
$$KB0 | KB1 | KB2 | KB3 | KL0 | KL1 | KL2 | KL3 = K_m + | - K_n | \langle Imm8 \rangle | \langle Imm32 \rangle ;$$

Function

These instructions add or subtract the operands in registers J_m/K_m and J_n/K_n . The result is placed in register J_s/K_s . This operation operates on dual registers or single registers plus an immediate value. The Imm can be 8 or 32 bits (using immediate extension—see “[Immediate Extension Operations](#)” on page 6-36). J denotes J-IALU registers; K denotes K-IALU registers.

If the $CJMP$ option is used, the result is placed in the $CJMP$ register as well as in J_s/K_s . The CB option can only be used when J_m/K_m is 0, 1, 2 or 3 and causes the operation to be executed by the circular buffer. If the BR option is used, the bit reverse adder is used. Only one of the options above may be used.

The alternative format to the instruction $J_s = J_m + J_n$ is $JB_i/JL_i = J_m + J_n$. The instructions are identical except that the result is placed in the JB or JL circular buffer register files instead of J_s .

When options CB or BR are selected, there is no overflow.

Status Flags

JZ/KZ	Set if all bits in result are zero
JN/KN	Set to the most significant bit of result
JV/KV	Set if overflow; else cleared

JC/KC Set to the carry out of the operation;
 Cleared if CB or BR options are used

Options

CJMP Computed Jump – the CJMP register is loaded with
 the result

CB Circular Buffer – use circular buffer operation for
 the result; this only applies when $Jm/Km = J/K[3:0]$

BR Bit Reverse – user bit reversed address

Examples

```
J3 = J5 + J29;;
K2 = K2 + 0x25;;
K6 = K1 + K8 (CB);;
J1 = J5 + J10 (BR);;
JB0 = J31 + 0x5;;
J4 = J2 + J3 (CJMP);;
```

```
J3 = J5 - J29;;
K2 = K2 - 0x25;;
K6 = K1 - K8 (CB);;
J1 = J5 - J10 (BR);;
JB0 = J31 - 0x5;;
J4 = J2 - J3 (CJMP);;
```

IALU (Integer) Instructions

Add/Subtract With Carry/Borrow (Integer)

Syntax

$$Js = Jm + Jn | \langle Imm8 \rangle | \langle Imm32 \rangle + JC ;$$

$$Js = Jm - Jn | \langle Imm8 \rangle | \langle Imm32 \rangle + JC - 1 ;$$

$$Ks = Km + Kn | \langle Imm8 \rangle | \langle Imm32 \rangle + KC ;$$

$$Ks = Km - Kn | \langle Imm8 \rangle | \langle Imm32 \rangle + KC - 1 ;$$

Function

These instructions add with carry or subtract with borrow the operands in registers Jm/Km and Jn/Kn in J-IALU with the carry flag from the $J/KSTAT$ register in the J-/K-IALU. The result is placed in register Js/Ks . This operation operates on dual registers or single registers plus an immediate value. The Imm can be 8 or 32 bits (using immediate extension—see [“Immediate Extension Operations” on page 6-36](#)). J denotes J-IALU and $JSTAT$ registers; K denotes K-IALU and $KSTAT$ registers.

Status Flags

JZ/KZ	Set if all bits in result are zero
JN/KN	Set to the most significant bit of result
JV/KV	Set overflow bit
JC/KC	Set to carry out the add operation

Examples

J4 = J2 + J8 + JC;;

K8 = K1 + K4 + KC;;

K3 = K1 + 0x2 + KC;;

J4 = J2 - J8 + JC-1;;

K8 = K1 - K4 + KC-1;;

K3 = K1 - 0x2 + KC-1;;

IALU (Integer) Instructions

Average (Integer)

Syntax

$$Js = (Jm + | - Jn | \langle Imm8 \rangle | \langle Imm32 \rangle) / 2 ;$$

$$Ks = (Km + | - Kn | \langle Imm8 \rangle | \langle Imm32 \rangle) / 2 ;$$

Function

These instructions add or subtract the operands in registers Jm/Km and Jn/Kn , then divide the result by two. The result is placed in register Js/Ks . This operation operates on dual registers or single registers plus an immediate value. The Imm can be 8 or 32 bits (using immediate extension—see “Immediate Extension Operations” on page 6-36).

This instruction uses an arithmetic right shift for division. Therefore, rounding is toward infinity, not zero—if the result is negative, it will round down. For example, the result of $(-1 - 2)/2$ would be -2 and $(-1 + 0)/2$ would be -1 , while the result of $(1 + 2)/2$ would be 1 and $(1 + 0)/2$ would be 0 .

A J denotes J-IALU registers; K denotes K-IALU registers.

Status Flags

JZ/KZ	Set if all bits in result are zero
JN/KN	Set to the most significant bit of result
JV/KV	Cleared
JC/KC	Cleared

Examples

```
J4 = (J2 + J8) / 2;;  
K9 = (K2 + 0x2) / 2 ;;
```

```
J4 = (J2 - J8) / 2;;  
K9 = (K2 - 0x2) / 2 ;;
```

IALU (Integer) Instructions

Compare (Integer)

Syntax

COMP(*Jm*, *Jn* | <*Imm8*> | <*Imm32*>) { (U) } ;

COMP(*Km*, *Kn* | <*Imm8*> | <*Imm32*>) { (U) } ;

Function

This instruction compares the operand in register *Jm/Km* with the operand in register *Jn/Kn*. This instruction sets the JZ/KZ flag if the two operands are equal, and the JN/KN flag if the operand in register *Jm/Km* is smaller than the operand in *Jn/Kn*. A K denotes K-IALU registers as opposed to J-IALU registers.

This operation operates on dual registers or single registers plus an immediate value. The *Imm* can be 8 or 32 bits (using immediate extension—see “[Immediate Extension Operations](#)” on page 6-36). The unsigned option is only relevant for compare operations and indicates if the comparison is to be made on unsigned numbers (positive only) or two’s-complements (signed).

Status Flags

JZ/KZ	Set if input operands are equal
JN/KN	Set if <i>Jm/Km</i> is less than <i>Jn/Kn</i>
JV/KV	Cleared
JC/KC	Cleared

Options

()	Signed
(U)	Unsigned comparison

Examples

COMP (J4,J8) (U) ;;

If J4 = 0xFFFF FFFC *and* J8 = 0x0000 0003
then the status flags are set as follows:

JZ = 0

JN = 0

COMP (J4,J8) ;;

If J4 = 0xFFFF FFFC *and* J8 = 0x0000 0003
then the status flags are set as follows:

JZ = 0

JN = 1

COMP (J4, 0x0000 0003) (U);;

If J4 = FFFF FFFC

Then the status flags are set as follows:

JZ = 0

JN = 0

IALU (Integer) Instructions

Maximum/Minimum (Integer)

Syntax

$$Js = \text{MAX} | \text{MIN} (Jm, Jn | \langle Imm8 \rangle | \langle Imm32 \rangle) ;$$
$$Ks = \text{MAX} | \text{MIN} (Km, Kn | \langle Imm8 \rangle | \langle Imm32 \rangle) ;$$

Function

These instructions return maximum (larger of) or minimum (smaller of) the two operands in the registers Jm/Km and Jn/Kn . The result is placed in register Js/Ks . MAX operations are always signed. This operation operates on dual registers or single registers plus an immediate value. The Imm can be 8 or 32 bits (using immediate extension—see [“Immediate Extension Operations” on page 6-36](#)). J denotes J-IALU registers; K denotes K-IALU registers.

Status Flags

JZ/KZ	Set if all bits in result are zero
JN/KN	Set to the most significant bit of result
JV/KV	Cleared
JC/KC	Cleared

Example

```
K4 = MAX (K2,K9) ;;  
If K2 = 5 and K9 = 1  
then K4 = 5
```

```
K4 = MAX (K2,0x1);;  
If K2=5  
then K4=5
```

```
J2 = MIN(J7,J8) ;;  
If J7 = 4 and J8 = 3  
then J2 = 3
```

```
J2 = MIN(J7,0x3);;  
If J7=4  
then J2=3
```

IALU (Integer) Instructions

Absolute Value (Integer)

Syntax

$J_s = \text{ABS } J_m ;$

$K_s = \text{ABS } K_m ;$

Function

This instruction determines the absolute value of the operand in register J_m/K_m . The result is placed in register J_s/K_s . K denotes K-IALU registers as opposed to J-IALU registers.

The ABS of the most negative number (0x8000 0000) causes the maximum positive number (0x7FFF FFFF) to be returned and sets the overflow flag.

Status Flags

JZ/KZ	Set if all bits in result are zero
JN/KN	Set to the most significant bit of the <i>input</i>
JV/KV	Set when <i>input</i> is the most negative number
JC/KC	Cleared

Examples

```
J5 = ABS J4 ;;  
If J4 = 0x8000 0000  
then J5 = 0x7FFF FFFF and both JV and JN are set  
If J4 = 0x7FFF FFFF  
then J5 = 0x7FFF FFFF  
If J4 = 0xF000 0000  
then J5 = 0x1000 0000 and JN is set
```


Logical AND/AND NOT/OR/XOR/NOT (Integer)

Syntax

$$Js = Jm \text{ OR|AND|XOR|AND NOT } Jn | \langle Imm8 \rangle | \langle Imm32 \rangle ;$$

$$Js = \text{NOT } Jm ;$$

$$Ks = Km \text{ OR|AND|XOR|AND NOT } Kn | \langle Imm8 \rangle | \langle Imm32 \rangle ;$$

$$Ks = \text{NOT } Km ;$$

Function

These instructions logically AND, AND NOT, OR, or XOR the operands, bit by bit, in registers Jm/Km and Jn/Jn . The NOT instruction logically complements the operand in Jm/Km . The result is placed in register Js/Ks . This operation operates on dual registers or single registers plus an immediate value. The Imm can be 8 or 32 bits (using immediate extension—see [“Immediate Extension Operations” on page 6-36](#)). J denotes J-IALU registers; K denotes K-IALU registers.

Status Flags

JZ/KZ	Set if all bits in result are zero
JN/KN	Set to the most significant bit of result
JV/KV	Cleared
JC/KC	Cleared

IALU (Integer) Instructions

Example

J5 = J4 AND J8 ;;

If J4 = b#...1001 *and* J8 = b#...1100
then J5 = b#...1000

K5 = K4 AND 0xC ;;

If K4 = b#1001 (*in binary*)
then K5 = b#1000 (*in binary*)

J6 = J2 AND NOT J5 ;;

If J2 = b#...1001 *and* J5 = b#...1100
then J6 = b#...0001

J6 = J2 AND NOT 0xC ;;

If J2 = b#1001 (*in binary*)
then J6 = b#0001 (*in binary*)

J5 = J3 OR J4 ;;

If J3 = b#...1001 *and* J4 = b#...1100
then J5 = b#...1101

K5 = K3 OR 0xC ;;

If J3 = 1001 (*in binary*)
then J5 = 1101 (*in binary*)

J3 = J2 XOR J7 ;;

If J2 = b#...1001 *and* J7 = b#...1100
then J3 = b#...0101

J3 = J2 XOR 0xC ;;

If J3 = b#1001 (*in binary*)
then J5 = b#0101 (*in binary*)

J7 = NOT J6 ;;

If J6 = b#...0110
then J7 = b#...1001

Arithmetic Shift/Logical Shift (Integer)

Syntax

$J_s = \text{ASHIFTR}|\text{LSHIFTR } J_m ;$

$K_s = \text{ASHIFTR}|\text{LSHIFTR } K_m ;$

Function

These instructions perform an arithmetic shift (extends sign on right shift) or logically shift (no sign extension) the operand in register J_m/K_m to the right by one bit. The shifted result is placed in register J_s/K_s . The shift values are two's-complement numbers. J denotes J-IALU registers; K denotes K-IALU registers.

Status Flags

JZ/KZ	Set if all bits in result are zero
JN/KN	Set to the most significant bit of result
JV/KV	Cleared
JC/KC	Set to the least significant bit in the input

Examples

```

K4 = ASHIFTR K3 ;;
If K3 = 0x8600 0000
then K4 = 0xC300 0000 and JN is set
If K3 = 0x0860 0000
then K4 = 0x0430 0000 and JN is cleared
If K3 = 0xFFFF FFFF
then K4 = 0xFFFF FFFF and both JN and JC are set

K4 = LSHIFTR K3 ;;
If K3 = 0x0800 0000
then K4 = 0x0400 0000

```

IALU (Integer) Instructions

If K3 = 0x8600 0000
then K4 = 0x4300 0000
If K3 = 0xFFFF FFFF
then K4 = 0x7FFF FFFF *and* JC *is set*

Left Rotate/Right Rotate (Integer)

Syntax

$J_s = \text{ROTR}|\text{ROTL } J_m ;$

$K_s = \text{ROTR}|\text{ROTL } K_m ;$

Function

These instruction rotate the operand in register J_m/K_m to the left or right. The rotated result is placed in register J_s/K_s . J denotes J-IALU registers; K denotes K-IALU registers.

Status Flags

JZ/KZ	Set if all bits in result are zero
JN/KN	Set to the most significant bit of result
JV/KV	Cleared
JC/KC	Cleared

Example

$J_5 = \text{ROTR } J_3 ; ;$

If $J_3 = \text{b}\#1000\dots0101$

then $J_5 = \text{b}\#1100\dots010$

$J_5 = \text{ROTL } J_3 ; ;$

If $J_3 = \text{b}\#1000\dots0101$

then $J_5 = \text{b}\#000\dots01011$

IALU (Load/Store/Transfer) Instructions

The TigerSHARC processor's two independent IALUs are referred to as the J-IALU and K-IALU. The IALUs provide memory addresses when data is transferred between memory and registers. Dual IALUs enable simultaneous addresses for multiple operand reads or writes. The IALU's *data addressing and data movement operations* include:

- Direct and indirect memory addressing
- Circular buffer addressing
- Bit reverse addressing
- Universal register (*Ureg*) moves and loads
- Memory pointer generation

For a description of IALU operations, status flags, conditions, and examples, see [“IALU” on page 6-1](#). The IALUs also provide integer arithmetic support instructions. For instruction reference pages on IALU arithmetic instructions, see [“IALU \(Integer\) Instructions” on page 8-200](#).

The conventions used in these reference pages for representing register names, optional items, and choices are covered in detail in [“IALU” on page 6-1](#). Briefly, these conventions are:

- { } – the curly braces enclose options; these braces are not part of the instruction syntax.
- | – the vertical bars separate choices; these bars are not part of the instruction syntax.

- *Jm* or *Km* – the letter J or K in register names in italic indicate selection of a register in the J-IALU or K-IALU.
- *Jm* – the register names in italic represent user selectable single (*Jm*, *Jn*, *Js*, *Rs*, *Ureg_s*), double (*Rsd*, *Rmd*, *Ureg_sd*, *Ureg_md*) or quad (*Rsq*, *Rmq*, *Ureg_sq*, *Ureg_mq*) register names.



Each instruction presented on these reference pages occupies one instruction slot in an instruction line. For more information about instruction lines and instruction combination constraints, see [“Instruction Line Syntax and Structure”](#) on page 1-20 and [“Instruction Parallelism Rules”](#) on page 1-24.

IALU (Load/Store/Transfer) Instructions

Universal Register Load (Data Addressing)

Syntax

```
Ureg_s = [ Jm +|+= Jn |<Imm8> |<Imm32> ] ;
```

```
Ureg_sd = L [ Jm +|+= Jn |<Imm8> |<Imm32> ] ;
```

```
Ureg_sq = Q [ Jm +|+= Jn |<Imm8> |<Imm32> ] ;
```

```
Ureg_s = [ Km +|+= Kn |<Imm8> |<Imm32> ] ;
```

```
Ureg_sd = L [ Km +|+= Kn |<Imm8> |<Imm32> ] ;
```

```
Ureg_sq = Q [ Km +|+= Kn |<Imm8> |<Imm32> ] ;
```

```
/* Ureg suffix indicates: _s=single, _sd=double, _sq=quad */
```

Function

These instructions load the destination register (to left of =) with the contents of the source memory location (to right of =). These instructions support pre-modify without update ([+] operator) addressing and post-modify with update ([+=] operator) addressing for memory accesses. For a description of addressing and memory access types, see [“IALU Data Addressing and Transfer Operations” on page 6-13](#).

Status Flags

None affected

Options

None

Example

For examples, see [“IALU Examples” on page 6-37](#).

Universal Register Store (Data Addressing)

Syntax

```

    [ Jm + | += Jn | <Imm8> | <Imm32> ] = Ureg_s ;
L [ Jm + | += Jn | <Imm8> | <Imm32> ] = Ureg_sd ;
Q [ Jm + | += Jn | <Imm8> | <Imm32> ] = Ureg_sq ;

    [ Km + | += Kn | <Imm8> | <Imm32> ] = Ureg_s ;
L [ Km + | += Kn | <Imm8> | <Imm32> ] = Ureg_sd ;
Q [ Km + | += Kn | <Imm8> | <Imm32> ] = Ureg_sq ;

```

Function

These instructions load the destination memory location (to left of =) with the contents of the source register (to right of =). These instructions support pre-modify without update ([+] operator) addressing and post-modify with update ([+=] operator) addressing for memory accesses. For a description of addressing and memory access types, see [“IALU Data Addressing and Transfer Operations” on page 6-13](#).

Status Flags

None affected

Options

None

Example

For examples, see [“IALU Examples” on page 6-37](#).

IALU (Load/Store/Transfer) Instructions

Data Register Load and DAB Operation (Data Addressing)

Syntax

```
{X|Y|XY}Rs = {CB|BR} [Jm += Jn|<Imm8>|<Imm32>] ;  
{X|Y|XY}Rsd = {CB|BR} L [Jm += Jn|<Imm8>|<Imm32>] ;  
{XY|YX}Rs = {CB|BR} L [Jm += Jn|<Imm8>|<Imm32>] ;  
{X|Y|XY}Rsq = {CB|BR|DAB|SDAB} Q [Jm += Jn|<Imm8>|<Imm32>] ;  
{XY|YX}Rsd = {CB|BR|DAB|SDAB} Q [Jm += Jn|<Imm8>|<Imm32>] ;  
  
{X|Y|XY}Rs = {CB|BR} [Km += Kn|<Imm8>|<Imm32>] ;  
{X|Y|XY}Rsd = {CB|BR} L [Km += Kn|<Imm8>|<Imm32>] ;  
{XY|YX}Rs = {CB|BR} L [Km += Kn|<Imm8>|<Imm32>] ;  
{X|Y|XY}Rsq = {CB|BR|DAB|SDAB} Q [Km += Kn|<Imm8>|<Imm32>] ;  
{XY|YX}Rsd = {CB|BR|DAB|SDAB} Q [Km += Kn|<Imm8>|<Imm32>] ;
```

```
/* R suffix indicates: _s=single, _sd=double, _sq=quad */  
/* m must be 0,1,2, or 3 for bit reverse or circular buffers */
```

Function

These instructions load the destination register (to left of =) with the contents of the source memory location (to right of =). These instructions support post-modify with update ([+=] operator) addressing for memory accesses. For a description of addressing and memory access types, see [“IALU Data Addressing and Transfer Operations” on page 6-13](#).

Status Flags

None affected

Options

()	Linear addressing
CB	Circular buffer addressing
BR	Bit-reversed output

DAB Data alignment buffer access

SDAB Short data alignment buffer access

Example

For examples, see [“IALU Examples”](#) on page 6-37.

IALU (Load/Store/Transfer) Instructions

Data Register Store (Data Addressing)

Syntax

```
{CB|BR} [Jm += Jn|<Imm8>|<Imm32>] = {X|Y}Rs ;
{CB|BR} L [Jm += Jn|<Imm8>|<Imm32>] = {X|Y}Rsd ;
{CB|BR} L [Jm += Jn|<Imm8>|<Imm32>] = {XY|YX}Rs ;
{CB|BR} Q [Jm += Jn|<Imm8>|<Imm32>] = {X|Y}Rsq ;
{CB|BR} Q [Jm += Jn|<Imm8>|<Imm32>] = {XY|YX}Rsd ;

{CB|BR} [Km += Kn|<Imm8>|<Imm32>] = {X|Y}Rs ;
{CB|BR} L [Km += Kn|<Imm8>|<Imm32>] = {X|Y}Rsd ;
{CB|BR} L [Km += Kn|<Imm8>|<Imm32>] = {XY|YX}Rs ;
{CB|BR} Q [Km += Kn|<Imm8>|<Imm32>] = {X|Y}Rsq ;
{CB|BR} Q [Km += Kn|<Imm8>|<Imm32>] = {XY|YX}Rsd ;
/* R suffix indicates: _s=single, _sd=double, _sq=quad */

/* m = 0,1,2 or 3 for bit reverse or circular buffers */
```

Function

These instructions load the destination memory location (to left of =) with the contents of the source register (to right of =). These instructions support post-modify with update ([+=] operator) addressing for memory accesses. For a description of addressing and memory access types, see [“IALU Data Addressing and Transfer Operations” on page 6-13](#).

Status Flags

None affected

Options

()	Linear addressing
CB	Circular buffer addressing

BR Bit-reversed output

Example

For examples, see [“IALU Examples”](#) on page 6-37.

IALU (Load/Store/Transfer) Instructions

Universal Register Transfer

Syntax

```
Ureg_s = <Imm15>|<Imm32> ;  
Ureg_s = Ureg_m ;  
Ureg_sd = Ureg_md ;  
Ureg_sq = Ureg_mq ;
```

Function

The *Ureg=Ureg* instructions move data from any source register in the chip to any destination register in the chip. The source and destination registers are identified by a group (six bits) and a register (five bits). This type of instruction may be executed by one of the IALUs according to programming. The assembler decides which IALU executes the instruction. To make a SIMD register transfer inside both compute blocks, use the compute block broadcast in both source and destination *Ureg* groups.



This is the only instruction that can access groups 63:32. In all other instructions, only register groups 31:0 can be accessed.

The *Ureg=Imm* instruction loads data word into a destination register. The destination register is identified by a group (five bits) and a register (five bits). The data is 15 bits sign extended unless there is an immediate extension in the instruction line. Immediate extension instructions are defined in [“Immediate Extension Operations” on page 6-36](#).

This type of instruction may be executed by either one of the IALUs— by the J-IALU or by the K-IALU. The assembler decides which IALU executes the instruction.

ALU Ureg Transfer Exceptions

The IALU *Ureg* transfer instructions may cause exceptions when the source *Ureg* is “compute XY broadcast” and the destination is not “compute XY broadcast”. For example:

$xR3:0 = R7:6; \Rightarrow$ *illegal and causes an exception, but*
 $R3:0 = R7:4; \Rightarrow$ *legal*

Refer to “[Program Sequencer](#)” on page 1-13 and “[IALU Data Addressing and Transfer Operations](#)” on page 6-13 for an explanation of how interrupts are introduced with transfer exceptions.

Example

```
xR3:0 = yR31:28;;
```

This instruction moves four registers between compute block X and Y.

```
SDRCON= K0;;
```

This instruction transfers data from K0 to the SDRCON register.

```
R3:2 = R9:8;;
```

This instruction transfers data from R9:8 to R3:2 in both compute blocks simultaneously.

```
XR0 = 0x12345678 ;;
```

```
/* Because this instruction uses 32-bit data, this instruction
must use the first instruction slot, so the DSP can use the sec-
ond instruction slot for the immediate extension data. */
```

Sequencer Instructions

The sequencer fetches instructions from memory and executes program flow control instructions. The operations that the sequencer supports include:


- Supply address of next instruction to fetch
- Maintain instruction alignment buffer (IAB), caching fetched instructions
- Maintain branch target buffer (BTB), reducing branch delays
- Decrement loop counters
- Evaluate conditions (for conditional instructions)
- Respond to interrupts (with changes to program flow)

For a description of sequencer operations, conditional execution, pipeline effects, and examples, see [“Program Sequencer” on page 7-1](#).

The conventions used in these reference pages for representing register names, optional items, and choices are covered in detail in [“Register File Registers” on page 2-5](#). Briefly, these conventions are:

- { } – the curly braces enclose options; these braces are not part of the instruction syntax.
- | – the vertical bars separate choices; these bars are not part of the instruction syntax.
- *Label* – the program label in italic represents user-selectable program label, PC-relative 16- or 32-bit address, or a 32-bit absolute address. When a program *Label* is used instead of an address, the assembler converts the *Label* to an address, using a 16-bit address

when the *Label* is contained in the same program `.SECTION` as the branch instruction and using a 32-bit address when the *Label* is not in the same program `.SECTION` as the branch instruction. For more information on relative and absolute addresses and branching, see [“Branching Execution” on page 7-16](#).

-  Each instruction presented here occupies one instruction slot in an instruction line. For more information about instruction lines and instruction combination constraints, see [“Instruction Line Syntax and Structure” on page 1-20](#) and [“Instruction Parallelism Rules” on page 1-24](#).

Sequencer Instructions

Jump/Call

Syntax

```
{IF Condition,} JUMP|CALL <Label> {(NP)} {(ABS)} ;
```

Function

These instructions provide branching execution through jumps and calls. A `JUMP` instruction transfers execution to address *Label* (or an immediate 16- or 32-bit address). A `CALL` instruction transfers execution to address *Label* (or to an immediate 16- or 32-bit address). When processing a call, the sequencer writes the return address (next sequential address after the call) to the `CJMP` register, then jumps to the subroutine address.

If the branch is conditional (prefixed with `If Condition`), the branch is executed if a condition is specified and is true. If the branch prediction option is set to Not Predicted (NP), the sequencer assumes that the branch is not taken. Unless the absolute address option (ABS) is used, this sequencer uses a PC-relative address for the branch. For more information, see [“Branching Execution” on page 7-16](#).

Options

NP	Branch prediction is set to Not Predicted
ABS	Target address is in immediate field; if using the ABS option and a negative number is used (in place of <label>), the number is zero (not sign) extended.

Example

```
R21 = R21 + R31;;  
IF AEQ, JUMP label1;;  
/* Will use logical OR of AEQ of compute block X and Y.  
   If true will jump to label1. */  
XFR31 = R6 * R2;;
```

```
IF XMLT, JUMP label2;;
/* Will use mult condition from compute block X
   If true will jump to label2. */
XR31 += ASHIFT R0 BY 1;;
IF XSEQ, JUMP label3 (NP);;
/* Will use shift condition from compute block X
   If true will jump to label3
   NP indicates no branch prediction */

R21 = R21 + R31;;
IF AEQ, CALL label1;;
/* Will use logical OR of AEQ of compute block X and Y.
   If true will call to label1. */
XFR31 = R6 * R2;;
IF XMLT, CALL label2;;
/* Will use mult condition from compute block X
   If true will call to label2. */
XR31 += ASHIFT R0 BY 1;;
IF XSEQ, CALL label3 (NP);;
/* Will use shift condition from compute block X
   If true will call to label3
   NP indicates no branch prediction */
```

Sequencer Instructions

Computed Jump/Call


Syntax

```
{IF Condition,} CJMP|CJMP_CALL {(NP)} {(ABS)} ;
```

Function

These instructions provide branching execution through computed jumps and calls. A `CJMP` instruction at the end of the subroutine (branched to using a `CALL`) causes the sequencer to jump to the address in the `CJMP` register. A `CJMP_CALL` instruction transfers execution to a subroutine using a computed jump address (`CJMP` register). One way to load the computed jump address is to use the (`CJMP`) option on the `IALU` add/subtract instruction. The `CJMP_CALL` transfers execution to the address indicated by the `CJMP` register, then loads the return address into `CJMP`. The `CJMP` instruction at the end of the subroutine causes the sequencer to jump to the address in the `CJMP` register.

If the branch is conditional (prefixed with `If Condition`), the branch is executed if a condition is specified and is true. If the branch prediction option is set to `Not Predicted (NP)`, the sequencer assumes that the branch is not taken. Unless the absolute address option (`ABS`) is used, this sequencer uses a `PC`-relative address for the branch. For more information, see [“Branching Execution” on page 7-16](#).

 If the prediction is true (option `NP` is not set), the call must be absolute (option `ABS` must be set).

Options

`NP` Branch prediction is `Not Predicted`

`ABS` Target address is in the immediate field

See section [“Branch Target Buffer \(BTB\)” on page 7-34](#) for more on `NP` and `ABS` options.

Example

```
IF AEQ, CJMP (ABS) ;; /* predicted, absolute address */
IF AEQ, CJMP (ABS) (NP) ;; /* NOT predicted, absolute address */
IF AEQ, CJMP (NP) ;; /* NOT predicted, PC-relative address */

IF AEQ, CJMP_CALL (ABS) ;; /* predicted, absolute address */
IF AEQ, CJMP_CALL (ABS) (NP) ;; /* NOT predicted, absolute
address */
IF AEQ, CJMP_CALL (NP) ;; /* NOT predicted, PC-relative address
*/
```

Sequencer Instructions

Return (from Interrupt)

Syntax

```
{IF Condition,} RETI|RTI {{(NP)}} {{(ABS)}} ;
```

Function

The sequencer supports interrupting execution through hardware interrupts (external IRQ3-0 pins and internal process conditions) and software interrupts (program sets an interrupt's latch bit). [Figure 8-64](#) provides a comparison of interrupt service variations using the RETI and RTI instructions.

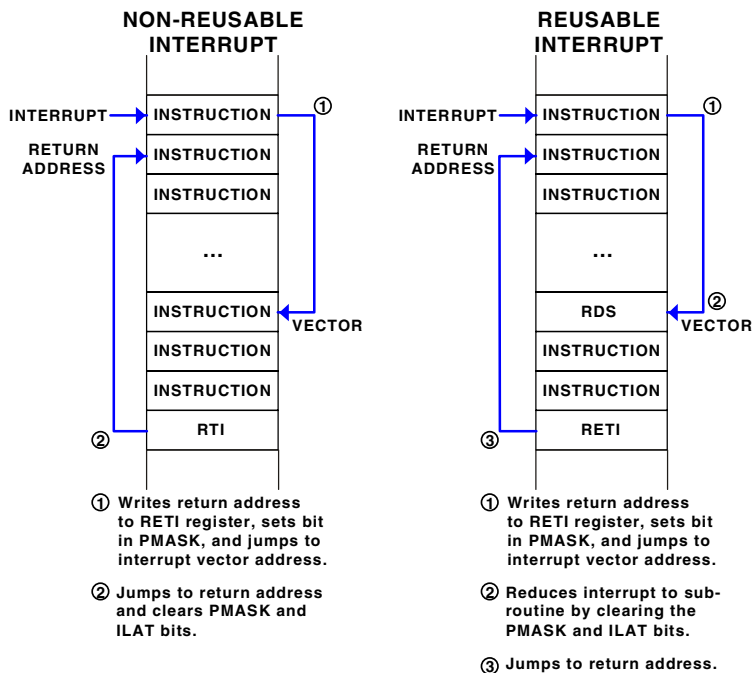


Figure 8-64. Non-Reusable Versus Reusable Interrupt Service

Sequencer Instructions

Reduce (Interrupt to Subroutine)

Syntax

```
{IF Condition,} RDS ;
```

Function

This instruction reduces the function to a subroutine, making the interrupt reusable. See [Figure 8-64 on page 8-234](#) for additional details about interrupt service routines for non-reusable interrupts and reusable interrupts.

If the reduce instruction is conditional (prefixed with If Condition), the reduction is executed if a condition is specified and is true. For more information, see [“Interrupting Execution” on page 7-20](#).

Options

None

Example

```
<interrupt service code part 1>  
RDS;;  
<interrupt service code part 2>
```

```
RETI;; /* After using RDS return is by RETI and not RTI. */
```

In part 1, all the interrupts that have a lower priority than the one currently serviced are disabled. In part 2, no interrupts are disabled following the RDS (which can be also conditional). Note that if this interrupt happens during the execution of a lower priority interrupt, the lower priority interrupt bit in `PMASK` is still set after the RDS.


If – Do (Conditional Execution)

Syntax

```
IF Condition;
    DO, instruction; DO, instruction; DO, instruction ;;
/* This syntax permits up to three instructions to be controlled
by a condition. Omitting the DO before the instruction makes the
instruction unconditional. */
```

Function

Any instruction in the instruction set can be conditional. The condition can be either the inverse of the branch condition (any type of branch instruction) or a stand-alone condition. For more information, see [“Conditional Execution” on page 7-12](#).

 The NOP, IDLE (1p), BTBINV, TRAP (<imm>), and EMUTRAP instructions may not be conditional. The following instruction line for example is not legal:

```
if aeq; do idle;; /* ILLEGAL! */
```

Example

```
if JEQ; do, J0=J1+J2; K0=K1+K2;;
```

When JEQ evaluates true, the J-IALU instruction is executed and when it evaluates false, the J-IALU instruction is not executed. The K-IALU instruction is always executed.

Sequencer Instructions


If – Else (Conditional Sequencing and Execution)

Syntax

```
IF Condition, JUMP|CALL|CJMP|CJMP_CALL ;  
    ELSE, instruction; ELSE, instruction; ELSE, instruction ;;  
/* This syntax permits up to three instructions to be controlled  
by a condition. Omitting the ELSE before the instruction makes  
the instruction unconditional. */
```

Function

Any instruction in the instruction set can be conditional. The condition can be either the inverse of the branch condition (any type of branch instruction) or a stand-alone condition. For more information, see “[Conditional Execution](#)” on page 7-12.

 The NOP, IDLE (1p), BTBINV, TRAP (<imm>), and EMUTRAP instructions may not be conditional. The following instruction line for example is not legal:

```
if aeq; do idle;; /* ILLEGAL! */
```

Example

```
if MEQ, cjmp; else, xR0 = R5 + R6; yR8 - R9 * R10;;
```

If previous multiply result is zero, the CJMP is taken and the ADD instruction is not executed. If previous multiply result is not zero, the ADD instruction is executed. The MUL instruction is always executed.

Static Flag Registers

Syntax

```
SF1|SF0 = Condition ;
```

```
SF1|SF0 += AND|OR|XOR Condition ;
```

Function

This instruction sets the static condition flags, defining the conditions upon which they are dependent.

Conditions are updated every time that the flags that create them are updated. You may need to use a condition that was created a few lines before, and has since been changed. In order to keep a condition valid, you can use the Static Condition Flag (SFREG) register. The SFREG can be loaded with the condition when valid, and used later. Another use for the SFREG is for complex conditions. Functions can be defined between the old value of static condition flags and other conditions.

Example

```
XSF0 = XMLT ;; /* Static flag xSF0 is set to XMLT. */
XSF1 += OR XAEQ ;; /* Static flag xSF1 ORed with XAEQ. */
```

Sequencer Instructions

Idle

Syntax

```
IDLE {(LP)} ;
```

Function

This instruction causes the TigerSHARC processor to go into `IDLE` state. In this state the TigerSHARC processor stops executing instructions and waits for any type of interrupt. If the `LP` option is set, the TigerSHARC processor will go into power save mode. In order to go into power save mode and keep the system in synchronization, a special sequence should be followed. See “Low Power Mode” in Chapter 5 Core Controls of the *ADSP-TS101 TigerSHARC Processor Hardware Reference*. Execution of `IDLE` instruction without `(LP)` option can be done any time.



The `NOP`, `IDLE (lp)`, `BTBINV`, `TRAP (<imm>)`, and `EMUTRAP` instructions may not be conditional. The following instruction line for example is not legal:

```
if aeq; do idle;; /* ILLEGAL! */
```


BTB Invalid

Syntax

BTBINV

Function

This instruction changes all BTB entries to be invalid. It must be executed whenever internal memory TigerSHARC processor code is replaced.

 The NOP, IDLE (1p), BTBINV, TRAP (<imm>), and EMUTRAP instructions may not be conditional. The following instruction line for example is not legal:

```
if aeq; do btbinv;; /* ILLEGAL! */
```

Sequencer Instructions

Trap

Syntax

```
TRAP (<Imm5>) ;;
```

Function

This instruction causes a trap and writes the 5-bit immediate into the SPVCMD field in the SQSTAT register. See “Exceptions” in Chapter 6 Interrupts of the *ADSP-TS101 TigerSHARC Processor Hardware Reference*.



The NOP, IDLE (lp), BTBINV, TRAP (<imm>), and EMUTRAP instructions may not be conditional. The following instruction line for example is not legal:

```
if aeq; do trap(0x3);; /* ILLEGAL! */
```

Emulator Trap

Syntax

```
EMUTRAP ;;
```

Function

This instruction causes an emulation trap after the current line. The next PC is saved in the DBGE register, and the sequencer starts reading instructions from the EMUIR register, which extracts instructions from JTAG. “Emulator” and “Emulation Debug” in the *ADSP-TS101 TigerSHARC Processor Hardware Reference*.



The NOP, IDLE (1p), BTBINV, TRAP (<imm>), and EMUTRAP instructions may not be conditional. The following instruction line for example is not legal:

```
if aeq; do emutrap;; /* ILLEGAL! */
```

Sequencer Instructions

No Operation

Syntax

```
NOP ;
```

Function

No operation – holds an instruction slot.



Can be executed with other Seq instructions in the same line.



The NOP, IDLE (1p), BTBINV, TRAP (<imm>), and EMUTRAP instructions may not be conditional. The following instruction line for example is not legal:

```
if aeq; do nop;; /* ILLEGAL! */
```


A QUICK REFERENCE

This chapter contains a concise description of the TigerSHARC processor programming model and assembly language. It is intended to be used as an assembly programming reference for language syntax and for typical code sequences. This chapter does not contain information on the functional aspects of the instruction set, nor does it describe in detail pipeline dependency and parallelism mechanisms.

Some sections in this text with which a programmer should be familiar before using this quick reference include:

- [“DSP Architecture” on page 1-6](#)
- [“Instruction Line Syntax and Structure” on page 1-20](#)
- [“Instruction Parallelism Rules” on page 1-24](#)
- [“Register File Registers” on page 2-5](#)
- [“ALU Operations” on page 3-5](#)
- [“Multiplier Operations” on page 4-4](#)
- [“Shifter Operations” on page 5-3](#)
- [“IALU Operations” on page 6-5](#)
- [“Sequencer Operations” on page 7-7](#)

ALU Quick Reference

For examples using these instructions, see “ALU Examples” on page 3-16 and “CLU Examples” on page 3-21.

Listing A-1. ALU Fixed-Point Instructions

$$\{X|Y|XY\}\{S|B\}Rs = Rm +|- Rn \{(\{S|SU\})\} ;^1$$

$$\{X|Y|XY\}\{L|S|B\}Rsd = Rmd +|- Rnd \{(\{S|SU\})\} ;^1$$

$$\{X|Y|XY\}Rs = Rm + CI \{-1\} ;$$

$$\{X|Y|XY\}LRsd = Rmd + CI \{-1\} ;$$

$$\{X|Y|XY\}\{S|B\}Rs = Rm +|- Rn + CI \{-1\} \{(\{S|SU\})\} ;^1$$

$$\{X|Y|XY\}\{L|S|B\}Rsd = Rmd +|- Rnd + CI \{-1\} \{(\{S|SU\})\} ;^1$$

$$\{X|Y|XY\}\{S|B\}Rs = (Rm +|- Rn)/2 \{(\{T\}\{U\})\} ;^2$$

$$\{X|Y|XY\}\{L|S|B\}Rsd = (Rmd +|- Rnd)/2 \{(\{T\}\{U\})\} ;^2$$

$$\{X|Y|XY\}\{S|B\}Rs = ABS Rm ;$$

$$\{X|Y|XY\}\{L|S|B\}Rsd = ABS Rmd ;$$

$$\{X|Y|XY\}\{S|B\}Rs = ABS (Rm + Rn) \{(X)\} ;^3$$

$$\{X|Y|XY\}\{L|S|B\}Rsd = ABS (Rmd + Rnd) \{(X)\} ;^3$$

$$\{X|Y|XY\}\{S|B\}Rs = ABS (Rm - Rn) \{(\{X\}\{U\})\} ;^4$$

$$\{X|Y|XY\}\{L|S|B\}Rsd = ABS (Rmd - Rnd) \{(\{X\}\{U\})\} ;^4$$

$$\{X|Y|XY\}\{S|B\}Rs = - Rm ;$$

$$\{X|Y|XY\}\{L|S|B\}Rsd = - Rmd ;$$

$$\{X|Y|XY\}\{S|B\}Rs = MAX|MIN (Rm, Rn) \{(\{U\}\{Z\})\} ;^5$$

$$\{X|Y|XY\}\{L|S|B\}Rsd = MAX|MIN (Rmd, Rnd) \{(\{U\}\{Z\})\} ;^5$$

¹ Options include: (): no saturation, (S): saturation, signed, (SU): saturation, unsigned

² Options include: (): signed, round-to-nearest even, (T): signed, truncate, (U): unsigned, round-to-nearest even, (TU): unsigned, truncate

³ Options include: (X): extend for ABS

⁴ Options include: (X): extend for ABS, (U): unsigned, round-to-nearest even, (XU): unsigned, extend

⁵ Options include: (): regular signed comparison, (U): comparison between unsigned numbers, (Z): returned result is zero if Rn is selected by MIN/MAX operation; otherwise returned result is Rm, (UZ): unsigned comparison with option (Z) as described above

```

{X|Y|XY}S|BRsd = VMAX|VMIN (Rmd, Rnd) ;
{X|Y|XY}{S|B}Rs = INC|DEC Rm {{(S|SU)}} ;1
{X|Y|XY}{L|S|B}Rsd = INC|DEC Rmd {{(S|SU)}} ;1
{X|Y|XY}{S|B}COMP(Rm, Rn) {(U)} ;5
{X|Y|XY}{L|S|B}COMP(Rnd,Rnd) {(U)} ;5
{X|Y|XY}{S|B}Rs = CLIP Rm BY Rn ;
{X|Y|XY}{L|S|B}Rsd = CLIP Rmd BY Rnd ;
{X|Y|XY}Rs = SUM S|B Rm {(U)} ;1
{X|Y|XY}Rs = SUM S|B Rmd {(U)} ;1
{X|Y|XY}Rs = ONES Rm|Rmd ;
{X|Y|XY}PR1:0 = Rmd ;
{X|Y|XY}Rsd = PR1:0 ;
{X|Y|XY}Rs = BFOINC Rmd ;
{X|Y|XY}PRO|PR1 += ABS (SRmd - SRnd){(U)} ;1
{X|Y|XY}PRO|PR1 += ABS (BRmd - BRnd){(U)} ;1
{X|Y|XY}PRO|PR1 += SUM SRm {(U)} ;1
{X|Y|XY}PRO|PR1 += SUM SRmd {(U)} ;1
{X|Y|XY}PRO|PR1 += SUM BRm {(U)} ;1
{X|Y|XY}PRO|PR1 += SUM BRmd {(U)} ;1
{X|Y|XY}{S|B}Rs = Rm + Rn, Ra = Rm - Rn ; (dual operation)
{X|Y|XY}{L|S|B}Rsd = Rmd + Rnd, Rad = Rmd - Rnd ; (dual operation)

```

Listing A-2. ALU Logical Operation Instructions

```

{X|Y|XY}Rs = PASS Rm ;
{X|Y|XY}LRsd = PASS Rmd ;
{X|Y|XY}Rs = Rm AND|AND NOT|OR|XOR Rn ;
{X|Y|XY}LRsd = Rmd AND|AND NOT|OR|XOR Rnd ;
{X|Y|XY}Rs = NOT Rm ;
{X|Y|XY}LRsd = NOT Rmd ;

```

¹ Options include: (): signed, (U): unsigned

ALU Quick Reference

Listing A-3. ALU Fixed-Point Miscellaneous

```
{X|Y|XY}Rsd = EXPAND SRm {+|- SRn} {{(I|IU)}} ;1
{X|Y|XY}Rsq = EXPAND SRmd {+|- SRnd} {{(I|IU)}} ;1
{X|Y|XY}Rsd = EXPAND BRm {+|- BRn} {{(I|IU)}} ;1
{X|Y|XY}Rsq = EXPAND BRmd {+|- BRnd} {{(I|IU)}} ;1
{X|Y|XY}SRs = COMPACT Rmd {+|- Rnd} {{(T|I|IS|ISU)}} ;2
{X|Y|XY}BRs = COMPACT SRmd {+|- SRnd} {{(T|I|IS|ISU)}} ;2
{X|Y|XY}BRsd = MERGE Rm, Rn ;
{X|Y|XY}BRsq = MERGE Rmd, Rnd ;
{X|Y|XY}SRsd = MERGE Rm, Rn ;
{X|Y|XY}SRsq = MERGE Rmd, Rnd ;
```

Listing A-4. Floating-Point ALU Instructions

```
{X|Y|XY}FRs = Rm +|- Rn {(T)} ;3
{X|Y|XY}FRsd = Rmd +|- Rnd {(T)} ;3
{X|Y|XY}FRs = (Rm +|- Rn)/2 {(T)} ;3
{X|Y|XY}FRsd = (Rmd +|- Rnd)/2 {(T)} ;3
{X|Y|XY}FRs = MAX|MIN (Rm +|- Rn) {(T)} ;4
{X|Y|XY}FRsd = MAX|MIN (Rmd +|- Rnd) {(T)} ;4
{X|Y|XY}FRs = ABS (Rm) ;
{X|Y|XY}FRsd = ABS (Rmd) ;
{X|Y|XY}FRs = ABS (Rm +|- Rn) {(T)} ;3
{X|Y|XY}FRsd = ABS (Rmd +|- Rnd) {(T)} ;3
{X|Y|XY}FRs = - Rm ;
{X|Y|XY}FRsd = - Rmd ;
{X|Y|XY}FCOMP (Rm, Rn) ;
```

¹ Options include: (): fractional, (I): integer signed, (IU): integer unsigned

² Options include: (): fractional round, (I): integer, no saturate, (T): fractional, truncate, (IS): integer, saturate, signed, (ISU): integer, saturate, unsigned

³ Options include: (): round, (T): truncate

⁴ Options include: (): round, (T): truncate (MIN only)

```

{X|Y|XY}FCOMP (Rmd, Rnd) ;
{X|Y|XY}Rs = FIX FRm|FRmd {BY Rn} {(T)} ;3
{X|Y|XY}FRs|FRsd = FLOAT Rm {BY Rn} {(T)} ;3
{X|Y|XY}FRsd = EXT D Rm ;
{X|Y|XY}FRs = SNGL Rmd ;
{X|Y|XY}FRs = CLIP Rm BY Rn ;
{X|Y|XY}FRsd = CLIP Rmd BY Rnd ;
{X|Y|XY}FRs = Rm COPYSIGN Rn ;
{X|Y|XY}FRsd = Rmd COPYSIGN Rnd ;
{X|Y|XY}FRs = SCALB FRm BY Rn ;
{X|Y|XY}FRsd = SCALB FRmd BY Rn ;
{X|Y|XY}FRs = PASS Rm ;
{X|Y|XY}FRsd = PASS Rmd ;
{X|Y|XY}FRs = RECIPS Rm ;
{X|Y|XY}FRsd = RECIPS Rmd ;
{X|Y|XY}FRs = RSQRTS Rm ;
{X|Y|XY}FRsd = RSQRTS Rmd ;
{X|Y|XY}Rs = MANT FRm|FRmd ;
{X|Y|XY}Rs = LOGB FRm|FRmd {(S)} ;1
{X|Y|XY}FRs = Rm + Rn, FRa = Rm - Rn ; (dual instruction)
{X|Y|XY}FRsd = Rmd + Rnd, FRad = Rmd - Rnd ; (dual instruction)

```

Listing A-5. Fixed-Point CLU Instructions

```

{X|Y|XY}{S}TRsd = TMAX(TRmd + Rmq_h, TRnd + Rmq_l) ;
{X|Y|XY}{S}TRsd = TMAX(TRmd - Rmq_h, TRnd - Rmq_l) ;
{X|Y|XY}{S}Rs = TMAX(TRm, TRn) ;
{X|Y|XY}{S}TRsd = MAX(TRmd + Rmq_h, TRnd + Rmq_l) ;
{X|Y|XY}{S}TRsd = MAX(TRmd - Rmq_h, TRnd - Rmq_l) ;
{X|Y|XY}Rs = TRm ;
{X|Y|XY}Rsd = TRmd ;
{X|Y|XY}Rsq = TRmq ;

```

¹ Options include: (): do not saturate, (S): saturate

Multiplier Quick Reference

$\{X|Y|XY\}TRs = Rm ;$
 $\{X|Y|XY\}TRsd = Rmd ;$
 $\{X|Y|XY\}TRsq = Rmq ;$
 $\{X|Y|XY\}Rs = THRm ;$
 $\{X|Y|XY\}Rsd = THRmd ;$
 $\{X|Y|XY\}Rsq = THRmq ;^1$
 $\{X|Y|XY\}THRs = Rm ;$
 $\{X|Y|XY\}THRsd = Rmd \{(i)\} ;$
 $\{X|Y|XY\}THRsq = Rmq ;^1$
 $\{X|Y|XY\}TRs = DESPREAD (Rmq, THRd) + TRn ;$
 $\{X|Y|XY\}Rs = TRs, TRs = DESPREAD (Rmq, THRd) ; (dual\ instruction)$
 $\{X|Y|XY\}Rsd = TRsd, TRsd = DESPREAD (Rmq, THRd) ; (dual\ instruction)$
 $\{X|Y|XY\}\{S\}TRsq = ACS (TRmd, TRnd, Rm) (TMAX) ;$
 $\{X|Y|XY\}Rsq = TRaq, \{S\}TRsq = ACS (TRmd, TRnd, Rm) (TMAX) ; (dual\ instr.)$
 $\{X|Y|XY\}Rsd = PERMUTE (Rmd, Rn) ;$
 $\{X|Y|XY\}Rsq = PERMUTE (Rmd, -Rmd, Rn) ;$

Multiplier Quick Reference

For examples using these instructions, see [“Multiplier Examples” on page 4-21](#).

Listing A-6. 32-Bit Fixed-Point Multiplication Instructions

$\{X|Y|XY\}Rs = Rm * Rn \{(\{U|nU\}\{I\}\{T\}\{S\})\} ;^2$
 $\{X|Y|XY\}Rsd = Rm * Rn \{(\{U|nU\}\{I\})\} ;$
 $\{X|Y|XY\}MRa += Rm * Rn \{(\{U\}\{I\}\{C|CR\})\} ;^3$

¹ Not implemented, but syntax reserved

² Options include: (): fractional, signed, and no saturation; (S): saturation, signed, (SU): saturation, unsigned

³ Options include: (): signed, round-to-nearest even, (T): signed, truncate, (U): unsigned, round-to-nearest even, (TU): unsigned, truncate

```

{X|Y|XY}MRa -= Rm * Rn {{{I}{C|CR}}};
{X|Y|XY}Rs = MRa, MRa += Rm * Rn {{{U}{I}{C}}}; dual operation
{X|Y|XY}Rsd = MRa, MRa += Rm * Rn {{{U}{I}{C}}}; dual operation
/* where MRa is either MR1:0 or MR3:2 */

```

Listing A-7. 16-Bit Fixed-Point Quad Multiplication Instructions

```

{X|Y|XY}Rsd = Rmd * Rnd {{{U}{I}{T}{S}}};
{X|Y|XY}Rsq = Rmd * Rnd {{{U}{I}}};
{X|Y|XY}MRb += Rmd * Rnd {{{U}{I}{C}}};
/* where MRb is either MR1:0, MR3:2 */
{X|Y|XY}MRb += Rmd * Rnd {{{U}{I}{C|CR}}};
{X|Y|XY}Rsd = MRb, MRb += Rmd * Rnd {{{I}{C}}}; dual operation
/* where MRb is either MR1:0, MR3:2, or MR3:0 */

```

Listing A-8. 16-Bit Fixed-Point Complex Multiplication Instructions

```

{X|Y|XY}MRa += Rm ** Rn {{{I}{C|CR}{J}}};
{X|Y|XY}Rs = MRa, MRa += Rm ** Rn {{{I}{C}{J}}}; dual operation
{X|Y|XY}Rsd = MRa, MRa += Rm ** Rn {{{I}{C}{J}}}; dual operation
/* where MRa is either MR1:0 or MR3:2 */

```

Listing A-9. 32- and 40-Bit Floating-Point Multiplication Instructions

```

{X|Y|XY}FRs = Rm * Rn {(T)};
{X|Y|XY}FRsd = Rmd * Rnd {(T)};

```

Listing A-10. Multiplier Register Load Instructions

```

{X|Y|XY}MRa = Rmd;
{X|Y|XY}MR4 = Rm;
{X|Y|XY}{S}Rsd = MRa {{{U}{S}}};
{X|Y|XY}Rsq = MR3:0 {{{U}{S}}};
{X|Y|XY}Rs = MR4;

```

Shifter Quick Reference

```
{X|Y|XY}Rs = COMPACT MRa {{{U}{I}{S}}};  
{X|Y|XY}SRsd = COMPACT MR3:0 {{{U}{I}{S}{C}}};  
/* where MRa is either MR1:0 or MR3:2 */
```

Shifter Quick Reference

For examples using these instructions, see [“Shifter Examples” on page 5-17](#).

Listing A-11. Shifter Instructions

```

{X|Y|XY}{B|S}Rs = LSHIFT|ASHIFT Rm BY Rn|<Imm> ;1,2
{X|Y|XY}{B|S|L}Rsd = LSHIFT|ASHIFT Rmd BY Rn|<Imm> ;1,2

{X|Y|XY}Rs = ROT Rm BY Rn|<Imm6> ;1
{X|Y|XY}{L}Rsd = ROT Rmd BY Rnd|<Imm> ;1,2

{X|Y|XY}Rs = FEXT Rm BY Rn|Rnd {(SE)} ;3
{X|Y|XY}LRsd = FEXT Rmd BY Rn|Rnd {(SE)} ;3

{X|Y|XY}Rs += FDEP Rm BY Rn|Rnd {(SE|ZF)} ;3
{X|Y|XY}LRsd += FDEP Rmd BY Rn|Rnd {(SE|ZF)} ;3

{X|Y|XY}Rs += MASK Rm BY Rn ;
{X|Y|XY}LRsd += MASK Rmd BY Rnd ;

{X|Y|XY}Rsd = GETBITS Rmq BY Rnd {(SE)} ;

{X|Y|XY}Rsd += PUTBITS Rmd BY Rnd ;

{X|Y|XY}BITEST Rm BY Rn|<Imm5> ;
{X|Y|XY}BITEST Rmd BY Rn|<Imm6> ;

{X|Y|XY}Rs = BCLR|BSET|BTGL Rm BY Rn|<Imm5> ;
{X|Y|XY}Rsd = BCLR|BSET|BTGL Rmd BY Rn|<Imm6> ;

{X|Y|XY}Rs = LDO|LD1 Rm|Rmd ;

{X|Y|XY}Rs = EXP Rm|Rmd ;

{X|Y}STAT = Rm ;

```

¹ The *Rn* data size (bits) for the shift magnitude varies with the output operand: Byte: 5, Short: 6, Normal: 7, Long: 8.

² The size in bits of the *Imm* data varies with the output operand: Byte: 4, Short: 5, Normal: 6, Long: 7.

³ The placement of the Pos8 and Len7 fields varies with the *Rn/Rnd* register, see [Figure 5-5 on page 5-8](#).

IALU Quick Reference

```
{X|Y}STATL = Rm ;  
{X|Y}Rs = {X|Y}STAT ;  
  
{X|Y|XY}BKFPTR Rmd, Rnd ;  
  
{X|Y|XY}Rsd = BFOTMP ;  
{X|Y|XY}BFOTMP = Rmd ;
```

IALU Quick Reference

For examples using these instructions, see [“IALU Examples” on page 6-37](#).

Listing A-12. IALU Arithmetic, Logical, and Function Instructions

```
Js = Jm +|- Jn|<Imm8>|<Imm32> {{(CJMP|CB|BR)}} ;  
JB0|JB1|JB2|JB3|JL0|JL1|JL2|JL3 = Jm +|- Jn|<Imm8>|<Imm32> ;  
Js = Jm + Jn|<Imm8>|<Imm32> + JC ;  
Js = Jm - Jn|<Imm8>|<Imm32> + JC - 1 ;  
Js = (Jm +|- Jn|<Imm8>|<Imm32>)/2 ;  
COMP(Jm, Jn|<Imm8>|<Imm32>) {(U)} ;  
Js = MAX|MIN (Jm, Jn|<Imm8>|<Imm32>) ;  
Js = ABS Jm ;  
Js = Jm OR|AND|XOR|AND NOT Jn|<Imm8>|<Imm32> ;  
Js = NOT Jm ;  
Js = ASHIFTR|LSHIFTR Jm ;  
Js = ROTR|ROTL Jm ;  
  
Ks = Km +|- Kn|<Imm8>|<Imm32> {{(CJMP|CB|BR)}} ;  
KB0|KB1|KB2|KB3|KL0|KL1|KL2|KL3 = Km +|- Kn|<Imm8>|<Imm32> ;  
Ks = Km + Kn|<Imm8>|<Imm32> + KC ;  
Ks = Km - Kn|<Imm8>|<Imm32> + KC - 1 ;  
Ks = (Km +|- Kn|<Imm8>|<Imm32>)/2 ;  
COMP(Km, Kn|<Imm8>|<Imm32>) {(U)} ;
```

```

Ks = MAX|MIN (Km, Kn|<Imm8>|<Imm32>) ;
Ks = ABS Km ;
Ks = Km OR|AND|XOR|AND NOT Kn|<Imm8>|<Imm32> ;
Ks = NOT Km ;
Ks = ASHIFTR|LSHIFTR Km ;
Ks = ROTR|ROTL Km ;

```

Listing A-13. IALU Ureg Register Load (Data Addressing) Instructions

```

Ureg_s = [Jm +|+= Jn|<Imm8>|<Imm32>] ;
Ureg_sd = L [Jm +|+= Jn|<Imm8>|<Imm32>] ;
Ureg_sq = Q [Jm +|+= Jn|<Imm8>|<Imm32>] ;

Ureg_s = [Km +|+= Kn|<Imm8>|<Imm32>] ;
Ureg_sd = L [Km +|+= Kn|<Imm8>|<Imm32>] ;
Ureg_sq = Q [Km +|+= Kn|<Imm8>|<Imm32>] ;

/* Ureg suffix indicates: _s=single, _sd=double, _sq=quad */

```

Listing A-14. IALU Dreg Register Load Data Addressing (and DAB Operation) Instructions

```

{X|Y|XY}Rs = {CB|BR} [Jm += Jn|<Imm8>|<Imm32>] ;
{X|Y|XY}Rsd = {CB|BR} L [Jm += Jn|<Imm8>|<Imm32>] ;
{XY|YX}Rs = {CB|BR} L [Jm += Jn|<Imm8>|<Imm32>] ;
{X|Y|XY}Rsq = {CB|BR|DAB|SDAB} Q [Jm += Jn|<Imm8>|<Imm32>] ;
{XY|YX}Rsd = {CB|BR|DAB|SDAB} Q [Jm += Jn|<Imm8>|<Imm32>] ;

{X|Y|XY}Rs = {CB|BR} [Km += Kn|<Imm8>|<Imm32>] ;
{X|Y|XY}Rsd = {CB|BR} L [Km += Kn|<Imm8>|<Imm32>] ;
{XY|YX}Rs = {CB|BR} L [Km += Kn|<Imm8>|<Imm32>] ;
{X|Y|XY}Rsq = {CB|BR|DAB|SDAB} Q [Km += Kn|<Imm8>|<Imm32>] ;
{XY|YX}Rsd = {CB|BR|DAB|SDAB} Q [Km += Kn|<Imm8>|<Imm32>] ;

/* R suffix indicates: _s=single, _sd=double, _sq=quad */
/* m must be 0,1,2, or 3 for bit reverse or circular buffers */

```

IALU Quick Reference

Listing A-15. IALU Ureg Register Store (Data Addressing) Instructions

```
[Jm +|+= Jn|<Imm8>|<Imm32>] = Ureg_s ;  
L [Jm +|+= Jn|<Imm8>|<Imm32>] = Ureg_sd ;  
Q [Jm +|+= Jn|<Imm8>|<Imm32>] = Ureg_sq ;  
  
[Km +|+= Kn|<Imm8>|<Imm32>] = Ureg_s ;  
L [Km +|+= Kn|<Imm8>|<Imm32>] = Ureg_sd ;  
Q [Km +|+= Kn|<Imm8>|<Imm32>] = Ureg_sq ;
```

Listing A-16. IALU Dreg Register Store (Data Addressing) Instructions

```
{CB|BR} [Jm += Jn|<Imm8>|<Imm32>] = {X|Y}Rs ;  
{CB|BR} L [Jm += Jn|<Imm8>|<Imm32>] = {X|Y}Rsd ;  
{CB|BR} L [Jm += Jn|<Imm8>|<Imm32>] = {XY|YX}Rs ;  
{CB|BR} Q [Jm += Jn|<Imm8>|<Imm32>] = {X|Y}Rsq ;  
{CB|BR} Q [Jm += Jn|<Imm8>|<Imm32>] = {XY|YX}Rsd ;  
  
{CB|BR} [Km += Kn|<Imm8>|<Imm32>] = {X|Y}Rs ;  
{CB|BR} L [Km += Kn|<Imm8>|<Imm32>] = {X|Y}Rsd ;  
{CB|BR} L [Km += Kn|<Imm8>|<Imm32>] = {XY|YX}Rs ;  
{CB|BR} Q [Km += Kn|<Imm8>|<Imm32>] = {X|Y}Rsq ;  
{CB|BR} Q [Km += Kn|<Imm8>|<Imm32>] = {XY|YX}Rsd ;  
  
/* R suffix indicates: _s=single, _sd=double, _sq=quad */  
/* m = 0,1,2 or 3 for bit reverse or circular buffers */
```

Listing A-17. IALU Universal Register Transfer Instructions

```
Ureg_s = <Imm15>|<Imm32> ;  
Ureg_s = Ureg_m ;  
Ureg_sd = Ureg_md ;  
Ureg_sq = Ureg_mq ;
```

Sequencer Quick Reference

For examples using these instructions, see [“Sequencer Examples” on page 7-72](#).

Listing A-18. Sequencer Instructions

```
{IF Condition,} JUMP|CALL <Label> {(NP)} {(ABS)} ;

{IF Condition,} CJMP|CJMP_CALL {(NP)} {(ABS)} ;

{IF Condition,} RETI|RTI {(NP)} {(ABS)} ;

{IF Condition,} RDS ;

IF Condition;
    DO, instruction; DO, instruction; DO, instruction ;;
/* This syntax permits up to three instructions to be controlled
by a condition. Omitting the DO before the instruction makes the
instruction unconditional. */

IF Condition, JUMP|CALL|CJMP|CJMP_CALL ;
    ELSE, instruction; ELSE, instruction; ELSE, instruction ;;
/* This syntax permits up to three instructions to be controlled
by a condition. Omitting the ELSE before the instruction makes
the instruction unconditional. */

SF1|SF0 = Condition ;

SF1|SF0 += AND|OR|XOR Condition ;

IDLE {(LP)} ;

BTBINV
```

Sequencer Quick Reference

TRAP (<Imm5>) ;;

EMUTRAP ;;

NOP ;

B REGISTER/BIT DEFINITIONS

When writing DSP programs, it is often necessary to set, clear, or test bits in the DSP's registers. While these bit operations can all be done by referring to the bit's location within a register or (for some operations) the register's address with a hexadecimal number, it is much easier to use symbols that correspond to the bit's or register's name. For convenience and consistency, Analog Devices provides a header file that provides these bit and registers definitions for the ADSP-TS101 TigerSHARC processor. For more information on TigerSHARC processor registers, see the "Registers and Memory" chapter in the *ADSP-TS101 TigerSHARC Processor Hardware Reference*.

Listing B-1. DEFTS101.H – Register and Bit #Defines File

```
/**
 * defTS101.h
 *
 * Copyright (c) 2001 Analog Devices, Inc., All rights reserved
 */

#if !defined(__DEFTS101_H_)
#define __DEFTS101_H_

////////////////////////////////////
// Macros
////////////////////////////////////
#if !defined(MAKE_BITMASK_)
```

```

#define MAKE_BITMASK_(x_) (1<<(x_)) // Make a bit mask from a bit
position
#endif
#if !defined(MAKE_LL_BITMASK_)
#define MAKE_LL_BITMASK_(x_) (1LL<<(x_)) // Make a bit mask from
a bit position (usable only in C)
#endif

//****
//*** Unmapped Registers Defines ****
//****

//*** XSTAT ****
// Bit positions
#define XSTAT_AZ_P ( 0)
#define XSTAT_AN_P ( 1)
#define XSTAT_AV_P ( 2)
#define XSTAT_AC_P ( 3)
#define XSTAT_MZ_P ( 4)
#define XSTAT_MN_P ( 5)
#define XSTAT_MV_P ( 6)
#define XSTAT_MU_P ( 7)
#define XSTAT_SZ_P ( 8)
#define XSTAT_SN_P ( 9)
#define XSTAT_BF_P (10)
#define XSTAT_AI_P (12)
#define XSTAT_MI_P (13)
#define XSTAT_UEN_P (20)
#define XSTAT_OEN_P (21)
#define XSTAT_IVEN_P (22)
#define XSTAT_AUS_P (24)
#define XSTAT_AVS_P (25)
#define XSTAT_AOS_P (26)
#define XSTAT_AIS_P (27)

```



```
#define XSTAT_MUS_P (28)
#define XSTAT_MVS_P (29)
#define XSTAT_MOS_P (30)
#define XSTAT_MIS_P (31)

// Bit Masks
#define XSTAT_AZ MAKE_BITMASK_(XSTAT_AZ_P)
#define XSTAT_AN MAKE_BITMASK_(XSTAT_AN_P)
#define XSTAT_AV MAKE_BITMASK_(XSTAT_AV_P)
#define XSTAT_AC MAKE_BITMASK_(XSTAT_AC_P)
#define XSTAT_MZ MAKE_BITMASK_(XSTAT_MZ_P)
#define XSTAT_MN MAKE_BITMASK_(XSTAT_MN_P)
#define XSTAT_MV MAKE_BITMASK_(XSTAT_MV_P)
#define XSTAT_MU MAKE_BITMASK_(XSTAT_MU_P)
#define XSTAT_SZ MAKE_BITMASK_(XSTAT_SZ_P)
#define XSTAT_SN MAKE_BITMASK_(XSTAT_SN_P)
#define XSTAT_BF MAKE_BITMASK_(XSTAT_BF_P)
#define XSTAT_AI MAKE_BITMASK_(XSTAT_AI_P)
#define XSTAT_MI MAKE_BITMASK_(XSTAT_MI_P)
#define XSTAT_UEN MAKE_BITMASK_(XSTAT_UEN_P)
#define XSTAT_OEN MAKE_BITMASK_(XSTAT_OEN_P)
#define XSTAT_IVEN MAKE_BITMASK_(XSTAT_IVEN_P)
#define XSTAT_AUS MAKE_BITMASK_(XSTAT_AUS_P)
#define XSTAT_AVS MAKE_BITMASK_(XSTAT_AVS_P)
#define XSTAT_AOS MAKE_BITMASK_(XSTAT_AOS_P)
#define XSTAT_AIS MAKE_BITMASK_(XSTAT_AIS_P)
#define XSTAT_MUS MAKE_BITMASK_(XSTAT_MUS_P)
#define XSTAT_MVS MAKE_BITMASK_(XSTAT_MVS_P)
#define XSTAT_MOS MAKE_BITMASK_(XSTAT_MOS_P)
#define XSTAT_MIS MAKE_BITMASK_(XSTAT_MIS_P)

//*** YSTAT *****
// Bit positions
#define YSTAT_AZ_P ( 0)
```

```

#define YSTAT_AN_P ( 1)
#define YSTAT_AV_P ( 2)
#define YSTAT_AC_P ( 3)
#define YSTAT_MZ_P ( 4)
#define YSTAT_MN_P ( 5)
#define YSTAT_MV_P ( 6)
#define YSTAT_MU_P ( 7)
#define YSTAT_SZ_P ( 8)
#define YSTAT_SN_P ( 9)
#define YSTAT_BF_P (10)
#define YSTAT_AI_P (12)
#define YSTAT_MI_P (13)
#define YSTAT_UEN_P (20)
#define YSTAT_OEN_P (21)
#define YSTAT_IVEN_P (22)
#define YSTAT_AUS_P (24)
#define YSTAT_AVSP (25)
#define YSTAT_AOS_P (26)
#define YSTAT_AIS_P (27)
#define YSTAT_MUS_P (28)
#define YSTAT_MVS_P (29)
#define YSTAT_MOS_P (30)
#define YSTAT_MIS_P (31)

// Bit Masks
#define YSTAT_AZ MAKE_BITMASK_(YSTAT_AZ_P)
#define YSTAT_AN MAKE_BITMASK_(YSTAT_AN_P)
#define YSTAT_AV MAKE_BITMASK_(YSTAT_AV_P)
#define YSTAT_AC MAKE_BITMASK_(YSTAT_AC_P)
#define YSTAT_MZ MAKE_BITMASK_(YSTAT_MZ_P)
#define YSTAT_MN MAKE_BITMASK_(YSTAT_MN_P)
#define YSTAT_MV MAKE_BITMASK_(YSTAT_MV_P)
#define YSTAT_MU MAKE_BITMASK_(YSTAT_MU_P)
#define YSTAT_SZ MAKE_BITMASK_(YSTAT_SZ_P)

```

```
#define YSTAT_SN MAKE_BITMASK_(YSTAT_SN_P)
#define YSTAT_BF MAKE_BITMASK_(YSTAT_BF_P)
#define YSTAT_AI MAKE_BITMASK_(YSTAT_AI_P)
#define YSTAT_MI MAKE_BITMASK_(YSTAT_MI_P)
#define YSTAT_UEN MAKE_BITMASK_(YSTAT_UEN_P)
#define YSTAT_OEN MAKE_BITMASK_(YSTAT_OEN_P)
#define YSTAT_IVEN MAKE_BITMASK_(YSTAT_IVEN_P)
#define YSTAT_AUS MAKE_BITMASK_(YSTAT_AUS_P)
#define YSTAT_AVS MAKE_BITMASK_(YSTAT_AVS_P)
#define YSTAT_AOS MAKE_BITMASK_(YSTAT_AOS_P)
#define YSTAT_AIS MAKE_BITMASK_(YSTAT_AIS_P)
#define YSTAT_MUS MAKE_BITMASK_(YSTAT_MUS_P)
#define YSTAT_MVS MAKE_BITMASK_(YSTAT_MVS_P)
#define YSTAT_MOS MAKE_BITMASK_(YSTAT_MOS_P)
#define YSTAT_MIS MAKE_BITMASK_(YSTAT_MIS_P)
```

```
//*****
```

```
//*** Mapped Register Defines *****
```

```
//*****
```

```
//*** X Comp Block *****
```

```
#define XR0_LOC (0x180000)
#define XR1_LOC (0x180001)
#define XR2_LOC (0x180002)
#define XR3_LOC (0x180003)
#define XR4_LOC (0x180004)
#define XR5_LOC (0x180005)
#define XR6_LOC (0x180006)
#define XR7_LOC (0x180007)
#define XR8_LOC (0x180008)
#define XR9_LOC (0x180009)
#define XR10_LOC (0x18000A)
```

```
#define XR11_LOC (0x18000B)
#define XR12_LOC (0x18000C)
#define XR13_LOC (0x18000D)
#define XR14_LOC (0x18000E)
#define XR15_LOC (0x18000F)
#define XR16_LOC (0x180010)
#define XR17_LOC (0x180011)
#define XR18_LOC (0x180012)
#define XR19_LOC (0x180013)
#define XR20_LOC (0x180014)
#define XR21_LOC (0x180015)
#define XR22_LOC (0x180016)
#define XR23_LOC (0x180017)
#define XR24_LOC (0x180018)
#define XR25_LOC (0x180019)
#define XR26_LOC (0x18001A)
#define XR27_LOC (0x18001B)
#define XR28_LOC (0x18001C)
#define XR29_LOC (0x18001D)
#define XR30_LOC (0x18001E)
#define XR31_LOC (0x18001F)
```

```
//*** Y Comp Block *****
```

```
#define YR0_LOC (0x180040)
#define YR1_LOC (0x180041)
#define YR2_LOC (0x180042)
#define YR3_LOC (0x180043)
#define YR4_LOC (0x180044)
#define YR5_LOC (0x180045)
#define YR6_LOC (0x180046)
#define YR7_LOC (0x180047)
#define YR8_LOC (0x180048)
#define YR9_LOC (0x180049)
```

```
#define YR10_LOC (0x18004A)
#define YR11_LOC (0x18004B)
#define YR12_LOC (0x18004C)
#define YR13_LOC (0x18004D)
#define YR14_LOC (0x18004E)
#define YR15_LOC (0x18004F)
#define YR16_LOC (0x180050)
#define YR17_LOC (0x180051)
#define YR18_LOC (0x180052)
#define YR19_LOC (0x180053)
#define YR20_LOC (0x180054)
#define YR21_LOC (0x180055)
#define YR22_LOC (0x180056)
#define YR23_LOC (0x180057)
#define YR24_LOC (0x180058)
#define YR25_LOC (0x180059)
#define YR26_LOC (0x18005A)
#define YR27_LOC (0x18005B)
#define YR28_LOC (0x18005C)
#define YR29_LOC (0x18005D)
#define YR30_LOC (0x18005E)
#define YR31_LOC (0x18005F)

//*** XY Comp Block Merged ****

#define XYR0_LOC (0x180080)
#define XYR1_LOC (0x180081)
#define XYR2_LOC (0x180082)
#define XYR3_LOC (0x180083)
#define XYR4_LOC (0x180084)
#define XYR5_LOC (0x180085)
#define XYR6_LOC (0x180086)
#define XYR7_LOC (0x180087)
#define XYR8_LOC (0x180088)
```

```
#define XYR9_LOC (0x180089)
#define XYR10_LOC (0x18008A)
#define XYR11_LOC (0x18008B)
#define XYR12_LOC (0x18008C)
#define XYR13_LOC (0x18008D)
#define XYR14_LOC (0x18008E)
#define XYR15_LOC (0x18008F)
#define XYR16_LOC (0x180090)
#define XYR17_LOC (0x180091)
#define XYR18_LOC (0x180092)
#define XYR19_LOC (0x180093)
#define XYR20_LOC (0x180094)
#define XYR21_LOC (0x180095)
#define XYR22_LOC (0x180096)
#define XYR23_LOC (0x180097)
#define XYR24_LOC (0x180098)
#define XYR25_LOC (0x180099)
#define XYR26_LOC (0x18009A)
#define XYR27_LOC (0x18009B)
#define XYR28_LOC (0x18009C)
#define XYR29_LOC (0x18009D)
#define XYR30_LOC (0x18009E)
#define XYR31_LOC (0x18009F)
```

```
/** YX Comp Block Merged ****
```

```
#define YXR0_LOC (0x1800C0)
#define YXR1_LOC (0x1800C1)
#define YXR2_LOC (0x1800C2)
#define YXR3_LOC (0x1800C3)
#define YXR4_LOC (0x1800C4)
#define YXR5_LOC (0x1800C5)
#define YXR6_LOC (0x1800C6)
#define YXR7_LOC (0x1800C7)
```

```
#define YXR8_LOC (0x1800C8)
#define YXR9_LOC (0x1800C9)
#define YXR10_LOC (0x1800CA)
#define YXR11_LOC (0x1800CB)
#define YXR12_LOC (0x1800CC)
#define YXR13_LOC (0x1800CD)
#define YXR14_LOC (0x1800CE)
#define YXR15_LOC (0x1800CF)
#define YXR16_LOC (0x1800D0)
#define YXR17_LOC (0x1800D1)
#define YXR18_LOC (0x1800D2)
#define YXR19_LOC (0x1800D3)
#define YXR20_LOC (0x1800D4)
#define YXR21_LOC (0x1800D5)
#define YXR22_LOC (0x1800D6)
#define YXR23_LOC (0x1800D7)
#define YXR24_LOC (0x1800D8)
#define YXR25_LOC (0x1800D9)
#define YXR26_LOC (0x1800DA)
#define YXR27_LOC (0x1800DB)
#define YXR28_LOC (0x1800DC)
#define YXR29_LOC (0x1800DD)
#define YXR30_LOC (0x1800DE)
#define YXR31_LOC (0x1800DF)

/** XY Comp Block Broadcast ****

#define XYBR0_LOC (0x180100)
#define XYBR1_LOC (0x180101)
#define XYBR2_LOC (0x180102)
#define XYBR3_LOC (0x180103)
#define XYBR4_LOC (0x180104)
#define XYBR5_LOC (0x180105)
#define XYBR6_LOC (0x180106)
```

```
#define XYBR7_LOC (0x180107)
#define XYBR8_LOC (0x180108)
#define XYBR9_LOC (0x180109)
#define XYBR10_LOC (0x18010A)
#define XYBR11_LOC (0x18010B)
#define XYBR12_LOC (0x18010C)
#define XYBR13_LOC (0x18010D)
#define XYBR14_LOC (0x18010E)
#define XYBR15_LOC (0x18010F)
#define XYBR16_LOC (0x180110)
#define XYBR17_LOC (0x180111)
#define XYBR18_LOC (0x180112)
#define XYBR19_LOC (0x180113)
#define XYBR20_LOC (0x180114)
#define XYBR21_LOC (0x180115)
#define XYBR22_LOC (0x180116)
#define XYBR23_LOC (0x180117)
#define XYBR24_LOC (0x180118)
#define XYBR25_LOC (0x180119)
#define XYBR26_LOC (0x18011A)
#define XYBR27_LOC (0x18011B)
#define XYBR28_LOC (0x18011C)
#define XYBR29_LOC (0x18011D)
#define XYBR30_LOC (0x18011E)
#define XYBR31_LOC (0x18011F)
```

```
//*** J-IALU ****
```

```
#define J0_LOC (0x180180)
#define J1_LOC (0x180181)
#define J2_LOC (0x180182)
#define J3_LOC (0x180183)
#define J4_LOC (0x180184)
#define J5_LOC (0x180185)
```



```
#define J6_LOC (0x180186)
#define J7_LOC (0x180187)
#define J8_LOC (0x180188)
#define J9_LOC (0x180189)
#define J10_LOC (0x18018A)
#define J11_LOC (0x18018B)
#define J12_LOC (0x18018C)
#define J13_LOC (0x18018D)
#define J14_LOC (0x18018E)
#define J15_LOC (0x18018F)
#define J16_LOC (0x180190)
#define J17_LOC (0x180191)
#define J18_LOC (0x180192)
#define J19_LOC (0x180193)
#define J20_LOC (0x180194)
#define J21_LOC (0x180195)
#define J22_LOC (0x180196)
#define J23_LOC (0x180197)
#define J24_LOC (0x180198)
#define J25_LOC (0x180199)
#define J26_LOC (0x18019A)
#define J27_LOC (0x18019B)
#define J28_LOC (0x18019C)
#define J29_LOC (0x18019D)
#define J30_LOC (0x18019E)
#define J31_LOC (0x18019F)

//*** K-IALU *****

#define K0_LOC (0x1801A0)
#define K1_LOC (0x1801A1)
#define K2_LOC (0x1801A2)
#define K3_LOC (0x1801A3)
#define K4_LOC (0x1801A4)
```

```
#define K5_LOC (0x1801A5)
#define K6_LOC (0x1801A6)
#define K7_LOC (0x1801A7)
#define K8_LOC (0x1801A8)
#define K9_LOC (0x1801A9)
#define K10_LOC (0x1801AA)
#define K11_LOC (0x1801AB)
#define K12_LOC (0x1801AC)
#define K13_LOC (0x1801AD)
#define K14_LOC (0x1801AE)
#define K15_LOC (0x1801AF)
#define K16_LOC (0x1801B0)
#define K17_LOC (0x1801B1)
#define K18_LOC (0x1801B2)
#define K19_LOC (0x1801B3)
#define K20_LOC (0x1801B4)
#define K21_LOC (0x1801B5)
#define K22_LOC (0x1801B6)
#define K23_LOC (0x1801B7)
#define K24_LOC (0x1801B8)
#define K25_LOC (0x1801B9)
#define K26_LOC (0x1801BA)
#define K27_LOC (0x1801BB)
#define K28_LOC (0x1801BC)
#define K29_LOC (0x1801BD)
#define K30_LOC (0x1801BE)
#define K31_LOC (0x1801BF)

/** J-IALU Circular ****

#define JB0_LOC (0x1801C0)
#define JB1_LOC (0x1801C1)
#define JB2_LOC (0x1801C2)
#define JB3_LOC (0x1801C3)
```

```
#define JL4_LOC (0x1801C4)
#define JL5_LOC (0x1801C5)
#define JL6_LOC (0x1801C6)
#define JL7_LOC (0x1801C7)

/** K-IALU Circular ****

#define KB0_LOC (0x1801E0)
#define KB1_LOC (0x1801E1)
#define KB2_LOC (0x1801E2)
#define KB3_LOC (0x1801E3)
#define KL4_LOC (0x1801E4)
#define KL5_LOC (0x1801E5)
#define KL6_LOC (0x1801E6)
#define KL7_LOC (0x1801E7)

/** Sequencer Registers ***

#define CJMP_LOC (0x180340)
#define RETI_LOC (0x180342)
#define RETS_LOC (0x180344)
#define DBGE_LOC (0x180345)
// #define ILATSTL_LOC (0x180346) // Should not be used due to silicon anomaly
// #define ILATSTH_LOC (0x180347)
#define LC0_LOC (0x180348)
#define LC1_LOC (0x180349)

/** ILAT and IMASK with bit defines ****

#define ILATL_LOC (0x18034A) // Use ILAT registers for reads only
#define ILATH_LOC (0x18034B)
#define IMASKL_LOC (0x18034C)
#define IMASKH_LOC (0x18034D)
```

```

#define PMASKL_LOC (0x18034E)
#define PMASKH_LOC (0x18034F)

// Bit positions
#define INT_RES0_P (0 )
#define INT_RES1_P (1 )
#define INT_TIMER0L_P (2 )
#define INT_TIMER1L_P (3 )
#define INT_RES4_P (4 )
#define INT_RES5_P (5 )
#define INT_LINK0_P (6 )
#define INT_LINK1_P (7 )
#define INT_LINK2_P (8 )
#define INT_LINK3_P (9 )
#define INT_RES_10_P (10)
#define INT_RES_11_P (11)
#define INT_RES_12_P (12)
#define INT_RES_13_P (13)
#define INT_DMA0_P (14)
#define INT_DMA1_P (15)
#define INT_DMA2_P (16)
#define INT_DMA3_P (17)
#define INT_RES18_P (18)
#define INT_RES19_P (19)
#define INT_RES20_P (20)
#define INT_RES21_P (21)
#define INT_DMA4_P (22)
#define INT_DMA5_P (23)
#define INT_DMA6_P (24)
#define INT_DMA7_P (25)
#define INT_RES26_P (26)
#define INT_RES27_P (27)
#define INT_RES28_P (28)
#define INT_DMA8_P (29)

```

```
#define INT_DMA9_P (30)
#define INT_DMA10_P (31)
#define INT_DMA11_P (0 )
#define INT_RES33_P (1 )
#define INT_RES34_P (2 )
#define INT_RES35_P (3 )
#define INT_RES36_P (4 )
#define INT_DMA12_P (5 )
#define INT_DMA13_P (6 )
#define INT_RES39_P (7 )
#define INT_RES40_P (8 )
#define INT_IRQ0_P (9 )
#define INT_IRQ1_P (10)
#define INT_IRQ2_P (11)
#define INT_IRQ3_P (12)
#define INT_RES45_P (13)
#define INT_RES46_P (14)
#define INT_RES47_P (15)
#define INT_VIRPT_P (16)
#define INT_RES49_P (17)
#define INT_BUSLOCK_P (18)
#define INT_RES51_P (19)
#define INT_TIMER0H_P (20)
#define INT_TIMER1H_P (21)
#define INT_RES54_P (22)
#define INT_RES55_P (23)
#define INT_RES56_P (24)
#define INT_HW_P (25)
#define INT_RES58_P (26)
#define INT_RES59_P (27)
#define INT_GIE_P (28)
#define INT_RES61_P (29)
#define INT_EXCEPT_P (30) //!!! Need to choose one or the other
!!!
```

```

#define INT_SW_P (30)
#define INT_EMUL_P (31)

// Bit Masks for C only
#define INT_RES0_64 MAKE_LL_BITMASK_(INT_RES0_P + 0 )
#define INT_RES1_64 MAKE_LL_BITMASK_(INT_RES1_P + 0 )
#define INT_TIMER0L_64 MAKE_LL_BITMASK_(INT_TIMER0L_P + 0 )
#define INT_TIMER1L_64 MAKE_LL_BITMASK_(INT_TIMER1L_P + 0 )
#define INT_RES4_64 MAKE_LL_BITMASK_(INT_RES4_P + 0 )
#define INT_RES5_64 MAKE_LL_BITMASK_(INT_RES5_P + 0 )
#define INT_LINK0_64 MAKE_LL_BITMASK_(INT_LINK0_P + 0 )
#define INT_LINK1_64 MAKE_LL_BITMASK_(INT_LINK1_P + 0 )
#define INT_LINK2_64 MAKE_LL_BITMASK_(INT_LINK2_P + 0 )
#define INT_LINK3_64 MAKE_LL_BITMASK_(INT_LINK3_P + 0 )
#define INT_RES_10_64 MAKE_LL_BITMASK_(INT_RES_10_P + 0 )
#define INT_RES_11_64 MAKE_LL_BITMASK_(INT_RES_11_P + 0 )
#define INT_RES_12_64 MAKE_LL_BITMASK_(INT_RES_12_P + 0 )
#define INT_RES_13_64 MAKE_LL_BITMASK_(INT_RES_13_P + 0 )
#define INT_DMA0_64 MAKE_LL_BITMASK_(INT_DMA0_P + 0 )
#define INT_DMA1_64 MAKE_LL_BITMASK_(INT_DMA1_P + 0 )
#define INT_DMA2_64 MAKE_LL_BITMASK_(INT_DMA2_P + 0 )
#define INT_DMA3_64 MAKE_LL_BITMASK_(INT_DMA3_P + 0 )
#define INT_RES18_64 MAKE_LL_BITMASK_(INT_RES18_P + 0 )
#define INT_RES19_64 MAKE_LL_BITMASK_(INT_RES19_P + 0 )
#define INT_RES20_64 MAKE_LL_BITMASK_(INT_RES20_P + 0 )
#define INT_RES21_64 MAKE_LL_BITMASK_(INT_RES21_P + 0 )
#define INT_DMA4_64 MAKE_LL_BITMASK_(INT_DMA4_P + 0 )
#define INT_DMA5_64 MAKE_LL_BITMASK_(INT_DMA5_P + 0 )
#define INT_DMA6_64 MAKE_LL_BITMASK_(INT_DMA6_P + 0 )
#define INT_DMA7_64 MAKE_LL_BITMASK_(INT_DMA7_P + 0 )
#define INT_RES26_64 MAKE_LL_BITMASK_(INT_RES26_P + 0 )
#define INT_RES27_64 MAKE_LL_BITMASK_(INT_RES27_P + 0 )
#define INT_RES28_64 MAKE_LL_BITMASK_(INT_RES28_P + 0 )
#define INT_DMA8_64 MAKE_LL_BITMASK_(INT_DMA8_P + 0 )

```

```
#define INT_DMA9_64 MAKE_LL_BITMASK_(INT_DMA9_P + 0 )
#define INT_DMA10_64 MAKE_LL_BITMASK_(INT_DMA10_P + 0 )
#define INT_DMA11_64 MAKE_LL_BITMASK_(INT_DMA11_P + 32)
#define INT_RES33_64 MAKE_LL_BITMASK_(INT_RES33_P + 32)
#define INT_RES34_64 MAKE_LL_BITMASK_(INT_RES34_P + 32)
#define INT_RES35_64 MAKE_LL_BITMASK_(INT_RES35_P + 32)
#define INT_RES36_64 MAKE_LL_BITMASK_(INT_RES36_P + 32)
#define INT_DMA12_64 MAKE_LL_BITMASK_(INT_DMA12_P + 32)
#define INT_DMA13_64 MAKE_LL_BITMASK_(INT_DMA13_P + 32)
#define INT_RES39_64 MAKE_LL_BITMASK_(INT_RES39_P + 32)
#define INT_RES40_64 MAKE_LL_BITMASK_(INT_RES40_P + 32)
#define INT_IRQ0_64 MAKE_LL_BITMASK_(INT_IRQ0_P + 32)
#define INT_IRQ1_64 MAKE_LL_BITMASK_(INT_IRQ1_P + 32)
#define INT_IRQ2_64 MAKE_LL_BITMASK_(INT_IRQ2_P + 32)
#define INT_IRQ3_64 MAKE_LL_BITMASK_(INT_IRQ3_P + 32)
#define INT_RES45_64 MAKE_LL_BITMASK_(INT_RES45_P + 32)
#define INT_RES46_64 MAKE_LL_BITMASK_(INT_RES46_P + 32)
#define INT_RES47_64 MAKE_LL_BITMASK_(INT_RES47_P + 32)
#define INT_VIRPT_64 MAKE_LL_BITMASK_(INT_VIRPT_P + 32)
#define INT_RES49_64 MAKE_LL_BITMASK_(INT_RES49_P + 32)
#define INT_BUSLOCK_64 MAKE_LL_BITMASK_(INT_BUSLOCK_P + 32)
#define INT_RES51_64 MAKE_LL_BITMASK_(INT_RES51_P + 32)
#define INT_TIMER0H_64 MAKE_LL_BITMASK_(INT_TIMER0H_P + 32)
#define INT_TIMER1H_64 MAKE_LL_BITMASK_(INT_TIMER1H_P + 32)
#define INT_RES54_64 MAKE_LL_BITMASK_(INT_RES54_P + 32)
#define INT_RES55_64 MAKE_LL_BITMASK_(INT_RES55_P + 32)
#define INT_RES56_64 MAKE_LL_BITMASK_(INT_RES56_P + 32)
#define INT_HW_64 MAKE_LL_BITMASK_(INT_HWERR_P + 32)
#define INT_RES58_64 MAKE_LL_BITMASK_(INT_RES58_P + 32)
#define INT_RES59_64 MAKE_LL_BITMASK_(INT_RES59_P + 32)
#define INT_RES60_64 MAKE_LL_BITMASK_(INT_RES60_P + 32)
#define INT_RES61_64 MAKE_LL_BITMASK_(INT_RES61_P + 32)
#define INT_EXCEPT_64 MAKE_LL_BITMASK_(INT_EXCEPT_P + 32)
#define INT_EMUL_64 MAKE_LL_BITMASK_(INT_EMUL_P + 32)
```

```

// Bit Masks
#define INT_RES0 MAKE_BITMASK_(INT_RES0_P )
#define INT_RES1 MAKE_BITMASK_(INT_RES1_P )
#define INT_TIMER0L MAKE_BITMASK_(INT_TIMER0L_P)
#define INT_TIMER1L MAKE_BITMASK_(INT_TIMER1L_P)
#define INT_RES4 MAKE_BITMASK_(INT_RES4_P )
#define INT_RES5 MAKE_BITMASK_(INT_RES5_P )
#define INT_LINK0 MAKE_BITMASK_(INT_LINK0_P )
#define INT_LINK1 MAKE_BITMASK_(INT_LINK1_P )
#define INT_LINK2 MAKE_BITMASK_(INT_LINK2_P )
#define INT_LINK3 MAKE_BITMASK_(INT_LINK3_P )
#define INT_RES_10 MAKE_BITMASK_(INT_RES_10_P )
#define INT_RES_11 MAKE_BITMASK_(INT_RES_11_P )
#define INT_RES_12 MAKE_BITMASK_(INT_RES_12_P )
#define INT_RES_13 MAKE_BITMASK_(INT_RES_13_P )
#define INT_DMA0 MAKE_BITMASK_(INT_DMA0_P )
#define INT_DMA1 MAKE_BITMASK_(INT_DMA1_P )
#define INT_DMA2 MAKE_BITMASK_(INT_DMA2_P )
#define INT_DMA3 MAKE_BITMASK_(INT_DMA3_P )
#define INT_RES18 MAKE_BITMASK_(INT_RES18_P )
#define INT_RES19 MAKE_BITMASK_(INT_RES19_P )
#define INT_RES20 MAKE_BITMASK_(INT_RES20_P )
#define INT_RES21 MAKE_BITMASK_(INT_RES21_P )
#define INT_DMA4 MAKE_BITMASK_(INT_DMA4_P )
#define INT_DMA5 MAKE_BITMASK_(INT_DMA5_P )
#define INT_DMA6 MAKE_BITMASK_(INT_DMA6_P )
#define INT_DMA7 MAKE_BITMASK_(INT_DMA7_P )
#define INT_RES26 MAKE_BITMASK_(INT_RES26_P )
#define INT_RES27 MAKE_BITMASK_(INT_RES27_P )
#define INT_RES28 MAKE_BITMASK_(INT_RES28_P )
#define INT_DMA8 MAKE_BITMASK_(INT_DMA8_P )
#define INT_DMA9 MAKE_BITMASK_(INT_DMA9_P )
#define INT_DMA10 MAKE_BITMASK_(INT_DMA10_P )

```



```
#define INT_DMA11 MAKE_BITMASK_(INT_DMA11_P )
#define INT_RES33 MAKE_BITMASK_(INT_RES33_P )
#define INT_RES34 MAKE_BITMASK_(INT_RES34_P )
#define INT_RES35 MAKE_BITMASK_(INT_RES35_P )
#define INT_RES36 MAKE_BITMASK_(INT_RES36_P )
#define INT_DMA12 MAKE_BITMASK_(INT_DMA12_P )
#define INT_DMA13 MAKE_BITMASK_(INT_DMA13_P )
#define INT_RES39 MAKE_BITMASK_(INT_RES39_P )
#define INT_RES40 MAKE_BITMASK_(INT_RES40_P )
#define INT_IRQ0 MAKE_BITMASK_(INT_IRQ0_P )
#define INT_IRQ1 MAKE_BITMASK_(INT_IRQ1_P )
#define INT_IRQ2 MAKE_BITMASK_(INT_IRQ2_P )
#define INT_IRQ3 MAKE_BITMASK_(INT_IRQ3_P )
#define INT_RES45 MAKE_BITMASK_(INT_RES45_P )
#define INT_RES46 MAKE_BITMASK_(INT_RES46_P )
#define INT_RES47 MAKE_BITMASK_(INT_RES47_P )
#define INT_VIRPT MAKE_BITMASK_(INT_VIRPT_P )
#define INT_RES49 MAKE_BITMASK_(INT_RES49_P )
#define INT_BUSLOCK MAKE_BITMASK_(INT_BUSLOCK_P)
#define INT_RES51 MAKE_BITMASK_(INT_RES51_P )
#define INT_TIMER0H MAKE_BITMASK_(INT_TIMER0H_P)
#define INT_TIMER1H MAKE_BITMASK_(INT_TIMER1H_P)
#define INT_RES54 MAKE_BITMASK_(INT_RES54_P )
#define INT_RES55 MAKE_BITMASK_(INT_RES55_P )
#define INT_RES56 MAKE_BITMASK_(INT_RES56_P )
#define INT_HWERR MAKE_BITMASK_(INT_HWERR_P )
#define INT_RES58 MAKE_BITMASK_(INT_RES58_P )
#define INT_RES59 MAKE_BITMASK_(INT_RES59_P )
#define INT_GIE MAKE_BITMASK_(INT_GIE_P )
#define INT_RES61 MAKE_BITMASK_(INT_RES61_P )
#define INT_SW MAKE_BITMASK_(INT_SW_P ) //!!! Need to choose !!!
#define INT_EXCEPT MAKE_BITMASK_(INT_EXCEPT_P )
#define INT_EMUL MAKE_BITMASK_(INT_EMUL_P )
```

```

/*****

#define TIMER0L_LOC (0x180350)
#define TIMER0H_LOC (0x180351)
#define TIMER1L_LOC (0x180352)
#define TIMER1H_LOC (0x180353)
#define TMRIN0L_LOC (0x180354)
#define TMRIN0H_LOC (0x180355)
#define TMRIN1L_LOC (0x180356)
#define TMRIN1H_LOC (0x180357)

/** SQCTL With Bit Defines ****
#define SQCTL_LOC (0x180358)
#define SQCTLST_LOC (0x180359)
#define SQCTLCL_LOC (0x18035A)

// Bit positions
#define SQCTL_BT BEN_P (0)
#define SQCTL_BTBLK_P (1)
#define SQCTL_SWRST_P (5)
#define SQCTL_DBGGEN_P (8)
#define SQCTL_NMOD_P (9)
#define SQCTL_TMR0RN_P (12)
#define SQCTL_TMR1RN_P (13)
#define SQCTL_IRQ0_EDGE_P (16)
#define SQCTL_IRQ1_EDGE_P (17)
#define SQCTL_IRQ2_EDGE_P (18)
#define SQCTL_IRQ3_EDGE_P (19)
#define SQCTL_FLAG0_EN_P (20) // FLAG0 output enable is bit 20 in
SQCTL
#define SQCTL_FLAG1_EN_P (21) // FLAG1 output enable is bit 21 in
SQCTL
#define SQCTL_FLAG2_EN_P (22) // FLAG2 output enable is bit 22 in
SQCTL

```

```
#define SQCTL_FLAG3_EN_P (23) // FLAG3 output enable is bit 23 in
SQCTL
#define SQCTL_FLAG0_OUT_P (24) // FLAG0 out pin is bit 24 in
SQCTL
#define SQCTL_FLAG1_OUT_P (25) // FLAG1 out pin is bit 25 in
SQCTL
#define SQCTL_FLAG2_OUT_P (26) // FLAG2 out pin is bit 26 in
SQCTL
#define SQCTL_FLAG3_OUT_P (27) // FLAG3 out pin is bit 27 in
SQCTL

// Bit Masks
#define SQCTL_BT BEN MAKE_BITMASK_(SQCTL_BT BEN_P)
#define SQCTL_BT BLK MAKE_BITMASK_(SQCTL_BT BLK_P)
#define SQCTL_SWRST MAKE_BITMASK_(SQCTL_SWRST_P)
#define SQCTL_DBGEN MAKE_BITMASK_(SQCTL_DBGEN_P)
#define SQCTL_NMOD MAKE_BITMASK_(SQCTL_NMOD_P)
#define SQCTL_TMR0RN MAKE_BITMASK_(SQCTL_TMR0RN_P)
#define SQCTL_TMR1RN MAKE_BITMASK_(SQCTL_TMR1RN_P)
#define SQCTL_IRQ0_EDGE MAKE_BITMASK_(SQCTL_IRQ0_EDGE_P)
#define SQCTL_IRQ1_EDGE MAKE_BITMASK_(SQCTL_IRQ1_EDGE_P)
#define SQCTL_IRQ2_EDGE MAKE_BITMASK_(SQCTL_IRQ2_EDGE_P)
#define SQCTL_IRQ3_EDGE MAKE_BITMASK_(SQCTL_IRQ3_EDGE_P)
#define SQCTL_FLAG0_EN MAKE_BITMASK_(SQCTL_FLAG0_EN_P)
#define SQCTL_FLAG1_EN MAKE_BITMASK_(SQCTL_FLAG1_EN_P)
#define SQCTL_FLAG2_EN MAKE_BITMASK_(SQCTL_FLAG2_EN_P)
#define SQCTL_FLAG3_EN MAKE_BITMASK_(SQCTL_FLAG3_EN_P)
#define SQCTL_FLAG0_OUT MAKE_BITMASK_(SQCTL_FLAG0_OUT_P)
#define SQCTL_FLAG1_OUT MAKE_BITMASK_(SQCTL_FLAG1_OUT_P)
#define SQCTL_FLAG2_OUT MAKE_BITMASK_(SQCTL_FLAG2_OUT_P)
#define SQCTL_FLAG3_OUT MAKE_BITMASK_(SQCTL_FLAG3_OUT_P)

/***/ SQSTAT With Bit Defines ***/
```

```

#define SQSTAT_LOC (0x18035B)

// Bit positions (of the masks)
#define SQSTAT_MODE_P (0)
#define SQSTAT_IDLE_P (2)
#define SQSTAT_SPVCMD_P (3)
#define SQSTAT_EXCAUSE_P (8)
#define SQSTAT_EMCAUSE_P (12)
#define SQSTAT_FLG_P (16)

// Bit masks
#define SQSTAT_MODE (0x00000003)
#define SQSTAT_IDLE (0x00000004)
#define SQSTAT_SPVCMD (0x000000F8)
#define SQSTAT_EXCAUSE (0x00000F00)
#define SQSTAT_EMCAUSE (0x0000F000)
#define SQSTAT_FLG (0x000F0000)

/** SFREG With Bit Defines ****

#define SFREG_LOC (0x18035C)

// Bit positions
#define SFREG_GSCF0_P (0)
#define SFREG_GSCF1_P (1)
#define SFREG_XSCF0_P (2)
#define SFREG_XSCF1_P (3)
#define SFREG_YSCF0_P (4)
#define SFREG_YSCF1_P (5)

// Bit Masks
#define SFREG_GSCF0 MAKE_BITMASK_(SFREG_GSCF0_P )
#define SFREG_GSCF1 MAKE_BITMASK_(SFREG_GSCF1_P )
#define SFREG_XSCF0 MAKE_BITMASK_(SFREG_XSCF0_P )

```

```
#define SFREG_XSCF1 MAKE_BITMASK_(SFREG_XSCF1_P )
#define SFREG_YSCF0 MAKE_BITMASK_(SFREG_YSCF0_P )
#define SFREG_YSCF1 MAKE_BITMASK_(SFREG_YSCF1_P )

//*****

//#define ILATCLL_LOC (0x18035E) //Should not be used due to silicon anomaly
//#define ILATCLH_LOC (0x18035F)

/** Emulation Registers **/

#define EMUCTL_LOC (0x180360)

// Bit positions
#define EMUCTL_EMEN_P (0)
#define EMUCTL_TEME_P (1)
#define EMUCTL_EMUOE_P (2)
#define EMUCTL_SPFDIS_P (3)

// Bit Masks
#define EMUCTL_EMEN MAKE_BITMASK_(EMUCTL_EMEN_P)
#define EMUCTL_TEME MAKE_BITMASK_(EMUCTL_TEME_P)
#define EMUCTL_EMUOE MAKE_BITMASK_(EMUCTL_EMUOE_P)
#define EMUCTL_SPFDIS MAKE_BITMASK_(EMUCTL_SPFDIS_P)

#define EMUSTAT_LOC (0x180361)

// Bit positions
#define EMUSTAT_EMUMOD_P (0)
#define EMUSTAT_IRFREE_P (1)

// Bit Masks
#define EMUSTAT_EMUMOD MAKE_BITMASK_(EMUSTAT_EMUMOD_P)
```

```

#define EMUSTAT_IRFREE MAKE_BITMASK_(EMUSTAT_IRFREE_P)

#define PRFM_LOC (0x180363)

// Bit Masks
// Non Granted Requests
#define PRFM_NGR_SEQ (0)
#define PRFM_NGR_JALU (1)
#define PRFM_NGR_KALU (2)
#define PRFM_NGR_DMAI (3)
#define PRFM_NGR_DMAE (4)
#define PRFM_NGR_BIU (5)

// Granted Requests
#define PRFM_GR_SEQ (6)
#define PRFM_GR_JALU (7)
#define PRFM_GR_KALU (8)
#define PRFM_GR_DMAI (9)
#define PRFM_GR_DMAE (10)
#define PRFM_GR_BIU (11)

// Bus 0
#define PRFM_BUS0_NORM (12)
#define PRFM_BUS0_LONG (13)
#define PRFM_BUS0_QUAD (14)

// Bus 1
#define PRFM_BUS1_NORM (15)
#define PRFM_BUS1_LONG (16)
#define PRFM_BUS1_QUAD (17)

// Bus 2
#define PRFM_BUS2_NORM (18)
#define PRFM_BUS2_LONG (19)

```

```
#define PRFM_BUS2_QUAD (20)

// Module Used
#define PRFM_MODULE_JALU (21)
#define PRFM_MODULE_KALU (22)
#define PRFM_MODULE_CBX (23)
#define PRFM_MODULE_CBY (24)
#define PRFM_MODULE_CTRL (25)

#define PRFM_VBT (26)
#define PRFM_SCYCLE (27)
#define PRFM_BTBP (28)
#define PRFM_ISL (29)
#define PRFM_CCYCLE (30)
#define PRFM_SUMEN (31)

#define CCNT0_LOC (0x180364)
#define CCNT1_LOC (0x180365)
#define PRFCNT_LOC (0x180366)
#define EMUDAT_LOC (0x180368)
#define EMUIR_LOC (0x18036C)
#define TRCBMASK_LOC (0x180370)
#define TRCBPTR_LOC (0x180378)
#define IDCODE_LOC (0x18037A)

/** TCBs With Bit Defines *****/

#define DCS0_LOC (0x180400)
#define DCD0_LOC (0x180404)
#define DCS1_LOC (0x180408)
#define DCD1_LOC (0x18040c)
#define DCS2_LOC (0x180410)
#define DCD2_LOC (0x180414)
```

```

#define DCS3_LOC (0x180418)
#define DCD3_LOC (0x18041C)
#define DC4_LOC (0x180420)
#define DC5_LOC (0x180424)
#define DC6_LOC (0x180428)
#define DC7_LOC (0x18042C)
#define DC8_LOC (0x180440)
#define DC9_LOC (0x180444)
#define DC10_LOC (0x180448)
#define DC11_LOC (0x18044C)
#define DC12_LOC (0x180458)
#define DC13_LOC (0x18045C)

// TYPES
#define TCB_EPROM (0xC0000000)
#define TCB_FLYBY (0xA0000000)
#define TCB_EXTMEM (0x80000000)
#define TCB_INTMEM (0x40000000)
#define TCB_LINK (0x20000000)
#define TCB_DISABLE (0x00000000)
// PRIORITY
#define TCB_HPRIORITY (0x10000000)
// 2D
#define TCB_TWODIM (0x08000000)
// GRANULARITY
#define TCB_QUAD (0x06000000)
#define TCB_LONG (0x04000000)
#define TCB_NORMAL (0x02000000)
// INTERRUPT
#define TCB_INT (0x01000000)
// DMA REQUEST
#define TCB_DMAR (0x00800000)
// CHAINING
#define TCB_CHAIN (0x00400000)

```



```
// CHAINED CHANNEL
#define TCB_DMA13DEST (0x002E0000)
#define TCB_DMA12DEST (0x002C0000)
#define TCB_DMA11DEST (0x00260000)
#define TCB_DMA10DEST (0x00240000)
#define TCB_DMA9DEST (0x00220000)
#define TCB_DMA8DEST (0x00200000)
#define TCB_DMA7DEST (0x00160000)
#define TCB_DMA6DEST (0x00140000)
#define TCB_DMA5DEST (0x00120000)
#define TCB_DMA4DEST (0x00100000)
#define TCB_DMA3DEST (0x000E0000)
#define TCB_DMA3SOURCE (0x000C0000)
#define TCB_DMA2DEST (0x000A0000)
#define TCB_DMA2SOURCE (0x00080000)
#define TCB_DMA1DEST (0x00060000)
#define TCB_DMA1SOURCE (0x00040000)
#define TCB_DMA0DEST (0x00020000)
#define TCB_DMA0SOURCE (0x00000000)
// MS FOR CHAIN POINTER
#define TCB_CHAINPTRM2 (0x00010000)
#define TCB_CHAINPTRM1 (0x00008000)
#define TCB_CHAINPTRM0 (0x00000000)

/***/ DMA Controls With Bit Defines *****/
#define DCNT_LOC (0x180460)
#define DCNTST_LOC (0x180464)
#define DCNTCL_LOC (0x180468)

// Bit positions
#define DCNT_DMA0_P (0)
#define DCNT_DMA1_P (1)
#define DCNT_DMA2_P (2)
#define DCNT_DMA3_P (3)
```

```

#define DCNT_DMA4_P (4)
#define DCNT_DMA5_P (5)
#define DCNT_DMA6_P (6)
#define DCNT_DMA7_P (7)
#define DCNT_DMA8_P (10)
#define DCNT_DMA9_P (11)
#define DCNT_DMA10_P (12)
#define DCNT_DMA11_P (13)
#define DCNT_DMA12_P (16)
#define DCNT_DMA13_P (17)

// Bit Masks
#define DCNT_DMA0 MAKE_BITMASK_(DCNT_DMA0_P)
#define DCNT_DMA1 MAKE_BITMASK_(DCNT_DMA1_P)
#define DCNT_DMA2 MAKE_BITMASK_(DCNT_DMA2_P)
#define DCNT_DMA3 MAKE_BITMASK_(DCNT_DMA3_P)
#define DCNT_DMA4 MAKE_BITMASK_(DCNT_DMA4_P)
#define DCNT_DMA5 MAKE_BITMASK_(DCNT_DMA5_P)
#define DCNT_DMA6 MAKE_BITMASK_(DCNT_DMA6_P)
#define DCNT_DMA7 MAKE_BITMASK_(DCNT_DMA7_P)
#define DCNT_DMA8 MAKE_BITMASK_(DCNT_DMA8_P)
#define DCNT_DMA9 MAKE_BITMASK_(DCNT_DMA9_P)
#define DCNT_DMA10 MAKE_BITMASK_(DCNT_DMA10_P)
#define DCNT_DMA11 MAKE_BITMASK_(DCNT_DMA11_P)
#define DCNT_DMA12 MAKE_BITMASK_(DCNT_DMA12_P)
#define DCNT_DMA13 MAKE_BITMASK_(DCNT_DMA13_P)

/** DMA Status With Bit Defines */
#define DSTATL_LOC (0x18046C)
#define DSTATCL_LOC (0x180470)

// Bit Masks
#define DSTAT_IDLE (0x00000000)
#define DSTAT_ACT (0x00000001)

```

```
#define DSTAT_DONE (0x00000002)
#define DSTAT_ACT_ERR (0x00000004)
#define DSTAT_CFG_ERR (0x00000005)
#define DSTAT_ADD_ERR (0x00000007)

// Field Extracts - use with fext instruction
#define DSTATL0 (0x0003) // 0th position of length 3
#define DSTATL1 (0x0303) // 3rd position of length 3
#define DSTATL2 (0x0603) // 6th position of length 3
#define DSTATL3 (0x0903) // 9th position of length 3
#define DSTATL4 (0x0C03) // 12th position of length 3
#define DSTATL5 (0x0F03) // 15th position of length 3
#define DSTATL6 (0x1203) // 18th position of length 3
#define DSTATL7 (0x1503) // 21st position of length 3

#define DSTATH_LOC (0x18046D)
#define DSTATCH_LOC (0x180471)

#define DSTATH8 (0x0003) // 0th position of length 3
#define DSTATH9 (0x0303) // 3rd position of length 3
#define DSTATH10 (0x0603) // 6th position of length 3
#define DSTATH11 (0x0903) // 9th position of length 3
#define DSTATH12 (0x1203) // 18th position of length 3
#define DSTATH13 (0x1503) // 21st position of length 3

/** SYSCON register With Bit Masks *****/
#define SYSCON_LOC (0x180480)

// Bit Masks
#define SYSCON_MS0_IDLE (0x00000001)
#define SYSCON_MS0_WT0 (0x00000000)
#define SYSCON_MS0_WT1 (0x00000002)
#define SYSCON_MS0_WT2 (0x00000004)
#define SYSCON_MS0_WT3 (0x00000006)
```

```

#define SYSCON_MSO_PIPE1 (0x00000000)
#define SYSCON_MSO_PIPE2 (0x00000008)
#define SYSCON_MSO_PIPE3 (0x00000010)
#define SYSCON_MSO_PIPE4 (0x00000018)
#define SYSCON_MSO_SLOW (0x00000020)
#define SYSCON_MS1_IDLE (SYSCON_MSO_IDLE << 6)
#define SYSCON_MS1_WT0 (SYSCON_MSO_WT0 << 6)
#define SYSCON_MS1_WT1 (SYSCON_MSO_WT1 << 6)
#define SYSCON_MS1_WT2 (SYSCON_MSO_WT2 << 6)
#define SYSCON_MS1_WT3 (SYSCON_MSO_WT3 << 6)
#define SYSCON_MS1_PIPE1 (SYSCON_MSO_PIPE1 << 6)
#define SYSCON_MS1_PIPE2 (SYSCON_MSO_PIPE2 << 6)
#define SYSCON_MS1_PIPE3 (SYSCON_MSO_PIPE3 << 6)
#define SYSCON_MS1_PIPE4 (SYSCON_MSO_PIPE4 << 6)
#define SYSCON_MS1_SLOW (SYSCON_MSO_SLOW << 6)
#define SYSCON_MSH_IDLE (SYSCON_MSO_IDLE << 12)
#define SYSCON_MSH_WT0 (SYSCON_MSO_WT0 << 12)
#define SYSCON_MSH_WT1 (SYSCON_MSO_WT1 << 12)
#define SYSCON_MSH_WT2 (SYSCON_MSO_WT2 << 12)
#define SYSCON_MSH_WT3 (SYSCON_MSO_WT3 << 12)
#define SYSCON_MSH_PIPE1 (SYSCON_MSO_PIPE1 << 12)
#define SYSCON_MSH_PIPE2 (SYSCON_MSO_PIPE2 << 12)
#define SYSCON_MSH_PIPE3 (SYSCON_MSO_PIPE3 << 12)
#define SYSCON_MSH_PIPE4 (SYSCON_MSO_PIPE4 << 12)
#define SYSCON_MSH_SLOW (SYSCON_MSO_SLOW << 12)
#define SYSCON_MEM_WID64 (0x00080000)
#define SYSCON_MP_WID64 (0x00100000)
#define SYSCON_HOST_WID64 (0x00200000)

/** SDRCON register With Bit Masks *****/
#define SDRCON_LOC (0x180484)

// Bit Masks
#define SDRCON_ENBL (0x00000001)

```

```
#define SDRCON_CLAT1 (0x00000000)
#define SDRCON_CLAT2 (0x00000002)
#define SDRCON_CLAT3 (0x00000004)
#define SDRCON_PIPE1 (0x00000008)
#define SDRCON_PG256 (0x00000000)
#define SDRCON_PG512 (0x00000010)
#define SDRCON_PG1K (0x00000020)
#define SDRCON_REF600 (0x00000000)
#define SDRCON_REF900 (0x00000080)
#define SDRCON_REF1200 (0x00000100)
#define SDRCON_REF2400 (0x00000180)
#define SDRCON_PC2RAS2 (0x00000000)
#define SDRCON_PC2RAS3 (0x00000200)
#define SDRCON_PC2RAS4 (0x00000400)
#define SDRCON_PC2RAS5 (0x00000600)
#define SDRCON_RAS2PC2 (0x00000000)
#define SDRCON_RAS2PC3 (0x00000800)
#define SDRCON_RAS2PC4 (0x00001000)
#define SDRCON_RAS2PC5 (0x00001800)
#define SDRCON_RAS2PC6 (0x00002000)
#define SDRCON_RAS2PC7 (0x00002800)
#define SDRCON_RAS2PC8 (0x00003000)
#define SDRCON_INIT (0x00004000)

/** Link Buffer Registers *****/
#define LBUFTX0_LOC (0x1804A0)
#define LBUFRX0_LOC (0x1804A4)
#define LBUFTX1_LOC (0x1804A8)
#define LBUFRX1_LOC (0x1804AC)
#define LBUFTX2_LOC (0x1804B0)
#define LBUFRX2_LOC (0x1804B4)
#define LBUFTX3_LOC (0x1804B8)
#define LBUFRX3_LOC (0x1804BC)
```

```

/**** Link Control Registers with Bit Masks ****
#define LCTL0_LOC (0x1804E0)
#define LCTL1_LOC (0x1804E1)
#define LCTL2_LOC (0x1804E2)
#define LCTL3_LOC (0x1804E3)

// Bit Masks
#define LCTL_DSBL (0x00000000)
#define LCTL_VER (0x00000004)
#define LCTL_DIV8 (0x00000000)
#define LCTL_DIV4 (0x00000008)
#define LCTL_DIV3 (0x00000010)
#define LCTL_DIV2 (0x00000018)
#define LCTL_LTEN (0x00000040)
#define LCTL_PSIZE (0x00000080)
#define LCTL_TTOE (0x00000100)
#define LCTL_CERE (0x00000200)
#define LCTL_LREN (0x00000400)
#define LCTL_RT0E (0x00000800)

/**** Link Status Registers with Bit Masks ****
#define LSTAT0_LOC (0x1804E0)
#define LSTAT1_LOC (0x1804E1)
#define LSTAT2_LOC (0x1804E2)
#define LSTAT3_LOC (0x1804E3)
#define LSTATC0_LOC (0x1804E0)
#define LSTATC1_LOC (0x1804E1)
#define LSTATC2_LOC (0x1804E2)
#define LSTATC3_LOC (0x1804E3)

// Bit Masks
#define LSTAT_RER (0x00000003)
#define LSTAT_RST (0x0000000C)
#define LSTAT_TER (0x00000030)

```

```
#define LSTAT_TST (0x000000C0)

//***** BTB Registers ***

// Tags - Way 0
#define BTB_WAY0_TG0_LOC (0x180600)
#define BTB_WAY0_TG1_LOC (0x180601)
#define BTB_WAY0_TG2_LOC (0x180602)
#define BTB_WAY0_TG3_LOC (0x180603)
#define BTB_WAY0_TG4_LOC (0x180604)
#define BTB_WAY0_TG5_LOC (0x180605)
#define BTB_WAY0_TG6_LOC (0x180606)
#define BTB_WAY0_TG7_LOC (0x180607)
#define BTB_WAY0_TG8_LOC (0x180608)
#define BTB_WAY0_TG9_LOC (0x180609)
#define BTB_WAY0_TG10_LOC (0x18060A)
#define BTB_WAY0_TG11_LOC (0x18060B)
#define BTB_WAY0_TG12_LOC (0x18060C)
#define BTB_WAY0_TG13_LOC (0x18060D)
#define BTB_WAY0_TG14_LOC (0x18060E)
#define BTB_WAY0_TG15_LOC (0x18060F)
#define BTB_WAY0_TG16_LOC (0x180610)
#define BTB_WAY0_TG17_LOC (0x180611)
#define BTB_WAY0_TG18_LOC (0x180612)
#define BTB_WAY0_TG19_LOC (0x180613)
#define BTB_WAY0_TG20_LOC (0x180614)
#define BTB_WAY0_TG21_LOC (0x180615)
#define BTB_WAY0_TG22_LOC (0x180616)
#define BTB_WAY0_TG23_LOC (0x180617)
#define BTB_WAY0_TG24_LOC (0x180618)
#define BTB_WAY0_TG25_LOC (0x180619)
#define BTB_WAY0_TG26_LOC (0x18061A)
#define BTB_WAY0_TG27_LOC (0x18061B)
#define BTB_WAY0_TG28_LOC (0x18061C)
```

```

#define BTB_WAY0_TG29_LOC (0x18061D)
#define BTB_WAY0_TG30_LOC (0x18061E)
#define BTB_WAY0_TG31_LOC (0x18061F)

// Tags - Way 1
#define BTB_WAY1_TG0_LOC (0x180620)
#define BTB_WAY1_TG1_LOC (0x180621)
#define BTB_WAY1_TG2_LOC (0x180622)
#define BTB_WAY1_TG3_LOC (0x180623)
#define BTB_WAY1_TG4_LOC (0x180624)
#define BTB_WAY1_TG5_LOC (0x180625)
#define BTB_WAY1_TG6_LOC (0x180626)
#define BTB_WAY1_TG7_LOC (0x180627)
#define BTB_WAY1_TG8_LOC (0x180628)
#define BTB_WAY1_TG9_LOC (0x180629)
#define BTB_WAY1_TG10_LOC (0x18062A)
#define BTB_WAY1_TG11_LOC (0x18062B)
#define BTB_WAY1_TG12_LOC (0x18062C)
#define BTB_WAY1_TG13_LOC (0x18062D)
#define BTB_WAY1_TG14_LOC (0x18062E)
#define BTB_WAY1_TG15_LOC (0x18062F)
#define BTB_WAY1_TG16_LOC (0x180630)
#define BTB_WAY1_TG17_LOC (0x180631)
#define BTB_WAY1_TG18_LOC (0x180632)
#define BTB_WAY1_TG19_LOC (0x180633)
#define BTB_WAY1_TG20_LOC (0x180634)
#define BTB_WAY1_TG21_LOC (0x180635)
#define BTB_WAY1_TG22_LOC (0x180636)
#define BTB_WAY1_TG23_LOC (0x180637)
#define BTB_WAY1_TG24_LOC (0x180638)
#define BTB_WAY1_TG25_LOC (0x180639)
#define BTB_WAY1_TG26_LOC (0x18063A)
#define BTB_WAY1_TG27_LOC (0x18063B)
#define BTB_WAY1_TG28_LOC (0x18063C)

```



```
#define BTB_WAY1_TG29_LOC (0x18063D)
#define BTB_WAY1_TG30_LOC (0x18063E)
#define BTB_WAY1_TG31_LOC (0x18063F)

// Tags - Way 2
#define BTB_WAY2_TG0_LOC (0x180640)
#define BTB_WAY2_TG1_LOC (0x180641)
#define BTB_WAY2_TG2_LOC (0x180642)
#define BTB_WAY2_TG3_LOC (0x180643)
#define BTB_WAY2_TG4_LOC (0x180644)
#define BTB_WAY2_TG5_LOC (0x180645)
#define BTB_WAY2_TG6_LOC (0x180646)
#define BTB_WAY2_TG7_LOC (0x180647)
#define BTB_WAY2_TG8_LOC (0x180648)
#define BTB_WAY2_TG9_LOC (0x180649)
#define BTB_WAY2_TG10_LOC (0x18064A)
#define BTB_WAY2_TG11_LOC (0x18064B)
#define BTB_WAY2_TG12_LOC (0x18064C)
#define BTB_WAY2_TG13_LOC (0x18064D)
#define BTB_WAY2_TG14_LOC (0x18064E)
#define BTB_WAY2_TG15_LOC (0x18064F)
#define BTB_WAY2_TG16_LOC (0x180650)
#define BTB_WAY2_TG17_LOC (0x180651)
#define BTB_WAY2_TG18_LOC (0x180652)
#define BTB_WAY2_TG19_LOC (0x180653)
#define BTB_WAY2_TG20_LOC (0x180654)
#define BTB_WAY2_TG21_LOC (0x180655)
#define BTB_WAY2_TG22_LOC (0x180656)
#define BTB_WAY2_TG23_LOC (0x180657)
#define BTB_WAY2_TG24_LOC (0x180658)
#define BTB_WAY2_TG25_LOC (0x180659)
#define BTB_WAY2_TG26_LOC (0x18065A)
#define BTB_WAY2_TG27_LOC (0x18065B)
#define BTB_WAY2_TG28_LOC (0x18065C)
```

```
#define BTB_WAY2_TG29_LOC (0x18065D)
#define BTB_WAY2_TG30_LOC (0x18065E)
#define BTB_WAY2_TG31_LOC (0x18065F)

// Tags - Way 3
#define BTB_WAY3_TG0_LOC (0x180660)
#define BTB_WAY3_TG1_LOC (0x180661)
#define BTB_WAY3_TG2_LOC (0x180662)
#define BTB_WAY3_TG3_LOC (0x180663)
#define BTB_WAY3_TG4_LOC (0x180664)
#define BTB_WAY3_TG5_LOC (0x180665)
#define BTB_WAY3_TG6_LOC (0x180666)
#define BTB_WAY3_TG7_LOC (0x180667)
#define BTB_WAY3_TG8_LOC (0x180668)
#define BTB_WAY3_TG9_LOC (0x180669)
#define BTB_WAY3_TG10_LOC (0x18066A)
#define BTB_WAY3_TG11_LOC (0x18066B)
#define BTB_WAY3_TG12_LOC (0x18066C)
#define BTB_WAY3_TG13_LOC (0x18066D)
#define BTB_WAY3_TG14_LOC (0x18066E)
#define BTB_WAY3_TG15_LOC (0x18066F)
#define BTB_WAY3_TG16_LOC (0x180670)
#define BTB_WAY3_TG17_LOC (0x180671)
#define BTB_WAY3_TG18_LOC (0x180672)
#define BTB_WAY3_TG19_LOC (0x180673)
#define BTB_WAY3_TG20_LOC (0x180674)
#define BTB_WAY3_TG21_LOC (0x180675)
#define BTB_WAY3_TG22_LOC (0x180676)
#define BTB_WAY3_TG23_LOC (0x180677)
#define BTB_WAY3_TG24_LOC (0x180678)
#define BTB_WAY3_TG25_LOC (0x180679)
#define BTB_WAY3_TG26_LOC (0x18067A)
#define BTB_WAY3_TG27_LOC (0x18067B)
#define BTB_WAY3_TG28_LOC (0x18067C)
```

```
#define BTB_WAY3_TG29_LOC (0x18067D)
#define BTB_WAY3_TG30_LOC (0x18067E)
#define BTB_WAY3_TG31_LOC (0x18067F)

// Targets - Way 0
#define BTB_WAY0_TR0_LOC (0x180680)
#define BTB_WAY0_TR1_LOC (0x180681)
#define BTB_WAY0_TR2_LOC (0x180682)
#define BTB_WAY0_TR3_LOC (0x180683)
#define BTB_WAY0_TR4_LOC (0x180684)
#define BTB_WAY0_TR5_LOC (0x180685)
#define BTB_WAY0_TR6_LOC (0x180686)
#define BTB_WAY0_TR7_LOC (0x180687)
#define BTB_WAY0_TR8_LOC (0x180688)
#define BTB_WAY0_TR9_LOC (0x180689)
#define BTB_WAY0_TR10_LOC (0x18068A)
#define BTB_WAY0_TR11_LOC (0x18068B)
#define BTB_WAY0_TR12_LOC (0x18068C)
#define BTB_WAY0_TR13_LOC (0x18068D)
#define BTB_WAY0_TR14_LOC (0x18068E)
#define BTB_WAY0_TR15_LOC (0x18068F)
#define BTB_WAY0_TR16_LOC (0x180690)
#define BTB_WAY0_TR17_LOC (0x180691)
#define BTB_WAY0_TR18_LOC (0x180692)
#define BTB_WAY0_TR19_LOC (0x180693)
#define BTB_WAY0_TR20_LOC (0x180694)
#define BTB_WAY0_TR21_LOC (0x180695)
#define BTB_WAY0_TR22_LOC (0x180696)
#define BTB_WAY0_TR23_LOC (0x180697)
#define BTB_WAY0_TR24_LOC (0x180698)
#define BTB_WAY0_TR25_LOC (0x180699)
#define BTB_WAY0_TR26_LOC (0x18069A)
#define BTB_WAY0_TR27_LOC (0x18069B)
#define BTB_WAY0_TR28_LOC (0x18069C)
```

```

#define BTB_WAY0_TR29_LOC (0x18069D)
#define BTB_WAY0_TR30_LOC (0x18069E)
#define BTB_WAY0_TR31_LOC (0x18069F)

// Targets - Way 1
#define BTB_WAY1_TR0_LOC (0x1806A0)
#define BTB_WAY1_TR1_LOC (0x1806A1)
#define BTB_WAY1_TR2_LOC (0x1806A2)
#define BTB_WAY1_TR3_LOC (0x1806A3)
#define BTB_WAY1_TR4_LOC (0x1806A4)
#define BTB_WAY1_TR5_LOC (0x1806A5)
#define BTB_WAY1_TR6_LOC (0x1806A6)
#define BTB_WAY1_TR7_LOC (0x1806A7)
#define BTB_WAY1_TR8_LOC (0x1806A8)
#define BTB_WAY1_TR9_LOC (0x1806A9)
#define BTB_WAY1_TR10_LOC (0x1806AA)
#define BTB_WAY1_TR11_LOC (0x1806AB)
#define BTB_WAY1_TR12_LOC (0x1806AC)
#define BTB_WAY1_TR13_LOC (0x1806AD)
#define BTB_WAY1_TR14_LOC (0x1806AE)
#define BTB_WAY1_TR15_LOC (0x1806AF)
#define BTB_WAY1_TR16_LOC (0x1806B0)
#define BTB_WAY1_TR17_LOC (0x1806B1)
#define BTB_WAY1_TR18_LOC (0x1806B2)
#define BTB_WAY1_TR19_LOC (0x1806B3)
#define BTB_WAY1_TR20_LOC (0x1806B4)
#define BTB_WAY1_TR21_LOC (0x1806B5)
#define BTB_WAY1_TR22_LOC (0x1806B6)
#define BTB_WAY1_TR23_LOC (0x1806B7)
#define BTB_WAY1_TR24_LOC (0x1806B8)
#define BTB_WAY1_TR25_LOC (0x1806B9)
#define BTB_WAY1_TR26_LOC (0x1806BA)
#define BTB_WAY1_TR27_LOC (0x1806BB)
#define BTB_WAY1_TR28_LOC (0x1806BC)

```

```
#define BTB_WAY1_TR29_LOC (0x1806BD)
#define BTB_WAY1_TR30_LOC (0x1806BE)
#define BTB_WAY1_TR31_LOC (0x1806BF)

// Targets - Way 2
#define BTB_WAY2_TR0_LOC (0x1806C0)
#define BTB_WAY2_TR1_LOC (0x1806C1)
#define BTB_WAY2_TR2_LOC (0x1806C2)
#define BTB_WAY2_TR3_LOC (0x1806C3)
#define BTB_WAY2_TR4_LOC (0x1806C4)
#define BTB_WAY2_TR5_LOC (0x1806C5)
#define BTB_WAY2_TR6_LOC (0x1806C6)
#define BTB_WAY2_TR7_LOC (0x1806C7)
#define BTB_WAY2_TR8_LOC (0x1806C8)
#define BTB_WAY2_TR9_LOC (0x1806C9)
#define BTB_WAY2_TR10_LOC (0x1806CA)
#define BTB_WAY2_TR11_LOC (0x1806CB)
#define BTB_WAY2_TR12_LOC (0x1806CC)
#define BTB_WAY2_TR13_LOC (0x1806CD)
#define BTB_WAY2_TR14_LOC (0x1806CE)
#define BTB_WAY2_TR15_LOC (0x1806CF)
#define BTB_WAY2_TR16_LOC (0x1806D0)
#define BTB_WAY2_TR17_LOC (0x1806D1)
#define BTB_WAY2_TR18_LOC (0x1806D2)
#define BTB_WAY2_TR19_LOC (0x1806D3)
#define BTB_WAY2_TR20_LOC (0x1806D4)
#define BTB_WAY2_TR21_LOC (0x1806D5)
#define BTB_WAY2_TR22_LOC (0x1806D6)
#define BTB_WAY2_TR23_LOC (0x1806D7)
#define BTB_WAY2_TR24_LOC (0x1806D8)
#define BTB_WAY2_TR25_LOC (0x1806D9)
#define BTB_WAY2_TR26_LOC (0x1806DA)
#define BTB_WAY2_TR27_LOC (0x1806DB)
#define BTB_WAY2_TR28_LOC (0x1806DC)
```

```
#define BTB_WAY2_TR29_LOC (0x1806DD)
#define BTB_WAY2_TR30_LOC (0x1806DE)
#define BTB_WAY2_TR31_LOC (0x1806DF)

// Targets - Way 3
#define BTB_WAY3_TR0_LOC (0x1806E0)
#define BTB_WAY3_TR1_LOC (0x1806E1)
#define BTB_WAY3_TR2_LOC (0x1806E2)
#define BTB_WAY3_TR3_LOC (0x1806E3)
#define BTB_WAY3_TR4_LOC (0x1806E4)
#define BTB_WAY3_TR5_LOC (0x1806E5)
#define BTB_WAY3_TR6_LOC (0x1806E6)
#define BTB_WAY3_TR7_LOC (0x1806E7)
#define BTB_WAY3_TR8_LOC (0x1806E8)
#define BTB_WAY3_TR9_LOC (0x1806E9)
#define BTB_WAY3_TR10_LOC (0x1806EA)
#define BTB_WAY3_TR11_LOC (0x1806EB)
#define BTB_WAY3_TR12_LOC (0x1806EC)
#define BTB_WAY3_TR13_LOC (0x1806ED)
#define BTB_WAY3_TR14_LOC (0x1806EE)
#define BTB_WAY3_TR15_LOC (0x1806EF)
#define BTB_WAY3_TR16_LOC (0x1806F0)
#define BTB_WAY3_TR17_LOC (0x1806F1)
#define BTB_WAY3_TR18_LOC (0x1806F2)
#define BTB_WAY3_TR19_LOC (0x1806F3)
#define BTB_WAY3_TR20_LOC (0x1806F4)
#define BTB_WAY3_TR21_LOC (0x1806F5)
#define BTB_WAY3_TR22_LOC (0x1806F6)
#define BTB_WAY3_TR23_LOC (0x1806F7)
#define BTB_WAY3_TR24_LOC (0x1806F8)
#define BTB_WAY3_TR25_LOC (0x1806F9)
#define BTB_WAY3_TR26_LOC (0x1806FA)
#define BTB_WAY3_TR27_LOC (0x1806FB)
#define BTB_WAY3_TR28_LOC (0x1806FC)
```

```
#define BTB_WAY3_TR29_LOC (0x1806FD)
#define BTB_WAY3_TR30_LOC (0x1806FE)
#define BTB_WAY3_TR31_LOC (0x1806FF)

//**** Interrupt Vectors ****

#define IVTIMER0LP_LOC (0x180702)
#define IVTIMER1LP_LOC (0x180703)
#define IVLINK0_LOC (0x180706)
#define IVLINK1_LOC (0x180707)
#define IVLINK2_LOC (0x180708)
#define IVLINK3_LOC (0x180709)
#define IVDMA0_LOC (0x18070E)
#define IVDMA1_LOC (0x18070F)
#define IVDMA2_LOC (0x180710)
#define IVDMA3_LOC (0x180711)
#define IVDMA4_LOC (0x180716)
#define IVDMA5_LOC (0x180717)
#define IVDMA6_LOC (0x180718)
#define IVDMA7_LOC (0x180719)
#define IVDMA8_LOC (0x18071D)
#define IVDMA9_LOC (0x18071E)
#define IVDMA10_LOC (0x18071F)
#define IVDMA11_LOC (0x180720)
#define IVDMA12_LOC (0x180725)
#define IVDMA13_LOC (0x180726)
#define IVIRQ0_LOC (0x180729)
#define IVIRQ1_LOC (0x18072A)
#define IVIRQ2_LOC (0x18072B)
#define IVIRQ3_LOC (0x18072C)
#define VIRPT_LOC (0x180730)
#define IVBUSLK_LOC (0x180732)
#define IVTIMER0HP_LOC (0x180734)
#define IVTIMER1HP_LOC (0x180735)
```

```

#define IVHW_LOC (0x180739)
#define IVSW_LOC (0x18073E)

//****

#define AUTODMA0_LOC (0x180740)
#define AUTODMA1_LOC (0x180744)

//**** Watchpoint Registers ****

#define WPOCTL_LOC (0x1807A0)
#define WP1CTL_LOC (0x1807A1)
#define WP2CTL_LOC (0x1807A2)

// Bit Masks
// OPMODE
#define WPCTL_DSBL (0x00000000)
#define WPCTL_ADDRESS (0x00000001)
#define WPCTL_RANGE (0x00000002)
#define WPCTL_NOTRANGE (0x00000003)
// BM
#define WPCTL_SEQ (0x00000004)
#define WPCTL_JALU (0x00000008)
#define WPCTL_KALU (0x00000010)
#define WPCTL_DMAI (0x00000020)
#define WPCTL_BIU (0x00000040)
#define WPCTL_SEQFETCH (0x00000080)
// R/W
#define WPCTL_READ (0x00000100)
#define WPCTL_WRITE (0x00000200)
// EXTYPE
#define WPCTL_NOEXCEPT (0x00000000)
#define WPCTL_EXCEPT (0x00000400)
#define WPCTL_EMUTRAP (0x00000800)

```



```
// SSTP,WPOR,WPAND
#define WPCTL_SSTP (0x00001000)
#define WPCTL_WPOR (0x00001000)
#define WPCTL_WPAND (0x00001000)

#define WPOSTAT_LOC (0x1807A4)
#define WP1STAT_LOC (0x1807A5)
#define WP2STAT_LOC (0x1807A6)

// Bit positions (of the masks)
#define WPSTAT_VALUE_P (0)
#define WPSTAT_EX_P (16)

// Bit masks
#define WPSTAT_VALUE (0x0000FFFF)
#define WPSTAT_EX (0x00030000)

#define WP0L_LOC (0x1807A8)
#define WP0H_LOC (0x1807A9)
#define WP1L_LOC (0x1807AA)
#define WP1H_LOC (0x1807AB)
#define WP2L_LOC (0x1807AC)
#define WP3H_LOC (0x1807AD)

/** Trace Buffer Registers *****/

#define TRCB0_LOC (0x1807C0)
#define TRCB1_LOC (0x1807C1)
#define TRCB2_LOC (0x1807C2)
#define TRCB3_LOC (0x1807C3)
#define TRCB4_LOC (0x1807C4)
#define TRCB5_LOC (0x1807C5)
#define TRCB6_LOC (0x1807C6)
```

```

#define TRCB7_LOC (0x1807C7)

//**** Global memory ****

#define BLOCK0_LOC (0x00000000) // Internal memory block 0
#define BLOCK1_LOC (0x00080000) // Internal memory block 1
#define BLOCK2_LOC (0x00100000) // Internal memory block 2

#define P0_OFFSET_LOC (0x02000000) // Processor ID0 MP memory
offset
#define P1_OFFSET_LOC (0x02400000) // Processor ID1 MP memory
offset
#define P2_OFFSET_LOC (0x02800000) // Processor ID2 MP memory
offset
#define P3_OFFSET_LOC (0x02C00000) // Processor ID3 MP memory
offset
#define P4_OFFSET_LOC (0x03000000) // Processor ID4 MP memory
offset
#define P5_OFFSET_LOC (0x03400000) // Processor ID5 MP memory
offset
#define P6_OFFSET_LOC (0x03800000) // Processor ID6 MP memory
offset
#define P7_OFFSET_LOC (0x03C00000) // Processor ID7 MP memory
offset

#endif /* !defined(__DEFTS101_H_) */

```

C INSTRUCTION DECODE

This chapter identifies operation codes (opcodes) for instructions. Use this chapter to learn how to construct opcodes.

Instruction Structure

TigerSHARC processor instructions are all 32-bit words, where the upper bits are identical to all instructions as shown in [Figure C-1](#).

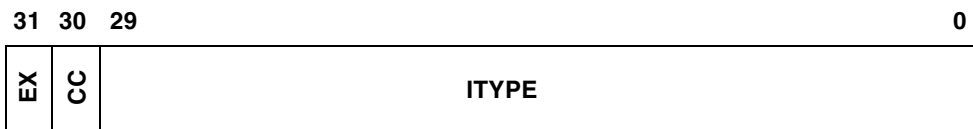


Figure C-1. Instruction Structure

The ITYPE field determines the execution group to which the instruction belongs. Its length varies according to the type of instruction. The format is strictly set by the two MSB's (bits[29-28]). The decoding for the ITYPE field appears in [Table C-1](#).

Instruction Structure

Table C-1. ITYPE Field

Bit	Field	Description
29–28	0<JK>	Integer ALU operation (calculation and/or Bits[29:28] load/store). The <JK> field in IALU instructions determines if the instruction refers to J or K IALU, and decodes as follows: 0 = J-IALU 1 = K-IALU
29–26	10<XY>	Compute block ALU instruction. The <XY> field in compute block instructions is a two bit field that determines if the instruction is targeted at the X compute block, the Y compute block, or both. It decodes as follows: 00 = Reserved for future use 01 = Compute block X 10 = Compute block Y 11 = Both compute block X and Y
29–25	10<XY>0	Compute block ALU instruction.
29–24	10<XY>10	Compute block shifter instruction.
29–24	10<XY>11	Compute block multiplier instruction.
29–26	1100	Control Flow instructions, Immediate Extension and others.
31	EX	When the EX bit is set, determines the instruction as the last in the instruction line.
30	CC	In most instructions, bit 30 is the CC bit. When set, conditions the execution of the instruction by the condition instruction in the instruction line. This applies to compute block, IALU and load/store instructions. This does not apply to conditional instructions, immediate extensions and others. (see “Sequencer Indirect Jump Instruction Format” on page C-36 and “Sequencer Flow Control Instructions” on page C-33).

Compute Block Instruction Format

The instruction format for all compute block instructions is as shown in [Figure C-2](#) and [Table C-2](#).

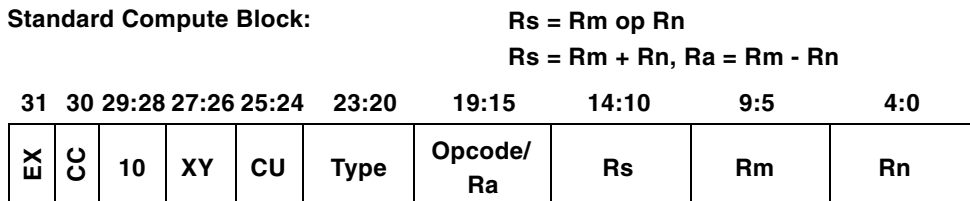


Figure C-2. Compute Block Instruction Format

Table C-2. Compute Block Instruction Opcode Fields

Bits	Field	Description
4–0	RN	Determines the second operand in the instruction.
9–5	RM	Determines the first operand in the instruction.
14–10	RS	Determines the result registers.
19–15	OPCODE/R A	See Table C-5 on page C-10 , Table C-7 on page C-16 , Table C-8 on page C-18 , Table C-9 on page C-20 , Table C-10 on page C-21 , Table C-11 on page C-22 , Table C-22 on page C-39 , and Table C-23 on page C-40 .
23–20	TYPE	See Table C-5 on page C-10 , Table C-7 on page C-16 , Table C-8 on page C-18 , Table C-9 on page C-20 , Table C-10 on page C-21 , Table C-11 on page C-22 , Table C-22 on page C-39 , and Table C-23 on page C-40 .
25–24	CU	Determines the type of compute block instruction: 00 = ALU fixed-point instruction 01 = ALU fixed and floating-point instruction 10 = Shifter instruction 11 = Multiplier instruction

Compute Block Instruction Format

Table C-2. Compute Block Instruction Opcode Fields (Cont'd)

Bits	Field	Description
27–26	XY	Discerns which compute block is to execute the operation: 00 = Reserved 01 = Sets compute block X as the executing unit 10 = Sets compute block Y as the executing unit 11 = Sets both compute blocks as the executing units
29–28	10	Determine that this is an operation to be executed by the compute blocks.
30	CC	Condition bit. When set, specifies the execution of the instruction by the condition instruction in this line.
31	EX	When set, determines the instruction as the last in the instruction line.

ALU Instructions

ALU instruction syntax and opcodes are covered in the following sections:

- “ALU Fixed-Point, Arithmetic and Logical Instructions (CU=00)” on page C-5
- “ALU Fixed-Point, Data Conversion Instructions (CU=01)” on page C-7
- “ALU Floating-Point, Arithmetic and Logical Instructions (CU=01)” on page C-10
- “CLU Instructions” on page C-12

ALU Fixed-Point, Arithmetic and Logical Instructions (CU=00)

Table C-3 lists the syntax, type-codes, and opcodes for ALU fixed point instructions. These instructions have the same data type and data size for operands and results.

Table C-3. ALU Fixed-Point, Arithmetic and Logical Instruction Syntax and Opcodes

Syntax	Type	Opcode	
$(B/S/L)Rs(d) = Rm + Rn$	0000-Rs 0011-(L)Rsd ¹ 0110-SRs 1001-BRs 1100-B/SRs ²	000	S1 S0 ³
$(B/S/L)Rs(d) = Rm - Rn$		001	S1 S0 ³
$(B/S/L)Rs(d) = ABS (Rm + Rn)$		010	0 X ⁴
$(B/S/L)Rs(d) = ABS (Rm - Rn)$		011	U X
$(B/S/L)Rs(d) = (Rm + Rn) / 2$		100	U T ⁵
$(B/S/L)Rs(d) = (Rm - Rn) / 2$		101	U T ⁵
$(B/S/L)Rs(d) = MAX (Rm, Rn)$		110	U Z ⁶
$(B/S/L)Rs(d) = MIN (Rm, Rn)$		111	U Z ⁶

Compute Block Instruction Format

Table C-3. ALU Fixed-Point, Arithmetic and Logical Instruction Syntax and Opcodes (Cont'd)

Syntax	Type	Opcode	
$(L)Rs(d) = Rm + Rn + CI^7$	0001-Rs 0100-(L)Rsd ¹ 0111-SRs 1010-BRs 1101-B/SRsd ²	000	S1 S0 ³
$(L)Rs(d) = Rm - Rn + CI - 1^7$		001	S1 S0 ³
$(B/S/L)Rs(d) = INC Rm$		010	S1 S0 ³
$(B/S/L)Rs(d) = DEC Rm$		011	S1 S0 ³
$(B/S/L)COMP (Rm, Rn) — signed$		100	00
$(B/S/L)COMP (Rm, Rn) — unsigned$		100	10
$(B/S/L)Rs = CLIP Rm \text{ by } Rn$		101	00
$(L)Rs(d) = PASS Rm^7$		101	01
$(B/S/L)Rs(d) = ABS Rm$		101	10
$(B/S/L)Rs(d) = -Rm$		101	11
$(L)Rs(d) = Rm \text{ AND } Rn^7$		110	00
$(L)Rs(d) = Rm \text{ OR } Rn^7$		110	01
$(L)Rs(d) = Rm \text{ XOR } Rn^7$		110	10
$(L)Rs(d) = NOT Rm^7$		110	11
$S/BRsd = VMAX (Rmd, Rnd)^8$		110	00
$S/BRsd = VMIN (Rmd, Rnd)^8$		111	00
$(L)Rs(d) = Rm + CI^7$		111	00
$(L)Rs(d) = Rm + CI - 1^7$		111	01
$(L)Rs(d) = Rm \text{ AND NOT } Rn^7$		111	11
$(B/S/L)Rs(d) = Rm(d) + Rn(d),$ $Ra(d) = Rm(d) - Rn(d)$		0010-Rs 0101-(L)Rsd ¹ 1000-SRs 1011-BRs 1110-B/SRsd ²	-Ra-

- 1 The LSB of Rn (bit 0 of the instruction) is used to decode operand size, where:
0 = determines dual normal
1 = determines long word
- 2 The LSB of Rn (bit 0 of the instruction) is used to decode operand size, where:
0 = determines octal byte
1 = determines quad short
- 3 S1 S0 values are: no saturation () – 00, saturation signed (S) – 01,
saturation unsigned (SU) – 11
- 4 Option X - extend for ABS
- 5 On instruction Rs = (Rm +/- Rn)/2 the decode of (TU) is as follows (Bits[16:15], U T):
11 = Unsigned, truncate (TU)
10 = Unsigned, round to nearest even (U)
01 = Signed, truncate (T)
00 = Signed, round to nearest even ()
- 6 On instruction MIN and MAX, options are: ()=00, (U)=10, (Z)=01 and (UZ)=11
- 7 Instruction is implemented only for single normal or long.
- 8 The VMIN and VMAX instructions use the same opcode as other instructions, but only for short or byte, while the other instructions are implemented in word or long only.

ALU Fixed-Point, Data Conversion Instructions (CU=01)

Table C-4 lists syntax, type-codes, and opcodes for ALU fixed-point instructions with short, byte, and miscellaneous operands. Note that the CU field is set to 01.

Table C-4. ALU Fixed-Point, Data Conversion Instruction Syntax and Opcodes

Syntax	CU = 01 Fixed	Type	Opcode	
Rsd = EXPAND SRm + SRn		0000	000	T1 T0 ¹
Rsqr = EXPAND SRmd +/- SRnd ¹		0000	001	T1 T0 ¹
SRsd = EXPAND BRm + BRn		0000	010	T1 T0 ¹
SRsq = EXPAND BRmd +/- BRnd ¹		0000	011	T1 T0 ¹
Rsd = EXPAND SRm - SRn		0000	100	T1 T0 ¹
SRsd = EXPAND BRm - BRn		0000	110	T1 T0 ¹
Rsd = EXPAND SRm	Rn = 00000	0000	111	T1 T0 ¹

Compute Block Instruction Format

Table C-4. ALU Fixed-Point, Data Conversion Instruction Syntax and Opcodes (Cont'd)

Syntax	CU = 01 Fixed	Type	Opcode	
$R_{sq} = \text{EXPAND } SR_{md}$	$R_n = 00001$	0000	111	$T_1 T_0^1$
$SR_{sd} = \text{EXPAND } BR_{md}$	$R_n = 00010$	0000	111	$T_1 T_0^1$
$SR_{sq} = \text{EXPAND } BR_{md}$	$R_n = 00011$	0000	111	$T_1 T_0^1$
$SR_s = \text{COMPACT } R_{md} +/- R_{nd}^2$		0001	00	$C_2 C_1 C_0^2$
$BR_s = \text{COMPACT } SR_{md} +/- SR_{nd}^2$		0001	01	$C_2 C_1 C_0^2$
$SR_s = \text{COMPACT } R_{md}$		0001	10	$C_2 C_1 C_0^2$
$BR_s = \text{COMPACT } SR_{md}$		0001	11	$C_2 C_1 C_0^2$
$BR_{sd} = \text{MERGE } R_m, R_n$		0010	000	00
$BR_{sq} = \text{MERGE } R_{md}, R_{nd}$		0010	000	01
$SR_{sd} = \text{MERGE } R_m, R_n$		0010	000	10
$SR_{sq} = \text{MERGE } R_{md}, R_{nd}$		0010	000	11
$R_s = \text{SUM } SR_m^3$	$R_n=00000$	0010	001	$S_0 0^4$
$R_s = \text{SUM } SR_{md}^3$	$R_n=00001$	0010	001	$S_0 0^4$
$R_s = \text{SUM } BR_m^3$	$R_n=00010$	0010	001	$S_0 0^4$
$R_s = \text{SUM } BR_{md}^3$	$R_n=00011$	0010	001	$S_0 0^4$
$R_s = \text{ONES } R_m$	$R_n=00100$	0010	001	00
$R_s = \text{ONES } R_{md}$	$R_n=00101$	0010	001	00
$R_{sd} = PR1:0$	$R_n=00110$	0010	001	00
$R_s = \text{BFOINC } R_{md}$		0010	010	00
$PRO += \text{ABS } (SR_{md} - SR_{nd})$		0011	000	$S_0 0^4$
$PRO += \text{ABS } (BR_{md} - BR_{nd})$		0011	001	$S_0 0^4$
$PR1 += \text{ABS } (SR_{md} - SR_{nd})$		0011	000	$S_0 0^4$
$PR1 += \text{ABS } (BR_{md} - BR_{nd})$		0011	001	$S_0 0^4$
$PRO += \text{SUM } SR_m$	$R_n=00000$	0011	010	$S_0 0^4$

Table C-4. ALU Fixed-Point, Data Conversion Instruction Syntax and Opcodes (Cont'd)

Syntax	CU = 01 Fixed	Type	Opcode	
PRO += SUM SRmd	Rn=00001	0011	010	S0 0 ⁴
PRO += SUM BRm	Rn=00010	0011	010	S0 0 ⁴
PRO += SUM BRmd	Rn=00011	0011	010	S0 0 ⁴
PR1 += SUM SRm	Rn=00100	0011	010	S0 0 ⁴
PR1 += SUM SRmd	Rn=00101	0011	010	S0 0 ⁴
PR1 += SUM BRm	Rn=00110	0011	010	S0 0 ⁴
PR1 += SUM BRmd	Rn=00111	0011	010	S0 0 ⁴
PR1:0 = Rmd	Rn=01000	0011	010	00
Reserved	Rn>01000	0011	010	xx

- 1 The LSB of Rn in EXPAND & COMPACT is used to decode +/-, where: LSB = 0 determines addition, and LSB = 1 determines subtraction
- 2 Compact Coding—C2 C1 C0—is used to determine a combination of options that the instruction may incorporate. The ensuing combinations determine the following options:
 - 000 = Fractional round
 - 001 = Integer, no saturate (I)
 - 100 = Fractional, truncate (T)
 - 101 = Integer, saturate, signed (IS)
 - 111 = Integer, saturate, unsigned (ISU)
- 3 SUM is sideways summation—for example, BR5 = SUM R1:0 adds the eight bytes in double register R1:0 and stores the results in R5. The PR0/1 SUMs accumulate the results in the PR registers, which are two accumulation registers primarily used for block matching.
- 4 S0 is 0 for signed, and 1 for unsigned.

Compute Block Instruction Format

ALU Floating-Point, Arithmetic and Logical Instructions (CU=01)

Table C-5 lists syntax, type-codes and opcodes for ALU floating-point instructions. Note that the CU field=01, and floating-point is distinguished by the type codes 0100, 0101, 0110 and 0111—as opposed to the fixed-point instructions listed in Table C-4, where the CU field is also set to 01 but the type codes are different.

Table C-5. ALU Floating-Point, Arithmetic Instruction Syntax and Opcodes

Syntax	(CU = 01 Float)	Type	Opcode
$FRs(d) = Rm(d) + Rn(d)$		0100	000 T d ^{1,2}
$FRs(d) = Rm(d) - Rn(d)$		0100	001 T d
$FRs(d) = (Rm(d) + Rn(d)) / 2$		0100	010 T d
$FRs(d) = (Rm(d) - Rn(d)) / 2$		0100	011 T d
$FRs(d) = ABS (Rm(d) + Rn(d))$		0100	100 T d
$FRs(d) = ABS (Rm(d) - Rn(d))$		0100	101 T d
$FRs(d) = FLOAT Rm \text{ by } Rn$		0100	110 T d
$Rs = FIX FRm(d) \text{ by } Rn$		0100	111 T d
$FRs(d) = CLIP Rm(d) \text{ by } Rn(d)$		0101	000 0 d
$FRs(d) = Rm(d) \text{ COPYSIGN } Rn(d)$		0101	000 1 d
$FRs(d) = SCALB FRm(d) \text{ by } Rn$		0101	001 0 d
$FRs(d) = FLOAT Rm$	$Rn = 00000$	0101	010 T d
$FRs(d) = ABS Rm(d)$	$Rn = 00001$	0101	010 0 d
$Rs = MANT FRm(d)$	$Rn = 00010$	0101	010 0 d
$FRs(d) = PASS Rm(d)$	$Rn = 00011$	0101	010 0 d
$FRs(d) = - Rm(d)$	$Rn = 00100$	0101	010 0 d
$FRs(d) = RECIPS Rm(d)$	$Rn = 00101$	0101	010 0 d

Table C-5. ALU Floating-Point, Arithmetic Instruction Syntax and Opcodes (Cont'd)

Syntax	(CU = 01 Float)	Type	Opcode
FRs(d) = RSQRTS Rm(d)	Rn = 00110	0101	010 0 d
Rs = FIX FRm(d)	Rn = 00111	0101	010 T d
Rs = LOGB FRm(d)	Rn = 01000	0101	010 S d ³
FRsd = EXTD Rm—extended prec output	Rn = 01001	0101	010 0 1
FRs = SNGL Rmd—single prec output	Rn = 01010	0101	010 T 0
FRs(d) = MAX (Rm(d), Rn(d))		0101	011 0 d
FRs(d) = MIN (Rm(d), Rn(d))		0101	011 1 d
FCOMP (Rm(d), Rn(d))		0101	100 0 d
FRs = Rm + Rn, FRa = Rm - Rn —always round		0110	–Ra–
FRsd = Rmd + Rnd, FRad = Rmd - Rnd—always round		0111	–Rad–

- 1 T: Round (T=0); Truncate (T=1) for all the floating point ALU instructions.
- 2 d: Extended precision format implied by operand size—for example, in the instruction FR1:0 = R3:2 + R5:4, d is set (d=1) to imply double register floating point using 40-bit extended precision format. In the instruction FR0 = R2 + R3, d is cleared (d=0) to imply normal single register 32-bit IEEE floating point. This applies for all floating point ALU instructions.
- 3 S is for saturation: 0 – no saturation, 1 – saturation is enabled.

Compute Block Instruction Format

CLU Instructions

The communications logic unit (CLU) instructions are compute block instructions, which look like ALU or shifter instructions. Bits 25:0 of the op-code is detailed in [Table C-6](#), while bits 31:26 are identical for all compute block instructions.

Table C-6. Communications Logic Unit (CLU) Instruction Syntax and Opcodes

Instruction	25–20	19/	18–17	16–15	14–11	10	9–5	4–3	2/1–0	
	CU/type	Op - Code								
$Rs(d)(q)=TRm(d)(q)$	01 1110		101	00		$Rs(d)(q)$	00000	NLQ	000	
$TRs(d)(q)=Rm(d)(q)$	10 1110		10, TRsd1	TRsd3–2		00000	$Rm(d)(q)$	NLQ ¹	0000	
$Rs(d)(q)=THRm(d)(q)$	01 1111		101	00		$Rs(d)(q)$	00000	NLQ ¹	THRm3–1	
$THRs(d)(q)=Rm(d)(q)(I)^1$	10 1111		10, THRsd1	THRsd3–2		00000	$Rm(d)$	NLQ ¹	i00	
(S)TRsd=MAX (TRmd+Rmq_h, TRnd+Rmq_l)	01 1110		11, Trsd1	TRsd3–2		$00S^2X^3$	TRnd3	Rmq	TRnd2–1	TRmd3–1
(S)TRsd=MAX (TRmd-Rmq_h, TRnd-Rmq_l)	01 1100		11, Trsd1	TRsd3–2		$00S^2X^4$	TRnd3	Rmq	TRnd2–1	TRmd3–1
(S)TRsd=TMAX (TRmd+Rmq_h, TRnd+Rmq_l)	01 1111		11, Trsd1	TRsd3–2		$00S^3X^4$	TRnd3	Rmq	TRnd2–1	TRmd3–1

Table C-6. Communications Logic Unit (CLU) Instruction Syntax and Opcodes (Cont'd)

Instruction	25-20	19/	18-17	16-15	14-11	10	9-5	4-3	2/1-0
	CU/type	Op - Code							
(S)TRsd= TMAX(TRmd- Rmq_h, TRnd-=Rmq_l)	01 1101	11, Trsd1		TRsd3-2	00S ³ X ⁴	TRnd3	Rmq	TRnd2-1	TRmd3-1
(S)Rs= TMAX(TRm, TRn)	01 1011	11, Trn0	TRn3, 0	Rs			00S ³ X ⁴ TRm0	TRn2-1	TRm3-1
(S)TRsq= ACS (TRmd, TRnd, Rm) (TMAX)	01 11S ³ T ⁴	100		TRsq3-2	000X ⁴	TRnd 3	Rm	TRnd2-1	TRmd3-1
Rsq=TRaq, (S)TRsq= ACS (TRmd, TRnd, Rm) (TMAX)	01 11S ³ T ⁵	0	TRaq3-2	TRsq3-2	Rsq4-2, X ⁴	TRnd3	Rm	TRnd2-1	TRmd3-1
TRs= DESPREAD (Rmq, THrd)+ TRn	01 1000	0	10	TRs3-2		TRn3	Rmq4-2, TRs 1-0	TRn2-0	THRD ⁵
Rs=TRs, TRs=DESPREAD (Rmq, THrd)	01 1000	0	00	TRs3-2	Rs		Rmq4-2, TRs1-0		THRD ⁵
Rsd=TRsd, TRsd=DESPREAD (Rmq, THrd)	01 1000	0	01	TRs3-2	Rsd		Rmq4-2, TRs1,0		THRD ⁵
Rsq=Permute (Rmd, -Rmd, Rn)	01 1100	1	01	00	Rsd		Rsq, Rmd		Rn
Rsd=Permute (Rmd, Rn)	01 1100	1	01	01	Rsd		Rmd	Rn	

Compute Block Instruction Format

Table C-6. Communications Logic Unit (CLU) Instruction Syntax and Opcodes (Cont'd)

Instruction	25–20	19/	18–17	16–15	14–11	10	9–5	4–3	2/1–0
	CU/type	Op - Code							
Rs=X/Ystat	According to existing definition of instruction - CLU refers only to bits 16–14								
X/Ystat=Rm	According to existing definition of instruction - CLU refers only to bits 16–14								

- 1 (i) is option “interleave”
- 2 S– Short operation (if clear) or regular 32-bit operation (if set).
- 3 X is “0”—the value “1” is reserved for non-saturation.
- 4 T is set when option (TMAX) is used, otherwise it is cleared.
- 5 THRD field can be only 00. Other values are reserved for future (when more THR registers are implemented).

Multiplier Instructions

In the multiplier the op-code is defined by the options. The options notation is as follows:

Bits ‘XY’ define the options (U) and (NU):

00: both operands signed – ()

10: both operands are unsigned – (U)

01: Rm signed and Rn unsigned – (nU)

Bit ‘U’ indicates unsigned (if set); default is signed (0).

Bit ‘I’ indicates integer (if set); default is fractional (0).

Bit ‘S’ indicates saturation (if set) or no saturation (if cleared).

Bit ‘T’ indicates truncate (if set) or round (if cleared). This bit is significant only when format is fractional.

Bits 'C' and 'R' for multiply-accumulate are as follows:

C=0, R=0 – normal multiply-accumulate

C=1, R=0 – Clear MR registers before multiply-accumulate

C=1, R=1 – Clear MR registers and set round bits before multiply-accumulate

Compute Block Instruction Format

Bits ‘ab’ for quad multiply-accumulate and transfer instruction are as follows:

$$Rsd=MR3:0, MR3:0+= Rmd * Rnd \Rightarrow ab=00$$

$$Rsd=MR3:2, MR3:2+= Rmd * Rnd \Rightarrow ab=01$$

$$Rsd=MR1:0, MR1:0+= Rmd * Rnd \Rightarrow ab=10$$

Table C-7 summarizes the syntax, type-codes and opcodes for multiplier instructions.

Table C-7. Multiplier Instruction Syntax and Opcodes

Syntax		Type	Opcode
$Rs = Rm * Rn$		0000	x I y T S
$Rsd = Rm * Rn$		0001	x I 0 0 y
$Rsd = Rmd * Rnd$		0010	U I 0 T S
$Rs_q = Rmd * Rnd$		0011	U I 0 0 0
$Rs = MRa, MRa += Rm * Rn$	a = 1 MR1:0 a = 0 MR3:2	0100	U I C a 1
$Rsd = MRa, MRa += Rm * Rn$		0101	U I C a 1
$Rs = MRa, MRa += Rm ** Rn$		1000	0 I C a 1
$Rsd = MRa, MRa += Rm ** Rn$		1001	0 I C a 1
$Rs = MRa, MRa += Rm ** Rn (J)$		1010	0 I C a 1
$Rsd = MRa, MRa += Rm ** Rn (J)$		1011	0 I C a 1
$Rsd = MRa, MRa += Rmd * Rnd.$		1100	U I C a b
$MR3:2 += Rm * Rn^1$	$Rs = 00000$	1101	U I C R 1
$MR1:0 += Rm * Rn$	$Rs = 00001$	1101	U I C R 1
$MR3:2 -= Rm * Rn$	$Rs = 00010$	1101	0 I C R 1
$MR1:0 -= Rm * Rn$	$Rs = 00011$	1101	0 I C R 1
$MR3:0 += Rmd * Rnd$	$Rs = 00100$	1101	U I C R 1

Table C-7. Multiplier Instruction Syntax and Opcodes (Cont'd)

Syntax		Type	Opcode
MR3:2 += Rmd * Rnd	Rs = 00101	1101	U I C 0 1
MR1:0 += Rmd * Rnd	Rs = 00110	1101	U I C 0 1
MR3:2+= Rm ** Rn	Rs = 00111	1101	0 I C R 1
MR1:0 += Rm ** Rn	Rs = 01000	1101	0 I C R 1
MR3:2+= Rm ** Rn (J)	Rs = 01001	1101	0 I C R 1
MR1:0 += Rm ** Rn (J)	Rs = 01010	1101	0 I C R 1
MR3:2 = Rmd	Rs = 01110	1101	0 0 0 0 0
MR1:0 = Rmd	Rs = 01111	1101	0 0 0 0 0
MR4 = Rm	Rs = 10000	1101	0 0 0 0 0
Rsd = MR3:2	Rm = 00000	1110	U 0 0 0 S
Rsd = MR1:0	Rm = 00001	1110	U 0 0 0 S
SRsd = MR3:2	Rm = 00010	1110	U 0 0 0 S
SRsd = MR1:0	Rm = 00011	1110	U 0 0 0 S
Rsq = MR3:0	Rm = 00100	1110	U 0 0 0 S
Rs = MR4	Rm = 00101	1110	0 0 0 0 0
Rs = COMPACT MR3:2	Rm = 00110	1110	U I 0 1 S
Rs = COMPACT MR1:0	Rm = 00111	1110	U I 0 1 S
SRsd = COMPACT MR3:0	Rm = 01000	1110	U I 0 1 S
FRs = Rm * Rn		1111	000 T 0
FRsd = Rmd * Rnd		1111	000 T 1

1 MR3:0 are four 32-bit accumulation registers. They overflow into R4, which stores two 16-bit overflows for 32-bit multiples, or four 8-bit overflows for quad 16-bit multiples.

Compute Block Instruction Format

Shifter Instructions

Shifter instruction syntax and opcodes are covered in the following sections:

- “Shifter Instructions Using Single Normal-Word Operands and Single Register” on page C-18
- “Shifter Instructions Using Single Long-Word or Dual Normal-Word Operands and Dual Register” on page C-19
- “Shifter Instructions Using Short or Byte Operands and Single or Dual Registers” on page C-20
- “Shifter Instructions Using Single Operand” on page C-22

Shifter Instructions Using Single Normal-Word Operands and Single Register

Table C-8 lists the syntax, type-codes and opcodes for shifter instructions with single, normal-word operands and single result registers.

Table C-8. Shifter Instruction Syntax and Opcodes (Single, Normal-Word Operands and Single Register)

Syntax	Type	Opcode		Comments
$R_s = \text{LSHIFT } R_n \text{ BY } R_m$	0000	000	xx	See ¹
$R_s = \text{LSHIFT } R_n \text{ BY } \langle \text{imm6} \rangle$	0000	001	$xi5^2$	
$R_s = \text{ASHIFT } R_n \text{ BY } R_m$	0000	010	xx	No options
$R_s = \text{ASHIFT } R_n \text{ BY } \langle \text{imm6} \rangle$	0000	011	$xi5^2$	
$R_s = \text{ROT } R_n \text{ BY } R_m$	0001	000	xx	No options
$R_s = \text{ROT } R_n \text{ BY } \langle \text{imm6} \rangle$	0001	001	$0i5^2$	
$R_s = \text{FEXT } R_n \text{ BY } R_m$	0001	010	x S0	S0=1 sign extend (SE)

Table C-8. Shifter Instruction Syntax and Opcodes
(Single, Normal-Word Operands and Single Register) (Cont'd)

Syntax	Type	Opcode		Comments
$R_s = \text{FEXT } R_n \text{ BY } R_{md}$	0001	011	x S0	
$R_s += \text{FDEP } R_n \text{ BY } R_m$	0010	000	S1 S0	S0=1 sign extend
$R_s += \text{FDEP } R_n \text{ BY } R_{md}$	0010	001	S1 S0	S1=1 zero fill ZF
$R_s += \text{MASK } R_n \text{ BY } R_m$	0010	010	xx	No options

- The following bits in R_m are used as the shift magnitude for the operation:
 Byte: [4:0]
 Short: [5:0]
 Word: [6:0]
 Long: [7:0]
- LSB of opcode field is bit 5 of the six bits immediate.

Shifter Instructions Using Single Long-Word or Dual Normal-Word Operands and Dual Register

Table C-9 lists the syntax, type-codes and opcodes for shifter instructions with single long-word or dual normal-word operands and dual result registers.

Notes:

- Single long-word operands have an L prefix, as in $LRsd = Rmd + Rnd$*
- Dual normal-word operands have no prefix, as in $Rsd = Rmd + Rnd$*
- The LSB of R_n (bit[0] of the instruction) is used to decode operand size, where $LSB = 0$ determines dual normal words and $LSB = 1$ determines long words*

Compute Block Instruction Format

Table C-9. Shifter Instruction Syntax and Opcodes
(Single Long-Word or Dual Normal-Word Operands and Dual Register)

Syntax	Type	Opcode	
(L)Rsd = LSHIFT Rnd BY Rm	0100	000	xx
(L)Rsd = LSHIFT Rnd BY <imm7>	0100	001	i6 i5 ¹
(L)Rsd = ASHIFT Rnd BY Rm	0100	010	xx
(L)Rsd = ASHIFT Rnd BY <imm7>	0100	011	i6 i5 ¹
(L)Rsd = ROT Rnd BY Rm	0101	000	xx
(L)Rsd = ROT Rnd BY <imm7>	0101	001	i6 i5 ¹
LRsd = FEXT Rnd BY Rm	0101	010	x S0
LRsd = FEXT Rnd BY Rmd	0101	011	x S0
Rsd = GETBITS Rmq BY Rnd	0101	100	x S0
LRsd += FDEP Rnd BY Rm	0110	000	S1 S0
LRsd += FDEP Rnd BY Rmd	0110	001	S1 S0
LRsd += MASK Rnd BY Rmd	0110	010	xx
Rsd += PUTBITS Rmd BY Rnd	0110	100	xx

1 Two LSBs of opcode field is bit 6:5 of the seven bits immediate.

Shifter Instructions Using Short or Byte Operands and Single or Dual Registers

Table C-10 lists the syntax, type-codes and opcodes for shifter instructions with short or byte operands and single or dual result registers.

Notes:

- *Dual short operands have an S prefix, as in $SRs = Rm + Rn$*
- *Quad short operands have an S prefix and a d suffix, as in $SRsd = Rmd + Rnd$*

- *Byte operands have a B prefix, as in $BRs = Rm + Rn$*
- *Octal byte operands have a B prefix and a d suffix, as in $BRsd = Rmd + Rnd$*
- *The LSB of R_n (bit[0] of the instruction) is used to decode operand size, where $LSB=0$ determines bytes and $LSB=1$ determines short words*

Table C-10. Shifter Instruction Syntax and Opcodes
(Short or Byte Operands and Single or Dual Registers)

Syntax	Type	Opcode	S1 S0
$SRs = LSHIFT R_n BY R_m$	1000	000	xx
$SRs = LSHIFT R_n BY <imm5>$	1000	001	
$SRs = ASHIFT R_n BY R_m$	1000	010	xx
$SRs = ASHIFT R_n BY <imm5>$	1000	011	
$BRs = LSHIFT R_n BY R_m$	1001	000	xx
$BRs = LSHIFT R_n BY <imm4>$	1001	001	
$BRs = ASHIFT R_n BY R_m$	1001	010	xx
$BRs = ASHIFT R_n BY <imm4>$	1001	011	
$(S/B)Rsd = LSHIFT R_{nd} BY R_m$	1010	000	xx
$(S/B)Rsd = LSHIFT R_{nd} BY <imm5>$	1010	001	
$(S/B)Rsd = ASHIFT R_{nd} BY R_m$	1010	010	xx
$(S/B)Rsd = ASHIFT R_{nd} BY <imm5>$	1010	011	

Compute Block Instruction Format

Shifter Instructions Using Single Operand

Opcode encoding for the following single-operand instructions differs from the rest of the shifter instructions and is specified by:

Bit[0] For bit i5 of immediate magnitude <imm6>

Bit[1] Operate on single or dual registers

Bits[2] Register file-based or immediate magnitude

Bits[4:3] Test, Clear, Set and Toggle

Table C-11 lists the syntax, type-codes and opcodes for single operand shifter instructions.

Table C-11. Shifter Instruction Syntax and Opcodes (Single Operand)

Syntax	Type	Opcode
BITEST Rn BY Rm	1011	1100x
BITEST Rn BY <imm5>	1011	1110x
BITEST Rnd BY Rm	1011	1101x
BITEST Rnd BY <imm6>	1011	1111i5
Rs = BCLR Rn BY Rm	1011	0000x
Rs = BCLR Rn BY <imm5>	1011	0010x
Rs = BSET Rn BY Rm	1011	0100x
Rs = BSET Rn BY <imm5>	1011	0110x
Rs = BTGL Rn BY Rm	1011	1000x
Rs = BTGL Rn BY <imm5>	1011	1010x
Rsd = BCLR Rnd BY Rm	1011	0001x
Rsd = BCLR Rnd BY <imm6>	1011	0011i5
Rsd = BSET Rnd BY Rm	1011	0101x

Table C-11. Shifter Instruction Syntax and Opcodes
(Single Operand) (Cont'd)

Syntax	Type	Opcode
Rsd = BSET Rnd BY <imm6>	1011	0111i5
Rsd = BTGL Rnd BY Rm	1011	1001x
Rsd = BTGL Rnd BY <imm6>	1011	1011i5
Rs = LD0 Rm	1100	00000
Rs = LD0 Rmd	1100	00010
Rs = LD1 Rm	1100	00001
Rs = LD1 Rmd	1100	00011
Rs = EXP Rm	1100	00100
Rs = EXP Rmd	1100	00110
X/YSTAT = Rm	1100	01000
Rs = X/YSTAT	1100	01001
X/YSTATL = Rm	1100	01110
BKFPT Rmd, Rnd	1100	01111
Rsd = BFOTMP	1100	01010
BFOTMP = Rmd	1100	01011

IALU (Integer) Instruction Format

The instruction format for regular IALU instructions is as shown in [Figure C-3](#) and [Table C-12](#).

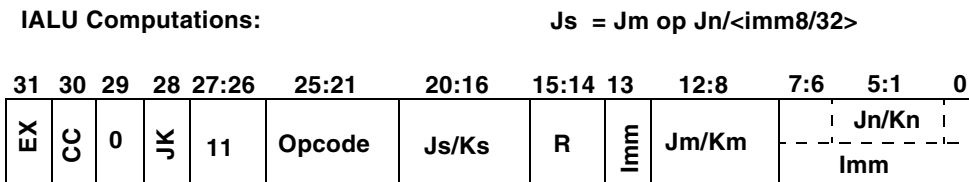


Figure C-3. IALU (Integer) Instruction Format

Table C-12. IALU (Integer) Instruction Opcode Fields

Bits	Field	Description
7–0	Jn/Kn/Imm	Jn/Kn or the immediate is the second operand of the instruction. The selection between Jn/Kn and immediate is done by bit IMM: Imm=0 [5:1] indicate the register. Imm=1 The operand is an immediate right justified two's-complement 8-bit value, and if there is an immediate extension in the same line for the same IALU, it is a 32-bit two's-complement immediate.
12–8	Jm/Km	The first operand of the instruction.
13	Imm	Determines the second operand: 0 = Sets the second operand as register 1 = Sets the second operand as immediate
15–14	R	Reserved.
20–16	Js/Ks	Result register, indicating one of the 32 registers in the IALU register file where the result is to be stored.
25–21	Opcode	See Table C-16 on page C-31 .
27–26	11	Determines that this is a regular IALU instruction.
28	JK	Determines IALU executing unit: 0 = Sets J-IALU as the executing unit 1 = Sets K-IALU as the executing unit

Table C-12. IALU (Integer) Instruction Opcode Fields (Cont'd)

Bits	Field	Description
29	0	Determines that this is an operation to be executed by the IALU.
30	CC	When set, specifies the execution of the instruction by the condition instruction in the same line.
31	EX	When set, determines the instruction as the last in the instruction line.

IALU Move Instruction Format

The instruction format for move register instructions is as shown in [Figure C-4](#) and [Table C-13](#).

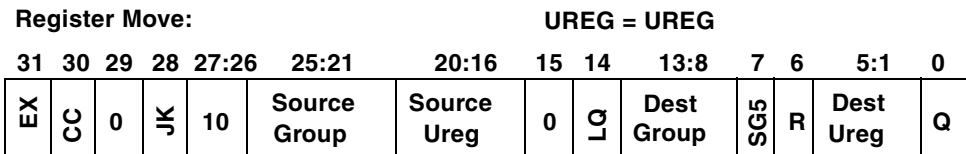


Figure C-4. IALU Move Instruction Format

Table C-13. IALU Move Instruction Opcode Fields

Bits	Field	Description
0	Q	Long or quad data size indication. If LQ is cleared and the data is word, bit[0] is unused; otherwise: 1 = Indicates the data to be quad (128 bits) 0 = Indicates the data to be long (64 bits) in the instruction line
5-1	DEST UREG	Determines the destination register.
6		Reserved
7	SG5	Most Significant Bit (MSB) of the Source Group.

IALU Move Instruction Format

Table C-13. IALU Move Instruction Opcode Fields (Cont'd)

Bits	Field	Description
13–8	DEST GROUP	Determines the destination group.
14	LQ	Data size indication. The size of the register must be aligned to the type of data. For example, if the data is quad, the size of the transaction must be divisible by four; if the data is long the size of the transaction must be divisible by two: 1 = Indicates the data to be either long or quad. 0 = Indicates the data to be word
15	0	Indicates that this is a Ureg transfer instruction.
20–16	SOURCE UREG	Determines the source register.
25–21	SOURCE GROUP	Define the first five bits ([4:0]) of the field that determines the source group. Bit[5] of the field is defined by the SG5 (bit[15] of the instruction).
27–26	10	Determines the type of instruction as a move register operation.
28	JK	Determines the IALU unit: 0 = Sets J-IALU as the executing unit 1 = Sets K-IALU as the executing unit
29	0	Determines that this is an operation to be executed by the IALU.
30	CC	Condition bit. When set, specifies the execution of the instruction by the condition instruction in this line.
31	EX	When set, determines the instruction as the last in the instruction line.

IALU Load Data Instruction Format

The instruction format for Load register instructions is as shown in [Figure C-5](#) and [Table C-14](#).

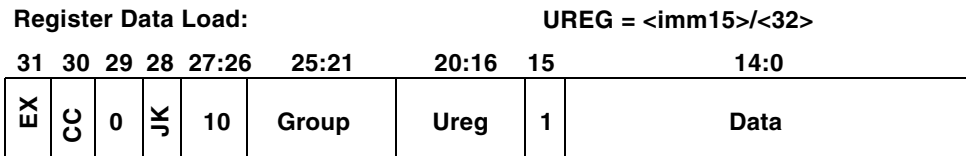


Figure C-5. IALU Load Data Instruction Format

Table C-14. IALU Load Data Instruction Format

Bits	Field	Description
14–0	DATA	Determines 15 bit signed data to be loaded.
15	1	Indicates that this is a Ureg data load instruction.
20–16	UREG	Determines the destination register.
25–21	GROUP	Determines the destination register group.
27–26	10	Determines the instruction type as load register operation.
28	JK	Determines the IALU unit: 0 = Sets J-IALU as the executing unit 1 = Sets K-IALU as the executing unit
29	0	Determines that this is an operation to be executed by the IALU.
30	CC	Condition bit. When set, specifies the execution of the instruction by the condition instruction in this line.
31	EX	When set, determines the instruction as the last in the instruction line.

IALU Load/Store Instruction Format

The instruction format for load register instructions is as shown in [Figure C-6](#) and [Table C-15](#).

Load with Register Update:

$$\text{UREG} = [\text{Jm} + \text{Jn}/\text{Imm}]$$

$$\text{UREG} = [\text{Jm} += \text{Jn}/\text{Imm}]$$

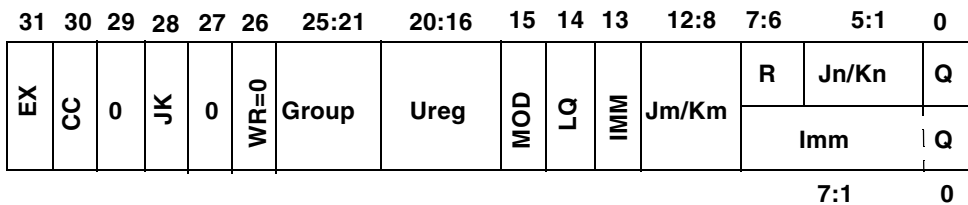


Figure C-6. IALU Load Instruction Format

The instruction format for store register instructions is as shown in [Figure C-7](#) and [Table C-15](#).

Store with Register Update:

$$[\text{Jm} + \text{Jn}/\text{Imm}] = \text{UREG}$$

$$[\text{Jm} += \text{Jn}/\text{Imm}] = \text{UREG}$$

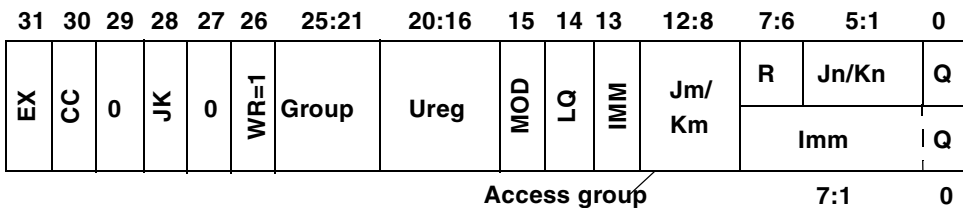


Figure C-7. IALU Store Instructions Format

Table C-15. IALU Load/Store Instruction Opcode Fields

Bits	Field	Description
0	Q	Long or quad data size indication. If LQ is clear, Bit[0] is unused only if [Jm += Jn] is used. If [Jm += imm] is used, Bit [0] is used as LSB of immediate. Note that if LQ is set, Bit [0] of immediate is assumed 0; otherwise: 1 = Indicates the data to be quad (128 bits) 0 = Indicates the data to be long (64 bits) in the instruction line
5–1	JN/KN	Defines the second address in the J-/K-IALU.
7–6	R	Reserved.
12–8	JM/KM	Defines the first address in J-/K-IALU.
13	Imm	Determines the instruction to be either register update (0) or immediate (1).
14	LQ	Data size indication. See also alternate access on page C-31 : 1 = Indicates the data to be either long or quad 0 = Indicates the data to be word
15	MOD	Modify indication: 0 = Determines the operation to be pre-modify no update (+) 1 = Determines the operation to be post modify and update (+=)
20–16	UREG	Determines the destination/source register.
25–21	GROUP	Determines the destination (for load) or source (for store) register group.
26	WR	Discerns between load and store operations: 0 = Determines a load operation, transferring data from memory to a destination register 1 = Determines a store operation, transferring data from a source register to memory
27	0	Determines instruction type as a load/store with register update operation.
28	JK	Determines the IALU unit: 0 = Sets the J-IALU as the executing unit 1 = Sets the K-IALU as the executing unit
29	0	Determines that this is an operation to be executed by the IALU.

IALU Load/Store Instruction Format

Table C-15. IALU Load/Store Instruction Opcode Fields (Cont'd)

Bits	Field	Description
30	CC	Condition bit. When set, specifies the execution of the instruction by the condition instruction in this line.
31	EX	When set, determines the instruction as the last in the instruction line.

The call for alternate access by load/store operations is identified in the group field (five bits), and can only be performed on registers $J_m/K_m0:3$ and the *Uregs* of the compute block. This frees three bits in the J_m/K_m operand field for an alternate access code. The compute block alternate access register groups are as follows:

- Group 1 = Group 0 with alternate access (compute block X)
- Group 3 = Group 2 with alternate access (compute block Y)
- Group 5 = Group 4 with alternate access (compute block X and Y — merge)
- Group 7 = Group 6 with alternate access (compute block Y and X — merge)
- Group 9 = Group 8 with alternate access (compute block X and Y — broadcast)

For alternate access (using circular buffer, bit-reversed, DAB, or SDAB) the J_m/K_m operand field ($bits[12:8]$) is different and defines the access. The field decoding is described in [Table C-16](#), and the assembler syntax for the different types of accesses is described in [Table C-17](#).

Table C-16. IALU Load/Store with Alternate Access (Using Circular Buffer, Bit-Reversed, DAB, or SDAB) Instruction Opcode Fields

Bits	Field	Description
9–8	J_m/K_m	Defines register J_m/K_m between 0 and 3
12–10	Access Group	Circular Buffer operations can only be used in post-modify address modes. For more information, see “Circular Buffer Addressing” on page 6-27 . The access groups are: 000 = Determines normal circular buffer (CB) access 001 = Sets Bit Reverse option 010 = Sets DAB at normal-word misalignment and CB access 011 = Sets DAB at short-word misalignment and CB access

Table C-17. IALU Load/Store with Alternate Access (Using Circular Buffer, Bit-Reversed, DAB, or SDAB) Instruction Assembler Syntax

Option	Assembler Syntax
Circular buffer	$xyRs q = CB \ q[J_m += J_n]$
DAB & circular buffer	$xyRs q = DAB \ q[J_m += J_n]$
Short alignment DAB & circular buffer	$xyRs q = SDAB \ q[J_m += J_n]$
Bit reverse	$xyRs q = BR \ q[J_m += J_n]$

IALU Immediate Extension Format

The instruction format for IALU immediate extensions is as shown in [Figure C-8](#) and [Table C-18](#).

Immediate Extension for IALU

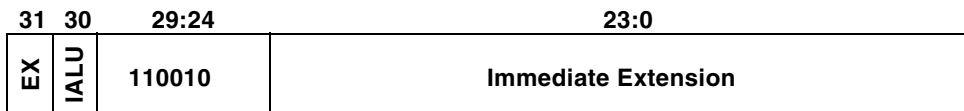


Figure C-8. IALU Immediate Extension Format

Table C-18. IALU Immediate Extension Opcode Fields

Bits	Field	Description
23:0	IMM EXT	Specifies the value of the immediate extension.
29:24	110010	Determines that this is an IALU Immediate Extension instruction.
30	IALU	Indicates the executing IALU: 0 = J-IALU 1 = K-IALU
30	Reserved	Reserved
31	EX	When set, identifies an instruction as the last one in a line.

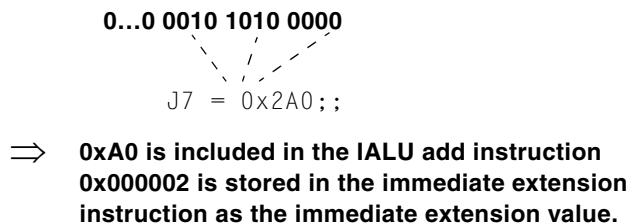


Figure C-9. Examples

Sequencer Instruction Format

Sequencer instruction syntax and opcodes are covered in the following sections:

- “Sequencer Flow Control Instructions” on page C-33
- “Sequencer Direct Jump/Call Instruction Format” on page C-34
- “Sequencer Indirect Jump Instruction Format” on page C-36
- “Condition Codes” on page C-39

Sequencer Flow Control Instructions

The sequencer instruction codes are listed in [Table C-19](#).

Table C-19. Sequencer Flow Control Instruction Syntax and Opcodes

Control flow instructions	op_code
if cond ¹ , JUMP <label> () (NP ²) (ABS)	0x3000 nnnn nnnn = immediate
if cond ¹ , JUMP <rel label> () (NP) ²	0x3100 nnnn nnnn = immediate
if cond ¹ , CALL <label> () (NP ²) (ABS)	0x3080 nnnn nnnn = immediate
if cond ¹ , CALL <label> () (NP) ²	0x3180 nnnn nnnn = immediate
if cond ¹ , CJMP (NP) (ABS) ²	0x3300 8000
if cond ¹ , CJMP (NP) ³	0x3300 C000
if cond ¹ , CJMP_CALL (NP) (ABS) ²	0x3300 A000
if cond ¹ , CJMP_CALL (NP) ³	0x3300 E000
if cond ¹ , RETI (NP) (ABS) ²	0x3300 4000

Sequencer Instruction Format

Table C-19. Sequencer Flow Control Instruction Syntax and Opcodes (Cont'd)

Control flow instructions	op_code
if cond ¹ , RTI (NP) (ABS) ²	0x3300 4040
if cond ¹ , RDS	0x3300 0040
if cond ¹ ; (do, instr;;)	0x3300 0000
SF1/0 ⁴ = cond ¹	0x3300 0400
SF1/0 ⁴ += AND cond ¹	0x3300 1000
SF1/0 ² += OR cond ¹	0x3300 1400
SF1/0 ² += XOR cond ¹	0x3300 1800

- 1 In all examples the condition and NC are all zero. The condition codes are set in bits 21:16, and the NC is set in bit 22.
- 2 Prediction is assumed “not taken” for prediction “taken” bit 30 should be set.
- 3 In CJMP relative and CJMP_CALL relative the prediction must be not taken.
- 4 In this example SF select is zero. if different, set bits 9:7 accordingly.

Sequencer Direct Jump/Call Instruction Format

The instruction format for direct jump/call instructions is as shown in [Figure C-10](#) and [Table C-20](#).

Direct Jump/Call: if cond, jump/call <label> (), (NP), (ABS)

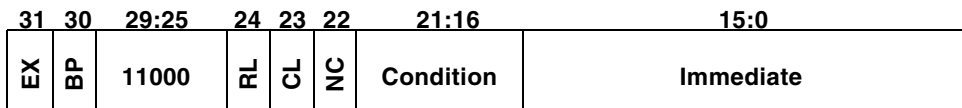


Figure C-10. Sequencer Direct Jump/Call Instruction Format

Table C-20. Sequencer Direct Jump/Call Instruction Opcode Fields

Bits	Field	Description
15:0	IMM	Specifies the immediate value.
21:16	CONDI-TION	Identifies the condition for the branch—see “Condition Codes” on page C-39 .
22	NC	Determines the negate condition: 1= Condition is the NOT of the indicated condition.
23	CL	Indicates instruction type: 1 = Indicates that instruction is a call: next PC is written into the CJMP register, thereby indicating the return address once the call has completed. 0 = Instruction is a Jump
24	RL	0 = Target address is the absolute address in the immediate field plus the immediate extension (if it exists), indicated with option (ABS)—see “Sequencer Operations” on page 7-7 where absolute is 0 and relative is 1. 1 = Branch/call is program counter relative: target address is PC + imm value.
29:25	11000	Determines that this is a direct jump/call instruction.
30	BP	Indicates branch prediction: 1 = The BP bit is set to 1 by default and inserts the entry into the BTB memory. The flow assumes that the branch is taken as default—for example, true. 0 = The BP bit is cleared when the assembly syntax includes the NP option and no branch is taken —see “Sequencer Operations” on page 7-7 .
31	EX	When set, identifies an instruction as the last one in a line. If there is more than one instruction in the line, the EX bit must be 0, since jumps are always the first in the line.

Sequencer Instruction Format

Sequencer Indirect Jump Instruction Format

The instruction format for indirect jump instructions is as shown in [Figure C-11](#) and [Table C-21](#).

Indirect Jump **if cond, cjmp/cjmp_call/rti/reti (np),(abs)**
Conditional **if cond; do <any instruction>**
SF Set: **sfb = psfb mod scond**

31	30	29:23	22	21:16	15:14	13	12:10	9:7	6	5:0
\overline{X}	\overline{B}	1100110	\overline{Z}	Condition	INDJ	\overline{C}	cond flag op	SCF sel	RTI	Reserved

Figure C-11. Sequencer Indirect Jump Instruction Format

Table C-21. Sequencer Indirect Jump Instruction Opcode Fields

Bits	Field	Description
5:0	Reserved	These must be set to 00000.
6	RTI	Indicates return from interrupt. In this case the PMASK register is updated accordingly. Identifies the condition for the branch—see “Condition Codes” on page C-39 . Note: The RDS opcode is 0xB3080040, assuming this is the only instruction in the line. It functions like the RTI with jump false.
9:7	SCF SELECT	Selects the condition flag as source and result of the SCF operation: 000 = Control SF0 001 = Control SF1 <xy> = compute block (X, Y or both) and SCF0 <xy>1 = compute block (X, Y or both) and SCF1

Table C-21. Sequencer Indirect Jump Instruction Opcode Fields (Cont'd)

Bits	Field	Description
12:10	COND FLAG OP	Together with the SF select fields, defines the result of the condition flag: SF1/0 += logic_operation cond The logic_operations are: 000 = No operation 001 = SF=cond; assigns condition to SF 100 = SF+=AND cond 101 = SF+=OR cond 110 = SF+=XOR cond
13	CL	Specifies type of jump call: 1 = Indicates CJMP_CALL a computed call, where the next PC is written into the CJMP register, thereby indicating the return address once the call has completed. 0 = Regular computed Jump
15:14	INDJ	Defines the type of indirect jump: 00 = Conditional instruction; no jump 01 = RTI/RETI – jump to address pointed by the value in the RETI register. For RTI, update interrupt mask register PMASK 10 = CJMP – computed jump to absolute address; PC=CJMP 11 = CJMP – relative jump address; PC+=CJMP
21:16	CONDI- TION	Identifies the condition for the branch—see “Condition Codes” on page C-39 .
22	NC	Determines the negate condition: 1 = Condition is the NOT of the indicated condition.
29:23	1100110	Determines that this is an indirect jump instruction.
30	BP	Indicates branch prediction: 1...The BP bit is set to 1 by default and inserts the entry into the BTB memory.
31	EX	When set, identifies an instruction as the last one in a line. If there is more than one instruction in the line, the EX bit must be 0, since jumps are always the first in the line.

Sequencer Instruction Format

Note that the basic instruction is:

```
if cond, jump... ; else, <any instruction>
```

The instruction `if cond; do <any instruction>` uses the same format as the jump instructions. The other instructions in the line, which are conditioned by the jump condition, are executed by the inverse of the condition. To keep the assembler readable when using “`if cond; do <any instruction>;`”

the condition coded in the machine code is the inverse of the condition in the assembler.



Two predicted jumps can not reside in the same quad-aligned word. The BTB cannot distinguish between the two and will cache them both to the same location. Thus, the execution will be wrong. Since, prior to the linker, there is no way to tell if the two jumps that are close to each other actually reside in the same quad-aligned word, you must insure that there are at least three instructions between any two jumps.

Another solution involves giving one of the jumps the `NP` option. However, you will pay an up to six-cycle penalty every time the jump is taken. For example:

```
if <cond1>, jump <label 1>; nop; nop; nop;;  
if <cond2>, jump <label2>;
```


Condition Codes

Condition codes for conditional instructions are covered in the following sections:

- “Compute Block Conditions” on page C-39
- “IALU Conditions” on page C-40
- “Sequencer and External Conditions” on page C-40

Compute Block Conditions

The compute block condition is identified when the two MSBs $\neq 0$ and the four LSBs are less than 1011. The decode of the conditions is listed in [Table C-22](#):

Table C-22. ALU, Multiplier, and Shifter Condition Codes

Code	Condition	
<XY>0000	AEQ	ALU result equals zero
<XY>0001	ALT	ALU result less than zero
<XY>0010	ALE	ALU result less than or equals zero
<XY>0011	MEQ	multiplier result equals zero
<XY>0100	MLT	multiplier result less than zero
<XY>0101	MLE	multiplier result less than or equals zero
<XY>0110	SEQ	Shifter result equals zero
<XY>0111	SLT	Shifter result less than zero
<XY>1001	SF0	Static Condition Flag #0
<XY>1010	SF1	Static Condition Flag #1
1111XX	Not Compute Block conditions	

Sequencer Instruction Format

The <XY> field decode is similar to this field in compute block instructions:

00 = None compute block condition

01 = Compute block X condition

10 = Compute block Y condition

11 = Both X and Y condition (excluding 1111XX)

IALU Conditions

The IALU conditions are identified by 000<JK> in the four MSBs. The <JK> bit is similar to <JK> in the IALU instructions:

0 = J-IALU

1 = K-IALU

The condition codes are listed in [Table C-23](#).

Table C-23. IALU Condition Codes

Code	Condition	
000<JK>00	JEQ/KEQ	J-/K-IALU result equals zero
000<JK>01	JLT/KLT	J-/K-IALU result less than zero
000<JK>10	JLE/KLE	J-/K-IALU result less than or equals zero

Sequencer and External Conditions

The external conditions refer to non-core conditions. These are identified by 001 in three MSBs.

The codes are listed in [Table C-24](#).

Table C-24. Sequencer and External Condition Codes

Code	Condition
001000	TRUE
001001	iSF0 – IALU Static Flag #0
001010	iSF1 – IALU Static Flag #1
001011	BM – Bus master
001100	LC0E – Loop counter #0 is zero
001101	LC1E – Loop counter #1 is zero
1111<FLG>	FLAG<n>_IN (n = 0 to 3) – external flag input 0 to 3

Sequencer Immediate Extension Format

The instruction format for Immediate Extensions for JUMP instructions is as shown in [Figure C-12](#) and [Table C-25](#).

Immediate Extension for Jump

31 30		29:22	21:16	15:0
EX	R	11001110	R	Immediate Extension

Figure C-12. Immediate Extensions (for JUMP Instruction) Format

Miscellaneous Instruction Format

Table C-25. Immediate Extensions for JUMP Instruction
Opcode Fields

Bits	Field	Description
15:0	IMM EXT	Specifies the value of the immediate extension.
21:16	Reserved	Reserved
29:22	11001110	Determines that this is an immediate extension for a jump instruction.
31	EX	When set, identifies an instruction as the last one in a line. Remember, when more than one instruction is included in the instruction line, this must be the second.

Miscellaneous Instruction Format

The instruction format for miscellaneous instructions is as shown in [Figure C-13](#) and [Table C-26](#).

31 30		29:22	21:18	17:5	4:0
EX	0	11001111	Opcode	Reserved	Imm

Figure C-13. Miscellaneous Instruction Format

Table C-26. Miscellaneous Instruction Opcode Fields

Bits	Field	Description
4:0	IMM	Immediate – for TRAP instruction. Bit 0 is also used for option (LP) in IDLE instruction.
17:5	Reserved	Reserved
21:18	OPCODE	Determines the operation code. See Table C-27 on page C-43.
29:22	11001111	Determines that this is an Other instruction type.

Table C-26. Miscellaneous Instruction Opcode Fields (Cont'd)

Bits	Field	Description
30	0	Constant 0 – no conditional for this group.
31	EX	When set, identifies an instruction as the last one in a line.

The operation codes for this type of instruction are listed in [Table C-27](#).

Table C-27. Other Instruction Syntax and Opcodes

Instruction Syntax	Type	Opcode	Comments	
NOP	11001111		0000	
IDLE	11001111		0001	Option (lp) is indicated by setting bit 0
BTBINV	11001111		0010	Use BTBINV when the code in internal memory is changed, even small portions of memory.
TRAP (spvcmd)	11001111		0100	The “spvcmd” is an immediate indicated by bits 4:0
EMUTRAP	11001111		1000	

Miscellaneous Instruction Format

I INDEX

Symbols

- ()/2 average operator, [8-8](#), [8-53](#), [8-206](#)
- * multiply operator, [4-11](#), [8-122](#),
[8-138](#), [8-163](#)
- * += multiply-accumulate operator,
[8-125](#), [8-130](#), [8-141](#), [8-146](#), [8-152](#),
[8-156](#)
- ** complex multiply operator, [4-3](#), [4-8](#),
[8-152](#), [8-156](#)
- + add operator, [8-3](#), [8-6](#), [8-36](#), [8-51](#),
[8-89](#), [8-113](#), [8-202](#), [8-204](#)
- negate operator, [8-13](#), [8-60](#)
- subtract operator, [8-3](#), [8-6](#), [8-36](#), [8-51](#),
[8-89](#), [8-202](#), [8-204](#)
- .D unit, *See* IALU or ALU
- .L unit, *See* ALU
- .M unit, *See* multiplier
- .S unit, *See* shifter
- [+] pre-modify operator, [1-12](#), [6-13](#),
[6-29](#), [8-220](#), [8-221](#), [C-29](#)
- [+=] post-modify operator, [1-12](#),
[1-36](#), [6-13](#), [6-28](#), [6-29](#), [6-31](#),
[8-220](#), [8-221](#), [8-222](#), [8-224](#), [C-31](#)

Numerics

- 16-bit short word, [2-22](#), [2-23](#)
- 32-bit normal word, [2-23](#), [2-24](#)

- 32-bit single-precision, [2-17](#)
- 40-bit extended-precision, [2-19](#)
- 64-bit long word, [2-24](#), [2-25](#)
- 8-bit byte word, [2-20](#), [2-21](#)

A

- ABS (absolute address) option, [7-17](#),
[7-78](#), [8-230](#), [8-232](#), [8-234](#), [A-13](#),
[C-33](#)
- ABS instruction, [2-4](#), [3-2](#), [3-19](#), [3-29](#),
[3-30](#), [3-32](#), [6-42](#), [8-10](#), [8-32](#), [8-57](#),
[8-212](#), [A-2](#), [A-3](#), [A-4](#), [A-10](#), [C-5](#),
[C-10](#)
See also X option
- absolute address, [7-17](#), [7-78](#), [8-230](#),
[8-232](#), [8-234](#), [A-13](#), [C-33](#)
See also ABS option
- absolute value, *See* ABS instruction
- AC (ALU carry) flag, *See* ALU status
- access
 - quad data, [1-18](#), [6-2](#), [6-4](#)
 - restrictions, [1-41](#), [1-43](#)
 - types, [6-16](#)
- accumulate, *See* multiply-accumulate
- ACS instruction, [3-3](#), [3-23](#), [3-25](#), [3-33](#),
[7-58](#), [8-105](#), [8-113](#), [A-6](#), [C-13](#)

INDEX

- add compare select, *See* ACS
 - instruction
- add instruction, 8-3, 8-36, 8-51, 8-89, 8-202
 - and accumulate *See* SUM instruction
 - and divide *See* ()/2 operator
 - and subtract *See* dual operation
 - See also* + operator
 - with average, 8-8, 8-206
 - with carry, 8-6, 8-204
- additional literature, [xix](#)
- address, from program label, 7-15
- ADDRESS() assembler command, 7-73
- addresses
 - absolute, 7-17, 7-78, 8-230, 8-232, 8-234, [A-13](#), [C-33](#)
 - direct, 6-14, 6-29, 8-220, 8-221, 8-224, [C-29](#), [C-31](#)
 - indirect, 1-12, 6-28, 6-29, 8-224, [C-29](#)
 - relative, 1-16
- address from program label, 7-73
- addressing
 - direct, 1-14
 - indirect, 1-14
 - See also* bit-reversed addressing and circular buffer addressing
- AEQ (ALU equal zero) condition, *See* ALU conditions
- AI (ALU invalid) flag, *See* ALU status
- ALE (ALU less or equal zero) condition, *See* ALU conditions
- algorithms, 3-24
 - CDMA, 3-21
 - digital filters, 1-13
 - FFT, 1-13
 - Fourier transforms, 1-13
 - turbo-code, 1-9, 3-21
 - Viterbi, 1-9, 3-21
- ALT (ALU less than zero) condition, *See* ALU conditions
- ALU, 3-1, [C-10](#), [C-11](#)
 - conditions, 3-14, 6-12, [C-39](#)
 - examples, 3-16
 - instructions, 8-2 to 8-90, [C-4](#)
 - instructions summary, 3-28
 - opcodes, [C-5](#) to [C-10](#)
 - operations, 3-5
 - options, 3-7
 - quick reference, [A-2](#)
 - status, 3-11, 3-12, 3-15, [B-2](#)
 - sticky status, 3-12, [B-2](#)
- AN (ALU negative) flag, *See* ALU status
- AND instruction, 3-2, 8-38, 8-213, [C-6](#)
- AND NOT instruction, 3-2, 8-38, 8-213, [C-6](#)
- arbitration, 1-17
- architecture
 - Static Superscalar, 1-7, 1-15
 - system, 1-4
- Arithmetic Logic Unit, *See* ALU
- arithmetic shift, *See* ASHIFT or ASHIFTR instructions

- ASHIFT instruction, 5-6, 5-21, 8-176, A-9, C-18, C-20, C-21
- ASHIFTR instruction, 6-42, 8-215, A-10
- assembly language, xix, A-1
- AUS, AVS, AOS, AIS bits, B-2
- AUS (ALU underflow, sticky) flag, 3-12
- AV (ALU overflow) flag, *See* ALU status
- average, *See* ()/2 average operator
- AVS (ALU overflow, sticky) flag, *See* ALU sticky status
- AZ (ALU zero) flag, *See* ALU status
- B**
- base, circular buffer, 6-4
- BCLR instruction, 5-7, 5-21, 8-192, A-9, C-22
- BFOINC instruction, 3-2, 3-30, 5-11, 5-12, 8-30, 8-187, A-3, C-8
- BFOTMP register, 1-11, 5-4, 5-13, 8-187, 8-189, 8-199, C-23
- BFP, *See* bit FIFO pointer
- BF (shifter block floating-point) flag, *See* shifter status
- bit
- clear/set/toggle, *See* BCLR, BSET, and BTGL instructions
 - deposit, *See* PUTBITS instruction
 - mask, *See* MASK instruction
 - operations, 1-11, 5-3, 5-7, 5-8, 5-11
 - test, *See* BITEST instruction
- BITEST instruction, 5-7, 5-21, 8-191, A-9, C-22
- bit FIFO increment, 3-2, 3-30, 5-11, 5-12, 8-30, 8-187, A-3, C-8
- bit FIFO pointer (BFP), 8-187
- bit-reversed addressing, 1-13, 1-44, 6-8, 6-9, 6-31, 6-32, 6-42, 6-43, 6-44, 8-202, 8-222, 8-224, A-10, A-11, A-12, C-31
- example, 6-33
- bit wise barrel shifter, *See* shifter
- BKFPT instruction, 5-22, 8-197, A-10, C-23
- block floating-point, *See* BKFPT instruction
- BM (bus master) flag, 7-45
- BM condition, *See* bus master
- booting, 1-19
- branch, 7-16
- CALL, 1-34, 1-36, 1-46, 7-16, 7-78, 8-230, A-13, C-33, C-41
 - CJMP, 1-31, 1-32, 7-17, 7-78, 8-232, A-13, C-33
 - CJMP_CALL, 7-16, 7-17, C-33
 - cost, 7-45
 - See also* conditional branch, effects on pipeline
 - in pipeline, 7-44
 - prediction, 1-15

INDEX

- branch target buffer, 1-7, 1-13, 1-14, 1-15, 7-11, 7-34, 7-44
 - invalidate, 8-241
 - registers, B-33
 - See also* BTBINV instruction
 - set, 7-36
 - way, 7-36
- BR (bit-reversed) option, 6-8, 6-9, 6-32
- broadcast accesses, 6-16
- BSET instruction, 5-7, 5-21, 8-192, A-9, C-22, C-23
- BTB, *See* branch target buffer
- BTBEN bit, 7-34, B-20
- BTBINV instruction, 7-34, 7-79, 8-237, 8-241, A-13, C-43
- BTBLK bit, 7-34, B-20
- BTGL instruction, 5-7, 5-21, 8-192, A-9, C-22, C-23
- bus, 1-17
 - lock, 7-22
 - master, 7-77
 - requests, 7-60
- byte word data, 2-10, 2-19
- C**
- C (clear) option, 4-14
- CALL instruction, 1-34, 1-36, 7-16, 7-78, 8-230, A-13, C-33
- carry, ALU, 3-12, 6-10
- CB (circular buffer) option, 1-44, 6-8
- CDMA algorithm, 3-21, 3-26
- circular buffer addressing, 1-44, 6-8, 6-27, 6-42, 6-43, 6-44, 8-202, 8-222, 8-224, A-10, A-11, A-12, C-31
- circular buffer registers, B-12, B-13
- circular buffers, 6-27
- CJMP_CALL instruction, 7-16, 7-17, C-33
- CJMP (computed jump) option, 7-55
- CJMP instruction, 7-17, 7-78, 8-232, A-13, C-33
- CJMP register, 6-8, 6-9, 6-42, 7-16, 7-17, 8-202, A-10, B-13
- clear, clear and round, *See* C and CR options
- clear bit, *See* BCLR instruction
- CLIP instruction, 3-2, 3-30, 3-32, 8-24, 8-72, A-3, A-5, C-6, C-10
- CLU, 1-9
 - data types and sizes, 3-22
 - examples, 3-21
 - instructions, 8-91 to 8-120, C-12
 - dependency, 7-57
 - opcodes, C-12
 - status, 3-28
- combination constraints, *See* restrictions
- communication logic unit, *See* CLU
- COMPACT instruction, 3-3, 3-4, 3-11, 3-31, 4-14, 4-26, 8-45, 8-171, A-4, A-8, C-8, C-17
- compact multiplier result, *See* COMPACT instruction

- compare, *See* COMP, FCOMP, or ACS instructions
- COMP instruction, 3-2, 3-30, 6-42, 8-22, 8-208, A-3, A-10, C-6
- complement, *See* negate or one's complement
- complex multiply-accumulate, *See* multiply-accumulate
- complex numbers, 1-11, 2-7, 4-3
- COM port, McBSP, *See* link ports
- compress, *See* COMPACT instruction
- compute block, 1-8, 2-1, 3-1, 4-1, 5-1
 - conditions, C-39
 - dependencies, 7-56
 - opcodes, C-32
 - pipeline, 7-29
 - registers, B-5, B-6, B-7
 - restrictions, 1-37
 - selection, 2-7
 - status, 2-4, 7-45, 7-77
- computed jump/call, 8-232
- conditional branch
 - CALL, 1-34, 1-36, 1-46, 7-16, 7-78, 8-230, A-13, C-33, C-41
 - CJMP, 1-31, 1-32, 7-17, 7-78, 8-232, A-13, C-33
 - CJMP_CALL, 7-16, 7-17, C-33
 - effects on pipeline, 7-44
 - NP branch, 7-18, 7-37, 7-44, 7-50, 7-78, 8-230, 8-232, 8-234, A-13, C-33
 - static flags, 8-239
- conditional execution, 1-24, 7-12, 8-237, 8-238
 - DO, 1-24, 4-20, 6-12, 7-12, 7-78, 8-237, A-13
 - ELSE, 1-24, 7-13, 7-78, 8-238, A-13
- conditional instruction, 1-24, 3-14, 4-20, 5-16, 6-12, 7-12, 7-13
- conditional sequencing, 8-238
- condition codes, 7-45, C-39
- condition flags, 7-45, 8-239
- conditions
 - ALU, 3-14, 6-12
 - dependency, 7-55
 - IALU, C-40
 - multiplier, 4-20
 - sequencer, C-40
 - shifter, 5-16
 - static flags, 8-239
 - TRUE, 7-45
- conjugate, *See* (complex conjugate) option
- constraints, *See* restrictions
- context switching, 1-16
- control flow, *See* sequencer instructions
- conventions, xxviii
- conventions, instruction notation, 1-22

INDEX

- conversion
 - 32- to 40-bit floating-point, 3-2, 3-32, 8-68, A-5, C-11
 - 40- to 32-bit floating-point, 3-2, 3-32, 8-70, A-5, C-11
 - fixed- to floating-point, 3-2, 3-32, 8-66, A-5, C-10
 - floating- to fixed-point, 3-2, 3-32, 8-64, A-5, C-10, C-11
- COPYSIGN instruction, 3-2, 3-32, 8-74, A-5, C-10
- counting ones, 8-28
- CR (clear/round) option, 4-14, 4-15
- customer support, [xx](#)

- D**
- D unit, *See* IALU or ALU
- DAB, *See* data alignment buffer
- DAB (data alignment buffer) option, 1-29, 1-33, 1-41, 1-44, 2-1, 6-18, 6-23, 6-43, 8-222, A-11, C-31
- data accesses, 1-18
- data addressing, 6-2, 8-218, 8-222, 8-224
- data alignment buffer, 1-18, 1-29, 1-33, 1-41, 1-44, 2-1, 6-18, 6-23, 6-26, 6-43, 8-222, A-11, C-31
 - accesses, 1-18, 6-23, C-31
 - load restrictions, 1-43
 - operation, 8-222
- data move, 6-2, 8-218
- data registers, 2-3, B-5, B-6, B-7, B-10, B-11
- data size, 3-4

- data type, *See* numeric formats
- DBGEN bit, B-20
- DBGE register, 8-243, B-13
- deadlock, 7-60
- debug enable, B-20
- DEC instruction, 3-2, 3-30, 8-20, A-3, C-6
- decode, 7-28
- decrement, *See* DEC instruction
- dependency, 7-55
 - compute block, 7-56, 7-62
 - condition, 7-55
 - IALU, 7-63, 7-64
 - load, 7-62, 7-63
 - resource and pipeline, 7-55
- dependency condition, 7-55
- deposit
 - bits, *See* PUTBITS instruction
 - field, *See* FDEP instruction
- DESPREAD instruction, 1-9, 3-3, 3-21, 3-22, 3-26, 3-27, 3-33, 8-105, 8-106, A-6, C-13
- difference
 - See* parallel absolute value of
 - See* - subtract operator
- digital filters, 1-13
- direct addressing, 1-12, 1-36, 6-13, 6-14, 6-28, 6-29, 6-31, 8-220, 8-221, 8-222, 8-224, C-29, C-31
- direct calls, 1-14
- direct jumps, 1-14
- DO instruction, 1-24, 4-20, 6-12, 7-12, 7-78, 8-237, A-13
- double register, 2-9

- Dreg (data registers), [2-5](#)
- DSP architecture, [1-6](#)
- dual operation
 - add and subtract, fixed-point, [8-36](#)
 - add and subtract, floating-point, [8-89](#)
- dynamic violations, *See* restrictions

- E**
- edge sensitivity, [B-20](#)
- ELSE instruction, [1-24](#), [7-13](#), [7-78](#), [8-238](#), [A-13](#)
- EMCAUSE bit, [B-22](#)
- EMUIR register, [8-243](#)
- emulation, [1-20](#), [7-11](#)
- EMUTRAP instruction, [8-237](#), [8-243](#), [C-43](#)
- EXCAUSE bit, [B-22](#)
- exceptions, [7-22](#), [7-72](#)
- execution flow, [7-1](#) to [7-71](#)
- execution status, [7-78](#), [8-239](#), [A-13](#), [C-34](#)
 - ALU, [3-15](#), [B-2](#)
 - IALU, [6-13](#)
 - multiplier, [4-21](#), [B-2](#), [B-3](#)
 - shifter, [B-2](#)
- expand, 8- or 16-bit, *See* EXPAND instruction
- EXPAND instruction, [3-3](#), [3-4](#), [3-11](#), [3-31](#), [8-40](#), [A-4](#), [C-7](#)
- EXP instruction, [5-21](#), [8-195](#), [A-9](#), [C-23](#)
- exponent, extract, *See* EXP instruction

- EXTD instruction, [3-2](#), [3-32](#), [8-68](#), [A-5](#), [C-11](#)
- extend, field, *See* FEXT instruction
- extended output range, ABS, *See* X option
- extended precision, [2-19](#)
- extended word, [8-68](#)
- extract
 - exponent, [8-195](#)
 - fields, [8-181](#), [8-187](#)
 - leading ones/zeros, [8-194](#)
 - mantissa, [8-85](#)

- F**
- factor, scaling block floating-point, [8-197](#)
- FCOMP instruction, [3-32](#), [8-62](#), [A-4](#), [C-11](#)
- FDEP instruction, [1-34](#), [5-3](#), [5-8](#), [5-21](#), [8-183](#), [A-9](#), [C-19](#), [C-20](#)
 - restrictions, [1-38](#)
- fetch, [7-28](#)
- fetch unit pipe, [7-28](#)
- FEXT instruction, [5-8](#), [5-21](#), [8-181](#), [A-9](#), [C-18](#), [C-19](#), [C-20](#)
- FFT algorithms, [1-13](#)
- field
 - deposit, [8-183](#)
 - extract, [8-181](#)
 - mask, [8-185](#)
- fixed-point formats, [1-8](#), [2-19](#), [4-3](#)
- fixed- to floating-point conversion, [8-66](#)

INDEX

FIX instruction, 3-2, 3-32, 8-64, A-5, C-10, C-11
flag I/O, B-20, B-21
flags
 condition, 7-45
 TRUE, 7-45
flag update, 3-12, 3-28, 4-19, 5-16, 6-10
FLAGx_EN bits, B-20
FLAGx_OUT bits, B-21
FLG3-0 pin conditions, 7-45, 7-77
FLG bit, B-22
floating-point extended to normal word conversion, *See* SNGL instruction
floating-point normal to extended word conversion, *See* EXTD instruction
floating- to fixed-point conversion, *See* FIX instruction
FLOAT instruction, 3-2, 3-32, 8-66, A-5, C-10
format, *See* data format
Fourier transforms, 1-13
fractional
 compaction, 8-45
 data, 2-7
 format, 3-11, 4-10, 6-7
 mode, 3-7, 4-9, 6-7
 multiplier, 4-10
 option, *See* I option
 results, 2-19

G

GETBITS instruction, 2-4, 5-12, 5-21, 8-187, 8-189, A-9, C-20
 restrictions, 1-38
GIE bit, B-15
global interrupt enable, *See* GIE bit
global memory, B-44
GSCFx bits, B-22

I

I (integer-fractional) option, 3-11, 4-10
I (interleave) option, 8-105
IAB, *See* instruction alignment buffer
IALU, 1-12
 conditions, 6-12, C-40
 examples, 6-37
 immediate extensions, 1-15, 6-36, 8-226, C-32, C-41
 instructions, 8-200 to 8-227
 instruction summary, 6-39
 opcodes, C-3, C-24, C-25, C-27, C-28, C-29, C-31
 operations, 6-5, 6-13
 options, 6-6
 quick reference, A-10
 registers, 1-13, 6-3, B-10, B-11, B-12, B-13
 restrictions, 1-39
 static flags, 6-13
 status, 6-10
IDLE bit, B-22
IDLE instruction, 7-3, 7-79, 8-237, 8-240, A-13, C-42, C-43
IEEE 754/854 format, 1-20, 2-16

- IEEE standards, exceptions, 1-20
- IF...DO instruction, 1-24, 4-20, 6-12, 7-12, 7-78, 8-237, A-13
- IF...ELSE instruction, 1-24, 7-13, 7-78, 8-238, A-13
- ILAT register, 7-66, B-13
- IMASK register, 7-66, 7-70, B-13
- immediate data operations, 6-2, 8-218
- immediate extension, 1-12, 1-13, 1-15, 1-34, 1-36, 1-46, 6-13, 6-28, 6-29, 6-31, 7-16, 7-78, 8-220, 8-221, 8-222, 8-224, 8-230, A-13, C-29, C-31, C-33, C-41
 - format, JUMP (sequencer), C-42
 - format (IALU), C-41
 - opcode fields (IALU), C-32
 - operations, 1-15, 6-36
 - restrictions, 1-45
- INC instruction, 3-2, 3-30, 8-20, A-3, C-6
- increment, *See* INC instruction
- index, circular buffer, 6-4
- indirect addressing, 1-12, 1-36, 6-13, 6-14, 6-28, 6-29, 6-31, 8-220, 8-221, 8-222, 8-224, C-29, C-31
- indirect jump, 1-14
- input, flag, B-20, B-21
- instruction alignment buffer, 7-28, 7-31
- instruction dispatch/decode, *See* sequencer
- instruction line, 1-20, 2-5, C-1
- instruction notation conventions, 1-22
- instruction parallelism rules, 1-24
- instruction pipeline operations, 7-26
- instructions
 - bit manipulation, 1-11, 5-3
 - combination constraints, 1-24
 - compute block, 2-1
 - conditional, 1-24, 3-14, 4-20, 5-16, 6-12, 7-12, 7-13
 - control flow, 1-13
 - dependency, 7-55
 - execution, 7-30
 - immediate extension, 1-13, 1-15
 - line, 1-20
 - quad, 1-15, 1-18
 - resource conflicts, 7-55
 - restriction, 1-36
- instruction set, 8-1 to 8-244
- instruction slot, 1-20, 2-5
- integer
 - compaction, 8-45
 - data, 2-7
 - format, 3-11, 4-10, 6-7
 - mode, 3-7, 4-9
 - multiplier, 4-10
- integer ALU pipe, 7-28, 7-29
- Integer Arithmetic Logic Unit, *See* IALU
- intended audience, xvii
- internal transfer, 1-18

INDEX

- interrupt, [1-14](#), [1-16](#), [7-3](#)
 - bus lock, [7-22](#)
 - conditional instruction, [7-68](#)
 - exception, [7-22](#)
 - external, [7-23](#)
 - interrupt disable, [7-70](#)
 - pipeline, [7-66](#)
 - sensitivity, [B-20](#)
 - software interrupt, [7-22](#)
 - vectors, [7-22](#), [B-41](#)
- interrupt masking, [7-21](#)
- invalid operation, ALU, [3-12](#)
- IRQ3–0 pins, [1-14](#)
- IRQx_EDGE bits, [B-20](#)
- ISFx (integer static flag) conditions, [7-77](#)
- ITYPE field, [C-2](#)
- IVT registers, [B-41](#)

- J**
- J (complex conjugate) option, [4-16](#)
- J-IALU, *See* IALU
- JTAG port, [1-20](#)
- JUMP instruction, [1-34](#), [1-46](#), [7-3](#), [7-16](#), [7-78](#), [8-230](#), [A-13](#), [C-33](#), [C-41](#)

- K**
- K-IALU, *See* IALU

- L**
- L unit, *See* ALU
- label to address conversion, [7-15](#)
- LCx register, [B-13](#)
- LCxE (loop counter expired)
 - condition, [7-77](#)
- LDx instruction, [5-21](#), [8-194](#), [A-9](#), [C-23](#)
- leading ones/zeros, extract, *See* LDx instruction
- least recently used (LRU), [7-34](#)
- left rotate, [8-217](#)
- length, circular buffer, [6-4](#)
- link interrupt, [7-23](#)
- link port, [1-3](#), [1-7](#)
- load, [1-12](#), [1-29](#), [1-30](#), [1-32](#), [1-33](#)
 - opcodes, [C-28](#), [C-29](#), [C-31](#)
 - operations, [6-2](#), [8-218](#)
 - register, [8-196](#), [8-199](#), [8-220](#), [8-222](#), [C-28](#), [C-29](#), [C-31](#)
 - restrictions, [1-40](#), [1-43](#)
- load data
 - instruction, [8-226](#)
 - opcodes, [C-27](#)
 - register, [8-222](#)
- logarithm, [3-2](#), [3-32](#), [8-87](#), [A-5](#), [C-11](#)
- LOGB instruction, [3-2](#), [3-32](#), [8-87](#), [A-5](#), [C-11](#)
- logical
 - AND/AND NOT/OR/XOR/NOT, [8-38](#), [8-213](#)
 - shift, [8-176](#)
 - shift (integer), [8-215](#)
- logical operations
 - ALU, [8-38](#), [C-6](#)
 - IALU, [3-2](#), [8-213](#), [C-6](#)
- long word accesses, [6-16](#)
- long word data, [2-10](#)

- loop, 7-3, 7-19
- LP (low power) option, 8-240
- LRU bit, *See* least recently used
- LSHIFT instruction, 5-5, 5-21, 8-176, A-9, C-18, C-20, C-21
- LSHIFTR instruction, 6-42, 8-215, A-10

- M**
- M unit, *See* multiplier
- MACs, *See* multiply-accumulate
- magnitude, rotate, 8-179
- MANT instruction, 3-2, 3-32, 8-85, A-5, C-10
- mantissa, *See* MANT instruction
- manual
 - audience, xvii
 - contents, xviii
 - conventions, xxviii
 - new in this edition, xix
 - related documents, xxiii
- mask, field, *See* MASK instruction
- masked interrupt, 7-21
- MASK instruction, 1-34, 5-3, 5-8, 5-21, 8-185, A-9, C-19, C-20
 - restrictions, 1-38
- maximum, *See* MAX, VMAX, or TMAX instructions
- MAX instruction, 3-2, 3-10, 3-25, 3-30, 3-32, 3-33, 6-42, 8-14, 8-55, 8-92, 8-99, 8-210, A-2, A-4, A-5, A-10, C-5, C-11, C-12

- memory
 - access restrictions, 1-40
 - internal, 1-7, 1-16
 - See also* broadcast accesses, merged accesses, and normal accesses
- merge, *See* MERGE instruction
- merged accesses, 6-16
- MERGE instruction, 3-3, 3-31, 8-49, A-4, C-8
- MI (multiplier invalid) flag, *See* multiplier status
- minimum, *See* MIN or VMIN instructions
- MIN instruction, 3-2, 3-10, 3-30, 3-32, 6-42, 8-14, 8-55, 8-210, A-2, A-4, A-10, C-5, C-11
- MIS (multiplier invalid, sticky) flag, *See* multiplier sticky status
- MN (multiplier negative) flag, *See* multiplier status
- mode, 7-11
 - emulation mode, 7-11
 - supervisor mode, 7-11
 - user mode, 7-11
- MODE bit, B-22
- modifier, circular buffer, 6-4
- MOS (multiplier overflow, sticky) flag, *See* multiplier sticky status
- move instruction, 8-226
- move restrictions, 1-40
- MR register, 4-5, 4-11, 4-15, 4-17, 4-18, 4-26, 8-165, A-7, C-17
- multichannel buffered serial port, McBSP, *See* link ports

INDEX

- multiplier, [4-1](#), [4-2](#), [8-121](#)
 - conditions, [4-20](#), [C-39](#)
 - examples, [4-21](#)
 - instructions, [8-121](#) to [8-174](#), [C-14](#)
 - instruction summary, [4-23](#)
 - integer, fractional, [4-10](#)
 - opcodes, [C-16](#)
 - operations, [4-4](#)
 - options, [4-8](#)
 - quick reference, [A-6](#)
 - result register, [8-165](#)
 - rounding, [4-14](#)
 - saturation, [4-11](#), [4-12](#)
 - static flags, [4-21](#)
 - status, [4-18](#), [4-19](#), [B-2](#)
 - sticky status, [B-3](#)
- multiply, *See* * operator
- multiply-accumulate, [8-125](#), [8-130](#),
[8-141](#), [8-146](#)
 - complex, [8-152](#), [8-156](#)
 - short word, [8-152](#)
 - short word with move, [8-156](#)
 - dual operation
 - normal word, [8-130](#)
 - quad-short word, [8-146](#)
 - normal word, [8-125](#)
 - quad-short word, [8-141](#)
 - See also* * += operator
- multiply-accumulator, *See* multiplier
- multiprocessing, [1-19](#)
 - configurations, [1-19](#)
 - enhanced capabilities, [1-19](#)
 - system, [1-6](#)
- MU (multiplier underflow) flag, *See*
multiplier status
- MUS (multiplier underflow, sticky)
flag, *See* multiplier sticky status
- MV (multiplier overflow) flag, *See*
multiplier status
- MVS (multiplier overflow, sticky) flag,
See multiplier sticky status
- MZ (multiplier zero) flag, *See*
multiplier status
- N
 - negate instruction, *See* - negate
operator
 - nested calls, [1-16](#)
 - nibble, [8-117](#)
 - NMOD bit, [B-20](#)
 - no operation, *See* NOP instruction
 - NOP instruction, [8-237](#), [8-244](#), [C-43](#)
 - normal accesses, [6-16](#)
 - normal mode, [B-20](#)
 - normal word data, [2-10](#)
 - notation conventions, instruction, [1-22](#)
 - NOT instruction, [8-38](#), [8-213](#), [C-6](#)
 - not predicted branch, [7-18](#)
 - not predicted (NP) branch, *See* NP
option or branch prediction
 - NP (not predicted) option, [7-18](#), [7-37](#),
[7-44](#), [7-50](#), [7-78](#), [8-230](#), [8-232](#),
[8-234](#), [8-235](#), [A-13](#), [C-33](#)
 - numeric formats, [2-16](#)

O

OEN bit, [B-2](#)
 one's complement, *See* - negate
 operator
 ones counting, *See* ONES instruction
 ONES instruction, [3-2](#), [3-30](#), [8-28](#),
[A-3](#), [C-8](#)
 opcodes, [xix](#)
 ALU instructions, [C-7](#), [C-10](#)
 CLU instructions, [C-12](#)
 constructing, [xix](#)
 IALU instructions, [C-24](#), [C-25](#),
[C-29](#), [C-31](#)
 immediate extension instructions,
[C-32](#)
 miscellaneous instructions , [C-42](#)
 multiplier instructions, [C-16](#)
 other instructions, [C-43](#)
 sequencer instructions, [C-35](#)
 sequencer instructions, [C-33](#), [C-36](#)
 shifter instructions, [C-18](#), [C-20](#),
[C-21](#), [C-22](#)
 operands, [2-10](#)
 operation mode, [7-11](#)
 emulation mode, [7-11](#)
 supervisor mode, [7-11](#)
 user mode, [7-11](#)
 OR instruction, [3-2](#), [8-38](#), [8-213](#), [C-6](#)
 output, flag, [B-20](#), [B-21](#)
 overflow, [4-11](#), [8-3](#)
 overflow, *See* AV and MV flags

P

packed data formats, [2-19](#)
 parallel absolute value of difference,
[8-32](#)
 parallel result register, [8-29](#)
 pass, *See* PASS instruction
 PASS instruction, [3-2](#), [3-31](#), [3-32](#),
[8-37](#), [8-78](#), [A-3](#), [A-5](#), [C-6](#), [C-10](#)
 PC register, [7-17](#)
 PC-relative address, [7-17](#)
 PERMUTE instruction, [3-3](#), [3-33](#),
[8-117](#), [8-119](#), [A-6](#)
 pipeline, [1-7](#), [1-14](#), [1-15](#)
 ALU, [1-12](#)
 branches, [7-44](#)
 compute block, [7-29](#)
 IALU, [7-28](#)
 illegal combinations, [7-45](#)
 operations, [7-26](#)
 restrictions, [7-45](#)
 stages, [7-6](#)
 stall, [7-55](#)
 post-modify addressing, [1-12](#), [1-36](#),
[6-13](#), [6-28](#), [6-29](#), [6-31](#), [8-220](#),
[8-221](#), [8-222](#), [8-224](#), [C-31](#)
 post-modify and update, [6-14](#)
 power save mode, [8-240](#)
 predicted branch, [7-18](#)
 pre-modify addressing, [1-12](#), [6-13](#),
[6-29](#), [8-220](#), [8-221](#), [C-29](#)
 pre-modify no update, [6-14](#)
 product information, [xxi](#)
 program fetch, *See* sequencer
 program flow, [7-3](#)

INDEX

program label to address, 7-15
program sequencer, *See* sequencer
PR register, 3-5, 3-19, 6-5, 8-29, 8-32, 8-34
purpose of this manual, xvii
PUTBITS instruction, 2-4, 5-12, 5-13, 5-21, 8-187, 8-189, A-9, C-20
 example, 8-190
 restrictions, 1-38

Q

quad access, 1-18
quad accesses, 1-18, 6-2, 6-16
quad instruction execution, 1-15
quad register, 2-9
quad word data, 6-16

R

RDS instruction, 7-24, 7-78, 8-236, A-13, C-34, C-36
real data, 2-7
reciprocal, *See* RECIPS instruction
reciprocal square root, *See* RSQRTS instruction
RECIPS instruction, 3-2, 3-32, 8-80, A-5, C-10
reduce interrupt to subroutine, *See* RDS instruction

registers
 files, 1-9, 2-1, 2-5
 file syntax summary, 2-13
 saving and restoring, 1-16
 SIMD, 1-41
 width, 2-8, 2-9
 writing to constraints, 1-36
related documents, xxiii
relative addresses, 1-16
relative addresses for relocation, 1-16
reset, software, B-20
resource conflict, 7-55
resources, 1-25
restriction, 1-36
restrictions, 1-26, 1-36, 1-46
 access, 1-43
 combination constraints, 1-24
 compute block, 1-37
 IALU, 1-39
 immediate extension, 1-45
 load, 1-40, 1-43
 memory access, 1-40
 move, 1-40
 parallelism rules, 1-24
 pipeline, 7-45
 sequencer, 1-45
 shifter, 1-38, 1-45
result flag, *See* ALU, multiplier, shifter, IALU status
RETI instruction, 1-31, 1-32, 7-78, 8-234, A-13
RETI register, 7-44, 7-67, B-13, C-33
RETS register, B-13

- return from interrupt, *See* RETI or RTI instructions
- rotate, 8-179
 - IALU, left, 6-42, 8-217, A-10
 - IALU, right, 6-37, 6-42, 8-217, A-10
- rotate, *See* ROT, ROTL, and ROTR instructions
- ROT instruction, 5-21, 8-179, A-9, C-18, C-20
- ROTL instruction, 6-42, 8-217, A-10
- ROTR instruction, 6-37, 6-42, 8-217, A-10
- round bit, 4-14
- rounding, 4-14, 8-51
- Round-to-nearest, 3-9, 4-12
- Round-to-zero, 3-10, 4-13
- RSQRTS instruction, 3-2, 3-32, 8-82, A-5, C-11
- RTI instruction, 7-78, 8-234, A-13, C-34, C-36

- S**
- S (saturation) option, 3-8, 4-11
- S unit, *See* shifter
- saturation, 3-7, 3-8, 4-9, 4-11, 4-12, 8-3, 8-6, 8-20
- scalability and multiprocessing, 1-19
- SCALB instruction, 3-2, 3-32, 8-76, A-5, C-10
- scale, floating-point, *See* SCALB instruction
- scaling factor, block floating-point, *See* BKFTP instruction

- SDAB (short DAB) option, 1-33, 6-18, 6-26, 6-43, 8-222, A-11, C-31
- SDRCON register, B-30
- SEQ (shifter equal zero) condition, *See* shifter conditions
- sequencer, 1-13
 - conditions, C-40
 - conditions, C-41
 - examples, 7-72
 - immediate extensions, 1-15, C-33, C-42
 - instructions, 8-228 to 8-244, C-24, C-33, C-34, C-36
 - instruction summary, 7-76
 - opcodes, C-33, C-35, C-36
 - operations, 7-7
 - quick reference, A-13
 - registers, B-13, B-20, B-21, B-22, B-29, B-30, B-33, B-41
 - restrictions, 1-45
 - status, 7-45
- SE (sign extended) option, 5-15
- SE (sign extend) option, 8-183
- set bit, 8-192
- set bits, 5-7, 5-21, 8-192, A-9, C-22, C-23
- SFREG register, 8-239, B-22
 - See also* static condition flag
- SFx (static flag) condition, 3-15, 4-21, 6-13, 7-78, 8-239, A-13, C-34
- shift, *See* ASHIFT, LSHIFT, ASHIFTR, and LSHIFTR instructions

INDEX

- shifter, 1-11
 - condition codes, C-39
 - conditions, 5-16
 - instructions, 8-175 to 8-199, C-18, C-19, C-20, C-22
 - instruction summary, 5-19
 - opcodes, C-18, C-20, C-21, C-22
 - operations, 5-3
 - options, 5-14
 - quick reference, A-8
 - restrictions, 1-38
 - static flags, 5-17
 - status, 5-15, 5-16, B-2
 - status flags, B-2
- shifter examples, 5-17
- short word data, 2-10, 2-19
- sideways sum, *See* SUM instruction
- sign, copy, *See* COPYSIGN instruction
- signed data, 3-8, 4-10
- signed operation, 3-7, 4-9, 6-7
- signed saturation, 3-8, 8-3, 8-6, 8-20
- SIMD, *See* Single-Instruction, Multiple-Data
- Single-Instruction, Multiple-Data, 2-8
 - execution, 2-8
 - registers, 8-226, B-7
- Single-Instruction, Single-Data, 2-8
- single-precision, 2-16
- Single-processor system, 1-4
- single register, 2-9
- SISD, *See* Single-Instruction, Single-Data
- SLT (shifter less than zero) conditions, *See* shifter conditions
- SNGL instruction, 3-2, 3-32, 8-70, A-5, C-11
- SN (shifter negative) flag, *See* shifter status
- software reset, B-20
- SPVCMD bit, B-22
- SQCTL register, 1-33, 7-8, 7-10, 7-11, 7-34, B-20
- SQSTAT register, 1-33, 7-8, 7-10, B-21
- stall, 1-14, 1-25, 7-55
 - bus conflict, 7-59
 - compute block dependency, 7-56, 7-62
 - external memory dependency, 7-64
 - IALU load dependency, 7-63, 7-64
 - instruction pipeline, 7-30
 - us request, 7-59
- static condition flag, 8-239
 - See also* SFREG register
- static flags, 3-15, 4-21, 6-13, 7-78, 8-239, A-13, B-22, C-34
- Static Superscalar, 1-7, 1-15, 1-24
- status
 - ALU, 3-11, 4-18, 6-10, B-2
 - CLU, 3-27
 - compute block, 7-45
 - multiplier, B-2
 - shifter, 5-15, B-2
- sticky status, 7-29
 - ALU, 3-12, 4-19, B-2
 - CLU, 3-27
 - multiplier, B-3

- store, 1-12, 1-30, 1-31, 8-221, 8-224
 - opcodes, C-28, C-29, C-31
 - register, 8-224, C-28, C-29, C-31
 - stray pointers, 1-26
 - subroutines, 7-3
 - subtract instruction, 8-3, 8-36, 8-51, 8-89, 8-202
 - See also* - operator
 - with borrow, 8-6
 - with borrow, integer, 8-204
 - SUM instruction, 2-4, 3-2, 3-19, 3-20, 3-30, 8-26, 8-34, A-3, C-8, C-9
 - supervisor mode, B-20
 - support, technical or customer, xx
 - SWRST bit, B-20
 - SYSCON register, B-29
 - system
 - development enhancements, 1-4
 - multi-processor, 1-6
 - on-a-chip (SOC), 1-1
 - single-processor, 1-4
 - SZ (shifter zero) flag, *See* shifter status
- T**
- T (truncation) option, 3-9, 4-12
 - technical or customer support, xx
 - technical support, xx
 - test bits, *See* BITEST instruction
 - timer, 7-11, 7-22, 7-23
 - timer run, B-20
 - TMAX instruction, 3-3, 3-23, 3-25, 3-33, 8-92, A-5, C-12, C-13, C-14
 - TMRxRN bits, B-20
 - toggle bit, *See* BTGL instruction
- trap, *See* TRAP and EMUTRAP
 - instructions
 - TRAP instruction, 7-79, 8-237, 8-242, A-14, C-42, C-43
 - trellis diagram, 3-25
 - trellis maximum, *See* TMAX
 - instruction
 - trellis overflow, *See* TROV and TRSOV bits
 - TROV (trellis overflow) bit, *See* CLU status
 - TROV (trellis overflow) flag, *See* CLU status
 - TRSOV (trellis overflow, sticky) bit, *See* CLU status
 - TRUE flag, 7-45
 - truncation, *See* rounding and T option
 - turbo-code algorithms, 1-9, 3-21
 - turbo decoding algorithm, 3-24
 - two's complement, *See* - operator
- U**
- U (unsigned/signed) option, 3-8, 4-10
 - UEN bit, B-2
 - unbiasing, 8-87
 - unconditional execution, 1-23
 - underflow, 8-3
 - underflow, *See* AZ, MZ, SZ, JZ, or KZ flag
 - universal registers, 2-2, 6-35, 8-220, 8-221, 8-226
 - unmasked interrupt, 7-21
 - unsigned data, 3-8, 4-10, 6-8
 - unsigned operation, 3-7, 6-7

INDEX

unsigned saturation, [3-8](#), [8-3](#), [8-4](#), [8-6](#),
[8-7](#), [8-20](#)

Ureg (universal registers), [2-5](#)

V

Viterbi algorithm, [1-9](#), [3-21](#), [3-24](#)

Viterbi maximum, *See* VMAX
instruction

Viterbi Maximum/Minimum, [8-17](#)

Viterbi minimum, *See* VMIN
instruction

VMAX instruction, [2-4](#), [3-2](#), [3-3](#), [3-19](#),
[8-18](#), [C-6](#)

VMIN instruction, [2-4](#), [3-19](#), [3-30](#),
[8-17](#), [A-3](#), [C-6](#)

X

X (extended output range) option, [3-9](#),
[8-10](#)

X option, [8-10](#)

XOR instruction, [3-2](#), [8-38](#), [8-213](#), [C-6](#)

XSCFx bits, [B-22](#)

XSTAT register, [2-4](#), [8-196](#)

X/YSTAT register
restrictions, [1-38](#)

Y

YSCFx bits, [B-22](#)

YSTAT register, [2-4](#), [8-196](#)

Z

ZF (zero filled) option, [5-15](#), [8-183](#)

Z (return zero) option, [3-10](#), [8-14](#)