

ADSP-TS201 TigerSHARC® Processor Programming Reference

Revision 1.1, April 2005

Part Number
82-000810-01

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	xxi
Intended Audience	xxi
Manual Contents	xxii
What's New in This Manual	xxiv
Technical or Customer Support	xxiv
Supported Processors	xxv
Product Information	xxvi
MyAnalog.com	xxvi
Processor Product Information	xxvi
Related Documents	xxvii
Online Technical Documentation	xxviii
Accessing Documentation From VisualDSP++	xxix
Accessing Documentation From Windows	xxix
Accessing Documentation From the Web	xxx

CONTENTS

Printed Manuals	xxx
VisualDSP++ Documentation Set	xxx
Hardware Tools Manuals	xxx
Processor Manuals	xxx
Data Sheets	xxx
Conventions	xxxii

INTRODUCTION

Processor Architecture	1-7
Compute Blocks	1-10
Arithmetic Logic Unit (ALU)	1-10
Communications Logic Unit (CLU)	1-12
Multiply Accumulator (Multiplier)	1-12
Bit Wise Barrel Shifter (Shifter)	1-13
Integer Arithmetic Logic Unit (IALU)	1-13
Program Sequencer	1-15
Quad Instruction Execution	1-17
Relative Addresses for Relocation	1-18
Nested Call and Interrupt	1-18
Context Switching	1-18
Internal Memory and Buses	1-19
Internal Buses	1-19
Internal Transfer	1-20
Data Accesses	1-21
Quad Data Access	1-21

Booting	1-21
Scalability and Multiprocessing	1-22
Emulation and Test Support	1-22
Instruction Line Syntax and Structure	1-23
Instruction Notation Conventions	1-24
Unconditional Execution Support	1-26
Conditional Execution Support	1-27
Instruction Parallelism Rules	1-27
General Restriction	1-38
Compute Block Instruction Restrictions	1-39
IALU Instruction Restrictions	1-43
Sequencer Instruction Restrictions	1-48

COMPUTE BLOCK REGISTERS

Register File Registers	2-6
Compute Block Selection	2-7
Register Width Selection	2-9
Operand Data Type Selection	2-10
Registers File Syntax Summary	2-12
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	3-4
ALU Instruction Options	3-6
Signed/Unsigned Option	3-7
Saturation Option	3-8
Extension (ABS) Option	3-9
Truncation Option	3-9
Return Zero (MAX/MIN) Option	3-10
Fractional/Integer Option	3-10
No Flag Update Option	3-11
ALU Execution Status	3-11
AZ – ALU Zero	3-13
AN – ALU Negative	3-13
AV – ALU Overflow	3-14
AI – ALU Invalid	3-14
AC – ALU Carry	3-15
ALU Execution Conditions	3-15
ALU Static Flags	3-16
ALU Examples	3-16
Example Parallel Addition of Byte Data	3-18
Example Sideways Sum of Byte Data	3-19
Example Parallel Result (PR) Register Usage	3-20
ALU Instruction Summary	3-22

CLU

CLU Operations 4-4

 TMAX Function 4-6

 Trellis Function 4-8

 Trellis Function of the Form
 $STRsd = TMAX(TRmd + Rmq_h, TRnd + Rmq_l)$; 4-10

 Trellis Function of the Form
 $STRsd = TMAX(TRmd - Rmq_h, TRnd - Rmq_l)$; 4-13

 Trellis Function of the Form
 $SRs = TMAX(TRm, TRn)$; 4-16

 Trellis Function of the Form
 $STRsd = MAX(TRmd + Rmq_h, TRnd + Rmq_l)$; 4-18

 Trellis Function of the Form
 $STRsd = MAX(TRmd - Rmq_h, TRnd - Rmq_l)$; 4-20

 Despread Function 4-22

 Despread Function of the Form
 $TRs += DESPREAD (Rmq, THRD)$; 4-25

 Despread Function of the Form
 $Rs = TRs, TRs = DESPREAD (Rmq, THRD)$; 4-26

 Despread Function of the Form
 $Rsd = TRsd, TRsd = DESPREAD (Rmq, THRD)$; 4-28

 Cross Correlations Function 4-30

 Add/Compare/Select Function 4-39

 CLU Instruction Options 4-41

 CLU Execution Status 4-42

CONTENTS

CLU Examples	4-43
CLU Instruction Summary	4-46

MULTIPLIER

Multiplier Operations	5-5
Multiplier Instruction Options	5-11
Signed/Unsigned Option	5-12
Fractional/Integer Option	5-13
Saturation Option	5-13
Truncation Option	5-15
Clear/Round Option	5-17
Complex Conjugate Option	5-19
No Flag Update Option	5-19
Multiplier Result Overflow (MR4) Register	5-19
Multiplier Execution Status	5-20
Multiplier Execution Conditions	5-22
Multiplier Static Flags	5-23
Multiplier Examples	5-24
Multiplier Instruction Summary	5-26

SHIFTER

Shifter Operations	6-4
Logical Shift Operation	6-6
Arithmetic Shift Operation	6-7
Bit Manipulation Operations	6-8

Bit Field Manipulation Operations	6-9
Bit Field Analysis Operations	6-12
Bit Stream Manipulation Operations	6-12
Shifter Instruction Options	6-17
Sign-Extended Option	6-18
Zero-Filled Option	6-18
No Flag Update Option	6-18
Shifter Execution Status	6-18
Shifter Execution Conditions	6-19
Shifter Static Flags	6-20
Shifter Examples	6-20
Shifter Instruction Summary	6-22

IALU

IALU Operations	7-6
IALU Integer (Arithmetic and Logic) Operations	7-6
IALU Instruction Options	7-7
IALU Data Types	7-8
Signed/Unsigned Option	7-8
Circular Buffer Option	7-8
Bit Reverse Option	7-9
Computed Jump Option	7-10
No Flag Update Option	7-10

CONTENTS

IALU Execution Status	7-10
JZ/KZ–IALU Zero	7-11
JN/KN–IALU Negative	7-12
JV/KV–IALU Overflow	7-12
JC/KC–IALU Carry	7-12
IALU Execution Conditions	7-13
IALU Static Flags	7-14
IALU Load, Store, and Transfer Operations	7-14
Direct and Indirect Addressing	7-15
Normal, Merged, and Broadcast Memory Accesses	7-17
Data Alignment Buffer (DAB) Accesses	7-26
Data Alignment Buffer (DAB) Accesses With Offset	7-30
Circular Buffer Addressing	7-33
Bit Reverse Addressing	7-38
Universal Register Transfer Operations	7-42
Immediate Extension Operations	7-42
IALU Examples	7-43
IALU Instruction Summary	7-47

PROGRAM SEQUENCER

Sequencer Operations	8-8
Conditional Execution	8-14
Branching Execution	8-19
Looping Execution	8-22
Interrupting Execution	8-26

Instruction Pipeline Operations	8-34
Instruction Alignment Buffer (IAB)	8-39
Branch Target Buffer (BTB)	8-42
Conditional Branch Effects on Pipeline	8-53
Dependency and Resource Effects on Pipeline	8-64
Stall From Compute Block Dependency	8-77
Stall From Bus Conflict	8-80
Stall From Compute Block Load Dependency	8-83
Stall From IALU Load Dependency	8-84
Stall From Load (From External Memory) Dependency	8-84
Stall From Conditional IALU Load Dependency	8-85
Interrupt Effects on Pipeline	8-86
Interrupt During Conditional Instruction	8-88
Interrupt During Interrupt Disable Instruction	8-90
Exception Effects on Pipeline	8-90
Sequencer Examples	8-92
Sequencer Instruction Summary	8-97

MEMORY AND BUSES

Memory Block Physical Structure	9-5
Prefetch Buffer	9-8
Read Buffer	9-10
Cache	9-11
Copyback Buffer	9-11

CONTENTS

Buffer/Cache Hit	9-12
Memory Block Terms, Sizes, and Addressing	9-13
Memory Block Logical Organization	9-14
Block	9-15
Half-Block	9-15
Sub-Array	9-15
Page	9-16
Memory Block Accesses	9-16
Activate	9-18
Precharge	9-18
Refresh	9-19
Memory System Controls and Status	9-19
Cache Operation	9-25
Data Access on Read	9-30
Data Access on Write	9-32
Memory Bus Arbitration	9-33
Memory Pipeline	9-35
Cache Miss Transaction	9-38
Cache Miss (With No Buffer Hit) Transaction	9-38
Prefetch Sequence	9-41
Prefetch Sequence Interrupted	9-43
Memory Access Penalty Summary	9-44

Memory Programming Guidelines	9-46
Sequential Access Policy	9-48
Local Access Policy	9-49
Cache Usage Policy	9-50
Cache Enable (After Reset and Boot)	9-50
Cache Disable	9-50
Read and Write Cacheability	9-51
Instruction Caching	9-51
Double Sequence Flow	9-52
Block Orientation	9-52
Memory Access Examples	9-53
Memory System Command Summary	9-56
Memory System Commands	9-58
Refresh Rate Select	9-60
Cache Enable	9-62
Cache Disable	9-63
Set Bus/Cacheability (for Bus Transactions)	9-64
Cache Lock Start	9-68
Cache Lock End	9-69
Cache Initialize (From Memory)	9-70
Cache Copyback (to Memory)	9-74
Cache Invalidate	9-78

CONTENTS

INSTRUCTION SET

ALU Instructions	10-2
Add/Subtract	10-3
Add/Subtract With Carry/Borrow	10-6
Add/Subtract With Divide by Two	10-9
Absolute Value/Absolute Value of Sum or Difference	10-11
Two's Complement	10-14
Maximum/Minimum	10-15
Viterbi Maximum/Minimum	10-18
Increment/Decrement	10-21
Compare	10-23
Clip	10-25
Sideways Sum	10-27
Ones Counting	10-29
Load/Transfer PR (Parallel Result) Register	10-30
Bit FIFO Increment	10-31
Absolute Value With Parallel Accumulate	10-33
Sideways Sum With Parallel Accumulate	10-35
Add/Subtract (Dual Operation)	10-37
Pass	10-38
Logical AND/AND NOT/OR/XOR/NOT	10-39
Expand	10-41
Compact	10-46
Merge	10-50

Permute (Byte Word) 10-52

Permute (Short Word) 10-54

Add/Subtract (Floating-Point) 10-56

Add/Subtract With Divide by Two (Floating-Point) 10-58

Maximum/Minimum (Floating-Point) 10-60

Absolute Value/
 Absolute Value of Sum or Difference (Floating-Point) 10-62

Complement Sign (Floating-Point) 10-65

Compare (Floating-Point) 10-67

Floating- to Fixed-Point Conversion 10-69

Fixed- to Floating-Point Conversion 10-71

Floating-Point Normal to Extended Word Conversion 10-73

Floating-Point Extended to Normal Word Conversion 10-75

Clip (Floating-Point) 10-77

Copysign (Floating-Point) 10-79

Scale (Floating-Point) 10-81

Pass (Floating-Point) 10-83

Reciprocal (Floating-Point) 10-85

Reciprocal Square Root (Floating-Point) 10-87

Mantissa (Floating-Point) 10-90

Logarithm (Floating-Point) 10-92

Add/Subtract (Dual Operation, Floating-Point) 10-94

CLU Instructions 10-96

Trellis Maximum (CLU) 10-97

Maximum (CLU) 10-98

CONTENTS

Transfer TR (Trellis), THR (Trellis History), or CMCTL (Communications Control) Registers	10-99
Despread With Transfer Trellis Register (Dual Operation, CLU)	10-101
Cross Correlations With Transfer Trellis Register (Dual Operation, CLU)	10-103
Add/Compare/Select (CLU)	10-105
Multiplier Instructions	10-107
Multiply (Normal Word)	10-108
Multiply-Accumulate (Normal Word)	10-111
Multiply-Accumulate With Transfer MR Register (Dual Operation, Normal Word)	10-117
Multiply (Quad-Short Word)	10-124
Multiply-Accumulate (Quad-Short Word)	10-127
Multiply-Accumulate With Transfer MR Register (Dual Operation, Quad-Short Word)	10-133
Complex Multiply-Accumulate (Short Word)	10-139
Complex Multiply-Accumulate With Transfer MR Register (Dual Operation, Short Word)	10-143
Multiply (Floating-Point, Normal/Extended Word)	10-150
Load/Transfer MR (Multiplier Result) Register	10-152
Extract Words From MR (Multiplier Result) Register	10-160
Compact MR (Multiplier Result) Register	10-165
Shifter Instructions	10-170
Arithmetic/Logical Shift	10-171
Rotate	10-174

Field Extract	10-176
Field Deposit	10-178
Field/Bit Mask	10-180
Get Bits	10-182
Put Bits	10-184
Bit Test	10-187
Bit Clear/Set/Toggle	10-188
Extract Leading Zeros	10-190
Extract Exponent	10-191
Block Floating-Point	10-192
Load/Transfer Bit FIFO Temporary (BFOTMP) Register	10-194
Load/Transfer Compute Block Status (X/YSTAT) Registers	10-195
Load TR (Trellis), TRH (Trellis History), or CMCTL (Communications Control) Registers (CLU)	10-196
IALU (Integer) Instructions	10-199
Add/Subtract (Integer)	10-201
Add/Subtract With Carry/Borrow (Integer)	10-203
Add/Subtract With Divide by Two (Integer)	10-205
Compare (Integer)	10-207
Maximum/Minimum (Integer)	10-209
Absolute Value (Integer)	10-211
Logical AND/AND NOT/OR/XOR/NOT (Integer)	10-213
Arithmetic Shift/Logical Shift (Integer)	10-215
Left Rotate/Right Rotate (Integer)	10-217
IALU (Load/Store/Transfer) Instructions	10-218

Load Ureg (Universal) Register (Data Addressing)	10-220
Store Ureg (Universal) Register (Data Addressing)	10-222
Load Dreg (Data) Register With DAB/SDAB (Data Addressing)	10-223
Load Dreg (Data) Register With DAB Offset (Data Addressing)	10-225
Store Dreg (Data) Register (Data Addressing)	10-227
Transfer Ureg (Universal) Register	10-229
Sequencer Instructions	10-232
Jump/Call	10-234
Computed Jump/Call	10-236
Return (From Interrupt)	10-238
Reduce (Interrupt to Subroutine)	10-240
If – Do (Conditional Execution)	10-241
If – Else (Conditional Sequencing and Execution)	10-242
Load Condition Into Static Condition Flag	10-243
Idle	10-244
BTB Enable/Disable	10-245
BTB Lock/End Lock	10-246
BTB Invalid	10-247
Trap	10-248
Emulator Trap	10-249
No Operation	10-250

QUICK REFERENCE

ALU Quick Reference	A-2
CLU Quick Reference	A-6
Multiplier Quick Reference	A-7
Shifter Quick Reference	A-9
IALU Quick Reference	A-11
Sequencer Quick Reference	A-14
Memory/Cache Quick Reference	A-16

REGISTER/BIT DEFINITIONS

INSTRUCTION DECODE

Instruction Structure	C-1
Compute Block Instruction Format	C-3
ALU Instructions	C-5
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	C-10
CLU Instructions	C-12
Multiplier Instructions	C-15

Shifter Instructions	C-19
Single Normal Word Operands and Single Register	C-20
Single Long or Dual Normal Word Operands and Dual Register	C-20
Short or Bte Operands and Single or Dual Registers	C-22
Single Operand	C-23
CLU Registers	C-25
IALU Instructions	C-26
IALU (Integer) Instruction Format	C-26
IALU Move Instruction Format	C-29
IALU Load Data Instruction Format	C-31
IALU Load/Store Instruction Format	C-32
IALU Immediate Extension Format	C-37
Sequencer Instruction Format	C-38
Sequencer Direct Jump/Call Instruction Format	C-38
Sequencer Indirect Jump Instruction Format	C-40
Condition Codes	C-43
Compute Block Conditions	C-43
IALU Conditions	C-44
Sequencer and External Conditions	C-45
Sequencer Immediate Extension Format	C-45
Miscellaneous Instruction Format	C-46

GLOSSARY

INDEX

PREFACE

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

Purpose of This Manual

The *ADSP-TS201 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) operation, includes ALU instruction examples, and provides the ALU instruction summary.
- Chapter 4, [CLU](#)
Provides a description of the communications logic unit (CLU) operation, includes CLU instruction examples, and provides the CLU instruction summary.
- Chapter 5, [Multiplier](#)
Provides a description of the multiply-accumulator (multiplier) operation, includes multiplier instruction examples, and provides the multiplier instruction summary.
- Chapter 6, [Shifter](#)
Provides a description of the bit wise, barrel shifter (shifter) operation, includes shifter instruction examples, and provides the shifter instruction summary.
- Chapter 7, [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 8, [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 9, [Memory and Buses](#)
Provides a description of the internal embedded DRAM memory system (memory blocks, buffers, and cache) and internal busses. This chapter also includes a memory system command summary.
- Chapter 10, [Instruction Set](#)
Describes the ADSP-TS201 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-TS201 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-TS201 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-TS201 TigerSHARC Processor Hardware Reference*.

What's New in This Manual

Revision 1.1 of the *ADSP-TS201 TigerSHARC Processor Programming Reference* corrects all known document errata issues. The changes include updates to the e-mail contact addresses, the European fax number, and the FTP address. Also, the `SQSTAT` register's bit definitions have been corrected in the `DEFTS201.H` file. (See [Listing B-1 on page B-1.](#))

Technical or Customer Support

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

- Visit the Embedded Processing and DSP products Web site at <http://www.analog.com/processors/technicalSupport>
- E-mail tools questions to dsptools.support@analog.com
- E-mail processor questions to processor.support@analog.com
processor.europe@analog.com
processor.china@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-21263, ADSP-21266, ADSP-21267,
ADSP-21362, ADSP-21363, ADSP-21364, ADSP-21365,
ADSP-21366, ADSP-21367, ADSP-21368, and ADSP-21369

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, ADSP-BF533, ADSP-BF534,
ADSP-BF535, ADSP-BF536, ADSP-BF537, ADSP-BF538,
ADSP-BF539, ADSP-BF561, and ADSP-BF566

Product Information

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

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

MyAnalog.com

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

Registration

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

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

Processor Product Information

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

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

- E-mail questions or requests for information to
processor.support@analog.com
processor.europe@analog.com
processor.china@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 (or ftp 137.71.25.69)
ftp://ftp.analog.com

Related Documents

The following publications that describe the ADSP-TS201 TigerSHARC processor (and related processors) can be ordered from any Analog Devices sales office:

- *ADSP-TS201S TigerSHARC Embedded Processor Data Sheet*
- *ADSP-TS202S TigerSHARC Embedded Processor Data Sheet*
- *ADSP-TS203S TigerSHARC Embedded Processor Data Sheet*
- *ADSP-TS201 TigerSHARC Processor Hardware Reference*
- *ADSP-TS201 TigerSHARC Processor Programming Reference*

Product Information

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*
- *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/resources/technicalLibrary>

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.

Product Information

Help system files (.CHM) are located in the Help folder, and .PDF files are located in the Docs folder of your VisualDSP++ installation CD-ROM. The Docs folder also contains the Dinkum Abridged C++ library and the FlexLM network license manager software documentation.

Using Windows Explorer

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

Using the Windows Start Button

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

Accessing Documentation From the Web

Download manuals at the following Web site:

<http://www.analog.com/processors/resources/technicalLibrary/manuals>

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

Printed Manuals

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

VisualDSP++ Documentation Set

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

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

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-TS201 TigerSHARC Processor Programming Reference* describes the ADSP-TS201 TigerSHARC processor architecture and instruction set. These descriptions provide the information required for programming TigerSHARC processor systems. This chapter introduces programming concepts for the processor with the following information:

- [“Processor Architecture” on page 1-7](#)
- [“Instruction Line Syntax and Structure” on page 1-23](#)
- [“Instruction Parallelism Rules” on page 1-27](#)

The ADSP-TS201 processor is a 128-bit, high performance TigerSHARC processor. The ADSP-TS201 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 ADSP-TS201 processor maintains a ‘system-on-chip’ scalable computing design philosophy, including 24M bit of on-chip DRAM, six 4K word caches (one per memory block), integrated I/O peripherals, a host processor interface, DMA controllers, LVDS link ports, and shared bus connectivity for glueless multiprocessing.

In addition to providing unprecedented performance in DSP applications in raw MFLOPS and MIPS, the ADSP-TS201 processor boosts performance measures such as MFLOPS/Watt and MFLOPS/square inch in multiprocessing applications.

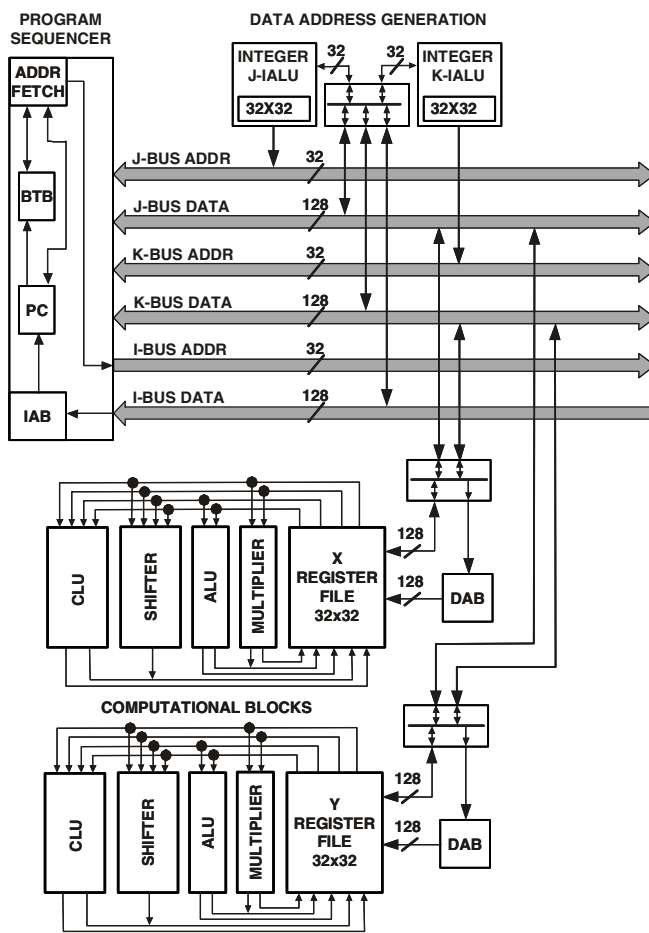


Figure 1-1. ADSP-TS201 TigerSHARC Processor Core Diagram

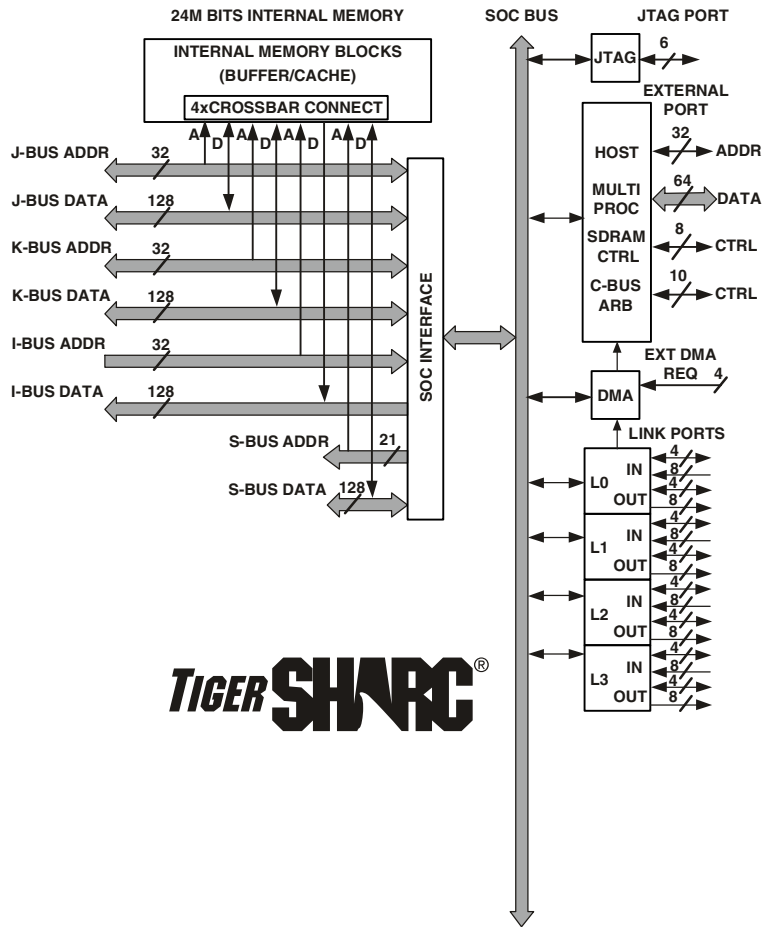


Figure 1-2. ADSP-TS201 TigerSHARC Processor Peripherals Diagram

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, CLU, shifter, and a 32-word register file
- Dual integer ALUs: J and K – each containing a 32-bit IALU and 32-word register file
- 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 internal memory and the rest of the processor core (compute blocks, IALUs, program sequencer, and SOC interface)
- A 128-bit bus providing high bandwidth connectivity between internal memory and external I/O peripherals (DMA, external port, and link ports)
- External port interface including the host interface, SDRAM controller, static pipelined interface, four DMA channels, four LVDS link ports (each with two DMA channels), and multiprocessing support
- 24M bits of internal memory organized as six 4M bit blocks—each block containing 128K words x 32 bits; each block connects to the crossbar through its own buffers and a 128K bit, 4-way set associative cache
- 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, SDRAM, and I/O devices.

Figure 1-3 illustrates a typical single processor system. A multiprocessor system is illustrated in Figure 1-4 and is discussed later in “Scalability and Multiprocessing” on page 1-22.

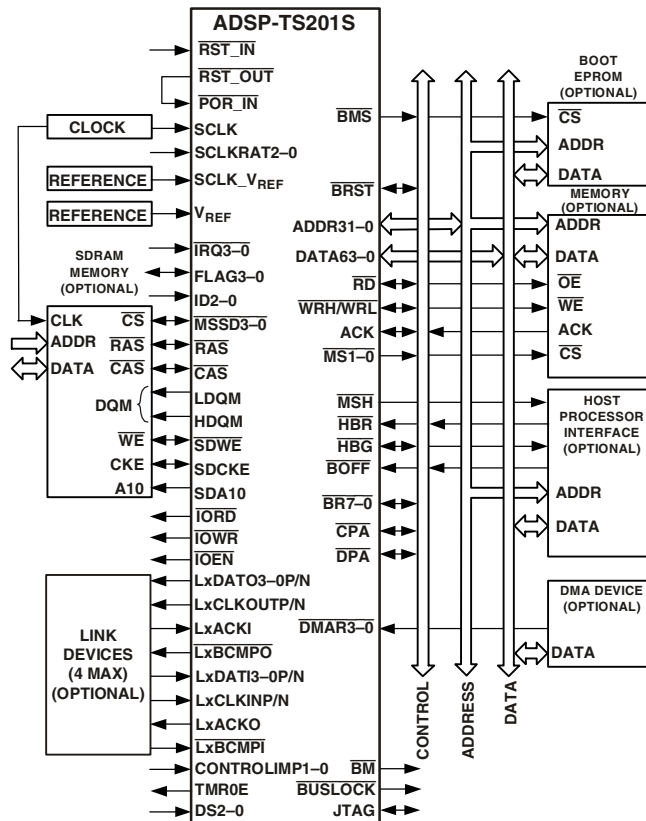


Figure 1-3. Single Processor Configuration

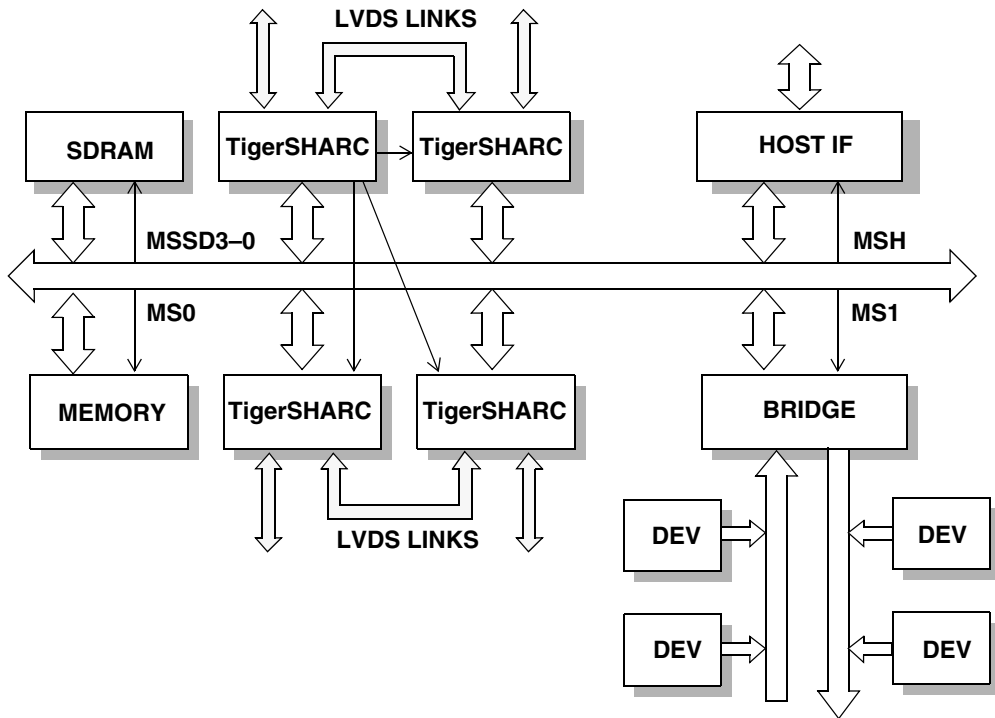


Figure 1-4. Multiprocessing Cluster Configuration

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

- Support of IEEE floating-point formats
- IEEE Standard 1149.1 Joint Test Action Group (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-bit (IEEE Standard 754/854) and 40-bit floating-point and 8-, 16-, 32-, and 64-bit fixed-point native data types
- Large address space
- Immediate address modify fields
- Easily supported relocatable code and data
- Fast save and restore of processor registers onto internal memory stacks

Processor Architecture

As shown in [Figure 1-1 on page 1-2](#) and [Figure 1-2 on page 1-3](#), the processor architecture consists of two divisions: the processor 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 processor 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-TS201 TigerSHARC Processor Hardware Reference*.

This section describes:

- [“Compute Blocks” on page 1-10](#)
- [“Integer Arithmetic Logic Unit \(IALU\)” on page 1-13](#)
- [“Program Sequencer” on page 1-15](#)

Processor Architecture

- [“Internal Memory and Buses” on page 1-19](#)
- [“Booting” on page 1-21](#)
- [“Scalability and Multiprocessing” on page 1-22g](#)
- [“Emulation and Test Support” on page 1-22](#)

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 24M bit internal memory is divided into six 128K word memory blocks. Each of the four internal address/data bus pairs connect to all of the six memory blocks via a crossbar interface. The six memory blocks support up to four accesses every cycle where each memory block can perform a 128-bit access in a cycle. Each block's cache and prefetch mechanism improve access performance of internal memory (embedded DRAM).

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 ADSP-TS201 processor uses a pipelined external bus with programmable pipeline depth for interprocessor communications and for Synchronous Flow-through SRAM (SSRAM) and SDRAM.

The four LVDS link ports support point-to-point high bandwidth data transfers. Each link port supports full-duplex communication.

The processor operates with a two cycle arithmetic pipeline. The branch pipeline is four to ten cycles. A branch target buffer (BTB) is implemented to reduce branch delay.

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 a 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.

The processing core of the ADSP-TS201 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 identical computation units support floating-point as well as fixed-point arithmetic. These 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 four independent computation units—an ALU, a CLU, 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.

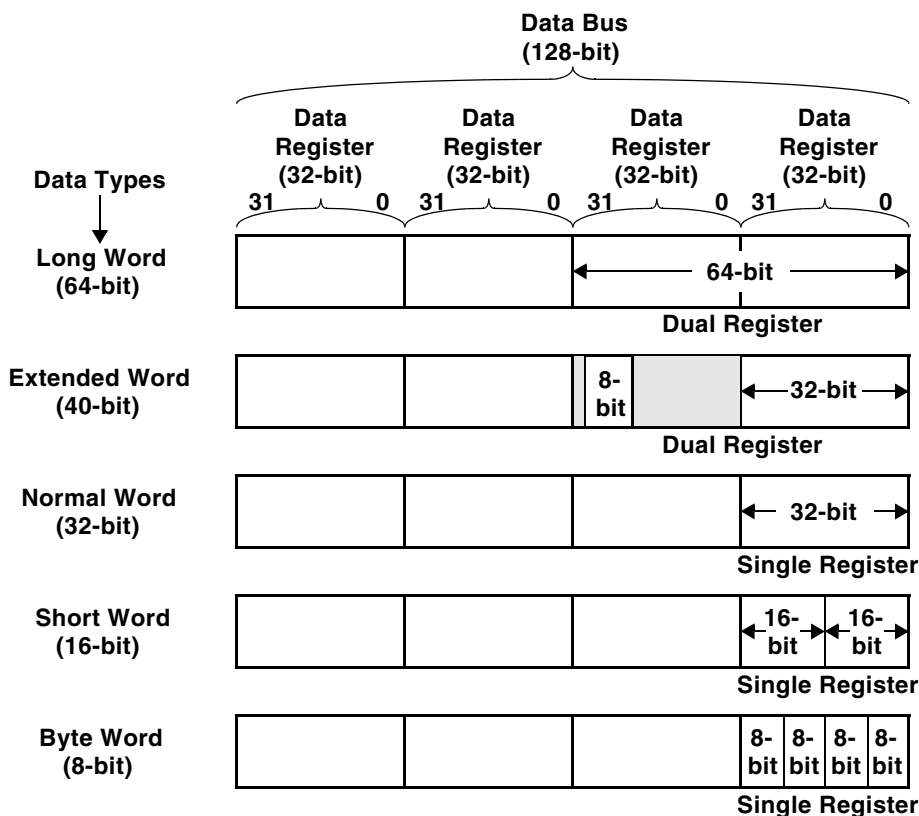
These formats are listed here and shown in [Figure 1-5](#):

- **Fixed-point format**
These include 64-bit long word, 32-bit normal word, 32-bit complex (16-bit real and 16-bit imaginary), 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 least significant bits (LSBs) of mantissa for greater accuracy.

Each compute block has a general-purpose, multiport, 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-, double-, 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.

Figure 1-5. Word Format Definitions^{1,2}

- 1 The TigerSHARC processor internal data buses are 128 bits (one quad word) wide. In a quad word, the processor can move 16 byte words, 8 short words, 4 normal words, or 2 long words over the bus at the same time.
- 2 For details on the numeric formats (fixed- and floating-point data formats) that can be used with each of the data types in [Figure 1-5](#), see “Numeric Formats” on page 2-16.

Communications Logic Unit (CLU)

On the ADSP-TS201 processor, there is a special purpose compute unit called 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
- Cross correlations used for path search

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

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 formats. 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 ADSP-TS201 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 LSBs of the input operand represent 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 5-1](#).

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 BFOTMP (which is internal to the shifter)
- Bit FIFO operations to support bit streams with fields of varying length
- Support for ADSP-21000 DSP family compatible fixed-point/floating-point conversion operations (such as exponent extract, number of leading ones or zeros)

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

Integer Arithmetic Logic Unit (IALU)

The IALUs can execute standard standalone ALU operations on IALU register files. The IALUs also execute register load, store, and transfer operations, providing memory addresses when data is transferred between memory and registers. The processor has dual IALUs (the J-IALU and the K-IALU) that enable simultaneous addresses for two transactions of up to 128 bits in parallel. The IALUs allow compute operations to execute with maximum efficiency because the computation units can be devoted exclusively to processing data.

Processor Architecture

Each IALU has a multiport, 32-word register file. All IALU calculations are executed in a single cycle. 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 ADSP-TS201 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. Normally, there are no dependency delays between IALU instructions, but if there are, four cycles of latency can occur.

For more information on the IALUs, see [“IALU” on page 7-1](#).


Program Sequencer

The program sequencer supplies instruction addresses to memory and, together with the IALUs, allows compute 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 two responsibilities of the sequencer are to decode fetched instructions—separating the instruction slots of the instruction line and sending each instruction to its execution unit (compute blocks, IALUs, or sequencer)—and to control the program flow. The sequencer’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.

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.

The ADSP-TS201 processor achieves its fast execution rate by means of a ten-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 compu-

tation instructions per compute block can be issued in each cycle, instructing the ALU, multiplier, or shifter to perform independent, simultaneous operations.

The ADSP-TS201 processor has four general-purpose external interrupts, $\overline{\text{IRQ3-0}}$. 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 ADSP-TS201 processor, can be compensated for by using 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 five to nine 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.

For more information on the sequencer, BTB, and immediate extensions, see [“Program Sequencer” on page 8-1](#).

Quad Instruction Execution

The ADSP-TS201 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 archi-

Processor Architecture

ecture because the instructions executed in each cycle are specified in the 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 ADSP-TS201 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 ADSP-TS201 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 ADSP-TS201 processor to dispense with on-chip PC stack or alternate registers for register files or status registers.

Internal Memory and Buses

The on-chip memory consists of six blocks of 4M bits each. Each block is 128K words, thus providing high bandwidth sufficient to support both computation units, the instruction stream and external I/O, even in very intensive operations. The ADSP-TS201 processor provides access to program, two data operands, and a system access (over the SOC bus) to different memory blocks without memory or bus constraints. The memory blocks can store instructions and data interchangeably.

Each memory block is organized as 128K 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. The six memory blocks 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-TS201 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 the DSP and real-time systems.

Internal Buses

The processor core has three buses (I-bus, J-bus, and K-bus), each connected to all of the internal memory blocks via a crossbar interface. 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

Processor Architecture

elements use these SOC bus and S-bus to access memory. Only one access to each memory block is allowed in each cycle, so if the application accesses a different memory segment for each purpose (instruction fetch, load/store J-IALU and K-IALU instructions and external accesses), all transactions can be executed with no stalls.

A separate system on chip (SOC) bus connects the external interfaces (external port, DMA, link ports, JTAG port, and others) to the memory system via the SOC interface and S-bus. This bus has a 128-bit wide data bus and a 32-bit wide address bus. It works at half the processor core clock rate. All data transferred between internal memory or core, and external port (cluster bus, link port, and others) passes through this bus.

Most registers of the ADSP-TS201 processor are classified as universal registers (*Ureg*). Instructions are provided for transferring data between any two *Ureg* registers, 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. All registers can be accessed by register-move instructions or by external master access (another ADSP-TS201 on the same cluster bus or host), but only the core registers can be accessed by load/store instructions or load-immediate instructions.

Internal Transfer

Most registers of the ADSP-TS201 processor are classified as universal registers (*Ureg*). Instructions are provided for transferring data between any two *Ureg* registers, 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 2-6](#).

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.

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. This data throughput leaves alternate cycles for I/O to the memories. Using data access this way is beneficial in applications with high I/O requirements because it allows the I/O to occur without degrading core processor performance.

Bootling

The internal memory of the ADSP-TS201 processor can be loaded from an 8-bit EPROM using a boot mechanism at system powerup. The processor can also be booted using an external master or through one of the link ports. Selection of the boot source is controlled by external pins. For information on booting the processor, see the *ADSP-TS201 TigerSHARC Processor Hardware Reference* or the processor data sheet.

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

Scalability and Multiprocessing

The ADSP-TS201 processor, like the ADSP-TS101 processor, 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 toroid, mesh, tree, crossbar, or others. The user can provide a personal interconnect method or use the on-chip communication ports.

The ADSP-TS201 processor includes the following multiprocessing capabilities:

- On-chip bus arbitration for glueless multiprocessing
- Globally accessible internal memory and registers
- Semaphore support
- Powerful, in-circuit multiprocessing emulation

Emulation and Test Support

The ADSP-TS201 processor supports the IEEE Standard P1149.1 Joint Test Action Group (JTAG) port 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

The TigerSHARC processor is a static superscalar DSP-type processor that executes from one to four 32-bit *instruction slots* in an *instruction line*. With few exceptions, instructions execute with a throughput of one instruction line (four instruction slots) per processor core clock (CCLK) cycle

Figure 1-6 shows the instruction slot and line structure.

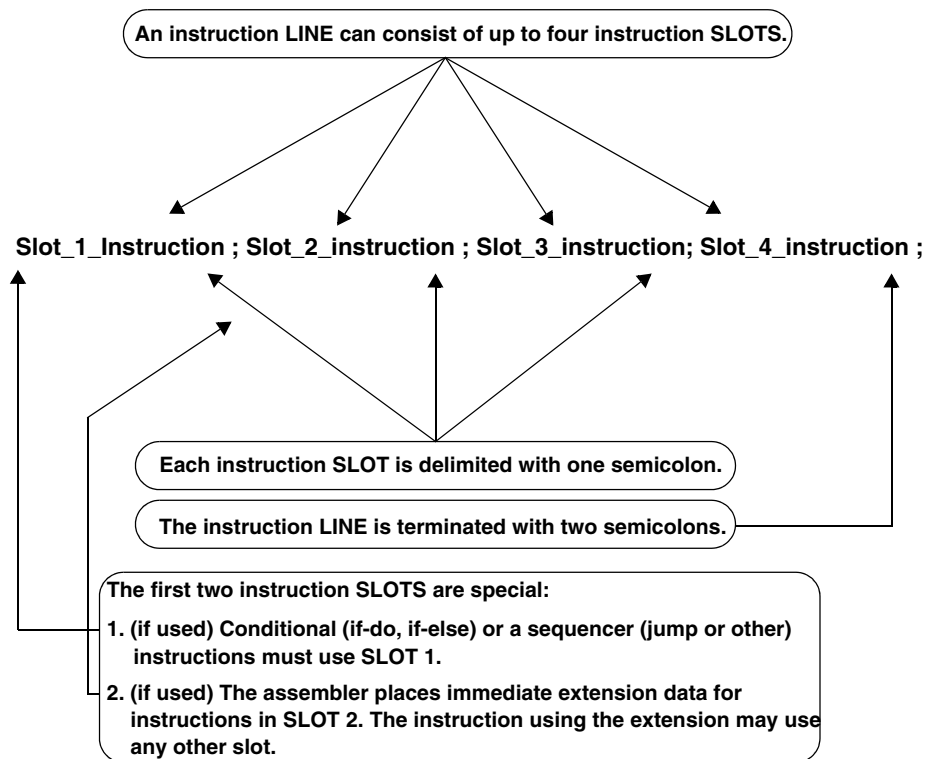


Figure 1-6. Instruction Line and Slot Structure

Instruction Line Syntax and 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 “;;”.
- Four instructions on an instruction line can be executed in parallel.
- Instructions that use 32-bit data (for example, 32-bit direct addresses or 32-bit immediate data) require two instruction slots to execute; one slot for the instruction and one slot for the immediate extensions.
- Program sequencer and immediate extension instructions require specific instruction slots. The immediate extension slot is assigned by the assembler; an immediate extension is not an instruction written specifically by the programmer.

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 processor executes or stalls the instructions in the same instruction line together.* Although the processor 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 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 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 processor executes and

the *option* in which the processor executes the operation. Operations include computations, data movements, and program flow controls. Options 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 processor’s instruction syntax—there are no mode bits to set in control registers for this processor.

This book presents instructions in summary format. This format shows 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 <i>one exception</i> . Empty parenthesis (no options selected) may not appear in an instruction.
<i>Rm Rmd Rmq</i>	Register names are replaceable items in the summary syntax and appear in italics. Register names indicate that the syntax requires a single (<i>Rm</i>), double (<i>Rmd</i>), or quad (<i>Rmq</i>) register. For more information on register name syntax, compute block selection, and data format selection, see “Register File Registers” on page 2-6 .

Instruction Line Syntax and Structure

$\langle imm\#\rangle$ Immediate data (literal values) in the summary syntax appears as $\langle imm\#\rangle$ 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 processor 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 two, four, and one 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;;
```

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 1-27](#).

Conditional Execution Support

All instructions can be executed conditionally (a mechanism also known as predicated execution). The condition field exists in the first 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 compute 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 these 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 8-14](#).

Instruction Parallelism Rules

The ADSP-TS201 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 processor 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 8-53](#)), 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).

Instruction Parallelism Rules

Table 1-1 on page 1-31 and Table 1-2 on page 1-36 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 1-38
- “Compute Block Instruction Restrictions” on page 1-39
- “IALU Instruction Restrictions” on page 1-43
- “Sequencer Instruction Restrictions” on page 1-48

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 processor 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 ADSP-TS201 processor. *Static* restrictions are distinguished from *dynamic* restrictions, in that static restrictions can be resolved by the 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 does 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` and `K0` through `K3`.

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.

Instruction Parallelism Rules

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, since the offset $0x40000$ is greater than 8 bits). The destination is $XR3:0$, which are X compute block registers. The ‘X Register File, $D_{reg} = XR31-0$ ’ 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
- 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 X and Y compute block input ports, and the J-IALU 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.	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	SOC Interface
⇒ Resources Available ⇒	4	1	1	2	1	1	1	1	2	2	1	1	2	2	1	1	1	1
↓ 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 ⁴ $Ureg = Ureg$	1			1														
<i>Immediate Load (resource total = instr. + Ureg)</i>																		
Immediate 15-bit Load $Ureg = Imm15$	1			1														
Immediate 32-bit Load $Ureg = Imm32$	2		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.	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	SOC Interface
⇒ Resources Available ⇒	4	1	1	2	1	1	1	1	2	2	1	1	2	2	1	1	1	1
⇓ Instruction Types ⇓																		
<i>Memory Load (resource total = instr. + Ureg)</i>																		
J-IALU Load ⁵ $Ureg = [Jm + += Jn imm8]$	1		1	1	1													
J-IALU Load, 32-bit offset $Ureg = [Jm + += imm32]$	2		1	1	1													
K-IALU Load ⁵ $Ureg = [Km + += Kn imm8]$	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.	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	SOC Interface
⇒ Resources Available ⇒	4	1	1	2	1	1	1	1	2	2	1	1	2	2	1	1	1	1
⇓ Instruction Types ⇓																		
<i>Ureg Transfer and Store (Source Register) Resources</i>																		
J-IALU <i>Ureg</i> = J30-0 JB3-0 JL3-0							1											
K-IALU <i>Ureg</i> = K30-0 KB3-0 KL3-0								1										
X-Register File <i>Dreg</i> = XR31-0									1		1							
Y-Register File <i>Dreg</i> = XR31-0													1		1			
XY-Register Files (SIMD) <i>Ureg</i> = XYR31-0									1		1		1		1			
Sequencer <i>Ureg</i> = CJMP RETI RETS ... ⁶																	1	
SOC Interface ⁵ <i>Ureg</i> = SOC <i>Ureg</i> ⁷																		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.	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	SOC Interface
⇒ Resources Available ⇒	4	1	1	2	1	1	1	1	2	2	1	1	2	2	1	1	1	1
⇓ 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 ...⁶</i>																		1
SOC Interface ⁵ <i>Ureg = SOC Ureg⁷</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.	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	SOC Interface
⇒ Resources Available ⇒	4	1	1	2	1	1	1	1	2	2	1	1	2	2	1	1	1	1
↓ Instruction Types ↓																		
<i>Memory Load Ureg (Destination Register) Resources</i>																		
X-Register File DAB/SDAB <i>XDreg</i> = DAB q[<i>addr</i>] <i>XDreg</i> = XR31-0									1	1		1						
Y-Register File DAB/SDAB <i>YDreg</i> = DAB q[<i>addr</i>] <i>YDreg</i> = YR31-0													1	1			1	
XY-Register Files DAB/SDAB <i>XYDreg</i> = DAB q[<i>addr</i>] <i>XYDreg</i> = XYR31-0									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 information. These constraints cannot be exceeded within the core.
- 4 Two register transfer instructions (Ureg=Ureg) may appear on an instruction line if transferring core Uregs. If transferring SOC Uregs, only one transfer may appear on an instruction line.
- 5 Register load and transfer instructions targeting the same register quad (for example, XR0 and XR2 are both in register quad XR3:0) may appear on an instruction line if the sources are internal locations. These loads/transfers targeting registers in the same quads may not appear on the same line if the sources are external locations (external memory or SOC Uregs).
- 6 Complete list is all sequencer registers in register groups 0x1A, 0x38, and 0x39: CJMP, RETI, RETIB, RETS, DBGE, ILATSTH/L, LCx, ILATH/L, IMASKH/L, PMASKH/L, TIMERxH/L, TMR-INxH/L, SQCTL, SQCTLST, SQCTLCL, SQSTAT, SFREG, and ILATCLH/L.
- 7 Complete list is all external port SOC Ureg registers in register groups 0x24 and 0x3A: SYSCON, BUSLK, SDRCON, SYSTAT, SYSTATCL, BMAX, BMAXC, and AUTODMAx.
Complete list is all DMA SOC Ureg registers in register groups 0x20 and 0x23: DCS3-0, DCD3-0, DCNT, DCNTST, DCNTCL, DSTAT, and DSTATC.
Complete list is all link port SOC Ureg registers in register groups 0x25 and 0x27: LBUFTX3-0, LBUFRX3-0, LCTL3-0, and LSTAT3-0.

Instruction Parallelism Rules

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

	Resources												
	Inst. slots used	First inst. slot ¹	Sec. inst. slot ²	X Comp Blk In.	X ALU	X CLU	X Multiplier	X Shifter	Y Comp Blk In.	Y ALU	Y CLU	Y Multiplier	Y Shifter
⇒ Resources Available ⇒	4	1	1	2	1	1	1	1	2	1	1	1	1
⇓ Instruction Types ⇓													
<i>Sequencer Instructions</i>													
Conditional Jump/Call, 15-bit immediate IF <i>cond</i> , JUMP CALL <i>Imm15</i>	1	1											
Conditional Jump/Call, 32-bit immediate IF <i>cond</i> , JUMP CALL <i>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 $X_{Dreg} = Dreg + Dreg$	1			1	1								
X CLU instructions ³	1			1		1		*					
X- Multiplier instruction, except quad output $X_{Dreg} = Dreg * Dreg$	1			1			1						
X Shifter instruction, except MASK, FDEP, XSTAT ³	1			1		*		1					
X ALU instruction with quad output ⁴ (add/subtract, EXPAND, MERGE)	1			1	1		*						
X Multiplier instruction with quad output ⁴	1			1	*		1						
X Shifter instructions MASK, FDEP, XSTAT ³	1			2									

Table 1-2. Parallelism Rules
for Compute Block and Sequencer Instructions (Cont'd)

	Resources												
	Inst. slots used	First inst. slot ¹	Sec. inst. slot ²	X Comp Blk In.	X ALU	X CLU	X Multiplier	X Shifter	Y Comp Blk In.	Y ALU	Y CLU	Y Multiplier	Y Shifter
⇒ Resources Available ⇒	4	1	1	2	1	1	1	1	2	1	1	1	1
⇓ Instruction Types ⇓													
<i>Y Compute Block Operations</i>													
Y ALU instruction, except quad output $YDreg = Dreg + Dreg$	1								1	1			
Y CLU instruction ³	1								1		1		*
Y Multiplier instruction, except quad output $YDreg = Dreg * Dreg$	1								1			1	
Y Shifter instruction, except MASK, FDEP, STAT ³	1								1		*		1
Y ALU instruction with quad output ⁴ (add_sub, EXPAND, MERGE)	1								1	1		*	1
Y Multiplier instruction with quad output ⁴	1								1	*		1	
Y Shifter instructions MASK, FDEP, YSTAT ³	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 CLU instruction ³	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, X/YSTAT ³	1			1				1	1		*		1
XY ALU instruction with quad output ⁴ (add/subtract, EXPAND, MERGE)	1			1	1			1	1	1		*	1
XY Multiplier instruction with quad output ⁴	1			1			1	1	1	*		1	
XY Shifter instructions MASK, FDEP, X/YSTAT ³	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.
- 3 For quad output, the CLU uses the shifter result bus for quad results (see [Figure 1-7 on page 1-40](#)).
- 4 For quad output, the ALU and multiplier share an quad result bus (see [Figure 1-7 on page 1-40](#)).

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 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 (Js) and index  
(Jm) 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 $Rs = R_m \text{ op } R_n$ or $XYRs = R_m \text{ op } R_n$ are issued to both the X- and Y-compute blocks

Figure 1-7 shows the data flow for input and result operands for the compute blocks. This architecture establishes limits for the number and type of compute instructions that may appear on an instruction line for parallel execution.

Looking at the input operands (arrows in Figure 1-7 going into the ALU, CLU, multiplier, and shifter), the compute data flow imposes two limitations on compute instruction parallelism:

- Only two compute instructions may be issued to a compute block per instruction line.
- Only one instruction may be issued to a computation unit per instruction line.

Looking at the output operands (arrows in Figure 1-7 coming from the ALU, CLU, multiplier, and shifter), the compute data flow imposes special limitations on quad output compute instruction parallelism:

- ALU instructions with quad output may not execute in parallel with multiplier instructions with quad output.
- CLU instructions with quad output may not execute in parallel with shifter instructions.

Instruction Parallelism Rules

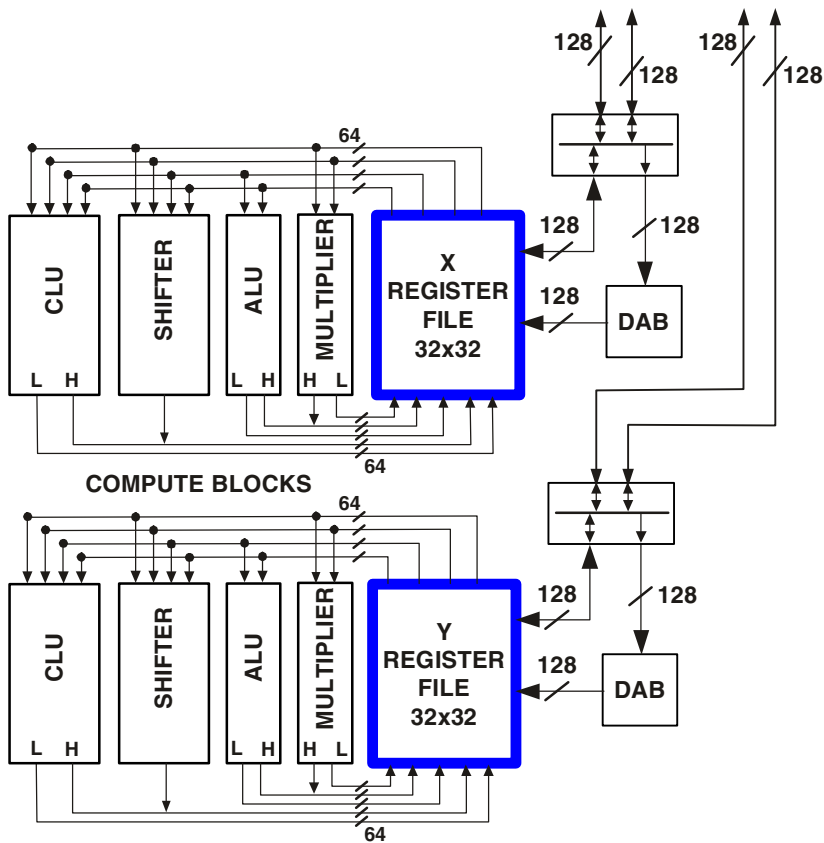


Figure 1-7. Data Register Files in Compute Block X and Y¹

- 1 On the outputs of the compute units, H indicates high half of quad result (bits 128–64), and L indicates low half of quad result (bits 63–0). For a result smaller or equal to 64 bits, compute units use the (L) lower result output.

The following are examples stemming from these conditions for issuing instructions to the compute blocks. Note that the assembler statically checks all of these input and output operand data flow 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, CLU, multiplier, or shifter) in a cycle. Each of the two instructions must be issued to a different unit (ALU, CLU, multiplier or shifter). For example:

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

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

- 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 compute unit (ALU, CLU, or multiplier) can use two result buses, *unless the second unit using two result buses is the CLU* (for example, a multiplier combined with a CLU). A unit uses two

Instruction Parallelism Rules

result buses either when the result is quad word or when there are two results (dual add and subtract instructions— $R0 = R1 + R2$, $R5 = R1 - R2$;). Another instruction is allowed in the same line. For example:

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

```
R0 = R1 + R2 , R5 = R1 - R2 ; XR11:8 = MR3:0 ;;  
/* Invalid; two instructions using two buses */
```

But, compute CLU operations that use two result buses *may not* be combined on an instruction line with a shifter instruction.

```
R3:0 = THR3:0; R6 = LSHIFT R1 BY R7;;  
/* Invalid; exceeds result bus for CLU and shifter */
```

- 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 the multiplier may be used only in these instructions:

```
MR3:2|MR1:0 +|- =  $R_m * R_n$  32-bit fractional multiply-accumulate  
MR3:2|MR1:0 +=  $R_m ** R_n$  Complex multiply-accumulate
```

- Two types of CLU related instructions execute on other computation units. Trellis (TR), Trellis History (THR), and Communications Control (CMCTL) register load instructions execute in the shifter computation unit, and PERMUTE instructions execute in the ALU computation unit.



Unlike previous TigerSHARC processors, all compute CLU instructions can execute in parallel with 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.

- 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] ;;
/* 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.

Instruction Parallelism Rules

- 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 */
```

- Access using the data alignment buffer (DAB) must be through a quad word load. It cannot be through “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 (DMA, external port, link port, interrupt controller, JTAG port, and autoDMA registers) 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 */
```

- 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 are 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 ; XR10 = [J17 + 2] ;;
/* Valid; one Ureg trans. and one load to comp. 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 */
```

Instruction Parallelism Rules

- 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
 - SOC interface registers: (external port) groups 0x24 and 0x3A, (DMA) groups 0x20 and 0x23, and (link port) groups 0x25 and 0x27
 - 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

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 */
```

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

$$J_s = J_m + | - J_n \text{ (CB)}$$

$$U_{reg} = \text{CB [L] [Q] } (J_m + | += J_n | Imm)$$

$$\text{CB [L] [Q] } (J_m + | += J_n | Imm) = U_{reg}$$

$$U_{reg} = \text{DAB [L] [Q] } (J_m + | += J_n | Imm)$$

$$U_{reg} = \text{BR [L] [Q] } (J_m + | += J_n | imm)$$

$$\text{BR [L] [Q] } (J_m + | += J_n | imm) = U_{reg}$$

$$U_{reg} = \text{BR [L] [Q] } (J_m + | += J_n | Imm)$$


The same restrictions apply to K-IALU instructions that use circular buffer or bit reversed addressing operations. The K_m (the index) may only be K0, K1, K2, or K3.

- 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 0x1E0000 to 0x1FFFFFF. For example:

```
Q[J2 + 0] = XR3:0 ;;
```

```
/* Invalid if J2 is in the range of 0x1E0000 to 0x1FFFFFF */
```

Instruction Parallelism Rules

- 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 */
```

- The following transactions are restricted:

```
Debug_Ureg = Sequencer_Ureg ;; /* Invalid */
```

```
Sequencer_Ureg = Debug_Ureg ;; /* Invalid */
```

```
Debug_Ureg = Debug_Ureg ;; /* Invalid */
```

```
/* Sequencer_Ureg: groups 0x1A, 0x30-0x39, and 0x3B */
```

```
/* Debug_Ureg: groups 0x1B, 0x3D-0x3F */
```

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

- 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 ;
XRO = 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 the logical operations on static flags instruction, SCFx += operation *Condition*, there can be no logical operation between compute block static flags (XSFO/1, YSF0/1, and XYSFO/1) and non-compute block conditions (J-IALU or K-IALU).
- The No Flag update (NF) option may be used independently for each instruction in the line, unless there is a conditional instruction in the line. For conditional instruction lines, either all instruction slots use the NF option or none uses the NF option.

Instruction Parallelism Rules

2 COMPUTE BLOCK REGISTERS

The ADSP-TS201 TigerSHARC processor core contains two compute blocks—compute block X and compute block Y. Each block contains a register file and four independent computation units. The units are the ALU, multiplier, shifter, and CLU. Because the execution of all compute instructions in the ADSP-TS201 processor depends on the input and output data formats and depends on whether the instruction is executed on one compute block or both, it is important to understand how to use the TigerSHARC processor’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.

This chapter describes:

- [“Register File Registers” on page 2-6](#)
- [“Numeric Formats” on page 2-16](#)

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 computation units within the compute blocks.

As shown in [Figure 2-1](#), data input to the register file may pass 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 7-1](#).

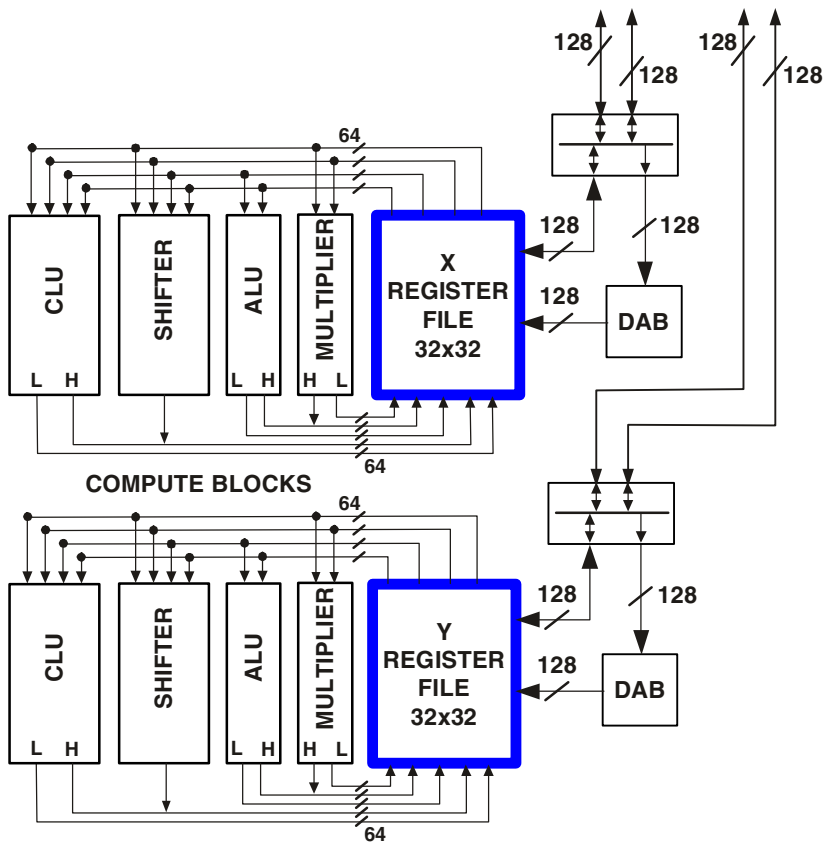



Figure 2-1. Data Register Files in Compute Block X and Y¹

- 1 On the outputs of the compute units, H indicates high half of quad result (bits 128–64), and L indicates low half of quad result (bits 63–0). For a result smaller or equal to 64 bits, compute units use the (L) lower result output.

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.

The distinction between memory-mapped and non-memory-mapped registers is important because the memory-mapped registers are *universal registers* (*Ureg*). These registers are considered universal because *Ureg* registers can be accessed over processor core internal buses, making *Ureg* registers accessible to all internal bus masters (for example, compute blocks, IALUs, and others) and accessible to external bus masters (for example, host processors or other TigerSHARC processors). 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*). The non-memory mapped registers (for example, MR, PR, and others) are only accessible to their related execution unit.

For operations in a multiprocessing system, it is very useful that most 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 processor can perform write access to its own registers by using the multiprocessor memory space, but the processor would have to tie up the external bus to access its own registers this way.

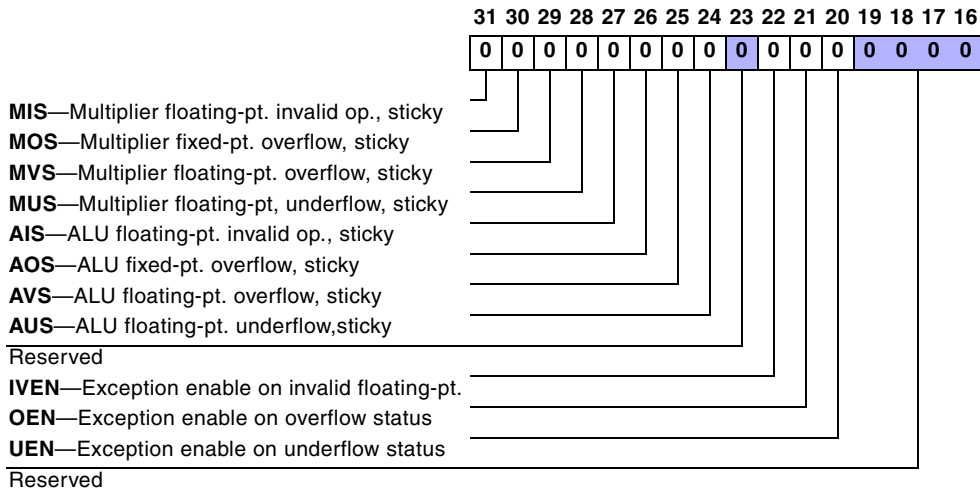


Figure 2-2. XSTAT/YSTAT (Upper) Register Bit Descriptions

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) and Multiplier Result Overflow (XMR4 and YMR4) registers—Multiplier
- Bit FIFO Overflow Temporary (XBFOTMP and YBFOTMP) registers—Shifter
- Trellis Result (XTR31-0 and YTR31-0), Trellis History Result (XTHR3:0 and YTHR3:0), and Communication Control (XCMCTL and YCMCTL) register—CLU

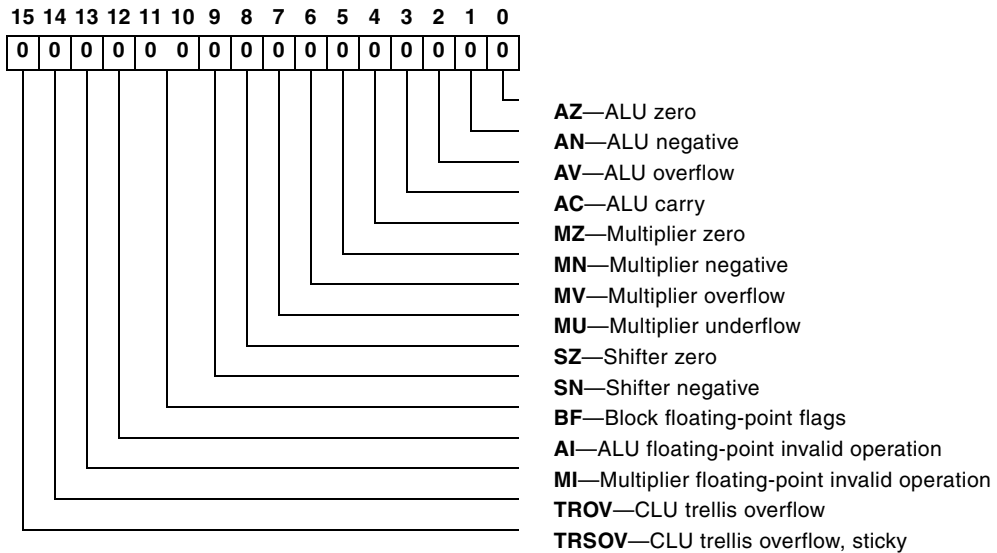


Figure 2-3. XSTAT/YSTAT (Lower) Register Bit Descriptions


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](#), also see [“Load/Transfer Compute Block Status \(X/YSTAT\) Registers” on page 10-195](#)) hold the status flags for each compute block. These flags are set or reset to indicate the status of an instruction’s execution in a compute block’s ALU, multiplier, shifter, and CLU. 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/YBF0TMP registers temporarily store or return overflow from GET-BITS and PUTBITS instructions.

The X/YTR31-0 and X/YTHR3:0 registers on previous TigerSHARC processors held data related to Trellis instructions. In the ADSP-TS201 processor, the trellis registers are also used for DESPREAD and XCORRS instructions. The Communication Control (X/YCCTL) register serves as an optional source for the CUT value in the XCORRS instruction.

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 the processor's internal bus and the computation units. The register file registers—XR31-0 and YR31-0—are both universal registers (*Ureg*) and data registers (*Dreg*).

Inputs for computations usually come from the register file. However, in some cases, inputs may come from the compute block's internal registers. The results go to the register file, except when they go to the compute block's internal registers.

 It is important to note that a register may be used once as an output operand (receives a result) in an instruction line, but the assembly syntax permits using a register multiple times as an input operand (provides an input) 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 a computation accesses a register, the processor 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 1-23](#) and [“Instruction Parallelism Rules” on page 1-27](#).

There are many ways to name registers in the TigerSHARC processor assembly syntax. The register name syntax provides selection of many features of compute 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.

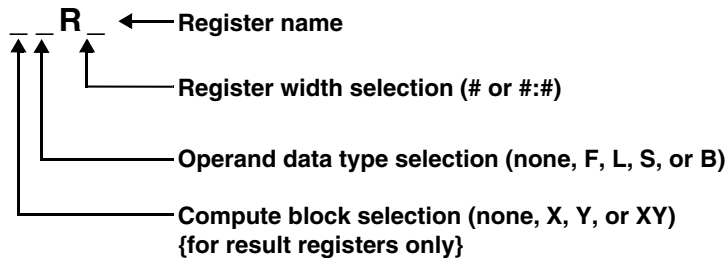


Figure 2-4. Register File Register Name Syntax

i The processor’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.
```

Register File Registers

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

```
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. The SIMD syntax ($XYRs = Rm \text{ op } Rn$), directs the processor to execute the same instruction in both compute blocks.

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.

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 processor'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 address must end in zero or 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 address must end in zero or 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 */
```

Register File Registers

```
XR1:0 = R3:2 + R1:0 ;;  
/* code, no substitution */
```

Operand Data Type 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.

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 5-5](#).

To distinguish between fixed- and floating-point data, the register name prefix F indicates that the register contains floating-point data. The processor 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*.

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;;
```

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.

Register File Registers

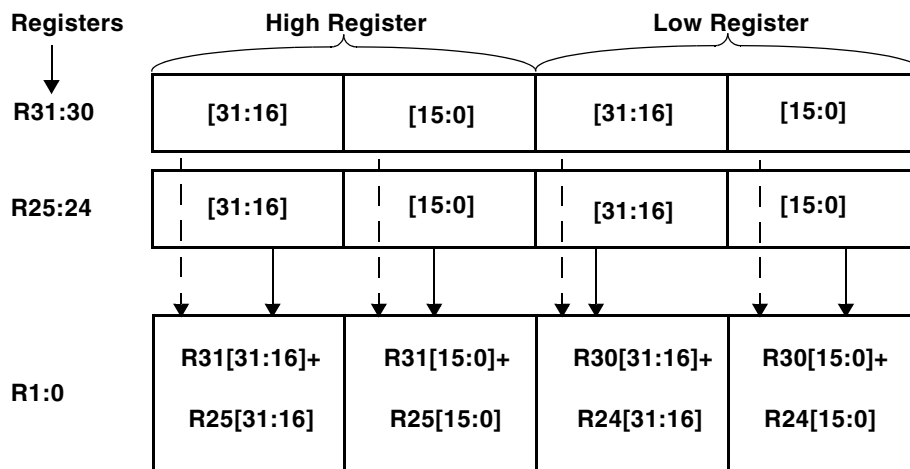


Figure 2-5. Addition of Four Short Word Operands in Double Registers

Registers File Syntax Summary

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

- “ALU” on page 3-1
- “CLU” on page 4-1
- “Multiplier” on page 5-1
- “Shifter” on page 6-1

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.

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.

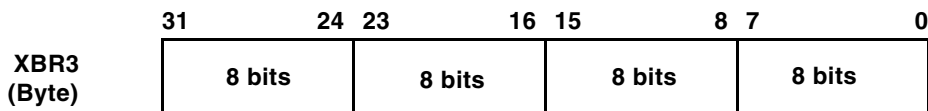


Figure 2-6. Register $R3$ in Compute Block X, Treated as Byte Data

Register File Registers

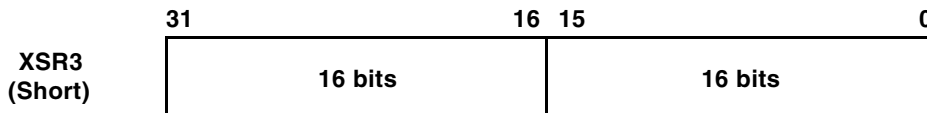


Figure 2-7. Register R3 in Compute Block X, Treated as Short Data

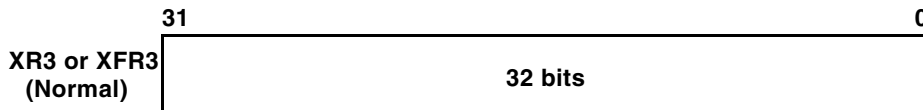


Figure 2-8. Register R3 in Compute Block X, Treated as Normal Data

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](#).

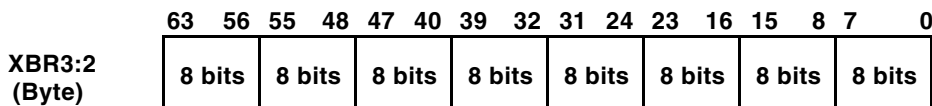


Figure 2-9. Register R3:2 in Compute Block X, Treated as Byte Data

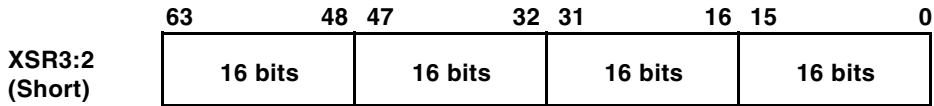


Figure 2-10. Register R3:2 in Compute Block X, Treated as Short Data

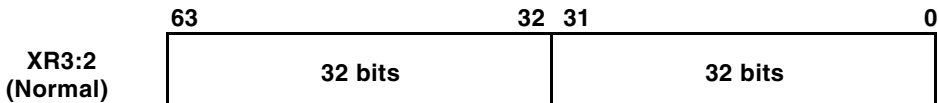


Figure 2-11. Register R3:2 in Compute Block X, Treated as Normal Data

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

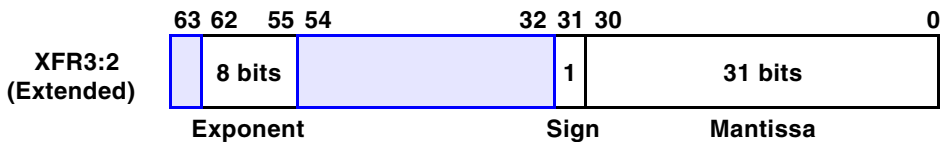


Figure 2-12. Register R3:2 in Compute Block X, Treated as Extended (Floating-Point) Data

Numeric Formats

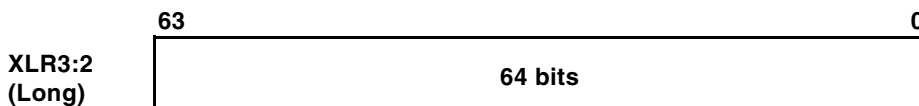


Figure 2-13. Register R3:2 in Compute Block X, Treated as Long Data

Numeric Formats

The processor supports the 32-bit single-precision floating-point data format defined in the IEEE Standard 754/854. In addition, the processor supports a 40-bit extended-precision version of the same format with eight additional bits in the mantissa. The processor 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 mantissa, and an 8-bit unsigned-magnitude exponent e .

For normalized numbers, the mantissa consists of a 23-bit fraction f and a hidden bit of 1 that is implicitly presumed to precede f_{22} in the mantissa. 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 mantissa to 24 bits from the 23 bits actually stored in the data format. This bit also insures that the mantissa of any number in the IEEE normalized number format is always greater than or equal to 1 and less than 2.

The 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 .

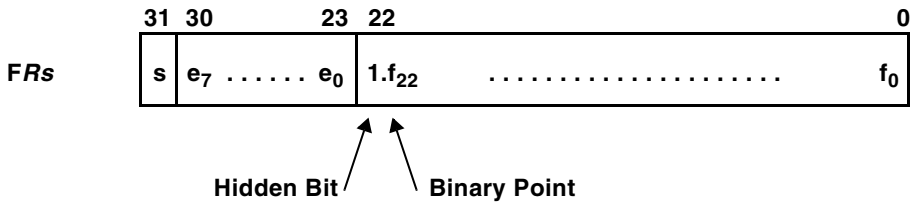


Figure 2-14. IEEE 32-Bit Single-Precision Floating-Point Format (Normal Word)

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 number 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 processor 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 ADSP-TS201 processor is compatible with the IEEE single-precision floating-point data format in all respects, except for:

- The ADSP-TS201 processor does not provide inexact flags.
- NAN inputs generate an invalid exception and return a quiet NAN. When one or both of the inputs are NANs, the sign bit of the operation is set as an XOR of the signs of the input sign bits.
- 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 mantissa. 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.

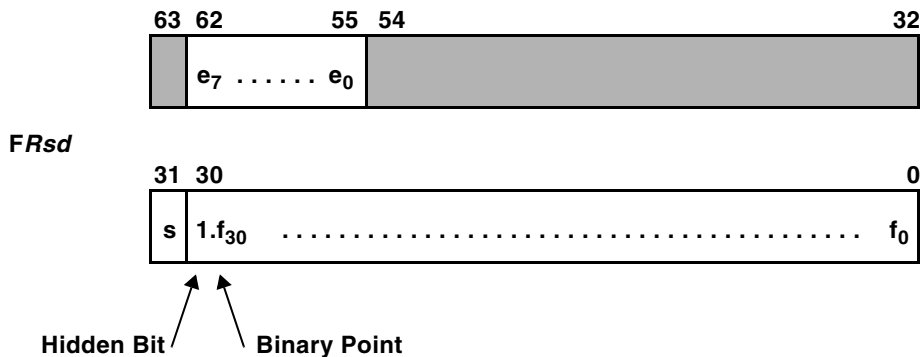


Figure 2-15. 40-Bit Extended-Precision Floating-Point Format (Extended Word)¹


¹ Bits 63 and 54–32 (in the upper register) are unused.

Fixed-Point Formats

The processor supports fixed-point fractional and integer formats for 16-, 32-, and 64-bit data. The processor supports a fixed-point, signed, integer format for 8-bit data. In these formats, numbers can be signed (two's complement) or unsigned. The possible combinations are shown in [Figure 2-16](#) through [Figure 2-31](#). In the fractional format, there is an implied binary point to the left of the most significant magnitude bit. In

Numeric Formats

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.

 Data in the 8- and 16-bit formats is always *packed* into 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.

ALU outputs usually have the same width and data format as the inputs. The multiplier, however, may produce a result either the same data size as the input operands or double the size of the input operands. Multiplying two 32-bit operands produces either a 32-bit or a 64-bit result. Multiplying two 16-bit operands produces either a 16-bit or a 32-bit result; this is a quad SIMD multiplication on double register operands. If operands are unsigned (integers or fractions), the result is unsigned. These formats are shown in [Figure 2-26](#), [Figure 2-27](#), [Figure 2-30](#) and [Figure 2-31](#).

If one multiplier operand is signed and the other unsigned, the result is signed. If both inputs are signed, the result is signed. If inputs are signed fractions, the result is 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 fraction multiplied by itself.) Thus, the left shift normally removes a redundant sign bit, increasing the precision of the most significant product.

Also, if the multiplier 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 5-1.

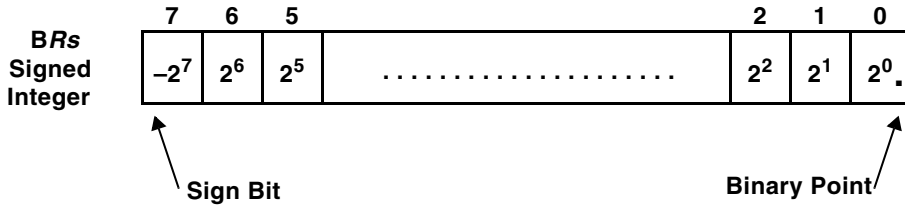


Figure 2-16. 8-Bit Fixed-Point Format, Signed Integer (Byte Word)

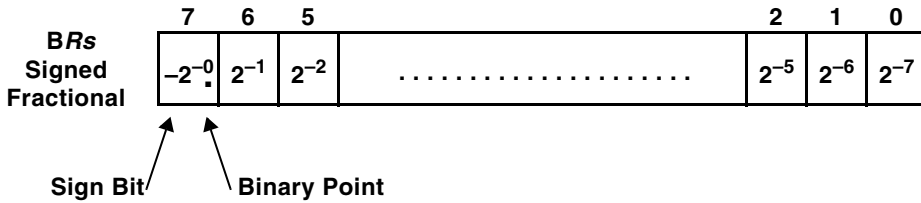


Figure 2-17. 8-Bit Fixed-Point Format, Signed Fractional (Byte Word)

Numeric Formats

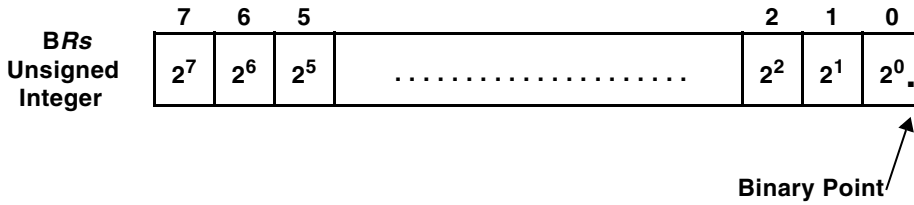


Figure 2-18. 8-Bit Fixed-Point Format, Unsigned Integer (Byte Word)

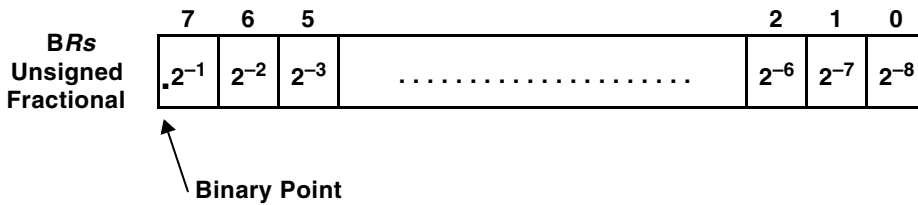


Figure 2-19. 8-Bit Fixed-Point Format, Unsigned Fractional (Byte Word)

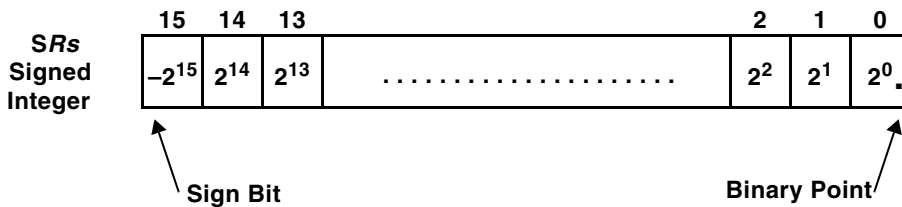


Figure 2-20. 16-Bit Fixed-Point Format, Signed Integer (Short Word)

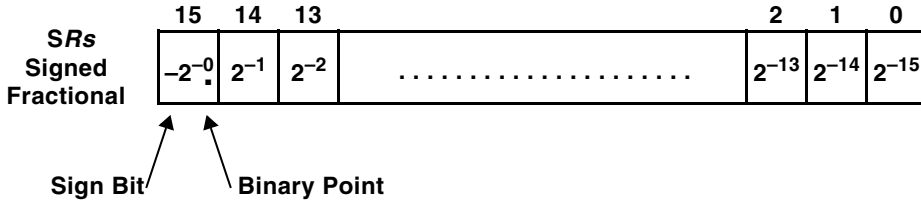


Figure 2-21. 16-Bit Fixed-Point Format, Signed Fractional (Short Word)

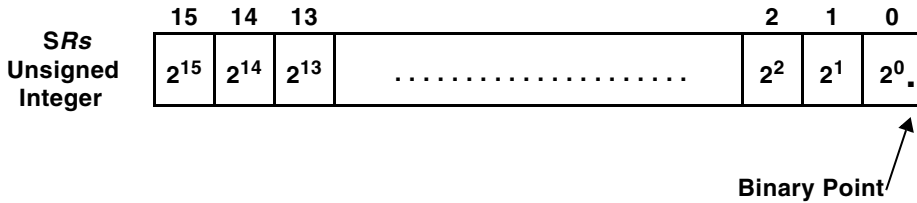


Figure 2-22. 16-Bit Fixed-Point Format, Unsigned Integer (Short Word)

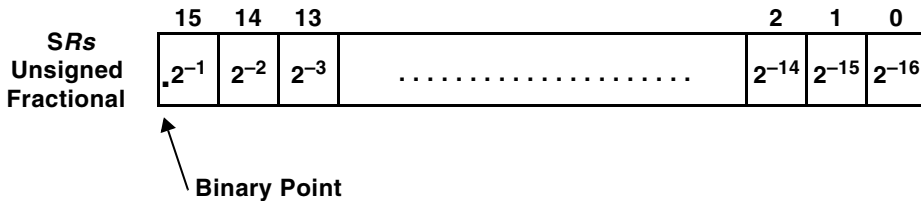


Figure 2-23. 16-Bit Fixed-Point Format, Unsigned Fractional (Short Word)

Numeric Formats

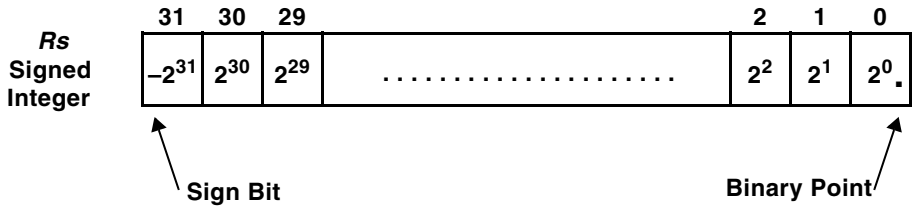


Figure 2-24. 32-Bit Fixed-Point Format, Signed Integer (Normal Word)

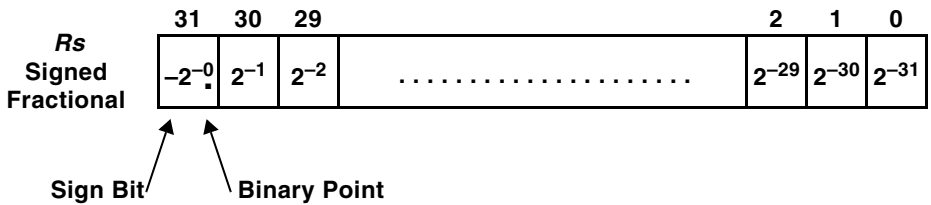


Figure 2-25. 32-Bit Fixed-Point Format, Signed Fractional (Normal Word)

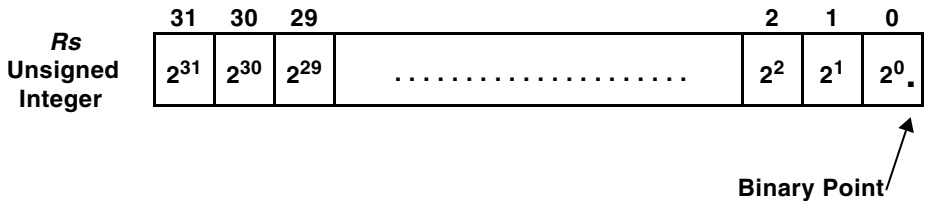


Figure 2-26. 32-Bit Fixed-Point Format, Unsigned Integer (Normal Word)

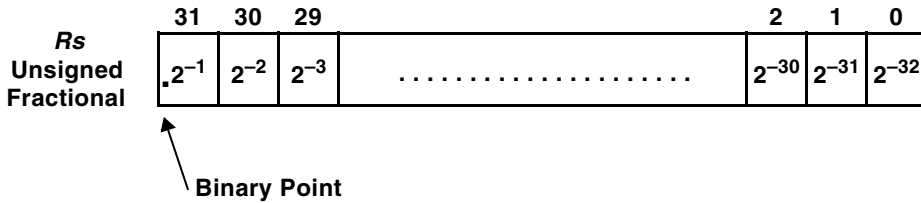


Figure 2-27. 32-Bit Fixed-Point Format, Unsigned Fractional (Normal Word)

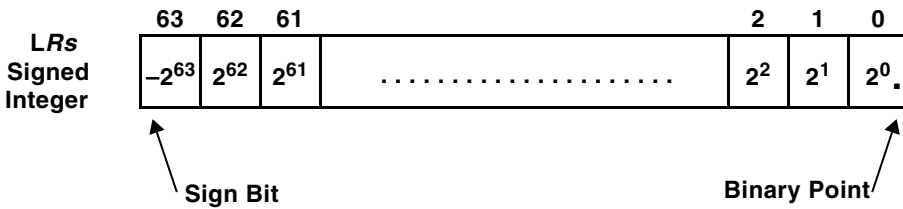


Figure 2-28. 64-Bit Fixed-Point Format, Signed Integer (Long Word)

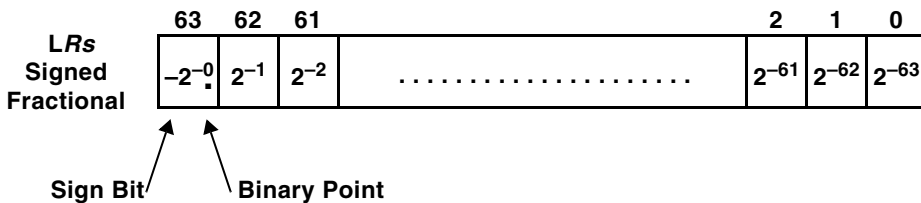


Figure 2-29. 64-Bit Fixed-Point Format, Signed Fractional (Long Word)

Numeric Formats

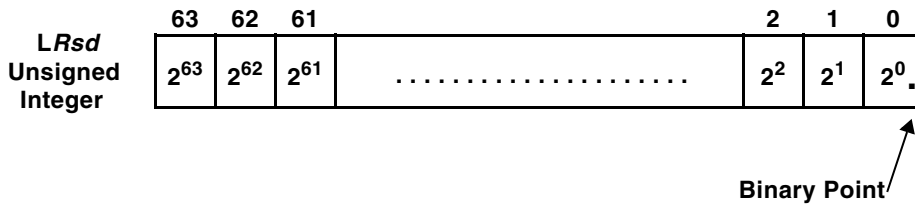


Figure 2-30. 64-Bit Fixed-Point Format, Unsigned Integer (Long Word)

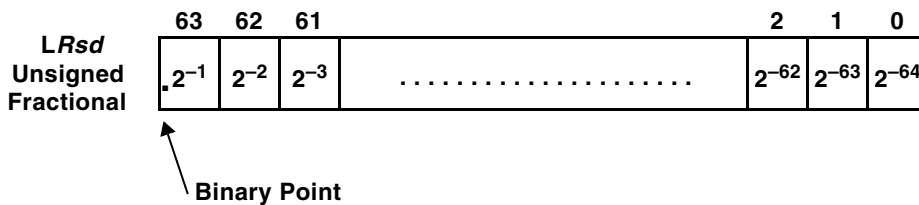


Figure 2-31. 64-Bit Fixed-Point Format, Unsigned Fractional (Long Word)

3 ALU

The ADSP-TS201 TigerSHARC processor core contains two computation units known as compute blocks. Each compute block contains a register file and four independent computation units—an ALU, a CLU, 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.

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.

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)
- Fixed-point only *data conversion (promotion/demotion) operations* – expand (EXPAND), compact (COMPACT), and merge (MERGE)

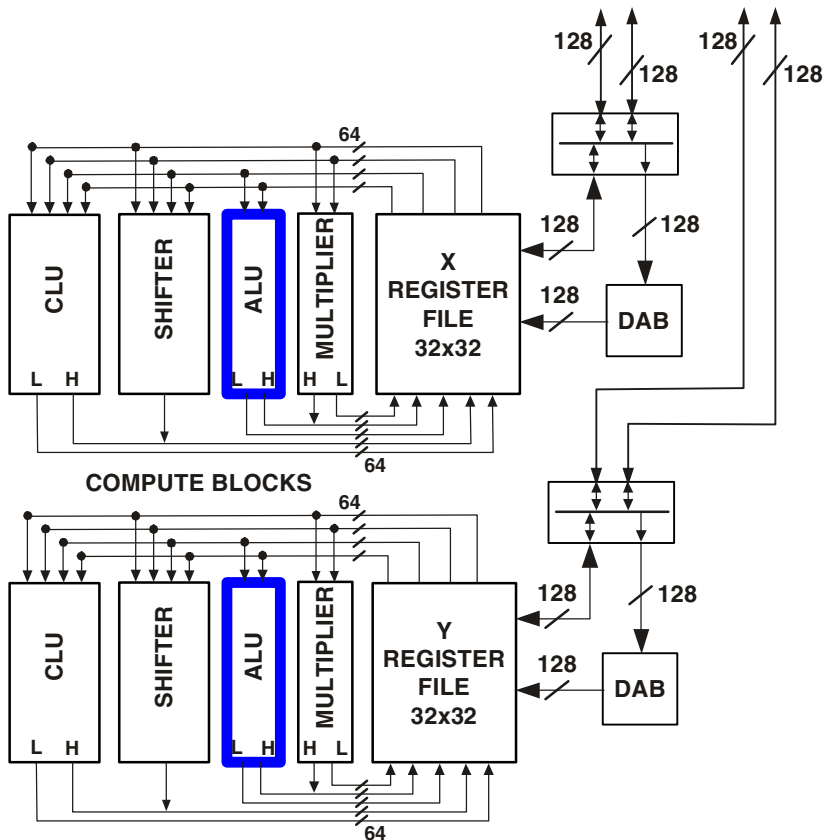


Figure 3-1. ALUs in Compute Block X and Y¹

¹ On the outputs of the compute units, H indicates high half of quad result (bits 128–64), and L indicates low half of quad result (bits 63–0). For a result smaller or equal to 64 bits, compute units use the (L) lower result output.

- *Logical Operations* – AND, AND NOT, OR, XOR, and PASS
- Floating-point only *arithmetic operations* – floating-point conversion (FLOAT), fixed-point conversion (FIX), copy sign (COPYSIGN), scaling (SCALB), reciprocal or division seed (RECIPS), square root or

reciprocal square root seed (RSQRTS), extract mantissa (MANT), extract exponent (LOGB), extend operand (EXTD), and translate extended to a single operand (SNGL)

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

- Fixed-point *arithmetic 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
- Fixed-point *data conversion operations* support:
 - 8-bit (byte) to 16-bit (short)
 - 16-bit (short) to 32-bit (normal)
 - 32-bit (normal) to 16-bit (short)
 - 64-bit (long) to 32-bit (normal)
 - 128-bit (quad) to 64-bit (long)
- *Logical Operations* support:
 - 32-bit (normal) input operands
 - 64-bit (long) input operands
 - output 32- or 64-bit results

ALU Operations

- Floating-point *arithmetic operations* support:
 - 32-bit (normal) input operands (IEEE standard)
 - 40-bit (extended) input operands
 - output 32- or 40-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 2-6](#).

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 1-23](#). For a list of ALU instructions and their syntax, see [“ALU Instruction Summary” on page 3-22](#).

ALU Operations

The ALU performs arithmetic operations on fixed-point and floating-point data and logical operations on fixed-point data. The processor 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 processor uses the PR register with the different types of SUM, ABS, 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 2-6](#). 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 an absolute value of a fixed-point subtract of the
64-bit input operand XR5:4 from XR3:2; the absolute value of the
result is placed in YLR1:0; the "L" in the result register name
causes the input and output to be treated 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. */

```

When multiple input operands (for example, short or byte word data) 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](#)). After executing the instruction YSR2 = R0 - R1;;, the YR2 register contains 0x0001FFFB (0x1 in upper half and -0x5 in lower half).

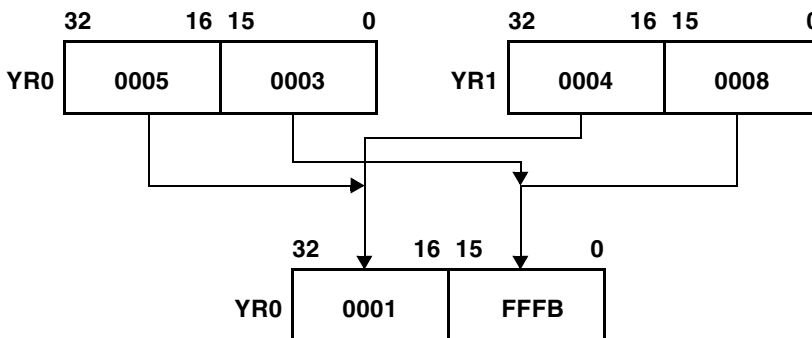


Figure 3-2. Input Operands for Parallel Subtract

ALU Operations

All ALU instructions generate status flags to indicate the status of the result, unless the No Flag Update (NF) option is used. Because multiple operations are occurring in multiple operands of one 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 3-22](#). The ALU instruction options include:

- (.) signed operation, no saturation¹, round-to-nearest even², fractional operation³
- (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

¹ Where saturation applies

² Where floating-point operation applies

³ Where fixed-point operation applies

⁴ Where truncation applies

- (Z) signed result returns zero operation for MIN/MAX
- (UZ) unsigned result returns zero operation for MIN/MAX
- (I) signed operation, integer operation⁴
- (IU) unsigned operation, integer operation³
- (IS) signed operation, saturation, integer operation³
- (ISU) unsigned operation, saturation, integer operation³
- (NF) no flag update

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 (normal word) input
operands with saturation. */
```

```
YLFR1:0 = ABS( R3:2 - R5:4 ) (T) ::
/* This is an absolute value of the difference o two
floating-point 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 and round-to-nearest even; this
is the same as omitting the parenthesis. */
```

Signed/Unsigned Option

The processor 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 ALU may be unsigned or two's complement. Floating-point data in the ALU is always signed-magnitude. For information on the supported numeric formats, see [“Numeric Formats” on page 2-16](#).

ALU Operations

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.

The maximum positive (positive overflow) and minimum negative (negative overflow) values for the *signed formats* are:

- Byte format, maximum positive is 0x7F, minimum negative is 0x80
- Short format, maximum positive is 0x7FFF, minimum negative is 0x8000
- Normal format, maximum positive is 0x7FFF FFFF, minimum negative is 0x8000 0000

The maximum positive (positive overflow) and minimum negative (negative overflow) values for the *unsigned formats* are:

- Byte format, maximum positive is 0xFF, minimum negative is 0x0
- Short format, maximum positive is 0xFFFF, minimum negative is 0x0
- Normal format, maximum positive is 0xFFFF FFFF, minimum negative is 0x0

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. The range is extended by one bit by changing the representation to unsigned. Without the `X`, the output MSB is always zero, and the maximum negative signed value (`0x80...0`) causes an overflow. When `ABS` with the `X` option is used, the output range is extended from `0x0` to `0x80...0`. The output numbers are unsigned in the extended range.

Truncation Option

For ALU instructions that support truncation for floating-point operations as the `T` option, this option permits selection of the results rounding operation. The processor supports two methods for rounding—round-toward-zero and round-toward-nearest-even. The rounding methods comply with the IEEE 754 Standard and have these definitions:

- Round-toward-nearest-even (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.

ALU Operations

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 operation.

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.

Without The Z Option, the pseudo code for the MAX instruction is:

If ($Rmd \geq Rnd$) *then* $Rsd = Rmd$; *else* $Rsd = Rnd$;

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

With the Z option, the pseudo code for the MAX instruction is:

If ($Rmd \geq Rnd$) *then* $Rsd = Rmd$; *else* $Rsd = 0$;

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 processor 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](#).

No Flag Update Option

Almost all compute operations generate status, which affects the compute unit's status register. Often, it is useful in some applications to retain status from an operation. There are a number of techniques for storing status (for example, storing the status register contents or loading flag values into the static flags (SFREG) register). The No Flag Update (NF) option provides a different method for working with status. Instead of storing an operation's status for future usage, programs can prevent status from being overwritten by using the (NF) option on following operations to prevent status generation. There are some restrictions on how the (NF) option can be used with conditional instructions. For more information, see [“Conditional Execution” on page 8-14](#).

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 3-15](#).

[Table 3-1](#) shows the flags in XSTAT or YSTAT that indicate ALU status 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 ¹	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 ²	All floating-point ops; cleared by fixed-point ops

1 If the OEN bit in the X/YSTAT register is set, an AV flag generates a software exception.

2 If the IVEN bit in the X/YSTAT register is set, an AI flag generates a software exception.

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 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 ¹	All floating-point ops
AVS	ALU floating-point overflow, sticky ²	All floating-point ops
AOS	ALU fixed-point overflow, sticky ²	All fixed-point ops
AIS	ALU floating-point invalid operation, sticky ³	All floating-point ops

1 If the UEN bit in the X/YSTAT register is set, an AUS flag generates a software exception.

2 If the OEN bit in the X/YSTAT register is set, an AVS or AOS flag generates a software exception.

3 If the IVEN bit in the X/YSTAT register is set, an AI flag generates a software exception.

Flag update occurs at the end of each operation and is available on the next instruction slot. A program cannot write the XSTAT or YSTAT register explicitly in the same cycle that status is being generated.

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

i When the (NF) option is used, the compute unit does not update the flags with status from the operation. Programs can use the (NF) option with all ALU instruction except for `PASS`, `COMP`, and `FCOMP`.

AZ – ALU Zero

The AZ flag is set whenever the result of an ALU operation is zero, or when a floating-point operation result is underflow.

i If the UEN bit in the X/YSTAT register is set, an underflow operation (AZ flag set by a floating-point underflow) generates a software exception.

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:

- $Rs = ABS Rm$; AN is Rm (input data) sign
- $FRs = ABS Rm$; AN is Rm (input data) sign
- $Rs = ABS (Rm \{+|- \} Rn)$; AN is set to be the sign of the result prior to ABS operation
- $Rs = MANT FRm$; AN is Rm (input data) sign
- $FRs = ABS (Rm \{+|- \} Rn)$; 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.

ALU Operations

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.


If in the following example R5 and R6 are 0x70...0 (large positive numbers), the result of the add instruction (0xD0...0) is larger than the format for signed operations (the MSB that in two's complement format indicates the sign is set).

```
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 in the above example of add with expand, the final result is 0x00...0D0...00. The overflow is defined by the final result and is not defined by intermediate results. In the case above, there is no overflow.

 If the OEN bit in the X/YSTAT register is set, an overflow operation (AV flag set) generates a software exception.

AI – ALU Invalid

The AI flag indicates an invalid floating-point operation as defined by IEEE floating-point standard. [For more information, see “IEEE Single-Precision Floating-Point Data Format” on page 2-16.](#)

 If the IVEN bit in the X/YSTAT register is set, an invalid floating-point operation (AI flag set) generates a software exception.

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 8-14](#).

Table 3-3. ALU Conditions

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

ALU Examples

ALU Static Flags

In the program sequencer, the static flag (SFREG) register can store status flag values for later usage in conditional instructions. With SFREG, each compute block has two dedicated static flag bits, X/YSCF0 (conditions are XSF0, YSF0, XYSF0, or SF0) and X/YSCF1 (conditions are XSF1, YSF1, XYSF1, or SF1). The following example shows how to load a compute block condition value into a static flag register.

```
XSF0 = XAEQ ;; /* Load X-compute block SEQ flag into XSCF0 bit in
static flags (SFREG) register */
IF XSF0, D0 XR5 = R4 + R3 ;; /* the XSF0 condition tests the XSCF0
static flag bit */
```

For more information on static flags, see [“Conditional Execution” on page 8-14](#).

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 part of the 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 part of the
instruction is a fixed-point subtract of the 32-bit input oper-
ands 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 ;;
```

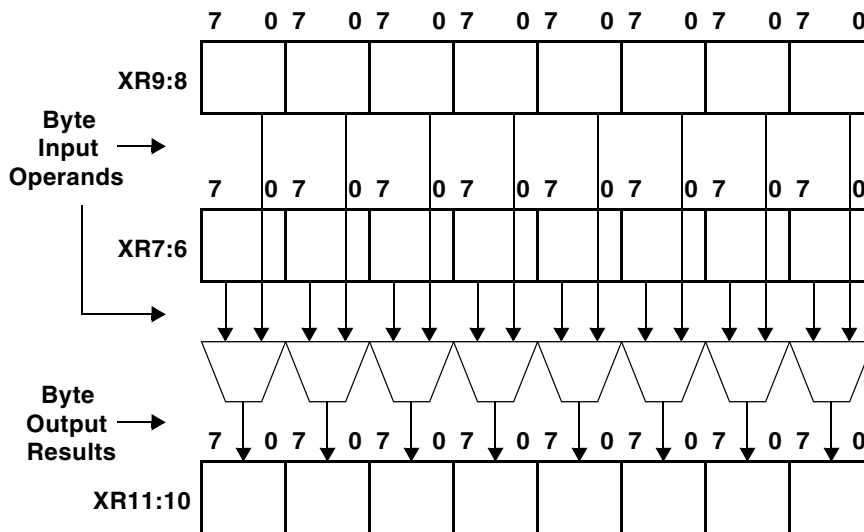


Figure 3-3. Input Operands for Parallel Add

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 Sum 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 */
```

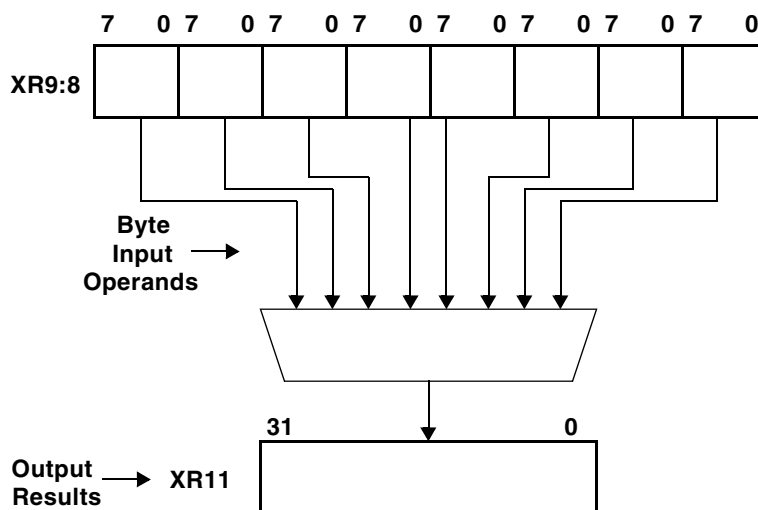



Figure 3-4. Input Operands for Sideways Add

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, 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 syntax:

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

 The `PR` load or store instructions 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 PR0 register.

```
PR0 += SUM SR5:4;;
```

Figure 3-5 shows register usage for parallel/sideways add.

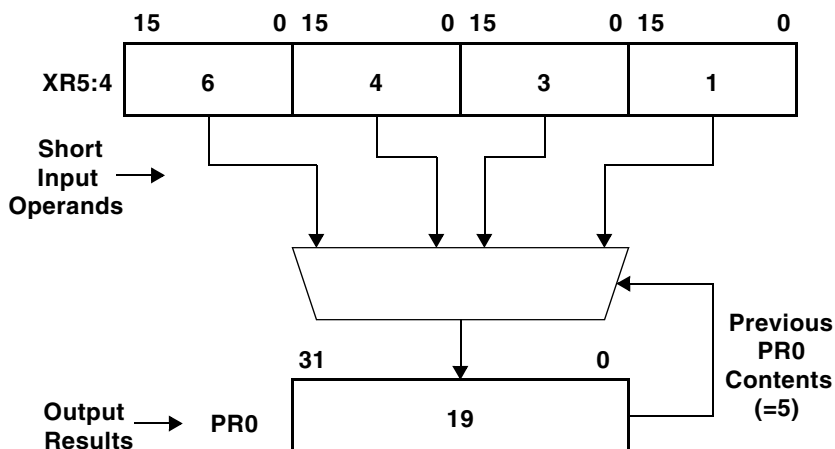


Figure 3-5. Input Operands for Parallel/Sideways Add

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”

The conventions used in these listings for representing register names, optional items, and choices are covered in detail in [“Register File Registers” on page 2-6](#). 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 1-23](#) and [“Instruction Parallelism Rules” on page 1-27](#).

Listing 3-2. ALU Fixed-Point Instructions

$$\{X|Y|XY\}\{S|B\}Rs = Rm +|- Rn \{(\{S|SU\}\{NF\})\} ;^1$$
$$\{X|Y|XY\}\{L|S|B\}Rsd = Rmd +|- Rnd \{(\{S|SU\}\{NF\})\} ;^1$$

$\{X|Y|XY\}Rs = Rm + CI \{-1\} ;$
 $\{X|Y|XY\}LRsd = Rmd + CI \{-1\} ;$
 $\{X|Y|XY\}Rs = Rm + Rn + CI \{(S|SU)\{NF)\}\} ;^1$
 $\{X|Y|XY\}Rs = Rm - Rn + CI -1 \{(S|SU)\{NF)\}\} ;^1$
 $\{X|Y|XY\}LRsd = Rmd + Rnd + CI \{(S|SU)\{NF)\}\} ;^1$
 $\{X|Y|XY\}LRsd = Rmd - Rnd + CI -1 \{(S|SU)\{NF)\}\} ;^1$
 $\{X|Y|XY\}\{S|B\}Rs = (Rm +|- Rn)/2 \{(T)\{U)\{NF)\}\} ;^1$
 $\{X|Y|XY\}\{L|S|B\}Rsd = (Rmd +|- Rnd)/2 \{(T)\{U)\{NF)\}\} ;^1$
 $\{X|Y|XY\}\{S|B\}Rs = ABS Rm \{(NF)\} ;$
 $\{X|Y|XY\}\{L|S|B\}Rsd = ABS Rmd \{(NF)\} ;$
 $\{X|Y|XY\}\{S|B\}Rs = ABS (Rm + Rn) \{(X)\{NF)\}\} ;^2$
 $\{X|Y|XY\}\{L|S|B\}Rsd = ABS (Rmd + Rnd) \{(X)\{NF)\}\} ;^2$
 $\{X|Y|XY\}\{S|B\}Rs = ABS (Rm - Rn) \{(X|U)\{NF)\}\} ;^3$
 $\{X|Y|XY\}\{L|S|B\}Rsd = ABS (Rmd - Rnd) \{(X|U)\{NF)\}\} ;^3$
 $\{X|Y|XY\}\{S|B\}Rs = - Rm \{(NF)\} ;$
 $\{X|Y|XY\}\{L|S|B\}Rsd = - Rmd \{(NF)\} ;$
 $\{X|Y|XY\}\{S|B\}Rs = MAX|MIN (Rm, Rn) \{(U)\{Z)\{NF)\}\} ;^4$
 $\{X|Y|XY\}\{L|S|B\}Rsd = MAX|MIN (Rmd, Rnd) \{(U)\{Z)\{NF)\}\} ;^4$
 $\{X|Y|XY\}S|BRsd = VMAX|VMIN (Rmd, Rnd) \{(NF)\} ;$
 $\{X|Y|XY\}\{S|B\}Rs = INC|DEC Rm \{(S|SU)\{NF)\}\} ;^1$
 $\{X|Y|XY\}\{L|S|B\}Rsd = INC|DEC Rmd \{(S|SU)\{NF)\}\} ;^1$
 $\{X|Y|XY\}\{S|B\}COMP(Rm, Rn) \{(U)\} ;^4$
 $\{X|Y|XY\}\{L|S|B\}COMP(Rnd,Rnd) \{(U)\} ;^4$
 $\{X|Y|XY\}\{S|B\}Rs = CLIP Rm BY Rn \{(NF)\} ;$
 $\{X|Y|XY\}\{L|S|B\}Rsd = CLIP Rmd BY Rnd \{(NF)\} ;$

-
- ¹ Options include: (): no saturation, (S): saturation, signed, (SU): saturation, unsigned
¹ Options include: (): signed, round-to-nearest even, (T): signed, truncate, (U): unsigned, round-to-nearest even, (TU): unsigned, truncate
² Options include: (): signed inputs, saturation, (X): extend for ABS uses unsigned saturation
³ Options include: (): signed inputs, saturation, (X): extend for ABS uses unsigned saturation, (U): unsigned
⁴ 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

ALU Instruction Summary

```
{X|Y|XY}Rs = SUM SRm|BRm {{(U){NF}}} ;1
{X|Y|XY}Rs = SUM SRmd|BRmd {{(U){NF}}} ;1
{X|Y|XY}Rs = ONES Rm|Rmd {(NF)} ;
{X|Y|XY}PR1:0 = Rmd {(NF)} ;
{X|Y|XY}Rsd = PR1:0 {(NF)} ;
{X|Y|XY}Rs = BFOINC Rmd {(NF)} ;
{X|Y|XY}PRO|PR1 += ABS (SRmd - SRnd){{(U){NF}}} ;1
{X|Y|XY}PRO|PR1 += ABS (BRmd - BRnd){{(U){NF}}} ;1
{X|Y|XY}PRO|PR1 += SUM SRm {{(U){NF}}} ;1
{X|Y|XY}PRO|PR1 += SUM SRmd {{(U){NF}}} ;1
{X|Y|XY}PRO|PR1 += SUM BRm {{(U){NF}}} ;1
{X|Y|XY}PRO|PR1 += SUM BRmd {{(U){NF}}} ;1
{X|Y|XY}{S|B}Rs = Rm + Rn, Ra = Rm - Rn {(NF)} ; (dual op.)
{X|Y|XY}{L|S|B}Rsd = Rmd + Rnd, Rad = Rmd - Rnd {(NF)} ; (dual op.)
```

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 {(NF)} ;
{X|Y|XY}LRsd = Rmd AND|AND NOT|OR|XOR Rnd {(NF)} ;
{X|Y|XY}Rs = NOT Rm {(NF)} ;
{X|Y|XY}LRsd = NOT Rmd {(NF)} ;
```

Listing 3-4. ALU Fixed-Point Miscellaneous Instructions

```
{X|Y|XY}Rsd = EXPAND SRm {+|- SRn} {{(I|IU){NF}}} ;2
{X|Y|XY}Rsq = EXPAND SRmd {+|- SRnd} {{(I|IU){NF}}} ;2
{X|Y|XY}SRsd = EXPAND BRm {+|- BRn} {{(I|IU){NF}}} ;2
{X|Y|XY}SRsq = EXPAND BRmd {+|- BRnd} {{(I|IU){NF}}} ;2
{X|Y|XY}SRs = COMPACT Rmd {+|- Rnd} {{(T|I|IS|ISU){NF}}} ;3
```

¹ Options include: (): signed, (U): unsigned

² Options include: (): fractional, (I): integer signed, (IU): integer unsigned

³ Options include: (): fractional round, (I): integer, no saturate, (T): fractional, truncate, (IS): integer, saturate, signed, (ISU): integer, saturate, unsigned

```

{X|Y|XY}BRs = COMPACT SRmd {+|- SRnd} {(T|I|IS|ISU}{NF)} ;3
{X|Y|XY}Rs = COMPACT LRmd {(U|IS|ISU}{NF)} ;
{X|Y|XY}LRsd = COMPACT QRmq {(U|IS|ISU}{NF)} ;
{X|Y|XY}BRsd = MERGE Rm, Rn {(NF)} ;
{X|Y|XY}BRsq = MERGE Rmd, Rnd {(NF)} ;
{X|Y|XY}SRsd = MERGE Rm, Rn {(NF)} ;
{X|Y|XY}SRsq = MERGE Rmd, Rnd {(NF)} ;
{X|Y|XY}Rsd = PERMUTE (Rmd, Rn) {(NF)} ;
{X|Y|XY}Rsq = PERMUTE (Rmd, -Rmd, Rn) {(NF)} ;

```

Listing 3-5. Floating-Point ALU Instructions

```

{X|Y|XY}FRs = Rm +|- Rn {(T){NF}} ;1
{X|Y|XY}FRsd = Rmd +|- Rnd {(T){NF}} ;1
{X|Y|XY}FRs = (Rm +|- Rn)/2 {(T){NF}} ;1
{X|Y|XY}FRsd = (Rmd +|- Rnd)/2 {(T){NF}} ;1
{X|Y|XY}FRs = MAX|MIN (Rm, Rn) {(NF)} ;2
{X|Y|XY}FRsd = MAX|MIN (Rmd, Rnd) {(NF)} ;2
{X|Y|XY}FRs = ABS Rm {(NF)} ;
{X|Y|XY}FRsd = ABS Rmd {(NF)} ;
{X|Y|XY}FRs = ABS (Rm +|- Rn) {(T){NF}} ;1
{X|Y|XY}FRsd = ABS (Rmd +|- Rnd) {(T){NF}} ;1
{X|Y|XY}FRs = - Rm {(NF)} ;
{X|Y|XY}FRsd = - Rmd {(NF)} ;
{X|Y|XY}FCOMP (Rm, Rn) ;
{X|Y|XY}FCOMP (Rmd, Rnd) ;
{X|Y|XY}Rs = FIX FRm|FRmd {BY Rn} {(T){NF}} ;1
{X|Y|XY}FRs|FRsd = FLOAT Rm {BY Rn} {(T){NF}} ;1
{X|Y|XY}FRsd = EXT D Rm {(NF)} ;
{X|Y|XY}FRs = SNGL Rmd {(T){NF}} ;
{X|Y|XY}FRs = CLIP Rm BY Rn {(NF)} ;

```

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

² Options include: (): round, (T): truncate (MIN only)

ALU Instruction Summary

$\{X|Y|XY\}FRsd = \text{CLIP } Rmd \text{ BY } Rnd \{(NF)\} ;$
 $\{X|Y|XY\}FRs = Rm \text{ COPYSIGN } Rn \{(NF)\} ;$
 $\{X|Y|XY\}FRsd = Rmd \text{ COPYSIGN } Rnd \{(NF)\} ;$
 $\{X|Y|XY\}FRs = \text{SCALB } FRm \text{ BY } Rn \{(NF)\} ;$
 $\{X|Y|XY\}FRsd = \text{SCALB } FRmd \text{ BY } Rn \{(NF)\} ;$
 $\{X|Y|XY\}FRs = \text{PASS } Rm ;$
 $\{X|Y|XY\}FRsd = \text{PASS } Rmd ;$
 $\{X|Y|XY\}FRs = \text{RECIPS } Rm \{(NF)\} ;$
 $\{X|Y|XY\}FRsd = \text{RECIPS } Rmd \{(NF)\} ;$
 $\{X|Y|XY\}FRs = \text{RSQRTS } Rm \{(NF)\} ;$
 $\{X|Y|XY\}FRsd = \text{RSQRTS } Rmd \{(NF)\} ;$
 $\{X|Y|XY\}Rs = \text{MANT } FRm|FRmd \{(NF)\} ;$
 $\{X|Y|XY\}Rs = \text{LOGB } FRm|FRmd \{(S)\{(NF)\}\} ;^1$
 $\{X|Y|XY\}FRs = Rm + Rn, FRa = Rm - Rn \{(NF)\} ; \textit{(dual op.)}$
 $\{X|Y|XY\}FRsd = Rmd + Rnd, FRad = Rmd - Rnd \{(NF)\} ; \textit{(dual op.)}$

¹ Options include: (): do not saturate, (S): saturate

4 CLU

The TigerSHARC processor core contains two computation units known as compute blocks. Each compute block contains a register file and four independent computation units—an ALU, a CLU, a multiplier, and a shifter. The Communications Logic Unit (CLU) is highlighted in [Figure 4-1](#). The CLU takes its inputs from the register file or TR/THR registers, and returns its outputs to the register file or TR/THR registers. Most CLU instructions operate on the Trellis Registers (TR) and Trellis History Registers (THR).

This unit performs specialized communications functions, primarily to support Viterbi decoding, turbo-code decoding, code-division multiple access (CDMA) decoding, despreading operations, and complex correlation. These functions may also be applied in non-communications algorithms.

This chapter contains:

- [“CLU Operations” on page 4-4](#)
- [“CLU Examples” on page 4-43](#)
- [“CLU Instruction Summary” on page 4-46](#)

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

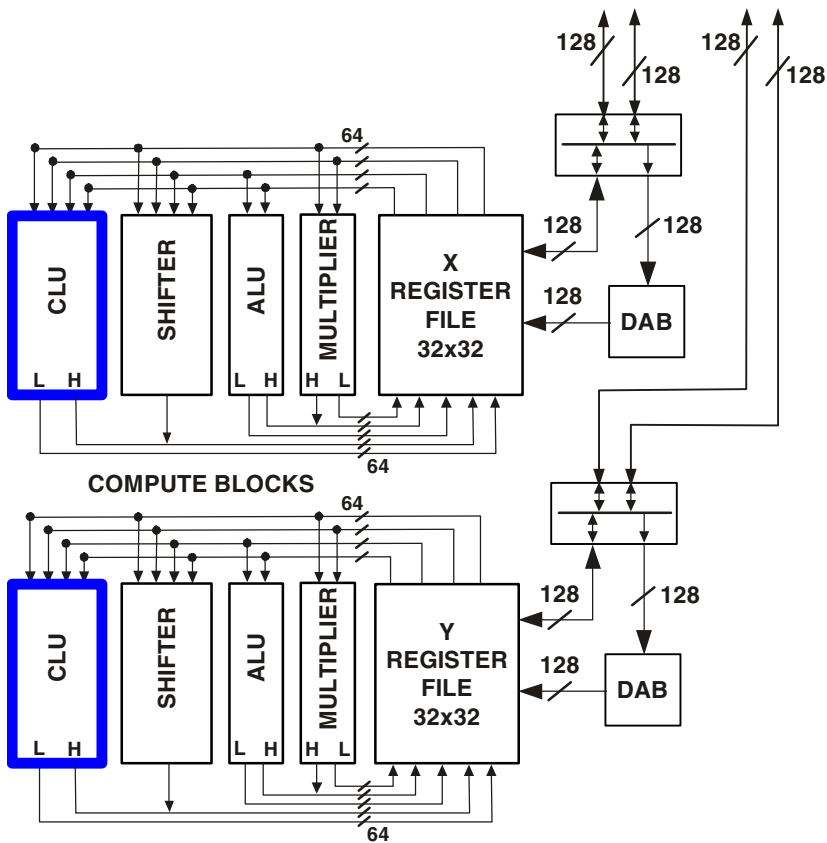


Figure 4-1. CLUs in Compute Block X and Y¹

- 1 On the outputs of the compute units, H indicates high half of quad result (bits 128–64), and L indicates low half of quad result (bits 63–0). For a result smaller or equal to 64 bits, compute units use the (L) lower result output.

purposes; for example, the T_{MAX} 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 four cycles in both compute blocks together.

CLU operations are applied to fixed-point data. Relating CLU operations and supported real and complex data types shows that the 128-bit CLU unit within each compute block supports:

- Jacobian logarithm for turbo decode (TMAX)
- CDMA despreader (DESPREAD)
- CDMA cross correlations (XCORRS)
- Polynomial reordering (PERMUTE)
- Trellis add/compare/select (ACS)

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

- 32-bit complex (16-bit real and 16-bit imaginary)
- 16-bit complex (8-bit real and 8-bit imaginary)
- 2-bit complex (1-bit real and 1-bit imaginary)
- 8-bit and 16-bit real

The data-type support in the CLU instructions is according to the algorithm need. Most instructions do not support all data types and sizes.

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 CLU instructions, see [“Register File Registers” on page 2-6](#).

CLU Operations

The remainder of this chapter presents descriptions of CLU instructions, options, and results using instruction syntax. For an explanation of the instruction syntax conventions used in CLU and other instructions, see [“Instruction Line Syntax and Structure” on page 1-23](#). For a list of CLU instructions and their syntax, see [“CLU Instruction Summary” on page 4-46](#).

CLU Operations

The CLU performs communications logic operations on fixed-point data. The ADSP-TS201 processor uses compute block registers for the input operands and output result from CLU operations. The CLU instructions refer to three types of registers:

- $R_{m,n,s}$ – register file (data) registers
- $TR_{m,n,s}$ – 32 (Trellis) registers that are dedicated to the CLU instructions
- THR – four (Trellis History) registers used by the ACS, DESPREAD, and XCORRS instructions for shifted data
- CMCTL – Communications Control registers used by the XCORRS instruction as an optional source for CUT value

For more information on the register files and register naming syntax for selecting data type and width, see [“Register File Registers” on page 2-6](#).

This section describes:

- [“TMAX Function” on page 4-6](#)
- [“Trellis Function” on page 4-8](#)
- [“” on page 4-22](#)
- [“Cross Correlations Function” on page 4-30](#)
- [“CLU Instruction Options” on page 4-41](#)
- [“CLU Execution Status” on page 4-42](#)

For turbo and Viterbi decoding, the 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).

All CLU 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. For more information on CLU status, see [“CLU Execution Status” on page 4-42](#).

TMAX Function

The T_{MAX} 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 T_{MAX} 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 4-1](#) 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 LSBs.

Table 4-1 shows the TMAX values. The maximum output error is one LSB.

Table 4-1. TMAX Values

Negative Input (Binary)	Positive Input (Binary)	Output (Binary)
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 (Figure 4-2) is a widely used tool in communications systems. For example, the Viterbi and turbo decoding algorithms both operate on trellises. The ADSP-TS201 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.

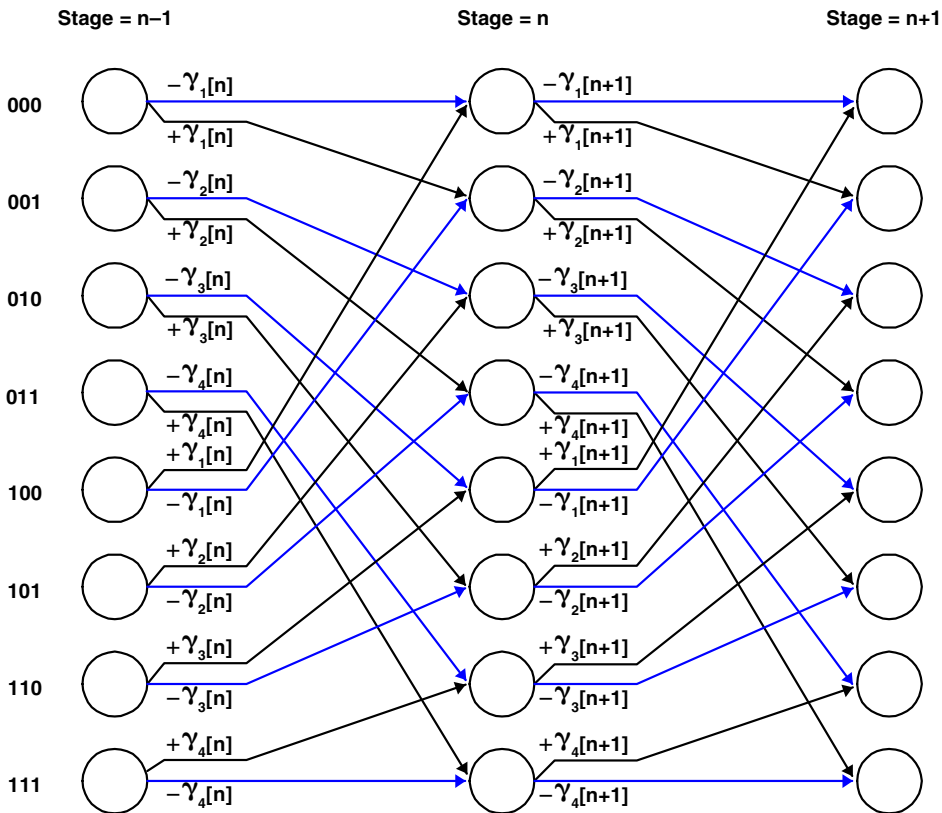


Figure 4-2. Trellis Diagram

A typical trellis diagram (shown in [Figure 4-2](#)) 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 depends on the metrics of each of its two possible previous states (at stage $n-1$) as well as transition metrics $\gamma[n]$ ($\text{gamma}[n]$) corresponding to particular input/output combinations.

The particular symmetry among the transition metrics shown in [Figure 4-2](#) 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 states 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 4-6](#).


CLU Operations

Trellis Function of the Form

$$STRsd = TMAX(TRmd + Rmq_h, TRnd + Rmq_l) ;$$

The high part of Rmq is added to $TRmd$, the low part of Rmq is added to $TRnd$, and the $TMAX$ function is executed between both add results, as illustrated in [Figure 4-3](#) and [Figure 4-4](#).

Saturation is supported in this instruction. For more details, see “[Saturation Option](#)” on page 3-8.

-  On previous TigerSHARC processors, this instruction could not be executed in parallel to CLU and ALU instructions of the same compute block. On the ADSP-TS201 processor, one CLU instruction can be executed in each compute block per instruction line, and any ALU instruction can be executed on the same instruction line with any CLU instruction.

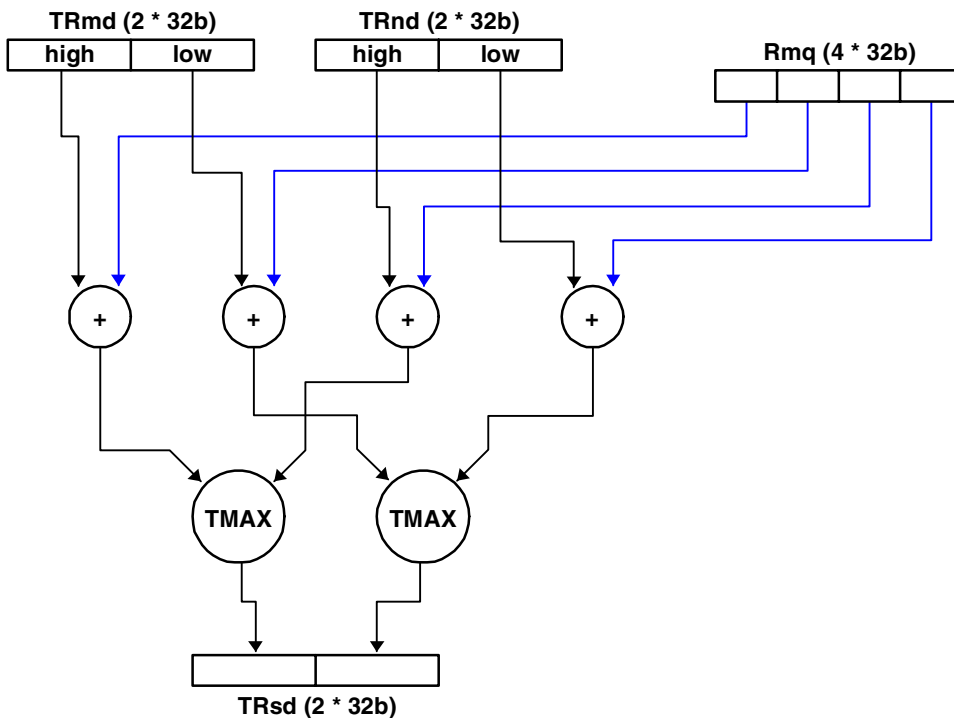


Figure 4-3. $TRsd = TMAX(TRmd + Rmq_h, TRnd + Rmq_l)$

CLU Operations

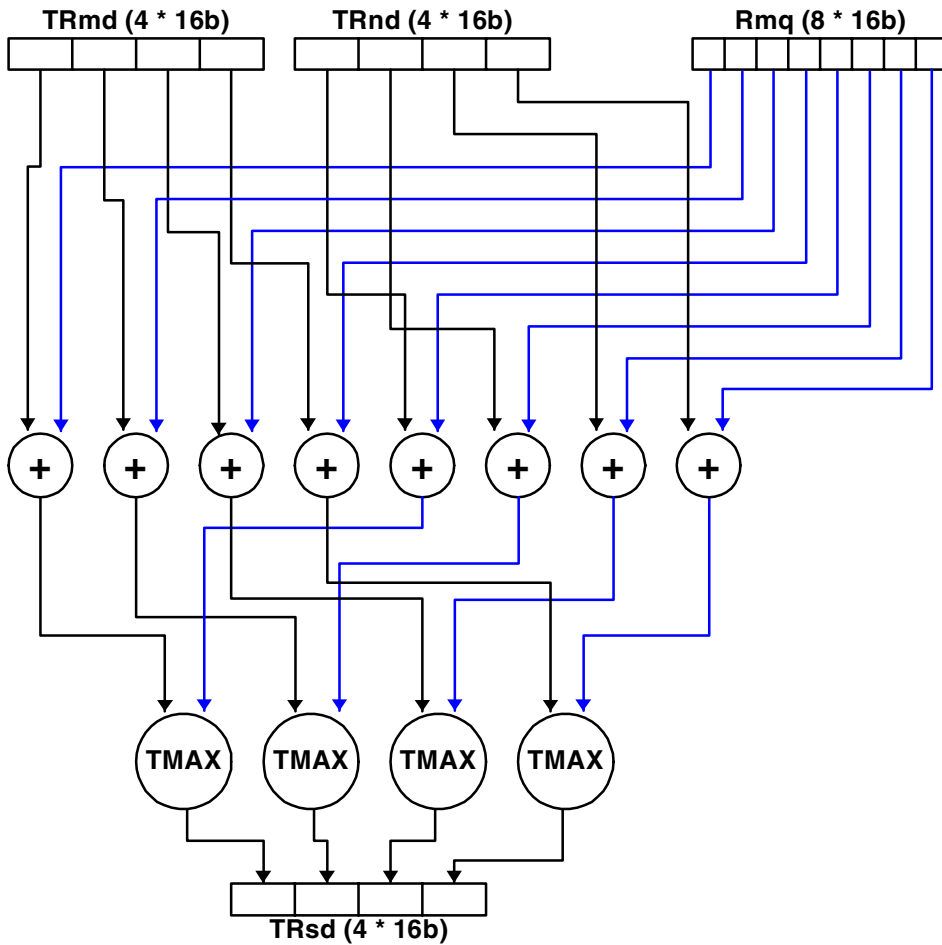



Figure 4-4. $STRsd = TMAX(TRmd + Rmq_h, TRnd + Rmq_l)$

Trellis Function of the Form

$$STRsd = TMAX(TRmd - Rmq_h, TRnd - Rmq_l) ;$$

The high part of Rmq is subtracted from $TRmd$, the low part of Rmq is subtracted from $TRnd$, and the $TMAX$ function is executed between both subtract results, as illustrated in [Figure 4-6](#) and [Figure 4-7](#). For subtraction, the order of operands appears in [Figure 4-5](#).

Saturation is supported in this instruction. For more details, see “[Saturation Option](#)” on page 3-8.

-  On previous TigerSHARC processors, this instruction could not be executed in parallel to CLU and ALU instructions of the same compute block. This limitation does not apply to the ADSP-TS201 processor.

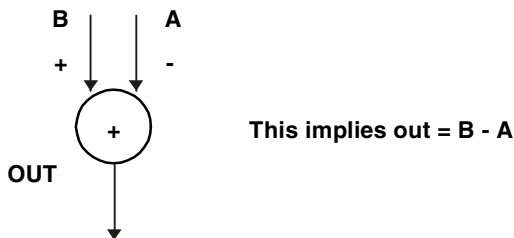


Figure 4-5. Order of Operands for Subtract Diagrams

CLU Operations

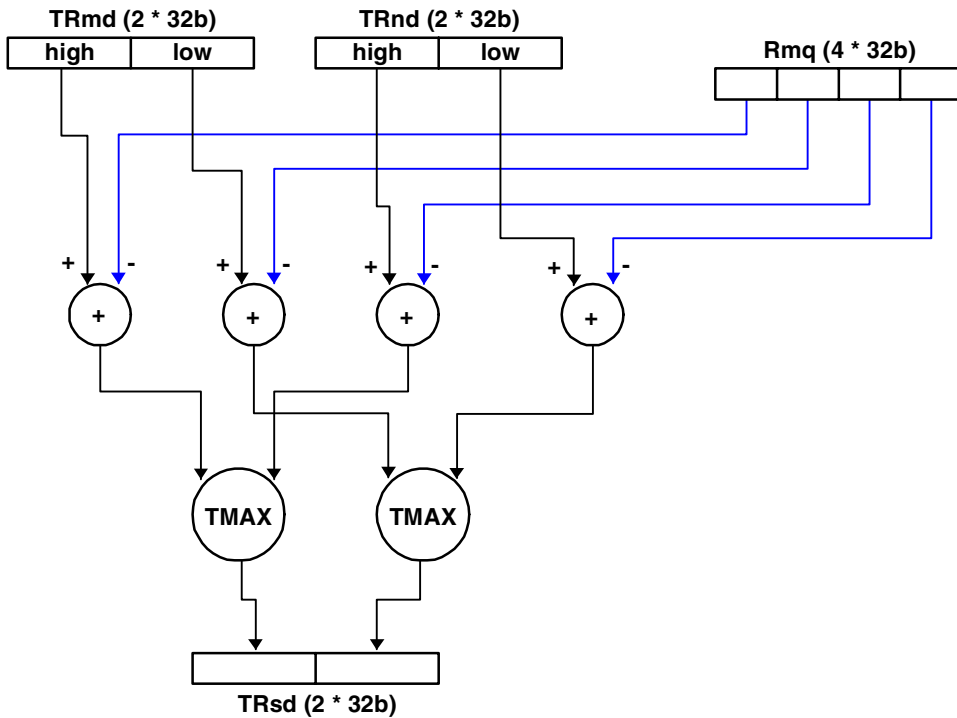


Figure 4-6. $TRsd = TMAX(TRmd - Rmq_h, TRnd - Rmq_l)$

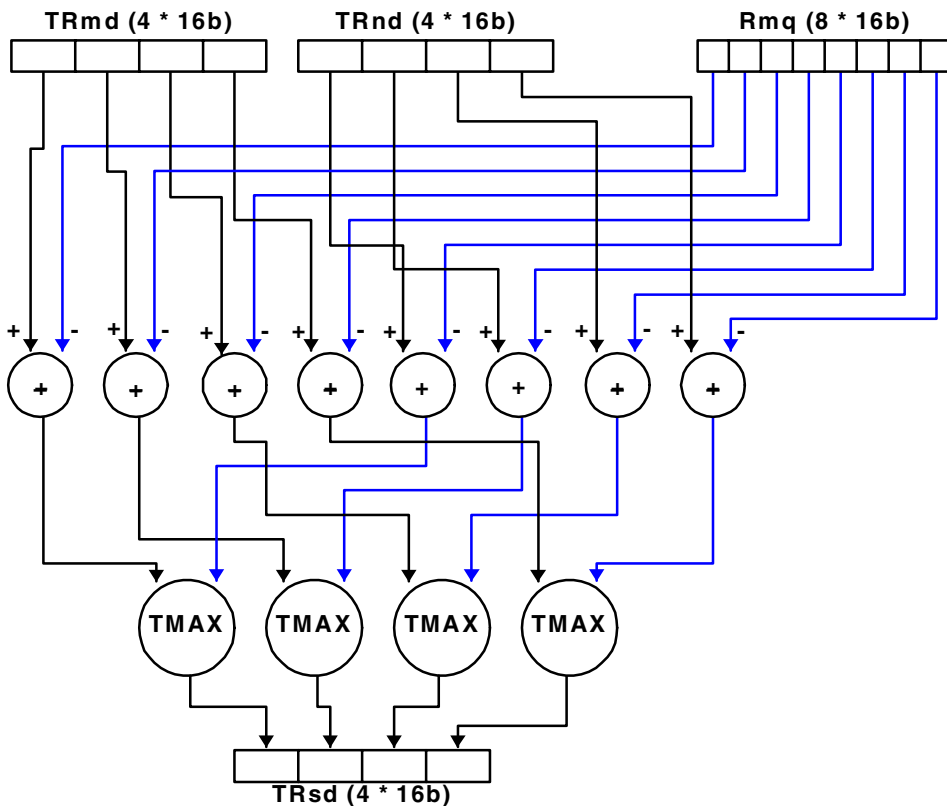


Figure 4-7. $STRsd = TMAX(TRmd - Rmq_h, TRnd - Rmq_l)$

CLU Operations

Trellis Function of the Form

$$Rs = TMAX(TRm, TRn) ;$$

The $TMAX$ function is executed between TRm and TRn , as illustrated in [Figure 4-8](#) and [Figure 4-9](#).

This instruction can be executed in parallel to ALU instructions, shifter instructions, multiplier instructions, and CLU register load. It cannot be executed in parallel to CLU instructions of the same compute block.

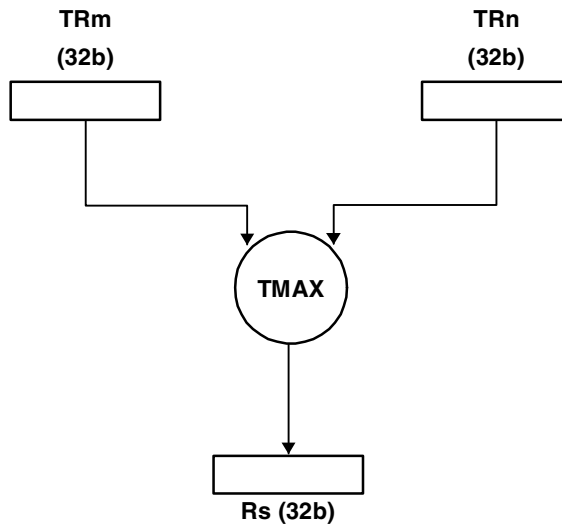


Figure 4-8. $Rs = TMAX(TRm, TRn)$ instruction processing

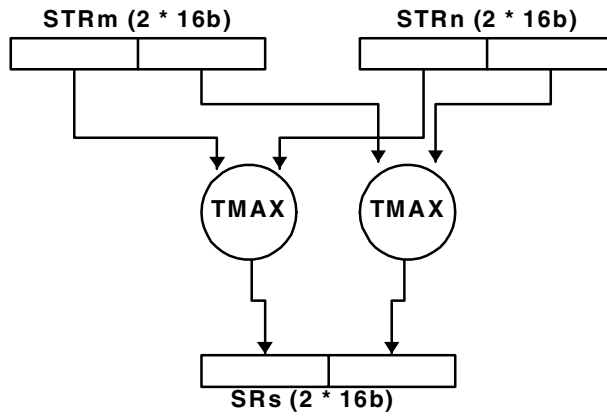


Figure 4-9. $SRs = TMAX(TRm, TRn)$

CLU Operations

Trellis Function of the Form

$$STRsd = \text{MAX}(TRmd + Rmq_h, TRnd + Rmq_l) ;$$

The high part of Rmq is added to $TRmd$, the low part of Rmq is added to $TRnd$, and the MAX function selects the maximum between both add results, as illustrated in Figure 4-10 and Figure 4-11.

Saturation is provided in this instruction. For more details, see “[Saturation Option](#)” on page 3-8.

i On previous TigerSHARC processors, this instruction could not be executed in parallel to CLU and ALU instructions of the same compute block. This limitation does not apply to the ADSP-TS201 processor.

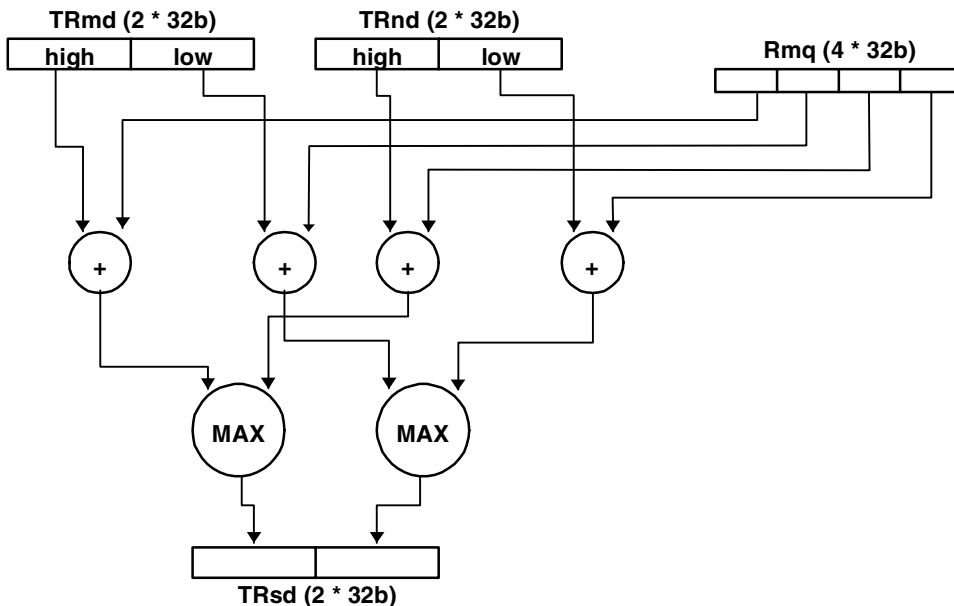


Figure 4-10. $TRsd = \text{MAX}(TRmd + Rmq_h, TRnd + Rmq_l)$

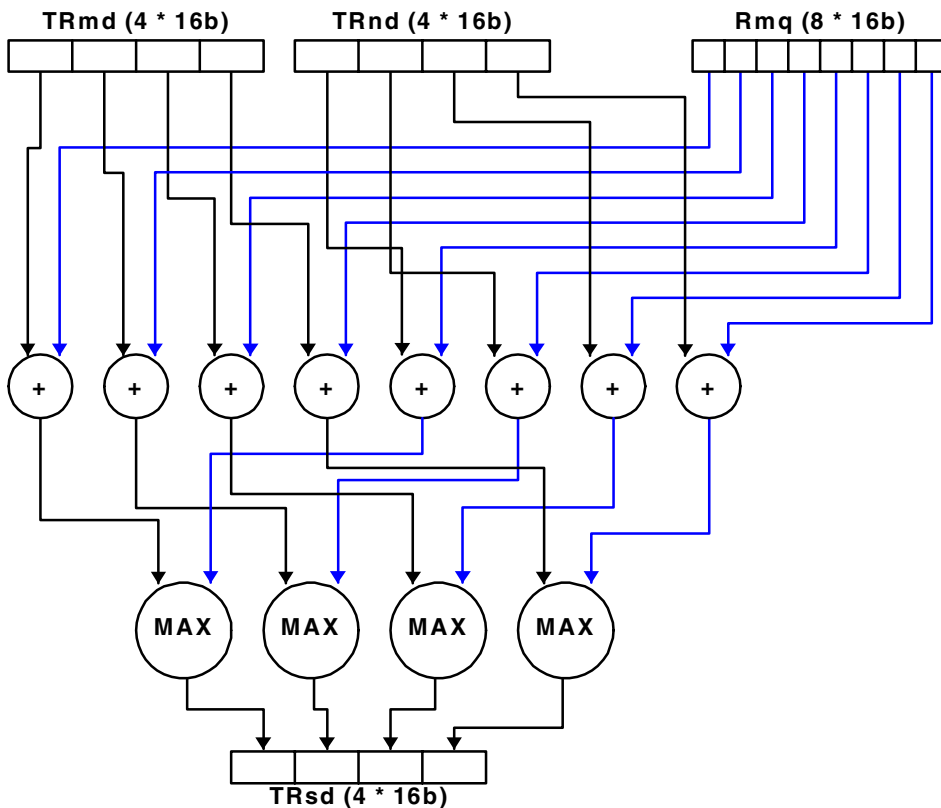


Figure 4-11. $STRsd = \text{MAX}(TRmd + Rmq_h, TRnd + Rmq_l)$

CLU Operations

Trellis Function of the Form

$$STRsd = \text{MAX}(TRmd - Rmq_h, TRnd - Rmq_l) ;$$

The high part of Rmq is subtracted from $TRmd$, the low part of Rmq is subtracted from $TRnd$, and the MAX function selects the maximum between both subtract results, as illustrated in Figure 4-12 and Figure 4-13.

Saturation is provided in this instruction. For more details, see “Saturation Option” on page 3-8.

i On previous TigerSHARC processors, this instruction could not be executed in parallel to CLU and ALU instructions of the same compute block. This limitation does not apply to the ADSP-TS201 processor.

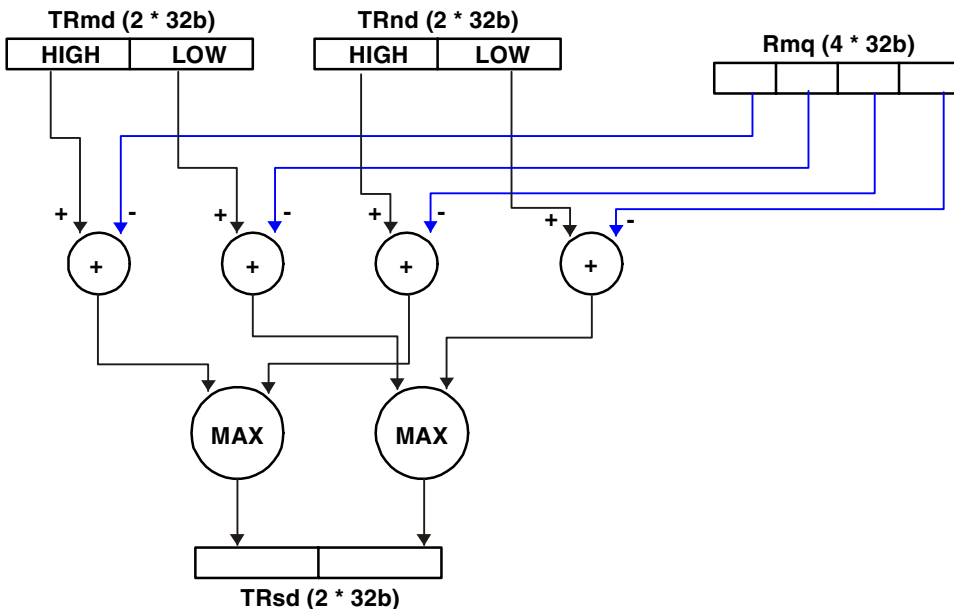


Figure 4-12. $TRsd = \text{MAX}(TRmd - Rmq_h, TRnd - Rmq_l)$

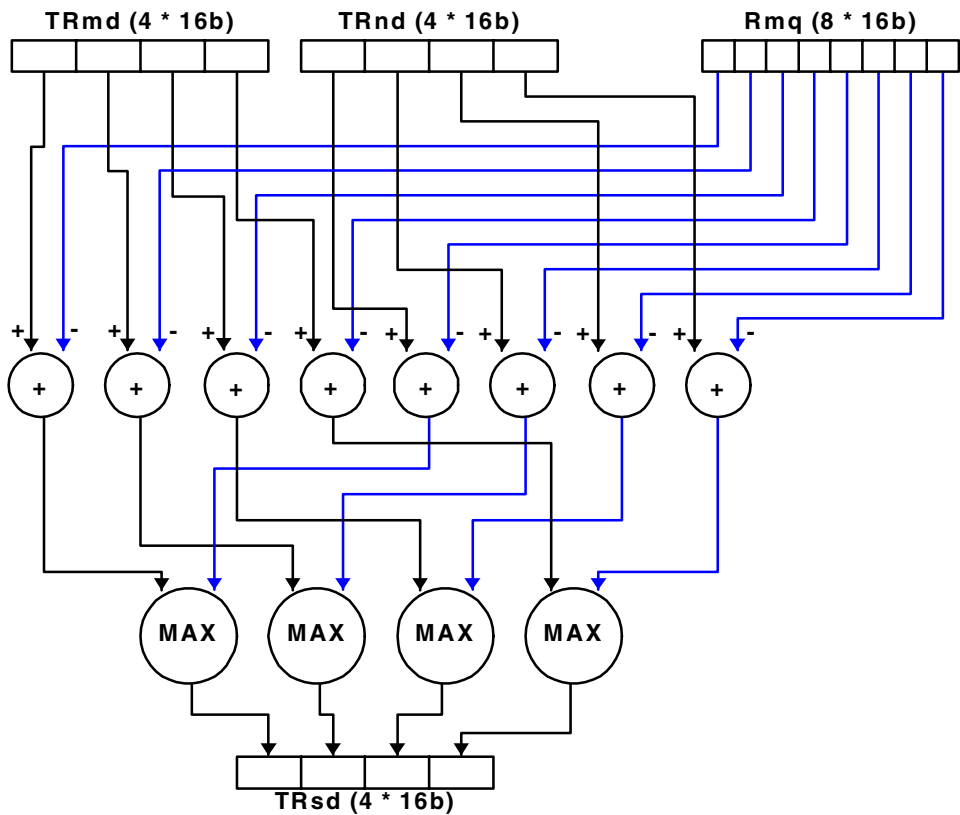


Figure 4-13. $STRsd = \text{MAX}(TRmd - Rmq_h, TRnd - Rmq_l)$

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.

The DESPREAD 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. As shown in Figure 4-14, D_n is composed of D_nI and D_nQ , where I denotes the real part and Q denotes the imaginary part.

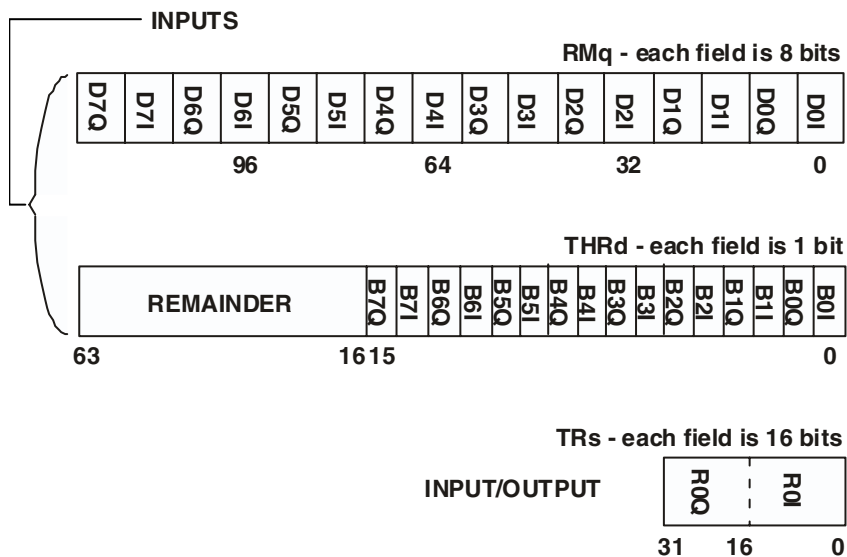


Figure 4-14. Bit Field for TRs += DESPREAD (Rmq , $THRd$) ;

CLU Operations

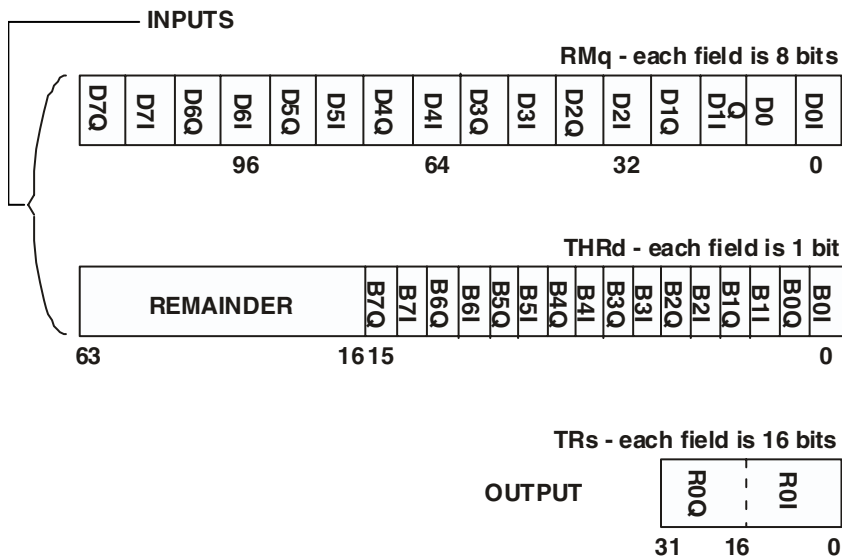



Figure 4-15. Bit Field for $R_s = TR_s$, $TR_s = DESPREAD (R_{mq}, THR_d)$;

The THR_d register is composed of 8 complex numbers $\{B_7...B_0\}$ 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 (when clear) or -1 (when set). THR_d 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.



When executing this instruction in parallel to THR register load, the THR load instruction takes priority on the THR shift in this instruction.

Saturation is active in this instruction. For more details, see “[Saturation Option](#)” on page 3-8.

 On previous TigerSHARC processors, this instruction could not be executed in parallel to ALU instructions of the same compute block. This limitation does not apply to the ADSP-TS201 processor.

Despread Function of the Form TRs += DESPREAD (Rmq, THrd) ;

The TRs register holds a complex word, composed of two shorts—real (least significant) and imaginary (most significant).

The function (illustrated in [Figure 4-16](#)) is:

$$RsI = (\text{sum } (n = 0 \text{ to } 7) (BnI * DnI - BnQ * DnQ))$$

$$RsQ = (\text{sum } (n = 0 \text{ to } 7) (BnI * DnQ + BnQ * DnI))$$

The multiplication is by integer (± 1), the result alignment is to least significant, and the sum is sign-extended.

CLU Operations

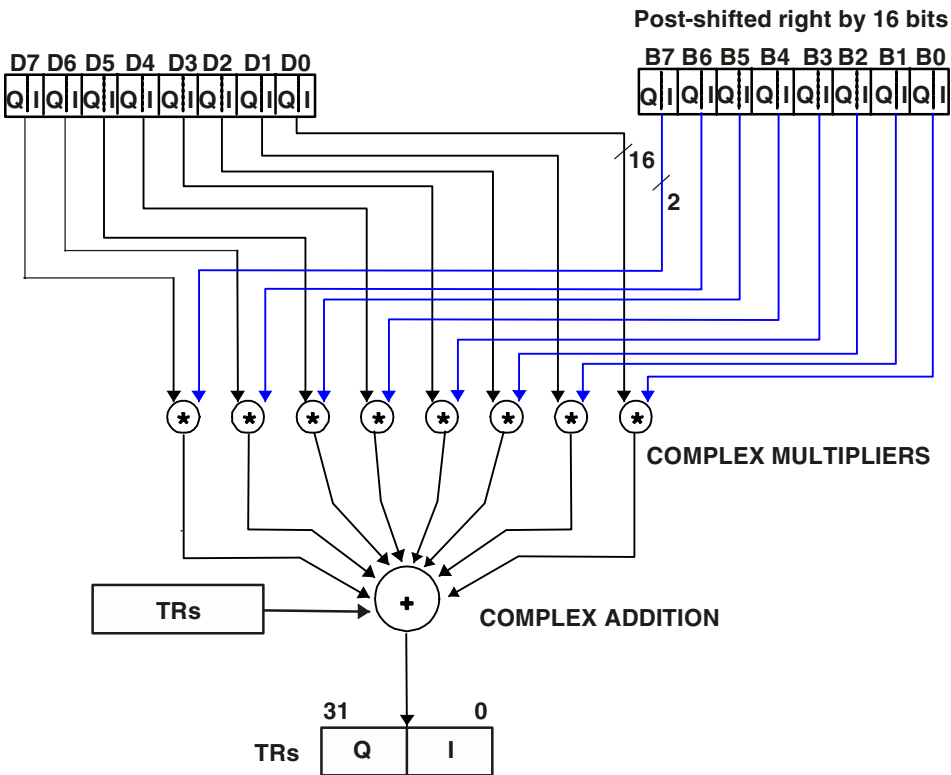


Figure 4-16. $TRs += DESPREAD(Rmq, THRd)$;

Despread Function of the Form

$Rs = TRs, TRs = DESPREAD(Rmq, THRd)$;

The TRs register holds a complex word, composed of two shorts—real (least significant) and imaginary (most significant).

The function (illustrated in Figure 4-17) is:

$$RsI = (\text{sum } (n = 0 \text{ to } 7) (BnI * DnI - BnQ * DnQ))$$

$$RsQ = (\text{sum } (n = 0 \text{ to } 7) (BnI * DnQ + BnQ * DnI))$$

The value of TRs before the operation is stored in Rs .

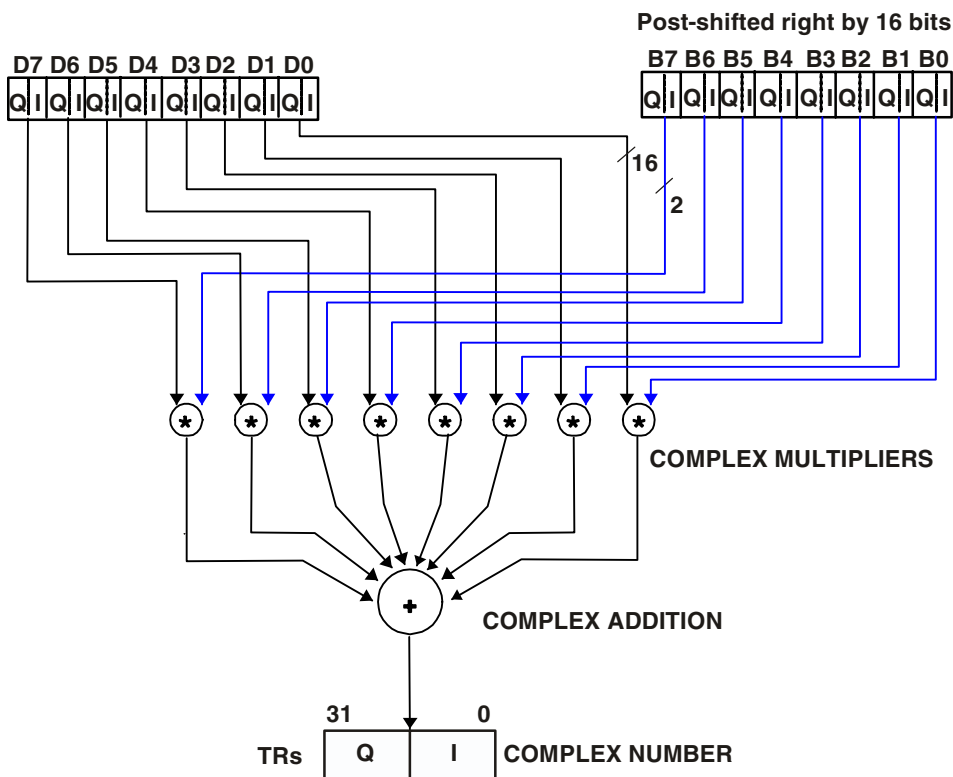


Figure 4-17. $Rs = TRs$, $TRs = \text{DESPREAD } (Rmq, \text{THRd})$;

CLU Operations

Despread Function of the Form

$Rsd = TRsd$, $TRsd = DESPREAD (Rmq, THRd)$;

The function (illustrated in [Figure 4-18](#) and [Figure 4-19](#)) is:

$$R0I = (\text{sum } (n = 0 \text{ to } 3) (BnI * DnI - BnQ * DnQ))$$

$$R0Q = (\text{sum } (n = 0 \text{ to } 3) (BnI * DnQ + BnQ * DnI))$$

$$R1I = (\text{sum } (n = 4 \text{ to } 7) (BnI * DnI - BnQ * DnQ))$$

$$R1Q = (\text{sum } (n = 4 \text{ to } 7) (BnI * DnQ + BnQ * DnI))$$

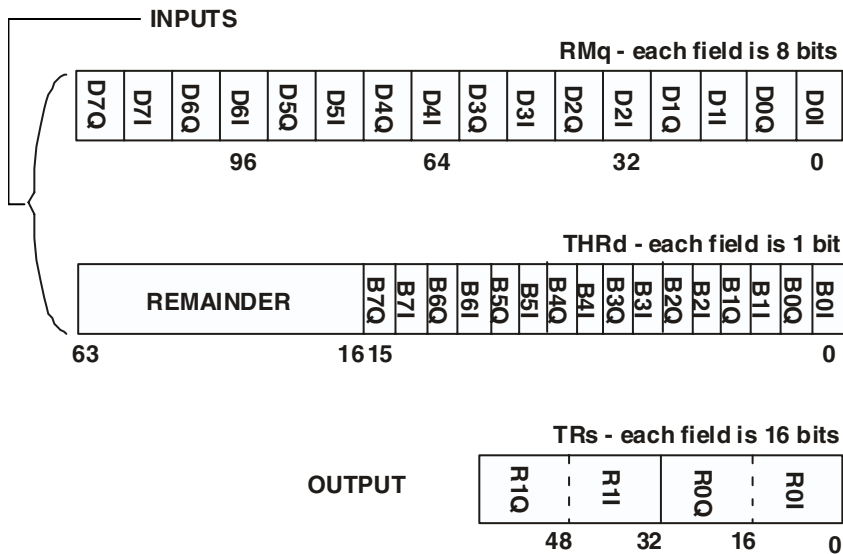


Figure 4-18. Bit Fields for
 $Rsd = TRsd$, $TRsd = DESPREAD (Rmq, THRd)$;



Note that output is two words.

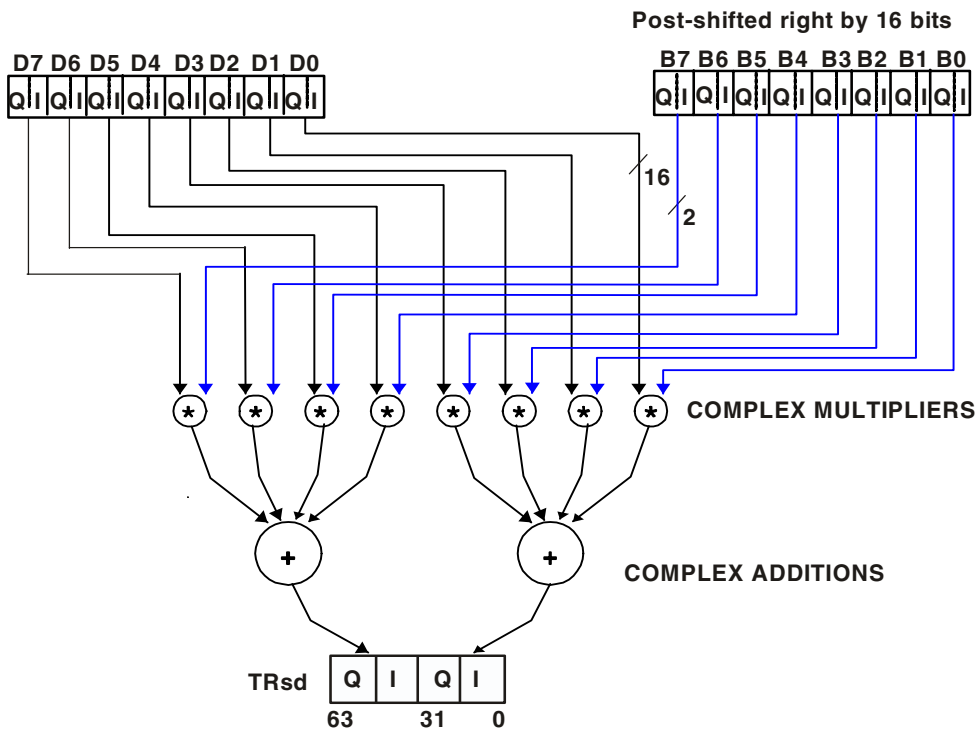


Figure 4-19. Rsd = TRsd, TRsd = DESPREAD (Rmq, THRD) ;

Cross Correlations Function

The `XCORRS` instruction correlates long input sequence (such as 2048 complex input numbers of an 8-bit pilot) with a known reference sequence for multiple delays. It is convenient to view the `XCORRS` instruction as a single cycle execution of 16 parallel `DESPREAD` instructions. Some important features of the `XCORRS` instruction include:

- Clear (`CLR`) option, providing a mechanism to clear the `TR` registers before beginning a new cross correlation
- Cut inputs (`CUT`) option, providing a mechanism to discard some input data numbers (for initial and final cycles; lower and upper triangles)
- Extended-precision (`EXT`) option, supporting 32-bit complex data input and 64-bit complex data accumulation (instead of the default 16-bit complex data input and 32-bit complex data accumulation)
- Outputs correlation strength for each delay

The `XCORRS` equations for every part X or Y perform the operations shown in [Figure 4-20](#) where (depending on the `CUT` value) the inputs are shown in [Figure 4-21](#). The `CUT` function values are shown in [Figure 4-22](#).

CLU Operations

Figure 4-24 and Figure 4-25 show the bit placement for an `XCORRS` instruction using single-precision inputs (default operation). Note that Figure 4-24 and Figure 4-25 represent 16 multiply accumulate operations.

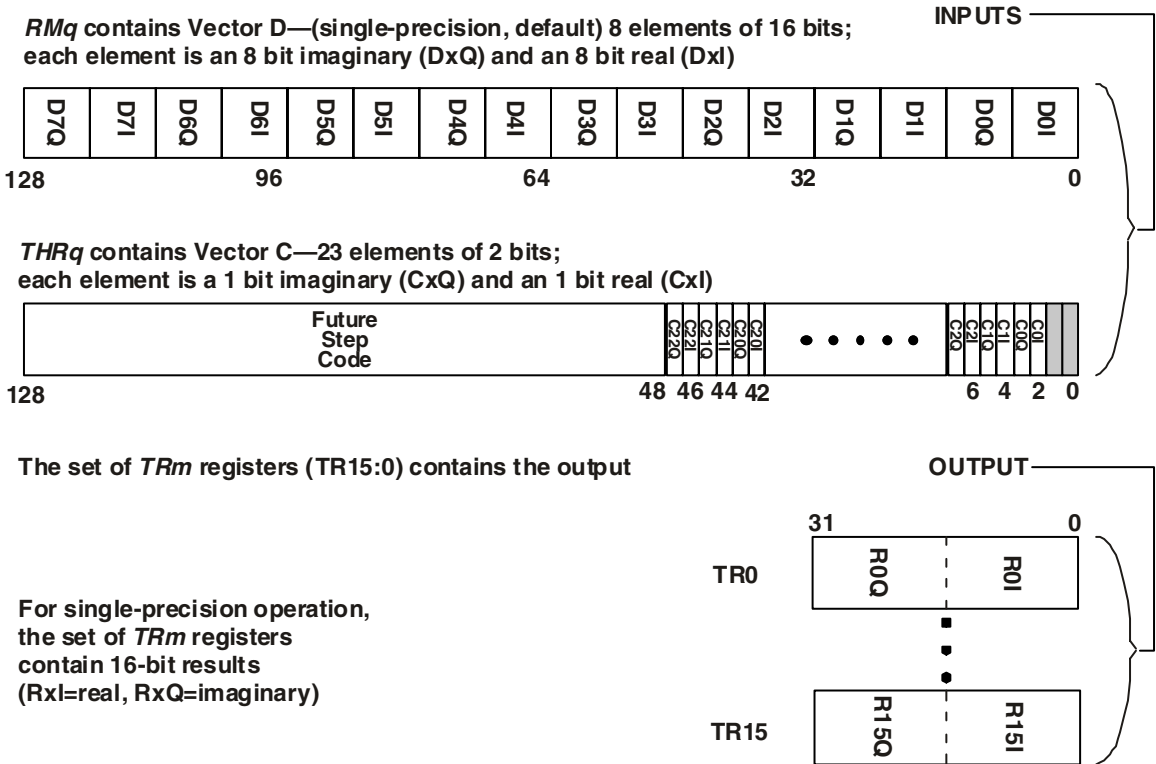


Figure 4-24. Bit Fields for `TRm = XCORRS(Rmq, THRd)`;

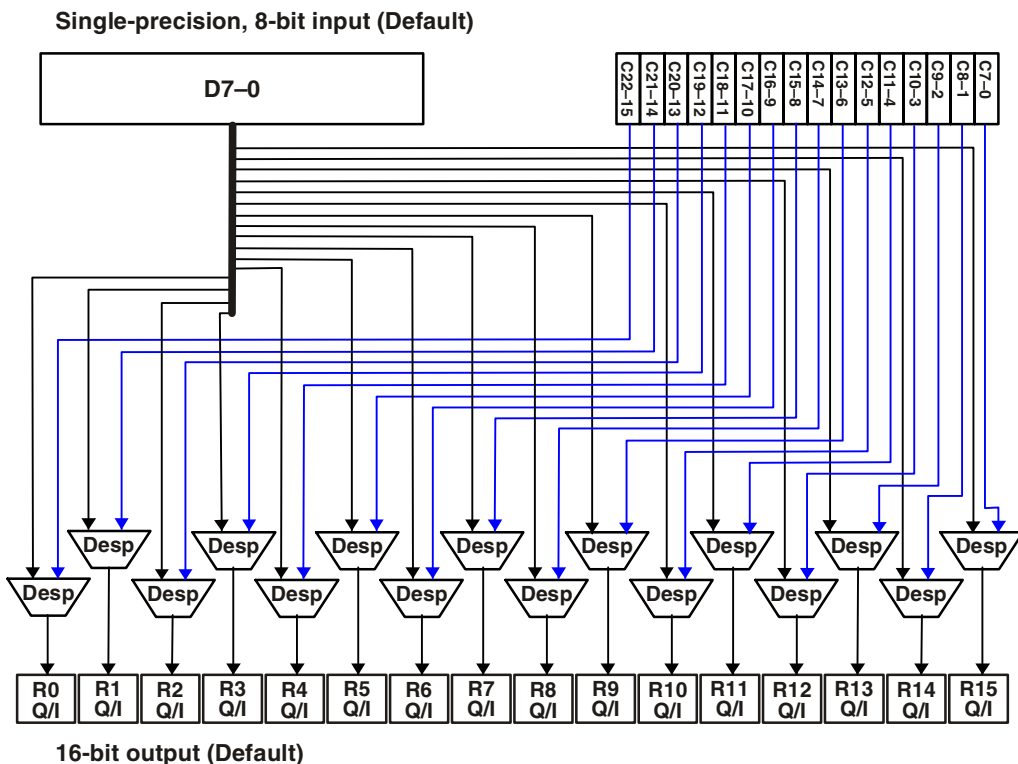


Figure 4-25. $TR_m = XCORRS(Rmq, THR_d)$;

Figure 4-26 and Figure 4-27 show the bit placement for an XCORRS instruction using extended-precision inputs (EXT option operation. Note that Figure 4-26 and Figure 4-27 represent 8 multiply accumulate operations.

CLU Operations

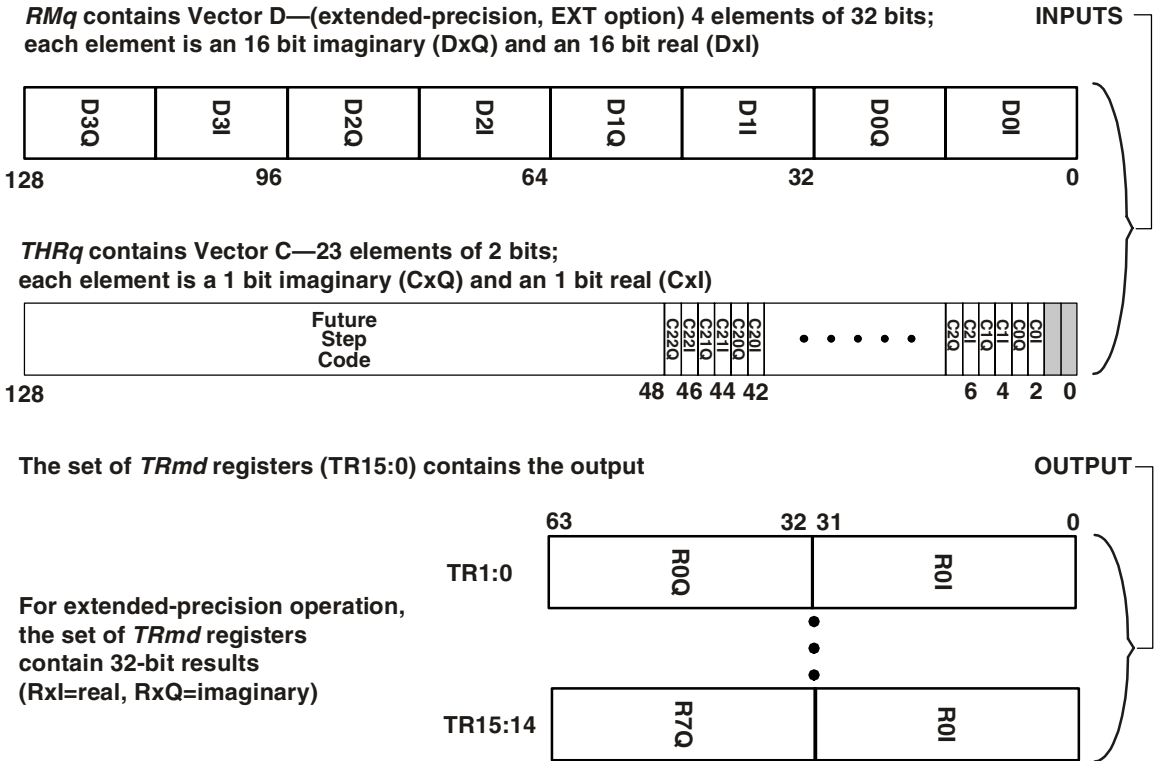


Figure 4-26. Bit Fields for $TR_m = XCORRS(R_{mq}, THR_d)$ (EXT);

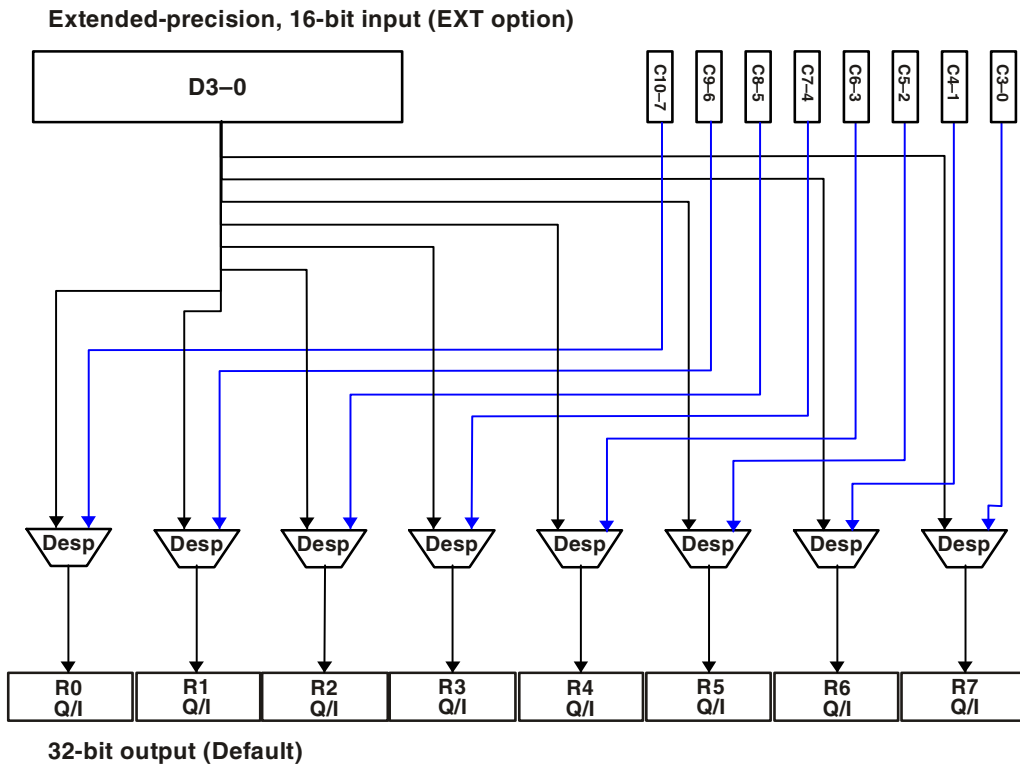


Figure 4-27. $TR_m = XCORRS(R_mq, THR_d)$ (EXT);



When executing this instruction in parallel to THR register load, the THR load instruction takes priority on the THR shift in this instruction.

CLU Operations

When the CUT option is omitted, the $CUT=0$, and the code index is as shown in Figure 4-28. Note that each code index in Figure 4-28 maps to a “C” input in Figure 4-25 on page 4-33.

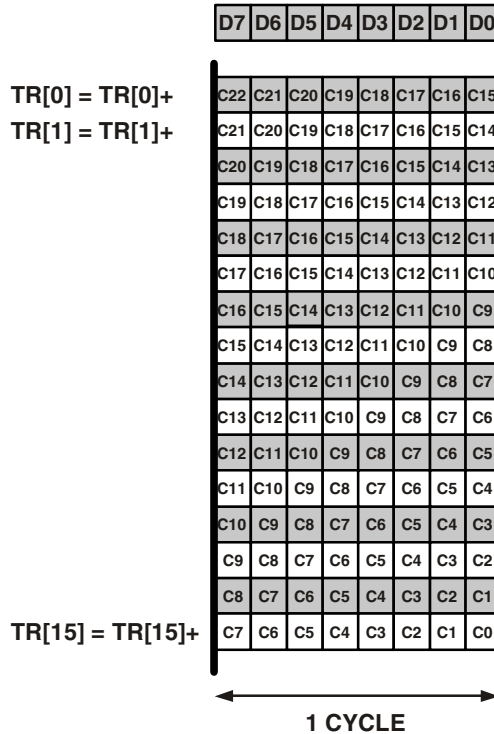


Figure 4-28. $CUT = 0$ Definition – Code Index

When the CUT option is negative (for example, $CUT = -k$), all multiply-and-accumulate operations under $C[k]$ are ignored.

Figure 4-29 shows an example with $CUT = -7$.

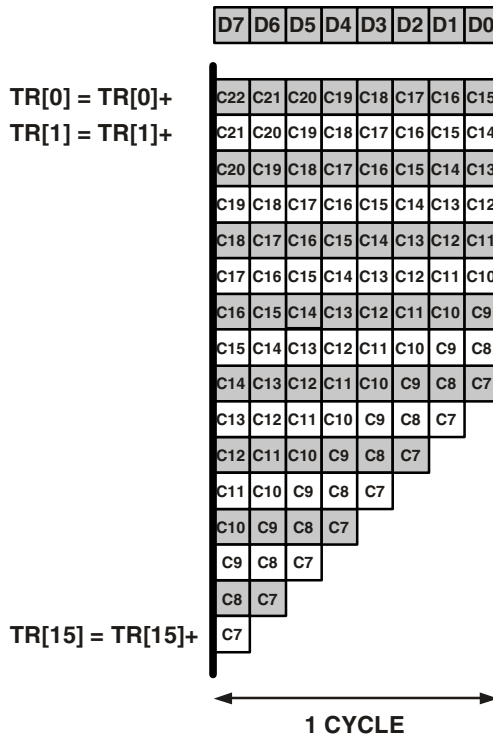


Figure 4-29. $CUT =$ Negative Example – Code Index

CLU Operations

When the `CUT` option is positive (for example, `CUT = +k`), all multiply-and-accumulate operations for `C[k]` and above are ignored. [Figure 4-30](#) shows an example with `CUT = +15`.

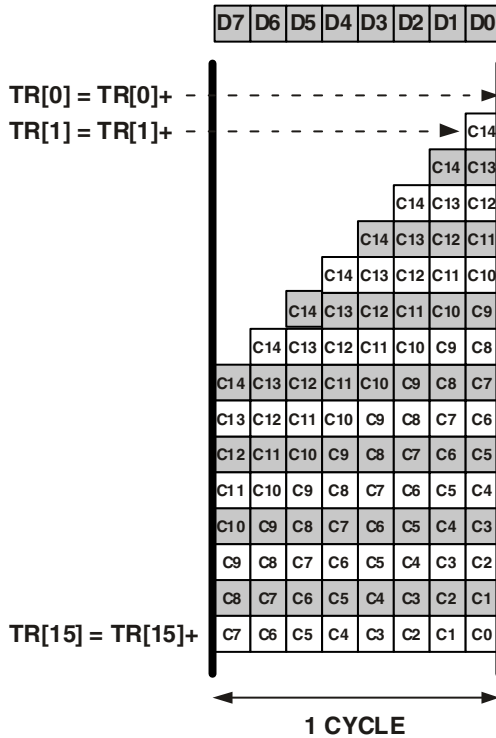


Figure 4-30. `CUT = Positive Example` – Code Index

Add/Compare/Select Function

In the ACS function $TRsq = ACS (TRmd, TRnd, Rm)$; , 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 4-31.

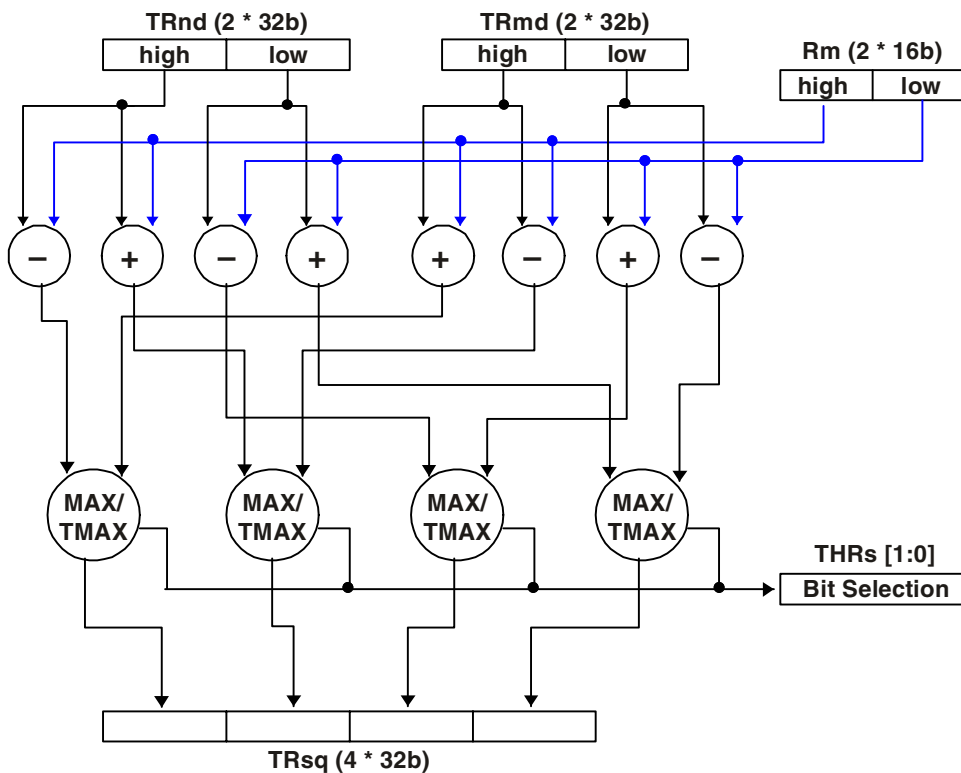


Figure 4-31. $TRsq = ACS (TRmd, TRnd, Rm)$;

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

CLU Operations

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 4-32.

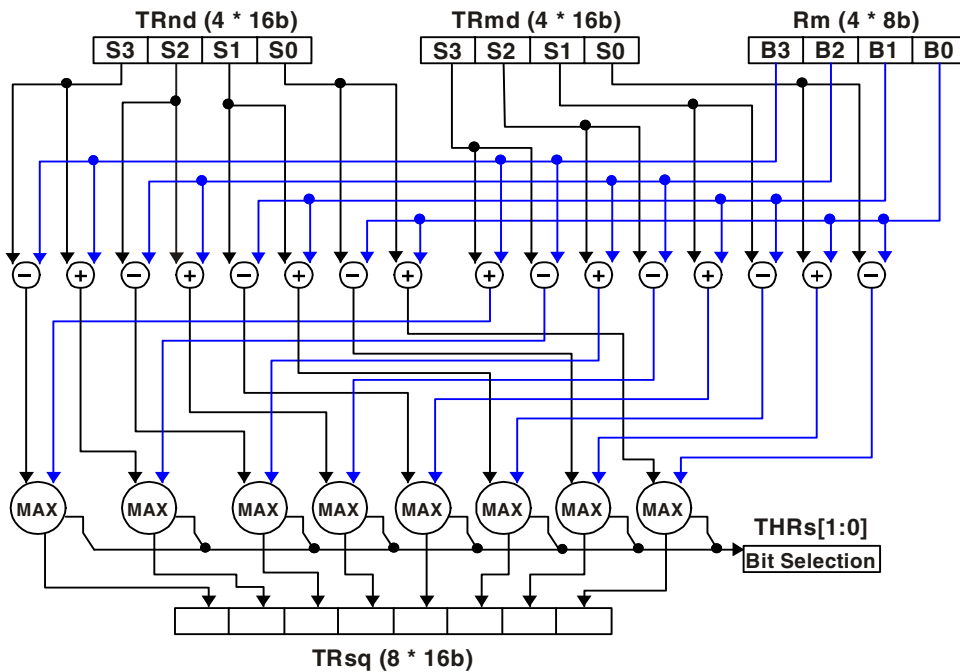


Figure 4-32. $STRsq = ACS (TRmd, TRnd, Rm)$;

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.

Optionally, a trellis register transfer can be added for dual operation, as in:
 $Rsq = TRaq, \{S\}TRsq = ACS (TRmd, TRnd, Rm);.$

CLU Instruction Options

Some of the CLU 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 CLU instructions, see [“CLU Instruction Summary” on page 4-46](#). The CLU instruction options include:

- (TMAX) selects TMAX operation instead of MAX operation¹
- (CUT *imm*) selects data to cut from cross correlation set²
- (CLR) clears the selected trellis register prior to cross correlation summation²
- (EXT) selects extended-precision for cross correlation input/results²
- (NF) no flag update

¹ ACS instruction only

² XCORRS instruction only

CLU Operations

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 4-2](#) shows the flags in XSTAT or YSTAT that indicate CLU status for the most recent CLU operation.

Table 4-2. 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 4-3](#) shows the flags in XSTAT or YSTAT that indicate CLU sticky status for the most recent CLU operation. Once set, a sticky flag remains high until explicitly cleared.

Table 4-3. CLU Status Sticky Flags

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

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

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

 When the (NF) option is used, the compute unit does not update the flags with status from the operation. Programs can use the (NF) option with all CLU instruction except for transfer TRx, THRx, and CMCTL register instructions.

CLU Examples

The 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 CDMA systems

[Listing 4-1](#) provides a DESPREAD instruction example, which highlights the calculations that go into the instruction.

Listing 4-1. DESPREAD (CLU Instruction) Example

With the following values in the registers:

```
xTR0 = 0x00010002
xTHR1 = 0x00001111
xTHRO = 0x11110000
xR3 = 0x00010203
xR2 = 0x04050607
```

CLU Examples

xR1 = 0x08090a0b

xR0 = 0x0c0d0e0f

Execute the following instruction:

```
xTR0 += DESPREAD (R3:0, THR1:0) ;;
```

Yielding the following values in the registers:

xTR0 = 0x00780008

xTHR1 = 0x00000000

xTHRO = 0x11111111

Real Result = (0x01 - 0x00)
+ (0x03 - 0x02)
+ (0x05 - 0x04)
+ (0x07 - 0x06)
+ (0x09 - 0x08)
+ (0x0b - 0x0a)
+ (0x0d - 0x0c)
+ (0x0f - 0x0e)
= 0x0008

Imaginary Result = (0x01+ 0x00)
+ (0x03 + 0x02)
+ (0x05 + 0x04)
+ (0x07 + 0x06)
+ (0x09 + 0x08)
+ (0x0b + 0x0a)
+ (0x0d + 0x0c)
+ (0x0f + 0x0e)
= 0x78

Note that the THR values have been updated and shifted to the right by 16 bits and that the TR0 result is promoted. Running the instruction again:

```
xTR0 += DESPREAD (R3:0, THR1:0) ;;
```

Before accumulation, the results are:

$$\begin{aligned}
 \textit{Real Result} &= (-0x01 + 0x00) \\
 &+ (-0x03 + 0x02) \\
 &+ (-0x05 + 0x04) \\
 &+ (-0x07 + 0x06) \\
 &+ (-0x09 + 0x08) \\
 &+ (-0x0b + 0x0a) \\
 &+ (-0x0d + 0x0c) \\
 &+ (-0x0f + 0x0e) \\
 &= 0xFFF8
 \end{aligned}$$

$$\begin{aligned}
 \textit{Imaginary Result} &= (-0x01 - 0x00) \\
 &+ (-0x03 - 0x02) \\
 &+ (-0x05 - 0x04) \\
 &+ (-0x07 - 0x06) \\
 &+ (-0x09 - 0x08) \\
 &+ (-0x0b - 0x0a) \\
 &+ (-0x0d - 0x0c) \\
 &+ (-0x0f - 0x0e) \\
 &= 0xFF88
 \end{aligned}$$

Adding the accumulation of TR0 from the last instruction gives a final result of:

```
xTR0 = 0x00000000
xTHR1 = 0x00000000
xTHRO = 0x00001111
```

Because the input (R3:0) was the same in the two DESPREAD instructions, and the code sequence changed from $1+j$ to $-1-j$, the accumulation cancels out completely after the second instruction.

CLU Instruction Summary

[Listing 4-2](#) “CLU Instructions” shows the CLU 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 2-6](#). 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*, *Rnd*), or quad (*Rsq*) 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 1-23](#) and [“Instruction Parallelism Rules” on page 1-27](#).

Listing 4-2. CLU Instructions

```
{X|Y|XY}{S}TRsd = TMAX(TRmd + Rmq_h, TRnd + Rmq_l) {(NF)} ;
{X|Y|XY}{S}TRsd = TMAX(TRmd - Rmq_h, TRnd - Rmq_l) {(NF)} ;
{X|Y|XY}{S}Rs = TMAX(TRm, TRn) {(NF)} ;
{X|Y|XY}{S}TRsd = MAX(TRmd + Rmq_h, TRnd + Rmq_l) {(NF)} ;
{X|Y|XY}{S}TRsd = MAX(TRmd - Rmq_h, TRnd - Rmq_l) {(NF)} ;
/* where Rmq_h must be the upper half of a quad register and
Rmq_l must be the lower half of the SAME quad register */

{X|Y|XY}Rs = TRm ;
{X|Y|XY}Rsd = TRmd ;
{X|Y|XY}Rsq = TRmq ;
```

```

{X|Y|XY}Rs = THRm ;
{X|Y|XY}Rsd = THRmd ;
{X|Y|XY}Rsq = THRmq ;
{X|Y|XY}Rs = CMCTL ;
/* For TR, THR, and CMCTL register load syntax, see "Shifter Quick Refer-
ence" on page A-9. */

{X|Y|XY}TRs += DESPREAD (Rmq, THRd) ;
{X|Y|XY}Rs = TRs, TRs = DESPREAD (Rmq, THRd) {(NF)} ; (dual op.)
{X|Y|XY}Rsd = TRsd, TRsd = DESPREAD (Rmq, THRd) {(NF)} ; (dual op.)

{X|Y|XY}TRsa = XCORRS (Rmq, THRnq) {(CUT
<Imm>|R)}{(CLR)}{(EXT)}{(NF)} ;
{X|Y|XY}Rsq = TRbq, TRsa = XCORRS (Rmq, THRnq)
{(CUT <Imm>|R)}{(CLR)}{(EXT)}{(NF)} ; (dual op.)
/* where TRsa = TR15:0 or TR31:16 */

{X|Y|XY}{S}TRsq = ACS (TRmd, TRnd, Rm) {(TMAX)} {(NF)} ;
{X|Y|XY}Rsq = TRaq, {S}TRsq = ACS (TRmd, TRnd, Rm)
{(TMAX)}{(NF)} ; (dual op.)

/* For PERMUTE instruction syntax, see "ALU Quick Reference" on
page A-2, Listing A-3. */

```

CLU Instruction Summary

5 MULTIPLIER

The ADSP-TS201 TigerSHARC processor core contains two computation units known as compute blocks. Each compute block contains a register file and four independent computation units—an ALU, a CLU, a multiplier, and a shifter. The multiplier is highlighted in [Figure 5-1](#). The multiplier takes its inputs from the register file, and returns its outputs to the register file or MR (accumulator) registers.

The chapter describes:

- “Multiplier Operations” on page 5-5
- “Multiplier Examples” on page 5-24
- “Multiplier Instruction Summary” on page 5-26

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-accumulate 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.

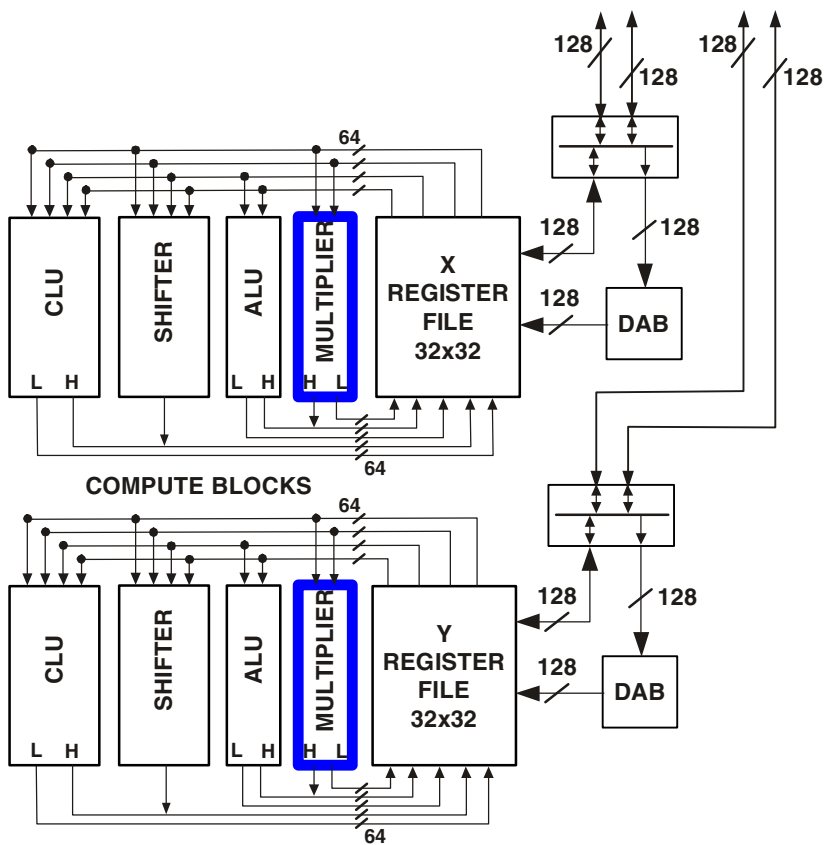


Figure 5-1. Multipliers in Compute Block X and Y¹

- 1 On the outputs of the compute units, H indicates high half of quad result (bits 128–64), and L indicates low half of quad result (bits 63–0). For a result smaller or equal to 64 bits, compute units use the (L) lower result output.

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-, 20-, 32-, or 40-bit results
 - Two 32-bit (normal) input operands with a 32-, 40-, 64-, or 80-bit result
- Fixed-point fractional and integer *complex multiply-accumulate operations* support:
 - Two 32-bit (16-bit real, 16-bit imaginary) input operands with either a 40-bit (20-bit real, 20-bit imaginary) result or an 80-bit (40-bit real, 40-bit imaginary) result
- Floating-point fractional *multiply operations* support:
 - Two 32-bit (single-precision) input operands (IEEE standard) with a 32-bit result
 - Two 40-bit (extended-precision) input operands with a 40-bit result
- Fixed-point *data compaction operations* support:
 - 20-bit (short) input operands
 - 40-bit (normal) input operands
 - 80-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 result.

The ADSP-TS201 processor supports complex multiply-accumulates. Complex numbers are represented by a pair of short words in a 32-bit register. The low half of the input operand registers (Rm_L , Rn_L) represent the real part, and the high half of the input operand registers (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:

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

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

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


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

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

The complex conjugate option is denoted with a (J) following the instruction. (See “[Complex Conjugate Option](#)” on page 5-19.)

The ADSP-TS201 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 2-6. For information on data size selection for multiplier instructions, see the examples in “[Multiplier Operations](#)” on page 5-5.

 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 1-23. For a list of multiplier instructions and their syntax, see “[Multiplier Instruction Summary](#)” on page 5-26.

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 single-precision and extended-precision. The input operands and output result of most operations is the compute block register file.


Multiplier Operations

The multiplier has one special purpose, five-word register—the MR register—for accumulated results. The ADSP-TS201 processor 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 MR4 register, see “Multiplier Result Overflow (MR4) Register” on page 5-19.

Figure 5-4 on page 5-9 through Figure 5-6 on page 5-10 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 rounded 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 set of four multiply operations of 16-bit signed
fractional operands:
XR5_upper_half * XR1_upper_half -> XR3_upper_half,
XR5_lower_half * XR1_lower_half -> XR3_lower_half,
XR4_upper_half * XR0_upper_half -> XR2_upper_half,
XR4_lower_half * XR0_lower_half -> XR2_lower_half;
*/
```

```

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. */

```

i 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 single-precision input operands XR1 and XR0 (IEEE format); the DSP places the single precision result in XR2. */
YFR1:0 = R5:4 * R3:2 ::
/* This is a floating-point multiply of two extended-precision 40-bit input operands YR5:4 and YR3:2; the DSP places the 40-bit result in YR1:0. */

```

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

Multiplier Operations

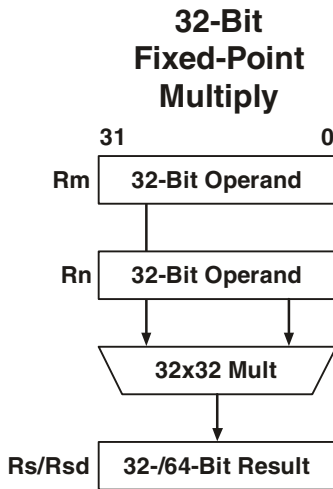
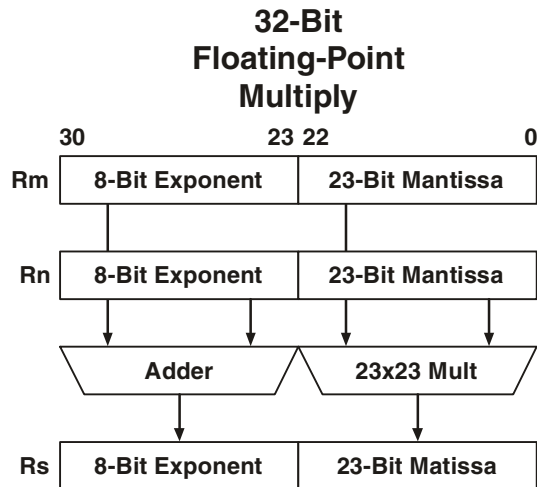


Figure 5-2. 32-Bit Fixed-Point Multiplier Operations



Note: 31 = sign-bit

Figure 5-3. 32-Bit Floating-Point Multiplier Operations

32-Bit Multiply-Accumulate

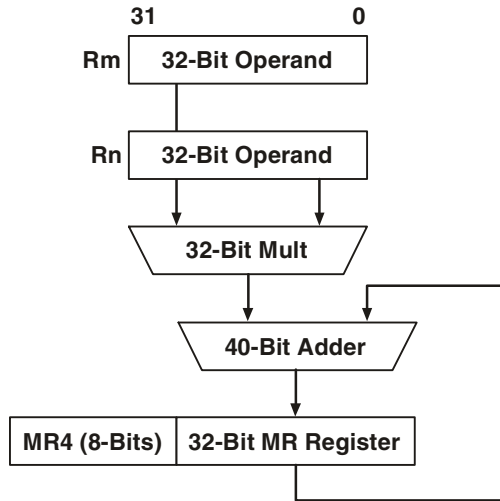


Figure 5-4. 32-Bit Multiply-Accumulate Operations

40-Bit Floating-Point Multiply

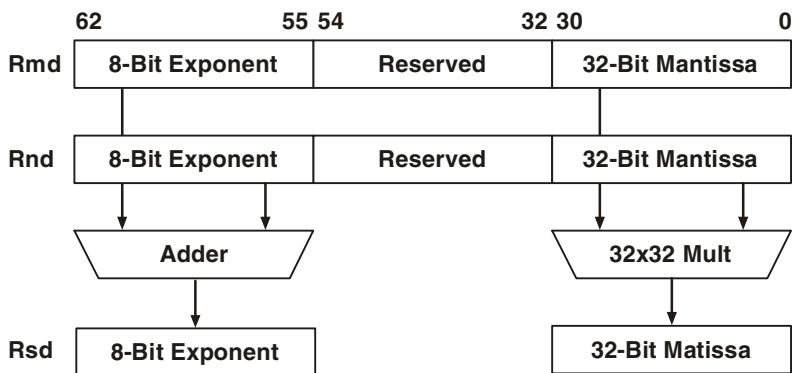


Figure 5-5. 40-Bit Multiplier Operations

Multiplier Operations

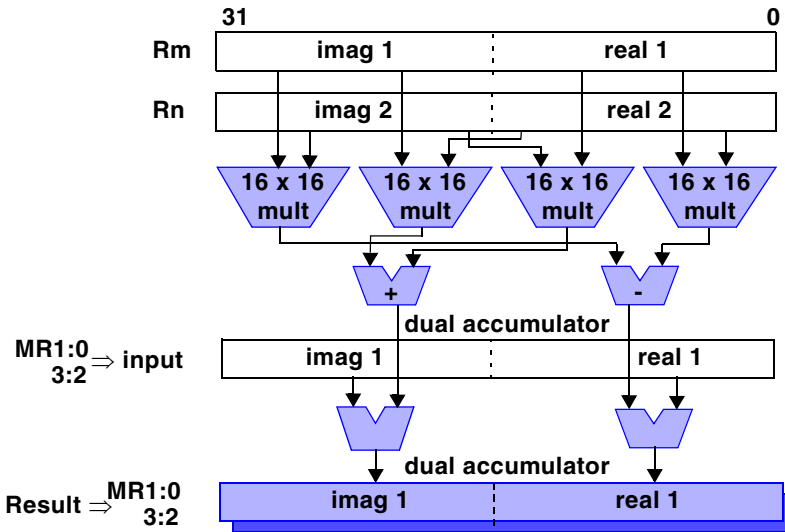


Figure 5-6. 16-Bit Complex Multiplier Operations

Quad 16-Bit Fixed-Point Multiply

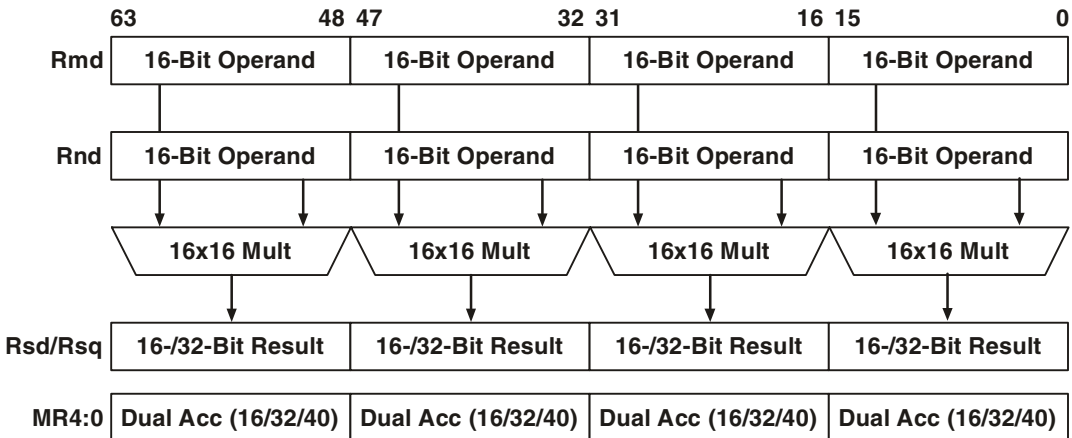


Figure 5-7. 16-Bit Multiplier Operations

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 5-26](#).

The multiplier instruction options include:

- () signed operation, no saturation¹, round-to-nearest even², fractional operation³
- (U) unsigned operation, no saturation¹, round-to-nearest even², fractional operation³
- (nU) the Rm operand is signed input, the Rn operand is unsigned input, no saturation¹, round-to-nearest even², fractional operation³
- (I) signed operation, integer operation³
- (S) signed operation, saturation¹
- (T) signed operation, truncation⁴
- (C) clear operation
- (CR) clear/round operation

¹ Where saturation applies

² Where rounding applies

³ Where fixed-point operation applies

⁴ Where truncation applies

Multiplier Operations

- (J) complex conjugate operation
- (NF) no flag update

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 rounded  
unsigned 32-bit result in XR2. */
```

```
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  
truncated 32-bit result in XR2. */
```

Signed/Unsigned Option

The processor 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-magnitude. 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) R_m is signed, R_n 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 processor 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. For information on the supported numeric formats, see [“Numeric Formats” on page 2-16](#).

The integer or 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 active¹ 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 ; however, there could not be a negative result for the unsigned operation. 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

¹ Except for $R_{sd} = R_m * R_n$, $R_{sq} = R_{md} * R_{nd}$, where it is not optional and implied in the instruction.

Multiplier Operations

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, $0 \times 7F \dots F$), if saturation¹ is chosen.

- 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 5-24). 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 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 5-24.

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

- $0 \times 7F \dots F$ – if operation is signed and result is positive
- $0 \times 80 \dots 0$ – if operation is signed and result is negative
- $0 \times FF \dots F$ – if operation is unsigned and result is positive
- $0 \times 00 \dots 0$ – 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 operation. The processor supports two methods for rounding—round-toward-zero and round-toward-nearest. The rounding methods comply with the IEEE 754 standard and have these definitions:

- Round-toward-nearest-even – not using the (T) option with either floating-point operands or fixed-point fractional operands. 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 the (T) option with floating-point operands. 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.
- Round-toward-zero – using the (T) option with fixed-point fractional operands. If the result before rounding has extra bits extending below the least significant bit (LSB) of the operation's format, these bits are ignored (truncated). When this truncation occurs, the result is rounded towards negative infinity (for positive results, rounds to the smaller value; for negative results, rounds to the larger negative value)

For example, a fixed-point multiply yields a 16-bit fractional result of 0x0.1FFFF before truncation. With truncation, this positive result becomes 0x0.1FFF, which is the smaller value. By comparison, a fixed-point multiply yields a 16-bit fractional result of 0x.FFFFF (or -2^{20}) before truncation. With truncation, this negative result becomes 0x.FFFF (or 2^{16}), which is the larger negative value.

Multiplier Operations

Statistically, using round-to-nearest-even, rounding up occurs as often as rounding down, so there is no significant rounding 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 operation.

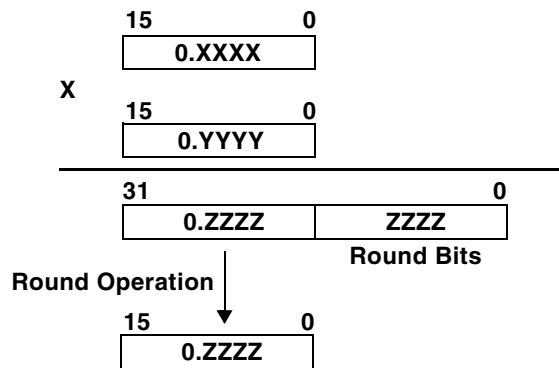


Figure 5-8. Rounding Multiplier Results

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 nonzero:

- If bit 15 is zero, the result is not incremented
- If bit 15 is 1 and bits 14–0 are nonzero, the LSB 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 5-17.](#)

Rounding options are:

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

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 relevant MR register part before the accumulate operation.

Multiply-accumulate operations (without an MR move¹) 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 5-15.](#)

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

¹ This is not a criterion. The distinction is the destination result width:
 $R_s = MR_a, MR_a += R_m * R_n (R).$

Multiplier Operations

(CR) Set target MR to zero and set round bit prior to multiply-accumulate operation



The 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 CR 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 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 two shorts are exactly the expected results. Note that this is round-to-nearest, and not round-to-nearest even.

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 5-15](#).

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 the R_n operand, negating the imaginary part of R_n . For more information, see the discussion [on page 5-4](#). The options are:

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

No Flag Update Option

Almost all compute operations generate status, which affects the compute unit's status register. Often, it is useful in some applications to retain status from an operation. There are a number of techniques for storing status (for example, storing the status register contents or loading flag values into the static flags (SFREG) register). The No Flag Update (NF) option provides a different method for working with status. Instead of storing an operation's status for future usage, programs can prevent status from being overwritten by using the (NF) option on following operations to prevent status generation. There are some restrictions on how the (NF) option can be used with conditional instructions. For more information, see [“Conditional Execution” on page 8-14](#).

Multiplier Result Overflow (MR4) Register

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

- [Figure 5-9](#), Result is a long word (80-bit accumulation)
- [Figure 5-10](#), Result is normal word (40-bit accumulation)
- [Figure 5-11](#), Result is short word (20-bit accumulation)

Multiplier Operations

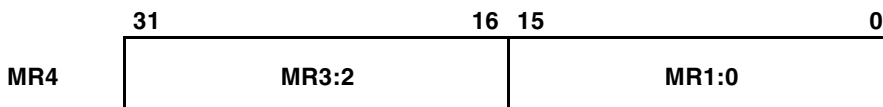


Figure 5-9. MR4 for Long Word Result (80-Bit Accumulation)

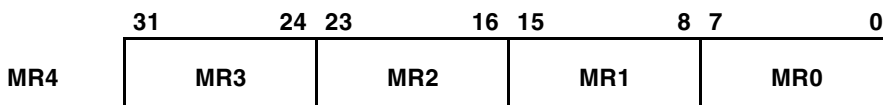
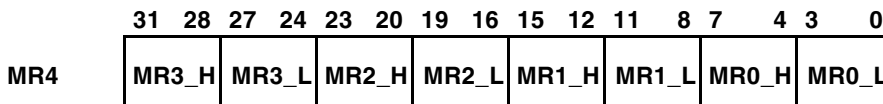


Figure 5-10. MR4 for Normal Word Result (40-Bit Accumulation)



_H indicates high short word field
_L indicates low short word field

Figure 5-11. 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 5-22. Table 5-1 shows the flags in XSTAT or YSTAT that indicate multiplier status for the most recent multiplier operation.

Table 5-1. Multiplier Status Flags

Flag	Definitions	Updated By...
MZ	Multiplier fixed-point zero and floating-point underflow or zero	All fixed- and floating-point multiplier ops, except multiply-accumulate
MN	Multiplier result is negative	All fixed- and floating-point multiplier ops, except multiply-accumulate
MV	Multiplier overflow ¹	All fixed- and floating-point multiplier ops, except multiply-accumulate
MU	Multiplier underflow ²	All floating-point multiplier ops; cleared by fixed-point ops, unchanged by multiply-accumulate
MI	Multiplier floating-point invalid operation ³	All floating-point multiplier ops; cleared by fixed-point ops, unchanged by multiply-accumulate

- 1 If the OEN bit in the X/YSTAT register is set, a floating-point operation that sets the MV flag generates a software exception.
- 2 If the UEN bit in the X/YSTAT register is set, a floating-point operation that sets the MU flag generates a software exception.
- 3 If the IVEN bit in the X/YSTAT register is set, a floating-point operation that sets the MI flag generates a software exception.

Multiplier operations also update sticky status flags in the compute block’s arithmetic status (XSTAT and YSTAT) register. Table 5-2 shows the flags in XSTAT or YSTAT that indicate multiplier sticky status for the most recent multiplier operation. Once set, a sticky flag remains high until explicitly cleared.

Flag update occurs at the end of each operation and is available with one cycle of latency; it becomes available on the second instruction line after status is generated. A program cannot write the arithmetic status register explicitly in the same cycle that status is being generated.


Multiplier Operations

Table 5-2. Multiplier Sticky Status Flags

Flag	Definition	Updated By...
MUS	Multiplier underflow, sticky ¹	All floating-point multiply ops
MVS	Multiplier floating-point overflow, sticky ²	All floating-point multiply ops
MOS	Multiplier fixed-point overflow, sticky	All fixed-point multiply ops
MIS	Multiplier floating-point invalid operation, sticky ³	All floating-point multiply ops

- 1 If the UEN bit in the X/YSTAT register is set, a floating-point operation that sets the MUS flag generates a software exception.
- 2 If the OEN bit in the X/YSTAT register is set, a floating-point operation that sets the MVS or MOS flag generates a software exception.
- 3 If the IVEN bit in the X/YSTAT register is set, a floating-point operation that sets the MIS flag generates a software exception.

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

 When the (NF) option is used, the compute unit does not update the flags with status from the operation. Programs can use the (NF) option with all multiplier instruction except for transfer MR4 register instructions.

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, instr. ;;
```

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

Table 5-3 lists the multiplier conditions. For more information on conditional instructions, see “Conditional Execution” on page 8-14.

Table 5-3. Multiplier Conditions

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

 Because status for a multiplier operation is available with one cycle of latency (becomes available on second instruction line after status is generated), using a multiplier condition in the cycle after it is generated causes a one cycle stall.

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 flag bits, X/YSCF0 (conditions are XSF0, YSF0, XYSF0, or SF0) and X/YSCF1 (conditions are XSF1, YSF1, XYSF1, or SF1). The following example shows how to load a compute block condition value into a static flag register. For more information on static flags, see “Conditional Execution” on page 8-14.

```
XSF0 = XMEQ ;; /* Load X-compute block MEQ flag into XSCF0 bit in
static flags (SFREG) register */
IF XSF0, DO XR5 = R4 * R3 ;; /* the SF0 condition tests the XSCF0
static flag bit */
```

Multiplier Examples

[Listing 5-1](#) 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 5-1. Multiplier Instruction Examples

```
XYR4 = R6 * R8 ;;
/* This instruction is a 32-bit fractional multiply in both compute blocks that produces a 32-bit rounded result. */

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

XR11:10 = R9:8 * R7:6 ;;
/* 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 (respectively). */

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_H. */

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_L. */
```



```
XR7 = MR3:2, MR3:2 += R1 * R0 ;;
/* This instruction executes a multiply-accumulate and transfers
the previous 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. */
```

Multiplier Instruction Summary

```
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 5-2 on page 5-27](#) “32-Bit Fixed-Point Multiplication Instructions”
- [Listing 5-3 on page 5-27](#) “16-Bit Fixed-Point Quad Multiplication Instructions”
- [Listing 5-4 on page 5-28](#) “16-Bit Fixed-Point Complex Multiplication Instructions”
- [Listing 5-5 on page 5-28](#) “32- and 40-Bit Floating-Point Multiplication Instructions”
- [Listing 5-6 on page 5-28](#) “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 2-6. 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*, *Rmq*, *Rnd*), or quad (*Rsq*) register names.

The MR3:0 registers are four 32-bit accumulation registers. They overflow into MR4, which stores two 16-bit overflows for 32-bit multiples, four 8-bit overflows for quad 16-bit multiples, or eight 4-bit overflows for quad short 16-bit multiples.

 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 1-23 and “[Instruction Parallelism Rules](#)” on page 1-27.

Listing 5-2. 32-Bit Fixed-Point Multiplication Instructions

```
{X|Y|XY}Rs = Rm * Rn {{{U|nU}{I|T}{S}{NF}}};1
{X|Y|XY}Rsd = Rm * Rn {{{U|nU}{I}{NF}}};
{X|Y|XY}MRa += Rm * Rn {{{U}{I}{C|CR}{NF}}};2
{X|Y|XY}MRa -= Rm * Rn {{{I}{C|CR}{NF}}};
{X|Y|XY}Rs = MRa, MRa += Rm * Rn {{{U}{I}{C|CR}{NF}}}; (dual op.)
{X|Y|XY}Rsd = MRa, MRa += Rm * Rn {{{U}{I}{C}{NF}}}; (dual op.)
/* where MRa is either MR1:0 or MR3:2 */
```

Listing 5-3. 16-Bit Fixed-Point Quad Multiplication Instructions

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

¹ Options include: (): fractional, signed, and 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

Multiplier Instruction Summary

Listing 5-4. 32-Bit Fixed-Point Complex Multiplication Instructions

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

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

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

Listing 5-6. Multiplier Register Load Instructions

```
{X|Y|XY}{S|L}MRa = Rmd {{{SE|ZE}{NF}}};  
{X|Y|XY}MR4 = Rm {{NF}};  
{X|Y|XY}{S}Rsd = MRa {{{U}{S}{NF}}};  
{X|Y|XY}Rsq = MR3:0 {{{U}{S}{NF}}};  
{X|Y|XY}Rs = MR4;  
/* where MRa is either MR1:0 or MR3:2 */  
  
{X|Y|XY}Rsd = SMRb {{{U}{NF}}}; /* extract 2 short words */  
{X|Y|XY}LRsd = MRb {{{U}{NF}}}; /* extract 1 normal word */  
/* where MRb is either MR0, MR1, MR2, or MR3 */  
  
{X|Y|XY}Rsq = SMRa {{{U}{NF}}}; /* extract 4 short words */  
{X|Y|XY}LRsq = MRa {{{U}{NF}}}; /* extract 2 normal words */  
{X|Y|XY}QRsq = LMRa {{{U}{NF}}}; /* extract 1 long word from MRa */  
*/  
/* where MRa is either MR1:0 or MR3:2 */  
  
{X|Y|XY}Rs = COMPACT MRa {{{U}{I}{S}{NF}}};  
{X|Y|XY}SRsd = COMPACT MR3:0 {{{U}{I}{S}{NF}}};  
/* where MRa is either MR1:0 or MR3:2 */
```

6 SHIFTER

The ADSP-TS201 TigerSHARC processor core contains two computation units known as compute blocks. Each compute block contains a register file and four independent computation units—an ALU, a CLU, a multiplier, and a shifter. The shifter is highlighted in [Figure 6-1](#). The shifter takes its inputs from the register file, and returns its outputs to the register file.

This chapter provides:

- [“Shifter Operations” on page 6-4](#)
- [“Shifter Examples” on page 6-20](#)
- [“Shifter Instruction Summary” on page 6-22](#)

The shifter performs *bit wise operations* (arithmetic and logical shifts) and performs *bit field operations* (field extraction and deposition) for the processor. 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

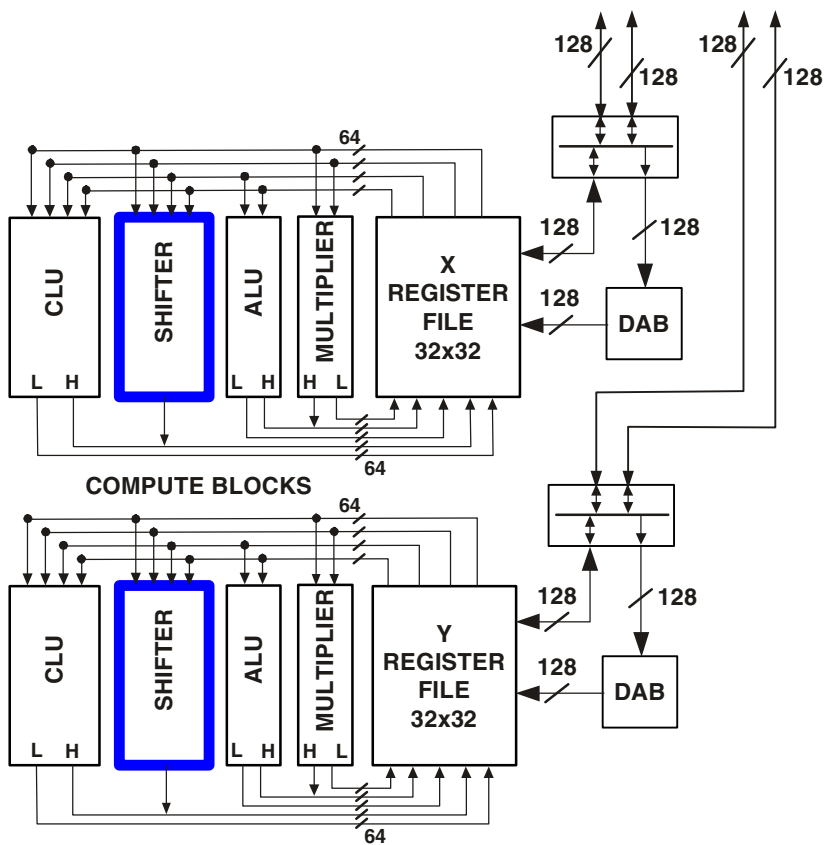


Figure 6-1. Shifters in Compute Block X and Y¹

- 1 On the outputs of the compute units, H indicates high half of quad result (bits 128–64), and L indicates low half of quad result (bits 63–0). For a result smaller or equal to 64 bits, compute units use the (L) lower result output.

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 6-1](#), the shifter has three inputs and one 64-bit output (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`, `GETBITS` and `PUTBITS` 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 1-27](#).

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 2-6](#).

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 1-23](#). For a list of shifter instructions and their syntax, see [“Shifter Instruction Summary” on page 6-22](#).

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
- 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-21000 family compatible fixed-point and floating-point conversion operations (such as exponent extract, number of leading ones or zeros)

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 result, where Rn is 8 bits for long word result, 7 bits for normal word result, 6 bits for short word result, and 6 bits for byte word result. 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:

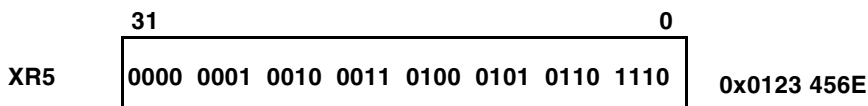
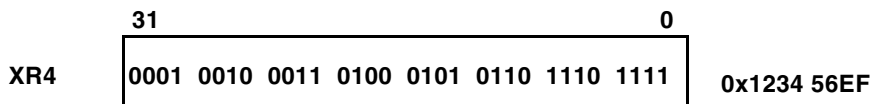
- [“Logical Shift Operation” on page 6-6](#)
- [“Arithmetic Shift Operation” on page 6-7](#)
- [“Bit Manipulation Operations” on page 6-8](#)
- [“Bit Field Manipulation Operations” on page 6-9](#)
- [“Bit Field Analysis Operations” on page 6-12](#)
- [“Bit Stream Manipulation Operations” on page 6-12](#)

Shifter Operations

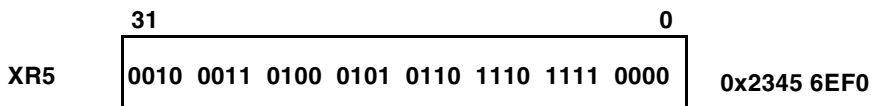
Logical Shift Operation

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 6-2 shows how the bits in register XR5 are placed for shift values of 4 (first example) and -4 (second example).

```
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.



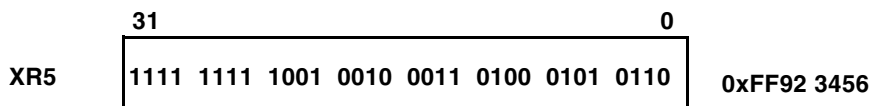
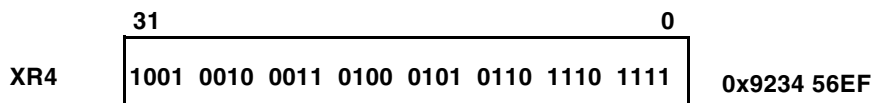
For a positive LSHIFT value, the shift is to the LEFT and ZERO-FILLED. Here, the LSHIFT value is 4, so bits 3-0 are zero-filled.

Figure 6-2. LSHIFT Instruction Example

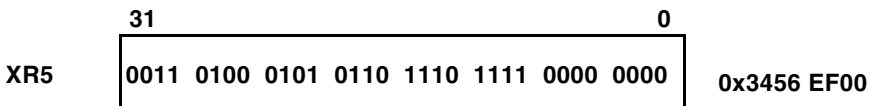
Arithmetic Shift Operation

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 6-3](#) shows how the bits in register XR5 are placed for shift values of 8 (first example) and -8 (second example).

```
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 6-3. ASHIFT Instruction Example

Shifter Operations

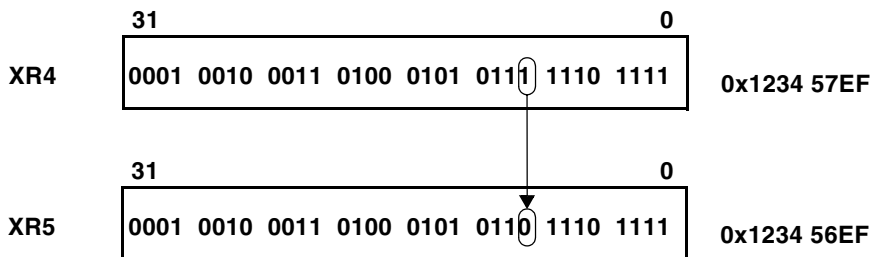
Bit Manipulation Operations

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 6-4](#) 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.**

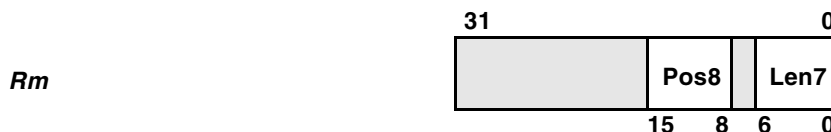
Figure 6-4. BCLR Instruction Example

Bit Field Manipulation Operations

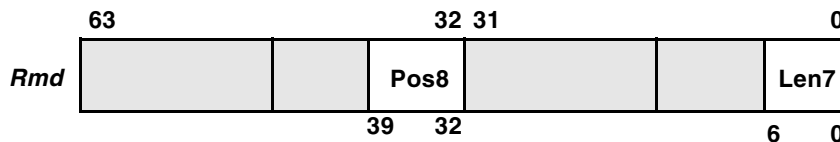
The shifter supports bit field manipulation operations including:

- 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: $Len7$ and $Pos8$. 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 6-5.



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.

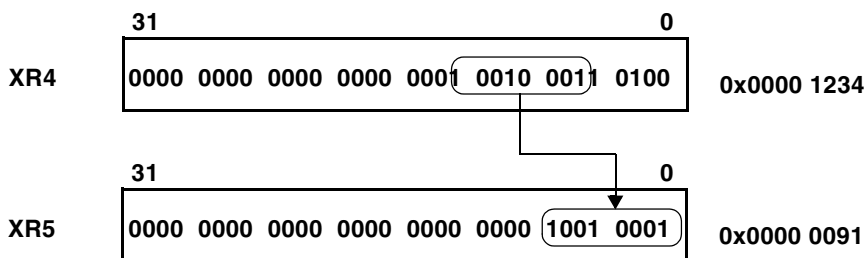
Figure 6-5. FEXT and FDEP Instructions $Pos8$ and $Len7$ Fields

Shifter Operations

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 (MSB) of the extracted field is extended. [Figure 6-6](#) 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 */
```

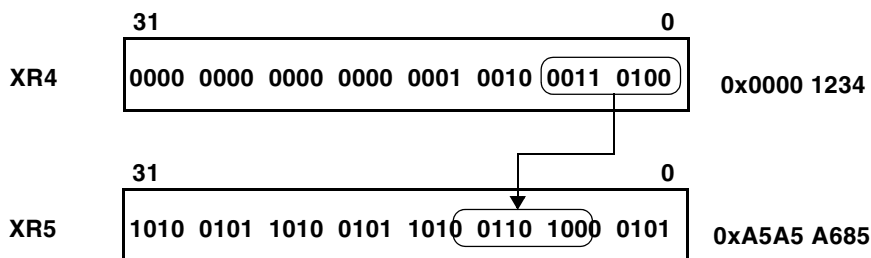


For a `FEXT` field extraction, the unused bits in the destination are CLEARED unless the `SE` option is used. Here, bits 31–8 are cleared.

Figure 6-6. `FEXT` Instruction Example

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 MSB of the extracted field is extended. If the FDEP instruction included the zero-filled (ZF) option, the most significant unused bits of result register are zero-filled. Figure 6-7 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 6-7. FDEP Instruction Example

Shifter Operations

The following instruction is an example of mask (MASK) instruction. The shifter takes the bits from XR4 corresponding to the mask XR3 and parses 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 Analysis Operations

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

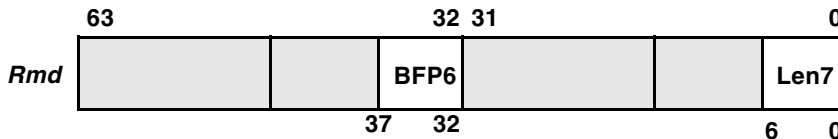
- BKFPT – determines scaling factor used in 16-bit block floating-point
- EXP – extracts the exponent
- LDx – extracts leading zeros (0) or ones (1)

Bit Stream Manipulation Operations

The bit stream manipulation operations, in conjunction with the ALU BF0INC 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
- BF0TMP – temporarily stores or returns overflow from GETBITS and PUTBITS instructions

For bit stream extract (GETBITS) and deposit (PUTBITS) operations, the *Rnd* operand contains the control information in two fields: BFP6 and Len7. These fields in the dual register, *Rnd*, appear in Figure 6-8.



The dual register operand provides the BFP6 and Len7 fields (bits 37–32 and 5–0). Note that the BFP must be incremented using the ALU's BFOINC instruction.

Figure 6-8. GETBITS and PUTBITS Instructions BFP6 and Len7 Fields

The GETBITS instruction extracts the number of bits indicated by Len7 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 (Len7) values specified in the XR7:6 dual register.

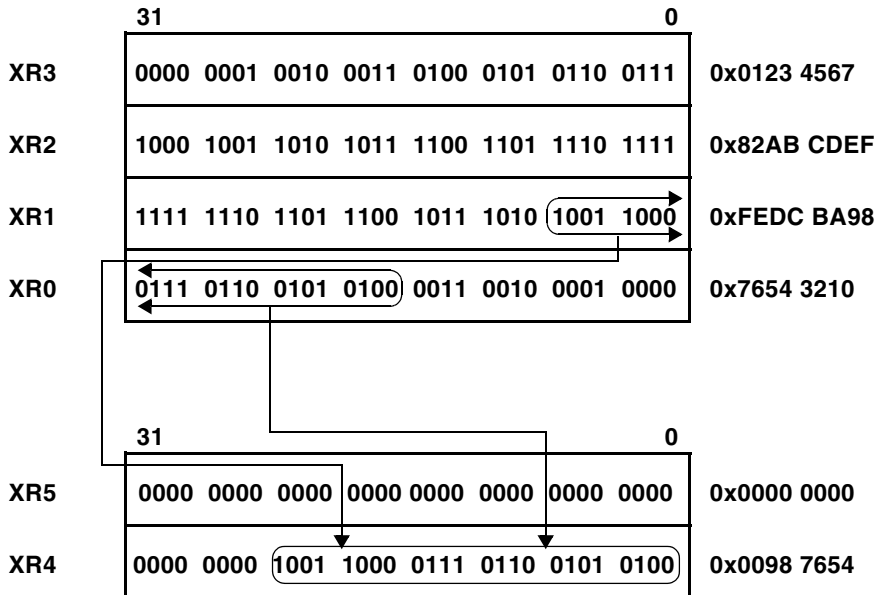
- i Use the ALU's BFOINC instruction to increment the bit FIFO pointer. Normally, an update of bit FIFO pointer is necessary after executing GETBITS. The ALU instruction BFOINC adds BFP6 and Len7 fields, divides them by 64, and returns the remainder (modulo-64 result) to the BFP6 field. If for example, BFP6 is 0x30 and Len7 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.

Shifter Operations

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 MSB of the extracted field is extended.

Figure 6-9 shows how the bits in register $XR5:4$ are affected for field position $BFP6=16$ and field length $Len7=24$.

```
XR5:4 = GETBITS R3:0 BY R7:6 ;;
/* BFP6=16, Len7=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 6-9. 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 `BFP6` field specifies the starting bit where the insertion begins in *Rsd* register, but the `Len7` field is ignored. Update of `BFP6` should be performed by the ALU with the instruction `BFOINC`.

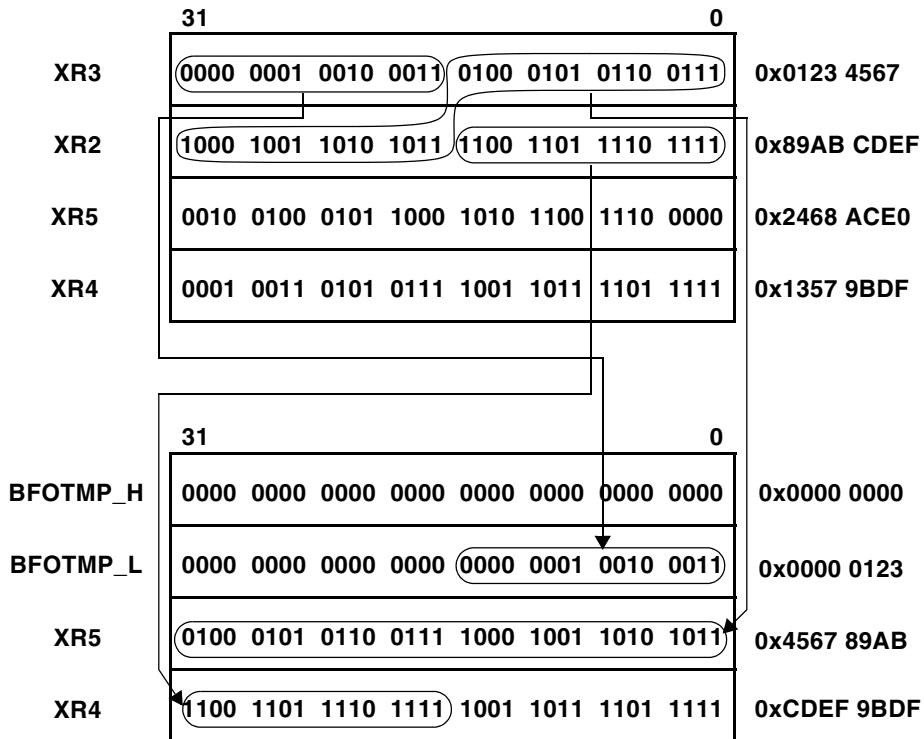
The following instruction is an example of bit stream placement (`PUTBITS`). The shifter puts the content of the registers `XR3:2` into the bit FIFO composed by `XR5:4` and `BFOTMP` beginning with bit 16 of `XR4` (specified into `BFP6` field of `XR7`). [Figure 6-10](#) shows how the bits in register `XR5:4` are affected for field position `BFP6=16`.

```

XR3 = 0x01234567 ;;
XR2 = 0x89abcdef ;;
XR5 = 0x2468ACE0 ;
XR4 = 0x13579BDF ;
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      = 0x456789AB
   XR4      = 0xCDEF9BDF

```

Shifter Operations



For a PUTBITS field deposit, the unused bits in the destination retain their previous data. Here, bits 15–0 of XR4 and bits 63–16 of BFOTMP retain their previous data (BFOTMP previous data assumed to be 0x0000 0000).

Figure 6-10. PUTBITS Instruction Example

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 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 6-22](#). 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
- (NF) no flag update

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 6-6 on
page 6-10 */
```

```
XR5 = FDEP R4 By R3 (ZF) ;;
/* The ZF option in this instruction clears bits 31–13 and 4–0 in Figure 6-7
on page 6-11 */
```

Shifter Operations

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 MSB 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 MSB of the placed bit field in the output register.

No Flag Update Option

Almost all compute operations generate status, which affects the compute unit's status register. Often, it is useful in some applications to retain status from an operation. There are a number of techniques for storing status (for example, storing the status register contents or loading flag values into the static flags (SFREG) register). The No Flag Update (NF) option provides a different method for working with status. Instead of storing an operation's status for future usage, programs can prevent status from being overwritten by using the (NF) option on following operations to prevent status generation. There are some restrictions on how the (NF) option can be used with conditional instructions. For more information, see [“Conditional Execution” on page 8-14](#).

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 6-19](#).


Table 6-1 shows the flags in XSTAT or YSTAT that indicate shifter status for the most recent shifter operation.

Table 6-1. 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

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

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

 When the (NF) option is used, the compute unit does not update the flags with status from the operation. Programs can use the (NF) option with all shifter instruction except for BITEST, BKFPT, and load X/YSTAT, TRx, THRx, and CMCTL registers.

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.

Shifter Examples

Table 6-2 lists the shifter conditions. For more information on conditional instructions, see [“Conditional Execution” on page 8-14](#).

Table 6-2. Shifter Conditions

Condition	Description	Flags Set
SEQ	Shifter equal to zero	SZ=1
SLT	Shifter less than zero	SN=1 and SZ=0
NSEQ	Not shifter equal to zero	SZ=0
NSLT	Not shifter less than zero	SN=0 or SZ=1

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 flag bits, X/YSCF0 (conditions are XSF0, YSF0, XYSF0, or SF0) and X/YSCF1 (conditions are XSF1, YSF1, XYSF1, or SF1). The following example shows how to load a compute block condition value into a static flag register.

```
XSF0 = XSEQ ;; /* Load X-compute block SEQ flag into XSCF0 bit in
static flags (SFREG) register */
IF XSF0, D0 XR5 = LSHIFT R4 BY R3 ;; /* the XSF0 condition tests
the XSCF0 static flag bit */
```

For more information on static flags, see [“Conditional Execution” on page 8-14](#).

Shifter Examples

Listing 6-1 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.

Listing 6-1. 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 each word 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 6-2](#) 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 2-6](#). Briefly, these conventions are:

- { } – the curly braces enclose options; these braces are not part of the instruction syntax.
- | – the vertical bar separates choices; this bar is 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;
```

- 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 1-23 and [“Instruction Parallelism Rules”](#) on page 1-27.

Shifter Instruction Summary

Listing 6-2. Shifter Instructions

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

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

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

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

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

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

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

{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> {(NF)} ;
{X|Y|XY}Rsd = BCLR|BSET|BTGL Rmd BY Rn|<Imm6> {(NF)} ;

{X|Y|XY}Rs = LDO|LD1 Rm|Rmd {(NF)} ;

{X|Y|XY}Rs = EXP Rm|Rmd {(NF)} ;
```

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 6-5 on page 6-9](#).

```

{X|Y}STAT = Rm ;
{X|Y}STAT_L = Rm ;
{X|Y}Rs = {X|Y}STAT ;

{X|Y|XY}BKFPTRmd, Rnd ;

{X|Y|XY}Rsd = BFOTMP {(NF)} ;
{X|Y|XY}BFOTMP = Rmd {(NF)} ;

{X|Y|XY}TRs = Rm ;
{X|Y|XY}TRsd = Rmd ;
{X|Y|XY}TRsq = Rmq ;
{X|Y|XY}THR_s = Rm ;
{X|Y|XY}THRsd = Rmd {(I)} ;
{X|Y|XY}THRsq = Rmq ;
{X|Y|XY}CMCTL = Rm ;

```

/ For TR, THR, and CMCTL register transfer syntax, see “CLU Quick Reference” on page A-6. */*

Shifter Instruction Summary

7 IALU

The ADSP-TS201 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 7-1](#).

The ADSP-TS201 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 IALU's *integer 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 two memory accesses (read or write). The IALU's *load, store, and transfer data operations* include:

- Direct and indirect memory addressing
- Circular buffer addressing
- Bit reverse addressing

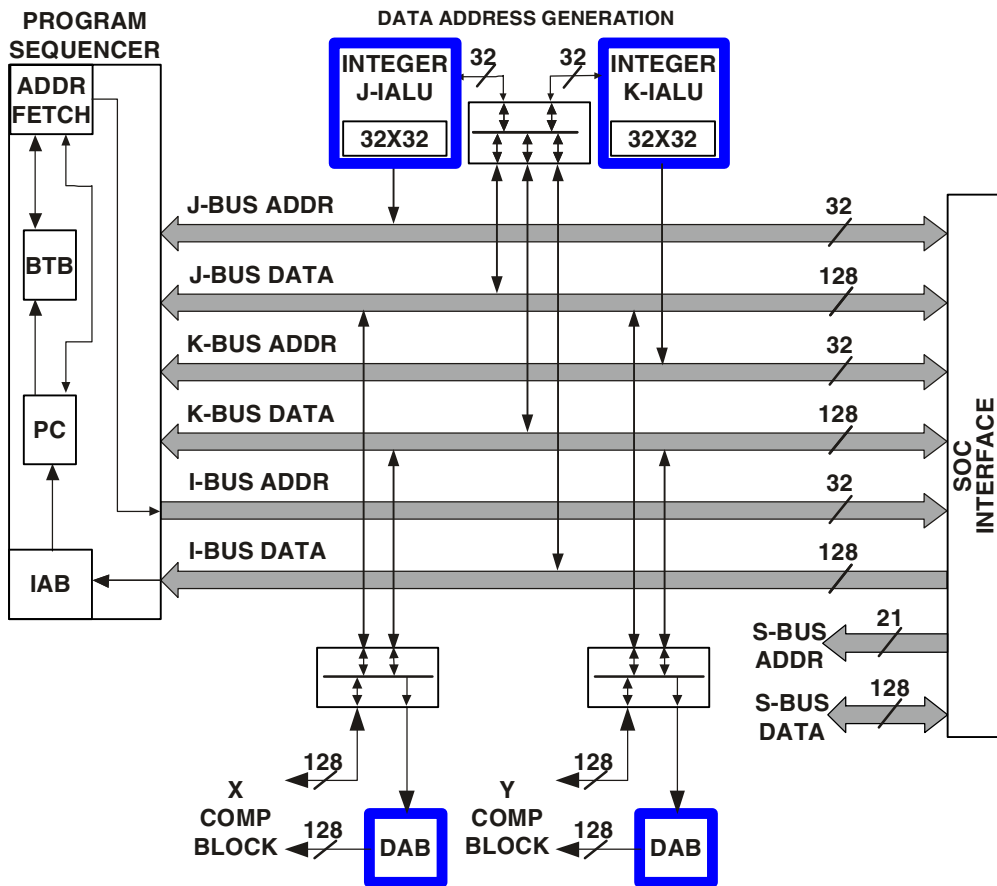


Figure 7-1. IALUs, DABs, and Data Buses

- 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 processor 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, four or five cycles of latency can occur. [For more information, see “Dependency and Resource Effects on Pipeline” on page 8-64.](#)

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 J30–J0, and the K-IALU register file registers are K30–K0. Except for J31 and K31, these registers are general-purpose and contain integer data only.
- The J31 and K31 registers are 32-bit status registers and can also be referred to as JSTAT and KSTAT.

The JSTAT (J31) and KSTAT (K31) registers appears in Figure 7-2. These registers have special operations:

- When used as an operand in an IALU arithmetic, logical, or function operation, these registers are referred to as J31 and K31 registers, and the register's contents are treated as zero. If J31 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 (as a source or destination, not as a pointer), these registers are referred to as JSTAT and KSTAT, and the operation does not clear the register contents.

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.

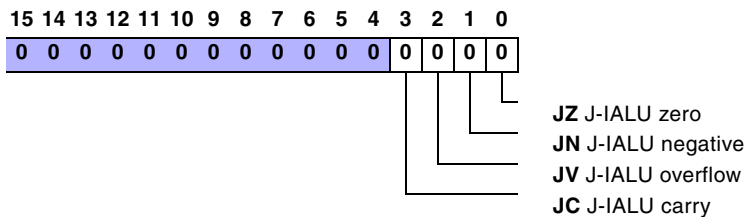


Figure 7-2. JSTAT/KSTAT (Lower Half) Register Bit Descriptions¹

¹ Bits 31–16 (not shown) are reserved (=0).

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 modifier may not be larger than the length of the circular buffer.
- 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 in the IALU controlling the circular buffer may serve as the modifier.

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 Integer \(Arithmetic and Logic\) Operations”](#) on page 7-6
- [“IALU Load, Store, and Transfer Operations”](#) on page 7-14

IALU Integer (Arithmetic and Logic) Operations

The IALU performs arithmetic and logical operations on fixed-point (integer) data. The processor 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 information on the register files and register naming syntax for selecting data type and width, see [“Register File Registers”](#) on page 2-6. 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 $((-2 + 0) / 2)$, the operation would set the `JN` flag (J-IALU, IALU result negative) because the operation resulted in a negative value. For more information on IALU status, see [“IALU Execution Status” on page 7-10](#).

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 7-47](#).

The IALU instruction options include:

- () signed integer operation
- (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
- (NF) no flag update

IALU Operations

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. */
```

IALU Data Types

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](#).

Signed/Unsigned Option

Fixed-point 32-bit data in the IALU is two’s complement format. For the COMP instruction only, unsigned format may be used. 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|<Imm8>|<Imm32> {{(CJMP|CB|BR)}} ;  
Ks = Km +|- Kn|<Imm8>|<Imm32> {{(CJMP|CB|BR)}} ;
```



For information on the conventions used in this instruction summary, see [“IALU Instruction Summary” on page 7-47](#).

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 J_s or K_s . (J_m or K_m is not modified.)


 For information on circular buffer operations, see [“Circular Buffer Addressing” on page 7-33](#).

Bit Reverse Option


The IALU add and subtract instructions support the bit reverse carry (BR) option. The instructions take the form:

$$J_s = J_m + | - J_n | \langle Imm8 \rangle | \langle Imm32 \rangle \{ (\{ CJMP | CB | BR \}) \} ;$$

$$K_s = K_m + | - K_n | \langle Imm8 \rangle | \langle Imm32 \rangle \{ (\{ CJMP | CB | BR \}) \} ;$$

 For information on the conventions used in this instruction summary, see [“IALU Instruction Summary” on page 7-47](#).

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 7-38](#).

IALU Operations

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 7-47](#).

The computed jump (CJMP) option directs the IALU to place the result in the program sequencer’s CJMP registers as well as the result (Js or Ks) register.

No Flag Update Option

Almost all IALU compute operations generate status, which affects the IALU’s status register. Often, it is useful in some applications to retain status from an operation. There are a number of techniques for storing status (for example, storing the status register contents or loading flag values into the static flags (SFREG) register). The No Flag Update (NF) option provides a different method for working with status. Instead of storing an operation’s status for future usage, programs can prevent status from being over written by using the (NF) option on following operations to prevent status generation. There are some restrictions on how the (NF) option can be used with conditional instructions. For more information, see [“Conditional Execution” on page 8-14](#).

IALU Execution Status

IALU operations update status flags in the IALUs’ status (JSTAT and KSTAT) registers. (See [Figure 7-2 on page 7-4](#).) 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 7-13](#).

Table 7-1 shows the flags in JSTAT or KSTAT that indicate IALU status for the most recent IALU operation.

Table 7-1. 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, cleared by logical op
JC	J-IALU carry	Set by add/subtract ops, cleared by all other 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, cleared by logical op
KC	K-IALU carry	Set by add/subtract ops, cleared by all other ops

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



When the (NF) option is used, the IALU does not update the flags with status from the operation. Programs can use the (NF) option with all IALU arithmetic, logical, and function instruction except for COMP.

JZ/KZ–IALU Zero

The JZ or KZ flag is set whenever the result of a J-IALU or K-IALU operation is zero. The flag is cleared if the result is non-zero.

IALU Operations

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 (MSB) 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.

If in the following example J5 and J6 are $0 \times 70 \dots 0$ (large positive numbers), the result of the add instruction produces a result that is larger than the maximum at the given format.

$J_{10} = J_5 + J_6 ; ;$

JC/KC–IALU Carry

The JC or KC flag is used as a 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 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 7-2](#) lists the IALU conditions. For more information on conditional instructions, see [“Conditional Execution” on page 8-14](#).

Table 7-2. IALU Conditions

Condition	Description	Flags Set
JEQ	J-IALU equal to zero	JZ=1
JLT	J-IALU less than zero	JN=1 and JZ=0
JLE	J-IALU less than or equal to zero	JN=1 or JZ=1
NJEQ	NOT(J-IALU equal to zero)	JZ=0
NJLT	NOT(J-IALU less than zero)	JN=0 or JZ=1
NJLE	NOT(J-IALU less than or equal to zero)	JN=0 and JZ=0
KEQ	K-IALU equal to zero	KZ=1
KLT	K-IALU less than zero	KN=1 and KZ=0
KLE	K-IALU less than or equal to zero	KN=1 or KZ=1
NKEQ	NOT(K-IALU equal to zero)	KZ=0
NKLT	NOT(K-IALU less than zero)	KN=0 or KZ=1
NKLE	NOT(K-IALU less than or equal to zero)	KN=0 and KZ=0

IALU Operations

IALU Static Flags

In the program sequencer, the static flag (SFREG) can store status flag values for later use in conditional instructions. With SFREG, the IALU has two global static flag bits, ISF0 (condition is ISF0) and ISF1 (condition is ISF1). The following example shows how to load an IALU condition value into a static flag register.

```
ISF0 = JEQ ;; /* Load J-IALU JEQ flag into ISF0 bit in static
flags (SFREG) register */
IF ISF0, D0 J5 = J4 + J3 ;; /* the ISF0 condition tests the ISF0
static flag bit */
```

In addition to IALU conditions (JEQ, JLT, JLE, KEQ, KLT, KLE, or their negated forms), the IALU's global static flags can also hold compute block conditions (ALU, multiplier, and shifter conditions) and sequencer conditions (LCx_E, ISFx, TRUE, BM, and FLAGx_IN). For more information on static flags, see [“Conditional Execution” on page 8-14](#).

IALU Load, Store, 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. Indirect addressing uses 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.

Direct and Indirect Addressing

Direct addressing uses an index that is set to zero (register J31) 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 any IALU register file register as an index and uses either a register or an immediate value for the modifier. As shown in [Figure 7-3](#), there are two types of indirect addressing.

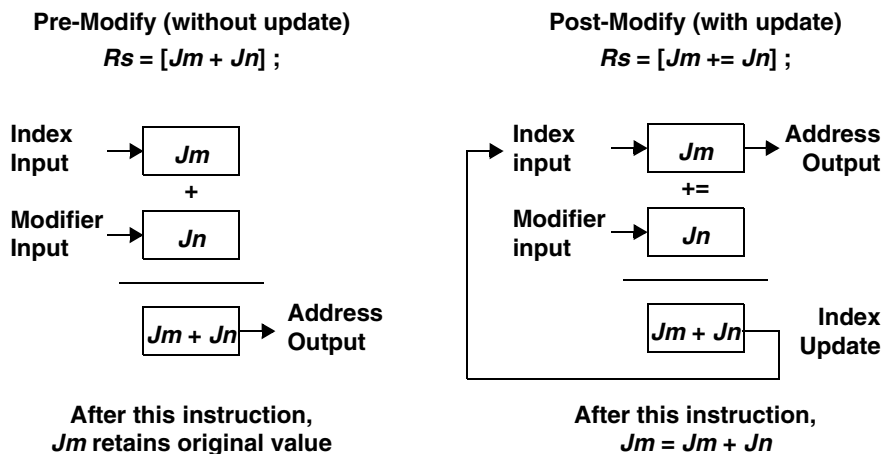


Figure 7-3. Pre- and Post-Modify Indirect Addressing¹

¹ For indirect addressing, the modifier may be Jn or an immediate value.


IALU Operations

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 required for circular buffer, DAB, and bit reversed addressing. For more information, see [“Circular Buffer Addressing” on page 7-33](#), [“Data Alignment Buffer \(DAB\) Accesses” on page 7-26](#), and [“Bit Reverse Addressing” on page 7-38](#).


Normal, Merged, and Broadcast Memory Accesses

The IALU uses direct or indirect addressing to perform read and write memory accesses, which load or store data in registers. There are three types of memory accesses—normal read/write accesses, merged read/write accesses, and broadcast load 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 one, two, or four 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 7-5](#), [Figure 7-8](#), [Figure 7-11](#), [Figure 7-13](#), [Figure 7-15](#), and [Figure 7-17](#).) Examples of normal accesses (destination \Leftrightarrow source) are:
 - Single register (R_s) \Leftrightarrow no operator
 - Dual register (R_{sd}) \Leftrightarrow L (long) operator
 - Quad register (R_{sq}) \Leftrightarrow Q (quad) operator
- Broadcast load access – broadcast load access reads data from memory and loads data in compute block data registers. Broadcast accesses read one, two, or four 32-bit words and load this data to destination registers in both compute blocks. Broadcast accesses occur when the source register size matches the IALU access operator and the register name uses XY (or no prefix) to indicate both compute blocks. (See [Figure 7-6](#), [Figure 7-9](#), and [Figure 7-12](#).) Examples of broadcast accesses (destination \Leftrightarrow 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

IALU Operations

- Merged read/write accesses – merged read/write memory accesses load or store data in compute block data registers. Merged data transfers are data transfers of long or quad words. The transfer is between the memory and both compute block register files, where the data is divided between X and Y. If a long word is read memory, one word is loaded to compute X and one is loaded to compute Y. If a quad-word is read from memory, two words are loaded to X and two words are loaded to Y. The merged access can work both directions (load or store) and the order between X and Y can be changed by using the prefix XY or YX. In case of XY the high part of the transaction (high word in long transaction or two high words in a quad transaction) is loaded to X registers, and the low part to Y registers. If the use is YX, the high part is loaded to Y and the low part is loaded to X. (See [Figure 7-7](#), [Figure 7-10](#), [Figure 7-14](#), and [Figure 7-16](#).) Example merged accesses (destination \Leftrightarrow source) are:
 - Single register (R_s) \Leftrightarrow L (long) operator
 - Dual register (R_{sd}) \Leftrightarrow Q (quad) operator

 [Figure 7-5](#) through [Figure 7-16](#) 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 7-47.

Looking at [Figure 7-4](#) (memory contents) and [Figure 7-5](#) through [Figure 7-16](#) (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 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 (.LDF) 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 7-26.

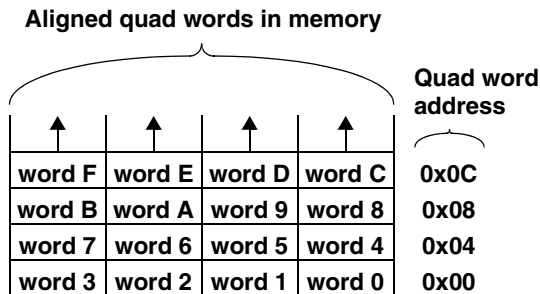
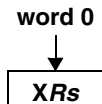


Figure 7-4. Memory Contents for Normal, Broadcast, and Merged Memory Access Examples

$XRs = [Jm += Jn] ; ; /* Jm = \text{address of word } 0.*/$

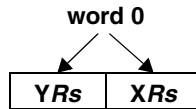


See [Figure 7-4](#) for memory contents.

Figure 7-5. Single Register Normal Read Accesses

IALU Operations

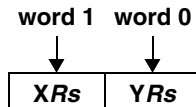
$XYRs = [Jm += Jn] ;; /* Jm = \text{address of word 0.*/$



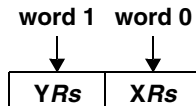
See [Figure 7-4](#) for memory contents.

Figure 7-6. Single Register Broadcast Load Accesses

$XYRs = L [Jm += Jn] ;; /* Jm = \text{address of word 0.*/$



$YXRs = L [Jm += Jn] ;; /* Jm = \text{address of word 0.*/$

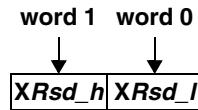


See [Figure 7-4](#) for memory contents.

Note that XY and YX syntax produce *different* results.

Figure 7-7. Single Register Merged Read Accesses

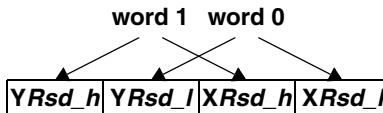
$XRsd = L [Jm += Jn] ;; /* Jm = address of word 0.*/$



See [Figure 7-4](#) for memory contents.

Figure 7-8. Dual Register Normal Read Accesses

$XYRsd = L [Jm += Jn] ;; /* Jm = address of word 0.*/$

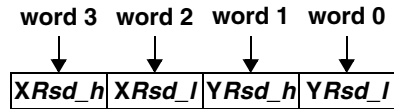


See [Figure 7-4](#) for memory contents.

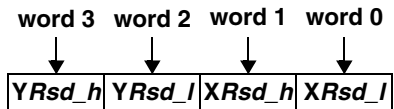
Figure 7-9. Dual Register Broadcast Load Accesses

IALU Operations

$XYRsd = Q [Jm += Jn] ;; /* Jm = address of word 0.*/$



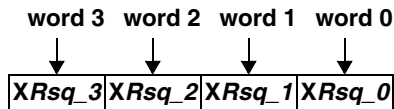
$YXRsd = Q [Jm += Jn] ;; /* Jm = address of word 0.*/$



See [Figure 7-4](#) for memory contents.
Note that XY and YX syntax produce *different* results.

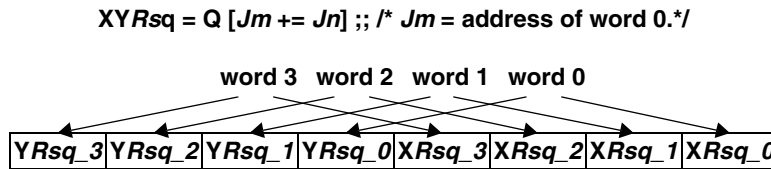
Figure 7-10. Dual Register Merged Read Accesses

$XRsq = Q [Jm += Jn] ;; /* Jm = address of word 0.*/$



See [Figure 7-4](#) for memory contents.

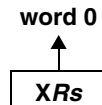
Figure 7-11. Quad Register Normal Read Accesses



See [Figure 7-4](#) for memory contents.

Figure 7-12. Quad Register Broadcast Load Accesses

$[Jm += Jn] = XRs ;; /* Jm = address of word 0.*/$

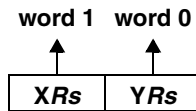


See [Figure 7-4](#) for memory contents.

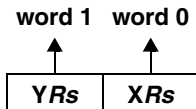
Figure 7-13. Single Register Normal Write Accesses

IALU Operations

L [Jm += Jn] = XYRs ;; /* Jm = address of word 0.*/



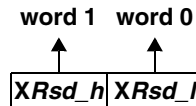
L [Jm += Jn] = YXR_s ;; /* Jm = address of word 0.*/



See [Figure 7-4](#) for memory contents.
Note that XY and YX syntax produce *different* results.

Figure 7-14. Single Register Merged Write Accesses

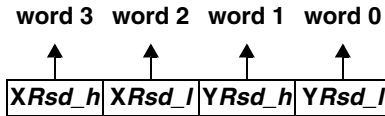
L [Jm += Jn] = XRsd ;; /* Jm = address of word 0.*/



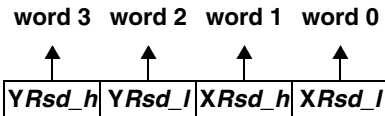
See [Figure 7-4](#) for memory contents.

Figure 7-15. Dual Register Normal Write Accesses

Q [Jm += Jn] = XYRsd ;; /* Jm = address of word 0.*/



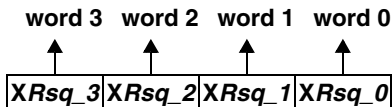
Q [Jm += Jn] = YXRsd ;; /* Jm = address of word 0.*/



See [Figure 7-4](#) for memory contents.
Note that XY and YX syntax produce *different* results.

Figure 7-16. Dual Register Merged Write Accesses

Q [Jm += Jn] = XRsq ;; /* Jm = address of word 0.*/




See [Figure 7-4](#) for memory contents.

Figure 7-17. 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 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 7-17.

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 7-18](#) 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 7-19](#).

[Figure 7-19](#) 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 addressing (*J3–J0*, *K3–K0*). For more information on circular buffer

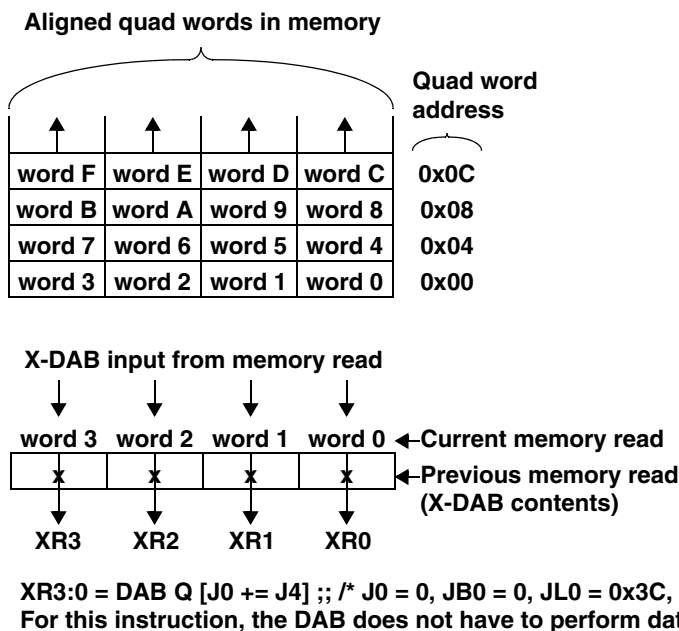
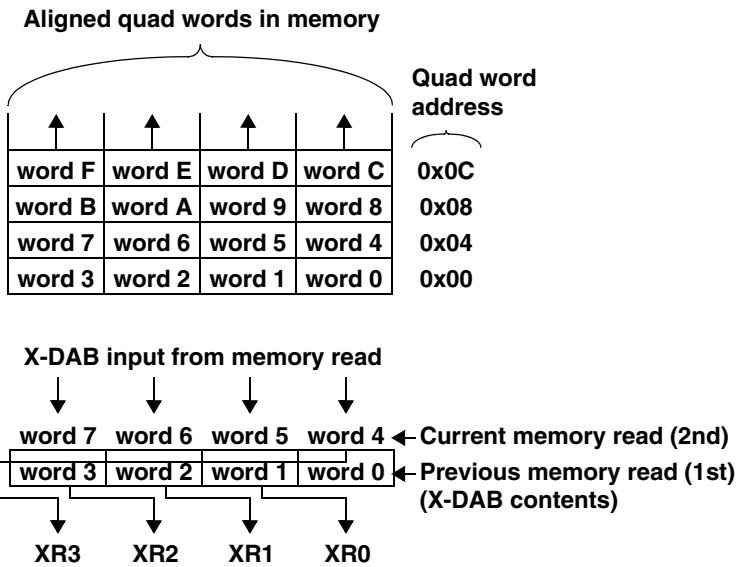


Figure 7-18. DAB Operation for Aligned Data

addressing, see [“Circular Buffer Addressing” on page 7-33](#). 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 Only post-modify addressing mode can be used with DAB operation. DAB operation works in the desired (described) way only if the address modifier in the address calculation expression is positive. If DAB accesses need to use linear addressing, set the circular buffer length (corresponding JL/KL register) to zero.

IALU Operations



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 7-19. DAB Operation for Non-Aligned Data

The DAB also provides access to non-aligned short word data in memory as shown in Figure 7-20. 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 (J_n or K_n) must be equal to eight.

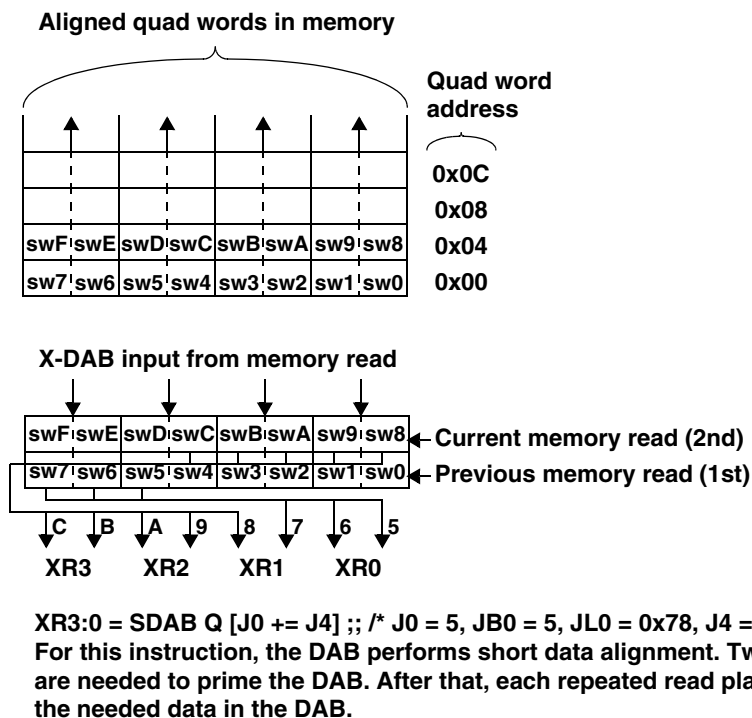


Figure 7-20. SDAB Operation for Non-Aligned Data

IALU Operations

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 7-21](#).

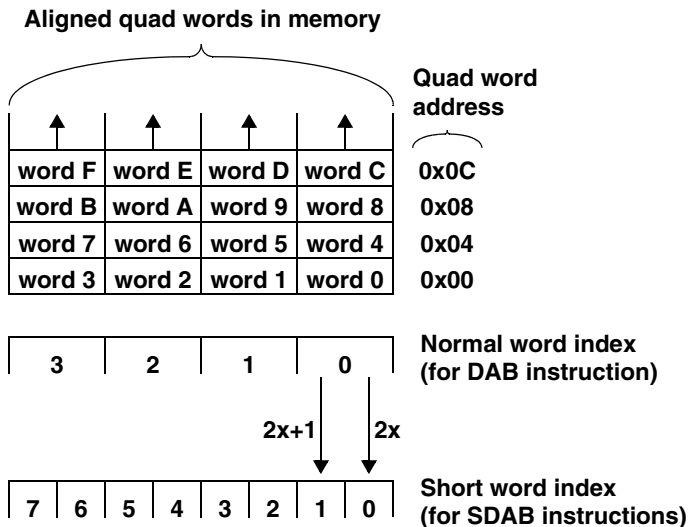


Figure 7-21. DAB Versus SDAB Index Values

Data Alignment Buffer (DAB) Accesses With Offset

The load of compute registers can be done with a special DAB shift. Unlike the regular DAB, this operation has a fixed shift of 1 for one compute and 0 for the other compute, on top of the regular DAB shift. The shift can be by one short (16-bit) word or one normal (32-bit) word. The instruction syntax is shown in [Listing 7-1](#). This special DAB shift makes it easier to use interleaved data, an important way to optimize some algorithms on this processor's SIMD compute blocks.

Listing 7-1. Syntax for DAB Accesses With Offset

```
{XY|YX}Rsq = XDAB Q [Jm += Jn|<Imm8>|<Imm32>] ;
/* shifts X by one 32-bit word more than address misalignment,
and shift Y by the misalignment */
```

```
{XY|YX}Rsq = XSDAB Q [Jm += Jn|<Imm8>|<Imm32>] ;
/* shifts X by one 16-bit more than address misalignment, and
shift Y by the misalignment */
```

```
{XY|YX}Rsq = YDAB Q [Jm += Jn|<Imm8>|<Imm32>] ;
/* shifts Y by one 32-bit word more than address misalignment,
and shift X by the misalignment */
```

```
{XY|YX}Rsq = YSDAB Q [Jm += Jn|<Imm8>|<Imm32>] ;
/* shifts Y by one 16-bit word more than address misalignment,
and shift X by the misalignment */
```

```
{XY|YX}Rsq = XDAB Q [Km += Kn|<Imm8>|<Imm32>] ;
/* shifts X by one 32-bit word more than address misalignment,
and shift Y by the misalignment */
```

```
{XY|YX}Rsq = XSDAB Q [Km += Kn|<Imm8>|<Imm32>] ;
/* shifts X by one 16-bit more than address misalignment, and
shift Y by the misalignment */
```

```
{XY|YX}Rsq = YDAB Q [Km += Kn|<Imm8>|<Imm32>] ;
/* shifts Y by one 32-bit word more than address misalignment,
and shift X by the misalignment */
```

IALU Operations

```
{XY|YX}Rsq = YSDAB Q [Km += Kn|<Imm8>|<Imm32>] ;
/* shifts Y by one 16-bit word more than address misalignment,
and shift X by the misalignment */

/* m must be 0,1,2, or 3 for bit reverse or circular buffers */
```

The type of access is selected with the access group bits within the data access instruction. For more information on the instruction decode for these instructions, see [Table C-18 on page C-36](#).

The alignment for DAB accesses with offset is shown in [Table 7-3](#).

Table 7-3. Data Alignment for DAB Accesses With Offset

Address ¹	Short DAB	Short DAB + misalignment	DAB	DAB + misalignment
000	Old[7:0]	New[0], Old[7:1]	Old[3:0]	New[0], Old[3:1]
001	New[0], Old[7:1]	New[1:0], Old[7:2]	New[0], Old[3:1]	New[1:0], Old[3:2]
010	New[1:0], Old[7:2]	New[2:0], Old[7:3]	New[1:0], Old[3:2]	New[2:0], Old[3]
011	New[2:0], Old[7:3]	New[3:0], Old[7:4]	New[2:0], Old[3]	New[3:0]
100	New[3:0], Old[7:4]	New[4:0], Old[7:5]		
101	New[4:0], Old[7:5]	New[5:0], Old[7:6]		
110	New[5:0], Old[7:6]	New[6:0], Old[7]		
111	New[6:0], Old[7]	New[7:0]		

¹ Address bits [2:0] ([1:0] for word access)

Circular Buffer Addressing

The IALUs support addressing *circular buffers*—a range of addresses containing data that IALU memory accesses continually in a modulo fashion.¹ The memory read or write access instruction uses the operator `CB` to select circular buffer addressing.

Circular buffer IALU load and store instructions differ from standard IALU load and store instructions in the method used to calculate a post-modified address. Standard IALU load and store instructions simply add the increment to the current address to generate the next address. Circular buffer IALU load and store instructions generate the next address by adding the increment to the current address in a modulo-like fashion.

The address calculation formula is exactly the same for circular buffer addressing and linear addressing, assuming the `LENGTH` value equals zero and assuming the `BASE` value equals the base address of the buffer. Each circular buffer calculation has associated with it four separate values: a `BASE` value, a `LENGTH` value, an `INDEX` value, and a `MODIFY` value.

- The `BASE` value is the base address of the circular buffer and is stored in the associated base register.
- The `LENGTH` value is the length of the circular buffer (number of 32-bit words) and is stored in the associated length register.
- The `INDEX` value is the current address that the circular buffer is indexing and is stored in the associated IALU register.
- The `MODIFY` value is the post-modify value that updates the `INDEX` value.

¹ Modulo fashion—to step through repeatedly, wrapping around to repeat stepping through the range of addresses in a circular pattern.

IALU Operations

The following pseudo code uses these definitions and shows the address calculation:

```
INDEX = INDEX + MODIFY
if ( INDEX >= (BASE + LENGTH) )
    INDEX = INDEX - LENGTH
if ( INDEX < BASE )
    INDEX = INDEX + LENGTH
```

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 a positive value (an address) 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.

Note that if the value of a length (JL/KL) register is set to zero, the address calculation for circular buffer IALU load and store instructions are equivalent to standard IALU load and store instructions.

- 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. The circular buffer initial index must be within the buffer range ($\text{buffer} \leq \text{index} < \text{base} + \text{length}$).

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.

Example code showing the IALU's support for circular buffer addressing appears in [Listing 7-2](#), and a description of the word access pattern for this example code appears in [Figure 7-22](#).

As shown in [Listing 7-2](#), 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 7-2. 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 7-22 shows the sequence order in which the IALU code in Listing 7-2 accesses the 11 buffer locations in one pass. On the twelfth access, the circular buffer wraparound occurs.

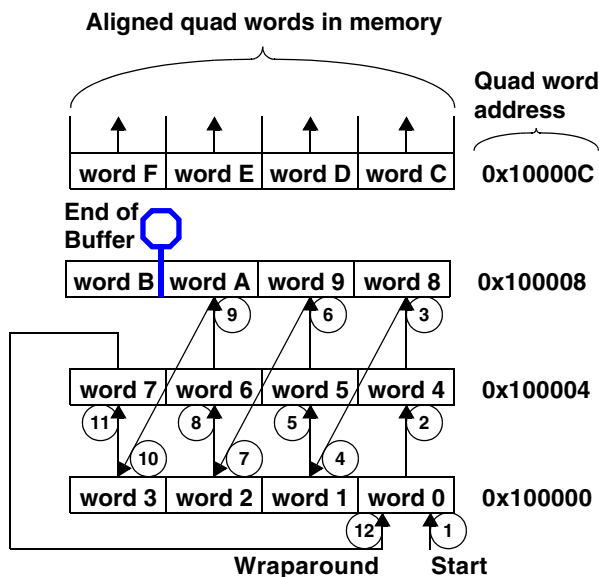


Figure 7-22. Circular Buffer Addressing – Word Access Order



Note that the circular buffer option is applicable also for add/subtract IALU instructions (for example, $J5 = J7 + J10 (CB);;$).

IALU Operations

Bit Reverse Addressing

The IALUs support bit reverse addressing through the bit reverse 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.

Figure 7-23 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 $0x9494$.

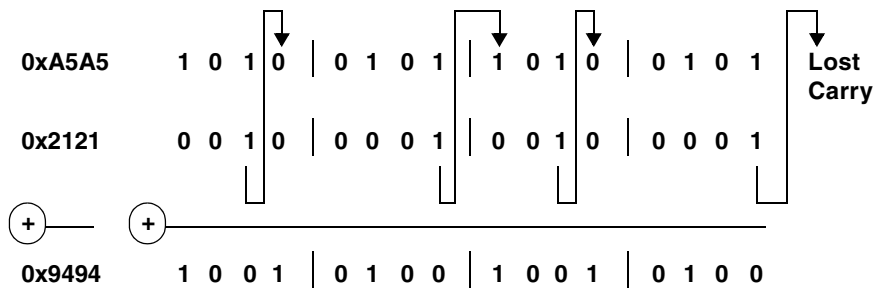


Figure 7-23. Bit Reverse Carry Operation (BR Option)

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 7-3 demonstrates bit reverse addressing. The word access order resulting from the bit reverse carry on the address appears in Figure 7-24.

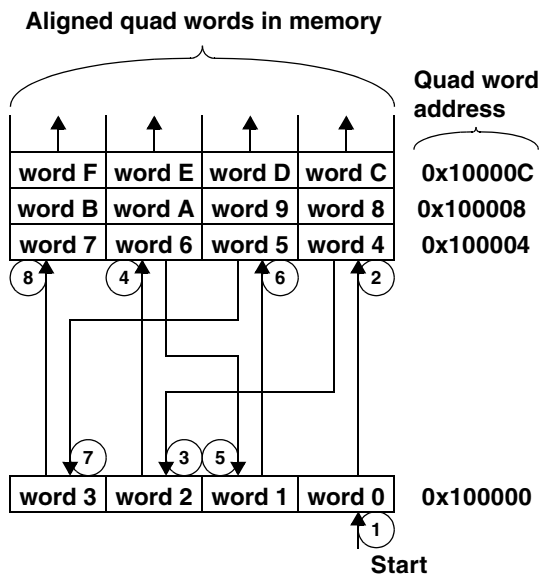


Figure 7-24. Bit Reverse Addressing – Word Access Order

Listing 7-3. 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;
```

IALU Operations

```
_main:
    j0 = j31 + ADDRESS(input) ;; /* Input pointer */
    j4 = j31 + ADDRESS(output) ;; /* Output pointer */
    LCO = N ;; /* Set up loop counter */
_my_loop:
    xr0 = BR [J0 += N/2] ;; /* Data read with bit reverse; modifier must be equal to N/2 */
    if NLC0E, jump _my_loop ; [j4+=1] = xr0 ;; /* Write linear */
```

Some important points to remember when reading [Listing 7-3](#) include:

- The number of bit reverse locations (length of buffer) must be a power of two (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 eight, and the buffer start address is aligned to an address that is a multiple of eight.
- The assembler provides evaluation of expressions. For example, where $N/2$ appears in the code, the assembler evaluates the expression as four. 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 7-4](#).
- [Listing 7-3](#) 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 8-14](#).

Table 7-4. 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)



Note that the bit reverse option is applicable also for add/subtract IALU instructions (for example, `J5 = J7 + J10 (BR);;`).

IALU Operations

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 */
```

```
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 */
```

Note that, because compute block register file registers may be treated as quad registers, there are many transfer options. For example,

```
XYR3:0 = XYR31:28 ;;
```


```
/* This instruction is equivalent to (XR3:0 = XR31:28;  
YR3:0 = YR31:28;), but uses only one instr. slot and one IALU  
slot. Also, XYR = YXR operations are legal. */
```

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 7-47, note the number of instructions that can use 32-bit immediate data ($\langle Imm32 \rangle$). These instructions all require an immediate extension.

The assembler 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. For example, if an immediate extension is used for an immediate jump instruction, then that is the only immediate extension permitted on that instruction line.

IALU Examples

Listing 7-4 and Listing 7-5 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 7-4. IALU Instruction Examples

```
J1 = J0 + 0x81 ; /* Js = Jm + Imm8 data */

K2 = ROTR K0 ; /* Ks = 1 bit right rotate Km */

SQCTL = [J3 + J5] ; /* Load Ureg from addr. Jm + Jn */
```

IALU Examples

```
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 Q [J0 += 0x8181] ; /* Load Rs from addr. Jm; post-modify
Jm with Imm32 and circular buffer addressing; uses immediate
extension */

CJMP = 0x8181 ; /* Load Ureg_s from Imm32; */

KSTAT = JSTAT ; /* Load Ureg_s from Ureg_m */

YR3:0 = XR7:4 ; /* Load Ureg_sq from Ureg_mq */
```

Listing 7-5. DAB Usage Example

```
/*the program loads all the elements of input_buffer using
regular quad fetches in yr15:0 registers. These loads are done
without using DAB because input_buffer is quad aligned*/
```

```
j0 = j31 + input_buffer;;

yr3:0 = q[j0+=4];; /* yr0=0x00001111, yr1=0x22223333,
yr2=0x44445555, yr3=0x66667777 */
yr7:4 = q[j0+=4];; /* yr4=0x88889999, yr5=0xaaabbbb,
yr6=0xcccdddd, yr7=0xeeefffff */
yr11:8 = q[j0+=4];; /* yr8=0x11110000, yr9=0x33332222,
yr10=0x55554444, yr11=0x77776666 */
yr15:12=q[j0+=4];; /* yr12=0x99998888, yr13=0xbbbbaaaa,
yr14=0xddddcccc, yr15=0xffffeeee */
```

```

/* the program now loads all the elements of input_buffer as a
circular buffer beginning with the third element of the buffer
using DAB in xr15:0 */

j0 = j31 + input_buffer + 2;;
jb0 = j31 + input_buffer;;
jl0 = j31 + 16;;

xr3:0 = dab q[j0+=4];; /* xDAB initialized with data from the
nearest quad boundary; xDAB=0x00001111, 0x22223333, 0x44445555,
0x66667777 */
xr3:0 = dab q[j0+=4];; /* xr0=0x44445555, xr1=0x66667777,
xr2=0x88889999, xr3=0xaaaabbbb */
xr7:4 = dab q[j0+=4];; /* xr4=0xccccdddd, xr5=0xeeeeffff,
xr6=0x11110000, xr7=0x33332222 */
xr11:8 = dab q[j0+=4];; /* xr8=0x55554444, xr9=0x77776666,
xr10=0x99998888, xr11=0xbbbbaaaa */
xr15:12= dab q[j0+=4];; /* xr12=0xddddcccc, xr13=0xffffeeee,
xr14=0x00001111, xr15=0x22223333 */

/* the program now loads all the short elements of input_buffer
as a circular buffer beginning with the fourth short element of
the buffer using DAB in yr31:16 */

j0 = j31 + 2*input_buffer + 3;;
jb0 = j31 + 2*input_buffer;;
jl0 = j31 + 32;;

yr19:16 = sdab q[j0+=8];; /* yDAB initialized with data from the
nearest quad boundary; yDAB=0x00001111, 0x22223333, 0x44445555,
0x66667777; ysDAB=0x1111, 0x0000,0x3333, 0x2222, 0x5555, 0x4444,
0x7777, 0x6666 */

```

IALU Examples

```
yr19:16 = sdab q[j0+=8];; /* yDAB contains now 0x88889999,  
0xaaaabbbb, 0xccccdddd, 0xeeeeffff; ysDAB=0x9999, 0x8888, 0xbbbb,  
0xaaaa, 0xdddd, 0xcccc, 0xffff, 0xeeee */  
/* 0x2222 and 0x5555 pass in as yr16=0x55552222 */  
/* 0x4444 and 0x7777 pass in as yr17=0x77774444 */  
/* 0x6666 and 0x9999 pass in as yr18=0x99996666 */  
/* 0x8888 and 0xbbbb pass in as yr19=0xbbbb8888 */  
/* yDAB contains now 0x88889999, 0xaaaabbbb, 0xccccdddd,  
0xeeeeffff; ysDAB=0x9999, 0x8888, 0xbbbb, 0xaaaa, 0xdddd, 0xcccc,  
0xffff, 0xeeee
```

```
yr23:20 = sdab q[j0+=8];; //yDAB contains now 0x11110000,  
0x33332222, 0x55554444, 0x77776666; ysDAB=0x0000, 0x1111, 0x2222,  
0x3333, 0x4444, 0x5555, 0x6666, 0x7777  
/* 0xaaaa and 0xdddd pass in as yr20=0xddddaaaa */  
/* 0xcccc and 0xffff pass in as yr21=0xffffcccc */  
/* 0xeeee and 0x0000 pass in as yr22=0x0000eeee */  
/ 80x1111 and 0x2222 pass in as yr23=0x22221111 */
```

```
yr27:24 = sdab q[j0+=8];; /* yDAB contains now 0x99998888,  
0xbbbbaaaa, 0xddddcccc, 0xffffeeee; ysDAB=0x8888, 0x9999, 0xaaaa,  
0xbbbb, 0xcccc, 0xdddd, 0xeeee, 0xffff  
/* 0x3333 and 0x4444 pass in as yr24=0x44443333 */  
/* 0x5555 and 0x6666 pass in as yr25=0x66665555 */  
/* 0x7777 and 0x8888 pass in as yr26=0x88887777 */  
/* 0x9999 and 0xaaaa pass in as yr27=0xaaaa9999 */
```

```
yr31:28 = sdab q[j0+=8];; * /yDAB contains now 0x00001111,  
0x22223333, 0x44445555, 0x66667777; ysDAB=0x1111, 0x0000, 0x3333,  
0x2222, 0x5555, 0x4444, 0x7777, 0x6666 */  
/* 0xbbbb and 0xcccc pass in as yr28=0xccccbbbb */  
/* 0xdddd and 0xeeee pass in as yr29=0xeeeedddd */  
/* 0xffff and 0x1111 pass in as yr30=0x1111ffff */  
/* 0x0000 and 0x3333 pass in as yr31=0x33330000 */
```

IALU Instruction Summary

The following listings show the IALU instructions' syntax:

- [Listing 7-6](#) “IALU Integer, Logical, and Function Instructions”
- [Listing 7-7](#) “IALU Ureg Register Load (Data Addressing) Instructions”
- [Listing 7-8](#) “IALU Dreg Register Load Data Addressing (and DAB Operation) Instructions”
- [Listing 7-9](#) “IALU Ureg Register Store (Data Addressing) Instructions”
- [Listing 7-10](#) “IALU Dreg Register Store (Data Addressing) Instructions”
- [Listing 7-11](#) “IALU Universal Register Transfer 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 2-6](#). 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.

IALU Instruction Summary

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 7-42, [“Instruction Line Syntax and Structure”](#) on page 1-23, and [“Instruction Parallelism Rules”](#) on page 1-27.

Listing 7-6. IALU Arithmetic, Logical, and Function Instructions

```
 $Js = Jm + | - Jn | <Imm8> | <Imm32> \{ ( \{ CJMP | CB | BR \} \{ NF \} ) \} ;$   
 $JB0 | JB1 | JB2 | JB3 | JL0 | JL1 | JL2 | JL3 = Jm + | - Jn | <Imm8> | <Imm32> \{ ( NF ) \} ;$   
 $Js = Jm + Jn | <Imm8> | <Imm32> + JC \{ ( NF ) \} ;$   
 $Js = Jm - Jn | <Imm8> | <Imm32> + JC - 1 \{ ( NF ) \} ;$   
 $Js = ( Jm + | - Jn | <Imm8> | <Imm32> ) / 2 \{ ( NF ) \} ;$   
 $COMP(Jm, Jn | <Imm8> | <Imm32>) \{ ( U ) \} ;$   
 $Js = MAX | MIN ( Jm, Jn | <Imm8> | <Imm32> ) \{ ( NF ) \} ;$   
 $Js = ABS Jm \{ ( NF ) \} ;$ 
```

```

Js = Jm OR|AND|XOR|AND NOT Jn|<Imm8>|<Imm32> {(NF)} ;
Js = NOT Jm {(NF)} ;
Js = ASHIFTR|LSHIFTR Jm {(NF)} ;
Js = ROTR|ROTL Jm {(NF)} ;

Ks = Km +|- Kn|<Imm8>|<Imm32> {(CJMP|CB|BR){NF}} ;
KB0|KB1|KB2|KB3|KLO|KL1|KL2|KL3 = Km +|- Kn|<Imm8>|<Imm32> {(NF)} ;
Ks = Km + Kn|<Imm8>|<Imm32> + KC {(NF)} ;
Ks = Km - Kn|<Imm8>|<Imm32> + KC - 1 {(NF)} ;
Ks = (Km +|- Kn|<Imm8>|<Imm32>)/2 {(NF)} ;
COMP(Km, Kn|<Imm8>|<Imm32>) {(U)} ;
Ks = MAX|MIN (Km, Kn|<Imm8>|<Imm32>) {(NF)} ;
Ks = ABS Km {(NF)} ;
Ks = Km OR|AND|XOR|AND NOT Kn|<Imm8>|<Imm32> {(NF)} ;
Ks = NOT Km {(NF)} ;
Ks = ASHIFTR|LSHIFTR Km {(NF)} ;
Ks = ROTR|ROTL Km {(NF)} ;

```

Listing 7-7. IALU Ureg Register Load (Data Addressing) Instructions

```

Ureg_s = {CB|BR} [Jm +|+= Jn|<Imm8>|<Imm32>] ;
Ureg_sd = {CB|BR} L [Jm +|+= Jn|<Imm8>|<Imm32>] ;
Ureg_sq = {CB|BR} Q [Jm +|+= Jn|<Imm8>|<Imm32>] ;

Ureg_s = {CB|BR} [Km +|+= Kn|<Imm8>|<Imm32>] ;
Ureg_sd = {CB|BR} L [Km +|+= Kn|<Imm8>|<Imm32>] ;
Ureg_sq = {CB|BR} Q [Km +|+= Kn|<Imm8>|<Imm32>] ;

/* Ureg suffix indicates: _s=single, _sd=double, _sq=quad */

```

Listing 7-8. 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>] ;

```

IALU Instruction Summary

```
{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} Q [Jm += Jn|<Imm8>|<Imm32>] ;

{XY}Rsq = XDAB Q [Jm += Jn|<Imm8>|<Imm32>] ;
{XY}Rsq = XSDAB Q [Jm += Jn|<Imm8>|<Imm32>] ;
{XY}Rsq = YDAB Q [Jm += Jn|<Imm8>|<Imm32>] ;
{XY}Rsq = YSDAB 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} Q [Km += Kn|<Imm8>|<Imm32>] ;

{XY}Rsq = XDAB Q [Km += Kn|<Imm8>|<Imm32>] ;
{XY}Rsq = XSDAB Q [Km += Kn|<Imm8>|<Imm32>] ;
{XY}Rsq = YDAB Q [Km += Kn|<Imm8>|<Imm32>] ;
{XY}Rsq = YSDAB 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 */
```

Listing 7-9. 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 7-10. 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 7-11. IALU Universal Register Transfer Instructions

```

Ureg_s = <Imm15> | <Imm32> ;
Ureg_s = Ureg_m ;
Ureg_sd = Ureg_md ;
Ureg_sq = Ureg_mq ;

```

IALU Instruction Summary

8 PROGRAM SEQUENCER

The ADSP-TS201 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 8-1](#)) manages program execution.

The sequencer fetches instructions from memory, extracts from the fetched data an instruction-line every cycle, decodes for each instruction in the line which unit(s) should execute it, and sends each instruction to its execution unit(s). The sequencer also executes program flow control instructions and monitors the program for stall requirements. The operations that the sequencer supports include:

- Supply address of next code quad word to fetch to the Instruction Alignment Buffer (IAB)
- Extracts the next instruction line to execute from the instruction alignment buffer
- Executes the control flow instructions, including:
 - Evaluate conditions
 - Maintain the Branch Target Buffer (BTB) for branch delay minimization
 - Decrement loop counters

- Stall the pipeline when required
- Respond to interrupts (with changes to program flow)

Figure 8-2 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 8-19](#).
- 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 8-2](#), the sequencer can support a number of program flow variations. Program flow in the processor is mostly linear with the processor executing program instructions sequentially. This linear flow varies occasionally when the program uses non-sequential program structures, such as those illustrated in [Figure 8-3](#).

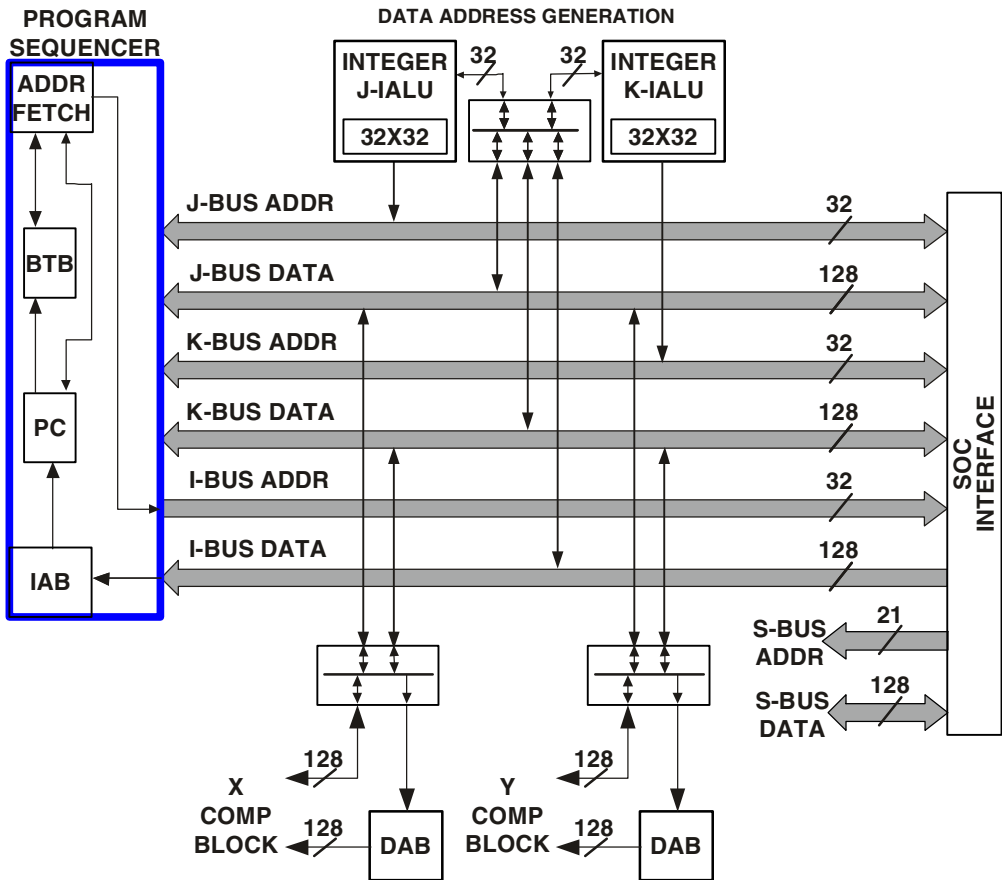


Figure 8-1. Program Sequencer

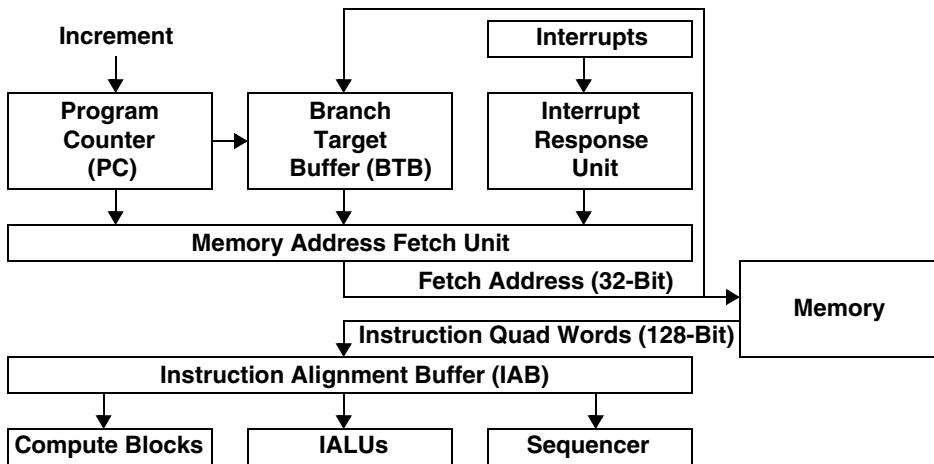


Figure 8-2. Sequencer Detailed Block Diagram

Non-sequential structures direct the processor to execute an instruction that is not at the next sequential address. These structures include:

- **Loops.** One sequence of instructions executes several times with 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.

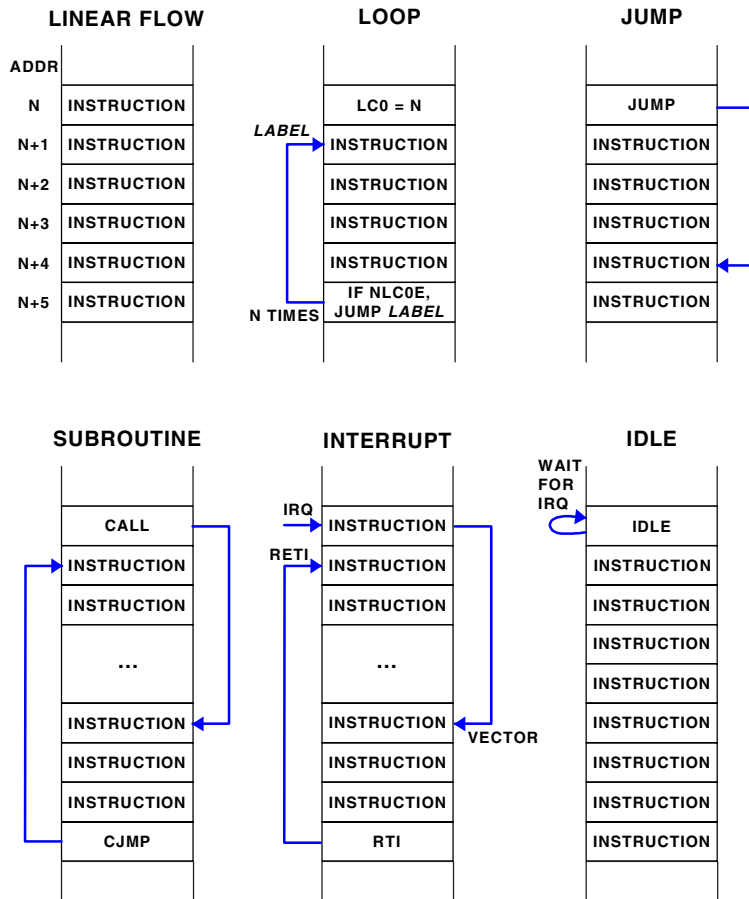


Figure 8-3. Program Flow Variations

For information on using each type of program flow, see [“Sequencer Operations” on page 8-8](#).

To support high speed execution, the ADSP-TS201 processor uses an instruction pipeline. The processor 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 8-4 shows the instruction pipeline stages and shows how the pipeline interacts with the BTB, IAB, and memory pipeline.

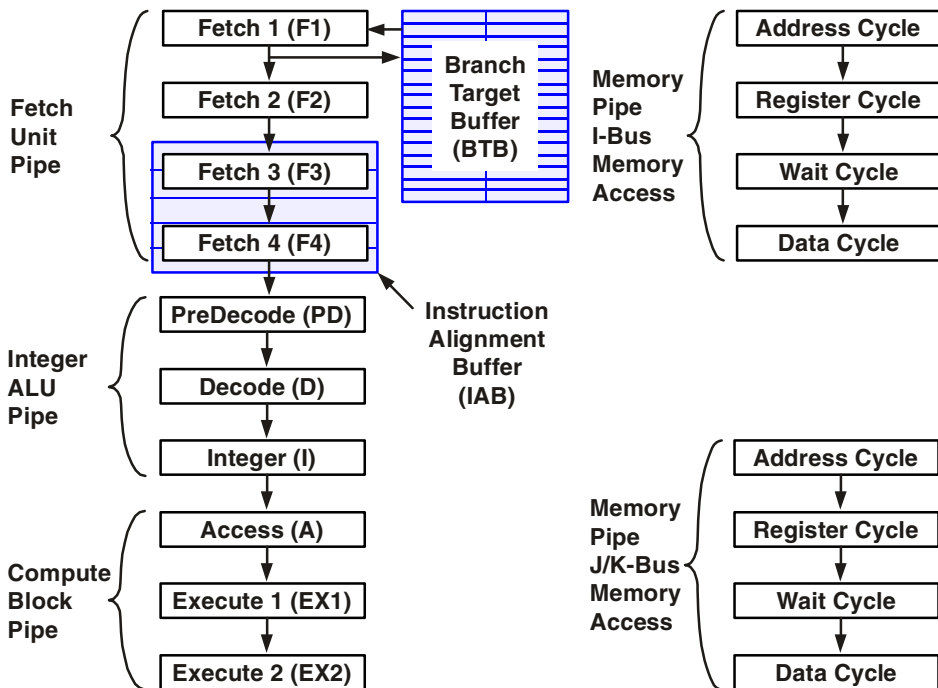


Figure 8-4. Instruction Pipeline, IAB, and BTB Versus Memory Pipeline

From start to finish, an instruction line requires at least ten cycles to traverse the pipeline. The execution throughput is one instruction line every processor core clock (CCLK) cycle. Due to pipeline effects for different execution units and memory access effects, it is convenient to analyze the pipeline flow in terms of sequential unit pipes.

The instruction fetch (F1, F2, F3 and F4) pipe stages fetch memory quad words and parse them into instruction lines. In the F1 stage, the fetch address is issued to memory and is issued to the BTB. At this point, the

BTB predicts whether the next fetch is sequential or a branch with prediction “taken” should take place. For more information, see [“Branching Execution” on page 8-19](#) and [“Branch Target Buffer \(BTB\)” on page 8-42](#). By the F4 stage, the memory quad word returns to the IAB, the IAB parses the instruction line from the memory quad word, and the sequencer issues each instruction from the line to its execution unit. The fetch pipeline stages work with memory quad words, while all the following pipeline stages work with instruction lines.

The remaining pipe stages are instruction driven. The execution differs between the IALU, compute block, and sequencer. The instruction driven pipe stages are PreDecode (PD), 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.

PreDecode	The sequencer issues predicted jumps at this stage and issues stalls for IALU dependency if needed. The IALU reads from register file and starts address/instruction calculations.
Decode	The sequencer recognizes mispredicted conditional jumps for IALU conditions at this stage and issues stalls for compute block dependency if needed. The IALUs complete address/instruction calculations and update status (IALU conditions).
Integer	The IALUs issue addresses for memory access.
Access	The IALUs receive an acknowledge to begin memory access (for applicable instructions). Compute block registers are accessed (for applicable instructions).
Execute 1	The compute blocks begin calculations. Data is transferred on the J/K buses.

Sequencer Operations

Execute 1 and 2 The sequencer recognizes mispredicted conditional jumps for compute block conditions at this stage. The compute blocks complete calculations and update status (compute block conditions). The IALUs complete execution of memory access instructions.


These differences in pipe operation between IALUs and computation units cause different pipeline effects when branching. For more information, see [“Instruction Pipeline Operations” on page 8-34](#).

 As shown in [Figure 8-4](#), the memory pipeline operates in parallel with the instruction pipeline. Stalls in the address cycle or register cycle of the memory pipeline appear as stalls in the corresponding stages of the instruction pipeline. For more information, see [“Dependency and Resource Effects on Pipeline” on page 8-64](#) and the *ADSP-TS201 TigerSHARC Processor Hardware Reference*.

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 8-14](#)
- [“Branching Execution” on page 8-19](#)
- [“Looping Execution” on page 8-22](#)
- [“Interrupting Execution” on page 8-26](#)

 Depending on programming and pipeline effects, some branching variations require the processor to automatically insert stall cycles. For more information on pipeline effects and stalls, see [“Instruction Pipeline Operations” on page 8-34](#).

Conditional instructions begin with the syntax `IF Condition`. For sequencer conditional instructions, the *Condition* can be:

- Any compute block condition, such as AEQ, ALT, ALE, MEQ, MLT, MLE, SEQ, SLT, SF0, or SF1
- Any IALU condition, such as JEQ, JLT, JLE, KEQ, KLT, KLE, ISF0, or ISF1
- Any sequencer condition, such as BM, LCOE, LC1E, FLAG0_IN, FLAG1_IN, FLAG2_IN, or FLAG3_IN
- Any negated compute block, IALU, or sequencer condition; condition prefixed with `N` (for NOT)

For a complete list of conditions and example usage, see [“Sequencer Instruction Summary” on page 8-97](#). For descriptions of these conditions, see [“ALU Execution Conditions” on page 3-15](#), [“Multiplier Execution Conditions” on page 5-22](#), [“Shifter Execution Conditions” on page 6-19](#), and [“IALU Execution Conditions” on page 7-13](#).

Almost all 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, BTBEN, BTBDIS, BTBLOCK, BTBELOCK, TRAP, and EMUTRAP. For more information on conditional operations, see [“Conditional Execution” on page 8-14](#).

In the sequencer, there are two registers that provide control and status. The Sequencer Control (SQCTL) and Sequencer Status (SQSTAT) registers appear in [Figure 8-5](#), [Figure 8-6](#), and [Figure 8-7](#).

Sequencer Operations

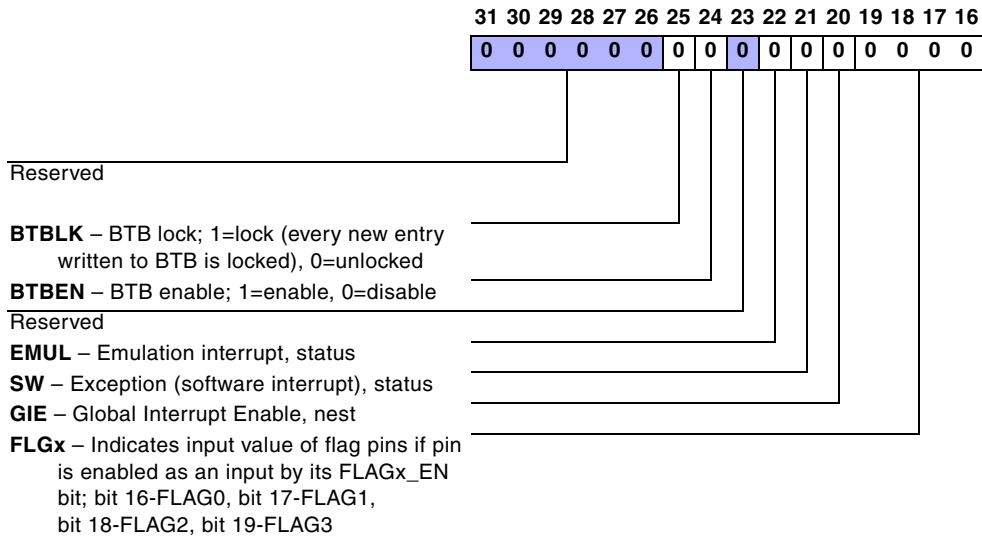


Figure 8-5. SQSTAT (Upper) Register Bit Descriptions

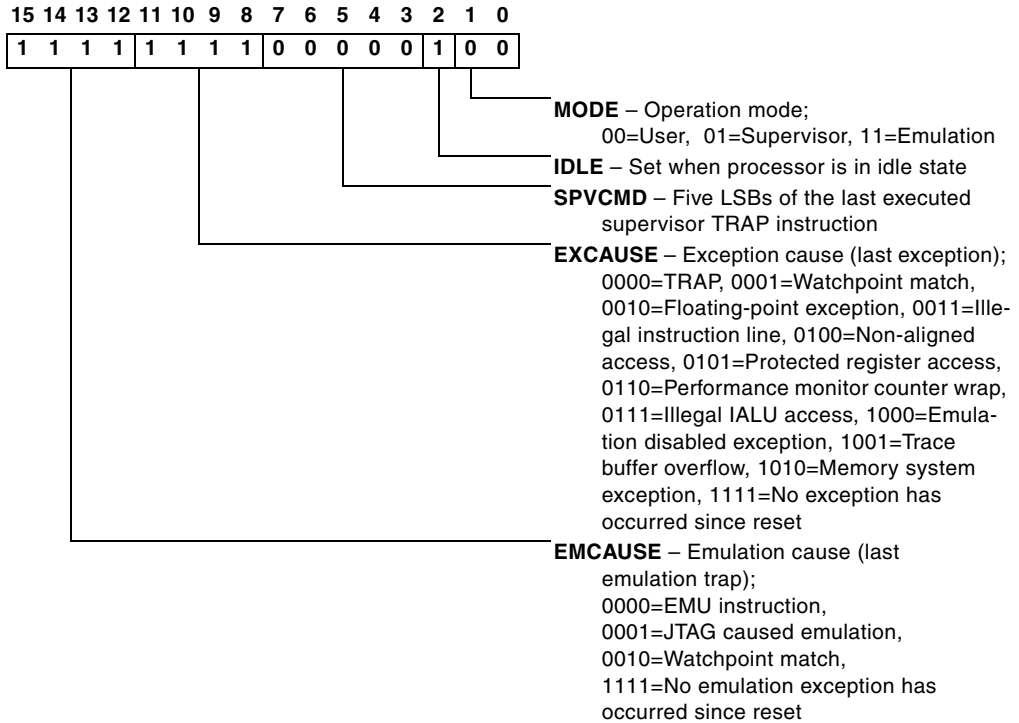


Figure 8-6. SQSTAT (Lower) Register Bit Descriptions

Sequencer Operations

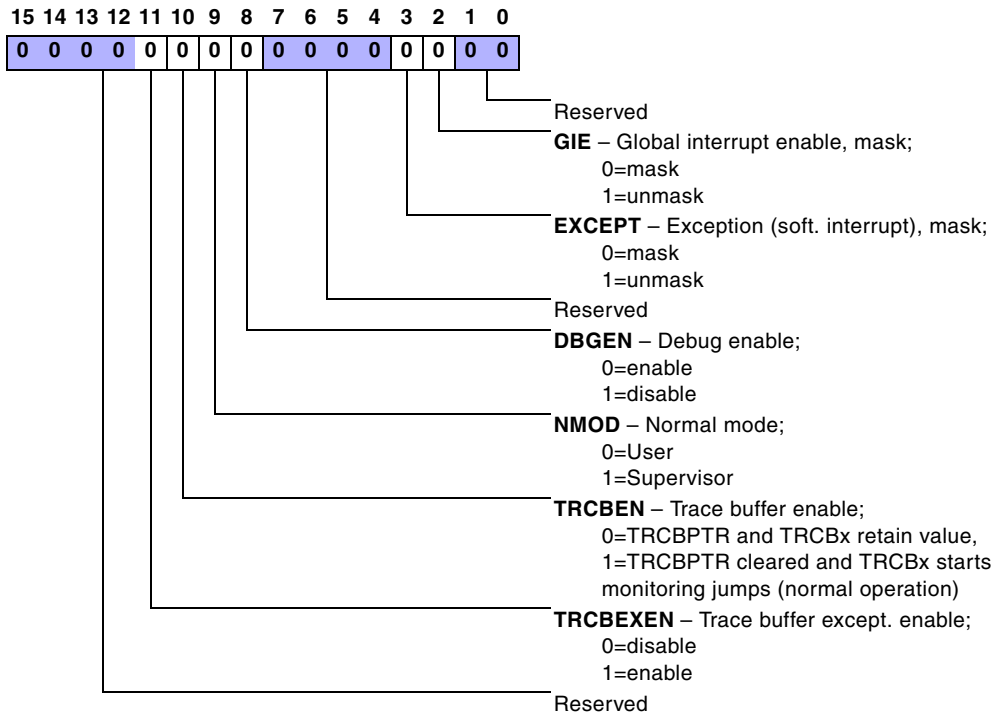


Figure 8-7. SQCTL (Lower) Register Bit Descriptions^{1,2}

- 1 Bits 31–16 (not shown) are reserved (=0).
- 2 There is a five cycle stall after any write to the SQCTL register.

The Sequencer Control (SQCTL) and Sequencer Status (SQSTAT) 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 processor automatically goes into supervisor mode. There is a three cycle latency on explicitly switching between these modes.
- **Global interrupt enable.** Using the global interrupt enable (GIE) bit, programs can globally mask interrupts.
- **Exception mask/pointer.** Using the exception (EXCEPT) bits in the SQCTL and SQSTAT registers, programs can mask and unmask software interrupts (exceptions).
- **Flag status.** Using the flag input (FLGX) bits, programs can observe the input value of an input flag pin.¹ Input/output control for flag pins is available in the FLAGREG register.
- **Branch Target Buffer control.** Using the BTB enable (BTBEN) and BTB lock (BTBLK) bits, programs control the BTB operation.

Other status information is also available. See [Figure 8-5](#), [Figure 8-6](#), and [Figure 8-7](#). Note that software reset of the processor core can be initiated using the SWRST bit in the EMUCTL register.

¹ The flag pin bit updates the corresponding flag pin with a delay of between 1 and 3 SCLK cycles after the value is written to the FLAGREG register. 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.

Conditional Execution

All ADSP-TS201 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—or execute unconditionally.

In a conditional compute or IALU instructions (*IF ... D0*), the execution of the instruction slots following the condition on the instruction line can each depend on the specified condition at the beginning of the instruction line. Conditional compute instructions take the form:

```
IF Condition;  
D0, Instruction; D0, Instruction; D0, Instruction ;;
```

This syntax permits up to three instruction slots to be controlled by a condition. Omitting the *D0* before an instruction slot makes the instruction in the slot unconditional.

[Listing 8-1](#) shows some example conditional ALU instructions. For a description of the ALU conditions used in these examples, see [“ALU Execution Conditions”](#) on page 3-15.

¹ Except for NOP, IDLE, BTBINV, BTBEN, BTBDIS, BTBLOCK, BTBELOCK, TRAP and EMUTRAP

Listing 8-1. 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 */

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 JLE; D0, XR0 = R1 + R2 ;;
/* conditional execution of the add in compute block X is based
on the JLE condition in the J-IALU */

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 (IF ... ELSE), the execution of the program sequencer instruction slot and the instruction slots that follow it on the instruction line can each depend on the specified 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 instruction slots after the sequencer instruction slot to be controlled by a condition. Omitting the ELSE before the instruction slot makes the instruction in the slot unconditional.

Sequencer Operations

This syntax permits program sequencer instructions to be based on compute conditions. [Listing 8-2](#) shows some example conditional program sequencer instructions based on ALU conditions.

Listing 8-2. 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 (and XAEQ is false);
the subtract in compute block Y is unconditional (always
executes) */

If MEQ, CJMP; ELSE R4 = R6 + R7;;
/* conditional execution of the jump is based on ORing the MEQ
condition in compute blocks X and Y; the add instruction is exe-
cuted in each compute based on the same compute flag - X add if
not XMEQ and Y add if not YMEQ */
```

The No Flag Update (NF) option has a usage restriction for conditional instructions (IF...DO and IF...ELSE). Either all instruction slots in a conditional instruction line generate flag status, or all slots use the (NF) option to prevent status generation. Instructions for which (NF) does not apply are exempt from this restriction. For example, .

```
IF ALE;
    DO, XR0 = R1 + R2 (NF) ;
    DO, XR5 = LSHIFT R4 BY -4 (NF) ;
```

```
D0, J3 = J5 + J29 (NF) ;;
/* all slots in this instruction line use (NF)*/

IF AEQ, JUMP my_label ;
ELSE, J1 = J0 + J2 (NF) ;
ELSE, XRO = R1 - R3 (NF) ;
ELSE, YRO = R5 * R6 (NF) ;;
/* all slots in this instruction line use (NF)*/

IF AEQ, JUMP my_other_label;
ELSE, COMP(J3,J4) ;
ELSE, XRO = R1 - R3 (NF) ;
ELSE, YRO = R5 * R6 (NF);;
/* COMP does not support (NF), but it can appear on NF line */
```

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 processor automatically uses an immediate extension (the second instruction slot) to hold the extra address bits.

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 15-bit address (if possible). 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 8-19](#).

Sequencer Operations

To provide conditional instruction support for static conditions, the sequencer has a Static Flag (SFREG) register (appears in [Figure 8-8](#)). The sequencer uses this register to store status flag values from the compute blocks and IALUs for later usage in conditional instructions.

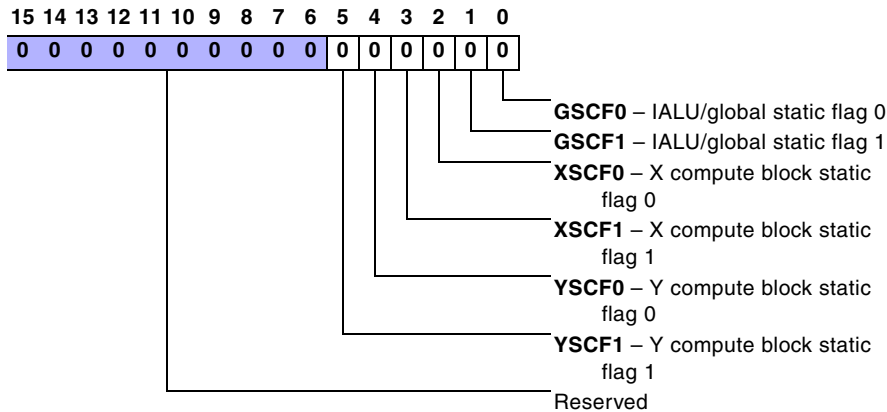


Figure 8-8. SFREG (Lower) Register Bit Descriptions¹

¹ Bits 31–16 (not shown) are reserved (=0).

For examples using the SFREG conditions in the compute blocks and IALUs, see “ALU Static Flags” on page 3-16, “Multiplier Static Flags” on page 5-23, “Shifter Static Flags” on page 6-20, “ALU Static Flags” on page 3-16, and “IALU Static Flags” on page 7-14.



Some combinations of instructions following an SFREG register load produce data dependency stalls. For more information, see the data dependency stall list in [Table 8-1 on page 8-66](#).

Branching Execution

The sequencer supports branching execution with the `JUMP`, `CALL`, `CJMP_CALL`, `CJMP`, `RETI`, `RDS`, and `RTI` instructions. [Figure 8-3 on page 8-5](#) provides a high level comparison between all branch variations. Looking at [Figure 8-9](#), note some additional details about the `JUMP`, `CALL`, and `CJMP_CALL` operations.

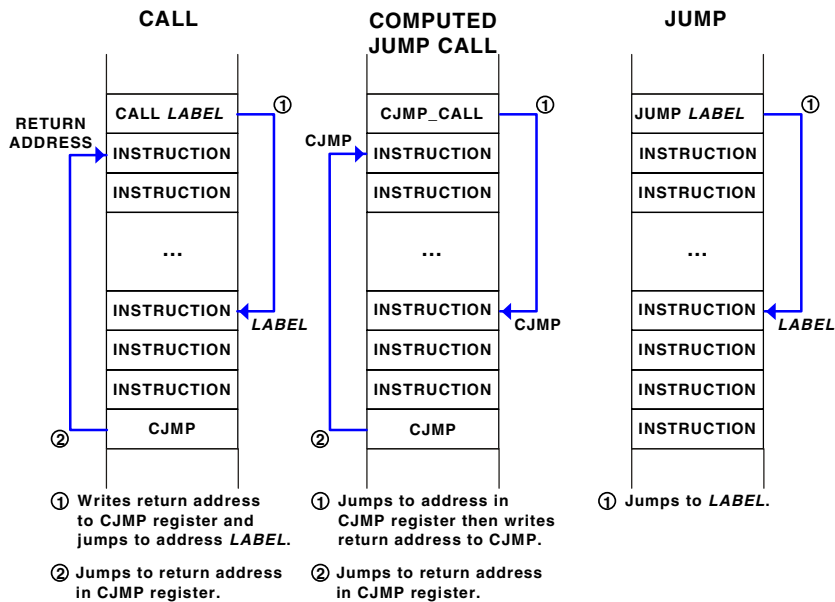


Figure 8-9. Call Versus Computed Jump Call Versus Jump


A `CALL` instruction transfers execution to address *Label* (or to an immediate 15- 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. The `CJMP` instruction at the end of the subroutine causes the sequencer to jump to the address in the `CJMP` register.

Sequencer Operations

A `CJMP_CALL` instruction transfers execution to a subroutine using a computed jump address (from the `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.

A `JUMP` instruction transfers execution to address *Label* (or an immediate 15- 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 PC's 32-bit 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 17-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 15-bit PC-relative  
address (if possible) */
```


```
CALL iir_routine (ABS) ;;
/* 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 8-92](#).

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 iir_routine (ABS) (NP) ;;
/* this conditional branching instruction is a not predicted
branch */
```

 Unconditional branches are treated as conditional 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 (F1).

Sequencer Operations

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 nine or five stall cycles to zero. For more information, see [“Branch Target Buffer \(BTB\)” on page 8-42](#).

Looping Execution

The sequencer supports zero-overhead and near-zero-overhead looping execution. As shown in [Figure 8-3 on page 8-5](#), 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 8-3](#).

Testing the loop counter expired or not expired conditions affects the loop counter registers. Every reference to NLCOE, NLC1E, LCOE, or LC1E decrements the associated loop counter registers, and the condition evaluates true when the counter reaches zero. Counters are first decremented and then tested.

Listing 8-3. Zero-Overhead Loop Example

/ This example shows how to setup a loop to eliminate stalls. The important points are the number of cycles between the LCx load and loop end, alignment of the quad word seven cycles after the LCx load, and the alignment of the loop label quad word. */*

```

    LCO = N ;;
        /* N = 10; load loop counter */ (E2)
    .align_code 4;
_start_loop:
    NOP ;; /* loop label; quad aligned */ (E1)
    NOP ;; /* any instruction */ (A)
    NOP ;; /* any instruction */ (I)
    NOP ;; /* any instruction */ (D)
    NOP ;; /* any instruction */ (PD)
    NOP ; NOP ; NOP ; NOP ;;
        /* any quad word */ (F4)
    .align_code 4;
    NOP ; NOP ; NOP ; NOP ;;
        /* any aligned quad word;
        7 cycles from load */ (F3 and IAB)
    NOP ; NOP ; NOP ; NOP ;; /* any quad word */ (IAB)
    NOP ; NOP ; NOP ; NOP ;; /* any quad word */ (IAB)
    NOP ; NOP ; NOP ; NOP ;; /* any quad word */ (IAB)
    NOP ; NOP ; NOP ; NOP ;; /* any quad word */ (IAB)
    NOP ; NOP ; NOP ; NOP ;; /* any quad word */ (IAB)
    NOP ; NOP ; NOP ; NOP ;; /* any quad word */ (F2)
    IF NLCOE, jump _start_loop ; NOP ; NOP ; NOP ;;
        /* quad word with NLCOE test */ (F1)
        /* this location is at least
        7 quad words + 7 cycles from LCO load */

```

Sequencer Operations

Many factors influence loop performance. This section discusses the following factors:

- Loop condition test selection (NLCxE and others)
- BTB interaction with the placement of loop counter load and loop condition test
- Instruction pipeline interaction with the placement of the loop label

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 8-8 on page 8-93](#).


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 8-34](#).

Because the processor has a ten stage pipeline, it possible to produce a pipeline related stall if the loop counter load instruction and end of loop test instruction are in the pipeline at the same time. For example, if the LCO loop counter load is executed in pipeline stage E2 and the end NLCOE loop condition test instruction is somewhere in the pipeline, the end of loop jump (branch) may be considered a BTB miss, and a four cycles penalty is introduced first time the loop condition test and jump executes. The problem is that branch may be considered a BTB miss even if the BTB is already loaded with the loop label address.

[Listing 8-3](#) presents one way to eliminate BTB miss stalls stemming from placing the `LCx` load too close to the end of loop test. To identify an effective placement for `LCx` register load relative to the end of the loop test, keep in mind that the pipeline stages F1, F2, and F3 and IAB always contain quad words. Also, note that the pipeline stages F4, PD, D, I, A, E1, and E2 contain instruction lines which may contain minimum one 32 bit word. For convenience, these pipeline stages are noted in [Listing 8-3](#).

Working through the pipeline effects leads to the loop programming guideline that the seventh instruction (at least) after the loop counter load instruction must be quad aligned and the instruction quad word with the end of loop test must be placed at least seven instruction quad words after the aligned instruction. This placement is noted in [Listing 8-3](#). This minimum spacing/alignment between the counter load and test prevents BTB miss stalls; the spacing can be greater, applying to loops of any length. For more information on pipeline effects, see “[Conditional Branch Effects on Pipeline](#)” on page 8-53.

Because the end of loop test leads to iterative jumps (branches) to the start of loop label (address), the address alignment of the start of loop label influences loop performance. If a branch crosses a quad word boundary, there is a one cycle stall added. To avoid this stall, the start of loop label must be quad aligned as shown in [Listing 8-3](#). This alignment ensures that a quad word boundary is not crossed during the end of loop jump.

 Due to instruction pipeline effects and BTB operation on the ADSP-TS201, the position of the end of the loop instruction relative to the next sequential instruction is irrelevant to loop performance on last iteration of the loop.

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 or 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. There are 31 different interrupts. Four of the interrupts are general-purpose interrupts associated with the $\overline{IRQ3-0}$ pins, and one interrupt is a general-purpose vector interrupt (VIRPT). All other interrupts are dedicated.



In the $IMASK$ register, a “1” means the interrupt is *unmasked* (processor recognizes and services interrupt). A “0” means the interrupt is *masked* (processor does not recognize interrupt). The $IMASK$, $ILAT$, and $PMASK$ registers all have the same bit definitions.

The sequencer manages interrupts using the `ILAT`, `IMASK`, and `PMASK` control registers. These registers appear in [Figure 8-12](#), [Figure 8-13](#), [Figure 8-14](#), and [Figure 8-15](#). The `ILAT`, `IMASK`, and `PMASK` control registers 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 interrupt individually (interrupt's bit =0) or all interrupts can be masked using the `GIE` bit in the `SQCTL` register.
- `PMASK` – Interrupt Mask Pointer Register, indicates each interrupt being serviced (interrupt's bit =1)

The priority of interrupts high to low matches their order of appearance in the `IMASK`, `ILAT`, and `PMASK` registers. For more information on interrupt sensitivity, interrupt latching, the IVT, and other interrupt issues, see the *ADSP-TS201 TigerSHARC Processor Hardware Reference*.

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. To produce predictable and correct response, a level sensitive interrupt may not be cleared before ISR execution, unless the interrupt is disabled at the same time (using its `IMASK` bit, using its `GIE` bit in `SQCTL`, or through a higher priority interrupt or exception being active at the same time).

Sequencer Operations

In the sequencer, there are two registers that provide control for interrupt, flag, and timer pins. The Flag Control (FLAGREG) and Interrupt Control (INTCTL) registers appear in [Figure 8-10](#) and [Figure 8-11](#) and support:

- **Interrupt sensitivity.** Using the interrupt edge (IRQ_EDGE) bit, programs select edge or level sensitivity for the $\overline{\text{IRQ3-0}}$ pins 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
- **Timer control.** Using the timer run (TMR0RN and TMR1RN) bits, programs can turn on the two timers independently.

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 (4 x LCLKRAT) instruction lines between the set and the clear to compensate for the delay.

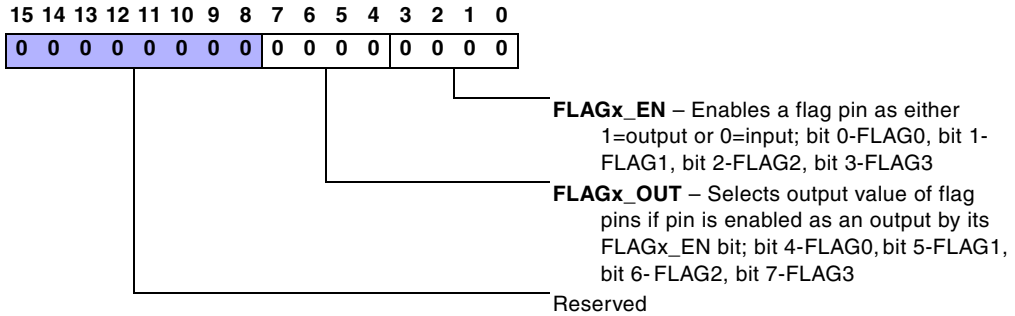


Figure 8-10. FLAGREG (Lower) Register Bit Descriptions¹

1 Bits 31–16 (not shown) are reserved (=0).

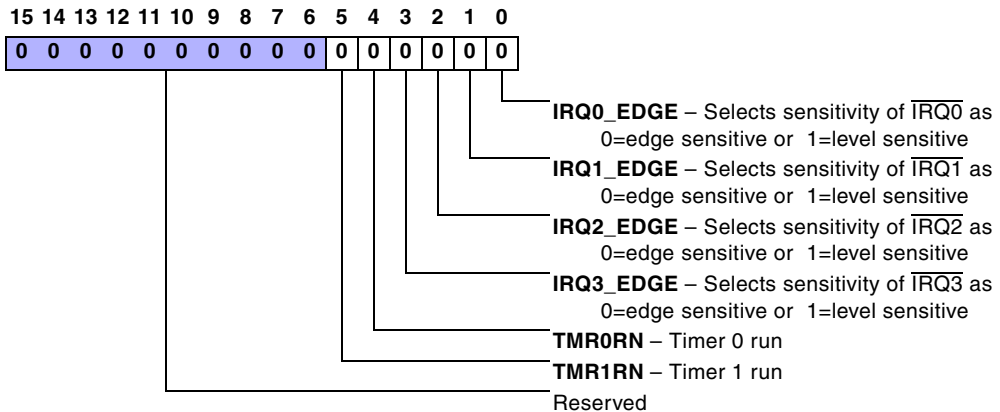


Figure 8-11. INCTL (Lower) Register Bit Descriptions^{1, 2}

1 Bits 31–16 (not shown) are reserved (=0).

2 If the IRQEN strap pin =1 at reset, the reset value for INTCTL is 0x0000 000F.
 If the IRQEN strap pin =0 at reset, the reset value for INTCTL is 0x0000 0000.

Sequencer Operations

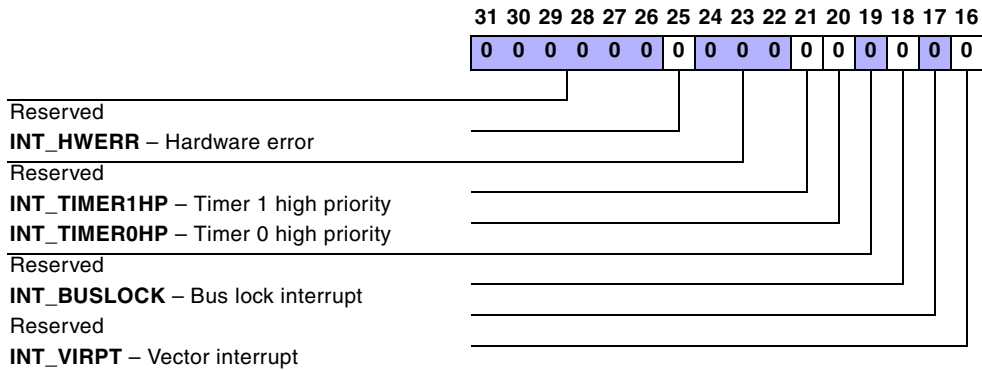


Figure 8-12. IMASKH, ILATH, PMASKH (Upper) Register Bit Descriptions

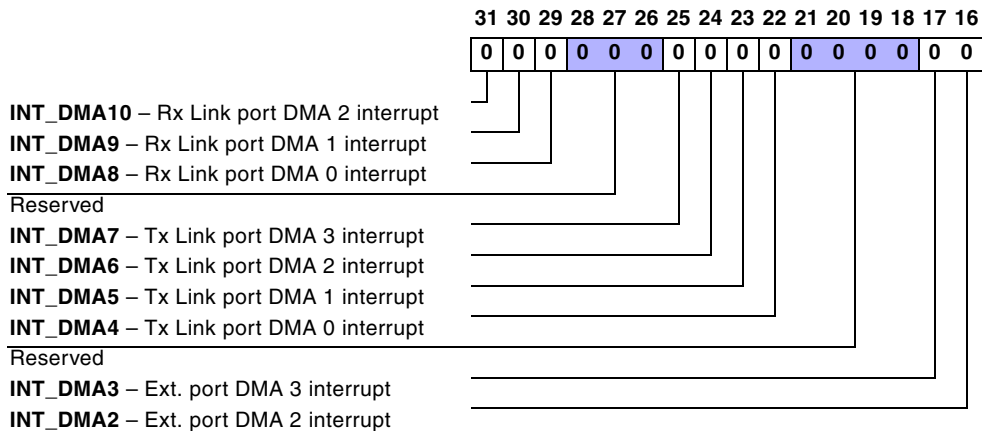


Figure 8-13. IMASKL, ILATL, PMASKL (Upper) Register Bit Descriptions

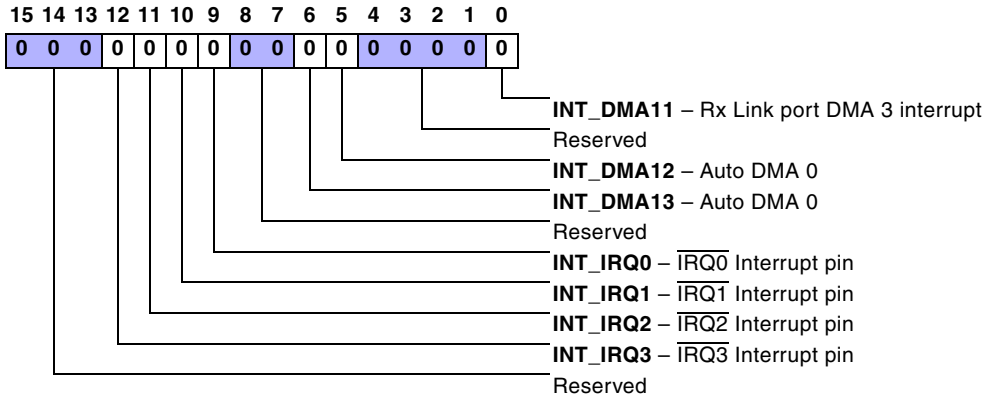


Figure 8-14. IMASKH, ILATH, PMASKH (Lower) Register Bit Descriptions

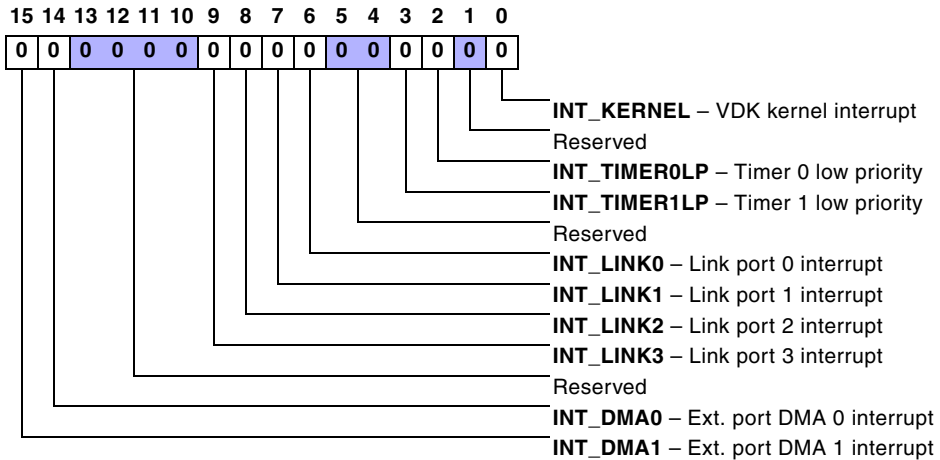


Figure 8-15. IMASKL, ILATL, PMASKL (Lower) Register Bit Descriptions

Sequencer Operations

Looking at [Figure 8-16](#), 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.

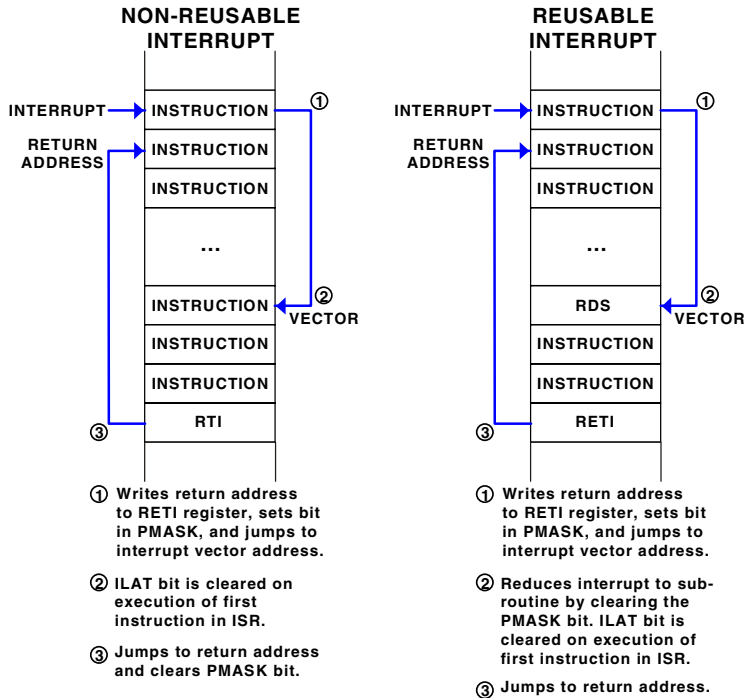


Figure 8-16. Non-Reusable Versus Reusable Interrupt Service

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 processor 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` bit, disabling the interrupt for further usage.
3. The processor executes interrupt service routine. Execution of the first instruction in the ISR clears the interrupt's `ILAT` bit.
4. Sequencer jumps execution to the return address on reaching the `RTI` instruction at the end of the interrupt service routine, and clears the interrupt's `PMASK` bits.

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 processor 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.


Programs must save the contents of the `RETI` register in the stack before the next step.

Instruction Pipeline Operations

3. Sequencer reduces the interrupt to a subroutine on reaching the RDS instruction at the beginning of the interrupt service routine and clears the interrupt's PMASK bits. Execution of the first instruction in the ISR clears the interrupt's ILAT bit.

Because the interrupt's ILAT and PMASK bits are cleared, the interrupt may be latched again before the interrupt service is completed.

4. The processor 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 8-34](#).

Instruction Pipeline Operations

As introduced in the instruction pipeline discussion [on page 8-5](#), the processor has a ten-stage instruction pipeline. The pipeline stages and their relationship to the Branch Target Buffer (BTB) and Instruction Alignment Buffer (IAB) appear in [Figure 8-4 on page 8-6](#). 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 8-17](#).

Looking at [Figure 8-17](#), note that the instruction pipeline stages appear on the vertical axis and CCLK (processor 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. Also, note from [Figure 8-17](#) the order in which results from an operation become available. Because IALU arithmetic operations execute

#1: $XR0 = R1 + R2$; NOP; NOP; NOP;; /* X compute block */ Because IALU instructions execute at I and compute block instructions execute at EX2, results are available earlier for IALU instructions.
 #2: $YR3 = R4 * R5$; NOP; NOP; NOP;; /* Y compute block */
 #3: $J0 = J1 + J2$; NOP; NOP; NOP;; /* J-IALU */
 #4: $K0 = K1 - K2$; NOP; NOP; NOP;; /* K-IALU */

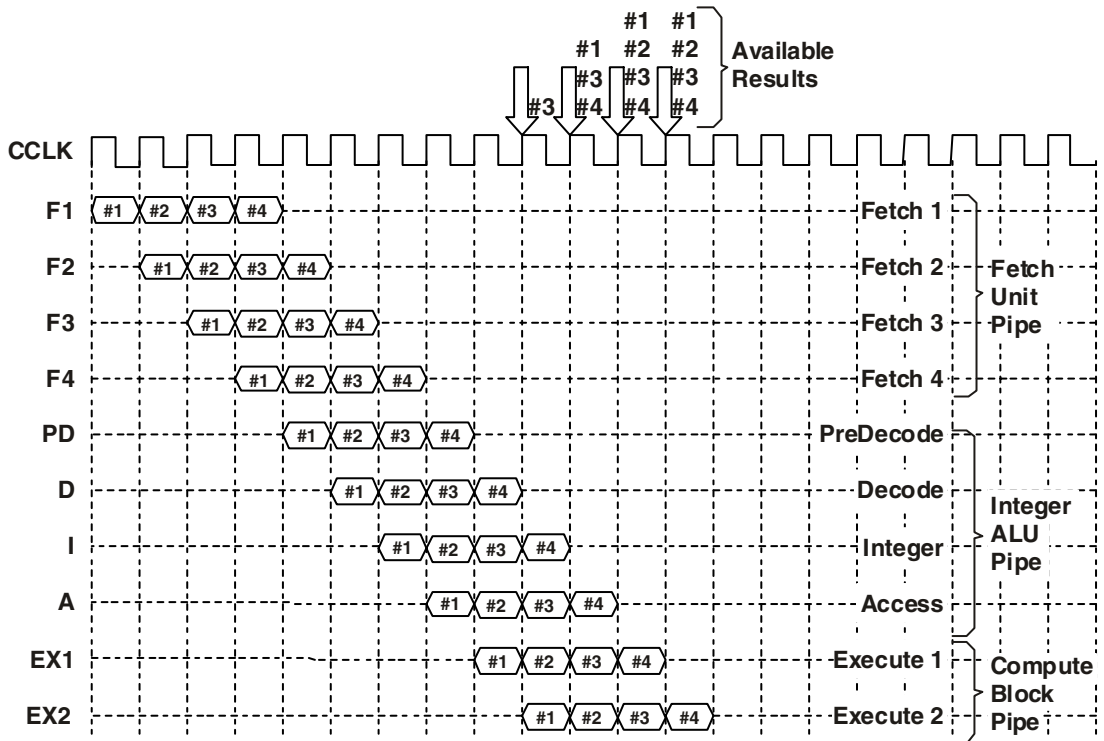


Figure 8-17. Timing for Stages of Instruction Pipeline

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.

The first four stages of the instruction pipeline (F1, F2, F3, and F4) 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 IAB whenever the

Instruction Pipeline Operations

IAB has less than four quad words. Since the execution units can read instructions with a throughput lower than or equal to the fetch throughput of four words every cycle, it is probable that the fetch unit fills the IAB faster than the execution units read the instructions from the IAB. The IAB can be filled with up to six 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 four instruction pipeline stages (PD, D, I, and A) are called the *integer ALU pipe*. The PreDecode and Decode stages are the first stages in this pipe. In the PreDecode cycle, the next full instruction line is extracted from the IAB, and the different instructions are distributed to the execution units during the Decode stage. 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 conditional instructions, and others

The 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 8-17 shows the instruction flow in the *integer ALU pipe*. The IALU instruction's calculation is executed at the Integer stage. If the IALU instruction includes a memory access, the memory block is requested on the Integer stage. In this case, the memory access begins at the Integer stage as long as the memory block 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 stage, the compute block gets the instruction and transfers it to the execution unit (ALU, CLU, 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 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 of CCLK after pipe stage EX2.

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.)

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


Instruction Pipeline Operations

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 is correct.

This type of problem is discussed in detail in [“Dependency and Resource Effects on Pipeline”](#) on page 8-64.

Instruction Alignment Buffer (IAB)

The IAB is a six quad word FIFO as shown in [Figure 8-18](#). 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.

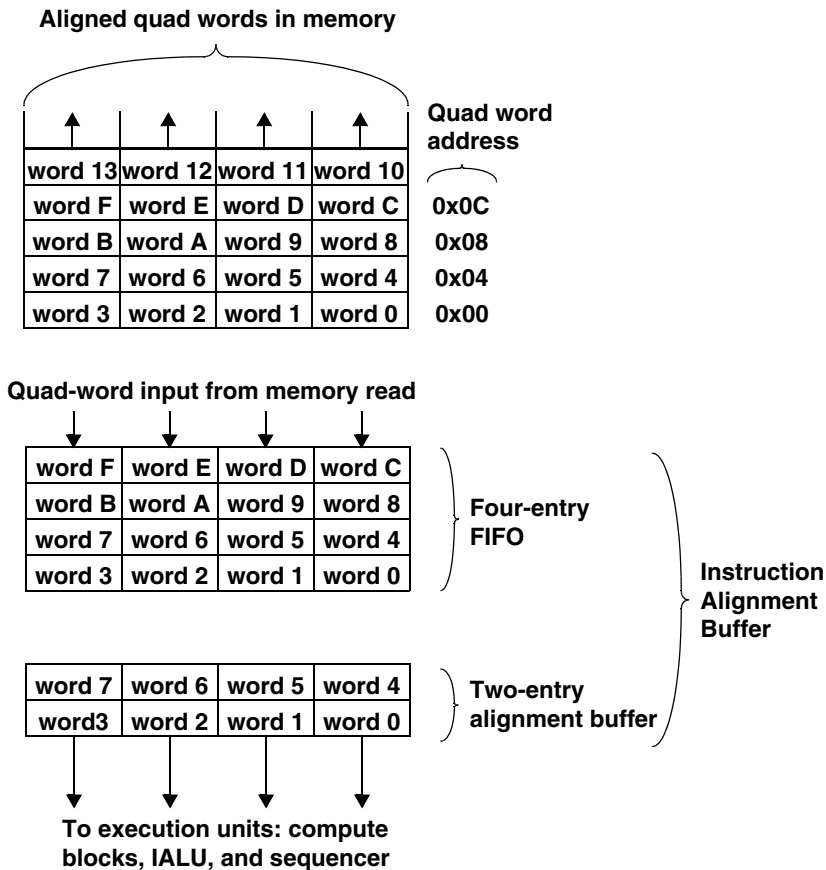


Figure 8-18. Instruction Alignment Buffer (IAB) Structure

Instruction Pipeline Operations

The IAB provides these services:

- Buffers 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.
- Aligns 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 bits) or instruction line length (one to four instructions). There is no wasted memory space. Instructions are executed in parallel as determined by the MSB of the opcode for each instruction, which indicates whether the opcode ends an instruction line.

- Distributes instructions from instruction lines to the execution units—IALUs, compute blocks, and sequencer.

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.



These descriptions of IAB operation apply to internal memory fetches only. Instruction fetches from external memory result in a much slower instruction flow. The sequencer does not issue serialized external fetches, rather the sequencer waits for the indication from the SOC interface that the data from an external fetch is ready before performing the next external fetch. 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 width, and other issues).

To understand the value of the buffer, align, and distribute services that the IAB provides, see [Figure 8-19](#). 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.

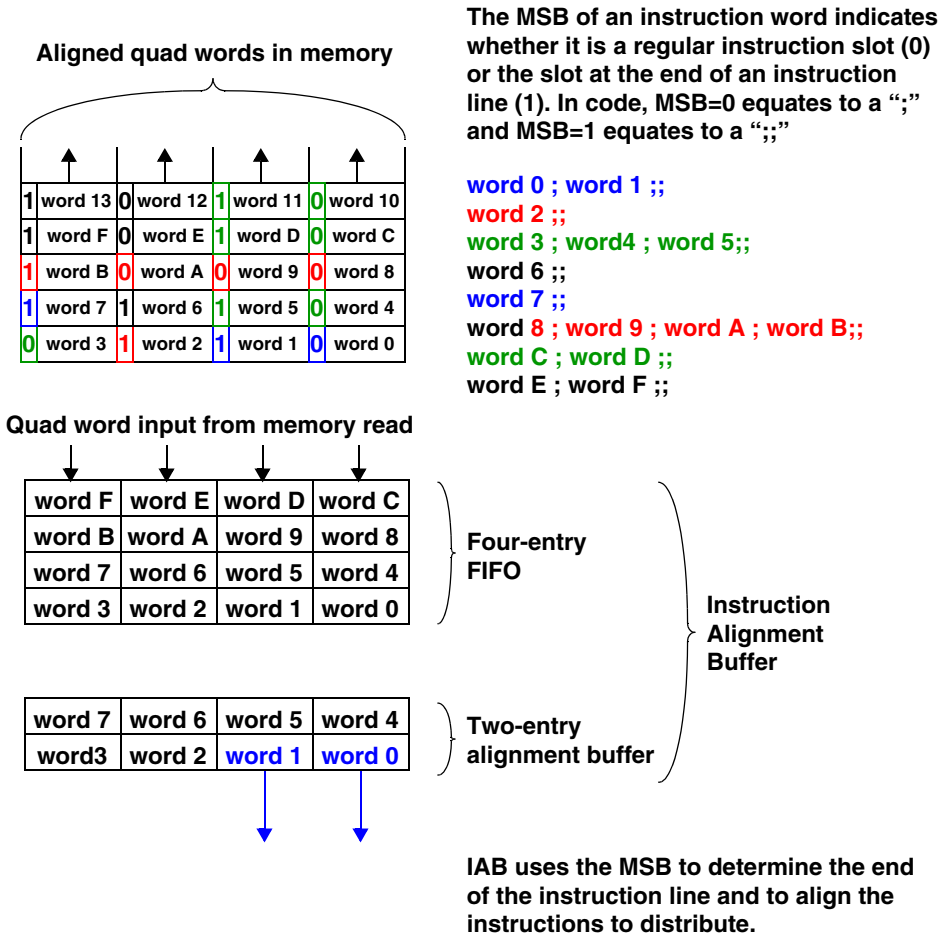


Figure 8-19. IAB Aligns Instruction Lines for Distribution/Execution

Branch Target Buffer (BTB)

The BTB entries make up an associative memory (cache) that records a history of predicted branches—branches predicted as taken. The BTB is active after being enabled with the `BTBEN` instruction. For more information, see “[BTB Enable/Disable](#)” on page 10-245.

BTB entries contain *branch addresses* (address of the instruction ending the line containing the branch instruction) and related *branch data* (target address for the branch). The branch address is stored in the BTB Tag field, and the branch data is stored in the BTB Target field.

The BTB examines the fetched quad word for predicted branch addresses when the sequencer fetches instructions at pipeline stage F1. The BTB issues a *BTB miss* if it encounters a predicted branch address in the quad word that *is not* already recorded in the BTB. The BTB issues a *BTB hit* if it encounters a predicted branch address in the quad word that *is* already recorded in the BTB.

When the BTB recognizes the address of an instruction that previously caused a jump (a BTB hit), the BTB substitutes the corresponding destination address (from the BTB Target field) as the next fetch address for the following instructions. Over time as the program executes, the BTB fills with cached predicted branches. As a result of storing and using this branch information, the pipeline performance loss due to branching is reduced from either ten or five stall cycles to zero.

The following types of branches get special processing by the BTB:

- Indirect jumps (`RTI`, `RETI` and `CJMP`)
- Loop counter condition based jumps and non-jump instructions

For indirect jumps (`RTI`, `RETI`, `CJMP`), the branch data (target address) comes from a register rather than the BTB. The BTB’s `CJMP` and `RTI` bits indicate the target address register (`CJMP` or `RETI`). To get a BTB hit on a predicted indirect jump, the program must make sure that the value

of the register is prepared ahead of time, so at pipeline stage F1 the register's value is correct. Failing to prepare the register in advance causes a branch penalty of up to 10 cycles, but does not cause any execution error.

For loop counter based conditional instructions, the BTB uses a special indication to eliminate a change of flow (BTB miss) in pipeline stage F1 if the loop counter is to be expired. The BTB entry saves information on:

- The loop counter condition: LCOE, LC1E, NLCOE, and NLC1E
- The branch (IF *Cond*, JUMP) or non-branch (IF *Cond*; DO) instruction selection

The BTB checks the loop counter value prediction for these loop counter cases and returns a BTB hit when the following conditions are fulfilled:

- Prediction of loop counter expiration matches evaluation of the loop counter condition (for example, all loop iteration until loop counter expiration for the instruction IF NLCOE, JUMP.)
- The instruction is a branch

The BTB cache contains 128 entries and is a 4-way set associative cache. The replacement mechanism is LRU with lock option. The lock's purpose is to allow critical parts of the program to be held in the BTB without replacement. (See [Figure 8-20](#).)

For permanent buffered program sections, program must lock the BTB using the BTBLOCK and BTBELOCK instructions. Executing the BTBLOCK instruction directs the BTB to lock every new entry that is added or accessed in the BTB. When the BTB entry is locked, it is not replaced

Instruction Pipeline Operations

until the whole BTB is flushed. The BTB stops locking entries when the `BTBLOCK` is executed. The purpose of this operation is to keep performance-critical jumps in BTB.



This lock feature should be used with care because when too many entries in the BTB are locked, other code does not use the BTB.

Note that when locking is disabled, there is still no replacement of previously locked entries

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.

Only internal memory branches are cached in the BTB. The width of the cached target addresses is 21 bits. The BTB stores only one tag entry per aligned quad word of program instructions. As a result, only one branch may be predicted per aligned quad word. If a program requires that 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.

The BTB structure appears in [Figure 8-20](#). 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.

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. The BTB Tag entry records bits 20–7 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. The BTB Target entry records bits 20–0 of the address of the instruction that is the target of the branch.
Valid	The Valid bit indicates whether the entry is valid (=1) or invalid (=0).
CJMP	The CJMP bit indicates whether the branch is a CJMP instruction (=1) or not (=0).
RTI	The RTI bit indicates whether the branch is an RTI instruction (=1) or not (=0).
ABS	The ABS bit indicates whether the branch uses an absolute address (=1) or a PC-relative address (=0).
Lock	The Lock bit indicates whether the entry is locked (=1) or unlocked (=0).
LRU	The LRU is the Least Recently Used field, which determines which way should be overwritten in the event of a BTB miss. Each set has an LRU field that contains the LRU value for all of the ways in the set.

Instruction Pipeline Operations

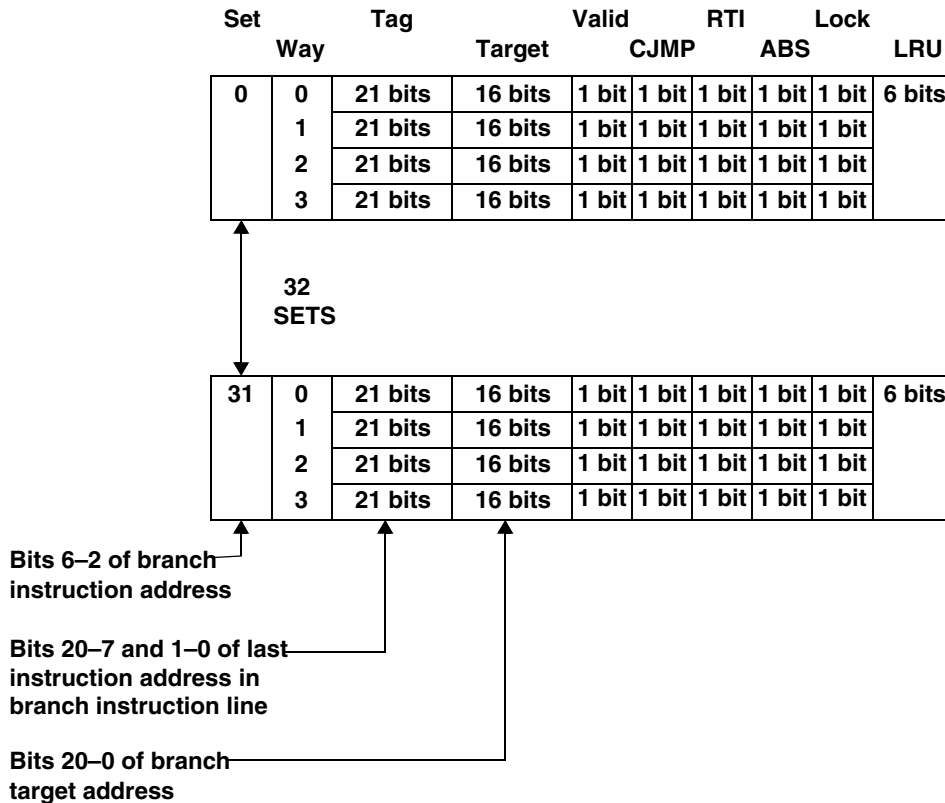


Figure 8-20. Branch Target Buffer (BTB) Structure

Now that the controls for the BTB 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 8-21](#).

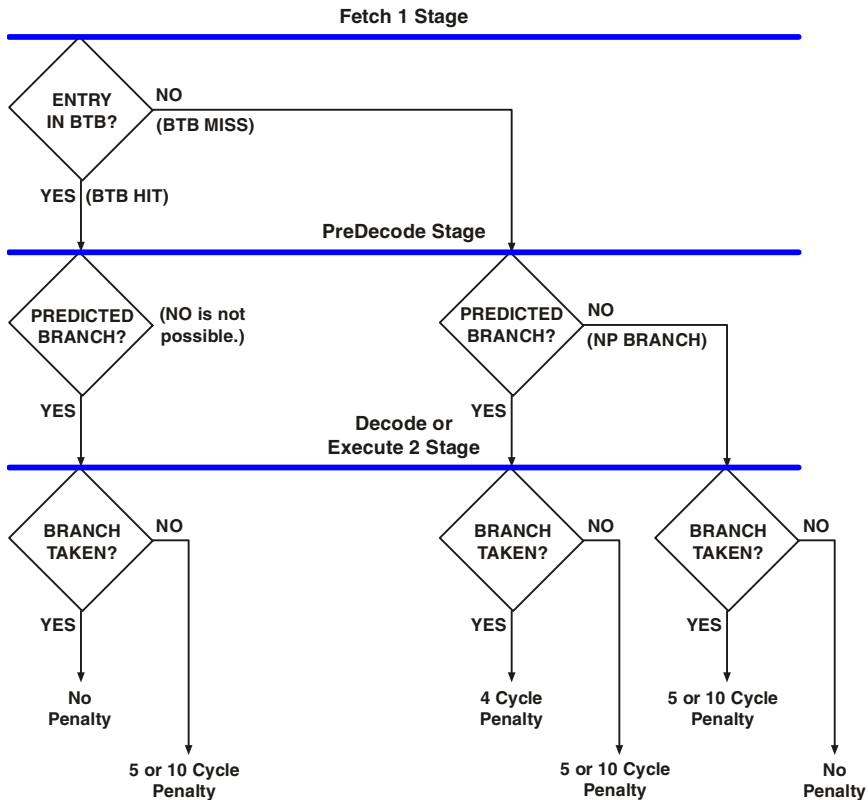



Figure 8-21. Branch Prediction Penalty Tree¹

¹ When the fetched instruction line crosses a quad word boundary, add one penalty cycle in all cases.

As shown in Figure 8-21, BTB usage and branch prediction are independent. The sequencer always compares the fetched instruction address 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 PreDecode 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

Instruction Pipeline Operations


is where the pipeline makes the decision on where to fetch from next. This prediction also affects whether 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) as are unconditional branches (treated as though prefixed with the condition `IF TRUE`). Only conditional branches with the not predicted (`NP`) option are predicted as not taken.

The penalty cycles (stalls) for 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 PreDecode to Decode. If predicted correctly, this case has four 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 five (for IALU condition) or ten (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.

Even assuming that the BTB was disabled, 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 effect 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 21 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, 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. You can also use the `.align_code` assembler directive.

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.

Instruction Pipeline Operations

It is useful to examine how different placement of words in memory results in different contents in the BTB. For example, the code example in [Listing 8-4](#) contains a predicted branch.

Listing 8-4. 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 8-18 on page 8-39](#) and discussed in “[Instruction Alignment Buffer \(IAB\)](#)” on page 8-39. The quad word address is the address of the first instruction in the quad word.

Depending on how the code in [Listing 8-4](#) aligns in memory, quad word address `0x00000004` could contain the quad words shown in [Figure 8-22](#).

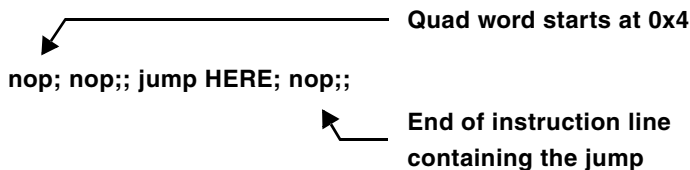


Figure 8-22. Quad Words and Jump Instructions (1)

If so, the BTB entry for the branch would contain:

Tag = `0x00000004`, Target Address = `HERE`

But, the code in [Listing 8-4](#) could align in memory differently. For example, this code could align such that quad word address `0x00000004` (first line) and `0x00000008` (second line) contain the quad words shown in [Figure 8-23](#).

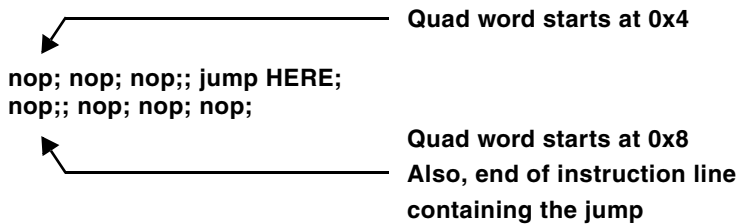


Figure 8-23. Quad Words and Jump Instructions (2)

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 processor does not modify the PC, and the sequencer continues to fetch subsequent quad words at the unmodified PC.

When the same instruction reaches the PreDecode 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 four instructions fetched prior to reaching the Decode stage (four stall cycles), and the processor 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.

Instruction Pipeline Operations

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, it's important to examine coding techniques that work with this BTB feature. The code example in [Figure 8-24](#) 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 8-24. 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;;
```

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 causes the ADSP-TS201 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 8-14.](#))
- BTB hit or miss (See [“Branch Target Buffer \(BTB\)” on page 8-42.](#))
- Condition pipe stage—pipeline stage Integer or Execute 2—when it is resolved (See the Branch Prediction Penalty Tree in [Figure 8-21 on page 8-47.](#))

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 8-42.](#) In

Instruction Pipeline Operations

general, prediction indicates if the default assumption for this branch is taken or is not taken. For example, think about 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 PreDecode, the ADSP-TS201 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 ADSP-TS201 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 8-42](#).

The various condition codes are resolved in different stages. IALU conditions (such as `LC0E`, `LC1E`, `TRUE`, and others) are resolved in stage Decode of the instruction that updates the condition flags. Compute block flags are updated in pipe stage EX2. The other flags (`BM`, `FLAG0_IN`, `FLAG1_IN`, `FLAG2_IN`, `FLAG3_IN`, 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 Decode, 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.

The different flows are shown in [Figure 8-25 on page 8-56](#) through [Figure 8-36 on page 8-85](#). 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 – four cycles
- Prediction taken, branch not taken (either BTB hit or miss); or prediction not taken, branch taken on an IALU condition (five cycles) or compute block condition (ten 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 but was not taken, and the sequential instruction line crosses a quad word boundary.

Instruction Pipeline Operations

Figure 8-25 shows a predicted branch that is based on an IALU condition. The BTB identifies instruction #2 as a predicted jump at the Fetch1 (F1) pipeline stage. 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 Decode (D) pipeline stage.

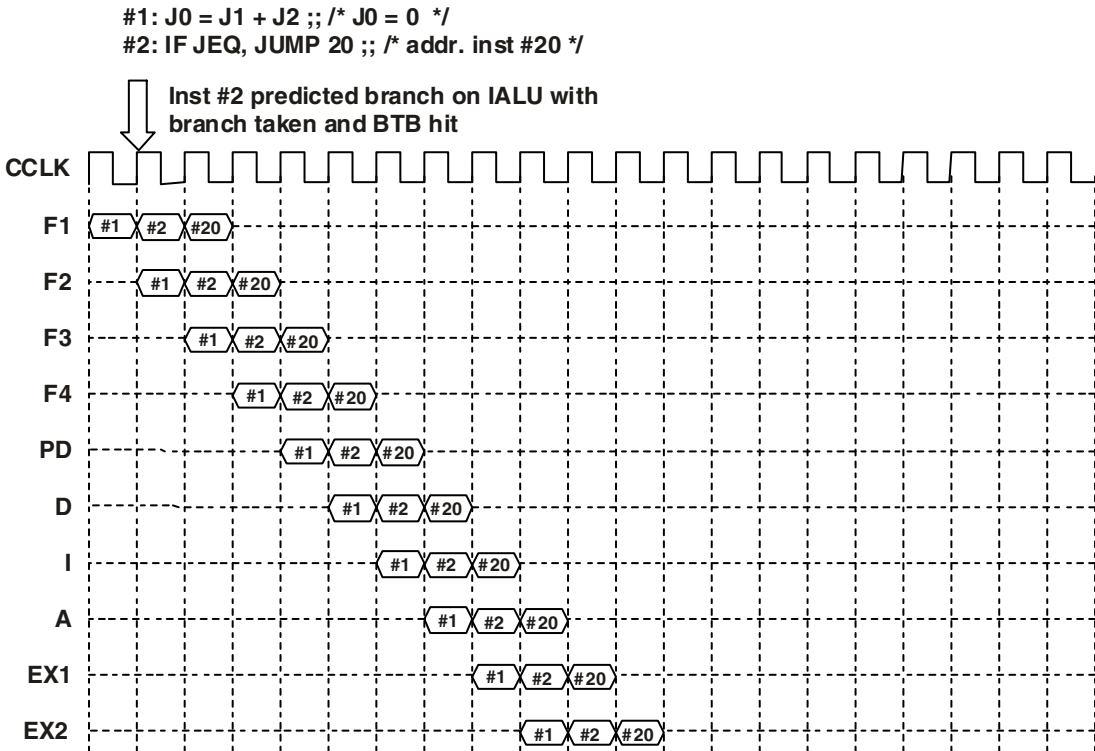


Figure 8-25. Predicted Branch Based on IALU Condition—Branch Taken—With BTB Hit

Figure 8-26 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 (four pipeline stages voided), and there are four lost cycles (branch cost). Note that the pipeline evaluates the IALU condition (JEQ flag set by instruction #1) when the conditional instruction reaches the Decode (D) pipeline stage. From this evaluation, the pipeline determines how many pipeline stages to void.

```
#1: J0 = J1 + J2 ;; /* J0 = 0 */
#2: IF JEQ, JUMP 20 ;; /* addr. inst #20 */
```

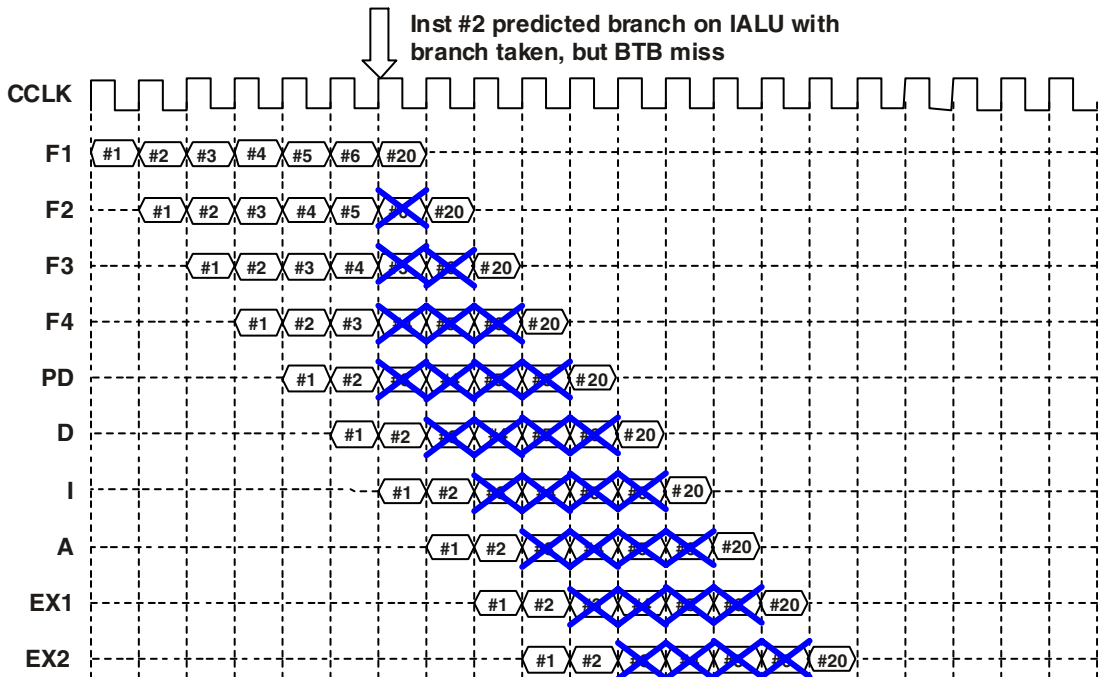


Figure 8-26. Predicted Branch Based on IALU Condition—Branch Taken—With BTB Miss

Instruction Pipeline Operations

Figure 8-27 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 (ten pipeline stages voided), and there are ten 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.

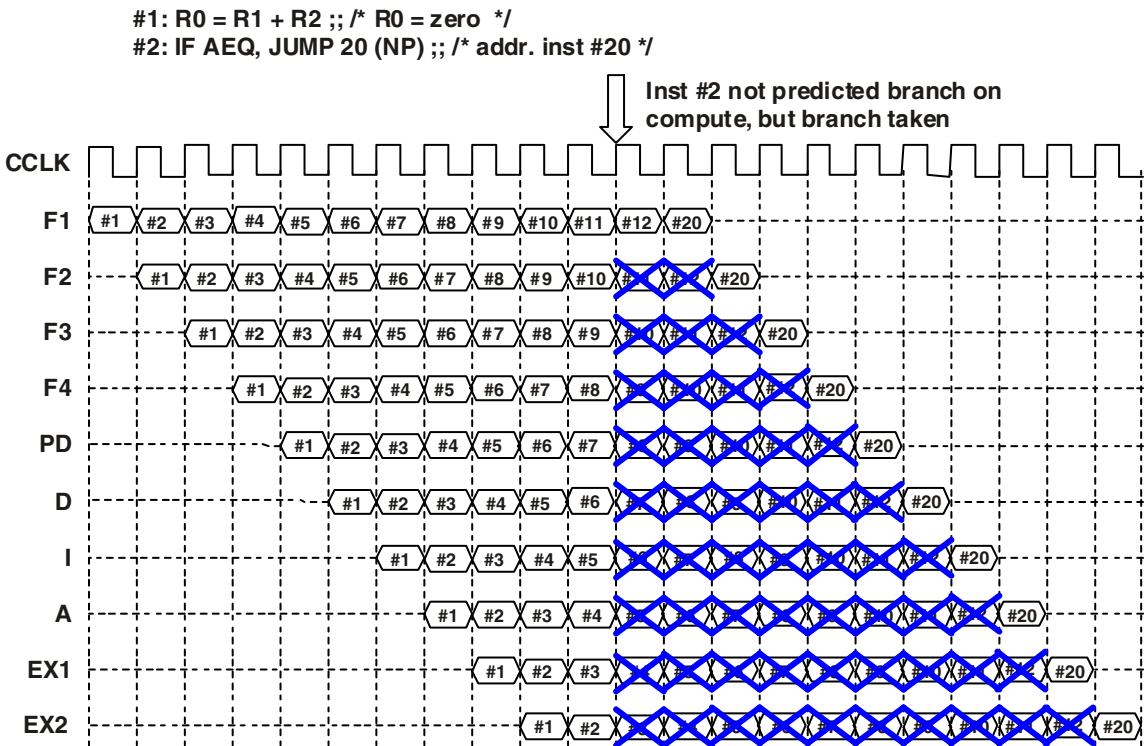


Figure 8-27. Not Predicted (NP) Branch Based on Compute Condition—Branch Taken

Figure 8-28 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 (five pipeline stages voided), and there are five lost cycles (branch cost). Note that the pipeline evaluates the IALU condition (JEQ flag set by instruction #1) when the conditional instruction reaches the Decode (D) pipeline stage. From this evaluation, the pipeline determines how many pipeline stages to void.

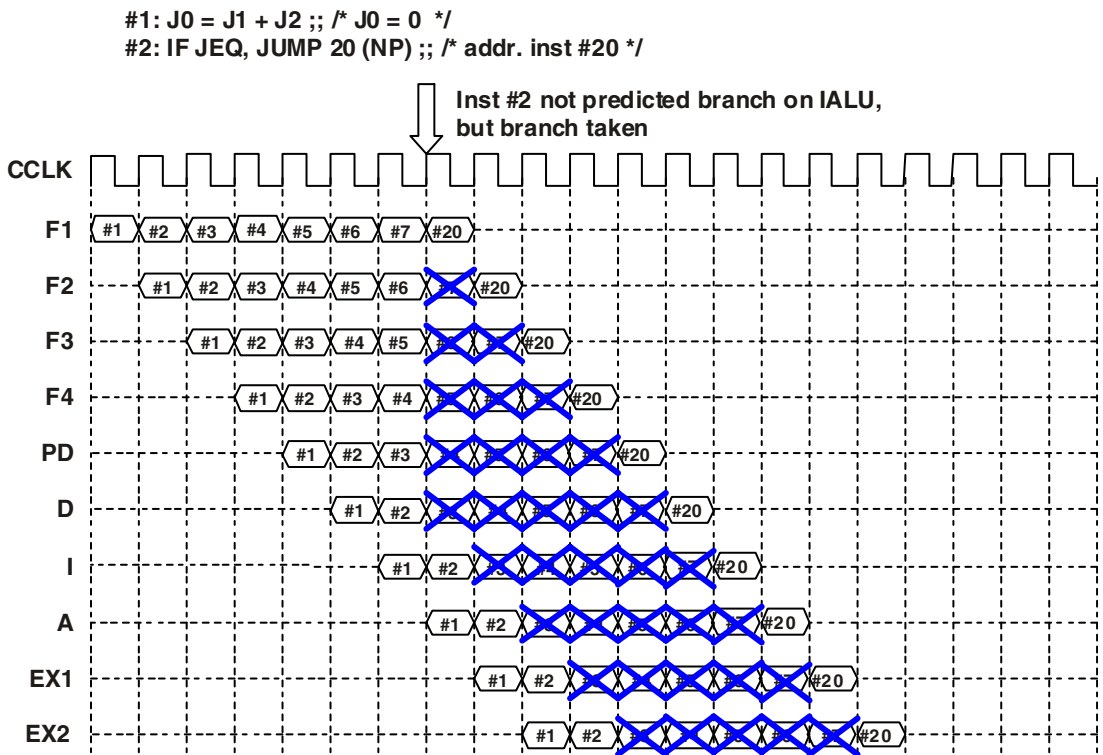


Figure 8-28. Not Predicted (NP) Branch Based on IALU Condition—Branch Taken

Instruction Pipeline Operations

Figure 8-29 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 (ten pipeline stages voided), and there are ten 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.

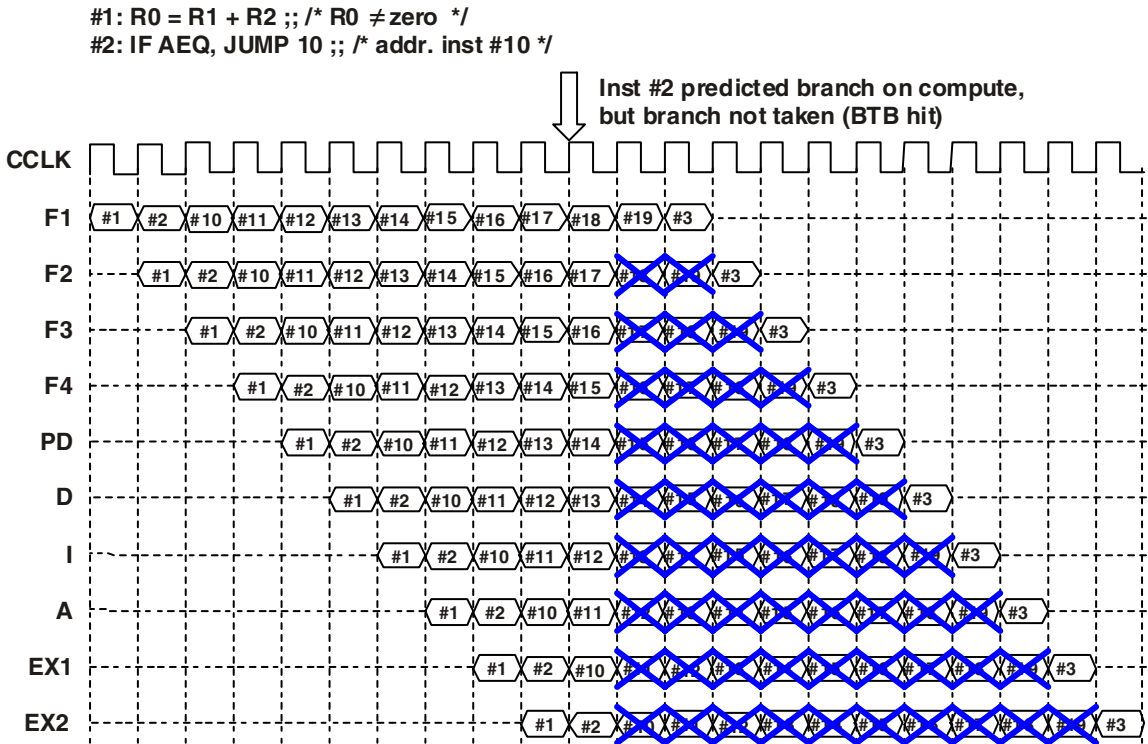


Figure 8-29. Predicted Branch Based on Compute Block Condition—Branch Not Taken—With BTB Hit

Figure 8-30 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 (five pipeline stages voided), and there are five lost cycles (branch cost). Note that the pipeline evaluates the compute condition (JEQ flag set by instruction #1) when the conditional instruction reaches the Decode (D) pipeline stage. From this evaluation, the pipeline determines how many pipeline stages to void.

```
#1: J0 = J1 + J2 ;; /* J0 ≠ 0 */
#2: IF JEQ, JUMP 10 ;; /* addr. inst #10 */
```

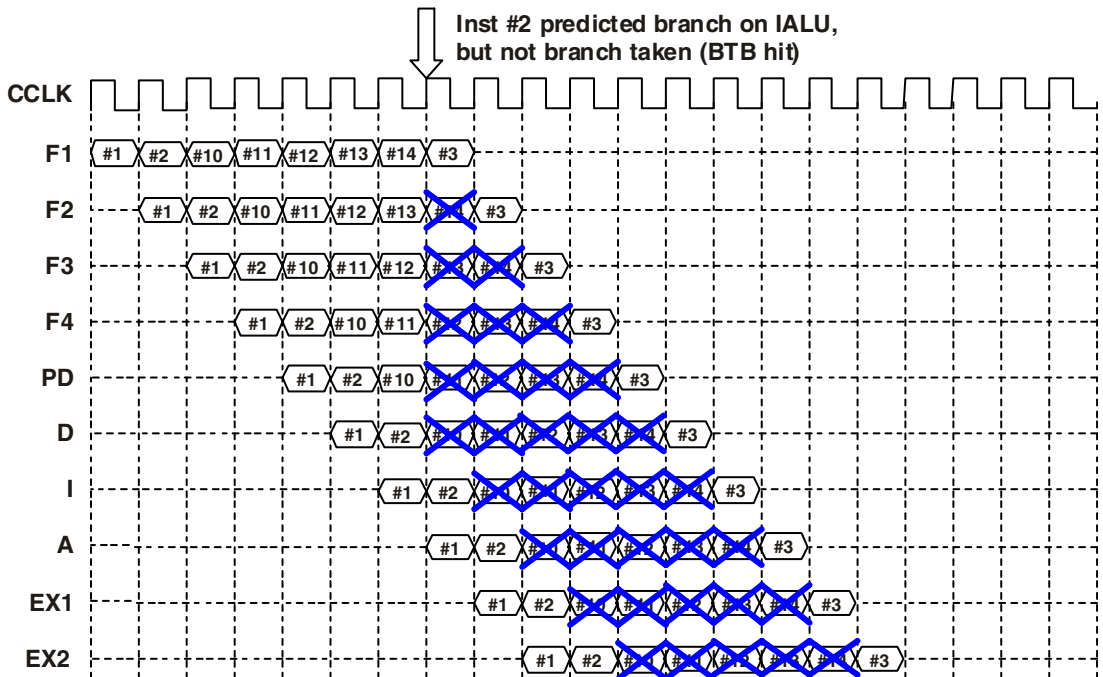


Figure 8-30. Predicted Branch Based on IALU Condition—Branch Not Taken—With BTB Hit

Instruction Pipeline Operations

Figure 8-31 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 (ten pipeline stages voided), and there are ten 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.

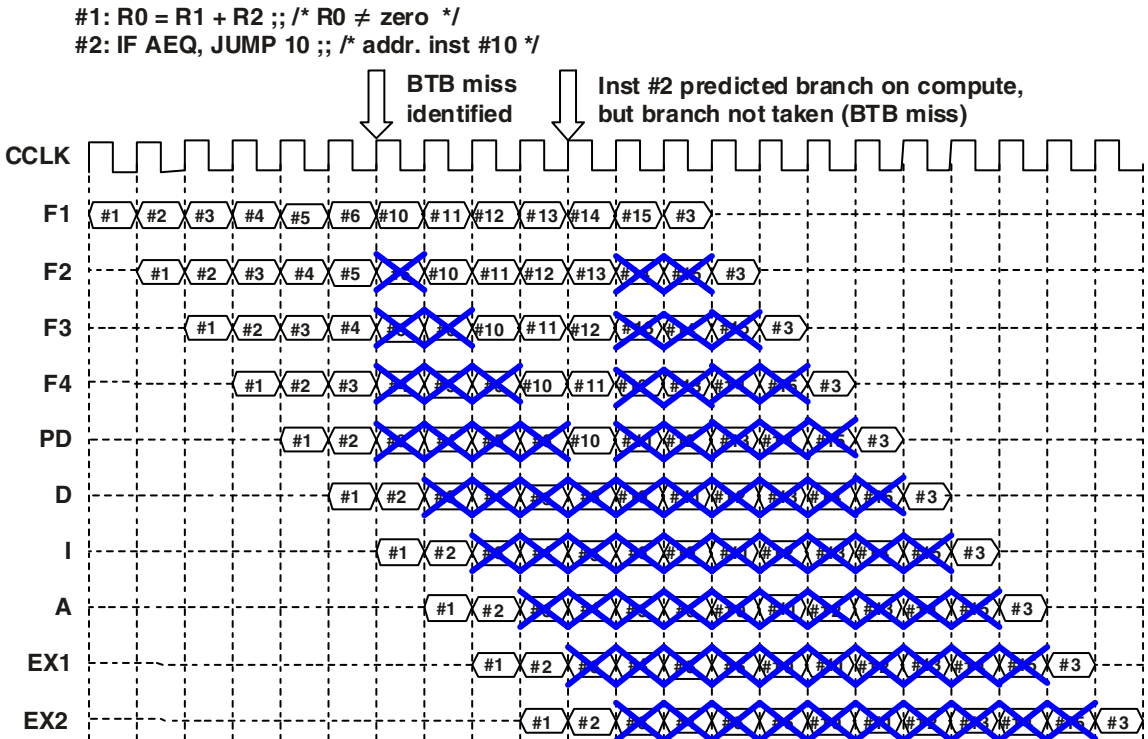


Figure 8-31. Predicted Branch Based on Compute Block Condition—Branch Not Taken—With BTB Miss

Figure 8-32 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 (five pipeline stages voided), and there are five lost cycles (branch cost). Note that the pipeline evaluates the compute condition (JEQ flag set by instruction #1) when the conditional instruction reaches the Decode (D) pipeline stage. From this evaluation, the pipeline determines how many pipeline stages to void.

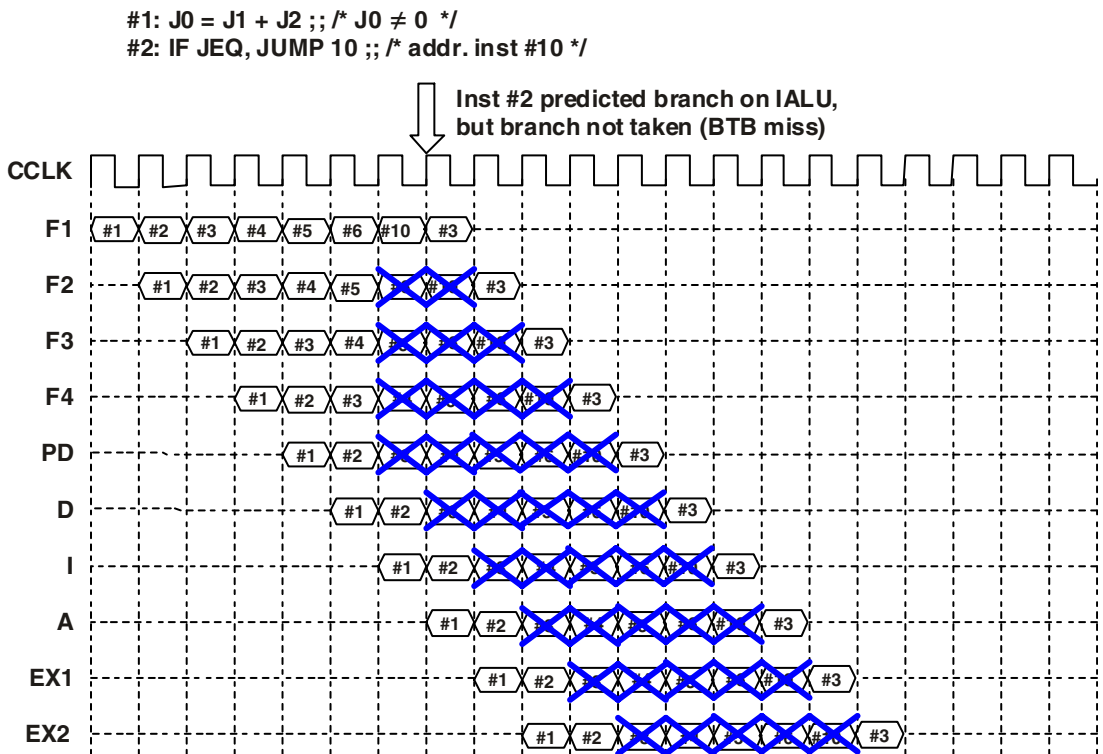


Figure 8-32. Predicted Branch Based on IALU Condition—Branch Not Taken—With BTB Miss

Dependency and Resource Effects on Pipeline

The ADSP-TS201 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.


A *resource conflict* is caused by K-bus contention with external access, conflict on memory block access, or memory busy delay.

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.

As shown in [Figure 8-4 on page 8-6](#), memory accesses use a memory pipeline that operates in parallel with the instruction pipeline. During instruction fetches, bus contention conflicts and cache misses appear as execution delays during the address and register cycles of the I-bus memory pipe. During other memory accesses, bus contention conflicts and cache misses appear as execution delays during the address and register cycles of the J/K-bus memory pipe. For more information on the memory pipeline and cache, see [“Memory and Buses” on page 9-1](#).

This section details the different cases of stalls. A *stall* is any delay in execution of an instruction 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.

 Some architectural differences lead to execution stalls on the ADSP-TS201 processors that did not occur on previous TigerSHARC processors. The ADSP-TS201 processor uses a 10 stage pipeline, which is two stages deeper than previous TigerSHARC processors.

In previous TigerSHARC processors, most of the dependency logic operates in pipeline stage Integer. In the ADSP-TS201 processors, most of the dependency logic operates in pipeline stage Decode (for computes) and PreDecode (for IALU). The stall logic on bus transactions operates in pipeline stage Access.

[Table 8-1](#) lists the dependency related stalls for the ADSP-TS201 processors. Dependency stalls¹ in the execution of instruction line 2 occur when a resource or bus used by instruction line 1 is needed by instruction line 2. Unless noted, the data type or size does not affect dependency related stalls. In the Pipe Stage column, D indicates the Decode stage, A indicates the Access stage, and PD indicates the PreDecode stage.

¹ The information in [Table 8-1](#) describes dependency stall, a delay in beginning execution of the second instruction in a sequence of instructions. You can also view these sequences in terms of *execution latency*, a delay in completion of execution of the first instruction in a sequence of instructions. Both methods of execution analysis are useful.

Instruction Pipeline Operations

Table 8-1. Resource Dependencies Stall List

Instruction 1	Instruction 2	Dependency	Stalls	Pipe Stage
Transfer X/YSTAT instr. Rs1 = XSTAT YSTAT;	Any compute instruction Rs = F (Rm2, Rn2);	Using same compute block	2	D
Transfer X/YSTAT Rs1 = XSTAT YSTAT;	Dreg store/transfer [address] = Rm2; Ureg = Rm2;	Using same Dreg Rs1 = Rm2	1	D
ACS instruction TRsq1 = ACS();	ACS instruction TRsq2 = ACS(TRmd2, TRnd2, Rm);	Using same TRx TRmd2 = TRsq1	0	D
ACS (quad) instruction STRsq1 = ACS();	ACS (quad) instruction STRsq2 = ACS(TRmd2, TRnd2, Rm);	Using same TRx TRmd2 = TRsq1	0	D
Cond. ACS instruction If <cond>; do, {S}TRsq1=ACS();	Any CLU instruction	Using same TRx TRmd2 = TRsq1 TRnd2 = TRsq1	2	D
Cond. Load TRx instruction If <cond>; do, TRsq1=Rm;	The following CLU instr. TRs += DESPREAD(); TRs = XCORRS();	Always	1	D
ACS (quad) instruction STRsq1 = ACS();	ACS instruction TRsq2 = ACS(TRmd2, TRnd2, Rm);	Using same TRx TRmd2 = TRsq1	1	D
ACS instruction TRsq1 = ACS();	ACS (quad) instruction STRsq2 = ACS(TRmd2, TRnd2, Rm);	Using same TRx TRmd2 = TRsq1	1	D
ACS instruction {S}TRsq1 = ACS();	ACS instruction {S}TRsq2 = ACS(TRmd2, TRnd2, Rm);	Using same TRx TRnd2 = TRsq1	1	D

Table 8-1. Resource Dependencies Stall List (Cont'd)

Instruction 1	Instruction 2	Dependency	Stalls	Pipe Stage
ACS instruction TRsq1 = ACS();	MAX/TMAX instruction TRsq2 = F(TRmd2, TRnd2, Rm);	Using same TRx TRmd2 = TRsq1 TRnd2 = TRsq1	1	D
MAX/TMAX instruction TRsd1 = F();	ACS/MAX/TMAX instr. TRsq2 = F(TRmd2, TRnd2, Rm);	Using same TRx TRmd2 = TRsq1 TRnd2 = TRsq1	1	D
ACS/TMAX/MAX instr. TRs1 = F();	The following CLU instr. TRs += DESPREAD(); TRs = XCORRS();	Always ¹	1	D
The following CLU instr. TRs += DESPREAD(); TRs = XCORRS();	ACS/MAX/TMAX instr. TRs1 = F();	Always ¹	2	D
Load TRx/THRx/CMCTL TRs = Rm; THRs = Rm; CMCTL = Rm;	Any CLU instruction	Always	0	D
Cond. load TRx instr. if <cond>; do, TRs1=Rm;	Any CLU instruction Rs TRs = F (TRm2, TRn2);	Using same TRx TRmd2 = TRs1 TRnd2 = TRs1	2	D
Cond. load THRx instr. if <cond>; do THR s1=Rm;	The following CLU instr. TRs += DESPREAD(); TRs = XCORRS();	Always ¹ THRx dependency	2	D
Conditional CLU instr.: ACS/DESPREAD/XCORRS if <cond>; do, TRs = F();	The following CLU instr. TRs = ACS(); TRs += DESPREAD(); TRs = XCORRS();	Always ¹ THRx dependency)	2	D
Cond. load CMCTL instr. if <cond>; do, CMCTL = Rm;	XCORRS instruction TRx:y=XCORRS()(cut R);	Always ¹ CMCTL dependency)	2	D

Instruction Pipeline Operations

Table 8-1. Resource Dependencies Stall List (Cont'd)

Instruction 1	Instruction 2	Dependency	Stalls	Pipe Stage
Any CLU instruction	Transfer TRx/THRx instr. Rs = TRm ; Rs = THRm ;	Always	0	D
SUM with PR accum. instr. PR += SUM () ;;	Conditional instruction based on ALU condition if <ALU_cond> ;;	Always	1	D
SUM instruction Rs = SUM Rm;;	Conditional instruction based on ALU condition if <ALU_cond> ;;	Always	1	D
ABS with PR accum. instr. PR += abs (...) ;;	Conditional instruction based on ALU condition if <ALU_cond> ;;	Always	1	D
Conditional instruction with MR accum. or trans. If cond; do, MRa += ; If cond; do, MRa -= ; If cond; do, MRa = Rm;	Instruction with MR accumulate or transfer MRb += ; MRb -= ; MRb = Rm;	Using same MRx or MR4 MRa = MRb MRa == MR4 MRb == MR4	1	D
Any compute instruction Rs1 = F(Rm, Rn);	Any compute instruction Rs = F(Rm2, Rn2);	Using same Dreg Rs1 = Rm2 Rs1 = Rn2	1	D
Any load/store instruction (memory accesses)	Memory accesses have associated <i>penalty cycles</i> —lost cycles due to cache miss or other memory interface conditions—that may occur in addition to resource dependency stalls. For more information on these penalties, see “ Memory Access Penalty Summary ” on page 9-44.			
Any load Dreg (from internal memory or core) instruction Rs1 = [address] ;; Rs1 = Ureg ;;	Any compute instruction Rs = F(Rm2, Rn2);	Using same Dreg Rs1 = Rm2 Rs1 = Rn2	1+miss ²	D, A

Table 8-1. Resource Dependencies Stall List (Cont'd)

Instruction 1	Instruction 2	Dependency	Stalls	Pipe Stage
Any load Dreg (from external memory or SOC) instruction Rs1 = [address];; Rs1 = SOC_Ureg ;; ³	Any compute instruction Rs2 = F (Rm2, Rn2);	Using same Dreg quad registers⁴ Rs1 = Rm2 Rs1 = Rn2 Rs1 = Rs2	3+SOC delay ⁵	D, A
Load Dreg (from external memory) with DAB instr. Rsq1 = DAB Q [address];;	Any load Dreg with DAB instruction Rsq2 = DAB Q[]	Always	3+SOC delay ⁵	D
Any compute instruction Rs1 = F (Rm, Rn);	Any store/transfer DREG instr. [address] = Rm; Ureg = Rm;	Always	0	
Any load mem. sys. cmd. register instruction CACMDi = <Ureg imm> ;	Any access to memory block "i"	Using same memory block	4	I
Any load mem. sys. register instruction CACMDi = <Ureg imm> ; CCAIRi = <Ureg imm> ; CADATAi = <Ureg imm> ;	Any store mem. sys. register instruction <Ureg> = CACMDi ; <Ureg> = CCAIRi ; <Ureg> = CADATAi ; <Ureg> = CASTATi ;	Using same memory block	4	I
Any access of internal memory that causes a cache miss	Any instruction	Always	Miss penalty	A
Any IALU integer instr. ⁶ Js1 = F (Jm, Jn) ;	Any IALU integer instr. ² Js = F (Jm2, Jn2) ;	Always	0	PD

Instruction Pipeline Operations

Table 8-1. Resource Dependencies Stall List (Cont'd)

Instruction 1	Instruction 2	Dependency	Stalls	Pipe Stage
Any cond. (compute) IALU integer or load/store Ureg instr. ⁶ if <compute_cond.>; do, Js1 = F (Jm, Jn) ; if (compute_cond.); do, Ureg = [Js1 +=] ; if (compute_cond.); do, [Js1 +=] = Ureg ;	Any IALU integer instr. ² Js = F (Jm2, Jn2) ;	Using same Jx Jm2 = Js1 Jn2 = Js1	4	PD
Any cond. (compute) IALU integer instruction ⁶ if <compute_cond.>; do, Js1 = F (Jm, Jn) ;	IALU add/sub with carry/borrow Js = Jm + Jn + Jc; Js = Jm - Jn + Jc - 1;	Using same Jx Jm2 = Js1 Jn2 = Js1	4	PD
Any cond. (compute) IALU integer instruction ⁶ if <compute_cond.>; do, Js1 = F (Jm, Jn) ;	Conditional instruction based on IALU condition if <IALU_cond> ;;	Always	4	PD
Any cond. (non-IALU) ⁷ IALU integer instr. ⁶ if <non_IALU_cond>; do, Js1 = F (Jm, Jn)	Conditional instruction based on IALU condition if <IALU_cond> ;;	Always	1	PD
Any cond. (non-IALU) ⁷ IALU integer or load/store Ureg instruction ⁶ if <non_IALU_cond>; do, Js1 = F (Jm, Jn) if <non_IALU_cond>; do, Ureg = [Js1 +=] if <non_IALU_cond>; do, [Js1 +=] = Ureg	Any IALU integer instruction ² Js = F (Jm2, Jn2);	Using same Jx Jm2 = Js1 Jn2 = Js1	1	PD

Table 8-1. Resource Dependencies Stall List (Cont'd)

Instruction 1	Instruction 2	Dependency	Stalls	Pipe Stage
IALU add/sub with result placed in JB/JL instruction ⁶ JB = Jm + - Jn ; JL = Jm + - Jn ;	IALU add/sub with circular buffer instruction Js = Jm2 + - Jn2 (CB);	Using a JB/JL that belongs to Jm2	5	PD
IALU add/sub with result placed in JB/JL instruction ⁶ JB = Jm + - Jn ; JL = Jm + - Jn ;	Load Dreg using post-modify addressing and circular buffer Dreg = (CB) [Jm2 += Jn2] ;	Using a JB/JL that belongs to Jm2	5	PD
Load/transfer JB/JL instr. ⁶ JB JL = [address] ; JB JL = Ureg ; JB JL = <imm> ;	IALU add/sub with circular buffer instruction Js = Jm2 + - Jn2 (CB);	Using a JB/JL that belongs to Jm2	5	PD
Load/transfer JB/JL instr. ⁶ JB JL = [address] ; JB JL = Ureg ; JB JL = <imm> ;	Load Dreg using post-modify addressing and circular buffer Dreg = (CB) [Jm2 += Jn2] ;	Using a JB/JL that belongs to Jm2	5	PD
Load/transfer IALU reg. instr. ⁶ Js1 = [address] ; Js1 = Ureg ; Js1 = <imm> ;	Any IALU integer instruction ² Js2 = F (Jm2, Jn2);	Using same Jx Jm2 = Js1 Jn2 = Js1	4 + miss ²	PD, A
Load/transfer J31 instr. ⁶ J31 = [address] ; J31 = Ureg ; J31 = <imm> ;	IALU add/sub with carry/borrow Js = Jm + Jn + Jc; Js = Jm - Jn + Jc - 1;	Always	5 + miss ²	PD, A
Load/transfer J31 instr. ⁶ J31 = [address] ; J31 = Ureg ; J31 = <imm> ;	Conditional instruction based on J-IALU condition if <J-IALU_cond>, jump ; if <J-IALU_cond>; do ;	Always	5 + miss ²	PD, A

Instruction Pipeline Operations

Table 8-1. Resource Dependencies Stall List (Cont'd)

Instruction 1	Instruction 2	Dependency	Stalls	Pipe Stage
Load/transfer J31 from external memory or SOC instruction J31 = [ext_memory] ; J31 = SOC_Ureg ;	IALU add/sub with carry/borrow Js = Jm + Jn + Jc ; Js = Jm - Jn + Jc - 1 ;	Always	3 + SOC delay	PD
Load/transfer J31 from external memory or SOC instruction J31 = [ext_memory] ; J31 = SOC_Ureg ;	Conditional instruction based on J-IALU condition if <J-IALU_cond>, jump ; if <J-IALU_cond>; do ;	Always	3 + SOC delay	PD
Load/transfer Jx from external memory or SOC instruction Js1 = [ext_memory] ; Js1 = SOC_Ureg ;	Any IALU integer instruction ² Js2 = F (Jm2, Jn2);	Using same Jx quad registers ⁸ Js1 = Jm2 Js1 = Jn2 Js1 = Js2	3+SOC delay ⁵	PD
Load/transfer JB/JL from external memory or SOC instr. ⁶ JB JL = [ext_memory] ; JB JL = SOC_Ureg ;	Any IALU instruction using circular-buffer registers	Using a JB/JL that belongs to Jm2	3+SOC delay	PD
Load/transfer J31 from external memory or SOC instruction J31 = [ext_memory] ; J31 = SOC_Ureg ;	Any IALU integer instruction ² Js2 = F (Jm2, Jn2);	Always	3+SOC delay	D
Transfer external port from Ureg <ep_register> = Ureg;;	Transfer Ureg from external port Ureg = ep_register ;	This sequence of instructions may produce a bus delay (stall). For more information, see the <i>ADSP-TS201 TigerSHARC Processor Hardware Reference</i> , Chapter 5, "Cluster Bus".		
Load/transfer SFREG instr. SFREG = [address] ; SFREG = Ureg ; SFREG = <imm> ;	Conditional instruction based on ISFx (global) static flag cond. if ISFx, jump ; if ISFx; do ;	Always	5+miss	PD

Table 8-1. Resource Dependencies Stall List (Cont'd)

Instruction 1	Instruction 2	Dependency	Stalls	Pipe Stage
Load/transfer SFREG instr. SFREG = [address] ; SFREG = Ureg ; SFREG = <imm> ;	Load a non-IALU ⁷ condition into an IALU (global) static flag instr. ISFx = += <non_IALU_cond>;	Always	4+miss	PD
Load/transfer SFREG instr. SFREG = [address] ; SFREG = Ureg ; SFREG = <imm> ;	Load an compute condition into an IALU (global) static flag instr. ISFx = += <compute_cond>;	Always	0	PD
Load/transfer SFREG instr. SFREG = [address] ; SFREG = Ureg ; SFREG = <imm> ;	Load a compute condition into a static flag (one compute block) XSFx = += <compute_cond>; YSFx = += <compute_cond>;	Always	0+miss	PD
Any compute or transfer X/YSTAT instruction Rs = F (Rm2, Rn2); Rs1 = XSTAT YSTAT;	Load a compute condition into an IALU (global) static flag instr. ISFx = += <compute_cond>;	Using a cond. updated by comp. instr. or trans. from X/YSTAT	1	PD
Load a compute or non-IALU ⁷ condition into a static flag instr. SFx = += <compute_cond>; ISFx = += <non_IALU_cond>;	Store/transfer SFREG instr. [address] = SFREG ; Ureg = SFREG ;	Always	1	PD
Load a compute condition into an IALU (global) static flag instr. ISFx = += <compute_cond>;	Load a non-IALU ⁷ condition into an IALU (global) static flag instr. ISFx = += <non_IALU_cond>;	Using same ISFx static flag	4	PD
Load a compute condition into an IALU (global) static flag instr. ISFx = += <compute_cond>;	Cond. instr. based on IALU static flag cond. or load an IALU static cond. into a static flag instr. if <ISFx>, jump ; if <ISFx>; do ; ISFx = += ISFx;	Using same ISFx static flag (not updated)	5	PD

Instruction Pipeline Operations

Table 8-1. Resource Dependencies Stall List (Cont'd)

Instruction 1	Instruction 2	Dependency	Stalls	Pipe Stage
Any multiplier compute or load X/YSTAT instruction Rs = F (Rm2, Rn2); XSTAT YSTAT = Rsl;	Cond. instr. based on multiplier flag cond. or load a mult. cond. into a static flag instr. if <mult_cond>, jump ; if <mult_cond>; do ; SFx = += <mult_cond>;	Always	1	D
Load/transfer SFREG instr. SFREG = [address] ; SFREG = Ureg ; SFREG = <imm> ;	Cond. instr. based on compute static flag cond. or load a comp. static flag cond. into an IALU (global) static flag if XSFx YSFx, jump ; if XSFx YSFx; do ; ISFx = += XSFx YSFx;	Always	1 +miss	PD
Load a compute condition into a static flag (one block) instruction XSFx = += <compute_cond>; YSFx = += <compute_cond>;	Cond. instr. based on compute static flag cond. or load a comp. condition into an IALU (global) static flag if SFx, jump ; if SFx; do ; ISFx = += XSFx YSFx;	Using same XSFx or YSFx static flag	1	PD
Load a loop counter instruction LCx = [address] ; LCx = Ureg ;	Conditional instruction based on loop counter condition if LCxE, jump ;	Using same LCx	5+miss	PD
Conditional instruction based on loop counter condition if LCxE, jump ;	Store a loop counter instruction [address] = LCx ; Ureg = LC0 1;	Using same LCx	1	PD
Any BTB control instruction BTBEN ; BTBDIS ; BTBLOCK ; BTBELOCK ; BTBINV ;	Any instruction	Always	10	PD

Table 8-1. Resource Dependencies Stall List (Cont'd)

Instruction 1	Instruction 2	Dependency	Stalls	Pipe Stage
Load/transfer SQCTL instr. SQCTL = [address] ; SQCTL = Ureg ; SQCTL = <imm> ;	Any instruction	Always	10	PD
Load/transfer and debug register instruction ⁹ <debug_reg> = [address] ; <debug_reg> = Ureg ; <debug_reg> = <imm> ;	Any instruction	Always	10	PD
Any load Ureg (from external memory or SOC) instruction Ureg = [address];; Ureg = SOC_Ureg ;;	Conditional return from interrupt instruction If <cond> RTI;	If the return is from a level triggered interrupt ¹⁰	External access delay	D
Load EMUCTL from Ureg or immediate instruction EMUCTL = <imm>;; EMUCTL = Ureg ;;	EMUCTL write done		Write delay	PD
Store SQSTAT instruction [address] = SQSTAT ; Ureg = SQSTAT ;	Any instruction	Always	2	PD
Load FLAGREGST/CL instr. FLAGREGST CL = [address]; FLAGREGST/CL = Ureg; FLAGREGST/CL = <imm>;	Store FLAGREG instruction [address] = FLAGREG; Ureg = FLAGREG;	Always	1	PD

Instruction Pipeline Operations

Table 8-1. Resource Dependencies Stall List (Cont'd)

Instruction 1	Instruction 2	Dependency	Stalls	Pipe Stage
Load CJMP or conditional branch instruction CJMP = [address] ; CJMP = Ureg ; CJMP = <imm> ; If <cond>, CALL ; If <cond>, CJMP_CALL ;	Conditional branch instruction If <cond>, CALL ; If <cond>, CJMP_CALL ;	Always	5+miss	PD
Load RETI or RETIB instruction (or entering an interrupt service routine) RETI = [address] ; RETI = Ureg ; RETI = <imm> ; RETIB = [address] ; RETIB = Ureg ; RETIB = <imm> ;	Conditional return from interrupt instruction If <cond> RTI;	Always	5+miss	PD

- 1 The stall occurs whether or not there is a data dependency.
- 2 Miss penalty by cache is added to the stall at Access stage. If there is a hit, the dependency penalty is one cycle for compute load dependency and four cycles for IALU load dependency.
- 3 SOC_Ureg is a Ureg in groups 0x20 to 0x3F - any Ureg in the SOC bus modules.
- 4 Rm2 resides in Rs1's quad register, Rn2 resides in Rs1's quad register, or Rs2 resides in Rs1's quad-register. For example, R0 and R3 reside in the same quad register R3:0.
- 5 SOC delay is the delay of the access to the external world.
- 6 All the IALU rules refer to K-IALU as well. The example is always given for J-IALU.
- 7 The non-IALU conditions (non_IALU_cond) are: LCx, ISFx, TRUE, and FLAGx_IN.
- 8 Jm2 resides in Js1's quad register, Jn2 resides in Js1's quad register, or Js2 resides in Js1's quad register. For example, J0 and J3 reside in the same quad register J3:0.
- 9 The debug registers are: WPxCTL, WPxSTAT, WPxL, WPxH, CCNTx, PRFM, PRFCNT, TRCBMASK, TRCBPTR, TRCB31-0, and TRCBVAL.
- 10 The purpose of the stall in a level interrupt is to prevent a situation where the interrupt indication that is cleared by the read (for example, read the link receive buffer that clears the buffer) is not yet cleared when exiting the interrupt routine, and this may cause a false new interrupt indication.

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 pipeline 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 pipeline 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.

Instruction Pipeline Operations

Figure 8-33 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 Decode stage (D) when the register dependency is recognized. When #2 reaches Decode, execution of instruction #2 is stalled for one cycle.

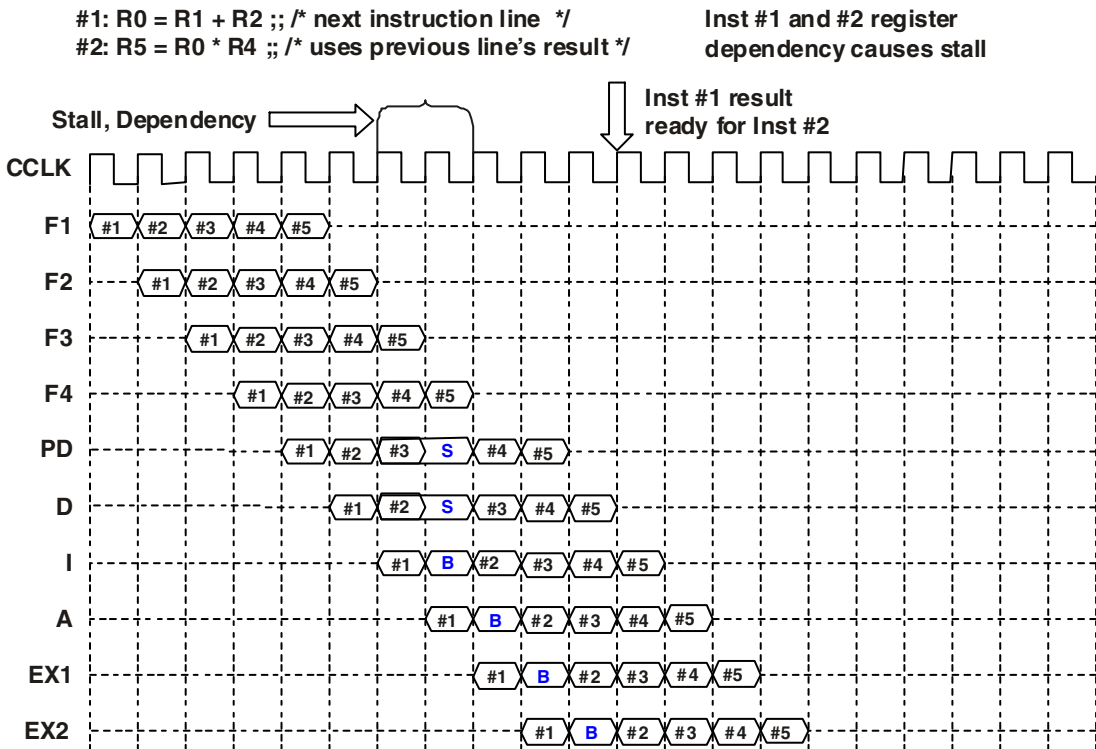


Figure 8-33. Compute Block Operations—
 Result Dependency Stalls (S) and Bubbles (B) Following Instruction

As with other compute block 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 special 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 the `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 as `ACS` instruction as `TRmd` with no stall. For example, the following sequence causes 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);;
```

Instruction Pipeline Operations

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);;
```

An ACS instruction with 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

Unlike previous TigerSHARC processors, the buses on the ADSP-TS201 processor are connected to specific bus masters (J-IALU to the J-bus, K-IALU to the K-bus, and SOC interface to the S-bus) and can access any block of memory. On previous TigerSHARC processors, the buses were connected to specific memory blocks and bus masters requested bus access. This architectural difference provides more control for eliminating bus stalls relating to bus conflicts on the ADSP-TS201 processor. To eliminate potential bus conflicts and related stalls, note that the execution of these instructions uses the following memory buses:

- $Ureg = [Jm + Jn | imm]$, $Ureg = [Km + Kn | imm]$
For all data types and options, the first $Ureg =$ uses J-bus, and second $Ureg =$ uses K-bus.
- $[Jm + Jn | imm] = Ureg$, $[Km + Kn | imm] = Ureg$
For all data types and options, the first $= Ureg$ uses J-bus, and second $= Ureg$ uses K-bus.

- $Ureg = Ureg$
 $Ureg = imm$
These use the J-bus even if both $Uregs$ are in the same register file.
- $Js = Jm + | - Jn$ (CJ)
This uses the J-bus.
- $Ks = Km + | - Kn$ (CJ)
This uses the K-bus.

Arbitration occurs on the J-bus and K-bus when the SOC interface attempts to access a processor core internal register. The SOC interface uses the K-bus to execute this transaction. The arbitration occurs when data is returned for a previous read of external memory or SOC register or when an external master (host or another TigerSHARC) tries to access a processor core register using multiprocessing space. The arbitration between the masters on the bus is detailed in “Bus Arbitration Protocol” in the *ADSP-TS201 TigerSHARC Processor Hardware Reference*. When the core tries to use the J-bus or K-bus and loses on arbitration, stall cycles are inserted.

The IALU always requests the memory block on pipe stage Integer. If the IALU does not receive access to the memory block, the execution of the bus transaction is delayed until the memory block is granted. The rest of the line is continued, including the other IALU operations (for example, post-modify of address). This is to prevent deadlock in case of two memory accesses in the same cycle to the same memory block. 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).

Instruction Pipeline Operations

Figure 8-34 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 J-bus and J-IALU bus master. Execution of instruction #1 is extended over three cycles—two for the stall and one for the access.

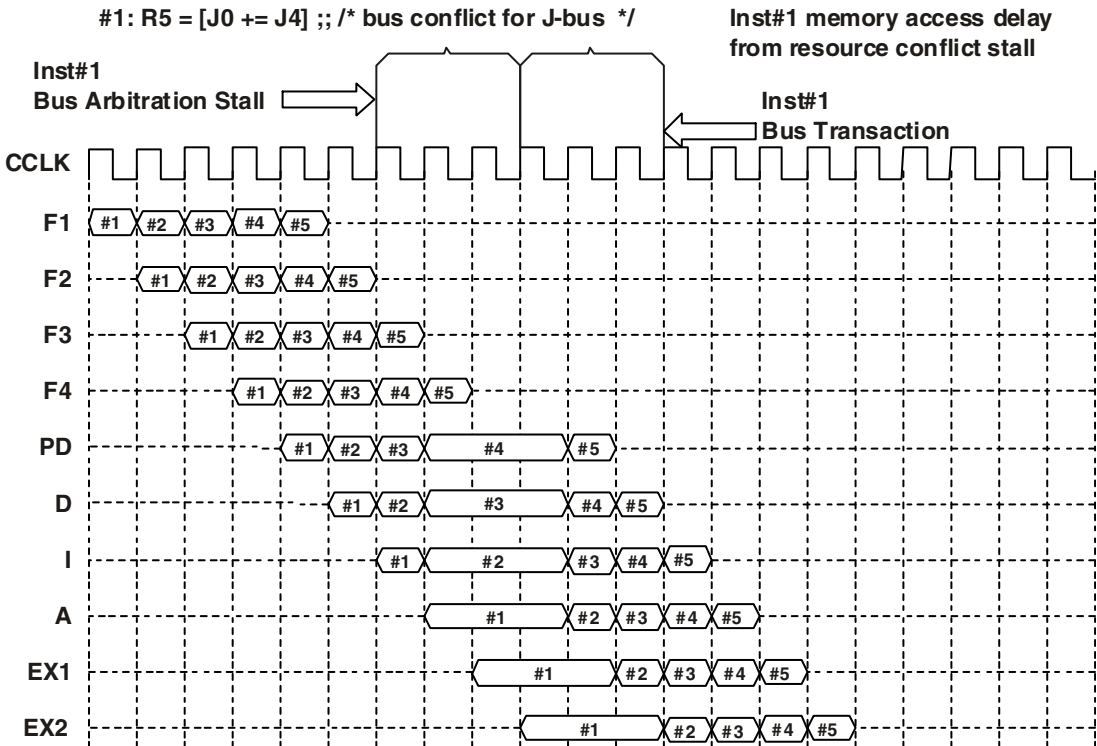


Figure 8-34. Register Load Using Post-Modify With Update—Resource Conflict Stalls Bus Access

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 (Figure 8-35).

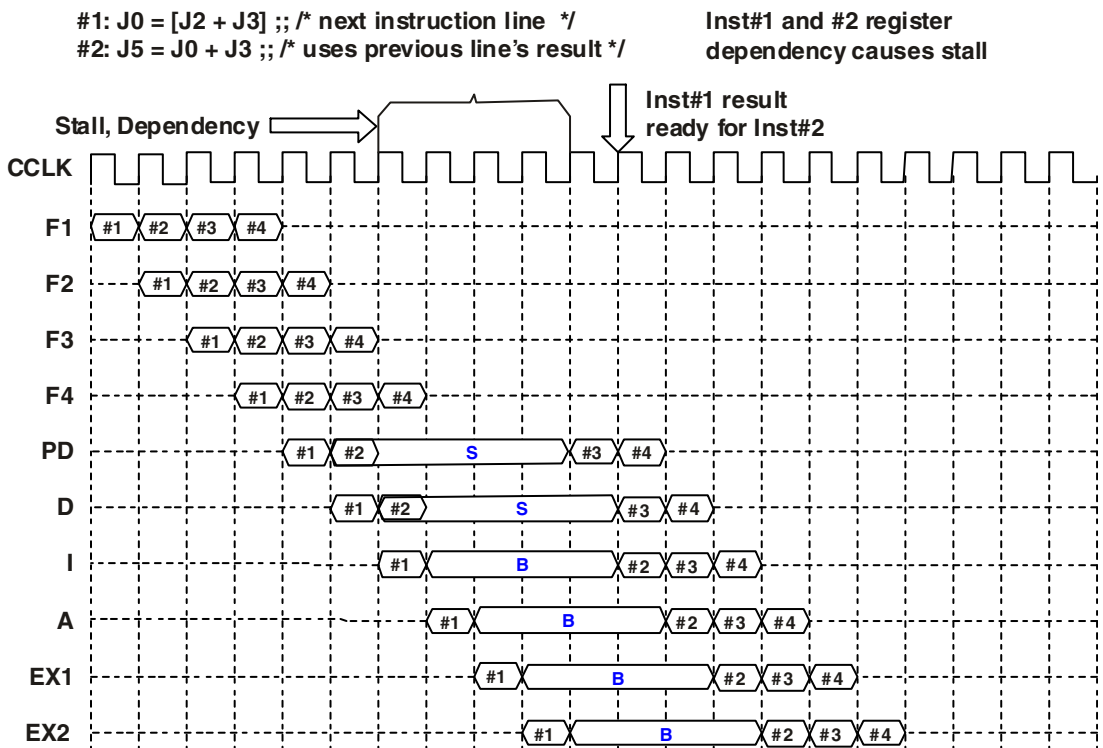


Figure 8-35. IALU Operations—Result Dependency Stalls (S) and Bubbles (B) Following Instruction

Instruction Pipeline Operations

For example, the following sequence:

```
XR0 = [J31 + memory_access] ;;  
XR5 = R0 * R4 ;; /* One cycle Stall */
```

This causes a one cycle delay, occurring when the load instruction comes from internal memory and the bus is accepted by the IALU that executed the transaction.

If the load is from external memory or the bus request is 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 PreDecode. To overcome this gap, four stalls are inserted before the instruction that is using the loaded data, as shown in [Figure 8-36](#).

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 8-36](#).

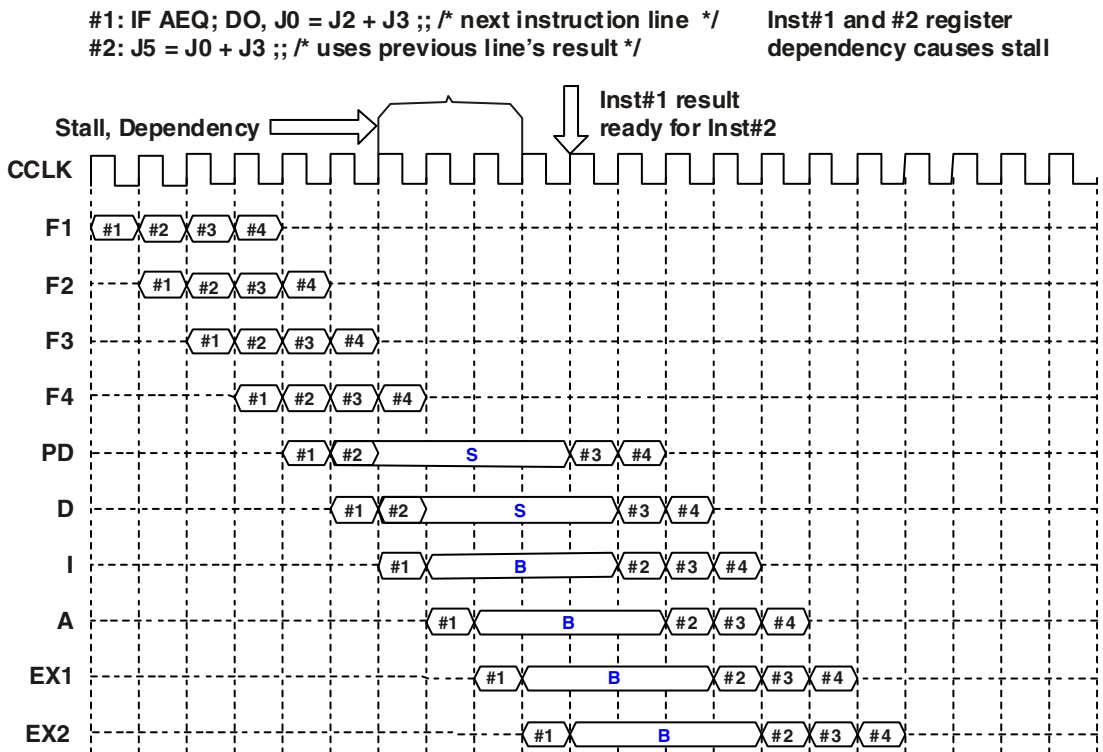


Figure 8-36. IALU Conditional Operations—
 Result Dependency Stalls (S) and Bubbles (B) Following Instruction

Instruction Pipeline Operations

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.

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-TS201 TigerSHARC Processor Hardware Reference*.

The interrupts in some applications are performance critical, and the ADSP-TS201 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 8-37](#). 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 ADSP-TS201 processor starts fetching from the interrupt routine address. The execution of the instructions 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.

#1: /* any unconditional, non-branching instruction */

#2: /* any unconditional, non-branching instruction */

#1i: /* interrupt service routine instruction #1 */

#2i: /* interrupt service routine instruction #2 */

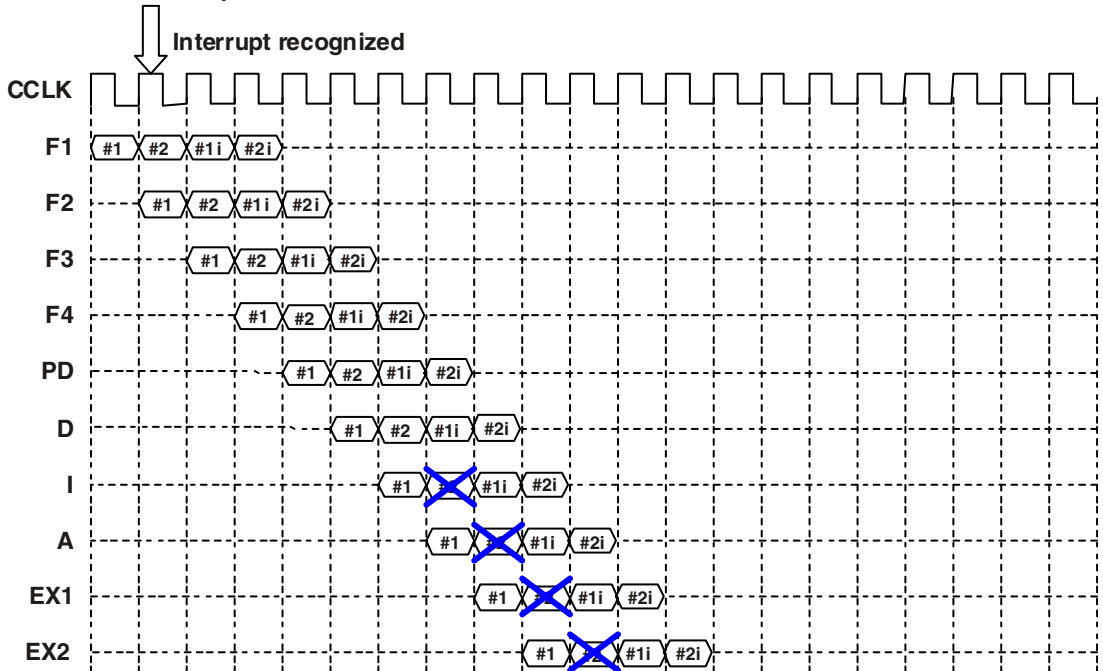


Figure 8-37. Interrupting Linear Execution (Interrupts Enabled, Interrupting Unconditional, Non-Branching Instructions)

Interrupt During Conditional Instruction

When a predicted branch or a not predicted branch instruction is fetched, the ADSP-TS201 processor cannot decide immediately if the branch is to be taken (as discussed in [“Branch Target Buffer \(BTB\)” on page 8-42](#)). 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 8-38](#). 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 8-39 on page 8-91](#)).

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 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.

#1: XR0 = R2 - R1 ;; /* updates flags for Inst#2 */
 #2: IF NXAEQ, JUMP 100 ;; /* conditional on result from Inst#1

#1i: /* interrupt service routine instruction #1 */
 #2i: /* interrupt service routine instruction #2 */

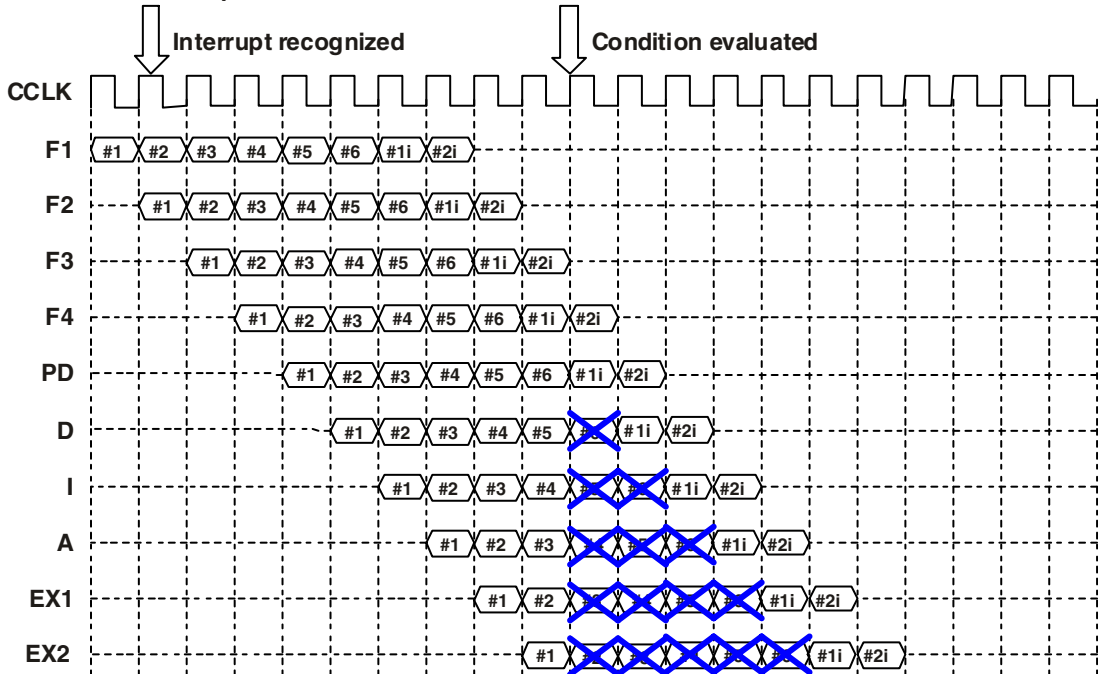


Figure 8-38. Interrupting Conditional Execution (Interrupts Enabled, Interrupting Conditional Instructions)

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 the `SQCTL` register 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 continues execution until reaching EX1, and only then is it aborted and the flow returns to normal.

An example of this flow is shown in [Figure 8-39](#). The interrupt is identified by the ADSP-TS201 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.

Exception Effects on Pipeline

An exception is normally caused by using a specific instruction line. Some sources of exceptions are based on compute block status (overflow, underflow, and invalid floating-point operation), while others are based on non-compute status (illegal memory access, emulator operations, or others). For more information on exception causes, see the `EXCAUSE` field in the `SQSTAT` register ([Figure 8-6 on page 8-11](#)). The exception handling 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 ADSP-TS201 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 8-27 on page 8-58](#)).

```

#_ : XR0 = SQCTL ;; /* load XR0 with current interrupt mask */
#_ : XR0 = BCLR R0 BY INT_GIE;;
    /* where INT_GIE = 0xFFFFF7B from defts201.h file;
       this clears the GIE bit, but retains other bit values */
#1: SQCTL = XR0 ;; /* load SQCTL with data globally disabling interrupts */
#2: /* any unconditional instruction */

```

```

#1i: /* interrupt service routine instruction #1 */
#2i: /* interrupt service routine instruction #2 */

```

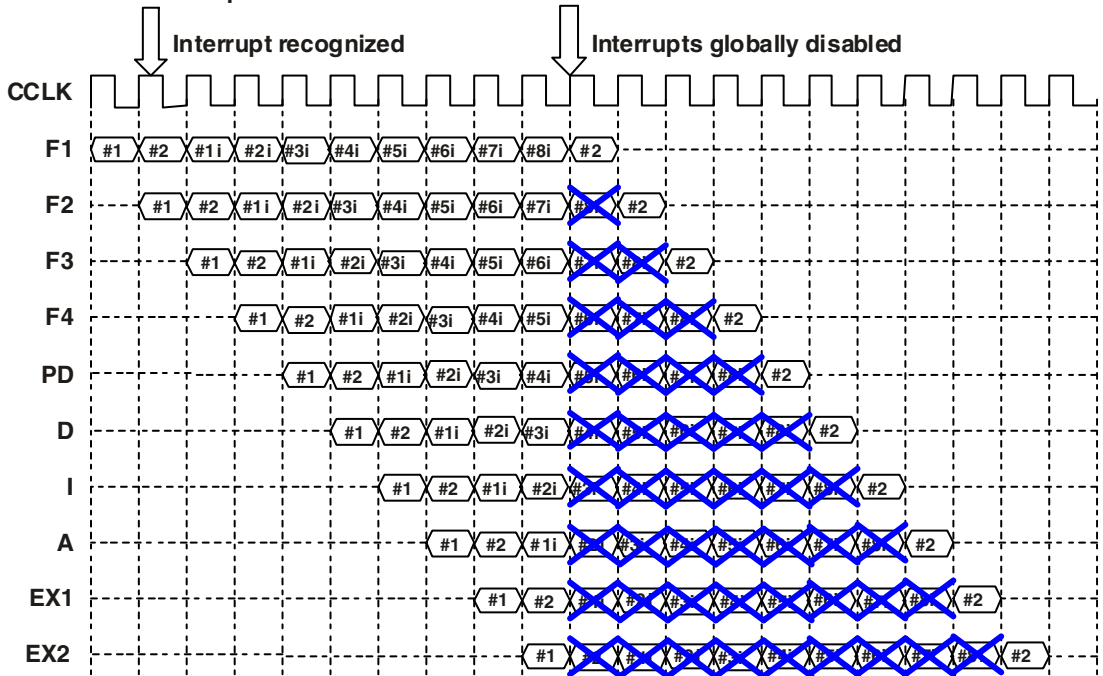


Figure 8-39. Interrupting Linear Execution (Interrupts Enabled, Interrupting Unconditional, Non-Branching Instructions)—Interrupt Disabled While In Pipeline

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 8-5. 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 8-6. 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 8-7. 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 8-8. Zero-Overhead and Near-Zero-Overhead Loops Example

```
J6 = J31 + 10;; /* load counter for _outer_loop */
.align_code 4;
_outer_loop:
    NOP ;; /* enter _outer_loop */
    NOP ;;
    NOP ;;
    NOP ;;
    NOP ;;
    LC1 = 5;; /* load counter for _middle_loop */
.align_code 4;
_middle_loop:
    NOP ;; /* enter _middle_loop */
```

Sequencer Examples

```
    NOP ;;
    NOP ;;
    NOP ;;
    NOP ;;
    LCO = 6;; /* load counter for _inner_loop */
.align_code 4;
_inner_loop:
    NOP ;; /* enter _inner_loop */
    NOP ;;
    NOP ;;
    NOP ;;
    NOP ;;
    NOP ; NOP ; NOP ; NOP ;;
.align_code 4;
    NOP ; NOP ; NOP ; NOP ;;
    NOP ; NOP ; NOP ; NOP ;;
    NOP ; NOP ; NOP ; NOP ;;
    NOP ; NOP ; NOP ; NOP ;;
    NOP ; NOP ; NOP ; NOP ;;
    NOP ; NOP ; NOP ; NOP ;;
    NOP ; NOP ; NOP ; NOP ;;
    IF NLC0E, JUMP _inner_loop ; NOP ; NOP ; NOP ;;
    /* test/exit _inner_loop */
    NOP ; NOP ; NOP ; NOP ;;
.align_code 4;
    NOP ; NOP ; NOP ; NOP ;;
    NOP ; NOP ; NOP ; NOP ;;
    NOP ; NOP ; NOP ; NOP ;;
    NOP ; NOP ; NOP ; NOP ;;
    NOP ; NOP ; NOP ; NOP ;;
    NOP ; NOP ; NOP ; NOP ;;
    NOP ; NOP ; NOP ; NOP ;;
    IF NLC1E, JUMP _middle_loop; NOP ; NOP ; NOP ;;
    /* test/exit _middle_loop */
```



```

        NOP ; NOP ; NOP ; NOP ;;
.align_code 4;
        NOP ; NOP ; NOP ; NOP ;;
        NOP ; NOP ; NOP ; NOP ;;
        NOP ; NOP ; NOP ; NOP ;;
        NOP ; NOP ; NOP ; NOP ;;
        NOP ; NOP ; NOP ; NOP ;;
        NOP ; NOP ; NOP ; NOP ;;
        J6 = J6 - 1; NOP ; NOP ; NOP ;;
        /* decrement counter for _outer_loop */
        If NJEQ, JUMP _outer_loop; NOP ; NOP ; NOP ;;
        /* End outer loop*/

```

Listing 8-9. Branch Target Buffer usage (No Branch Prediction)

```

LC0 = 100;; /* Initialize loop count */
nop;;
nop;;
nop;;
nop;;
nop;;
nop;;
nop;;
.align_code 4;
_loop:
nop;nop;nop;nop;;
nop;nop;nop;nop;;
nop;nop;nop;nop;;
nop;nop;nop;nop;;
nop;nop;nop;nop;;
nop;nop;nop;nop;;
nop;nop;nop;nop;;
If NLC0E, JUMP _loop (NP); nop;; /* End loop */

```

The loop in [Listing 8-9](#) executes in $7+99(7+5)+1*7=1202$ cycles. The first 99 loop iterations get 5 cycles penalty at loop branches. The last iteration does not have any penalty because of NP.

Sequencer Instruction Summary

[Listing 8-11](#) 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 2-6](#).

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 15- 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 15-bit address (if possible) 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 8-19](#).



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 1-23](#) and [“Instruction Parallelism Rules” on page 1-27](#).

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.

Sequencer Instruction Summary

In compute block conditional instructions, the *Condition* is one of the following compute block conditions: AEQ, ALT, ALE, MEQ, MLT, MLE, SEQ, SLT, SF0, or SF1. To select the compute block for the condition, prefix it with X, Y, or XY, as in XAEQ. Omitting this prefix selects XY. To negate the condition, prefix it with N, as in NXAEQ.

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.

In IALU conditional instructions, the *Condition* is one of the following IALU conditions: EQ, LT, LE, ISF0, or ISF1. To select the IALU for the EQ, LT, or LE conditions, prefix it with J or K; the ISF0 and ISF1 hold condition from either IALU. To negate the condition, prefix it with N, as in NJEQ.

The JEQ, JLT, and JLE conditions are the J-IALU equal, less than, and less than or equal to zero conditions and, similarly, for K-IALU. The ISF0 and ISF1 conditions are the global/IALU static flag conditions.

In sequencer conditional instructions, the *Condition* can be any compute block condition, IALU condition, or one of the following sequencer conditions: BM, LCOE, LC1E, FLAG0_IN, FLAG1_IN, FLAG2_IN, or FLAG3_IN. To negate the condition, prefix it with N, as in NBM.


The BM condition is bus master. The LCOE and LC1E conditions are the loop counter 0 and 1 equal to zero conditions. The FLAG0_IN through FLAG3_IN conditions are the flag pin conditions.

An example sequencer instruction using a compute block condition is:

```
IF NXMLT, JUMP label_10;;
```

This instruction executes a jump to the address that corresponds to `label_10` if the condition `NXMLT` (not, X compute block, multiplier less than zero) evaluates as true.

Instructions that follow after a conditional instruction or after a jump/call in a line can have the keyword `DO|ELSE` or have no prefix. The absence of a 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`, `BTBINV`, `BTBEN`, `BTBDIS`, `BTBLOCK`, `BTBELOCK`, `TRAP`, and `EMUTRAP` instructions may not be conditional. For example, the following instruction line is not legal:

```
if aeq; do idle;; /* ILLEGAL! */
```

Listing 8-11. 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. */
```

Sequencer Instruction Summary

```
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. */
```

```
{X|Y|XY}SF1|SF0 = Compute_Cond. ;
```

```
{X|Y|XY}SF1|SF0 += AND|OR|XOR Compute_Cond. ;
```

```
ISF1|ISF0 = IALU_Cond.|Compute_Cond.|Seq_Cond. ;
```

```
ISF1|ISF0 += AND|OR|XOR IALU_Cond.|Compute_Cond.|Seq_Cond. ;
```

```
IDLE ;
```

```
BTBEN ;
```

```
BTBDIS ;
```

```
BTBLOCK ;
```

```
BTBELOCK ;
```

```
BTBINV ;
```

```
TRAP (<Imm5>) ;;
```

```
EMUTRAP ;;
```

```
NOP ;
```

9 MEMORY AND BUSES

The ADSP-TS201 TigerSHARC processor core contains a large embedded DRAM internal memory and provides access to external memory through the processor's external port. This chapter describes the processor's internal memory and shows how programs can use it. For information on connecting and timing accesses to external memory, see the "Cluster Bus" and "SDRAM Interface" chapters of the *ADSP-TS201 TigerSHARC Processor Hardware Reference*.

Most microprocessors use a single address and single data bus for memory access. This type of memory architecture is called *Von Neumann* architecture. But, DSPs require greater data throughput than Von Neumann architecture provides, so many DSPs use memory architectures that have separate address and data buses for program and data storage. The two sets of buses let the processor fetch a data word and an instruction simultaneously. This type of memory architecture is called *Harvard* architecture.

The SHARC DSPs go a step further by using a *Super Harvard* architecture. This architecture has program and data buses, but provides a single, unified address space for program and data storage. The data memory bus only carries data, and the program memory bus handles instructions or data.

TigerSHARC processors advance beyond SHARC architecture with a more open access bus structure shown in [Figure 9-1](#). The TigerSHARC processor architecture has four buses, but still provides a single, unified address space for program and data storage. The J-bus and K-bus only carry data, the I-bus handles instructions, and the S-bus handles (external) accesses from external masters (a host or other TigerSHARC processors).

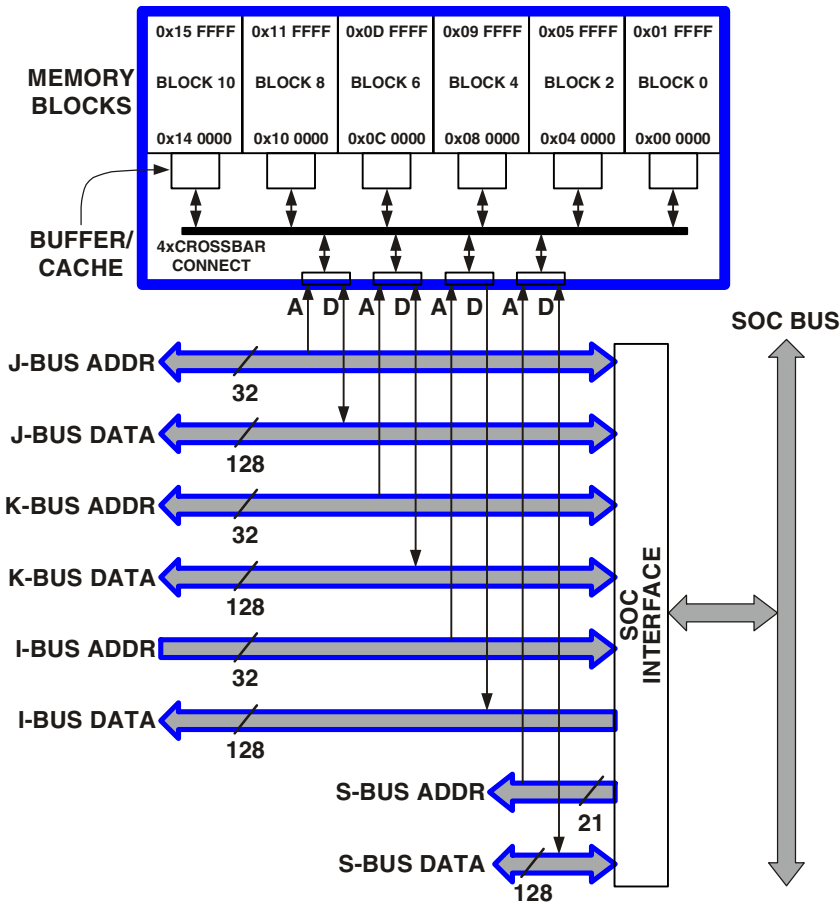


Figure 9-1. ADSP-TS201 Processor Memory and Buses

The ADSP-TS201 processor's internal memory has 24M bits of embedded DRAM memory, divided into six blocks of 4M bits (128K words × 32 bits). Each block can store instructions, data, or both, so applications can configure memory to suit specific needs. Placing instructions and data in different memory blocks, however, enables the processor to access data while simultaneously performing an instruction fetch. Each memory block contains a 128K bit cache to enable accesses with no latency induced bus stalls.


An embedded DRAM interface between each block and its crossbar connector contains a prefetch buffer, read buffer, copyback buffer, and cache. These buffers and cache optimize access between the memory (operating at one-half the processor core speed) and the buses by buffering double-width (256-bit) transfers.

The six internal memory blocks connect to the four 128-bit wide internal buses through a crossbar interface connection, enabling the processor to perform four memory transfers in the same cycle. The processor's internal bus architecture lets the core and I/O access twelve 32-bit data words and four 32-bit instructions each cycle. The processor's flexible memory structure enables:

- Processor core and system-on-chip (SOC) bus I/O accesses to different memory blocks in the same cycle
- Processor core access to three memory blocks in parallel—one instruction and two data accesses
- Programmable partitioning of program and data memory
- Program access of all memory as 32- or 64-bit words
- Program access of all memory as 128-bit words in 16- or 32-bit alignment using the Data Alignment Buffer (DAB)
- External interface access of all memory as 128-bit words

Because memory buses (J-, K-, I-, and S-bus) are associated with functional blocks (J-IALU, K-IALU, program sequencer, and SOC interface) rather than the buses being associated with memory blocks, memory access conflicts occur when the two functional blocks attempt to access the same internal memory block in the same cycle. When this conflict

happens, extra penalty cycles are incurred for the conflicting access. For more information, see [“Memory Bus Arbitration” on page 9-33](#) and [“Memory Pipeline” on page 9-35](#).

 On previous TigerSHARC processors, the three buses are associated with memory blocks. On the ADSP-TS201 processor, the four buses are associated with functional blocks. This bus difference leads to greater throughput on the ADSP-TS201 processor.

During a single cycle, dual-data access, the processor core uses the independent J-bus and K-bus to simultaneously access data from two memory blocks. Though dual-data accesses provide greater data throughput, it is important to note some limitations on how programs may use them.

The limitations on single cycle, dual-data accesses are:

- The two pieces of data must come from different memory blocks.

If the core tries to access two words from the same memory block for a single instruction, a stall cycle is issued. For more information on how the buses access these blocks, see [“Memory Pipeline” on page 9-35](#).

- The data must come from a different memory block than the instruction being fetched.

Efficient memory usage relies on how the program and data are arranged in memory and depends on how the program accesses the data. For more information, see [“Memory Programming Guidelines” on page 9-46](#).

As shown in [Figure 9-1](#), the processor has four internal buses connected to its internal memory. Memory accesses from the processor’s core (computation units, integer ALUs, or program sequencer) use the J-bus, K-bus, or I-bus, while the processor’s peripherals (external port and link ports) use the S-bus for memory accesses. Using the S-bus, the SOC interface can

provide data transfers between internal memory and off-chip connections (for example, link ports, external memory, and other processors) without hindering the processor core's access to memory.

The processor uses a priority order when arbitrating between two (or more) buses making simultaneous accesses to the same internal memory block. For more information, see [“Memory Bus Arbitration” on page 9-33](#).

[Figure 9-2](#) shows the ADSP-TS201 processor's unified memory map.

While the processor's internal memory is divided into *blocks* of internal embedded DRAM with associated buffer and cache, the processor's external memory spaces are divided into *banks* of SRAM and SDRAM with associated memory select lines. For more information, see the “Cluster Bus” and “SDRAM Interface” chapters of the *ADSP-TS201 TigerSHARC Processor Hardware Reference*.

Memory Block Physical Structure

[Figure 9-3](#) shows the embedded DRAM controller organization and data flow operations for an internal memory block.

The internal memory consists of six blocks; each block is a 4M bit embedded DRAM. The blocks are partitioned into 4 sub-arrays of 1M bit each. Each sub-array has its own 2K bit page buffer.

The embedded DRAM works in a one half frequency clock. The largest access to the embedded DRAM is of 8 words, 256 bits. To keep full execution performance, the memory system provides the processor core with up to 4 accesses per cycle from different blocks. Each block is able to continuously supply one access of up to 128 bits per CCLK cycle. Assuming that the embedded DRAM operates at one half the CCLK frequency, the embedded DRAM can keep the throughput by using the full 256-bit data input and output buses.

Memory Block Physical Structure

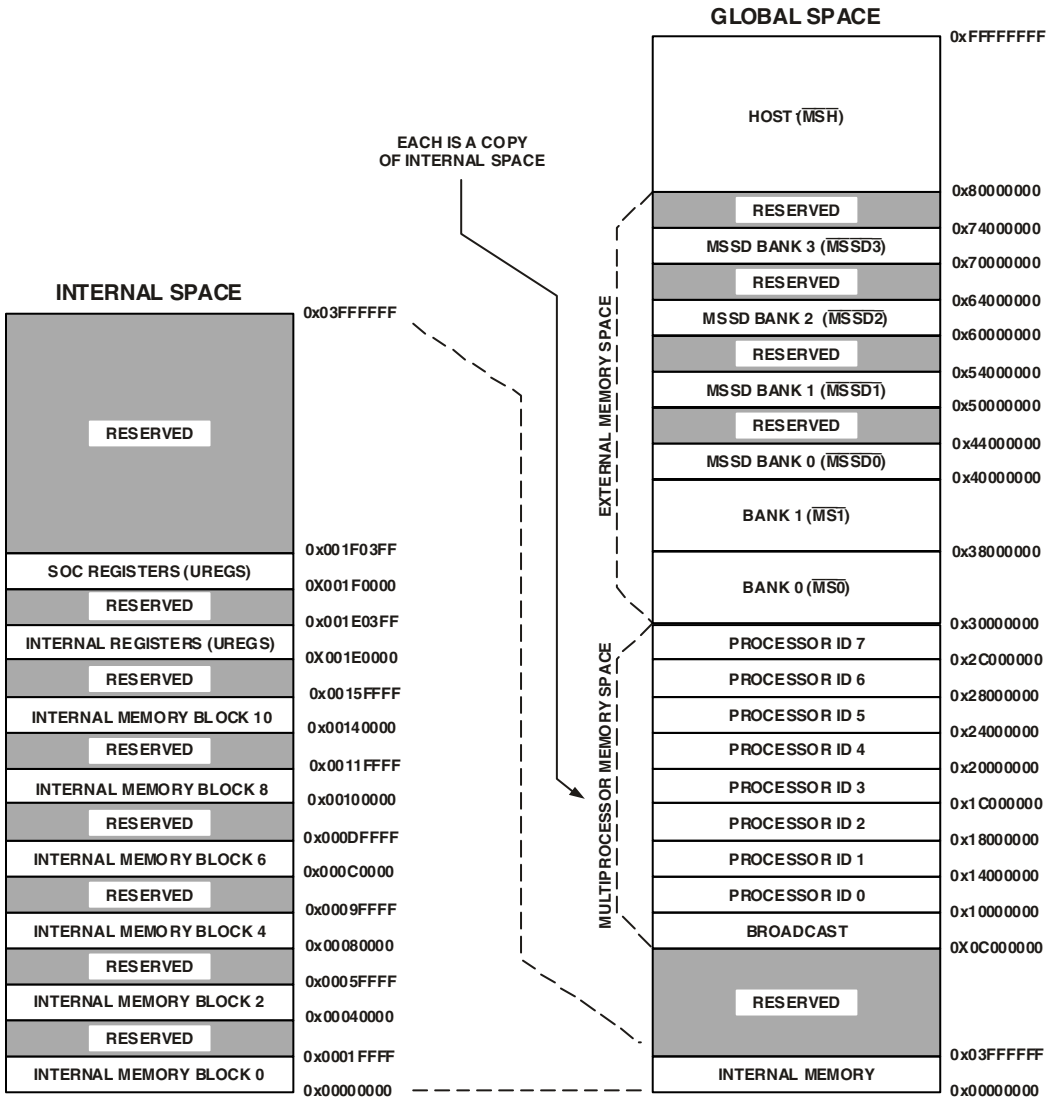


Figure 9-2. ADSP-TS201 TigerSHARC Processor Memory Map

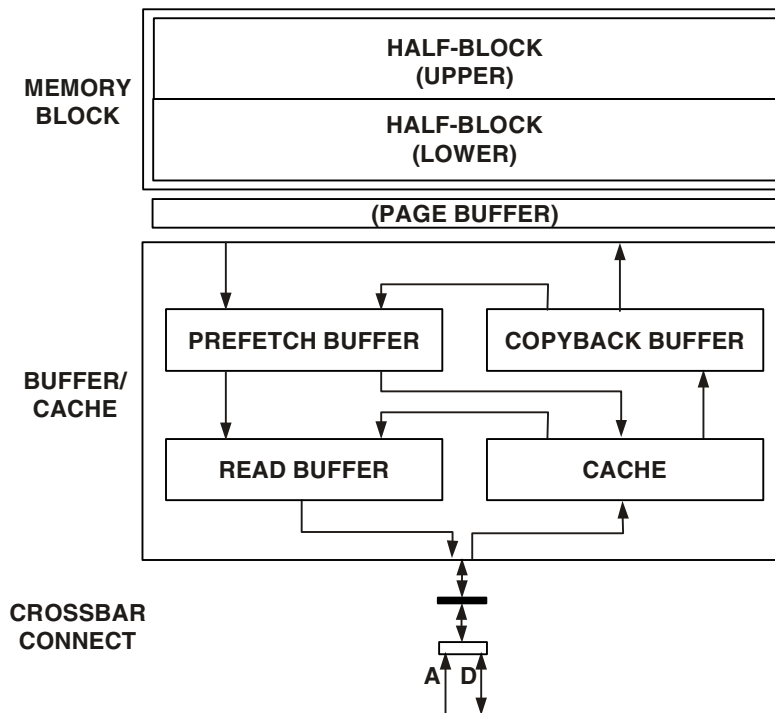


Figure 9-3. Memory Block Physical Structure

To access embedded DRAM data (read or write), the data must be in the page buffer. Each set accesses to a given page begin with an “activate” cycle followed by read or write cycles. When the accesses to the page are completed, or when there is a need to open (activate) another page in the same sub-array, the page is closed by execution of a “precharge” cycle. If the accesses to an open page are interrupted before completion by an access to a different sub-array, the new page in the other sub-array may be opened without closing the old page before. If the new page and the old page are in different half segments the old page is kept open and later accesses to the old page may be resumed without the overhead of an “activate” of this page.

Memory Block Physical Structure

During a *memory transaction*—a read or write access to a memory block, its buffers, or its cache—the crossbar connection does not directly access the memory block. The data being accessed must be in the page buffer. Each access to a page in the memory block begins with an embedded DRAM activate cycle followed by read or write cycles. When the accesses to the page are completed, or when there is a need to open (activate) another page in the same sub-array, the page in the memory block is closed by execution of an embedded DRAM precharge cycle. These processes and logical structures are explained in more detail in [“Memory Block Logical Organization” on page 9-14](#).

The operation of the page buffer is optimized through the operation of the prefetch buffer, read buffer, copyback buffer, and cache. The definitions for [Prefetch Buffer](#), [Read Buffer](#), [Cache](#), [Copyback Buffer](#), and [Buffer/Cache Hit](#) in this section provide terminology for understanding the memory controller organization and data flow operations.

Prefetch Buffer

A memory block’s *prefetch buffer* is an 8192-bit buffer organized as four 2048-bit prefetch pages. Each prefetch page is parsed into eight 256-bit words. The four prefetch pages are allocated such that two prefetch pages are dedicated to each memory half-block. Note that the prefetch pages in the prefetch buffer come from the embedded DRAM’s page buffer (see discussion [on page 9-16](#)).

The prefetch buffer handles read transactions only and provides intermediate storage for read data before the data is transferred to the read buffer and cache. The function of the prefetch buffer is to anticipate sequential read transactions and queue data from the embedded DRAM in advance of the actual transaction requests.

An automatic prefetch mechanism can load the prefetch buffer. The first access sets the prefetch direction according to its source as follows:

- Core instruction fetch – prefetch forward
- Core data read:
 - Pre-modify access – no prefetch
 - Post-modify access – prefetch according to the direction of the modifier
- DMA – prefetch direction according to the increment sign
- Cluster bus access - no prefetch

Accesses to the other half-block—in the same memory block as the page—do not affect the automatic prefetch mechanism. Also, intervening accesses that are a cache hit do not affect the automatic prefetch mechanism, even if these accesses are to the same half-block.

After determining the fetch direction, the automatic prefetch mechanism transfers sequential 256-bit octal words from the embedded DRAM to the prefetch buffer automatically.

This prefetch mechanism lets the embedded DRAM interface accommodate sustained sequential read transactions at maximum processor core throughput without incurring penalties (stall cycles) even though the embedded DRAM operates at one-half the frequency of the processor core. Only unanticipated accesses incur penalties.

Each of the four prefetch pages in the prefetch buffer may contain data from a single page of the embedded DRAM at any time (see discussion [on page 9-16](#)). Each prefetch page of the prefetch buffer utilizes a page tag to indicate which page of the embedded DRAM is currently allocated. Further, each of the eight 256-bit entries in each prefetch page has a valid bit to indicate the entry is equivalent to the corresponding octal word in the embedded DRAM. A valid bit is set when data is transferred from the

Memory Block Physical Structure

embedded DRAM to the prefetch buffer. The valid bit is cleared when this part of the prefetch buffer is assigned to a different page. If data from a prefetch page not currently queued is required, the least recently used (LRU) prefetch page of the two prefetch pages dedicated to the particular half-block is cleared and is used for the new prefetch page.

An octal word in the prefetch buffer is invalidated when data pertaining to the same octal word enters the copyback buffer. This invalidation occurs following a cache replacement of dirty data (or a direct copyback buffer write) when the cache is disabled or locked.

The prefetch buffer also acts as a cache. When a read transaction matches one of the page tags of the four pages of the prefetch buffer and the corresponding valid bits are set, the transaction is a *prefetch buffer hit*, and the transaction incurs no penalties.

Read Buffer

A memory block's *read buffer* is a 512-bit buffer organized as two 256-bit octal words—one for each half-block. The read buffer handles read transactions only and connects the memory crossbar to both the cache and the prefetch buffer for all read transactions. Note that read transaction's data ultimately pass through the read buffer regardless of its origins.

Each read transaction transfers a 256-bit octal word from the cache or the prefetch buffer to the read buffer. From the read buffer, 32, 64, or 128 bits are extracted to service the transaction request. The extracted data is placed on the data bus (J-, K-, I-, or S-bus), matching the transaction requestor. When new data enters the read buffer, the data also enters the cache.

The read buffer's purpose is to cache a 256-bit access. If the next access is within the same 256-bits, the read buffer enables internal memory-system operations that usually take two cycles (for example, cache replacement with copyback) to continue while the second transaction is completed without stalls. A word tag is maintained for each of the 256-bit entries. If

a read transaction matches the word tag of a read buffer entry and the valid bits of the accessed words are set, the transaction is a *read buffer hit*. If a transaction is a read buffer hit, the transaction incurs no penalty and does not affect other parts of the memory system.

The read buffer keeps a valid bit for each of its 32-bit words. When a given octal word exists in the read buffer, a write transaction to one or more 32-bit words in this octal word clears the valid bits corresponding to the written 32-bit words, which in turn are written into the cache. A subsequent read to any portion of the octal word returns the correct data by merging the valid portions of the cache and the read buffer before returning the data to the requesting bus.

Cache

A memory block's 128K bit, 4-way set associative *cache* is organized as 128 *cache sets* with each set containing four *cache ways*. Each cache way contains a *cache data* that is 256 bits wide and is organized as eight contiguous 32-bit words. Each word has its own valid bit to allow read and write transactions using 32-bit word granularity. Each memory block's cache serves as a unified data/instruction storage cache. The cache replacement policy is least recently used (LRU) with last replaced page (LRP) optimization for embedded DRAM. The cache write policy is copy-back and no write allocate. [For more information, see “Cache Operation” on page 9-25.](#)

Copyback Buffer

A memory block's *copyback buffer* contains two separate 4096-bit buffers—one for each half-block of the embedded DRAM. Each buffer accommodates sixteen 256-bit entries.

Memory Block Physical Structure

When a way's cache data is replaced and the cache way's *dirty bit is set*, the cache data must be copied back to the embedded DRAM. This transfer is done using the copyback buffer. The copyback buffer serves as an intermediate holding location for replaced cache data that must eventually be written back to the embedded DRAM.

The prefetch buffer can snoop (observe) the contents of the copyback buffer. This snooping means that when the prefetch buffer reads data in the copyback buffer (that is also in the embedded DRAM), the data is taken from the copyback buffer and not from the embedded DRAM. This snooping mechanism is only for data correctness and not access performance. The data is returned from the copyback buffer with the same delay that it is returned from the embedded DRAM. When the prefetch buffer gets data by snooping the contents of the copyback buffer, the transaction is a *copyback buffer hit*.

Note that the copyback buffer is the sole interface to the embedded DRAM for all write transactions. When a write transaction is not cacheable, the copyback buffer provides a bridge between the crossbar interface and the embedded DRAM for write transactions.

Buffer/Cache Hit

In the memory system, there are multiple sources of cached data. In addition to a memory block's cache, the memory block's prefetch buffer and read buffer also act as caches. A *buffer/cache hit* occurs when a prefetch buffer hit, a read buffer hit, or a cache hit (or a combination of these) occurs. For example, one combination resulting in a buffer/cache hit would be a 128-bit (quad word) read request that produces a prefetch buffer hit while one of the 32-bit words in the requested quad word produces a cache hit. In this case, data from multiple sources is merged into the read buffer before being transferred to the crossbar interface. Note that a buffer/cache hit does not include a copyback buffer hit.

Memory Block Terms, Sizes, and Addressing

The terms [Cache](#), [Block](#), [Half-Block](#), [Sub-Array](#), and [Page](#) have associated sizes and address relationships. [Table 9-1](#) shows the specifications for these and other memory block terms.

Table 9-1. Memory Block Terms, Sizes, and Addressing

	Block	Half-Block	Sub-Array	Cache	Page	Cache Set	Cache Line
Addressing in the unit	16–0	15–0	15–7,5–0	-	5–0	-	5–3
Address bits to select unit	20–17	16	16, 6	-	16–6	16–10	16–3
Number of blocks	1	-	-	-	-	-	-
Number of half-blocks	2	1	-	-	-	-	-
Number of sub-arrays	4	2	1	-	-	-	-
Number of pages	2K	1K	512	64	1	-	-
Number of cache sets	-	-	-	128	-	1	-
Number of cache ways	16K	8K	4K	512	8	4	1
Number of quads	32K	16K	8K	1K	16	8	2
Number of words	128K	64K	32K	4K	64	32	8

Memory Block Logical Organization

Figure 9-4 shows the memory data organization within a memory block. The definitions for **Block**, **Half-Block**, **Sub-Array**, and **Page** in this section provide terminology for understanding the memory data organization.

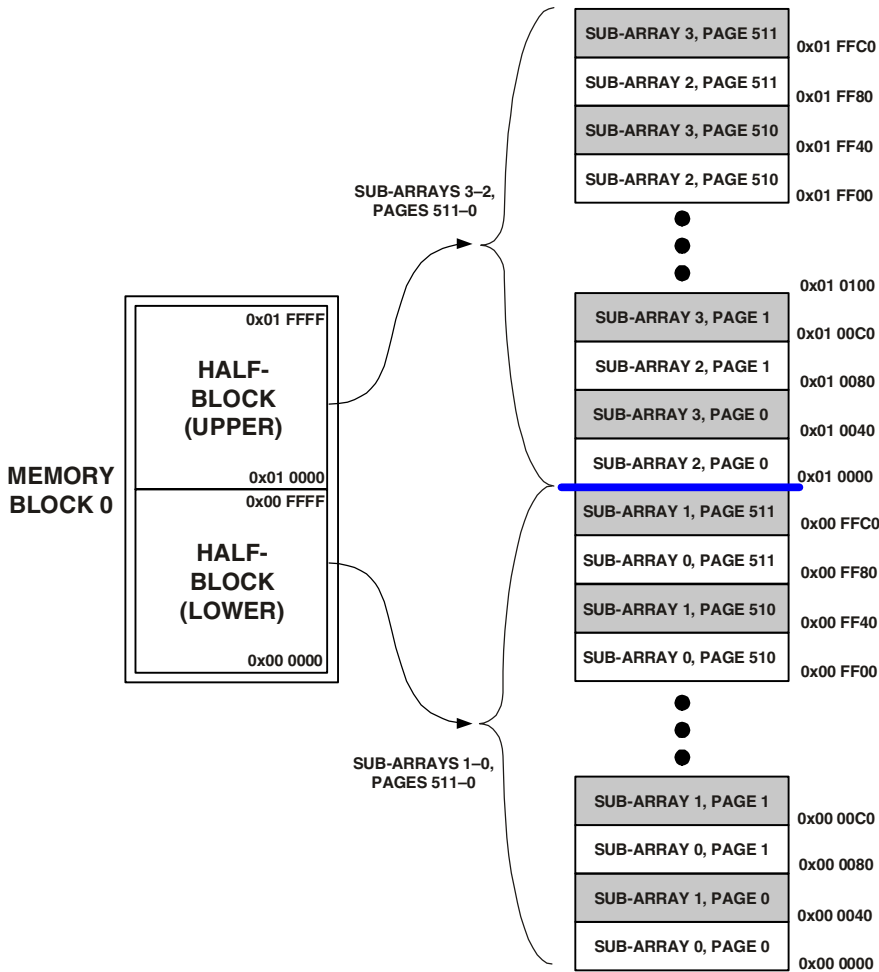


Figure 9-4. Memory Block, Half-Block, Sub-Arrays, and Pages

Block

A memory *block* is a 4M bit (128K word) unit of memory that includes a stand alone block of embedded DRAM with its associated buffers, cache, and controls. On the ADSP-TS201 processor, there are six blocks of internal memory. Each memory block may accept one access every cycle. These accesses use the memory pipeline. A memory block contains two half-blocks, composed of two sub-arrays each.

Half-Block

A memory *half-block* (64K word) is an important subdivision of memory because each half-block of memory can support sustained sequential accesses to a single data block. Each memory half-block has its own prefetch buffer, read buffer, and copyback buffer, but each memory block has one cache. This independent buffering for each memory half-block lets each memory block support sustained sequential accesses to two independent data blocks—one in each memory half-block. Each memory half-block contains two interleaved memory sub-arrays.

Sub-Array

A memory *sub-array* is a 1M bit (32K word) unit of embedded DRAM memory that is interleaved in sets of 2K bit pages with another sub-array to form a half-block as shown in [Figure 9-4](#). Sub-arrays 0 and 1 are interleaved over the lower half of the block, and sub-arrays 2 and 3 are interleaved over the upper half of the block. Each of the four sub-arrays has its own dedicated *page buffer* that is loaded by an activate command. After the activate command, the data on the open page is accessible. For more information on “activate” and other embedded DRAM terminology, see “[Memory Block Accesses](#)” on [page 9-16](#). Each sub-array contains 512 memory pages.

Memory Block Accesses

Page

A memory *page* is a 2K bit (64 x 32-bit word) unit of memory. During a memory read access (on a cache miss), the embedded DRAM interface reads the whole page containing the accessed location into the page buffer corresponding to the sub-array containing the page. After the data access, the embedded DRAM interface copies the page to the prefetch buffer for usage in future accesses. On write embedded DRAM accesses, the page is copied to the page buffer prior to the write, and the data is written from the copyback buffer to the page buffer. If there are more octal-words to be written to the same page, it is being done sequentially (without closing and opening the same page). At the end of the write process, the page is closed.

Memory Block Accesses

When a buffer/cache miss occurs, the prefetch buffer must fetch the required data from the embedded DRAM. [Figure 9-5](#) provides an overview of the embedded DRAM access procedure and provides a general guideline of when *penalties*—extra clock cycles required to activate and load a page of embedded DRAM—occur. The penalties associated with accessing embedded DRAM are variable and influenced by multiple factors. Understanding the origins of the penalties is critical when developing strategies to minimize the impact on processing efficiency and to identify how these penalties may be avoided.

Examining [Figure 9-5](#), it is important to note that the embedded DRAM interface does not need to open and close a page for every access. When a page is opened, more than one transaction may make use of the open page. Also note that the page open and close operations are handled automatically by the embedded DRAM interface. Programmers only need to be aware of this process for penalty assessment.

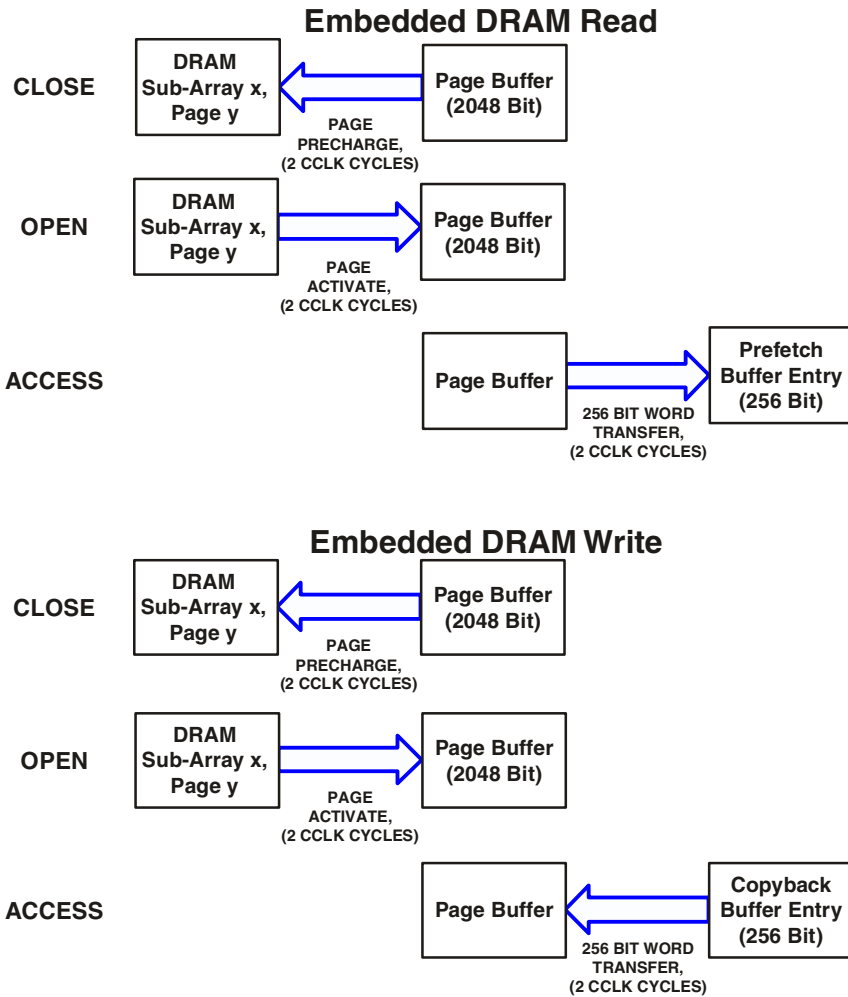


Figure 9-5. Memory Block Access¹

¹ Before closing a memory page, a synchronization process is performed, which requires one to two CCLK cycles.

Memory Block Accesses

The embedded DRAM clock (DCLK) operates at one-half of the frequency of the processor core clock (CCLK). All accesses to the embedded DRAM are eight words (256 bits). To support full speed processor core execution, the memory system provides the core with up to four accesses per cycle from different memory blocks. Each block is able to accept one access of 128 bits per CCLK cycle. With the embedded DRAM operating at one-half CCLK frequency, the embedded DRAM maintains throughput by using the full 256-bit data input and output buses.

Accesses to one sub-array can be interrupted before completion by accesses to a different sub-array. When this happens, the new page in the second sub-array may be opened without closing the open page on the first sub-array. If the second page and the first page are in different half-blocks, the first page is kept open, and accesses to the first page may be resumed without the activation overhead of this page.

The definitions for [Activate](#), [Precharge](#), and [Refresh](#) in this section provide terminology for understanding embedded DRAM memory operations.

Activate

When a memory transaction requires placing a new page in the sub-array's page buffer, the embedded DRAM system must *activate* (open) a page and place it in the page buffer. After the activate operation, the data can be accessed directly from the page buffer until the sub-array is precharged. The activate operation has a two processor core clock (CCLK) latency.

Precharge

When a memory transaction requires that an active page in the sub-array's page buffer must be replaced with a new page from the same sub-array, the embedded DRAM system must *precharge* (close) the active page and

update page data in the sub-array from the buffer. After precharge (a two CCLK process), a new activate operation is required to place (open) the requested new page in the buffer.

Refresh

The embedded DRAM system must *refresh* the data in every address at least once every 3.2 milliseconds to prevent data loss through process leakage; in other words, a refresh should occur once every 1.6 microseconds. The technology for embedded DRAM memory cells is based on charging and discharging of small capacitors. A capacitor is charged to a level of 0 or 1, and the charge is theoretically kept until it is read. A limitation on this technology is *process leakage*, which may cause the capacitor to lose its value. The refresh (a six CCLK process) reads the data and writes it back to the embedded DRAM cell. This process is automatically executed by the embedded DRAM at the refresh rate applied with the [Refresh Rate Select](#) command.


Memory System Controls and Status

To optimize the embedded DRAM interface's operation activate, precharge, and refresh processes, programmers can set up the interface features to match their system needs. The embedded DRAM refresh rate and other embedded DRAM interface controls are applied to the interface using memory system control and status registers for each internal memory block. These registers include:

- Cache Status (CASTATx) registers
- Memory System Command (CACMDx) registers
- Cache Address/Index (CCAIRx) registers

Memory Block Accesses

The value for x can be 0, 2, 4, 6, 8, 10 or B (for broadcast to all blocks). For example, CACMD0 is the memory system command register for memory block 0, CACMD2 is the memory system command register for memory block 2, and so on. This numbering applies to the CCAIR x and CASTAT x registers as well.

 Note that reading a memory block on the cycle following a write to that block's CACMD x or CCAIR x registers causes a four cycle stall.

The CASTAT x register is a read-only register that indicates the cache status. This register can be read using the $Ureg = Ureg$ register transfer instruction. For bit definitions of these registers, see [Figure 9-6](#), [Figure 9-7](#), and [“DEFTS201.H – Register and Bit #Defines File”](#) on page B-1.

Programs issue memory system commands by setting or clearing bits in the CACMD x register. These registers are accessed using register transfer instructions ($CACMDx = IALU_DATA_Ureg$), or register immediate load instructions ($CACMDx = Imm$). These commands are available as macros in the DEFTS201.H file. For bit definitions of the commands for the CACMD x registers, see [“DEFTS201.H – Register and Bit #Defines File”](#) on page B-1 or [Table 9-2](#).

The bits in the CCAIR x register only may be set or cleared using register transfer instructions ($CCAIRx = IALU_DATA_Ureg$)¹.

There are different types of memory system commands. These commands are described in:

- [“Refresh Rate Select”](#) on page 9-60
- [“Cache Enable”](#) on page 9-62
- [“Cache Disable”](#) on page 9-63
- [“Set Bus/Cacheability \(for Bus Transactions\)”](#) on page 9-64

¹ Where $IALU_DATA_Ureg$ is any IALU data register (J30–0 or K30–0)

- “Cache Lock Start” on page 9-68
- “Cache Lock End” on page 9-69
- “Cache Initialize (From Memory)” on page 9-70
- “Cache Copyback (to Memory)” on page 9-74
- “Cache Invalidate” on page 9-78

Most of these instructions are executed as soon as the Memory System Command register is written (EX2 instruction pipeline stage executes the writes to CACMD_x). Some of these Memory System Commands are executed as a sequence, and the execution is in the background to normal program execution. These commands are cache initialize, cache copyback, and cache invalidate. While the command is executed, other memory accesses (to the cache) are allowed. The accesses of core and system (DMA or external master) are in higher priority than the command execution. An instruction fetch has the same priority as memory system command execution. Therefore, the selection between the two sources—fetch or execute—if there is a conflict) takes turns.

For the cache initialize, cache copyback, and cache invalidate commands, the CCAIR_x register must be preloaded to hold either a start address or a start index depending on the command being executed in the corresponding CACMD_x register.

Memory Block Accesses

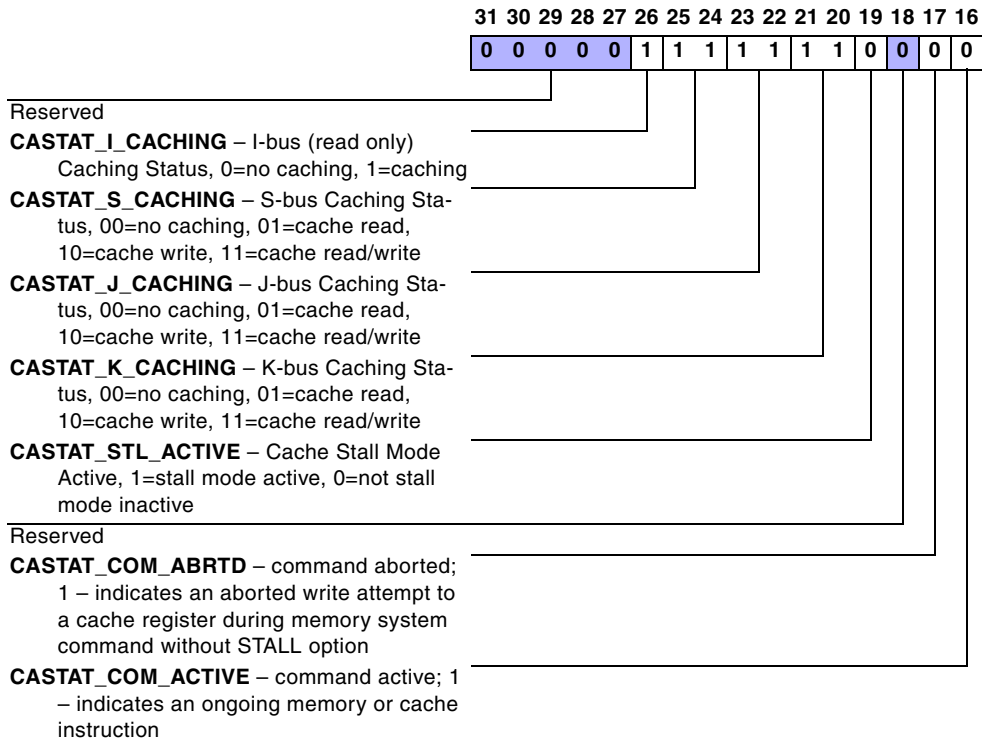


Figure 9-6. CASTATx (Upper) Register Bit Descriptions

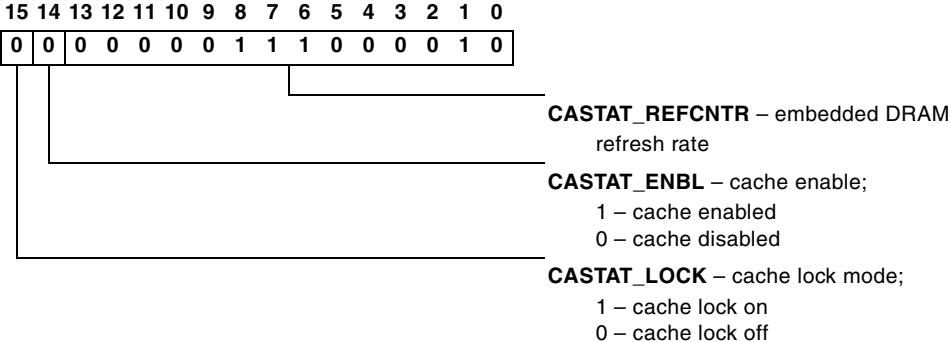


Figure 9-7. CASTATx (Lower) Register Bit Descriptions

Memory Block Accesses

Table 9-2. CACMDx Register Bit Definitions

Command Mnemonic	Opcode Bits				
	31–26	25–24	23–15	14	13–0
CACMD_REFRESH	010100	00	00000000	0	Refresh_Rate
CACMD_EN	000000	00	00000000	0	00 0000 0000 0000
CACMD_DIS	000001	00	00000000	0	00 0000 0000 0000
CACMD_SLOCK	000010	00	00000000	0	00 0000 0000 0000
CACMD_ELOCK	000011	00	00000000	0	00 0000 0000 0000
CACHE_INIT	010000	00	Length	Stall_Mode	00 0000 0000 0000
CACHE_INIT_LOCK	010001	00	Length	Stall_Mode	00 0000 0000 0000
CACMD_CB	000100	00	Length	Stall_Mode	00 0000 0000 0000
CACMD_INV	000101	00	Length	Stall_Mode	00 0000 0000 0000
CACMD_SET_BUS	000111	00	Caching ¹	0	00 0000 0000 0000

¹ Where caching sets bus caching ability for each bus; I-bus (bit 21) 0=no caching, 1=read caching; S-bus (bits 20–19), J-bus (bits 18–17), and K-bus (bits 16–) 00=no caching, 01=read caching, 10=write caching, and 11=read/write caching

In these cases, the bits in CCAIRx hold the following:

- When used as a start address, CCAIRx holds a 14-bit address, using bits 16–3 (bits 2–0 are ignored).
- When used as a start index, CCAIRx holds a 7-bit index, using bits 16–10 (bits 9–0 are ignored).



While a long execution command is in progress, the CACMDx and CCAIRx registers may not be changed. For more information, see the Stall_Mode option for “Cache Initialize (From Memory)” on page 9-70, “Cache Copyback (to Memory)” on page 9-74, and “Cache Invalidate” on page 9-78.

Cache Operation

Before examining read and write accesses in more detail (see “Data Access on Read” on page 9-30 and “Data Access on Write” on page 9-32), it is important to gain a detailed understanding of cache operation. Figure 9-8 shows the cache architecture for a memory block.

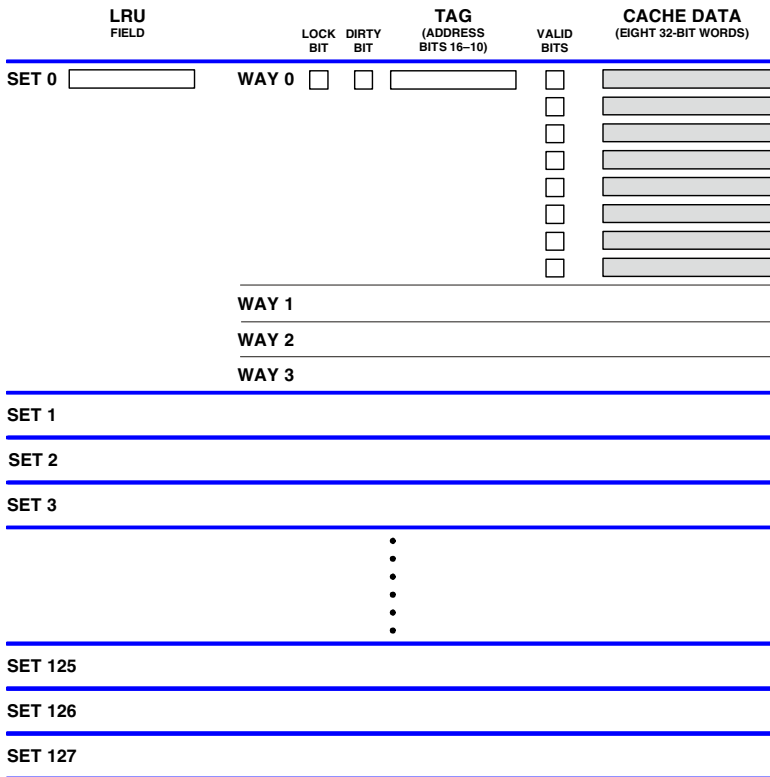


Figure 9-8. Instruction/Data Cache Architecture

Memory Block Accesses

Each cache set has the following control bits to manage its cache ways:

- The least recently used (LRU) field—six bits per cache set—indicates the order in which the ways have been accessed. These bits are used in the replacement decision process.
- The lock bit—one per cache way—indicates that a way is locked and may not be replaced.
- The dirty bit—one per cache way—indicates that one or more words of way's cache data differ from (are “fresher than”) the corresponding words of the embedded DRAM.
- The valid bits—eight per cache way—indicate the validity of each 32-bit word of the cache data within the way. On a read access, the valid bits are set on a cache miss when 8-words of cache data are loaded from the embedded DRAM or the prefetch buffer. On a write access, only the valid bit(s) associated with the words from the cache data that are written by the transaction are set. The valid bits are cleared when a write replaces cache data (the valid bits of the words that were not written are cleared) or following a cache invalidate command.
- The tag bits—seven per cache way—correspond to bits 16–10 of the line's address in embedded DRAM.

Figure 9-9 shows how the cache partitions the address of every memory transactions into fields. The fields from the address are:

- Tag field – (bits 16–10 of the address) – these bits are stored for comparison with the tag field in the cache.
- Index field – (bits 9–3 of the address) – these bits select the cache set for the comparison.
- Displace field – (bits 2–0 of the address) – these bits select the word(s) within the cache data.

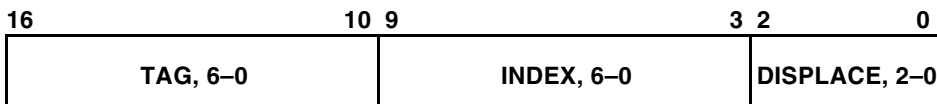


Figure 9-9. Memory Transaction Address Partitioning

During each memory transaction, the cache extracts the tag field (bits 16–10 of the address) and the index field (bits 9–3 of the address). The cache selects the set in the cache according to the index field and compares each of the set’s way’s tag fields to identify the tag field of the transactions. The comparison determines whether the transaction is a cache hit or cache miss.

- For a write transaction, the tag field of the transaction must match the tag field of a cache way for a cache hit.
- For a read transaction, the tag field (bits 16–10) of the transaction must match the tag bits of a cache way (and the cache data must be *valid*—valid bit set) for a cache hit.

Memory Block Accesses

The valid bits are set according to the type of memory transaction and displace bits:

- Single word (32-bit) access:

Displace = 0: Valid 0, Displace = 1: Valid 1, Displace = 2: Valid 2,
Displace = 3: Valid 3, Displace = 4: Valid 4, Displace = 5: Valid 5,
Displace = 6: Valid 6, Displace = 7: Valid 7

- Dual word (64-bit) access:

Displace = 0: Valid 0 and 1, Displace = 2: Valid 2 and 3,
Displace = 4: Valid 4 and 5, Displace = 6: Valid 6 and 7

- Quad word (128-bit) access:


Displace = 0: Valid 0, 1, 2, and 3,
Displace = 4: Valid 4, 5, 6, and 7

If a transaction is a cache hit, the buffer/cache interface reads or writes the corresponding cache data without penalty. If the transaction is a cache miss, a penalty occurs if the buffer/cache interface must read or write the needed data in the embedded DRAM. If a read transaction is a cache miss, the data is read from either prefetch buffer, read buffer, or the embedded DRAM. If a write transaction is a cache miss, a replacement occurs and the data is written to the replaced cache way.

When there is a cache miss, it may be necessary to replace existing cache data. *Regular replacement* of cache contents is the procedure of reassignment of a cache way to a different address of the embedded DRAM. Replacement occurs every time there is a cache miss, provided there is no free (invalid) way in the cache set that is associated with the miss transaction. Regular replacement differs from *copyback replacement*—replacement of a cache way that is selected for replacement, has its dirty bit set, and has to have its cache data copied back to the embedded DRAM before replacement.


After a miss transaction address cycle, the cache uses the least recently used (LRU) algorithm to find a replaceable way according to the following priority:

- Is there an invalid way—a way that is not assigned to any address—in the cache? (highest priority for replacement)
- (Else) Use the LRU cache data for replacement. (lowest priority for replacement)

 Locked ways are not used for cache replacements; the lock overrides LRU. Locked ways are not replaced until they are invalidated.

The cache lock affects cache replacement. If a program needs a certain data or code block to be in the cache and locked, the program should use the `CACMD_SLOCK` memory system command to start the lock, then run through the block. Running through the block copies lines to the cache. Every way that is touched—no matter if it is a new data copied from embedded DRAM or data that is a hit—is locked. When all ways in a set are loaded and locked, nothing more can be put into this cache set. This continues until the `CACMD_ELOCK` memory system command ends the lock. Locking data also could be done using the `CACHE_INIT_LOCK` memory system command.

After the lock procedure, the ways that are locked are not replaced. If all four ways in a set are locked, there are no replacements there. A miss that belongs to this set causes a prefetch start on a read, but the data is not put into cache. In case of a write miss to a fully locked set, the data goes directly to the copyback buffer.

 Note that the cacheability for transactions on each of the buses (I-, S-, J-, and K-bus) are configurable, and this configuration affects cache replacement. For the I-bus (instruction fetch), each memory block can be configured to cache read accesses or not to cache read accesses. For the S-, J-, and K- buses, each memory block can be configured to cache read accesses, to cache write accesses, to cache read and write accesses, or not to cache accesses. The cache checks

Memory Block Accesses

the cacheability of a transaction when determining cache replacement (after a cache miss). If the bus for the transaction was not cacheable, there is no cache replacement; the transaction operates as if the corresponding cache way is locked. For more information on cacheability, see [“Set Bus/Cacheability \(for Bus Transactions\)” on page 9-64](#).

Data Access on Read

When a transaction is transferred into the cache, the buffer/cache interface reads the tag bits, LRU field, valid bits, dirty bit, and lock bit from the memory block’s cache control bits (see [“Cache Operation” on page 9-25](#)). The cache hit or miss logic determines if there is a hit and which is the selected cache way. If there is a miss, the indication is passed back to the hit/miss logic that delays the address acknowledge output. The address and the way indication are transferred to the memory interface. In case of a hit, the cache data is copied to the read buffer and passed to the data bus.

In parallel with the check for a cache hit, the prefetch buffer and read buffer are also checked. If there is a prefetch buffer hit, the data is read from the prefetch buffer to both the cache—causing a replacement—and to the destination bus of the data. If there is a read buffer hit, the data is read from the read buffer, ignoring the cache. Events of cache miss and read buffer hit are rare, but in many cases read buffer hit may occur when the cache is busy with a previous operation (for example, copyback replacement from previous cycle). In this case, the read buffer hit avoids the need to stall.

If there is a buffer/cache miss on read access, the page is read from the embedded DRAM. The data that was read from the embedded DRAM goes both to the cache (replacing old data) and to the data output. This procedure typically causes five to six stall cycles on the bus (J-IALU,

K-IALU, sequencer, or SOC interface) that has requested the data. Because the page buffer is opened, all of the data of this page may be placed into the prefetch buffer.

After the data that was requested is read from the embedded DRAM, there may be other accesses that are executed in parallel to the prefetch. There are different scenarios for the following accesses and the reaction of the cache controller.

- Access to other parts of the same cache data (the eight 32-bit words associated with a way) – The full cache data that was fetched from the embedded DRAM is kept in the read buffer, and in this case, the controller indicates a read buffer hit. The data is returned with no overhead while the memory controller continues the prefetch procedure.
- Prefetch buffer hit access – If the access is to an address that has already been prefetched, the prefetch buffer indicates a hit. The data is transferred in parallel to the bus for the transaction execution and to the cache for replacement. If the replacement requires a copyback, the replaced data is copied into the copyback buffer.
- Cache hit access – This transaction is completed with no extra stalls and does not interfere with the prefetch procedure.
- Miss access while prefetch is in progress – In this case, the prefetch procedure is stopped and the memory system begins an access in the embedded DRAM to the requested data. At this point, the issue of using memory blocks partitioned into half-blocks becomes important. There is a performance difference if the new access is in the same half-block as the prefetch (not recommended) or in the second half (recommended):
 - If the access is to the same half-block, the active prefetch procedure is aborted, the page is closed, and the new access begins. In this case, there are two penalties on top of the normal miss penalty. First, if the new access is in the same

Memory Block Accesses

sub-array as the aborted page, there are two extra cycles to precharge the sub-array before the new page is opened. In this case, the penalty is 7–8 cycles. A second penalty is paid later if there are new accesses to the first page. The full overhead of a new miss access also applies.

- If the access is to the second half, the prefetch procedure is paused and the access to the requested data is immediately in effect. In this case, the new access overhead is 5–6 cycles. Also, if later there is an access to the previous page, it is either a prefetch buffer hit (no overhead) or a miss from an open page (two to four cycle overhead). Also, the prefetch procedure shall continue from the point that it was paused.

Data Access on Write

Write accesses differ from read accesses with regard to cache operations. There is no overhead on either write hit or miss. In both cases, the data is written to the cache data (see [“Cache Operation” on page 9-25](#)). If the access is a hit, it is written to the way that the tag was matched. In case of a miss, the data is written to the replaced way. The write access does not directly cause any access to the embedded DRAM. The write to the embedded DRAM occurs only when the written cache way is replaced using the copyback buffer.

Some penalty cycles are incurred for a write miss if the copyback buffer is full. This penalty may occur in one of two cases. One case occurs when the replacement is a copyback replacement (cache way is marked dirty). The other case occurs if the write transaction is defined as non-cacheable. In both cases, a stall occurs until the copyback buffer frees some entries by writing to the embedded DRAM.

There is no write allocate policy in this memory system, and this service is provided by the eight valid bits per way. The hit criteria on write is tag match with no reference to the valid bits. As a result of the write hit trans-

action, the valid bits of the written words are set, while other valid bits are not changed. In case of a write miss, the valid bits of the written data are set while the other valid bits are cleared.

Memory Bus Arbitration

Each memory block connects to the processor's buses through the crossbar interconnection shown in [Figure 9-1 on page 9-2](#). Memory transactions can run on all four buses (S-bus, I-bus, J-bus, and K-bus). If the buffer/cache interface is busy serving a stalled miss transaction, the buffer/cache interface stalls any transaction that is targeted to its memory block. To arbitrate conflicting accesses to a memory block, the address decoder monitors all transactions on all buses. At first, the decoder checks if a transaction is targeted to itself. If more than one transaction is targeted to itself, it transfers the highest priority access.

The crossbar passes the selected transaction to the memory system. The non-selected transaction is kept in local FIFO buffers until its turn. There is a two entry FIFO for every source—one for the I-bus, one for S-bus, and one shared for J-bus and K-bus. The FIFO of J-bus and K-bus is common, but each entry can hold two transactions that occurred in the same cycle. One transaction from J-bus and one from K-bus.

The processor uses the following priority order when arbitrating between two (or more) buses making simultaneous accesses to the same internal memory block. Each transaction source can come from a bus or the bus's FIFO. In the following priority order, a FIFO indicates a transaction (held in the FIFO) that was not acknowledged by the memory block on the first access, while a bus represents the first access itself.

1. S-bus FIFO (SOC interface FIFO, high priority)
2. S-bus (SOC interface, high priority)
3. J-bus/K-bus FIFO)

Memory Block Accesses

4. J-bus
5. K-bus
6. S-bus FIFO (SOC interface FIFO, low priority)
7. S-bus (SOC interface, low priority)
8. I-bus (instruction fetch FIFO)
9. I-bus (instruction fetch)

A transaction from SOC interface is in high priority when it is:

- A read transaction from cluster bus
- A DMA access where the priority in the internal TCB was high
- Any access when the SOC interface IFIFO is over a threshold

In all other cases, the SOC transactions are low priority.

When a transaction is stalled because of priority, it is pending until its turn to execute according to priority. New transactions may bypass old transactions from different sources, if their priority is higher. For example, the SOC high priority transaction may bypass a transaction issued by the J-IALU. The order of transactions from the same source must be kept. At the same time, a high priority SOC transaction may not bypass an earlier low priority SOC transaction, but it does raise the priority of the pending SOC transactions. The order of transactions from the same source also is preserved between different blocks.

Note that the priority of J-IALU transactions over K-IALU transactions refers to transactions in the same cycle. A K-IALU delayed transaction takes priority on a newer J-IALU transaction. In other words, J-IALU transactions and K-IALU transactions are from the same source.

Memory Pipeline

To support high speed operation, the TigerSHARC processor uses a memory pipeline. The memory pipeline operates in parallel with the instruction pipeline. Figure 9-10 shows the memory pipeline stages and shows how the pipeline interacts with the instruction pipeline.

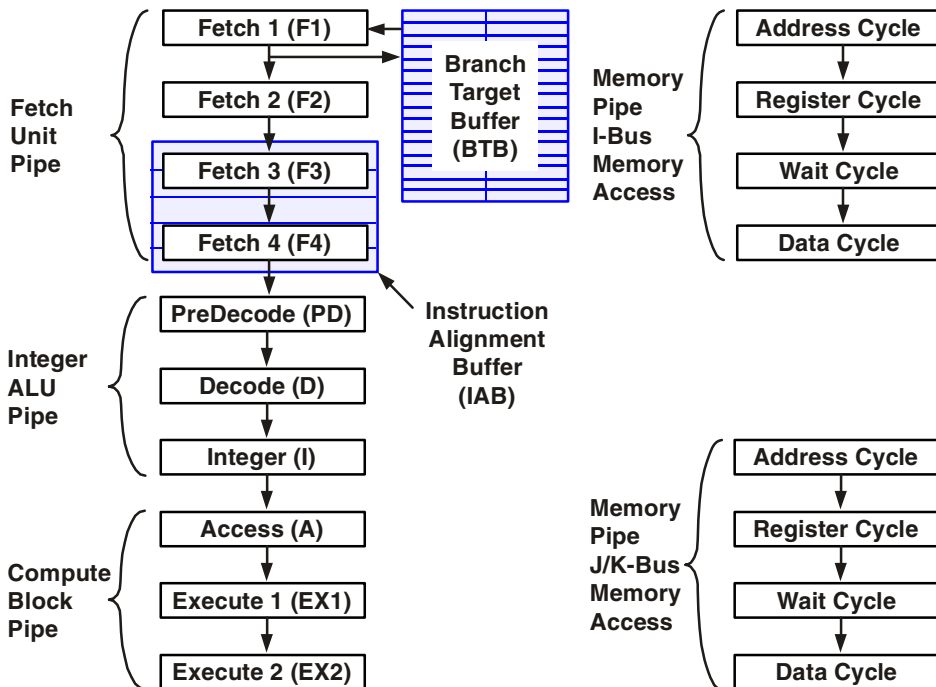


Figure 9-10. Instruction Pipeline Versus Memory Pipeline



As shown in Figure 9-10, the memory pipeline operates in parallel with the instruction pipeline. Stalls in the register cycle of the memory pipeline appear as stalls in the corresponding stages of the instruction pipeline.

Memory Pipeline

From start to finish, a memory transaction requires four cycles to traverse the memory pipeline. The data throughput is four bus transactions every processor core clock (CCLK) cycle. The memory pipeline stages are Address, Register, Wait, and Data.

Address	<p>The cache hit/miss logic compares the address of the transaction. The comparison includes checks for:</p> <ul style="list-style-type: none">• Accesses to the same memory block—the highest priority access continues while the other accesses are stalled.• Accesses to a busy memory block—all accesses to a busy block are stalled.• Cache, prefetch buffer, and read buffer hit or miss—in case of a buffer/cache miss, the stall is activated in the register cycle.
Register	<p>The cache controller logic accesses the cache during this pipeline stage. Miss transactions are stalled in this cycle.</p>
Wait	<p>The cache controller logic reads data from the cache on read transactions.</p>
Data	<p>The cache controller logic transfers the data from the cache to the transaction destination. In read cycles, data is at the target register. On write transactions, the data is at the cache.</p>

There are important relationships between the memory pipeline and the instruction pipeline. As shown in [Figure 9-10](#) and [Figure 9-11](#), the memory pipeline parallels the instruction pipeline in two places—during Fetch 1, 2, 3, and 4 for instruction fetch (I-bus transactions) and during Integer,

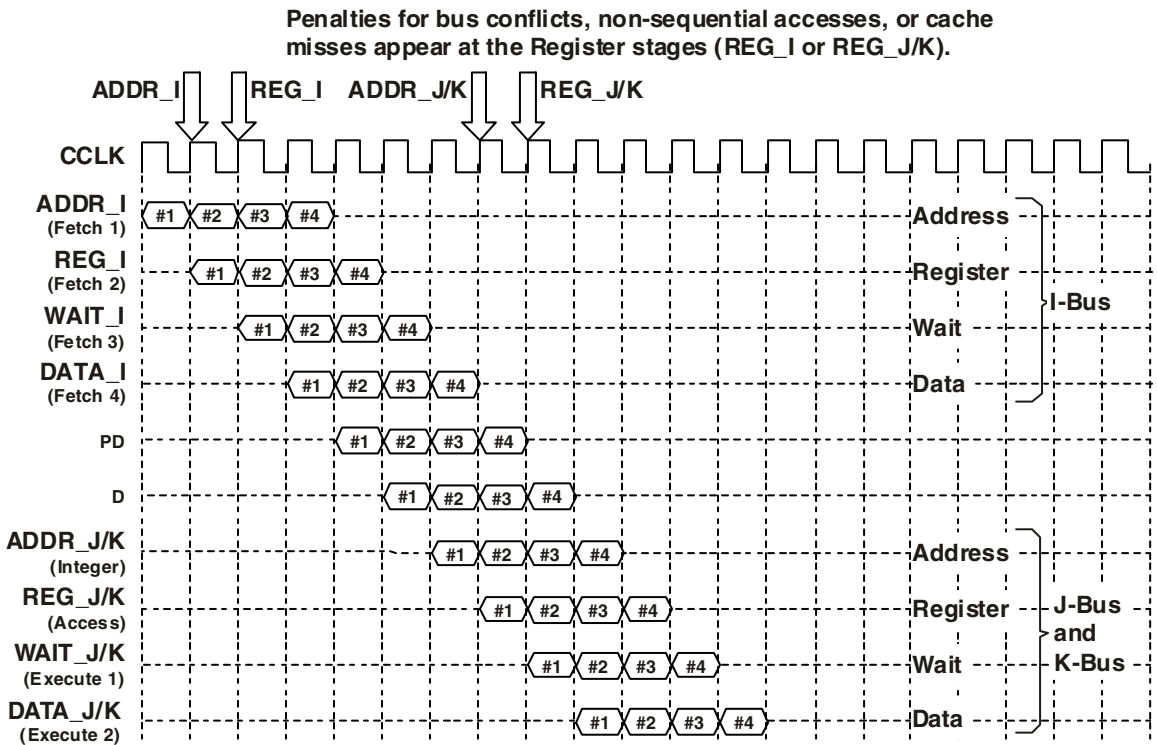



Figure 9-11. Memory Pipeline Stages

Access, Execute 1, and Execute 2 for loads/stores (J- and K-bus transactions). The S-Bus transactions (driven by the SOC interface) are detached from the core pipeline.

In the sections that follow, a range of memory transaction scenarios are presented using pipeline flow diagrams and transaction descriptions. Depending on the transaction and the data in the memory block's buffer/cache (prefetch buffer, read buffer, and cache), memory transactions can have penalty cycles. For a summary of these penalties, see

Memory Pipeline

“[Memory Access Penalty Summary](#)” on page 9-44. For suggestions and techniques for reducing penalties during memory access, see “[Memory Programming Guidelines](#)” on page 9-46.

 In the pipeline diagrams ([Figure 9-12 on page 9-39](#) and [Figure 9-13 on page 9-40](#)), the letter “S” indicates stall, and the letter “B” indicates bubble.

Cache Miss Transaction

The pipeline diagram in [Figure 9-11 on page 9-37](#) shows an example flow of instructions in the pipeline. The flow assumes no stalls.

A cache/buffer miss causes a stall on the register cycle on both the memory block that was accessed and on the bus that the transaction was accessed through. A block conflict (two or more buses access the same memory block) causes the lower priority buses to stall in the register cycle. Also, a new transaction that is targeted to a block that is stalled in the register cycle is stalled at the register cycle. When such a stall occurs on a J-bus or K-bus transaction, the instruction line following the instruction line that caused the stalled transactions is stalled, and all previous transactions are stalled – Pre-Decode, Decode and Integer. The instruction alignment buffer between Fetch 4 and Pre-Decode stages reduces the effect on the fetch stages.

Cache Miss (With No Buffer Hit) Transaction

On cache miss (assuming there is no prefetch buffer hit or read buffer hit), the transaction gets the miss indication, stalls at the Register cycle, and starts accessing the embedded DRAM.

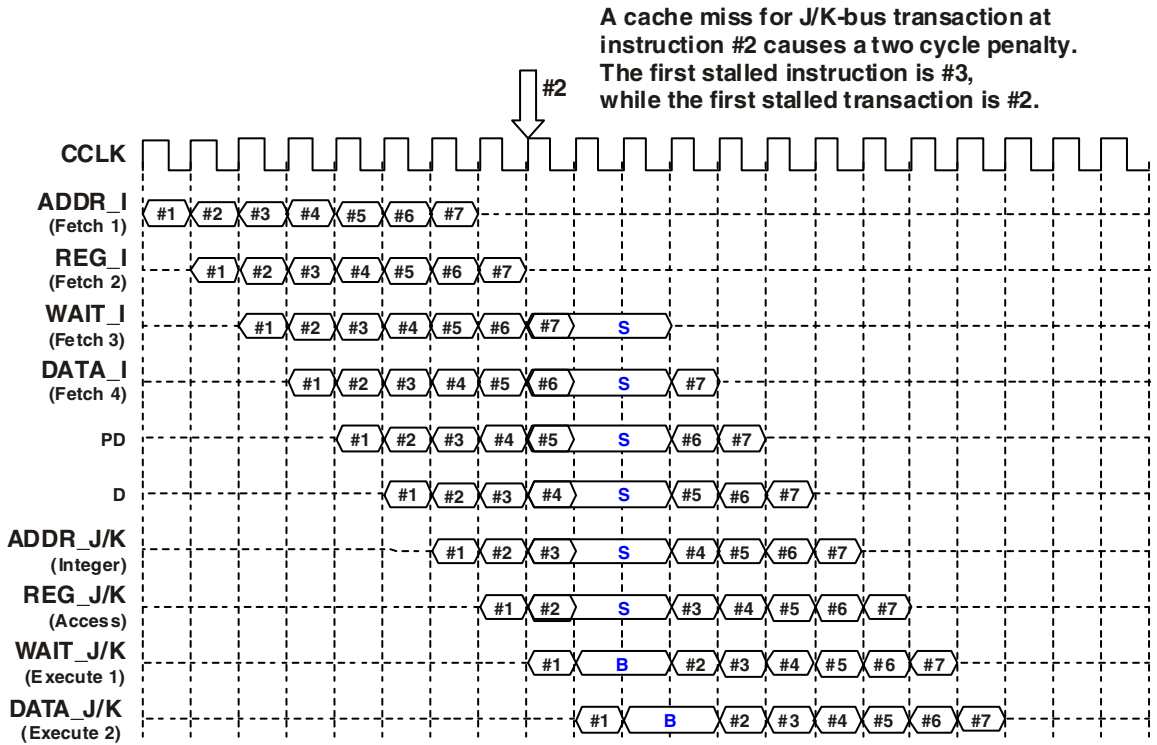


Figure 9-12. Memory Pipeline—Cache Miss, Prefetch Miss (Out-of-Order Access to Page Not Prefetched) Penalties

Memory Pipeline

The embedded DRAM access is 8 words of cache data. The data is put both in cache (if the data is cacheable) and in the read buffer. If the next transaction (transaction 3 in Figure 9-13) is to the same cache octal word or is a cache hit, there is no overhead on the access.

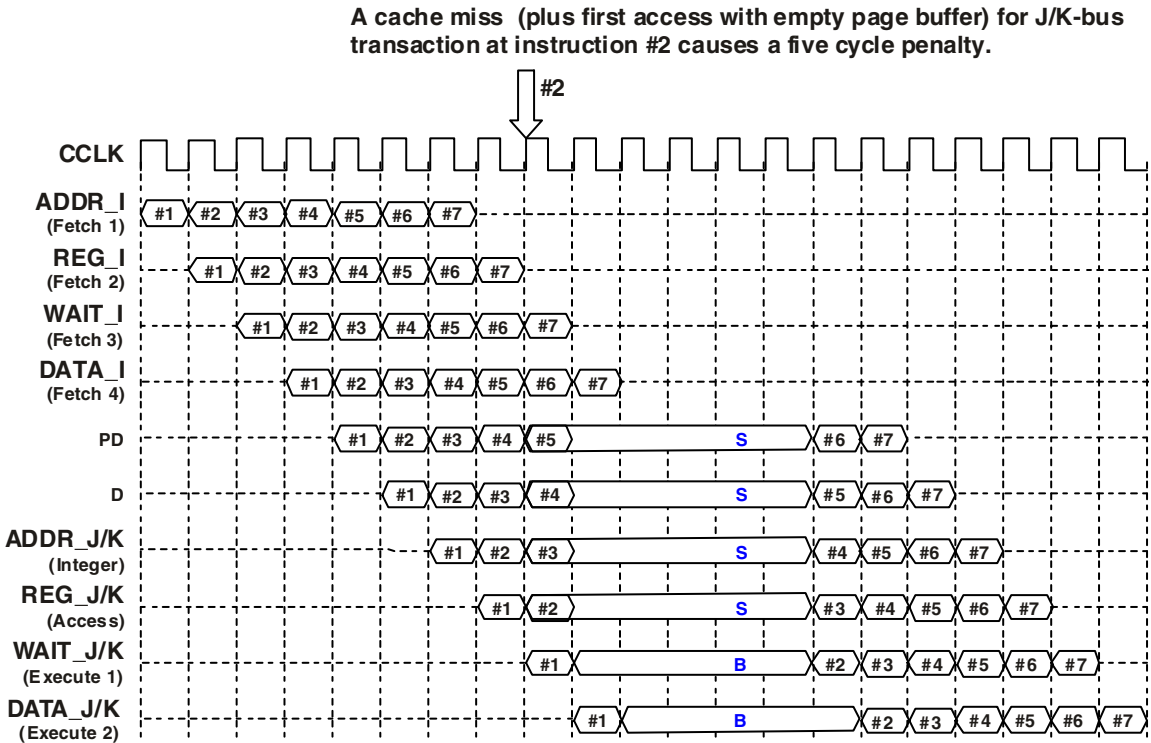


Figure 9-13. Memory Pipeline—Cache Miss, Prefetch Miss (First Access With Empty Page Buffer) Penalties

i Note that on all the transactions to the embedded DRAM there is an uncertainty of one cycle. The reason for the uncertainty is the interaction between the core clock (CCLK) and the embedded DRAM clock (DCLK).

Prefetch Sequence

Prefetch is a sequence that begins after the first read miss access in a given page. The prefetch direction depends on which internal bus is making the access and the sign of the address modifier. The prefetch direction is determined as follows:

- J-bus or K-bus (IALU accesses):
 - $U_{reg} = [J_m += J_n / imm];;$
direction according to J_n / imm sign (up for positive)
 - $U_{reg} = [J_m + J_n / imm];;$
no direction (no prefetch)
- I-bus (sequencer accesses): direction is always up
- S-bus (SOC interface accesses):
 - DMA accesses:
 - Regular DMA: direction according to dx sign (stride or X stride in two dimensional DMA)
 - Chained DMA: no direction (no prefetch)
 - Cluster bus: direction is always up

The prefetch sequence copies the page to the prefetch buffer from the read miss data to the end of the page. When the prefetched data is accessed by a transaction, there is a Prefetch buffer hit, and the data is transferred from the Prefetch buffer to the bus. If the access is cacheable, the data is also copied into the cache and in future accesses it is identified as a cache hit.

An interesting case occurs when the accesses are to the same page that is prefetched now, but not in the order of the access. In this case, there may be accesses that follow the prefetch and accesses that precede the prefetch. In the case that the transaction follows the prefetch, the sequence is simi-

Memory Pipeline

lar to the case described above. The data is kept in the Prefetch buffer until it is used (Prefetch buffer hit), and then it is transferred both to the bus and to the cache with no penalty. The case where an access precedes the data is a bit more complicated. If the prefetch buffer has already requested this data from the embedded DRAM, but has not yet received the data, one or two stall cycles are inserted. If the prefetch buffer did not yet access this data, the access is referred as a new miss, the page is closed and reopened. The penalty on this case is 7 to 8 stall cycles.

A page is prefetched as result of two events:

- Read miss – after the data is accessed, the rest of the page is prefetched.
- Cross page prefetch – when a page prefetch is completed, not interrupted and at least one data in this page has been used, the next page is prefetched.

When there is a new miss access that causes a page prefetch, the selection of which page to replace is the LRU page.

If a non-empty page is replaced, all the data of the old page is cleared from the prefetch buffer immediately, and another access to this page—if it is a cache miss—causes a miss and an embedded DRAM access.

When the prefetch of a page is completed, the memory controller may continue to the next page and prefetch it. The decision to continue to the next page is taken under these conditions:

- There were no out-of-order accesses to a different page in the same half-block (prefetch was not aborted).
- There was at least one bus access to the current page.
- The prefetch has not reached the half-block boundary or the block boundary.

When all the conditions above are true, the next page is prefetched. The prefetch flow can continue immediately after the current page prefetch is completed, but in some cases the next page prefetch can be delayed until there is a first bus access to the current page. When a prefetch flow reaches the end of a half-block, it wraps around.

Prefetch Sequence Interrupted

The prefetch accesses are executed as second priority to real transactions to the embedded DRAM (a new miss access). For this reason, a prefetch sequence is interrupted every time there is a need to read from the embedded DRAM because of a miss transaction in a different page. There are three different cases for the interruption consequences according to the sub-arrays which are targets of the new transaction and the interrupted transaction. The two transactions may be in the same sub-array, different sub-arrays in the same half-block or in the two different half-blocks.

In the first case, the overhead is maximal. When the interrupting transaction occurs, the prefetch buffer stops the current prefetch sequence, precharges the current page, then activates the new page in order to read it. In this case, the direct penalty is 7–8 cycles depending on the embedded DRAM clock phase that was accessed. Also, there is an indirect penalty. If the program returns to the first page later, the page access prefetch sequence must be restarted with full miss penalty.

In case the new miss access is not in the same sub-array, but in the same half-block, the flow is very similar, except for the fact that precharge of the old page and activation of the new page can be executed in parallel. This saves two penalty cycles, but still the prefetch sequence is aborted. In case the program returns to the first page later (page access), prefetch sequence must be restarted with full miss penalty. The direct penalty here is 5–6 cycles.

Memory Pipeline

If the new sequence is in the second half-block, the overhead is similar to a normal miss, and in some cases it is recommended to work this way. In this case, the prefetch sequence is not aborted by the new access. The prefetch buffer only pauses. If there is a new access to the paused page the content of the prefetch buffer, the embedded DRAM page buffer and the state of the prefetch sequence are still kept and the sequence is resumed from the paused point. This way the program may interleave two sequences. After the first miss in each of the pages, all accesses are prefetch buffer hits or read buffer hits.

Memory Access Penalty Summary

[Figure 9-14](#) shows a high level summary of memory transaction penalties and identifies where these penalties occur in a sequence.

These penalties (stalls) occur only when there is a cache miss, prefetch buffer miss, or read buffer miss. If there is a cache hit, prefetch buffer hit, or read buffer hit, there are no penalties (stalls) for the transaction.

The penalty-free transactions include:

- Cache miss on read:
 - Sequential accesses in the direction defined by the prefetch rules (prefetch buffer hit, no stalls)
 - Sequential continuation to the next page (prefetch buffer hit, no stalls)
 - Second access in the same 8-word unit (read buffer hit, no stalls)
- Cache miss on write:
 - The replaced way in cache has its dirty bit cleared (no stalls)
 - The copyback buffer is not full (no stalls)

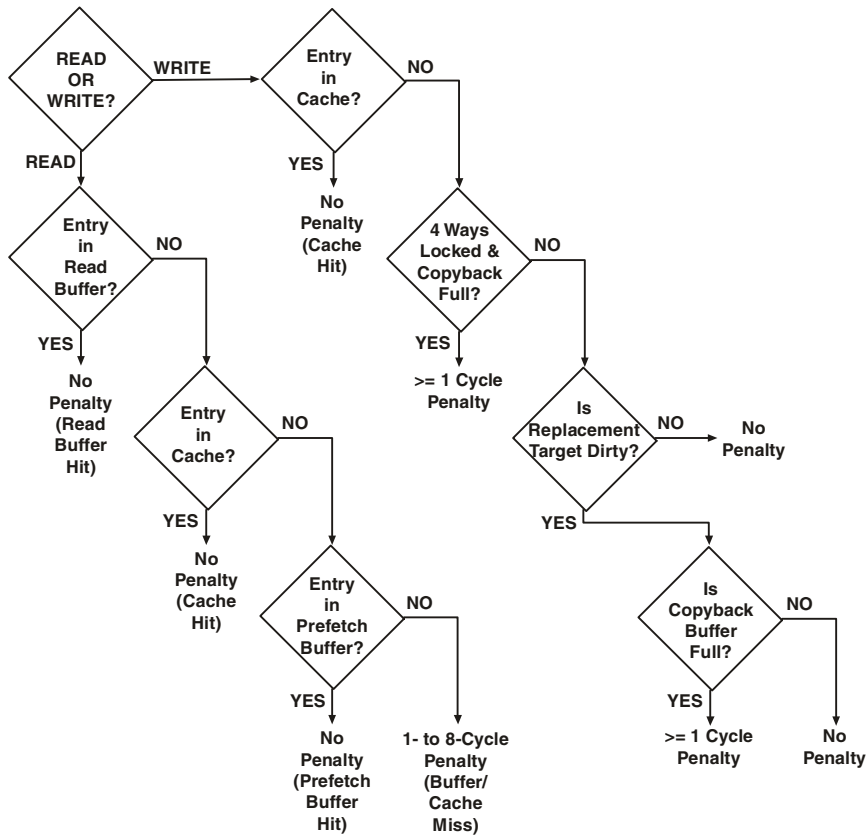


Figure 9-14. Memory Access Penalty Summary¹

¹ This is a high level, approximate summary of buffer/cache hit miss penalties. For precise penalty analysis, use the simulator or emulator development tools.

Memory Programming Guidelines

The penalty-laden transactions include:

- Cache miss on read:
 - First access when page buffer was previously empty (5–6 penalty cycles)
 - First access when interrupting a prefetch access to a different page in the same sub-array (7–8 penalty cycles)
- Cache miss on write:
 - The replaced way in cache has its dirty bit set and the copy-back buffer is almost full (≥ 1 penalty cycles)¹
 - All four ways in the cache set are locked and the copyback buffer is almost full (≥ 1 penalty cycles)¹
- Write followed by read to the same octal word; aligned 8-word unit (4 penalty cycles)

Memory Programming Guidelines

The purpose of a memory system with a cache is to have a large and relatively slow memory—the embedded DRAM operates at one-half the speed of the processor core—look like a fast memory. One technique uses the cache to keep the data that has been used, assuming this data shall be used again in the near future. Another technique uses the prefetch to anticipate which data is used soon and access it in advance. This memory system combines the two techniques. To maximize the performance of the embedded DRAM system, it is important to understand the different cache and prefetch mechanisms and manage the program and data flow accordingly.

¹ If the write occurs while the embedded DRAM has a different page open in the same sub-array, two extra penalty cycles occur.

The memory is partitioned into six blocks, and each block has its own cache and cache control logic. The caches in the different blocks work independently. All the guidelines in this section focus on data management in a single block. Each block may be operated differently with different flow and different controls (for example, one cache may be locked while others are not).

There are some effective techniques that work well with the cache:

- [Sequential Access Policy](#)
- [Local Access Policy](#)
- [Instruction Caching](#)
- [Double Sequence Flow](#)
- [Block Orientation](#)

The sequential access policy uses the prefetch mechanism. This technique works best for processing data blocks that are larger than the cache size.

The local access policy is effective when the same data is used again and again, and the order of access is not sequential. In case of local access policy, the data block must be smaller than the cache size (4K words).

The most ineffective way to use a cache with local access policy is to process data blocks that are a bit larger than the cache size, because then the access penalty is paid at the first access, and the data is replaced before the second access. This pattern is *cache thrashing*—repeated emptying and filling of the cache. In these cases, programs should try to break the data into smaller pieces that are distributed in a few blocks or use the sequential access policy.

Sequential Access Policy

This technique is effective on large blocks of data even if the data is used only once. The programming orientation differs for read, write, and read-modify-write flows. Although the sequential access policy works best for a single flow in each block, it can work with two interleaved flows if each of the data blocks is in a different half-block.

Any access that is sequential in address to the previous access is expected to be a prefetch buffer hit or a read buffer hit, and there is no penalty on these accesses. The prefetch flow is interrupted only by accesses to the same half-block that miss the cache, the read buffer, and the prefetch buffer. Miss access to the other half-block only pauses the prefetch sequence. The penalty of this flow break is much less, but not always zero.

Cache hit accesses in the middle of sequential access flow have zero penalties. To use the sequential access policy effectively, programs should:

- Avoid conflicting or multiple streams of accesses to the same block.
- Make maximum usage of the read buffer:
 - Try to access the same cache data (aligned 8-word unit) at least twice in a sequence (although, between the two accesses to the same cache data, an access to the second half-block doesn't invalidate the read buffer, but uses the second read buffer).
 - Make sure every second access is either a prefetch buffer hit or a read buffer hit.
 - If there is only one access to each way's cache data, try to access the same block once every two or more cycles.
 - If the accesses are only once to each data element, define the master that is executing the accesses (read or write) as non-cacheable.

A second access to data that has been previously accessed in sequential flow without penalties is possible under the following conditions:

- The master that is executing the accesses must be defined as cacheable.
- The non-sequential accesses are to data that has already been fetched before.
- The distance between the point in the sequential flow and the non-sequential data access is less than the cache size (4K addresses for a single flow, 2K for dual flow to the two half-blocks). Otherwise the non-sequential data is probably already replaced.

When executing the double sequence flow it is recommended to run the two flows symmetrically—two or more accesses to the first data block in the first half-block, and two or more accesses to the second data block in the second half-block.

Local Access Policy

This technique is effective when the data block is smaller than the cache size (4K words). When keeping this rule, the only penalty on the accesses is on the first access and at the replacement (for a written or modified data block). If the penalty of first load is significant and needs to be minimized, it can be done in these ways:

- Try to use the sequential access policy for the first access.
- Try to preload the data into the cache in advance. It can be done either by the `CACHE_INIT` command, or by loading the data via the cache (assuming this data block is loaded via a DMA).

If the block was written or modified, there is overhead when it is replaced. To avoid these unexpected penalties on replacements, use memory system commands that work in the background. The `CACMD_CB` (cache copyback)

Memory Programming Guidelines

command can be used for cleaning the cache dirty bits, and the `CACMD_INV` (cache invalidate) command may be used to clear data that is irrelevant or locked.

Another tool that can reduce overhead when using the local access policy is the cache lock mechanism. If a certain data block is very critical, it can be locked when it is loaded for the first time. From this point on, it is not replaced until the cache is invalidated. Programs should use the cache lock carefully, because the locked data may obstruct the cache replacement policy and because the locked data is cleared from the cache only when invalidated.

Cache Usage Policy

Managing each memory block's cache can greatly improve system performance. The cache usage policy in this section provides guidelines for using the cache enable, disable, set cacheability, and instruction caching features.

Cache Enable (After Reset and Boot)

After reset, the caches are all disabled and there is no guarantee of the status of the tag and LRU values. In order to have the cache ready to work, all valid bits must be cleared and all four tags in every index must have four different values. Do this through cache invalidation of the whole cache range (`index=0`, `length=127`).

Cache Disable

The main purposes of the cache disable mode are initialization after reset and diagnostics. In both cases, the mode is only changed from disable to enable. In case of changing mode from enable to disable, there are few complications.

When cache is disabled, it is not updated further in case of memory write. If there are cache ways with the dirty bit set, the program gets the wrong value in the embedded DRAM and does not get the correct value in cache.

In order to keep the memory system coherent, there are few recommendations for mode change. Before disabling the cache, if there is valid data in the cache, it should be written back. Use the command `CACMD_CB` on the full cache. During the execution of this procedure, do not access this memory block.

After the cache is copied back to memory, it can be disabled. Before it is enabled again, the cache needs to be invalidated. This can be done by executing the command `CACHE_INV` on the full cache.

Read and Write Cacheability


Programs can define for each master (I-, J-, K-, and S-buses) whether its transactions are cacheable per direction (read or write). The purpose of this feature is to enable different policies working in parallel to each other. A simple example is a data buffer that is being loaded from I/O and used in parallel (half new data buffer is loaded from I/O and the old half buffer is being processed by the core). In this case, any DMA (I/O) access to the memory may write to the cache, and this may replace data that is being used by the core. In such a case, the best configuration would be to define the DMA as non-cacheable and define the core accesses as cacheable. Similar manipulations may be used if the J- and K-buses are accessing (at different times) the same memory block for different purposes. If the J-bus is accessing sequentially, while the K-bus is accessing locally, the J-bus should be defined as non-cacheable, while the K-bus should be defined as cacheable.

Instruction Caching

Caching the instructions may be as critical for execution as caching the data, and it is easier to manage instruction caching than data caching. For instruction caching, programs use all the sequential and local access tech-

Memory Programming Guidelines

niques. Normally, linear code is executed as single flow sequential access. Loops are executed as local access, after the first sequential run-through. In some cases, cache lock may also be used.

 It is strongly recommended to keep instructions in a separate block from the data that is accessed by the same code. Do not mix a group of instructions in a block with data that is used by that group of instructions.

Double Sequence Flow

The double sequence flow technique may be used in two different ways. One way is for a program that accesses two vectors. The other way is to run the program on one flow while the DMA is paging a data block from the second half.

In order to use the double sequence, apply the rules of sequential access for each of the two flows. The two flows can be interleaved in any possible way. The duplication of the prefetch buffer, copyback buffer, and read buffer assures that the prefetch flows of each of the two sequences work according to the rules of a single flow. The only exception is that two accesses to the same memory in a single cycle (even if it is in the two different half-blocks) are serialized.

Block Orientation

Although the bus implementation is different, the code optimization technique for placing blocks of data with memory blocks is identical as on previous TigerSHARC processors. It is recommended in applications to dedicate one memory block for the code and at least two other memory blocks for data. If the program is making maximum use of both IALUs for the memory accesses, in any given cycle each IALU should access a separate memory block.

System accesses (DMA or external masters on the cluster bus) may be targeted either to the same blocks that the code is using or to different blocks. It is simpler and safer to target the system accesses to different blocks, but this may limit the number of blocks that can be used by the processor core. If the system should access the same memory that the core is using, make sure of the following:

1. The full bandwidth of the core and system to this port is less than one transaction every cycle.
2. If both core and system accesses are for data read, try to work in the dual sequence technique. Keep each of the two data blocks in separate half-blocks. If the sequences are long enough, consider working in cache disable mode.
3. If the local access technique is used, make sure that the cache (4K words) is large enough for both data blocks.

If one of the accesses (core or I/O) is based on sequential accesses and the other is based on local accesses, define the sequential as non-cacheable and define the other as cacheable.

4. In any case, keep the system accesses in lower priority than core accesses. This way the system accesses wait in the SOC interface FIFO until there is a free cycle on the block, and the core is not stalled.

Memory Access Examples

[Listing 9-1](#) provides examples of memory system command usage. The comments with the instructions identify the key features of the commands, such as setting embedded DRAM refresh rate and enabling cache.

[Listing 9-2](#) provides an example linker description file (LDF) for an ADSP-TS201 processor system. (For information on LDF syntax, see the *VisualDSP++ Linker and Utilities Manual for TigerSHARC Processors*.)

Memory Access Examples

This LDF demonstrates how to define *memory segments* (the software equivalent of hardware memory blocks and banks) and shows how to place code or data sections within the segments.

Listing 9-1. Memory System Command Examples

```
CACMDB = CACMD_REFRESH | 750;;
/* set the refresh rate for embedded DRAM to every 750 cycles */
CACMDB = 0x00000000;;
/* enable the cache associated with each memory block */
CACMDB = 0x1C3F8000;;
/* set cache/bus enable associated with each memory block */
```

Listing 9-2. ADSP-TS201 Linker Description File (LDF) Example

```
ARCHITECTURE(ADSP-TS201)
SEARCH_DIR( $ADI_DSP\TS\lib )
$OBJECTS = $COMMAND_LINE_OBJECTS;
MEMORY{
  M0Code {TYPE(RAM) START(0x00000000) END(0x0001FFFF) WIDTH(32)}
  M2DataA{TYPE(RAM) START(0x00040000) END(0x0004FFFF) WIDTH(32)}
  M2DataB{TYPE(RAM) START(0x00050000) END(0x0005FFFF) WIDTH(32)}
  M4DataA{TYPE(RAM) START(0x00080000) END(0x0008FFFF) WIDTH(32)}
  M4DataB{TYPE(RAM) START(0x00090000) END(0x0009FFFF) WIDTH(32)}
  M6DataA{TYPE(RAM) START(0x000C0000) END(0x000CFFFF) WIDTH(32)}
  M6DataB{TYPE(RAM) START(0x000D0000) END(0x000DFFFF) WIDTH(32)}
  M8DataA{TYPE(RAM) START(0x00100000) END(0x0010FFFF) WIDTH(32)}
  M8DataB{TYPE(RAM) START(0x00110000) END(0x0011FFFF) WIDTH(32)}
  M10DataA{TYPE(RAM) START(0x00140000) END(0x0014FFFF) WIDTH(32)}
  M10DataB{TYPE(RAM) START(0x00150000) END(0x0015FFFF) WIDTH(32)}
  /* segment definitions for external memory banks */
  MS0    {TYPE(RAM) START(0x30000000) END(0x37FFFFFF) WIDTH(32)}
  MS1    {TYPE(RAM) START(0x38000000) END(0x3FFFFFFF) WIDTH(32)}
  MSSD0  {TYPE(RAM) START(0x40000000) END(0x43FFFFFF) WIDTH(32)}
```

```
MSSD1 {TYPE(RAM) START(0x50000000) END(0x53FFFFFF) WIDTH(32)}
MSSD2 {TYPE(RAM) START(0x60000000) END(0x63FFFFFF) WIDTH(32)}
MSSD3 {TYPE(RAM) START(0x70000000) END(0x73FFFFFF) WIDTH(32)}
} /* segment definitions for internal memory blocks */

PROCESSOR p0{
  OUTPUT( $COMMAND_LINE_OUTPUT_FILE )
  SECTIONS{
    code{
      FILL(0xb3c00000)
      INPUT_SECTION_ALIGN(4)
      INPUT_SECTIONS( $OBJECTS(program))
      . = . + 10 ; /* keeps uninitialized memory out of pipe */
    } >M0Code
    data1 { INPUT_SECTIONS( $OBJECTS(data1)) } > M4DataA
    data2 { INPUT_SECTIONS( $OBJECTS(data2)) } > M8DataA
    data1a{ INPUT_SECTIONS( $OBJECTS(data1a)) } > M2DataA
    data1b{ INPUT_SECTIONS( $OBJECTS(data1b)) } > M2DataB
    data2a{ INPUT_SECTIONS( $OBJECTS(data2a)) } > M4DataA
    data2b{ INPUT_SECTIONS( $OBJECTS(data2b)) } > M4DataB
    data3a{ INPUT_SECTIONS( $OBJECTS(data3a)) } > M6DataA
    data3b{ INPUT_SECTIONS( $OBJECTS(data3b)) } > M6DataB
    data4a{ INPUT_SECTIONS( $OBJECTS(data4a)) } > M8DataA
    data4b{ INPUT_SECTIONS( $OBJECTS(data4b)) } > M8DataB
    data5a{ INPUT_SECTIONS( $OBJECTS(data5a)) } > M10DataA
    data5b{ INPUT_SECTIONS( $OBJECTS(data5b)) } > M10DataB
  } /* end code and data section assignments to segments */
} /* end processor p0 */
```

Memory System Command Summary

[Listing 9-3](#) shows the memory system commands' syntax. The conventions used in these listings for representing register names, optional items, and choices are covered in detail in [“Memory System Commands” on page 9-58](#) and [“Register File Registers” on page 2-6](#).

Listing 9-3. Memory System Commands

```
/* These commands use values from the DEFTS201.H file. In all of
these commands, the Js is any IALU data register (J30-0 or
K30-0), and the x in CACMDx or CCAIRx is the block number: 0, 2,
4, 6, 8, 10, or B (for broadcast) */

Js = (CACMD_REFRESH | Refresh_Rate) ;
CACMDx = Js ;
/* where Refresh_Rate is an immediate 13-bit value */

Js = CACMD_EN ;
CACMDx = Js ;

Js = CACMD_DIS ;
CACMDx = Js ;

Js = (CACMD_SET_BUS | I_Caching | S_Caching | J_Caching | K_Caching) ;
CACMDx = Js ;
/* where _Caching selects caching for a bus's transactions as
none (no transactions cache), read-only, write-only, or
read-write (all transactions cached) */

Js = CACMD_SLOCK ;
CACMDx = Js ;

Js = CACMD_ELOCK ;
CACMDx = Js ;
```

```

Js = Start_Address ;
CCAIRx = Js ;
Js = (CACHE_INIT | (Length << CACMD_LEN_P) {| CACMD_NOSTALL});
CACMDx = Js;
Js = (CACHE_INIT_LOCK | (Length << CACMD_LEN_P) {|
CACMD_NOSTALL});
CACMDx = Js;
/* where Start_Address is a 14-bit address, using bits 16-3 (bits
2-0 are ignored), Length is a value up to the size of the cache,
and Stall_Mode is CACMD_NOSTALL (for STALL) or omitted (STALL) */

Js = Start_Index ;
CCAIRx = Js ;
Js = (CACMD_CB | (Length <<CACMD_LEN_P) {| CACMD_NOSTALL} ) ;
CACMDx = Js ;
/* where Start_Index is a 7-bit index, using bits 16-10 (bits 9-0
are ignored), Length is a value ranging from 0 to 511, and
Stall_Mode is CACMD_NOSTALL (no STALL) or omitted (STALL) */

Js = Start_Index ;
CCAIRx = Js ;
Js = (CACMD_INV | (Length <<CACHE_LEN_D) {| CACMD_NOSTALL}) ;
CACMDx = Js ;
/* where Start_Index is a 7-bit index, using bits 16-10 (bits 9-0
are ignored), Length is a value ranging from 0xFF8000 to 0x8000,
Stall_Mode is CACMD_NOSTALL (no STALL) or omitted (STALL) */

/* If the CACMDx value and CCAIRx value are written into a double
register (for example, CACMDx in J0 and CCAIRx in J1), a double
register transfer loads both values at once. As in:
Jm = CACMDx_value ; /* low register */
Jn = CCAIRx_value ; /* high register */
CACMDx = Jmd ;      /* long word transfer */

```

Memory System Commands

The memory system commands are not instructions. Rather, these commands are applied by loading a value into a memory block's cache/memory command (CACMDx) register and cache address/index (CCAIRx) register (if applicable). Loading a value that corresponds to a command executes a configuration operation of internal memory and cache. These commands include:

- [Refresh Rate Select](#)
- [Cache Enable](#), [Cache Disable](#), and [Set Bus/Cacheability \(for Bus Transactions\)](#)
- [Cache Lock Start](#) and [Cache Lock End](#)
- [Cache Initialize \(From Memory\)](#), [Cache Copyback \(to Memory\)](#), and [Cache Invalidate](#)

For a complete list of the bit values that correspond to commands in the CACMDx and CCAIRx registers, see [“DEFTS201.H – Register and Bit #Defines File”](#) on page B-1 or [Table 9-2](#) on page 9-24. The command executes when the value is written to the CACMDx register, so load the CCAIRx register first for commands that use both registers.

The CACMDx and CCAIRx registers may be accessed only using the *Ureg = <imm>*; and *Ureg = Ureg*; instructions. And, the *Ureg* must be a processor core register. Only the processor core can access the CACMDx, CCAIRx, and CASTAT registers.



Because the memory system commands are applied by loading values into the CACMDx and CCAIRx registers, a set of predefined macros can make it easier to apply these values through assembler expressions. The instruction reference pages for these commands use the following predefined macros from the [“DEFTS201.H – Register](#)

and [Bit #Defines File](#)” on page B-1: CACMD_EN, CACMD_DIS, CACMD_SLOCK, CACMD_ELOCK, CACMD_CB, CACMD_INV, CACHE_INIT, CACHE_INIT_LOCK, CACMD_REFRESH, and CACMD_SET_BUS.

The conventions used in these reference pages for representing register names, predefined macros, and assembler expressions differ from those described for other reference pages. Briefly, these conventions are:

- *CACMD_x* – the italicized labels indicate replaceable items; these labels are replaced with register names or immediate values in the instruction syntax.
- CACMD_REFRESH – the non-italicized (roman) labels indicate register names or assembly macros that are not replaceable; these items are part of the instruction syntax.
- (|) – the vertical bar within parentheses indicate a bit wise logical OR of the values in an assembler expression; the bar is part of the instruction syntax.
- { } – the curly braces enclose options; these braces are not part of the instruction syntax.



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 1-23 and [“Instruction Parallelism Rules”](#) on page 1-27.

Memory System Commands

Refresh Rate Select

Syntax

```
Js = (CACMD_REFRESH | Refresh_Rate) ;  
CACMDx = Js ;  
/* where Js is any IALU data register (J30-0 or K30-0),  
Refresh_Rate is an immediate 13-bit value, and x in CACMDx is the  
block number: 0, 2, 4, 6, 8, 10, or B */
```

Function

This memory system command sets the refresh counter value to the value in bits 13–0 of the memory block’s CACMD*x* register. In addition to read and write transactions, the embedded DRAM interface must periodically refresh each page of memory. Refreshing a page is equivalent to activating then precharging the page. The refresh process is required to retain accurate data in memory cells. Programs manage refresh by setting the refresh period for each memory block with the memory refresh command (applied using the CACMD*x* register). After this value is set, the refresh takes place automatically without interrupting a page access—page prefetch or copyback sequences within the same page are completed before refresh. The value of *Refresh_Rate* should be selected according to CCLK frequency, for example:

- 0x384 for 900 cycles (600 MHz CCLK)
- 0x2EE for 750 cycles (500 – 599 MHz CCLK)
- 0x258 for 600 cycles (400 – 499 MHz CCLK)
- 0x1C2 for 450 cycles, default value (300 – 399 MHz CCLK)

Example

```
J0 = (CACMD_REFRESH | 0x1C2) ;;
CACMDB = J0 ;;
/* set the refresh rate to 450 cycles for all blocks */
/* uses CACMD_REFRESH value from DEFTS201.H file */
```

Memory System Commands

Cache Enable

Syntax

```
Js = CACMD_EN ;  
CACMDx = Js ;  
/* where Js is any IALU data register (J30-0 or K30-0) and x in  
CACMDx is the block number: 0, 2, 4, 6, 8, 10, or B */
```

Function

This memory system command enables the cache in each memory block. When the cache is enabled, every access to the memory can add data to the cache, and the cache checks access for cache hits/misses.



The cache must be enabled before most memory system commands can work.

Example

```
J0 = CACMD_EN ;;  
CACMDB = J0 ;;  
/* enable the cache on all blocks */  
/* uses CACMD_EN value from DEFTS201.H file */
```

Cache Disable

Syntax

```
Js = CACMD_DIS ;
CACMDx = Js ;
/* where Js is any IALU data register (J30-0 or K30-0) and x in
CACMDx is the block number: 0, 2, 4, 6, 8, 10, or B */
```

Function

This memory system command disables the cache in each memory block. When the cache is disabled, accesses to the memory do not go through the cache in each memory block. The cache does not check for cache hits and does not update cache data.



The cache must be enabled before most memory system commands can work.

In cache disable mode, the prefetch and copyback buffers operate normally. The copyback buffer gets the data of every write transaction.

If the cache has dirty data when it is disabled, this data may be lost. If the program needs to disable cache during normal work and the block's cache has dirty data, the program should run the [Cache Copyback \(to Memory\)](#) command (see discussion [on page 9-74](#)) before disabling the cache.

Example

```
J0 = CACMD_DIS ;;
CACMDB = J0 ;;
/* disable the cache on all blocks */
/* uses CACMD_DIS value from DEFTS201.H file */
```

Memory System Commands

Set Bus/Cacheability (for Bus Transactions)

Syntax

```
Js = (CACMD_SET_BUS | I_Caching | S_Caching | J_Caching | K_Caching);
CACMDx = Js ;
/* where Js is any IALU data register (J30-0 or K30-0), _Caching
selects caching for a bus's transactions as none (no transactions
cache), read-only, write-only, or read-write (all transactions
cached), and x in CACMDx is the block number: 0, 2, 4, 6, 8, 10,
or B */
```

Function

This memory system command enables or disables caching for transactions on each bus (I-, S-, J-, and K-bus) to or from a memory block's cache. At reset, caching ability is enabled for all read/write transactions on all memory blocks (settings in the `CASTAT_I_CACHING`, `CASTAT_S_CACHING`, `CASTAT_J_CACHING`, and `CASTAT_K_CACHING`, bits in the `CASTAT` register). The parameters for this command include selections for *I_Caching*, *S_Caching*, *J_Caching*, and *K_Caching*.

I_Caching

Use the following input for this field:

- `CACMD_I_BUS_R_`
cache I-bus read transactions (instruction fetch)
- `CACMD_I_BUS_N_`
no caching for I-bus transactions (no instruction cache)

S_Caching

Use the following input for this field:

- CACMD_S_BUS_RW
cache S-bus read and write transactions
- CACMD_S_BUS_R_
cache S-bus read transactions
- CACMD_S_BUS_W_
cache S-bus write transactions
- CACMD_S_BUS_N_
no caching for S-bus transactions

J_Caching

Use the following input for this field:

- CACMD_J_BUS_RW
cache J-bus read and write transactions
- CACMD_J_BUS_R_
cache J-bus read transactions
- CACMD_J_BUS_W_
cache J-bus write transactions
- CACMD_J_BUS_N_
no caching for J-bus transactions

K_Caching

Use the following input for this field:

- CACMD_K_BUS_RW
cache K-bus read and write transactions
- CACMD_K_BUS_R_
cache K-bus read transactions

Memory System Commands

- CACMD_K_BUS_W_
cache K-bus write transactions
- CACMD_K_BUS_N_
no caching for K-bus transactions



The cache must be enabled before the set bus/cacheability command has any affect.

When reading data from the embedded DRAM memory while the cache is dirty, cache replacement is needed. These cache replacement operations cause the copyback buffer to be filled, and (soon) an arbitration conflict develops between the copyback buffer (which needs to empty the data into the embedded DRAM) and the prefetch buffer (which needs to fetch data from the embedded DRAM). This arbitration conflict between the copyback and prefetch buffers may cause several stalls.

Because there are many applications that use sequential data only once, it is possible to improve the performance of these applications by controlling the prefetch mechanism for each bus and memory block. The set bus/cacheability command provides this control of the bus, selecting whether transactions on a bus to a selected memory block is or is not put into the block's cache.

If the bus's transactions to a block are not set to be cached, the data from those transactions does not go into the cache, and there is no cache replacement. The copyback buffer does not fill, and the prefetch mechanism is not delayed with arbitration conflicts between the copyback and prefetch buffers.

The S-, J-, and K-buses can be set to cache read transactions only, cache write transactions only, cache read and write transactions, or cache no transactions. The I-bus is used only for reads and can be set to cache read transactions (instruction caching) or cache no transactions.

If cacheability is disabled but a transaction gets a cache hit, the data is updated in the cache. The update differs for read versus write transactions:

- When cacheability on read is disabled, a read transaction that hits the cache is served from the cache, as usual. A transaction that misses the cache is served from either the embedded DRAM, the prefetch buffer, or the read buffer, but neither a cache allocation nor a replacement takes place.
- When cacheability on write is disabled, a write transaction that causes a cache tag hit updates the relevant words of the cache data, as usual. A write transaction that misses the cache is written into the copyback buffer only.

Example

```
J0 = ( CACMD_SET_BUS   | CACMD_I_BUS_R_ | CACMD_S_BUS_N_
      | CACMD_J_BUS_N_ | CACMD_K_BUS_N_ ) ;
CACMD0 = J0;
/* sets caching ability for memory block 2 to cache I-Bus reads
and not to cache S-, J-, and K-Bus transactions. This is instruc-
tion caching only. */

J1 = ( CACMD_SET_BUS   | CACMD_I_BUS_R_ | CACMD_S_BUS_RW
      | CACMD_J_BUS_RW | CACMD_K_BUS_RW ) ;
CACMD2 = J1;
/* sets caching ability for memory block 0 to cache I-Bus reads,
S-Bus reads/writes, J-Bus reads/writes, and K-Bus reads/writes.
This is the reset state. */

J2 = ( CACMD_SET_BUS   | CACMD_I_BUS_N_ | CACMD_S_BUS_N_
      | CACMD_J_BUS_RW | CACMD_K_BUS_RW ) ;
CACMD4 = J2 ;
/* sets caching ability for memory block 4 to cache J- and K-Bus
reads/writes and not to cache I- or S-Bus transactions. This is
internal data caching only. */
```

Memory System Commands

Cache Lock Start

Syntax

```
Js = CACMD_SLOCK ;  
CACMDx = Js ;  
/* where Js is any IALU data register (J30-0 or K30-0) and x in  
CACMDx is the block number: 0, 2, 4, 6, 8, 10, or B */
```

Function

This memory system command starts the cache lock in each memory block. When the cache lock is on, every access to the buffer/cache or memory block—except for a read buffer hit—sets the lock bit of the affected cache way. The cache way is locked if the access was a hit (address and/or data are already in the cache) or if the access was a miss (the address and/or data are not in the cache and there is a need to perform replacement). After a way is locked, there is no replacement of that way. To clear the lock bit of the entry, use the [Cache Invalidate](#) command (see discussion on [page 9-78](#)).



The cache must be enabled before the Cache Lock Start command can work.

Example

```
J0 = CACMD_SLOCK ;;  
CACMDB = J0 ;;  
/* start the cache lock on all blocks */  
/* uses CACMD_SLOCK value from DEFTS201.H file */
```

Cache Lock End

Syntax

```
Js = CACMD_ELOCK ;
CACMDx = Js ;
/* where Js is any IALU data register (J30-0 or K30-0) and x in
CACMDx is the block number: 0, 2, 4, 6, 8, 10, or B */
```

Function

This memory system command ends the cache lock in each memory block. When the cache lock is off, every access to the buffer/cache or memory block—including a read buffer hit—does not set the lock bit of the affected cache way. If the affected cache way is already locked, it remains locked until the cache way is invalidated.



The cache must be enabled before the Cache Lock End command can work.

Example

```
J0 = CACMD_ELOCK ;;
CACMDB = J0 ;;
/* end the cache lock on all blocks */
/* uses CACMD_ELOCK value from DEFTS201.H file */
```

Memory System Commands

Cache Initialize (From Memory)

Syntax

```
Js = Start_Address ;  
CCAIRx = Js ;  
  
Js = (CACHE_INIT | (Length << CACMD_LEN_P) {| CACMD_NOSTALL});  
CACMDx = Js;  
  
Js = (CACHE_INIT_LOCK | (Length << CACMD_LEN_P) {|  
CACMD_NOSTALL});  
CACMDx = Js;  
  
/* where Js is any IALU data register (J30-0 or K30-0),  
Start_Address is a 14-bit address, using bits 16-3 (bits 2-0 are  
ignored), Length is a value up to the size of the cache,  
Stall_Mode is CACMD_NOSTALL (for no STALL) or omitted (for  
STALL), and x in CCAIRx or CACMDx is the block number: 0, 2, 4, 6,  
8, 10, or B */  
  
/* If the CACMDx value and CCAIRx value are written into a double  
register (for example, CACMDx in J0 and CCAIRx in J1), a double  
register transfer loads both values at once. As in:  
Jm = CACMDx_value ; /* low register */  
Jn = CCAIRx_value ; /* high register */  
CACMDx = Jmd ;      /* long word transfer */
```

Function

This memory system command loads data from memory blocks into each block's cache. This command prepares data in cache in order to avoid the first miss penalty. The parameters for this command include the *Start_Address* in memory, the *Length* of the data, and the *Stall_Mode* for the process.

<i>Start_Address</i>	Load this value into the corresponding CCAIRx register before loading the CACMDx register with the command. This value is an address relative to the start address of the memory block. For example, a <i>Start_Address</i> value of 0x0ABC8 in CCAIR4 corresponds to address 0x8ABC8 (address 0x0ABC8 relative to the beginning of block 4). The <i>Start_Address</i> is a 14-bit field using bits 16–3 (bits 31–17 and 2–0 are ignored, assumed zero; programs may use J-IALU or K-IALU register file register or JB/KB register for this purpose).
<i>Length</i>	Load this bit field (with the command and <i>Stall_Mode</i> bit fields) into the corresponding CACMDx register to execute the command. This bit field selects the length of the data (number of cache ways) minus one. For example, a value of 15 written to this field selects a length of 16. The <i>Length</i> is a 9-bit field using bits 23–15.
<i>Stall_Mode</i>	Load this bit (with the command and <i>Length</i> bit fields) into the corresponding CACMDx register to execute the command. This bit (14) selects the <i>Stall_Mode</i> as: <ul style="list-style-type: none">• <i>Stall_Mode</i> = 0; (stall enabled; omit CACMD_NOSTALL option) stall other writes to corresponding CACMDx and CCAIRx registers until command with stall is processed. If a stall occurs, the CASTAT_STL_ACTIVE bit (bit 19) is set in the corresponding CASTATx register.

Memory System Commands

- *Stall_Mode* = 1; (stall disabled; add CACMD_NOSTALL option) stall other writes to corresponding CACMDx and CCAIRx registers while the command is processed. If the other write is stalled for more than eight cycles, the processor aborts the other write, sets the CASTAT_COM_ABRTD bit (bit 17) of the corresponding CASTATx register, and issues a memory system exception (setting EXCAUSE bits to 0b1010). To identify completion of command execution, programs can poll the CASTAT_COM_ACTIVE bit (bit 16) of the corresponding CASTATx register for completion status of the memory system command.

`_LOCK`


The `CACHE_INIT_LOCK` command and `CACHE_INIT` commands are identical, except that the `_LOCK` version sets the lock bit on all accessed cache ways.



If the cache initialize process encounters previously locked ways in the cache, the data in the locked ways is retained and is not initialized by the cache initialize command. As long as sufficient unlocked ways are available, the cache initialization process is not hindered.

The cache initialize command operates differently than other memory system commands. This command executes in a sequence, and the execution is in the background to normal program execution. While the command executes, other accesses are allowed. The accesses of processor core and system (DMA or external master) are in higher priority than the command execution. Instruction fetch is in the same priority as the command execution, and the selection between the two sources (if this is a conflict) is round robin.

The cache initialize command also starts a prefetch sequence. If the command is followed by read accesses to the sequential addresses, these are prefetch buffer hits, and the prefetch sequence continues.

 The cache must be enabled before the cache initialize command can work.

Example

```

/* uses values from DEFTS201.H file */

J0 = (CACHE_INIT | (0x80 << CACMD_LEN_P) ) ;;
J1 = 0x00000000 ;;
CCAMDO = J1:0 ;;
/* In CCAIRO, sets the cache Start_Address to 0x0000 for memory
block 0. In CACMD0, sets the CACMD0 register to 0x40400000, exe-
cuting the cache initiate command (0x40000000) with a length of
0x80 (0x00400000) and stall mode (omitted) */

J2 = (CACHE_INIT_LOCK | (0x80 << CACMD_LEN_P) | CACMD_NOSTALL);;
J3 = 0x00000000 ;;
CCAMDO = J3:2 ;;
/* In CCAIRO, sets the cache Start_Address to 0x0000 for memory
block 0. In CACMD0, sets the CACMD0 register to 0x44404000, exe-
cuting the cache initiate and lock command (0x44000000) with a
length of 0x80 and no stall mode (0x00004000) */

```

Memory System Commands

Cache Copyback (to Memory)

Syntax

```
Js = Start_Index ;  
CCAIRx = Js ;  
  
Js = (CACMD_CB | (Length <<CACMD_LEN_P) { | CACMD_NOSTALL } ) ;  
CACMDx = Js ;  
  
/* where Js is any IALU data register (J30-0 or K30-0),  
Start_Index is a 7-bit index, using bits 16-10 (bits 9-0 are  
ignored), Length is a value ranging from 0 to 511, Stall_Mode is  
CACMD_NOSTALL (for no STALL) or omitted (for STALL), and x in  
CCAIRx or CACMDx is the block number: 0, 2, 4, 6, 8, 10, or B */  
  
/* If the CACMDx value and CCAIRx value are written into a double  
register (for example, CACMDx in J0 and CCAIRx in J1), a double  
register transfer loads both values at once. As in:  
Jm = CACMDx_value ; /* low register */  
Jn = CCAIRx_value ; /* high register */  
CACMDx = Jmd ;      /* long word transfer */
```


Function

This memory system command writes (copies back) dirty cache ways from a memory block's cache into the memory block. Only cache ways with their dirty bit set are actually written to memory. This command uses "holes" in the application for copying back data to the embedded DRAM. Once the data is driven to the copyback buffer, the way's dirty bit is cleared. Later access to the cache that causes replacement does not generate stalls that result from the copyback to the replaced entries. The parameters for this command include the *Start_Index* in cache, the *Length* of the data, and the *Stall_Mode* for the process.


<i>Start_Index</i>	Load this value into the corresponding CCAIRx register before loading the CACMDx register with the command. This value is a cache set number. The <i>Start_Index</i> is a 7-bit field using bits 16–10 (bits 31–17 and 9–0 are ignored, assumed zero; programs may use J-IALU or K-IALU register file register or JB/KB register for this purpose).
<i>Length</i>	Load this bit field (with the command and <i>Stall_Mode</i> bit fields) into the corresponding CACMDx register to execute the command. This bit field selects the length of the data (number of cache ways) minus one. For example, a value of 15 written to this field selects a length of 16. The <i>Length</i> is a 9-bit field using bits 23–15.
<i>Stall_Mode</i>	Load this bit (with the command and <i>Length</i> bit fields) into the corresponding CACMDx register to execute the command. This bit (14) selects the <i>Stall_Mode</i> as: <ul style="list-style-type: none">• <i>Stall_Mode</i> = 0; (stall enabled; omit CACMD_NOSTALL option) stall other writes to corresponding CACMDx and CCAIRx registers until command with stall is processed. If a stall occurs, the CASTAT_STL_ACTIVE bit (bit 19) is set in the corresponding CASTATx register.

Memory System Commands

- *Stall_Mode* = 1; (stall disabled; add CACMD_NOSTALL option) stall other writes to corresponding CACMDx and CCAIRx registers while the command is processed. If the other write is stalled for more than eight cycles, the processor aborts the other write, sets the CASTAT_COM_ABRTD bit (bit 17) of the corresponding CASTATx register, and issues a memory system exception (setting EXCAUSE bits to 0b1010). To identify completion of command execution, programs can poll the CASTAT_COM_ACTIVE bit (bit 16) of the corresponding CASTATx register for completion status of the memory system command.

 The cache copyback and cache invalidate commands use a starting index (in cache), not a starting address (in memory). Access memory for these commands is in order of sequential cache sets with four scans per set (one for each way).

The cache copyback command operates differently than other memory system commands. This command executes in a sequence, and the execution is in the background to normal program execution. While this command executes, other accesses are allowed. The accesses of processor core and system (DMA or external master) are in higher priority than the command execution. Instruction fetch is in the same priority as the command execution, and the selection between the two sources (if there is a conflict) is round robin. It is not recommended to access the index range that the cache copyback command is copying while the command is executing.

 The cache copyback command works whether or not the cache is enabled.

Example

```
/* uses CACMD_CB and CACMD_NOSTALL values from DEFTS201.H file */  
  
J0 = (CACMD_CB | (0x80 << CACMD_LEN_P) | CACMD_NOSTALL ) ;;  
J1 = 0x00000000 ;;  
CACMDB = J1:0 ;;  
  
/* In CCAIRx, sets the cache Start_Index to 0x00 for all memory  
blocks. In CACMDx, sets the CACMDx register to 0x10404000, exe-  
cuting the cache copyback command (0x10000000) with a length of  
0x80 and no stall mode (0x00004000) */
```

Memory System Commands

Cache Invalidate

Syntax

```
Js = Start_Index ;  
CCAIRx = Js ;
```

```
Js = (CACMD_INV | (Length <<CACHE_LEN_D) {| CACMD_NOSTALL}) ;  
CACMDx = Js ;
```

```
/* where Js is any IALU data register (J30-0 or K30-0),  
Start_Index is a 7-bit index, using bits 16-10 (bits 9-0 are  
ignored), Length is a value ranging from 0xFF8000 to 0x8000,  
Stall_Mode is CACMD_NOSTALL (for no STALL) or omitted (for  
STALL), and x in CCAIRx or CACMDx is the block number: 0, 2, 4, 6,  
8, 10, or B */
```

```
/* If the CACMDx value and CCAIRx value are written into a double  
register (for example, CACMDx in J0 and CCAIRx in J1), a double  
register transfer loads both values at once. As in:
```

```
Jm = CACMDx_value ; /* low register */  
Jn = CCAIRx_value ; /* high register */  
CACMDx = Jmd ; /* long word transfer */
```

Function

This memory system command goes through the selected range of cache ways clearing the LRU field, lock bit, dirty bit, and valid bits and updating the tag field of each way (way 0 tag = 0x1, way 1 tag = 0x2, way 2 tag

= 0x4, and way 3 tag = 0x8) to prevent multiple hits on write when the cache is enabled. The cache invalidate command needs to be used in the following situations.

- After power-up reset, before enabling the cache.
- When (old) cache data is not needed anymore and is not to be used again.
- To unlock cache ways—there is no other way to unlock ways in the cache.

The parameters for this command include the *Start_Index* in cache, the *Length* of the data, and the *Stall_Mode* for the process.

<i>Start_Index</i>	Load this value into the corresponding CCAIRx register before loading the CACMDx register with the command. This value is cache set number. The <i>Start_Index</i> is a 7-bit field using bits 16–10 (bits 9–0 are ignored, assumed zero; programs may use J-IALU or K-IALU register file register or JB/KB register for this purpose).
<i>Length</i>	Load this bit field (with the command and <i>Stall_Mode</i> bit fields) into the corresponding CACMDx register to execute the command. This bit field selects the length of the data (number of cache ways) minus one. For example, a value of 15 written to this field selects a length of 16. The <i>Length</i> is a 9-bit field using bits 23–15.

Memory System Commands

Stall_Mode

Load this bit (with the command and *Length* bit fields) into the corresponding *CACMDx* register to execute the command. This bit (14) selects the *Stall_Mode* as:

- *Stall_Mode* = 0; (stall enabled; omit *CACMD_NOSTALL* option) stall other writes to corresponding *CACMDx* and *CCAIRx* registers until command with stall is processed. If a stall occurs, the *CASTAT_STL_ACTIVE* bit (bit 19) is set in the corresponding *CASTATx* register.
- *Stall_Mode* = 1; (stall disabled; add *CACMD_NOSTALL* option) stall other writes to corresponding *CACMDx* and *CCAIRx* registers while the command is processed. If the other write is stalled for more than eight cycles, the processor aborts the other write, sets the *CASTAT_COM_ABRTD* bit (bit 17) of the corresponding *CASTATx* register, and issues a memory system exception (setting *EXCAUSE* bits to 0b1010). To identify completion of command execution, programs can poll the *CASTAT_COM_ACTIVE* bit (bit 16) of the corresponding *CASTATx* register for completion status of the memory system command.



The cache copyback and cache invalidate commands use a starting index (in cache), not a starting address (in memory). Access memory for these commands is in order of sequential cache sets with four scans per set (one for each way).

The cache invalidate command operates differently than other memory system commands. This command executes in a sequence, and the execution is in the background to normal program execution. While this

command executes, other accesses are allowed. The accesses of processor core and system (DMA or external master) are in higher priority than the command execution. Instruction fetch is in the same priority as the command execution, and the selection between the two sources (if conflict) is round robin. It is not recommended to access the index range that the cache invalidate command is invalidating while the command is executing.



The cache must be disabled before the cache invalidate command can work.

Example

```
/* uses CACMD_INV and CACMD_LEN_P values from DEFTS201.H file */  
  
J0 = (CACMD_INV | (0x80 << CACMD_LEN_P) ) ;;  
J1 = 0x00000000 ;;  
CACMDB = J1:0 ;;  
  
/* In CCAIRx, sets the cache Start_Index to 0x00 for all memory  
blocks. In CACMDX, sets the CACMDx register to 0x10404000, exe-  
cuting the cache invalidate command (0x14000000) with a length of  
0x80 and no stall mode (0x00004000) */
```

Memory System Commands

10 INSTRUCTION SET

This chapter describes the ADSP-TS201 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 10-2](#)
- [“CLU Instructions” on page 10-96](#)
- [“Multiplier Instructions” on page 10-107](#)
- [“Shifter Instructions” on page 10-170](#)
- [“IALU \(Integer\) Instructions” on page 10-199](#)
- [“IALU \(Load/Store/Transfer\) Instructions” on page 10-218](#)
- [“Sequencer Instructions” on page 10-232](#)
- [“Memory System Commands” on page 9-58](#)

For information on these architectural units of the TigerSHARC processor core, see [“ALU” on page 3-1](#), [“CLU” on page 4-1](#), [“Multiplier” on page 5-1](#), [“Shifter” on page 6-1](#), [“IALU” on page 7-1](#), and [“Program Sequencer” on page 8-1](#). For more information on registers (and register naming syntax) used in the instructions, see [“Compute Block Registers” on page 2-1](#). The instructions within each group are described in detail in this chapter. For a summary of all instruction syntax, see [“Quick Reference” on page A-1](#). 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 2-6](#). Briefly, these conventions are:

- { } – the curly braces enclose options; these braces are not part of the instruction syntax.
- | – the vertical bar separates choices; this bar is 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 1-23](#) and [“Instruction Parallelism Rules” on page 1-27](#).

Add/Subtract

Syntax

$$\{X|Y|XY\}\{S|B\}Rs = Rm +|- Rn \{(\{S|SU\}\{NF\})\} ;$$

$$\{X|Y|XY\}\{L|S|B\}Rsd = Rmd +|- Rnd \{(\{S|SU\}\{NF\})\} ;$$

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 “Register File Registers” on page 2-6.

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
(NF)	No flag (status) update

Examples

$YBR9 = R2 + R8 (S); ;$ see [Figure 10-1](#)

$YSR2 = R1 - R0 (SU); ;$ see [Figure 10-2](#)

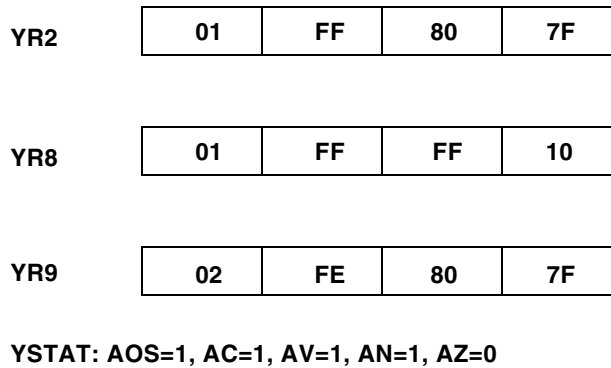


Figure 10-1. Quad Byte, Single Register, Signed Saturating Addition in Compute Block Y

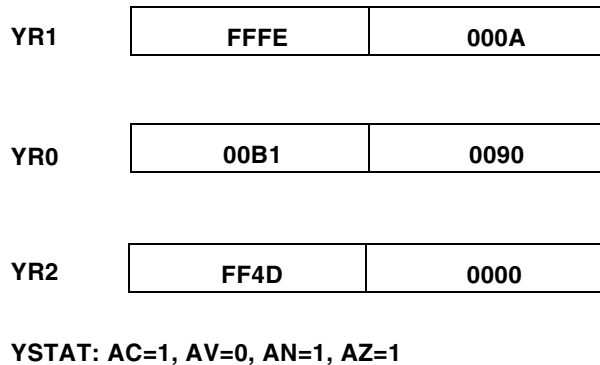


Figure 10-2. Dual Short, Single Register, Unsigned Saturating Subtraction in Compute Block Y

ALU Instructions

Add/Subtract With Carry/Borrow

Syntax

$$\{X|Y|XY\}Rs = Rm + CI \{-1\} ;$$
$$\{X|Y|XY\}LRsd = Rmd + CI \{-1\} ;$$
$$\{X|Y|XY\}Rs = Rm + Rn + CI \{(\{S|SU\}\{NF\})\} ;$$
$$\{X|Y|XY\}Rs = Rm - Rn + CI -1 \{(\{S|SU\}\{NF\})\} ;$$
$$\{X|Y|XY\}LRsd = Rmd + Rnd + CI \{(\{S|SU\}\{NF\})\} ;$$
$$\{X|Y|XY\}LRsd = Rmd - Rnd + CI -1 \{(\{S|SU\}\{NF\})\} ;$$

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
(NF)	No flag (status) update

Examples

```
xR2 = 0x00000001;;
xR0 = 0xFFFFFFFF;;
xR1 = R0 + R0;; /* generates carry */
xR3 = R2 + R2 + CI;;
```

The result in xR3 is 0x00000003, and XSTAT = 0.

ALU Instructions

$R9 = R3 - R10 + CI - 1;;$

If $R3 = 0x17$ and $R10 = 0x4$ and the carry from the previous ALU operation = 1

then $R9 = 0x13$ and the carry out is $AC = 1$

Add/Subtract With Divide by Two

Syntax

$$\{X|Y|XY\}\{S|B\}Rs = (Rm +|- Rn)/2 \{(\{T\}\{U\}\{NF\})\} ;$$

$$\{X|Y|XY\}\{L|S|B\}Rsd = (Rmd +|- Rnd)/2 \{(\{T\}\{U\}\{NF\})\} ;$$

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 can occur. Rounding is to nearest even (unbiased round) or by truncation. In case of negative result in unsigned, Rs is set to zero, and overflow bit is set.

The L, S, and B prefixes denote the operand type and d denotes operand size—see [“Register File Registers”](#) on page 2-6.

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

ALU Instructions

(TU) Unsigned truncate

(NF) No flag (status) update

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 $R9 = 0xC0000001$

Absolute Value/Absolute Value of Sum or Difference

Syntax

```

{X|Y|XY}{S|B}Rs = ABS Rm {(NF)} ;
{X|Y|XY}{L|S|B}Rsd = ABS Rmd {(NF)} ;
{X|Y|XY}{S|B}Rs = ABS (Rm + Rn) {(X){NF}} ;
{X|Y|XY}{L|S|B}Rsd = ABS (Rmd + Rnd) {(X){NF}} ;
{X|Y|XY}{S|B}Rs = ABS (Rm - Rn) {(X|U){NF}} ;
{X|Y|XY}{L|S|B}Rsd = ABS (Rmd - Rnd) {(X|U){NF}} ;

```

Function

These instructions add or subtract the signed operands in registers *Rm* and *Rn*, then place absolute value of the normalized result in register *Rs*. The *Rn* operand may be omitted, performing the absolute value operation on *Rm*.

The L, S, and B prefixes denote the operand type and d denotes operand size—see [“Register File Registers”](#) on page 2-6.

For add, option X provides an extended output range. The normal range without option X is from zero to the maximum positive number, or 0x0 to 0x7F...F. All outputs outside of this range are saturated to the maximum positive number. With option X, the output numbers are unsigned in the extended range 0x0 to 0xF...F.

For subtract, option X provides an extended output range. The normal range without option X is from zero to the maximum positive number, or 0x0 to 0x7F...F. Normal range can be either for signed (), or for unsigned inputs (U). All outputs outside of this range are saturated to the maximum positive number. With option X, the output numbers are unsigned in the extended range 0x0 to 0xF...F, and the inputs are assumed to be signed.

ALU Instructions

Status Flags

AZ	Set if all bits in result are zero
AN	Set to the most significant bit of the result prior to ABS
AV (AOS)	For ABS, set when Rm is the most negative number; otherwise AV cleared 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 $0x80...0$ For ABS (subtract), overflow, when option (X) is not used, if the result is more than the maximum signed; otherwise AV cleared
AC	Cleared

Options

()	Signed outputs in range $[0x0, 0x7F...F]$
(X)	Unsigned outputs in extended range $[0x0, 0xF...F]$
(U)	Unsigned inputs and outputs
(NF)	No flag (status) update

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 = 0x7FFFFFF8 *and* R3 = 0xFFFFFFFF0
then R5 = 0x7FFFFFFF
If option X is active, then R5 = 0x80000008

ALU Instructions

Two's Complement

Syntax

```
{X|Y|XY}{S|B}Rs = - Rm {(NF)} ;  
{X|Y|XY}{L|S|B}Rsd = - Rmd {(NF)} ;
```

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 [“Register File Registers” on page 2-6](#).

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	Cleared

Options

(NF) No flag (status) update

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
```

Maximum/Minimum

Syntax

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

$$\{X|Y|XY\}\{L|S|B\}Rsd = \text{MAX|MIN} (Rmd, Rnd) \{(\{U\}\{Z\}\{NF\})\} ;$$

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, word-by-word, or long-by-long—depending on the data size, where the maximum or minimum value of each comparison is passed to the corresponding byte/short/word/long 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 “[Register File Registers](#)” on page 2-6.

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

ALU Instructions

- (Z) Signed Zero option
- (UZ) Unsigned Zero option
- (NF) No flag (status) update

Examples

SR9:8= MAX (R3:2, R1:0);; *see Figure 10-3*

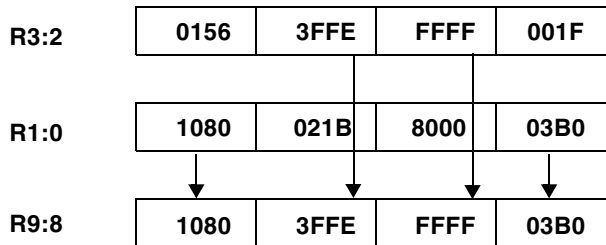


Figure 10-3. Example in Quad Short

SR9:8 = MAX (R3:2, R1:0) (Z);; *see Figure 10-4*

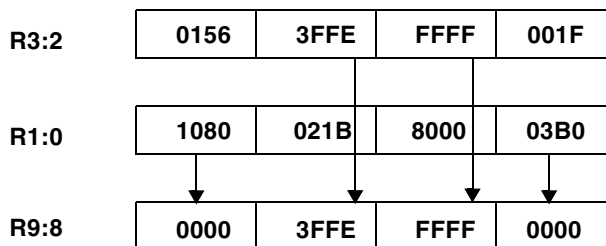


Figure 10-4. Example in Quad Short

SR9:8 = MIN (R3:2, R1:0) ;; *see Figure 10-5*

SR9:8 = MIN (R3:2, R1:0) (Z);; *see Figure 10-6*

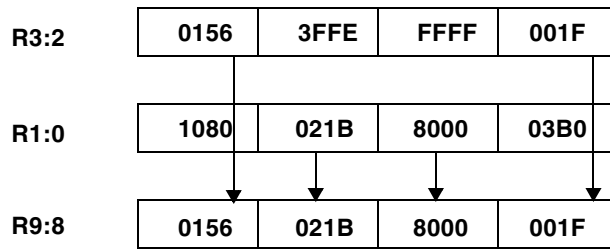


Figure 10-5. Example in Quad Short

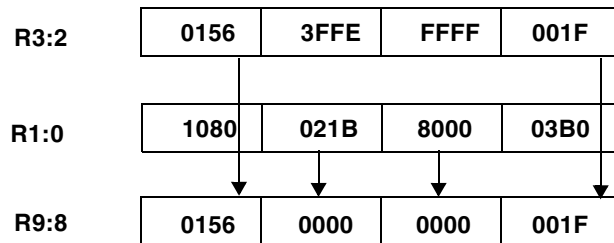


Figure 10-6. Example in Quad Short

ALU Instructions

Viterbi Maximum/Minimum

Syntax

```
{X|Y|XY}S|BRsd = VMAX|VMIN (Rmd, Rnd) {(NF)} ;
```

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)
```

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 “[Register File Registers](#)” on page 2-6. 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

Options

(NF)	No flag (status) update
------	-------------------------

Examples

```

Initial PR1:0 = 0xabc01281 987fed35
SR9:8= VMAX (R3:2, R1:0);; see Figure 10-7

```

```

Initial PR1:0 = 0xabc01281 987fed35
SR9:8= VMIN (R1:0, R3:2);; see Figure 10-8

```

ALU Instructions

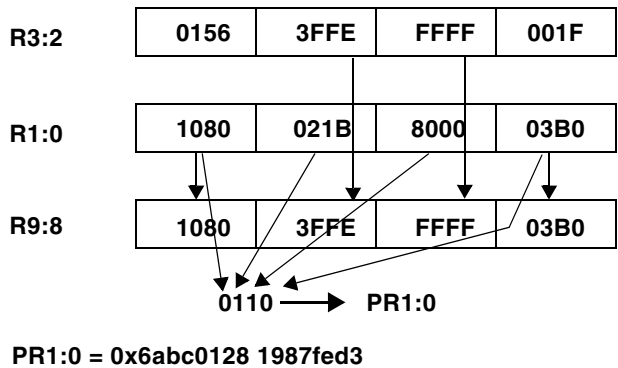


Figure 10-7. Example in Quad Short

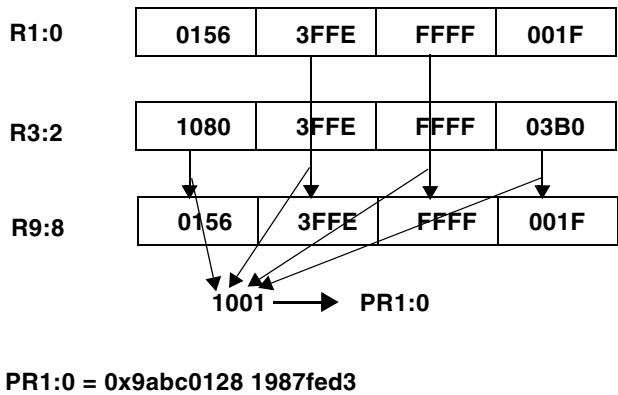


Figure 10-8. Example in Quad Short

Increment/Decrement

Syntax

```
{X|Y|XY}{S|B}Rs = INC|DEC Rm {{(S|SU){NF}}};
{X|Y|XY}{L|S|B}Rsd = INC|DEC Rmd {{(S|SU){NF}}};
```

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 [“Register File Registers” on page 2-6](#).

Signed saturation—option (S):

- If $(Rm+1) > \text{MAX_SN}$, then $Rs = \text{MAX_SN}$
- If $(Rm-1) < \text{MIN_SN}$, then $Rs = \text{MIN_SN}$

Unsigned saturation—option (SU):

- If $(Rm-1) < 0$, then $Rs = 0$
- If $(Rm+1) > \text{MAX_U}$, then $Rs = \text{MAX_U}$

MAX_SN and MIN_SN are the maximum and minimum signed numbers representable in the output format, and MAX_U is the maximum unsigned number. 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

ALU Instructions

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
(NF) No flag (status) update

Examples

```
R6 = INC R3 ;;  
If R3 = 0x00...01D  
then R6 = 0x00...01E
```

```
R6 = DEC R3 ;;  
If R3 = 0x00...01D  
then R6 = 0x00...01C
```

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, word-by-word, or long-by-long 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 [“Register File Registers” on page 2-6](#).

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

ALU Instructions

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
```

```
R3 = 0x00000000 ;;
```

```
R5 = 0x7F7F0000 ;;
```

```
XBCOMP(R3, R5) ;;
```

```
If XALT, JUMP my_label ;;
```

```
/* The jump is taken if any byte in R3 is less than the  
corresponding byte in R5. */
```

```
XBCOMP(R5, R3) ;;
```

```
IF NXALE JUMP my_label ;;
```

```
/* The jump is taken if all bytes in R3 are less than the  
corresponding byte in R5. */
```


Clip

Syntax

```
{X|Y|XY}{S|B}Rs = CLIP Rm BY Rn {(NF)} ;
{X|Y|XY}{L|S|B}Rsd = CLIP Rmd BY Rnd {(NF)} ;
```

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 “Register File Registers” on page 2-6.

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

(NF)	No flag (status) update
------	-------------------------

Example

```
XR3:2 = CLIP R9:8 BY R1:0;; see Figure 10-9
```

ALU Instructions

XR9:8	F0000000	0000077B
XR1:0	00FFABCD	FFFA0000
XR3:2	FF005433	0000077B

XSTAT: AC=0, AV=0, AN=1, AZ=0

Figure 10-9. Dual Normal Word In Compute Block X

Sideways Sum

Syntax

$$\{X|Y|XY\}Rs = \text{SUM } SRm|BRm \{(\{U\}\{NF\})\} ;$$

$$\{X|Y|XY\}Rs = \text{SUM } SRmd|BRmd \{(\{U\}\{NF\})\} ;$$

Function

This instruction adds the bytes/shorts in register *Rm* and stores the result in 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 [“Register File Registers” on page 2-6](#).

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 integer
(U)	Unsigned integer
(NF)	No flag (status) update

ALU Instructions

Examples

XR4 = SUM SR3:2;; see *Figure 10-10*

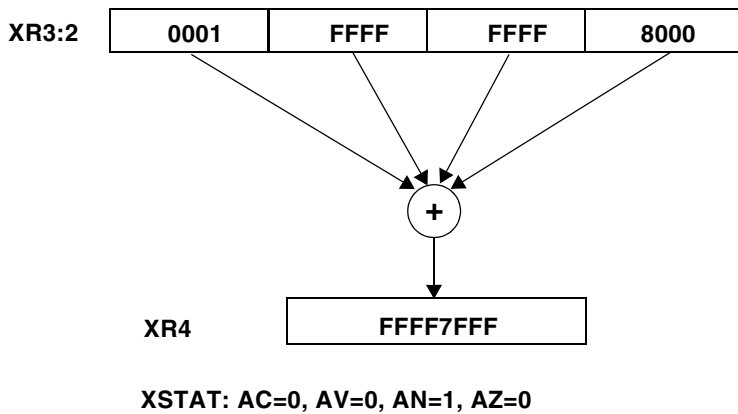


Figure 10-10. Signed SUM

XR4 = SUM SR3:2 (U);; see *Figure 10-11*

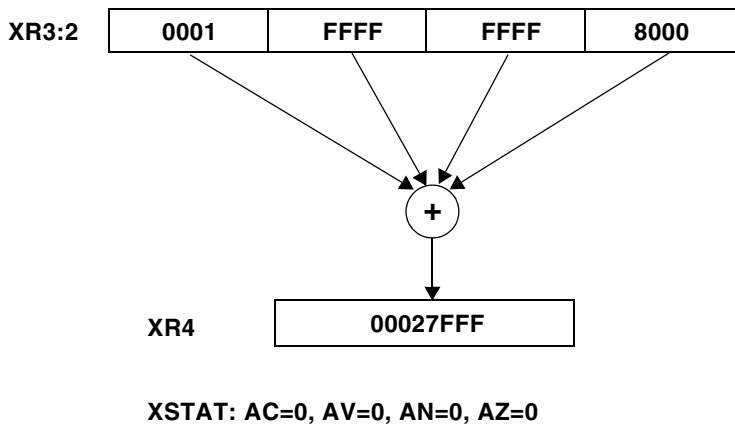


Figure 10-11. Unsigned SUM

Ones Counting

Syntax

$$\{X|Y|XY\}Rs = \text{ONES } Rm|Rmd \{(NF)\} ;$$

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 [“Register File Registers” on page 2-6](#).

Status Flags

AZ	Set if all bits in result are zero
AN	Cleared
AV (AOS)	Cleared (not changed)
AC	Cleared

Options

(NF)	No flag (status) update
------	-------------------------

Examples

```
R6 = ONES R3;;
If R3 = b#00...011101
then R6 = 0x00...04
```

ALU Instructions

Load/Transfer PR (Parallel Result) Register

Syntax

```
{X|Y|XY}PR1:0 = Rmd {(NF)} ;  
{X|Y|XY}Rsd = PR1:0 {(NF)} ;
```

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 [“Register File Registers” on page 2-6](#).

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

(NF)	No flag (status) update
------	-------------------------

Bit FIFO Increment

Syntax

$\{X|Y|XY\}Rs = \text{BFOINC } Rmd \{(NF)\} ;$

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 10-12). For more information, see “Bit Stream Manipulation Operations” on page 6-12.

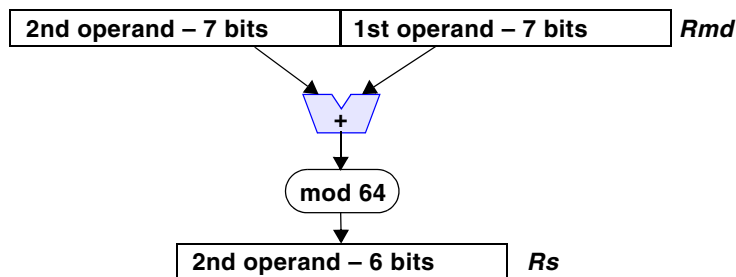


Figure 10-12. BFOINC Remainder

Status Flags

AZ	Set if all bits in result are zero
AN	Set when addition passes 63
AV (AOS)	Cleared (not changed)
AC	Cleared

Options

(NF)	No flag (status) update
------	-------------------------

ALU Instructions

Examples

$\{X|Y|XY\}Rs = \text{BFOINC } Rmd;$
If $Rmd = 0x0\dots030 \ 0\dots020$
then $Rs = 0x0\dots010$ *and* AN *is set*

Absolute Value With Parallel Accumulate

Syntax

```
{X|Y|XY}PR0|PR1 += ABS (SRmd - SRnd){({U}{NF})} ;
{X|Y|XY}PR0|PR1 += ABS (BRmd - BRnd){({U}{NF})} ;
```

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 signed, unless the unsigned (U) option is used. Saturation is always active.

The S and B prefixes denote the operand type and the d suffix denotes operand size—see [“Register File Registers” on page 2-6](#).

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

ALU Instructions

Options

- () Signed inputs
- (U) Unsigned inputs
- (NF) No flag (status) update

Examples

`XPR0 += ABS (SR3:2 - SR1:0);;` see [Figure 10-13](#)

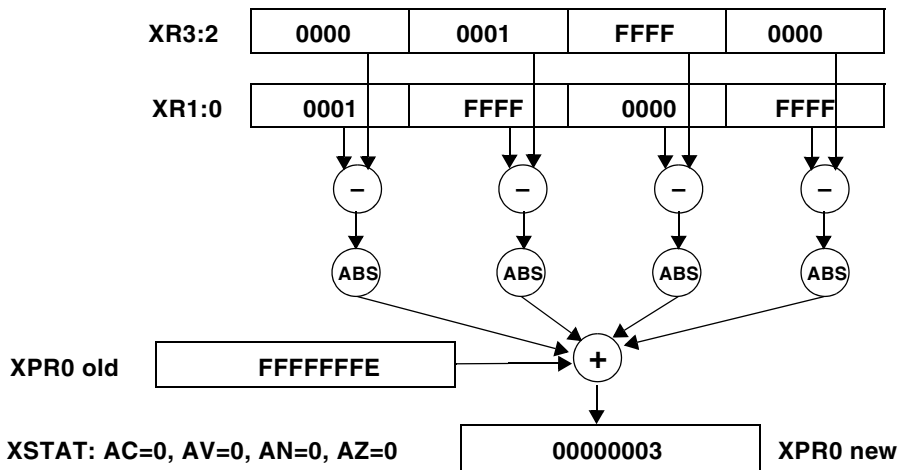


Figure 10-13. Parallel ABS

Sideways Sum With Parallel Accumulate

Syntax

```
{X|Y|XY}PRO|PR1 += SUM SRm {{(U){NF}}} ;
{X|Y|XY}PRO|PR1 += SUM SRmd {{(U){NF}}} ;
{X|Y|XY}PRO|PR1 += SUM BRm {{(U){NF}}} ;
{X|Y|XY}PRO|PR1 += SUM BRmd {{(U){NF}}} ;
```

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 “Register File Registers” on page 2-6.

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

ALU Instructions

Options

- () Signed inputs and result
- (U) Unsigned inputs and result
- (NF) No flag (status) update

Examples

PRO += SUM SR3:2;; see *Figure 10-14*

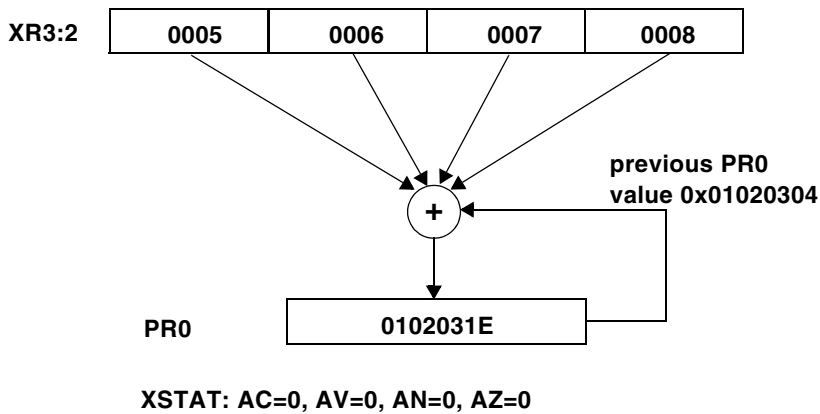


Figure 10-14. SUM With Parallel Accumulate

Add/Subtract (Dual Operation)

Syntax

$$\{X|Y|XY\}\{S|B\}Rs = Rm + Rn, Ra = Rm - Rn \{(NF)\} ; \textit{(dual op.)}$$

$$\{X|Y|XY\}\{L|S|B\}Rsd = Rmd + Rnd, Rad = Rmd - Rnd \{(NF)\} ; \textit{(dual op.)}$$

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 [“Register File Registers” on page 2-6](#).

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	Cleared

Options

(NF)	No flag (status) update
------	-------------------------

Examples

```
R9 = R4 + R8, R2 = R4 - R8;;
If R4 = 8 and R8 = 2
then R9 = 10 and R2 = 6
```

ALU Instructions

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 [“Register File Registers” on page 2-6](#).

Status Flags

AZ	Set if all bits in result are zero
AN	Set to the most significant bit Rm
AV (AOS)	Cleared (not changed)
AC	Cleared

Examples

```
XR6 = PASS R3;;  
If R3 = 0x80000000  
then R6 = 0x80000000 and XSTAT: AC=0, AV=0, AN=1, AZ=0
```

Logical AND/AND NOT/OR/XOR/NOT

Syntax

$$\{X|Y|XY\}Rs = Rm \text{ AND|AND NOT|OR|XOR } Rn \{(NF)\} ;$$

$$\{X|Y|XY\}LRsd = Rmd \text{ AND|AND NOT|OR|XOR } Rnd \{(NF)\} ;$$

$$\{X|Y|XY\}Rs = \text{NOT } Rm \{(NF)\} ;$$

$$\{X|Y|XY\}LRsd = \text{NOT } Rmd \{(NF)\} ;$$

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 [“Register File Registers” on page 2-6](#).

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

(NF)	No flag (status) update
------	-------------------------

ALU Instructions

Examples

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.

Expand

Syntax

```

{X|Y|XY}Rsd = EXPAND SRm {+|- SRn} {{{I|IU}{NF}}} ;
{X|Y|XY}Rsq = EXPAND SRmd {+|- SRnd} {{{I|IU}{NF}}} ;
{X|Y|XY}SRsd = EXPAND BRm {+|- BRn} {{{I|IU}{NF}}} ;
{X|Y|XY}SRsq = EXPAND BRmd {+|- BRnd} {{{I|IU}{NF}}} ;

```

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 negative 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
- *SRsd*=EXPAND *BRm* expands four bytes to four shorts
- *SRsq*=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* prefix denotes byte-word operands. The suffices *d* and *q* denote operand size—see “[Register File Registers](#)” on page 2-6.

Status Flags

AZ	Set if all bits in result are zero
AN	Set to the most significant bit of result

ALU Instructions

AV (AOS)	Set if overflow occurs in fractional operation or if the result is negative in the IU option
AC	Cleared

Options

()	Fractional
(I)	Signed integer
(IU)	Unsigned integer
(NF)	No flag (status) update

Examples

$\{X|Y|XY\}Rsd = \text{EXPAND } SRm$; see *Figure 10-15*

$\{X|Y|XY\}Rsq = \text{EXPAND } SRmd$;

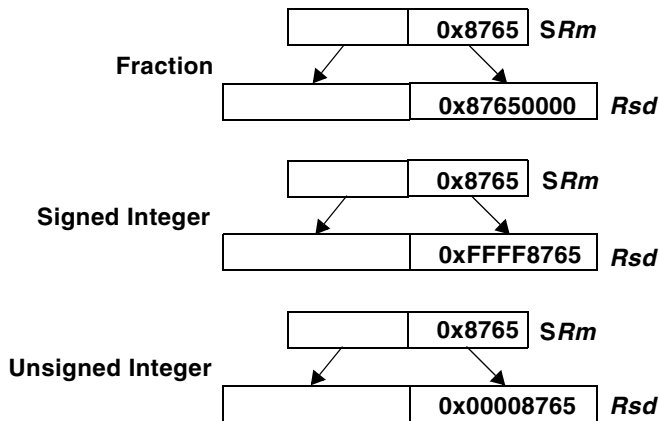


Figure 10-15. Fractional, Signed Integer, and Unsigned Integer EXPAND

$\{X|Y|XY\}SRsd = \text{EXPAND } BRm; \text{ see Figure 10-16}$
 $\{X|Y|XY\}SRsq = \text{EXPAND } BRmd;$

**Byte to short expand
in fraction, signed int,
and unsigned int op**

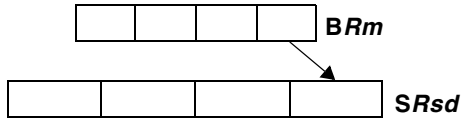


Figure 10-16. Expand Byte Words to Short Words

$\{X|Y|XY\}Rsd = \text{EXPAND } SRm + SRn; \text{ see Figure 10-17}$
 $\{X|Y|XY\}Rsq = \text{EXPAND } SRmd + SRnd;$

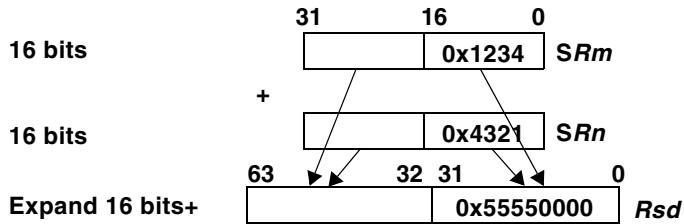


Figure 10-17. Expand Short Words to Normal Words

ALU Instructions

$\{X|Y|XY\}SRsd = \text{EXPAND } BRm + BRm;$

$\{X|Y|XY\}SRsq = \text{EXPAND } BRmd + BRmd;$ see [Figure 10-18](#)

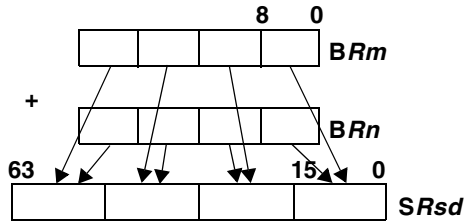


Figure 10-18. Add and Expand Byte Words to Short Words

$\{X|Y|XY\}Rsd = \text{EXPAND } SRm - SRn;$ see [Figure 10-19](#)

$\{X|Y|XY\}Rsq = \text{EXPAND } SRmd - SRnd;$

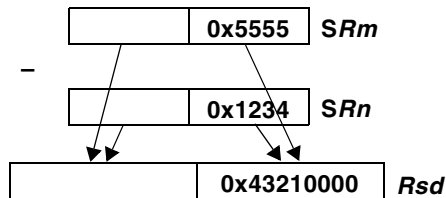


Figure 10-19. Subtract and Expand Short Words to Normal Words

$\{X|Y|XY\}SRsd = \text{EXPAND } BRm - BRm; \text{ see Figure 10-20}$
 $\{X|Y|XY\}SRsq = \text{EXPAND } BRmd - BRmd;$

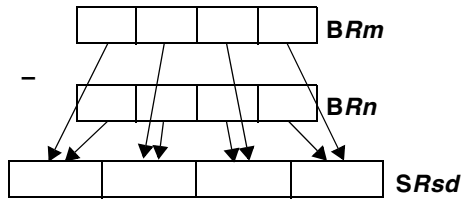


Figure 10-20. Subtract and Expand Byte Words to Short Words

ALU Instructions

Compact

Syntax

```
{X|Y|XY}SRs = COMPACT Rmd {+|- Rnd} {(T|I|IS|ISU}{NF}} ;  
{X|Y|XY}BRs = COMPACT SRmd {+|- SRnd} {(T|I|IS|ISU}{NF}} ;  
{X|Y|XY}Rs = COMPACT LRmd {(U|IS|ISU}{NF}} ;  
{X|Y|XY}LRsd = COMPACT QRmq {(U|IS|ISU}{NF}} ;
```

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 as follows:

- Two normal words into two short words ($SRs = COMPACT Rmd$)
- Four short words into four byte words ($BRs = COMPACT SRmd$)
- One long word into one normal word ($Rs = COMPACT LRmd$)
- One quad word into one long word ($LRsd = COMPACT QRmq$)

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.

The prefixes denote operand types, including short-word (S), byte-word (B), long-word (L), quad-word (Q). The d and q suffixes denote operand size—see [“Register File Registers” on page 2-6](#).

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
(U)	Unsigned
(I)	Integer signed, no saturation
(IS)	Signed integer, saturation
(ISU)	Unsigned integer, saturation
(NF)	No flag (status) update

ALU Instructions

Examples

$\{X|Y|XY\}SRs = \text{COMPACT } Rmd$; see *Figure 10-21*

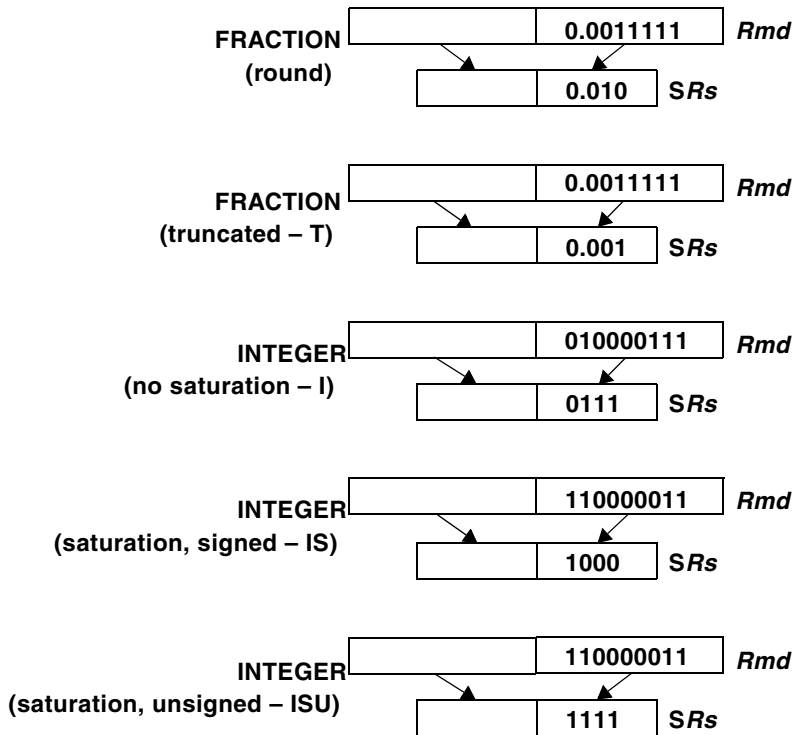


Figure 10-21. COMPACT Instruction Options

$\{X|Y|XY\}BRs = \text{COMPACT } SRmd$; see *Figure 10-22*

**Short to byte compact
in fraction round,
fraction trunc, int no sat,
signed int and unsigned
int operation**

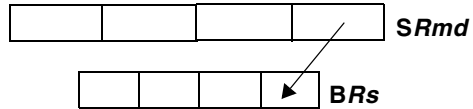


Figure 10-22. Compact Short Words to Byte Words

$\{X|Y|XY\}SRs = \text{COMPACT } Rmd + Rnd$; see *Figure 10-23*

$\{X|Y|XY\}BRsq = \text{COMPACT } SRmd + SRnd$

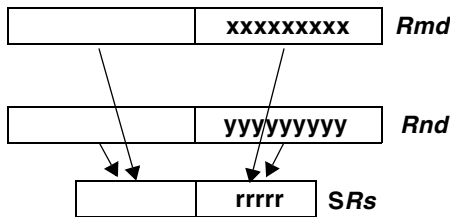


Figure 10-23. Add and Compact Normal Words to Short Words

$\{X|Y|XY\}BRs = \text{COMPACT } SRmd + SRnd$; see *Figure 10-24*

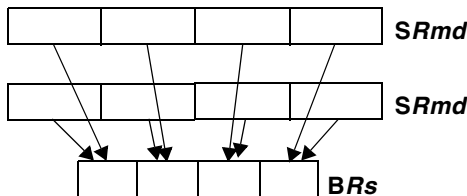


Figure 10-24. Add and Compact Short Words to Byte Words

ALU Instructions

Merge

Syntax

```
{X|Y|XY}BRsd = MERGE Rm, Rn {(NF)} ;  
{X|Y|XY}BRsq = MERGE Rmd, Rnd {(NF)} ;  
{X|Y|XY}SRsd = MERGE Rm, Rn {(NF)} ;  
{X|Y|XY}SRsq = MERGE Rmd, Rnd {(NF)} ;
```

Function

This instruction merges (transposes) the operands in registers *Rm* and *Rn*. The result is placed in register *Rs* ([Figure 10-25](#)).

The B and S prefixes denote byte- and short-word operand types respectively, and the suffices d and q denote operand size—see “[Register File Registers](#)” on page 2-6.

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

(NF) No flag (status) update

Example

XBR3:2 = MERGE R0, R1;;

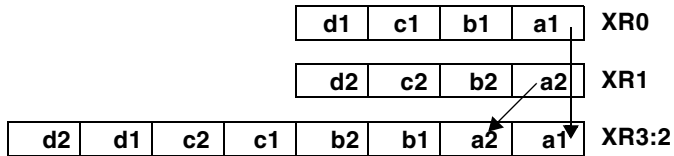


Figure 10-25. Merge Function Data Flow

Instruction `MERGE` may be used to transpose shorts and bytes by repeatedly merging the results.

ALU Instructions

Permute (Byte Word)

Syntax

{X|Y|XY}Rsd = PERMUTE (Rmd, Rn) {(NF)} ;

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

Options

(NF) No flag (status) update

Example

R1:0 = 0x01234567_89abcdef

R4 = 0x03172454

R7:6 = permute (R1:0, R4)

Result R7:6 is 0xef89cd01_ab674567

ALU Instructions

Permute (Short Word)

Syntax

$\{X|Y|XY\}Rsq = \text{PERMUTE} (Rmd, -Rmd, Rn) \{(NF)\} ;$

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 cannot be executed in parallel to some multiplier instructions, because ALU instructions with quad results use a multiplier bus resource.

Status Flags

AV	Overflow (AV) is calculated every cycle, set if one of the shorts in <i>Rmd</i> is maximum negative (0x8000) and it is selected by at least one of the nibbles of <i>Rn</i> to the result.
AOS	Set whenever an overflow occurs and cleared only by X/YSTAT load.

Options

(NF)	No flag (status) update
------	-------------------------

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

```

ALU Instructions

Add/Subtract (Floating-Point)

Syntax

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

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 [“Register File Registers” on page 2-6](#).

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](#).

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
(NF)	No flag (status) update

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.

ALU Instructions

Add/Subtract With Divide by Two (Floating-Point)

Syntax

$$\{X|Y|XY\}FRs = (Rm +|- Rn)/2 \{(\{T\}\{NF\})\} ;$$
$$\{X|Y|XY\}FRsd = (Rmd +|- Rnd)/2 \{(\{T\}\{NF\})\} ;$$

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 “[Register File Registers](#)” on [page 2-6](#).

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 - 0xFE

² Maximum normal value - mantissa - all 1 (0x7F...0), exponent - 0xFE

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
(NF)	No flag (status) update

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

ALU Instructions

Maximum/Minimum (Floating-Point)

Syntax

$$\{X|Y|XY\}FRs = \text{MAX|MIN} (Rm, Rn) \{(NF)\} ;$$
$$\{X|Y|XY\}FRsd = \text{MAX|MIN} (Rmd, Rnd) \{(NF)\} ;$$

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 “[Register File Registers](#)” on page 2-6.

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 ±zero.

Status Flags

AZ	Set if result is ±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.

Options

(NF) No flag (status) update

Examples

```
XFR0 = MAX (R1, R2) ;; /* x compute */  
YFR1:0 = MAX (R3:2, R5:4) ;; /* y compute block with extended  
40-bit precision with rounding (by default) */
```

```
XFR0 = MIN (R1, R2) ;; /* x compute */  
YFR1:0 = MIN (R3:2, R5:4) ;; /* y compute block with extended  
40-bit precision with rounding (by default) */
```

³ See “[ALU Execution Status](#)” on page 3-11.

ALU Instructions

Absolute Value/ Absolute Value of Sum or Difference (Floating-Point)

Syntax

```
{X|Y|XY}FRs = ABS Rm {(NF)} ;  
{X|Y|XY}FRsd = ABS Rmd {(NF)} ;  
{X|Y|XY}FRs = ABS (Rm +|- Rn) {(T){NF}} ;  
{X|Y|XY}FRsd = ABS (Rmd +|- Rnd) {(T){NF}} ;
```

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 “[Register File Registers](#)” on page 2-6.

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 \pm zero
AUS	Set if operand (post-rounded result for add or subtract) is denormal; otherwise not cleared ²

¹ Maximum normal value - mantissa - all 1 (0x7F...0), exponent - 0xFE

² Maximum normal value - mantissa - all 1 (0x7F...0), exponent - 0xFE

AN	Set if result of addition operation prior to ABS is negative
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
(NF)	No flag (status) update

¹ See “ALU Execution Status” on page 3-11.

² See “ALU Execution Status” on page 3-11.

ALU Instructions

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 */
```


Complement Sign (Floating-Point)

Syntax

$$\{X|Y|XY\}FRs = - Rm \{(NF)\} ;$$

$$\{X|Y|XY\}FRsd = - Rmd \{(NF)\} ;$$

Function

This instruction complements the sign bit of the floating-point operand in register. The complemented result is placed in register FRs . The d suffix denotes extended operand size—see [“Register File Registers” on page 2-6](#).

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](#).

ALU Instructions

Options

(NF)

No flag (status) update

Examples

```
XFR0 = -R1 ;; /* x compute block complement */  
YFR1:0 = -R3:2;; /* extended 40-bit precision */
```

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 “[Register File Registers](#)” on page 2-6.

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 $Rm = Rn$ and neither <i>Rm</i> or <i>Rn</i> are NaNs
<i>AUS</i>	Not cleared ¹
<i>AN</i>	Set if $Rm < Rn$ and neither <i>Rm</i> or <i>Rn</i> 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.

ALU Instructions

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 3-15. 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`.

Examples

```
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 ;;
```

Floating- to Fixed-Point Conversion

Syntax

$$\{X|Y|XY\}Rs = \text{FIX } FRm|FRmd \{BY Rn\} \{(T)\{NF\}\} ;$$

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 or truncated. The result is placed in register *Rs*. The *d* suffix denotes extended operand size—see “[Register File Registers](#)” on page 2-6.

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 158; unless the operand is negative, the sum is equal to 158, and the mantissa is all zero

¹ See “[ALU Execution Status](#)” on page 3-11.

ALU Instructions

AVS	Set if the exponent of the floating-point result is larger or equal to 158; unless the operand is negative, the sum is equal to 158, 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
(NF)	No flag (status) update

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](#).

Fixed- to Floating-Point Conversion

Syntax

$$\{X|Y|XY\}FRs|FRsd = \text{FLOAT } Rm \{BY Rn\} \{(\{T\}\{NF\})\} ;$$

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 1-23.

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

¹ Maximum normal value - mantissa - all 1 (0x7F...0), exponent - 0xFE

² Maximum normal value - mantissa - all 1 (0x7F...0), exponent - 0xFE

ALU Instructions

AV	For not scaled, cleared; for scaled, set if post-rounded result overflows
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
(NF)	No flag (status) update

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.

Floating-Point Normal to Extended Word Conversion

Syntax

$$\{X|Y|XY\}FRsd = \text{EXTD } Rm \{(NF)\} ;$$

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 “[Register File Registers](#)” on page 2-6.

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.

ALU Instructions

Options

(NF) No flag (status) update

Example

```
XFR1:0 = EXT D R2 ;; /* x compute block extend to 40 bit */
```

Floating-Point Extended to Normal Word Conversion

Syntax

$$\{X|Y|XY\}FRs = \text{SNGL } Rmd \{(\{T\}\{NF\})\} ;$$

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 “[Register File Registers](#)” on page 2-6.

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 $\pm\text{NORM.MAX}^1$ if (T) option is set, and $\pm\text{infinity}$ otherwise.

Status Flags

AZ	Set if result is $\pm\text{zero}$
AUS	Not cleared ²
AN	Set if result is negative
AV	Set if the exponent is $0x\text{FE}$ and the mantissa is more than $0x7\text{FFFFFF}80$ and option (T) is not used
AVS	Set if the exponent is $0x\text{FE}$ and the mantissa is more than $0x7\text{FFFFFF}80$, otherwise not cleared ³
AC	Cleared
AI	Set if input is a NAN

¹ Maximum normal value - mantissa - all 1 ($0x7\text{F}...0$), exponent - $0x\text{FE}$

² See “[ALU Execution Status](#)” on page 3-11.

³ See “[ALU Execution Status](#)” on page 3-11.

ALU Instructions

AIS Set if input is a NAN; otherwise not cleared¹

Options

() Round
(T) Truncate
(NF) No flag (status) update

Example

```
XFR0 = SNGL R3:2 (T) ;; /* x compute block with truncate option
*/
```

¹ See [“ALU Execution Status”](#) on page 3-11.

Clip (Floating-Point)

Syntax

```
{X|Y|XY}FRs = CLIP Rm BY Rn {(NF)} ;
{X|Y|XY}FRsd = CLIP Rmd BY Rnd {(NF)} ;
```

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 (Figure 10-26). The d suffix denotes extended operand size—see “Register File Registers” on page 2-6.

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.

ALU Instructions

AIS Set if either input operand is a NAN; otherwise not cleared¹

Options

(NF) No flag (status) update

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 10-26. 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.

Copysign (Floating-Point)

Syntax

$$\{X|Y|XY\}FRs = Rm \text{ COPYSIGN } Rn \{(NF)\} ;$$

$$\{X|Y|XY\}FRsd = Rmd \text{ COPYSIGN } Rnd \{(NF)\} ;$$

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 “[Register File Registers](#)” on page 2-6.

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 ¹
AC	Cleared
AI	Set if either input operand is a NAN
AIS	Set if either input operand is a NAN; otherwise not cleared ¹

¹ See “[ALU Execution Status](#)” on page 3-11.

ALU Instructions

Options

(NF)

No flag (status) update

Example

```
YFR0 = R1 COPYSIGN R2 ;; /* Y compute block */  
XFR1:0 = R3:2 COPYSIGN R5:4 ;; /* X compute block, extended  
40-bit precision */
```


Scale (Floating-Point)

Syntax

$$\{X|Y|XY\}FRs = \text{SCALB } FRm \text{ BY } Rn \{(NF)\} ;$$

$$\{X|Y|XY\}FRsd = \text{SCALB } FRmd \text{ BY } Rn \{(NF)\} ;$$

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 “[Register File Registers](#)” on page 2-6. If the sum of FRm exponent and Rn is zero, the result is rounded 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 ¹
AC	Cleared
AI	Set if either input operand is a NAN

¹ See “[ALU Execution Status](#)” on page 3-11.

ALU Instructions

AIS Set if either input operand is a NAN; otherwise not cleared¹

Options

(NF) No flag (status) update

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.

Pass (Floating-Point)

Syntax

$$\{X|Y|XY\}FRs = \text{PASS } Rm ;$$

$$\{X|Y|XY\}FRsd = \text{PASS } Rmd ;$$

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 [“Register File Registers” on page 2-6](#).

Denormal inputs are flushed to \pm zero. A NAN input returns an all ones result.

Status Flags

AZ	Set if result is \pm zero
AUS	Not cleared ¹
AN	Set if result is negative
AV	Cleared;
AVS	Not cleared; retains value from previous event ¹
AC	Cleared
AI	Set if input is NAN
AIS	Set if input is NAN; otherwise not cleared ¹

¹ See [“ALU Execution Status” on page 3-11](#).

ALU Instructions

Examples

```
FRO = PASS R1 ;; /* SIMD pass */  
XFR1:0 = PASS R3:2 ;; /* 40-bit extended-precision */
```

Reciprocal (Floating-Point)

Syntax

```
{X|Y|XY}FRs = RECIPS Rm {(NF)} ;
{X|Y|XY}FRsd = RECIPS Rmd {(NF)} ;
```

Function

This instruction creates an 8-bit accurate approximation for $1/Rm$, the reciprocal of Rm . The *d* suffix denotes the operand size—see “[Register File Registers](#)” on page 2-6.

The mantissa of the approximation is determined from a table using the seven MSBs (excluding the hidden bit) of the Rm mantissa as an index. The unbiased exponent of the seed is calculated as the two’s complement of the unbiased Rm exponent, decremented by one. If e is the unbiased exponent of Rm , 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 Rm is greater than +125, the result is $\pm zero$. A NAN input returns an all ones result.

Status Flags

AZ	Set if result is $\pm zero$
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 ¹
AC	Cleared
AI	Set if input is NAN

¹ See “[ALU Execution Status](#)” on page 3-11.

ALU Instructions

AIS Set if input is NAN; otherwise not cleared¹

Options

(NF) No flag (status) update

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.

Reciprocal Square Root (Floating-Point)

Syntax

```
{X|Y|XY}FRs = RSQRTS Rm {(NF)} ;
{X|Y|XY}FRsd = RSQRTS Rmd {(NF)} ;
```

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 “Register File Registers” on page 2-6.

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 $FRs = -[e/2] - 1$. 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 ¹

¹ See “ALU Execution Status” on page 3-11.

ALU Instructions

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

Options

(NF)	No flag (status) update
------	-------------------------

Examples

```
/* This code calculates a floating-point reciprocal square root
using a Newton Raphson iteration algorithm. The result is accu-
rate 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 ;;
```

¹ See “[ALU Execution Status](#)” on page 3-11.


```
XFR4 = R4 * R12 ;; /* reciprocal square root of xr0 is output is  
in r4 */  
endhere:  
    NOP ;;  
    IDLE ;;  
    JUMP endhere ;;
```

ALU Instructions

Mantissa (Floating-Point)

Syntax

$\{X|Y|XY\}Rs = \text{MANT } FRm|FRmd \{(NF)\} ;$

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 “[Register File Registers](#)” on page 2-6.

Note that the AN flag is set to the sign bit of the input operand.

Rounding options 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 (FRm) is negative
AV	Cleared
AVS	Not cleared ¹
AC	Cleared
AI	Set if input is NAN or infinity

¹ See “[ALU Execution Status](#)” on page 3-11.

AIS Set if input is NAN or infinity; otherwise not cleared¹

Options

(NF) No flag (status) update

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.

ALU Instructions

Logarithm (Floating-Point)

Syntax

$\{X|Y|XY\}Rs = \text{LOGB } FRm|FRmd \{(\{S\}\{NF\})\} ;$

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 [“Register File Registers” on page 2-6](#).

Unbiasing is performed by subtracting 127 from the floating-point exponent in *Rm*. If saturation option is not set, a \pm infinity input returns a floating-point +infinity; otherwise it returns the maximum positive value (0x7FFF FFFF). A \pm zero input returns a floating-point -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 ¹
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 ¹
AC	Cleared

¹ See [“ALU Execution Status” on page 3-11](#).

AI	Set if input is a NAN
AIS	Set if input is a NAN; otherwise not cleared ¹

Options

()	Saturation inactive
(S)	Saturation active
(NF)	No flag (status) update

Examples

```
XR0 = LOGB XFR1 (S) ;; /* x compute block with saturate active
option */
YR0 = LOGB YFR3:2 ;; /* y compute blocks 40-bit extended-precision
*/
```

¹ See [“ALU Execution Status”](#) on page 3-11.

ALU Instructions

Add/Subtract (Dual Operation, Floating-Point)

Syntax

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

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 [“Register File Registers” on page 2-6](#).

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 ¹
AC	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](#).

AIS Set if either input operand is a NAN, or if both operands are Infinities; otherwise not cleared¹

Options

(NF) No flag (status) update

Examples

```
XFR0 = R1 + R2 , FR3 = R1 - R2 ;; /* x compute block add and subtract */
```

```
XFR1:0 = R3:2 + R5:4 , FR7:6 = 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 Operations” on page 4-4](#).

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 2-6](#). 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 1-23](#) and [“Instruction Parallelism Rules” on page 1-27](#).

Trellis Maximum (CLU)

Syntax

```
{X|Y|XY}{S}TRsd = TMAX(TRmd + Rmq_h, TRnd + Rmq_l) {(NF)} ;
{X|Y|XY}{S}TRsd = TMAX(TRmd - Rmq_h, TRnd - Rmq_l) {(NF)} ;
{X|Y|XY}{S}Rs = TMAX(TRm, TRn) {(NF)} ;
/* where Rmq_h must be the upper half of a quad register and
Rmq_l must be the lower half of the SAME quad register */
```

Function:

The **TMAX** function is executed between TRm and TRn or is executed between either add or subtract results. For more detail on **TMAX** function execution, see [“TMAX Function” on page 4-6](#) and [“Trellis Function” on page 4-8](#).



Saturation is active in this instruction. For more details, see [“Saturation Option” on page 3-8](#).

Status Flags

TROV	Overflow (TROV) is calculated every cycle
TRSOV	Set whenever an overflow occurs and cleared only by X/YSTAT load

Options

(NF)	No flag (status) update
------	-------------------------

Examples

```
XYTR1:0 = TMAX(TR3:2 + R3:2, TR5:4 + R1:0) ;;
XYTR1:0 = TMAX(TR3:2 - R7:6, TR5:4 - R5:4) ;;
XYR8 = TMAX(TR4, TR0) ;;
```

CLU Instructions


Maximum (CLU)

Syntax

$$\{X|Y|XY\}\{S\}TRsd = \text{MAX}(TRmd + Rmq_h, TRnd + Rmq_l) \{(NF)\} ;$$
$$\{X|Y|XY\}\{S\}TRsd = \text{MAX}(TRmd - Rmq_h, TRnd - Rmq_l) \{(NF)\} ;$$

Function:

The MAX function is executed between either add or subtract results. For more detail on MAX function execution, see [“Trellis Function” on page 4-8](#).

 Saturation is supported in this instruction. For more details, see [“Saturation Option” on page 3-8](#).

Status Flags

TROV	Overflow (TROV) is calculated every cycle
TRSOV	Set whenever an overflow occurs and cleared only by X/YSTAT load

Options

(NF)	No flag (status) update
------	-------------------------

Examples

$$XYTR1:0 = \text{MAX}(TR3:2 + R3:2, TR5:4 + R1:0) ; ;$$
$$XYTR1:0 = \text{MAX}(TR3:2 - R7:6, TR5:4 - R5:4) ; ;$$

Transfer TR (Trellis), THR (Trellis History), or CMCTL (Communications Control) Registers

Syntax

```

{X|Y|XY}Rs = TRm ; /* TRx transfer */
{X|Y|XY}Rsd = TRmd ;
{X|Y|XY}Rsq = TRmq ;

{X|Y|XY}Rs = THRm ; /* THRx transfer */
{X|Y|XY}Rsd = THRmd ;
{X|Y|XY}Rsq = THRmq ;

{X|Y|XY}Rs = CMCTL ; /* CMCTL transfer */

```

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.

Status Flags

None

Options

None

CLU Instructions

Examples

```
XYR8 = TR4 ;;  
XYR1:0 = TR3:2 ;;  
XYR11:8 = TR7:4 ;;
```

```
XYR8 = THR3 ;;  
XYR1:0 = THR3:2 ;;  
XYR11:8 = THR3:0 ;;
```

```
XYR8 = CMCTL ;;
```

See Also

[Load TR \(Trellis\), TRH \(Trellis History\), or CMCTL \(Communications Control\) Registers \(CLU\)](#)


Despread With Transfer Trellis Register (Dual Operation, CLU)


Syntax

```
{X|Y|XY}TRs += DESPREAD (Rmq, THRD) ;
{X|Y|XY}Rs = TRs, TRs = DESPREAD (Rmq, THRD) {(NF)} ; (dual op.)
{X|Y|XY}Rsd = TRsd, TRsd = DESPREAD (Rmq, THRD) {(NF)} ; (dual op.)
```

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. For more detail on the `DESPREAD` function execution, see [“Despread Function” on page 4-22](#).

 Saturation is active in this instruction. For more details, see [“Saturation Option” on page 3-8](#).

 When you execute this instruction in parallel to `THR` register load the `THR` load instruction takes priority on the `THR` shift in this instruction.

Status Flags

TROV	Overflow (TROV) is calculated every cycle
TRSOV	Set whenever an overflow occurs and cleared only by <code>X/YSTAT</code> load

CLU Instructions

Options

(NF)

No flag (status) update

Examples

```
XYTR8 += DESPREAD (R7:4, THR1:0) ;;  
XYR8 = TR8, TR8 = DESPREAD (R7:4, THR1:0) ;;  
XYR1:0 = TR1:0, TR1:0 = DESPREAD (R7:4, THR1:0) ;;
```

Cross Correlations With Transfer Trellis Register (Dual Operation, CLU)

Syntax

```
{X|Y|XY}TRsa = XCORRS (Rmq, THRnq) {(CUT
<Imm>|R)}{(CLR)}{(EXT)}{(NF)} ;
{X|Y|XY}Rsq = TRbq, TRsa = XCORRS (Rmq, THRnq)
  {(CUT <Imm>|R)}{(CLR)}{(EXT)}{(NF)} ; (dual op.)
/* where TRsa = TR15:0 or TR31:16 */
```

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 byte is the real.

The history register $THRnq$ contains 63 complex elements of vector C (C[62] through C[0]), and two unused bits. Each element (C[k]) is composed of 2 bits: the imaginary bit is found in $THRnq$ bit $2k+3$, and the real bit is in $THRnq$ bit $2k+2$. The two least significant bits of $THRnq$ are not used. In each calculation pass, only elements C[22] through C[0] are used. Elements C[62] through C[23] are used in following calculation passes after each shift.

For more information, see [“Cross Correlations Function” on page 4-30](#).

Status Flags

TROV/TRSOV	If overflow occurs in one of the TR registers, TROV and TRSOV are set; otherwise, TROV is cleared and TRSOV doesn't change.
------------	---

CLU Instructions

Options

(CUT <Imm> R)	CUT selects which bits to cut from the set; if R is used instead of an immediate value, the cut value comes from the CMCTL register; if omitted, default CUT is none-includes all bits. The range of valid cut values is from -22 to 22. The CMCTL register bits 5-0 may hold the cut value in signed integer format; bits 31-6 are reserved
(CLR)	CLR clears the selected $TRsa$ register prior to summation; if omitted, $TRsa$ is not cleared
(EXT)	EXT selects 16-bit precision for operands (32-bit accumulation); if omitted, operands are 8-bit precision (16-bit accumulation)
(NF)	No flag (status) update



Saturation is active in this instruction. For more details, see [“Saturation Option” on page 3-8](#).

Examples

```
XYTR15:0 = XCORRS (R7:4, THR3:0) (CLR) ;;  
XYR11:8 = TR11:8, TR31:16 = XCORRS (R7:4,THR3:0) (CLR) ;;
```


Add/Compare/Select (CLU)

Syntax

$$\{X|Y|XY\}\{S\}TRsq = ACS (TRmd, TRnd, Rm) \{(TMAX)\} \{(NF)\} ;$$

$$\{X|Y|XY\}Rs q = TRa q, \{S\}TRsq = ACS (TRmd, TRnd, Rm)$$

$$\{(TMAX)\}\{(NF)\} ; \textit{(dual op.)}$$

Function

For $TRsq = ACS (TRmd, TRnd, Rm)$; , each *short* in Rm is added to and subtracted from the corresponding normal word in $TRmd$ and $TRnd$. The *four* results of the add and subtract are compared in trellis order, as illustrated in [Figure 4-31 on page 4-39](#).

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 short word in $TRmd$ and $TRnd$. The *eight* results of the add and subtract are compared in trellis order, as illustrated in [Figure 4-32](#).


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.

Optionally, a trellis register transfer can be added for dual operation, as in:
 $Rs q = TRa q, \{S\}TRsq = ACS (TRmd, TRnd, Rm); .$

CLU Instructions

This instruction can be executed in parallel to shifter, ALU, multiplier, and CLU register load instructions. It can not be executed in parallel to other CLU instructions of the same compute block.

For more information, see [“Cross Correlations Function” on page 4-30](#).

 Saturation is active in this instruction. For more details, see [“Saturation Option” on page 3-8](#).

Status Flags

TROV	Overflow (TROV) is calculated every cycle
TRSOV	Set whenever an overflow occurs and cleared only by X/Y_{stat} load

Options

(TMAX)	The result of the comparison is based on the following expression: $\text{Ln}(1 + e^{(- A - B)})$ where A and B are the two compared values
(NF)	No flag (status) update

Examples

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;;
```

Multiplier Instructions

The multiplier performs *multiply operations* for the processor on fixed- and floating-point data and performs *multiply-accumulate operations* for the processor on fixed-point data. This unit also performs *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 multiplier operations, status flags, conditions, and examples, see [“Multiplier” 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 2-6](#). 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*) 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 1-23](#) and [“Instruction Parallelism Rules” on page 1-27](#).

Multiplier Instructions

Multiply (Normal Word)

Syntax

$\{X|Y|XY\}Rs = Rm * Rn \{(\{U|nU\}\{I|T\}\{S\}\{NF\})\} ;$
 $\{X|Y|XY\}Rsd = Rm * Rn \{(\{U|nU\}\{I\}\{NF\})\} ;$

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 10-27).

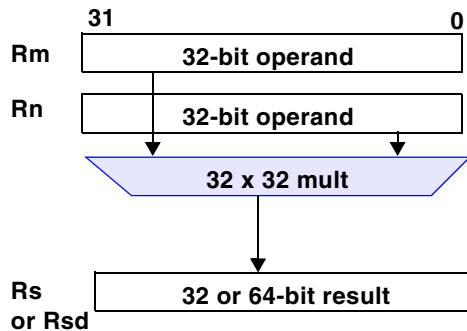


Figure 10-27. Multiply (Normal Word) Data Flow

Status Flags

- | | |
|----|------------------------------------|
| MZ | Set if all bits in result are zero |
| MN | Set if result is negative |

MV (MOS)	<p>Set according to the data format, under the following conditions (MOS unchanged if MV is cleared):</p> <p>$Rsd = Rm * Rn; \Rightarrow$ no overflow</p> <p>Fractional \Rightarrow No overflow.¹</p> <p>Signed integer \Rightarrow Upper 33 bits of multiplier result are not all zeros or all ones</p> <p>Unsigned integer \Rightarrow Upper 32 bits of multiplier result are not all zeros</p>
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)
(NF)	No flag (status) update

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 5-11](#).

¹ Except when multiplying -1.0 and -1.0 , which results in $+1.0$ (overflow condition)

Multiplier Instructions

Examples

Listing 10-1. Multiply of Two Fractions

```
XR1 = 0xFF46AC0C ;; /* xR1 = -0xB953F4 */  
XR2 = 0xAF305216 ;;  
XR0 = R1 * R2 (nUT) ;; /* xR0 = 0xFF812CA1 */
```

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.

Listing 10-2. Multiply of Two Unsigned Integers

```
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}{NF}}};
{X|Y|XY}MRa -= Rm * Rn {{(I){C|CR}{NF}}};
/* 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.

[Figure 10-28](#) 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

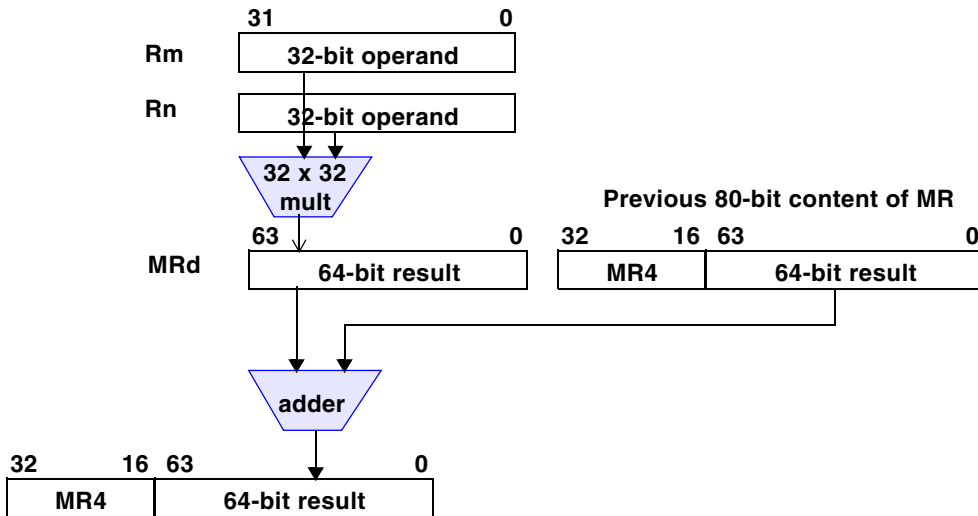


Figure 10-28. Multiply Accumulate (Normal Word) Data Flow

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 larger 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
- (NF) No flag (status) update

See “Multiplier Instruction Options” on page 5-11 for more details about available options.

Examples

Listing 10-3. Multiply -Accumulate Example 1

```

/*{X|Y|XY}MRa += Rm * Rn {{U}{I}{C|CR}} ; */
R2 = -2 ;;
R1 = 5 ;;
MR3:2 += R1 * R2 (I) ;;

```

In the example in Listing 10-3, if the previous contents of the MR3:2 register was 7, the new contents would be -3. No flags would be set.

Listing 10-4. Multiply -Accumulate Example 2

```

R0 = 0xF0060054 ;;
R2 = 0xF036AC42 ;;
MR3:2 += R0 * R2 (UI) ;;
MR3:2 += R0 * R2 (UI) ;;

```

The example in Listing 10-4 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

```

Multiplier Instructions

```
MR3 = 0xC271C629
MR4 = 0x00010000
```

Listing 10-5. Multiply -Accumulate Example 3

```
R0 = 0xF0060054 ;;
R2 = 0xF036AC42 ;;
MR1:0 += R0 * R2 (UI) ;;
MR1:0 += R0 * R2 (UI) ;;
```

The example in [Listing 10-5](#) 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.

Listing 10-6. Multiply -Accumulate Example 4

```
/* {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);;
```

For the example in [Listing 10-6](#), 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
```

Listing 10-7. Multiply -Accumulate Example 5

```
R0 = 2;;
R1 = 5;;
R2 = -10;;
R3 = 6;;
MR1:0 -= R0 * R1 (I);;
MR1:0 -= R2 * R3 (I);;
```

For the example in [Listing 10-7](#), the contents of the MR4:0 registers after the first multiply-accumulate instruction are:

```
MR0 = 0xFFFFFFFF6
MR1 = 0xFFFFFFFFF
MR2 = 0x00000000
MR3 = 0x00000000
MR4 = 0x0000FFFF
```

Multiplier Instructions

After the second multiply-accumulate, the results are:

MR0 = 0x00000032

MR1 = 0x00000000

MR2 = 0x00000000

MR3 = 0x00000000

MR4 = 0x00000000

Multiply-Accumulate With Transfer MR Register (Dual Operation, Normal Word)

Syntax

```
{X|Y|XY}Rs = MRa, MRa += Rm * Rn {{(U){I}{C|CR}{NF)}} ; (dual op.)
{X|Y|XY}Rsd = MRa, MRa += Rm * Rn {{(U){I}{C}{NF)}} ; (dual op.)
/* 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 Rm with the value in Rn , 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 (does not transfer overflow from MR4).

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 5-5](#)). 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

Multiplier Instructions

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}

Other flags are unaffected.

Options

(U)	Rm, Rn unsigned
(I)	Integer
(C)	Clear MR after transfer to register file, and prior to accumulation
(CR)	Clear and Round
(NF)	No flag (status) update

See “[Multiplier Instruction Options](#)” on page 5-11 for more details about available options.

Examples

Listing 10-8. Multiply-Accumulate With Transfer MR 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) ;;

```

In the example shown in [Listing 10-8](#), 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
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
MR4 = 0x00000000

```

The R4 register is loaded with the value 0x0000000A.

Listing 10-9. Multiply-Accumulate With Transfer MR Example 2

```

R0 = 0x12345678 ;;
R1 = 0x87654321 ;;
R2 = 0xA74EF254 ;;
R3 = 0xB4ED9032 ;;

```

Multiplier Instructions

```
R4 = MR3:2, MR3:2 += R0 * R1 ;;  
R4 = MR3:2, MR3:2 += R2 * R3 ;;
```

In the example shown in [Listing 10-9](#), 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):

```
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:

```
MR2 = 0xDC6E43C0  
MR3 = 0x22DD717B  
MR4 = 0x00000000
```

The R4 register is loaded with the value 0xEED8ED1A.

Listing 10-10. Multiply-Accumulate With Transfer MR Example 3

```
R0 = 2000;;  
R1 = 4000;;  
R2 = 3000;;  
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);;
```


In the example shown in [Listing 10-10](#), 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
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
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
MR4 = 0x00000000
```

The R5 register is loaded with the value 0xFFFFFFFF, and the R4 register is loaded with 0xFF953040.

Listing 10-11. Multiply-Accumulate With Transfer MR Example 4

```
R0 = 100 ;;
R1 = 50 ;;
R2 = 300 ;;
R3 = 70 ;;
R5:4 = MR1:0, MR1:0 += R0 * R1 (UI) ;;
R5:4 = MR1:0, MR1:0 += R2 * R3 (UI) ;;
```

Multiplier Instructions

Assume the previous contents of the MR4:0 registers are as follows:

```
MR0 = 0xFFFFFFFF500
MR1 = 0xFFFFFFFF
MR4 = 0x00000000
```

In the example shown in [Listing 10-11](#), 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
MR4 = 0x00000001
```

The R5 register is loaded with the value 0xFFFFFFFF, and the R4 register is loaded with the value 0xFFFFFFFF500.

After execution of the second instruction, the MR4:0 registers contain:

```
MR0 = 0x00005A90
MR1 = 0x00000000
MR4 = 0x00000001
```

The R5 register is loaded with the value 0xFFFFFFFF, and the R4 register is loaded with the value 0xFFFFFFFF500. As the contents of the MR4 register indicate the result in the MR1:0 register is greater than a 64-bit unsigned integer, the maximum 64 bit unsigned integer that can be represented is in the R5:4 double register. The MOS flag is not set.

Listing 10-12. Multiply-Accumulate With Transfer MR 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:

```
MR2 = 0x12345678
MR3 = 0xFFFFFFFF
MR4 = 0x7FFF0000
```

In the example shown in [Listing 10-12](#), after the execution of the first instruction containing the register move and multiply, the results of the MR4:0 register are:

```
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:

```
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

$$\{X|Y|XY\}Rsd = Rmd * Rnd \{(\{U\}\{I|T\}\{S\}\{NF\})\} ;$$
$$\{X|Y|XY\}Rsq = Rmd * Rnd \{(\{U\}\{I\}\{NF\})\} ;$$

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 10-29).

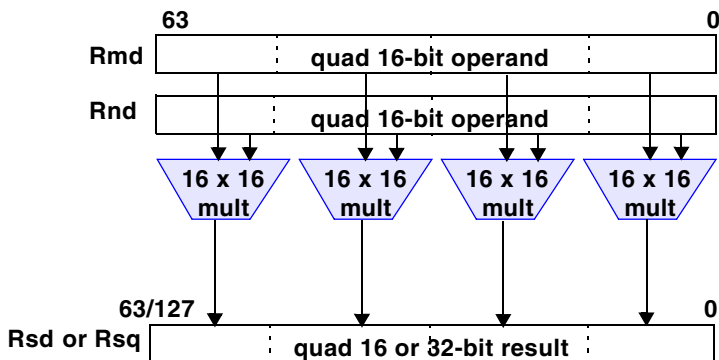


Figure 10-29. Multiply (Quad Short Word) Data Flow

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)	<p>Set according to the data format, under the following conditions (MOS unchanged if MV is cleared):</p> <p>$Rsq = Rmd * Rnd; \Rightarrow$ no overflow</p> <p>Fractional \Rightarrow No overflow¹</p> <p>Signed integer \Rightarrow Upper 17 bits of the intermediate 32-bit result are not all zeros or all ones</p> <p>Unsigned integer \Rightarrow Upper 16 bits of bits of the intermediate 32-bit result are not all zeros</p>
MU (MUS)	Cleared
MI	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)
(NF)	No flag (status) update

See “[Multiplier Instruction Options](#)” on page 5-11 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

Examples

Listing 10-13. Multiply (Quad-Short Word) Example 1

```
XR0 = 0x42861234 ;;
XR1 = 0x782F4217 ;;
XR2 = 0x8AC90FEA ;;
XR3 = 0x724D76EB ;;
XR5:4 = R1:0 * R3:2 (T) ;; /* xR5 = 6B52 3D66 , xR4 = C314 0243 */
```

In the example shown in [Listing 10-13](#), the code 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.

Listing 10-14. Multiply (Quad-Short Word) Example 2

```
XR0 = 0x00060004 ;;
XR1 = 0x00030007 ;;
XR2 = 0x00090000 ;;
XR3 = 0x00050008 ;;
XR7:4 = R1:0 * R3:2 (I) ;;
/* xR7 = 0x0000000F, xR6 = 0x00000038, xR5 = 0x00000036
xR4 = 0x00000000 */
```

In the example shown in [Listing 10-14](#), the code 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}MR3:0 += Rmd * Rnd {{(U){I}{C|CR}{NF}}};
{X|Y|XY}MRb += Rmd * Rnd {{(U){I}{C}{NF}}};
/* where MRb is either MR1:0 or MR3:2 */
```

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 40-bit precision. When using either MR3:2 or MR1:0, the four multiplier results are accumulated in two MR registers at 20-bit precision.

The extra bits from the multiplications are stored in MR4. Saturation is always active. See [Figure 10-30](#) and [Figure 10-31](#).

Refer to “[Multiplier Result Overflow \(MR4\) Register](#)” on page 5-19 for a description of the fields in MR4.

Multiplier Instructions

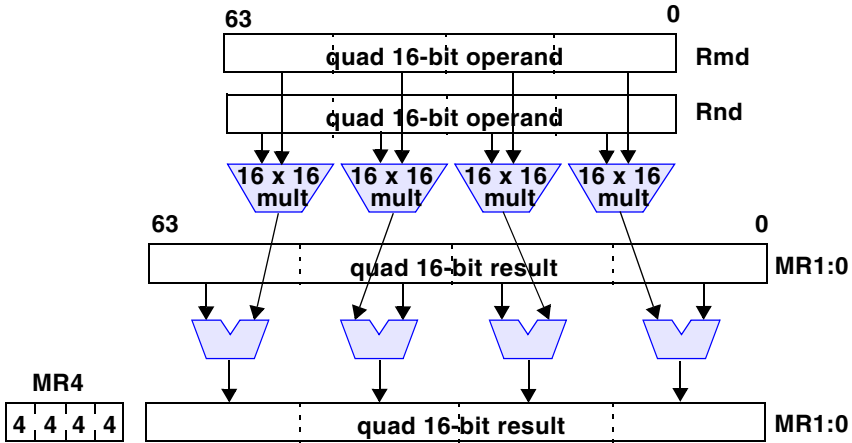


Figure 10-30. Quad 16-Bit Result

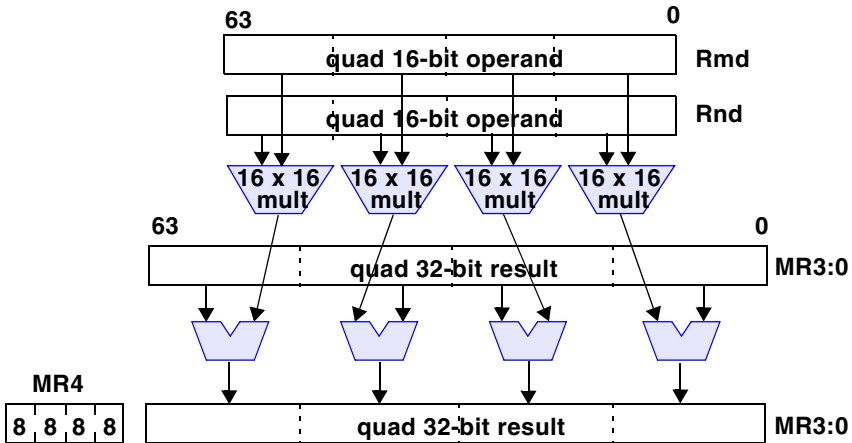


Figure 10-31. Quad 32-Bit Result

Status Flags

MZ	Unaffected
MN	Unaffected
MV	Unaffected
MU (MUS)	Unaffected
MOS	Set according to the final result of the sum and data type and size. (See the 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}

Multiplier Instructions

Options

(U)	<i>Rm, Rn</i> unsigned
(I)	Integer
(C)	Clear MR prior to accumulation
(CR)	Clear and Round
(NF)	No flag (status) update

See “[Multiplier Instruction Options](#)” on page 5-11 for more details about available options.

Examples

Listing 10-15. Multiply-Accumulate (Quad-Short Word) Example 1

```
R0 = 0x70000004 ;;
R1 = 0x06000002 ;;
R2 = 0xF0000005 ;;
R3 = 0xD0000006 ;;
MR3:0 += R1:0 * R3:2 (UIC) ;;
MR3:0 += R1:0 * R3:2 (UI) ;;
```

In the example shown in [Listing 10-15](#), the example shows a quad 16-bit multiply-accumulate where the result of each multiply is stored at full 32-bit precision.

The results of the MR4:0 registers after the second multiply-accumulate are:

```
MR0 = 0x00000028
MR1 = 0xD2000000
MR2 = 0x00000018
MR3 = 0x09C00000
MR4 = 0x00000000
```

No flags were set during this example.

Listing 10-16. Multiply-Accumulate (Quad-Short Word) Example 2

```
R0 = 0x70000004 ;;
R1 = 0x06000002 ;;
R2 = 0xF0000005 ;;
R3 = 0xD0000006 ;;
MR3:2 += R1:0 * R3:2 (UIC) ;;
MR3:2 += R1:0 * R3:2 (UI) ;;
```

In the example shown in [Listing 10-16](#), the results of the MR4:0 registers after the second multiply-accumulate are as follows:

```
MR0 = 0x00000000
MR1 = 0x00000000
MR2 = 0xFFFFF028
MR3 = 0xFFFFF018
MR4 = 0xF0F00000
```

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 ([Listing 10-17](#)) is identical to the previous, except it uses the MR1:0 registers for the result instead of MR3:2.

Multiplier Instructions

Listing 10-17. Multiply-Accumulate (Quad-Short Word) Example 3

```
R0 = 0x70000004;;  
R1 = 0x06000002;;  
R2 = 0xF0000005;;  
R3 = 0xD0000006;;  
MR1:0 += R1:0 * R3:2 (UIC);;  
MR1:0 += R1:0 * R3:2 (UI);;
```

In the example shown in [Listing 10-17](#), the results of the MR4:0 registers after the second multiply-accumulate are as follows:

```
MR0 = 0xFFFF0028  
MR1 = 0xFFFF0018  
MR2 = 0x00000000  
MR3 = 0x00000000  
MR4 = 0x0000F0F0
```

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 the positioning of the set bits in the MR4 register. This is due to the MR4 register only containing 4 bits worth of overflow for the accumulation (20 bits in total).

Multiply-Accumulate With Transfer MR Register (Dual Operation, Quad-Short Word)

Syntax

```
{X|Y|XY}Rsd = MRb, MR3:0 += Rmd * Rnd {{{I}}{C|CR}{NF}}; (dual op.)
{X|Y|XY}Rsd = MRb, MRb += Rmd * Rnd {{{I}}{C}{NF}} ; (dual op.)
/* where MRb is either MR1:0 or MR3:2 */
```

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 *MRb*, and transfers the contents of *MRb* into *Rs*.

When using *MR[3:0]*, the four multiplication results are accumulated in the four MR registers at full 40-bit precision. When using either *MR[3:2]* or *MR[1:0]*, the four multiplier results are accumulated in two MR registers at 20-bit precision.

The clear option (C) clears the *MRb* register prior to adding the new multiplication result. The operations are always saturated. The *MRb* to register file transfer is always truncated (overflow is not transferred).

For results in a dual register (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 5-19. 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 5-19. 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*.

Multiplier Instructions

When the MR registers that serve as a source for the transfer are an *MRb* 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 the 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 after transfer of truncated value to register file (prior to accumulation)
(CR)	Clear MR after transfer of rounded value to register file (prior to accumulation)
(NF)	No flag (status) update

See “Multiplier Instruction Options” on page 5-11 for more details about available options.

Examples

Listing 10-18. Multiply-Accumulate With Transfer MR (Dual Operation, Quad-Short Word) Example 1

```

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) ;;

```

Multiplier Instructions

If the previous contents of the MR register were:

```
MR0 = 0x00800050
MR1 = 0x01000200
MR4 = 0x00000001
```

In the example shown in [Listing 10-18](#), 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
MR4 = 0x00000000
```

After execution of the next instruction, the above registers are set to the following:

```
R4 = 0x00200015
R5 = 0x000C0005
MR0 = 0x0040002A
MR1 = 0x0018000A
MR4 = 0x00000000
```

No flags are set during the process.

Listing 10-19. Multiply-Accumulate With Transfer MR (Dual Operation, Quad-Short Word) Example 2

```
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:

```
MR2 = 0x00300040
MR3 = 0x00500060
MR4 = 0x00000001
```

In the example shown in [Listing 10-19](#), after execution of the first register transfer and multiply instruction, the values of all modified registers are:

```
R4 = 0x00300040
R5 = 0x00500060
MR2 = 0x00200015
MR3 = 0x000C0005
```

After execution of the next instruction, the above registers are set to the following:

```
R4 = 0x00200015
R5 = 0x000C0005
MR2 = 0x0040002A
MR3 = 0x0018000A
```

No flags are set during the process.

Listing 10-20. Multiply-Accumulate With Transfer MR (Dual Operation, Quad-Short Word) Example 3

```
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) ;;
```

Multiplier Instructions

If the previous contents of the MR register were:

```
MR0 = 0x00000050
MR1 = 0x00000200
MR2 = 0x00000040
MR3 = 0x00000060
MR4 = 0x00000000
```

In the example shown in [Listing 10-20](#), after execution of the first register transfer and multiply instruction, the values of all modified registers are:

```
R4 = 0x02000050
R5 = 0x00600040
MR0 = 0x000C6000
MR1 = 0x00085000
MR2 = 0x00000005
MR3 = 0x0000000C
MR4 = 0x00000000
```

After execution of the next instruction, the above registers are set to the following:

```
R4 = 0xFFFFFFFF
R5 = 0x000C0005
MR0 = 0x0018C000
MR1 = 0x0000000A
MR2 = 0x0000000A
MR3 = 0x00000018
MR4 = 0x00000000
```

No flags are set during the process.

Complex Multiply-Accumulate (Short Word)

Syntax

```
{X|Y|XY}MRa += Rm ** Rn {{(I){C|CR}{J}{NF}}};
{X|Y|XY}MRa -= Rm ** Rn {{(I){C|CR}{J}{NF}}};
/* 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 or subtracts 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 5-19 and Figure 10-32.

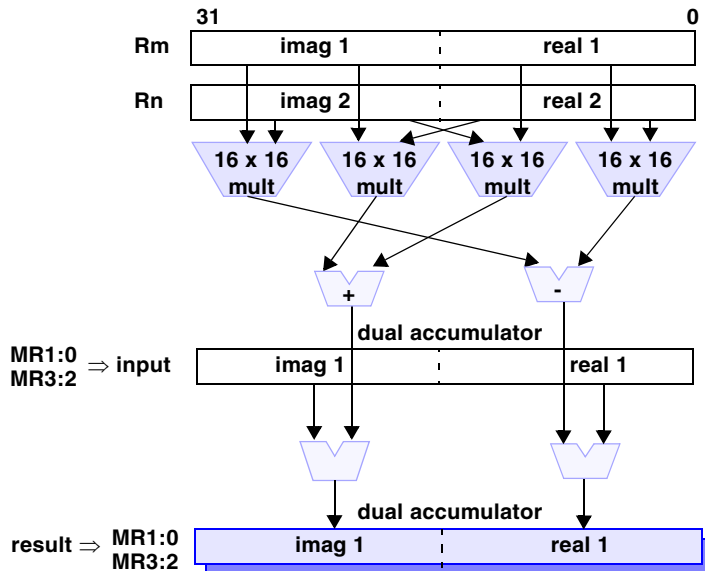


Figure 10-32. Complex Multiply-Accumulate (Short Word) Data Flow

Multiplier Instructions

There are eight overflow guard 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 the following.) <ul style="list-style-type: none">• 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}.

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
(NF)	No flag (status) update

See “[Multiplier Instruction Options](#)” on page 5-11 for more details about available options.

Examples**Listing 10-21. Complex Multiply-Accumulate (Short Word) Example 1**

```
R0 = 0xFFF50009;;
R1 = 0x00060004;;
MR1:0 += R0 ** R1 (IC);;
```

If the previous contents of the MR register were:

```
MR0 = 0x00800050
MR1 = 0x01000200
MR4 = 0x00000001
```

In the example shown in [Listing 10-21](#), after execution of the complex multiply-accumulate instruction, the values of all modified registers are:

```
MR0 = 0x00000066
MR1 = 0x0000000A
MR4 = 0x00000000
```

No flags are set.

Multiplier Instructions

Listing 10-22. Complex Multiply-Accumulate (Short Word) Example 2

```
R0 = 0xFFFF50009;;  
R1 = 0x00060004;;  
MR3:2 += R0 ** R1 (ICJ);;
```

If the previous contents of the MR register were:

```
MR2 = 0x00000040  
MR3 = 0x00000060  
MR4 = 0x00000001
```

In the example shown in [Listing 10-22](#), after execution of the complex multiply-accumulate instruction, the values of all modified registers are:

```
MR2 = 0xFFFFFFE2  
MR3 = 0xFFFFF9E  
MR4 = 0xFFFF0001
```

No flags are set during the operation.

Complex Multiply-Accumulate With Transfer MR Register (Dual Operation, Short Word)

Syntax

```
{X|Y|XY}Rs = MRa, MRa += Rm ** Rn {{(I){C|CR}{J}{NF}}}; (dual op.)
{X|Y|XY}Rsd = MRa, MRa += Rm ** Rn {{(I){C}{J}{NF}}}; (dual op.)
/* 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 or subtracts the result to the specified MR registers, and transfers the previous contents of MRa to Rs . The C (clear) option prevents the old value of MRa from being added to the new multiplication result. The operations are always truncated.

As shown in [Figure 10-33](#), 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 .

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 40-bit real signed integer component stored in MR2 or MR0 exceeds 2^{15} or -2^{15} , 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 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.

Multiplier Instructions

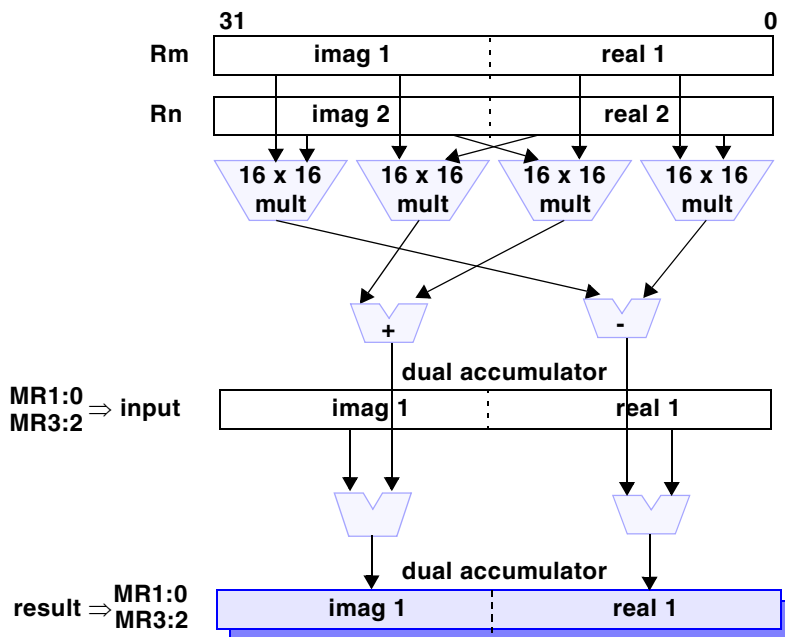


Figure 10-33. Complex Multiply-Accumulate With Transfer MR Register (Dual Operation, Short Word) Data Flow

Status Flags

MZ	Unaffected
MN	Unaffected
MV	Unaffected
MU	Unaffected
MUS	Unaffected

MV (MOS) Set according to the final result of the sum and data type and size. (See the following.)

- 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}

Options

(I)	Integer
(C)	Clear MR after transfer to register file, and prior to accumulation
(CR)	Clear and Round
(J)	Conjugate
(NF)	No flag (status) update

See “[Multiplier Instruction Options](#)” on page 5-11 for more details about available options.

Examples

Listing 10-23. Complex Multiply-Accumulate With Transfer MR (Dual Operation, Short Word) Example 1

```
R0 = 0xFFF50009 ;;
R1 = 0x00060004 ;;
R4 = MR1:0, MR1:0 += R0 ** R1 (IC) ;;
R4 = MR1:0, MR1:0 += R0 ** R1 (I) ;;
```

Multiplier Instructions

If the previous contents of the MR register were:

```
MR0 = 0x00800050
MR1 = 0x01000200
MR4 = 0x00000001
```

In the example shown in [Listing 10-23](#), 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
MR4 = 0x00000000
```

No flags are set during the operation.

After the next instruction is executed, the contents become:

```
R4 = 0x000A0066
MR0 = 0x000000CC
MR1 = 0x00000014
MR4 = 0x00000000
```

No flags are set.

Listing 10-24. Complex Multiply-Accumulate With Transfer MR (Dual Operation, Short Word) Example 2

```
R0 = 0xFFFF50009;;
R1 = 0x00060004;;
R4 = MR3:2, MR3:2 += R0 ** R1 (ICJ);;
R4 = MR3:2, MR3:2 += R0 ** R1 (IJ);;
```

If the previous contents of the MR register were:

```
MR2 = 0x00000040
MR3 = 0x00000060
MR4 = 0x00000001
```

In the example shown in [Listing 10-24](#), after execution of the first register transfer and complex multiply-accumulate instruction, the values of all modified registers are:

```
R4 = 0x00600040
MR2 = 0xFFFFFFFFE2
MR3 = 0xFFFFFFFF9E
MR4 = 0xFFFFF0001
```

No flags are set during the operation.

After the next instruction is executed, the contents become:

```
R4 = 0xFF9EFFE2
MR2 = 0xFFFFF4C4
MR3 = 0xFFFFF3C
MR4 = 0xFFFFF0001
```

No flags are set.

Listing 10-25. Complex Multiply-Accumulate With Transfer MR (Dual Operation, Short Word) Example 3

```
R0 = 0xFFFF50009;;
R1 = 0x00060004;;
R5:4 = MR1:0, MR1:0 += R0 ** R1 (IC);;
R5:4 = MR1:0, MR1:0 += R0 ** R1 (I);;
```

Multiplier Instructions

If the previous contents of the MR register were:

```
MR0 = 0x00800050
MR1 = 0x01000200
MR4 = 0x00000001
```

In the example shown in [Listing 10-25](#), 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
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
MR4 = 0x00000000
```

No flags are set.

Listing 10-26. Complex Multiply-Accumulate With Transfer MR (Dual Operation, Short Word) Example 4

```
R0 = 0xFFFF50009;;
R1 = 0x00060004;;
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:

MR2 = 0x00000040

MR3 = 0x00000060

MR4 = 0x00000001

In the example shown in [Listing 10-26](#), after execution of the first register transfer and complex multiply-accumulate instruction, the values of all modified registers are:

R4 = 0x00000040

R5 = 0x00000060

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

MR2 = 0xFFFFF0C4

MR3 = 0xFFFFF03C

MR4 = 0xFFFFF0001

No flags are set.

Multiplier Instructions

Multiply (Floating-Point, Normal/Extended Word)

Syntax

```
{X|Y|XY}FRs = Rm * Rn {{(T){NF}}} ; /* Normal, 32-bit */
```

```
{X|Y|XY}FRsd = Rmd * Rnd {{(T){NF}}} ; /* Extended, 40-bit */
```

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
MVS	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
MIS	Set if either input is a NAN; otherwise unaffected

Options

()	Round
(T)	Truncate
(NF)	No flag (status) update

Examples

Listing 10-27. Multiply (Floating-Point, Norm./Ext. Word) Example 1

```
R0 = 0x3AF5EC81 ;; /* 0.00187625 */  
R1 = 0x3AF58CDF ;; /* 0.0018734 */  
FR2 = R0 * R1 ;;
```

In the example shown in [Listing 10-27](#), after executing the floating-point multiply, the result stored in R2 is 0x366BE2AB (3.51497E-006).

Multiplier Instructions

Load/Transfer MR (Multiplier Result) Register

Syntax

```
{X|Y|XY}{S|L}MRa = Rmd {({SE|ZE}{NF})} ;  
{X|Y|XY}MR4 = Rm {(NF)} ;  
{X|Y|XY}{S}Rsd = MRa {({U}{S}{NF})} ;  
{X|Y|XY}Rsq = MR3:0 {({U}{S}{NF})} ;  
{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 destination MR register can hold a saturated result that is larger than the source *Rmd*, the DSP can sign extend (SE) or zero extend (ZE) the data into the MR4 register. Operation is as follows:

- *SMRa* denotes the transfer of four 16-bit short words into four 20-bit results
- *MRA* denotes the transfer of two 32-bit normal words into two 40-bit result
- *LMRa* denotes the transfer of one 64-bit long word into one 80-bit result

Because the source MR register can hold a saturated result that is larger than the destination Rs , the DSP has a mechanism 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 $MR4$.

- 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 10-34](#), [Figure 10-35](#), and [Figure 10-36](#).

$Rsd = MR1:0$; see [Figure 10-34](#)

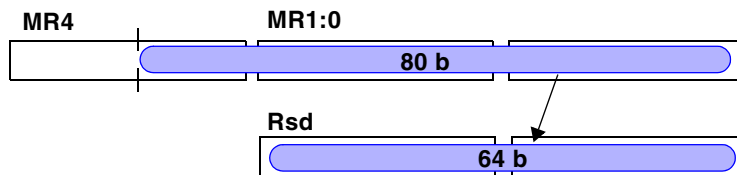


Figure 10-34. Move One 80-Bit Result Into One 64-Bit Register

$SRsd = MR1:0$; see [Figure 10-35](#)

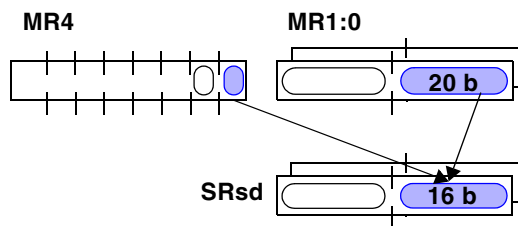


Figure 10-35. Move Four 20-Bit Results Into Four 16-Bit Shorts

Multiplier Instructions

$Rsq = MR3:0$; see *Figure 10-36*

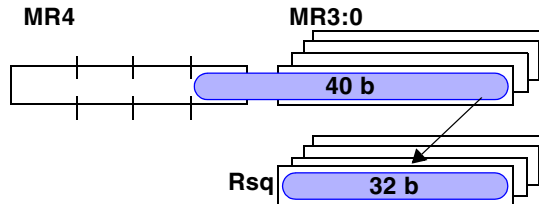


Figure 10-36. Move Four 40-Bit Results Into Four 32-Bit Registers

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)
MI	Cleared

Options

(SE)	Sign extended
(ZF)	Zero filled
(U)	Unsigned
(S)	Saturate
(NF)	No flag (status) update

See “Multiplier Instruction Options” on page 5-11 for more details about available options.

Examples

For the examples in this section, the current contents of MR register are:

```
MR0 = 0x00000000
MR1 = 0x11111111
MR2 = 0x22222222
MR3 = 0x0FFFFFFF5
MR4 = 0x0000FF00
```

Listing 10-28. Load/Transfer MR (Multiplier Result) Example 1

```
R1:0 = MR3:2 (S);;
```

In the example shown in [Listing 10-28](#), after executing the MR3:2 transfer instruction, the contents of R1:0 are:

```
R0 = 0x22222222
R1 = 0x0FFFFFFF5
```

No flags are set.

Listing 10-29. Load/Transfer MR (Multiplier Result) Example 2

```
R3:2 = MR3:2;;
```

In the example shown in [Listing 10-29](#), after executing the MR3:2 transfer instruction, the contents of R3:2 are:

```
R2 = 0x22222222
R3 = 0x0FFFFFFF5
```

No flags are set.

Multiplier Instructions

Listing 10-30. Load/Transfer MR (Multiplier Result) Example 3

```
R1:0 = MR1:0 (S);;
```

In the example shown in [Listing 10-30](#), after executing the MR1:0 transfer instruction, the contents of R1:0 are:

```
R0 = 0x00000000
```

```
R1 = 0x80000000
```

The MV, MN, and MOS flag are all set.

Listing 10-31. Load/Transfer MR (Multiplier Result) Example 4

```
R3:2 = MR1:0 (U);;
```

In the example shown in [Listing 10-31](#), after executing the MR1:0 transfer instruction, the contents of R3:2 are:

```
R2 = 0x00000000
```

```
R3 = 0x11111111
```

The MV and MOS flags would be set.

Listing 10-32. Load/Transfer MR (Multiplier Result) Example 5

```
SR1:0 = MR3:2 (S);;
```

In the example shown in [Listing 10-32](#), after executing the MR3:2 transfer instruction, the contents of R1:0 are:

```
R0 = 0x22222222
```

```
R1 = 0x0FFF7FFF
```

The MV and MOS flags are set.

Listing 10-33. Load/Transfer MR (Multiplier Result) Example 6

```
SR3:2 = MR3:2 (U);;
```

In the example shown in [Listing 10-33](#), after executing the MR3:2 transfer instruction, the contents of R3:2 are:

```
R2 = 0x22222222
R3 = 0x0FFFFFFF5
```

No flags are set.

Listing 10-34. Load/Transfer MR (Multiplier Result) Example 7

```
SR1:0 = MR1:0 (S);;
```

In the example shown in [Listing 10-34](#), after executing the MR1:0 transfer instruction, the contents of R1:0 are:

```
R0 = 0x00000000
R1 = 0x80008000
```

The MV, MN, MZ, and MOS flags are all set.

Listing 10-35. Load/Transfer MR (Multiplier Result) Example 8

```
SR3:2 = MR1:0 (U);;
```

In the example shown in [Listing 10-35](#), after executing the MR1:0 transfer instruction, the contents of R3:2 are:

```
R2 = 0x00000000
R3 = 0x11111111
```

The MZ, MV, and MOS flags are set.

Multiplier Instructions

Listing 10-36. Load/Transfer MR (Multiplier Result) Example 9

```
R3:0 = MR3:0 (S);;
```

In the example shown in [Listing 10-36](#), after executing the MR3:0 transfer instruction, the contents of R3:0 are:

```
R0 = 0x00000000  
R1 = 0x80000000  
R2 = 0x22222222  
R3 = 0x0FFFFFFF5
```

The MV, MN, MZ, and MOS flags are all set.

Listing 10-37. Load/Transfer MR (Multiplier Result) Example 10

```
R3:0 = MR3:0 (U);;
```

In the example shown in [Listing 10-37](#), after executing the MR3:0 transfer instruction, the contents of R3:0 are:

```
R0 = 0x00000000  
R1 = 0x11111111  
R2 = 0x22222222  
R3 = 0x0FFFFFFF5
```

The MV, MZ, and MOS flags are all set.

Listing 10-38. Load/Transfer MR (Multiplier Result) Example 11

```
R0 = 0x11111111 ;;  
R1 = 0x22222222 ;;  
R2 = 0x33333333 ;;  
R3 = 0x44444444 ;;  
R5 = 0x55555555 ;;  
MR3:2 = R1:0 ;;
```

```
MR1:0 = R3:2 ;;
```

```
MR4 = R5 ;;
```

In the example shown in [Listing 10-38](#), after executing the MR3:2, MR1:0, and MR4 load instructions, all set multiplier flags (except sticky) are cleared.

Multiplier Instructions

Extract Words From MR (Multiplier Result) Register

Syntax

```
{X|Y|XY}Rsd = SMRb {{{U}}NF}} ; /* extract 2 short words */
{X|Y|XY}LRsd = MRb {{{U}}NF}} ; /* extract 1 normal word */
/* where MRb is either MR0, MR1, MR2, or MR3 */

{X|Y|XY}Rsq = SMRa {{{U}}NF}} ; /* extract 4 short words */
{X|Y|XY}LRsq = MRa {{{U}}NF}} ; /* extract 2 normal words */
{X|Y|XY}QRsq = LMRa {{{U}}NF}} ; /* extract 1 long word */
/* where MRa is either MR1:0 or MR3:2 */
```

Function

These instructions extract the value(s) in the source (to right of =) *MRb* or *MRa* register and extract the related guard bits from the MR4 register then store the extracted data in the destination (to left of =) *Rsd* or *Rsq* register. If the (U) option is not specified, the data is sign extended.

As indicated in the comments for the syntax summary, depending on the options used, the source *MRb* or *MRa* register and destination *Rsd* or *Rsq* register can hold different numbers of short (16-bit), normal (32-bit), or long (64-bit) words. For more information, see [“Examples” on page 10-164](#) and the bit placement illustrations in [Figure 10-37](#), [Figure 10-38](#), [Figure 10-39](#), [Figure 10-40](#), and [Figure 10-41](#).

$XRsd = SMR0$; see *Figure 10-37*

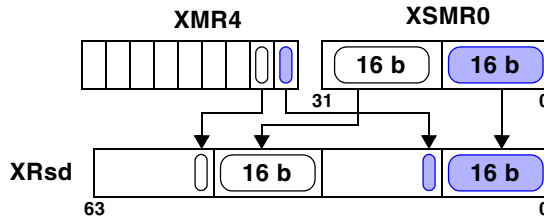


Figure 10-37. Extract/Store Two Short Values and Guard Bits

$XLRs d = MR0$; see *Figure 10-38*

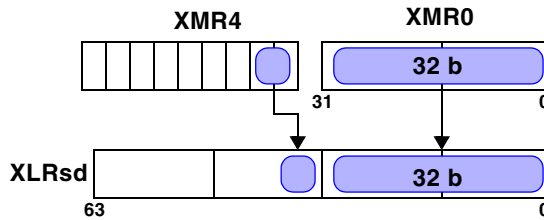


Figure 10-38. Extract/Store One Normal Values and Guard Bits

Multiplier Instructions

$XRsq = SMR1:0$; see [Figure 10-39](#)

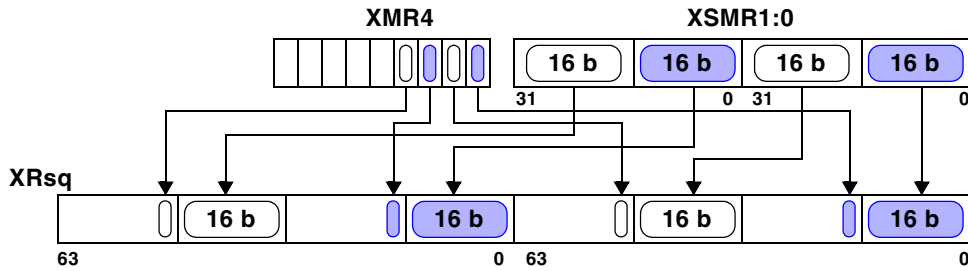


Figure 10-39. Extract/Store Four Short Values and Guard Bits

$XLRsq = MR1:0$; see [Figure 10-40](#)

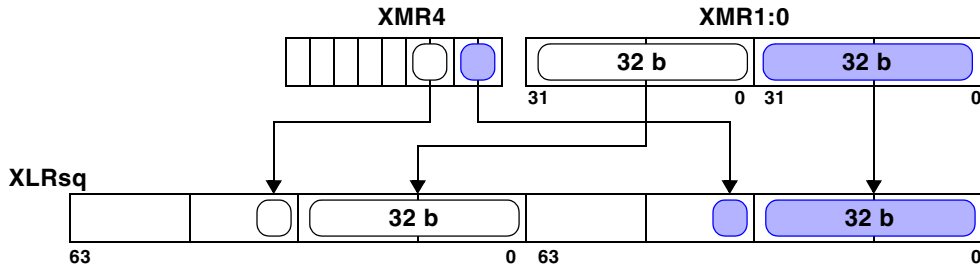


Figure 10-40. Extract/Store Two Normal Values and Guard Bits

$XQRsq = LMR1:0$; see *Figure 10-41*

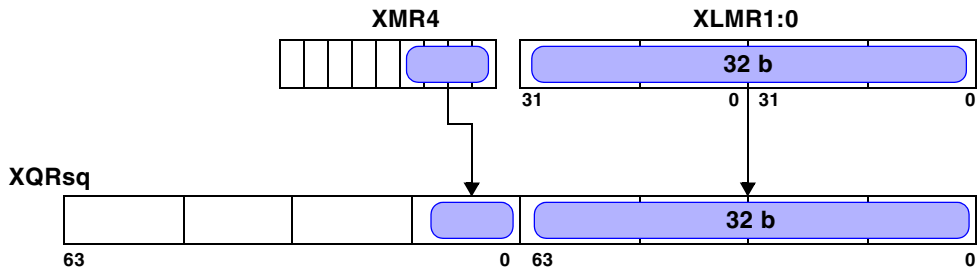


Figure 10-41. Extract/Store One Long Value and Guard Bits

Status Flags

MZ	Set if operand (or one of the operands if data type is smaller than data size (for example, $Rsd = SMR$) is zero
MN	Set if operand is negative (not set for unsigned operation, (U) option)
MV	Cleared (overflow can not occur on these instructions)
MOS	Unchanged (overflow can not occur on these instructions)
MU (MUS)	Cleared (unchanged)

Options

(U)	Unsigned
(NF)	No flag (status) update

Multiplier Instructions

See “[Multiplier Instruction Options](#)” on page 5-11 for more details about available options.

Examples

Listing 10-39. Extract Words From MR (Multiplier Result) Example 1

```
R1:0 = SMR2 ;;  
/* If MR2 = 0x30f26ab2 and MR4 = 0x00d30000 */  
/* then R1 = 0xffffd30f2 and R0 = 0x00036ab2 */
```

The instruction `Rsd = SMR0/1/2/3 (u)`; (see [Listing 10-39](#)) extracts the two short values of a single MR register and their guard bits in MR4 to a double register. If option (u) is used, the extension is zero extension. If (u) is added in [Listing 10-39](#), `R1 = 0x000d30f2` and `R0 = 0x00036ab2`.

The instruction `Rsq = SMR3:2/1:0 (u)`; similar to the instruction `Rsd = SMR0/1/2/3 (u)`; and is executed for 4 shorts in double MR register and their MR4 extensions.

Listing 10-40. Extract Words From MR (Multiplier Result) Example 2

```
LR1:0 = MR2 ;;  
/* If MR2 = 0x30f26ab2 and MR4 = 0x00d30000 */  
/* then R1:0 = 0xffffffffd330f26ab2 */
```

The instruction `LRsd = MR0/1/2/3 (u)`; (see [Listing 10-40](#)) extracts the word value of a single MR register and its guard bits in MR4 to a double register. If option (u) is used, the extension is zero extension. If (u) is added in [Listing 10-40](#), `R1:0 = 0x000000d330f26ab2`.

The instruction `LRsq = MR3:2/1:0 (u)`; is similar to the instruction `LRsd = MR0/1/2/3 (u)`; and is executed for 2 word values in double MR register and their MR4 extensions.

Compact MR (Multiplier Result) Register

Syntax

```
{X|Y|XY}Rs = COMPACT MRa {{{U}{I}{S}{NF}}};
{X|Y|XY}SRsd = COMPACT MR3:0 {{{U}{I}{S}{NF}}};
/* 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 if fractional. The compact `MRa` operation appears in [Figure 10-42](#).

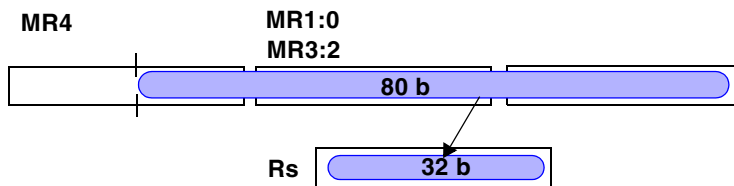


Figure 10-42. Compact MRa

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 if fractional. The compact `MR3:0` operation appears in [Figure 10-43](#).

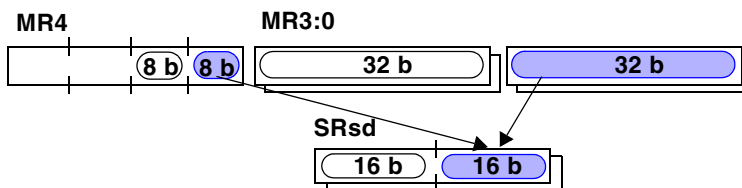


Figure 10-43. Compact MR3:0

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
(U)	Unsigned
(NF)	No flag (status) update

See “[Multiplier Instruction Options](#)” on page 5-11 for more details about available options.

Examples

For the examples in [Listing 10-41](#) and [Listing 10-42](#), the current contents of MR register are:

```
MR2 = 0x22222222
MR3 = 0x0FFFFFFF5
MR4 = 0x0000FF00
```

Listing 10-41. Compact MR (Multiplier Result) Example 1

```
R0 = COMPACT MR3:2 (UI) ;;
```

In the example shown in [Listing 10-41](#), after executing the COMPACT, the contents of R0 = 0x22222222, and the MV and MOS flags are set.

Listing 10-42. Compact MR (Multiplier Result) Example 2

```
R0 = COMPACT MR3:2 (U) ;;
```

In the example shown in [Listing 10-42](#), after executing the COMPACT, the contents of R0 = 0x0FFFFFFF5, and no flags are set.

For the examples in [Listing 10-43](#) and [Listing 10-44](#), the current contents of MR register are:

```
MR0 = 0x00000000
MR1 = 0x11111111
MR2 = 0x22222222
MR3 = 0x0FFFFFFF5
MR4 = 0x0000FF00
```

Multiplier Instructions

Listing 10-43. Compact MR (Multiplier Result) Example 3

```
R0 = COMPACT MR1:0 (UI) ;;
```

In the example shown in [Listing 10-43](#), after executing the COMPACT, the contents of R0 = 0x00000000, and the MZ, MV, and MOS flags are set.

Listing 10-44. Compact MR (Multiplier Result) Example 4

```
R0 = COMPACT MR1:0 (U) ;;
```

In the example shown in [Listing 10-44](#), after executing the COMPACT, the contents of R0 = 0x11111111, and the MV, and MOS flags are set.

For the examples in [Listing 10-45](#) and [Listing 10-46](#), the current contents of MR register are:

```
MR0 = 0x00000000  
MR1 = 0x11111111  
MR2 = 0x22222222  
MR3 = 0x0FFFFFFF  
MR4 = 0x0000FF00
```

Listing 10-45. Compact MR (Multiplier Result) Example 5

```
SR1:0 = COMPACT MR3:0 (UI) ;;
```

In the example shown in [Listing 10-45](#), after executing the COMPACT, the contents of R0 = 0x11110000, R1 = 0xFFF52222, and the MZ, MV, and MOS flags are set.

Listing 10-46. Compact MR (Multiplier Result) Example 6

```
SR1:0 = COMPACT MR3:0 (S) ;;
```

In the example shown in [Listing 10-46](#), after executing the COMPACT, the contents of R0 = 0x80000000, R1 = 0x0FFF2222, and 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 shifter operations, status flags, conditions, and examples, see [“Shifter” on page 6-1](#).

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 2-6](#). 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 1-23](#) and [“Instruction Parallelism Rules” on page 1-27](#).

Arithmetic/Logical Shift

Syntax

$$\{X|Y|XY\}\{B|S\}Rs = \text{LSHIFT}|\text{ASHIFT } Rm \text{ BY } Rn|<Imm> \{(NF)\} ;$$

$$\{X|Y|XY\}\{B|S|L\}Rsd = \text{LSHIFT}|\text{ASHIFT } Rmd \text{ BY } Rn|<Imm> \{(NF)\} ;$$

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 “Register File Registers” on page 2-6.

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 6-1.

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

Options

(NF)	No flag (status) update
------	-------------------------

Shifter Instructions

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

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

Shifter Instructions

Rotate

Syntax

$$\{X|Y|XY\}Rs = \text{ROT } Rm \text{ BY } Rn|<Imm6> \{(NF)\} ;$$
$$\{X|Y|XY\}\{L\}Rsd = \text{ROT } Rmd \text{ BY } Rn|<Imm> \{(NF)\} ;$$

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 “[Register File Registers](#)” on [page 2-6](#).

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 6-1](#).

Status Flags

SZ	Set if result is zero
SN	Equals the MSB of the result

Options

(NF)	No flag (status) update
------	-------------------------

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;

Shifter Instructions

Field Extract

Syntax

$\{X|Y|XY\}Rs = \text{FEXT } Rm \text{ BY } Rn|Rnd \{(\{SE\}\{NF\})\} ;$
 $\{X|Y|XY\}LRsd = \text{FEXT } Rmd \text{ BY } Rn|Rnd \{(\{SE\}\{NF\})\} ;$

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 10-44).

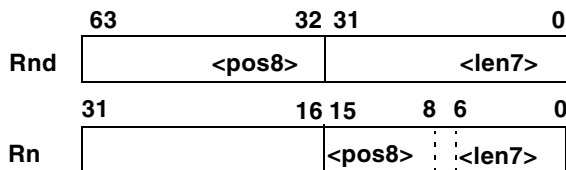


Figure 10-44. Len7 and Pos8 Fields for Dual and Single Word Registers

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 (Figure 10-45).

The L prefix denotes long operand type and d denotes operand size—see “Register File Registers” on page 2-6.

Status Flags

- SZ Set if result is zero
- SN Equals the MSB of the result

Options

- (SE) Sign extended
- (NF) No flag (status) update

Examples

FEXT extracts the field specified in *Rn* from *Rm* and places it in *Rs*:

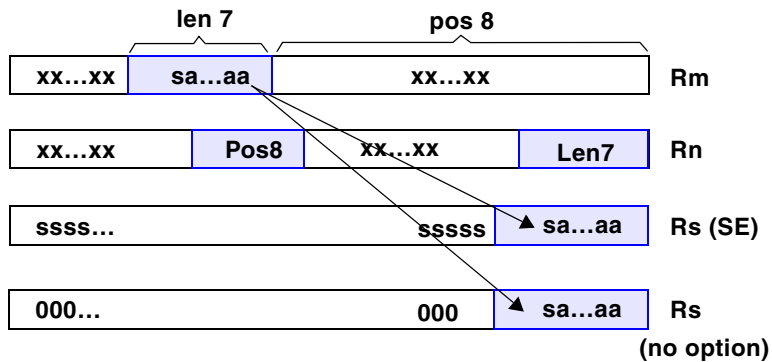


Figure 10-45. FEXT Instruction Performed on Single Word Registers

Shifter Instructions

Field Deposit

Syntax

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

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 10-46).

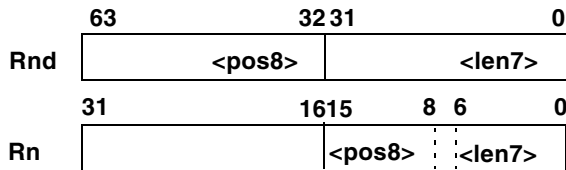


Figure 10-46. Len7 and Pos8 Fields for Dual and Single Word Registers

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 Figure 10-47.

The *L* prefix denotes long operand type and *d* denotes operand size—see “Register File Registers” on page 2-6.

Status Flags

- SZ Set if result is zero
- SN Equals the MSB of the result

Options

- (SE) Sign extended
- (ZF) Zero filled
- (NF) No flag (status) update

Example

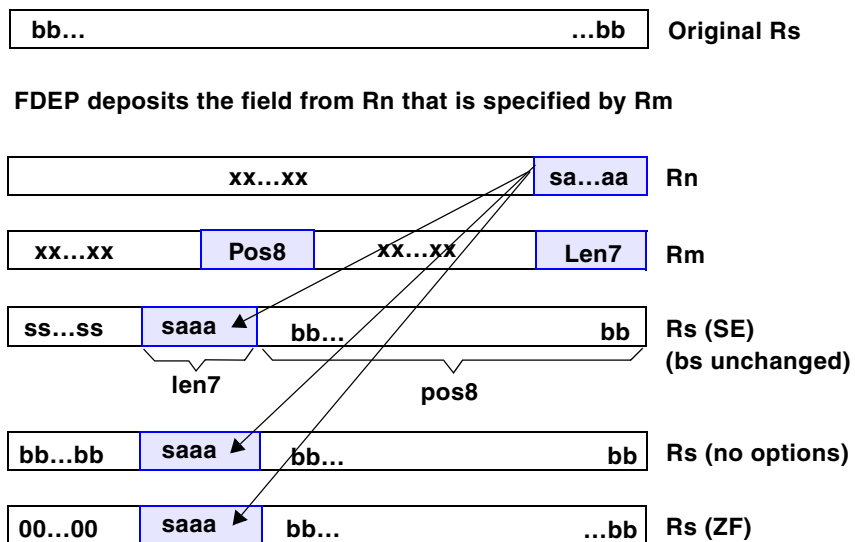


Figure 10-47. FDEP Instruction Performed on Single Word Registers

Shifter Instructions

Field/Bit Mask

Syntax

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

Function

This instruction substitutes some bits in register R_s by the bits in register R_m , as determined by the set bits in register R_n ([Figure 10-48](#)). The R_s register 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 [“Register File Registers” on page 2-6](#).

Status Flags

SZ	Set if result is zero
SN	Equals the MSB of the result

Options

(NF)	No flag (status) update
------	-------------------------

Example

`Rs += MASK Rm BY Rn;;`

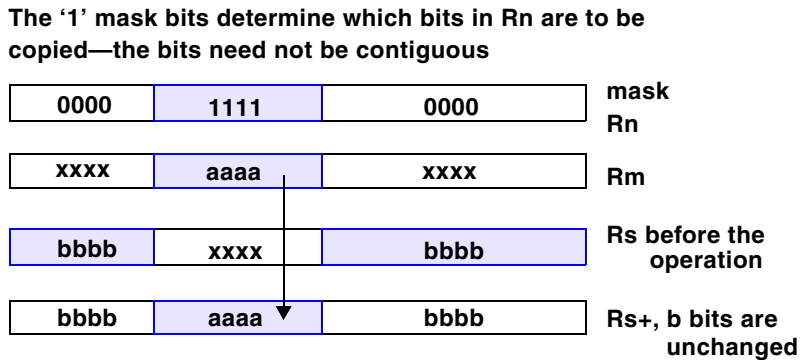


Figure 10-48. Field/Bit Mask Data Flow

```
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 */
```

Shifter Instructions

Get Bits

Syntax

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

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 [“Register File Registers” on page 2-6](#).

Status Flags

SZ	Set if bits in extracted field are all zero
SN	MSB of extracted field

Options

- (SE) Sign extended
- (NF) No flag (status) update

Example

```
R1:0 = GETBITS R7:4 BY R17:16;;
R17 = BFOINC R17:16;;
```

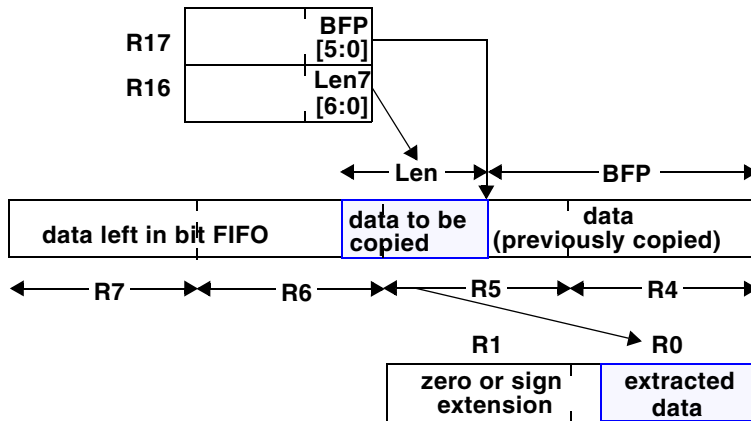


Figure 10-49. Get Bits Data Flow

Shifter Instructions

Put Bits

Syntax

```
{X|Y|XY}Rsd += PUTBITS Rnd BY Rnd {(NF)} ;
```

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 *Len7* field of *Rnd* is ignored. This instruction (in conjunction with shifter instructions *GETBITS* and *BFOTMP*, and *ALU* instruction *BF0INC*) 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 *BFP6* (stored in *Rnd*) to perform the bit insertion; it does not use the length field in *Rnd*. The *PUTBITS* instruction also does not update the *BFP*. Update to the *BFP* should be performed by the *ALU* with instruction *BF0INC*.

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 10-50](#) for a suggested code sequence used to implement a bit *FIFO* insertion.

The *d* suffix denotes operand size—see [“Register File Registers” on page 2-6](#).

Status Flags

SZ Cleared
 SN Cleared

Options

(NF) No flag (status) update

Examples

Figure 10-50 shows the data flow for a PUTBITS instruction, and Listing 10-47 shows the corresponding code for this instruction.

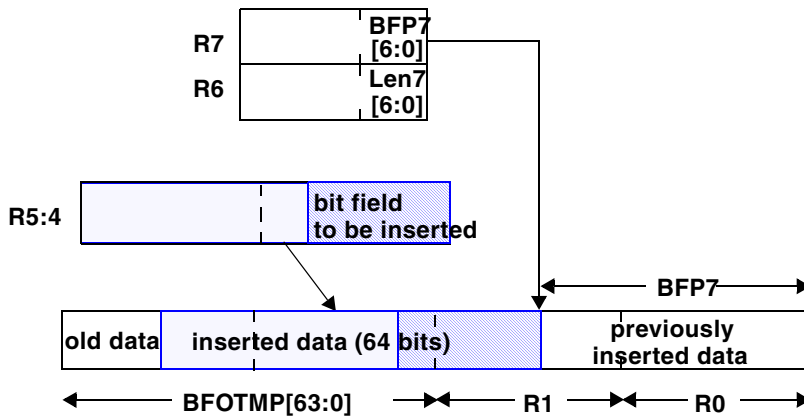


Figure 10-50. Data Flow for Instruction:
 R1:0 += PUTBITS R5:4 BY R7:6;;

Listing 10-47. PUTBITS Example

```
r7 = 0x30;; // insertion begins at bit 48
r6 = 0x20;; // length used to update the bit fifo
```

Shifter Instructions

```
bfotmp = r7:6;; // can have any initial value
r4 = 0xaaaaaaaa;;
r5 = 0x55555555;;
// content of r5:4 contains the field to be inserted

r1:0 = r1:0 - r1:0;;
// initial value of r1:0 equals 0 for better visualization

r1:0 += putbits r5:4 by r7:6;;
// r5:4 inserted into bfotmp and r1:0 from bit 48
// r1 = 0xaaaa0000; r0 = 0x00000000
// bfotmp = 0x00005555 5555aaaa

r7 = bfoinc r7:6;; // bit fifo updated.
/* In this case, r7 becomes 0x50*mod(64)=0x10, so bit 63 is
passed and the AN flag is set */

if xALT; do, 1r3:2 = pass r1:0; r1:0 = bfotmp;;
/* if bit 63 passed, save r1:0 into r3:2 and pass the new content
of bfotmp into r1:0 to prepare bfotmp for the next step */
// r3:2 = 0xaaaa0000 00000000
// r1:0 = 0x00005555 5555aaaa
```

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*. In this instruction, 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> {(NF)} ;  
{X|Y|XY}Rsd = BCLR|BSET|BTGL Rmd BY Rn|<Imm6> {(NF)} ;
```

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 “Register File Registers” on page 2-6.

Status Flags

SZ	Set if result is zero
SN	Equals the MSB of the result

Options

(NF)	No flag (status) update
------	-------------------------

Examples

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 {(NF)} ;

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 result is zero. This result is achieved if the MSB of <i>Rm</i> is 1 with LD0 or if the MSB of <i>Rm</i> is 0 with LD1.
SN	Cleared

Options

(NF)	No flag (status) update
------	-------------------------

Examples

```
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

$$\{X|Y|XY\}Rs = \text{EXP } Rm|Rmd \{(NF)\} ;$$

Function

This instruction extracts the exponent of the operand in register Rm . If the number of leading sign bits in Rm is n , the result in Rs is $-(n-1)$.

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.

Options

(NF)	No flag (status) update
------	-------------------------

Examples

```
R5 = EXP R3;
```

```
If R3 = 0xE4000000
```

```
then R5 = -2
```

```
If R3 = 0x00000101
```

```
then R5 = -22
```

```
R5=exp R3:2;
```

```
If R3=0x140056A3 and R2=0x87654321
```

```
then R5=-2 (in hexadecimal, R5=0xFFFFFFFF)
```

Shifter Instructions

Block Floating-Point

Syntax

{X|Y|XY}BKFP *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 BKFP 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 LSBs as b#001 or b#110, then BF1:0 = b#01.

If at least one of the eight input numbers has the three LSBs as b#011 or b#100, then BF1:0 = b#10.

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

Shifter Instructions

Load/Transfer Bit FIFO Temporary (BFOTMP) Register

Syntax

```
{X|Y|XY}Rsd = BFOTMP {(NF)} ;  
{X|Y|XY}BFOTMP = Rmd {(NF)} ;
```

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. The BFOTMP register is 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

Options

(NF) No flag (status) update

Examples

```
R9:8 = BFOTMP;;
```

```
If BFOTMP = 0x17000016  
then R9:8 = 0x17000016
```

```
BFOTMP = R9:8;;
```

```
If R9:8 = 0x40000005  
then BFOTMP = 0x40000005
```

Load/Transfer Compute Block Status (X/YSTAT) Registers

Syntax

```
{X|Y}STAT = Rm ;
{X|Y}STAT_L = 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.

Examples

```
XSTAT_L = R6;;
```

```
If R6 = 0x910
then XSTAT = 0x910
```

```
If R6 = 0x90010
then XSTAT = 0x10
```

```
XR6 = STAT;;
```

```
If XSTAT = 0x10
then R6 = 0x10
```

Shifter Instructions

Load TR (Trellis), TRH (Trellis History), or CMCTL (Communications Control) Registers (CLU)


Syntax

```
{X|Y|XY}TRs = Rm ; /* TRx load */  
{X|Y|XY}TRsd = Rmd ;  
{X|Y|XY}TRsq = Rmq ;  
  
{X|Y|XY}THRs = Rm ; /* THRx load */  
{X|Y|XY}THRsd = Rmd {(I)} ;  
{X|Y|XY}THRsq = Rmq ;  
  
{X|Y|XY}CMCTL = Rm ; /* CMCTL load */
```

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 from *Rm* or *Rmd* on the EX2 pipeline stage.

 In case of executing the $THR_s = Rm$; or $THR_{sd} = Rmd \{(i)\}$; instructions in parallel to an instruction that shifts the THR register (for example, ACS, DESPREAD, and others), the THR load instruction takes priority on the THR shift in this instruction

Status Flags

None

Shifter Instructions

If the input and output of the instruction is a quad (in/out[127:0]), the instruction executes:

$$\text{Out [127:0]} == \{in[127], in[63], in[126], in[62], in[125], in[61] \dots in[65], in[1], in[64], in[0]\}$$

Examples

```
XYTR8 = R4 ;;
XYTR1:0 = R3:2 ;;
XYTR11:8 = R7:4 ;;

XYTHR3 = R4 ;;
XYTHR1:0 = R3:2 (I) ;;
XYTHR3:0 = R7:4 ;;

XYCMCTL = R4 ;;
```

See Also

[Transfer TR \(Trellis\), THR \(Trellis History\), or CMCTL \(Communications Control\) Registers](#)

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 *integer 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 7-1](#). 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 10-218](#).

The conventions used in these reference pages for representing register names, optional items, and choices are covered in detail in [“IALU” on page 7-1](#). 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.

IALU (Integer) Instructions

- *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 1-23](#) and [“Instruction Parallelism Rules” on page 1-27](#).

Add/Subtract (Integer)

Syntax

$$J_s = J_m + | - J_n | \langle Imm8 \rangle | \langle Imm32 \rangle \{ (\{ CJMP | CB | BR \} \{ NF \}) \} ;$$

$$JB0 | JB1 | JB2 | JB3 | JL0 | JL1 | JL2 | JL3 = J_m + | - J_n | \langle Imm8 \rangle | \langle Imm32 \rangle \{ (NF) \} ;$$

$$K_s = K_m + | - K_n | \langle Imm8 \rangle | \langle Imm32 \rangle \{ (\{ CJMP | CB | BR \} \{ NF \}) \} ;$$

$$KB0 | KB1 | KB2 | KB3 | KLO | KL1 | KL2 | KL3 = K_m + | - K_n | \langle Imm8 \rangle | \langle Imm32 \rangle \{ (NF) \} ;$$

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 7-42). The letter 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

IALU (Integer) Instructions

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
(NF)	No flag (status) update

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);;
```

Add/Subtract With Carry/Borrow (Integer)

Syntax

$$\begin{aligned}
 J_s &= J_m + J_n | \langle Imm8 \rangle | \langle Imm32 \rangle + JC \{ (NF) \} ; \\
 J_s &= J_m - J_n | \langle Imm8 \rangle | \langle Imm32 \rangle + JC - 1 \{ (NF) \} ; \\
 K_s &= K_m + K_n | \langle Imm8 \rangle | \langle Imm32 \rangle + KC \{ (NF) \} ; \\
 K_s &= K_m - K_n | \langle Imm8 \rangle | \langle Imm32 \rangle + KC - 1 \{ (NF) \} ;
 \end{aligned}$$

Function

These instructions add with carry or subtract with borrow the operands in registers J_m/K_m and J_n/K_n in J-IALU with the carry flag from the J/KSTAT register in the J-/K-IALU. 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 7-42](#)). The letter 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

Options

(NF)	No flag (status) update
------	-------------------------

IALU (Integer) Instructions

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;;
```

Add/Subtract With Divide by Two (Integer)

Syntax

$$Js = (Jm + | - Jn | \langle Imm8 \rangle | \langle Imm32 \rangle) / 2 \{ (NF) \} ;$$

$$Ks = (Km + | - Kn | \langle Imm8 \rangle | \langle Imm32 \rangle) / 2 \{ (NF) \} ;$$

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 7-42](#)).

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

Options

(NF)	No flag (status) update
--------	-------------------------

IALU (Integer) Instructions

Examples

```
J4 = (J2 + J8) / 2;;  
K9 = (K2 + 0x2) / 2;;
```

```
J4 = (J2 - J8) / 2;;  
K9 = (K2 - 0x2) / 2;;
```

Compare (Integer)

Syntax

COMP(*Jm*, *Jn* | <*Imm8*> | <*Imm32*>) { (U) } ;

COMP(*Jm*, *Jn* | <*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 7-42). 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

IALU (Integer) Instructions

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
```


Maximum/Minimum (Integer)

Syntax

$$Js = \text{MAX} | \text{MIN} (Jm, Jn | \langle Imm8 \rangle | \langle Imm32 \rangle) \{ (NF) \} ;$$

$$Ks = \text{MAX} | \text{MIN} (Km, Kn | \langle Imm8 \rangle | \langle Imm32 \rangle) \{ (NF) \} ;$$

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 7-42](#)). The letter 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

Options

(NF)	No flag (status) update
------	-------------------------

IALU (Integer) Instructions

Examples

```
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
```

Absolute Value (Integer)

Syntax

$$J_s = \text{ABS } J_m \{(\text{NF})\} ;$$

$$K_s = \text{ABS } K_m \{(\text{NF})\} ;$$

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 . The letter 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 input
JV/KV	Set when input is the most negative number
JC/KC	Cleared

Options

(NF) No flag (status) update

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
```

IALU (Integer) Instructions

If J4 = 0xF000 0000
then J5 = 0x1000 0000 *and* JN *is set*

Logical AND/AND NOT/OR/XOR/NOT (Integer)

Syntax

$$Js = Jm \text{ OR|AND|XOR|AND NOT } Jn | \langle Imm8 \rangle | \langle Imm32 \rangle \{ (NF) \} ;$$

$$Js = \text{NOT } Jm \{ (NF) \} ;$$

$$Ks = Km \text{ OR|AND|XOR|AND NOT } Kn | \langle Imm8 \rangle | \langle Imm32 \rangle \{ (NF) \} ;$$

$$Ks = \text{NOT } Km \{ (NF) \} ;$$

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 7-42](#)). The letter 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

Options

(NF)	No flag (status) update
------	-------------------------

IALU (Integer) Instructions

Examples

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

$$Js = \text{ASHIFTR} | \text{LSHIFTR } Jm \{ (NF) \} ;$$

$$Ks = \text{ASHIFTR} | \text{LSHIFTR } Km \{ (NF) \} ;$$

Function

These instructions perform an arithmetic shift (extends sign on right shift) or logically shift (no sign extension) the operand in register Jm/Km to the right by one bit. The shifted result is placed in register Js/Ks . The shift values are two's complement numbers. The letter 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

Options

(NF) No flag (status) update

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

```

IALU (Integer) Instructions

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

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

$$J_s = \text{ROTR}|\text{ROTL } J_m \{(\text{NF})\} ;$$

$$K_s = \text{ROTR}|\text{ROTL } K_m \{(\text{NF})\} ;$$

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 . The letter 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

Options

(NF)	No flag (status) update
------	-------------------------

Examples

```
J5 = ROTR J3;;
```

```
If J3 = b#1000...0101
```

```
then J5 = b#1100...010
```

```
J5 = ROTL J3;;
```

```
If J3 = b#1000...0101
```

```
then J5 = b#000...01011
```

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 load, store, and transfer (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 7-1](#). The IALUs also provide integer arithmetic support instructions. For instruction reference pages on IALU arithmetic instructions, see [“IALU \(Integer\) Instructions” on page 10-199](#).

The conventions used in these reference pages for representing register names, optional items, and choices are covered in detail in [“IALU” on page 7-1](#). 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 1-23](#) and [“Instruction Parallelism Rules” on page 1-27](#).

IALU (Load/Store/Transfer) Instructions

Load Ureg (Universal) Register (Data Addressing)

Syntax

```
Ureg_s  = {CB|BR}  [Jm +|+= Jn|<Imm8>|<Imm32>] ;  
Ureg_sd = {CB|BR} L [Jm +|+= Jn|<Imm8>|<Imm32>] ;  
Ureg_sq = {CB|BR} Q [Jm +|+= Jn|<Imm8>|<Imm32>] ;
```

```
Ureg_s  = {CB|BR}  [Km +|+= Kn|<Imm8>|<Imm32>] ;  
Ureg_sd = {CB|BR} L [Km +|+= Kn|<Imm8>|<Imm32>] ;  
Ureg_sq = {CB|BR} 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 Load, Store, and Transfer Operations” on page 7-14](#).



There is a five cycle stall following any write to the SQCTL register or debug registers. The debug registers are: WPxCTL, WPxSTAT, WPxL, WPxH, CCNTx, PRFM, PRFCNT, TRCBMASK, TRCBPTR, TRCB31-0, and TRCBVAL.

Status Flags

None affected

Options

CB Circular buffer addressing

BR Bit reversed output

Examples

For examples, see [“IALU Examples”](#) on page 7-43.

IALU (Load/Store/Transfer) Instructions

Store Ureg (Universal) Register (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 Load, Store, and Transfer Operations” on page 7-14.](#)

Status Flags

None affected

Options

None

Examples

For examples, see [“IALU Examples” on page 7-43.](#)

Load Dreg (Data) Register With DAB/SDAB (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} 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} 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 Load, Store, and Transfer Operations”](#) on page 7-14.

Status Flags

None affected

IALU (Load/Store/Transfer) Instructions

Options

()	Linear addressing
CB	Circular buffer addressing
BR	Bit reversed output
DAB	Data alignment buffer access
SDAB	Short data alignment buffer access

Examples

For examples, see [“IALU Examples” on page 7-43](#).

Load Dreg (Data) Register With DAB Offset (Data Addressing)

Syntax

```
{XY}Rsq = XDAB Q [Jm += Jn|<Imm8>|<Imm32>] ;
/* shifts X by one 32-bit word more than address misalignment,
and shift Y by the misalignment */
```

```
{XY}Rsq = XSDAB Q [Jm += Jn|<Imm8>|<Imm32>] ;
/* shifts X by one 16-bit more than address misalignment, and
shift Y by the misalignment */
```

```
{XY}Rsq = YDAB Q [Jm += Jn|<Imm8>|<Imm32>] ;
/* shifts Y by one 32-bit word more than address misalignment,
and shift X by the misalignment */
```

```
{XY}Rsq = YSDAB Q [Jm += Jn|<Imm8>|<Imm32>] ;
/* shifts Y by one 16-bit word more than address misalignment,
and shift X by the misalignment */
```

```
{XY}Rsq = XDAB Q [Km += Kn|<Imm8>|<Imm32>] ;
/* shifts X by one 32-bit word more than address misalignment,
and shift Y by the misalignment */
```

```
{XY}Rsq = XSDAB Q [Km += Kn|<Imm8>|<Imm32>] ;
/* shifts X by one 16-bit more than address misalignment, and
shift Y by the misalignment */
```

```
{XY}Rsq = YDAB Q [Km += Kn|<Imm8>|<Imm32>] ;
/* shifts Y by one 32-bit word more than address misalignment,
and shift X by the misalignment */
```

IALU (Load/Store/Transfer) Instructions

```
{XY}Rsq = YSDAB Q [Km += Kn|<Imm8>|<Imm32>] ;  
/* shifts Y by one 16-bit word more than address misalignment,  
and shift X by the misalignment */  
  
/* 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 Load, Store, and Transfer Operations” on page 7-14](#).

Status Flags

None affected

Options

None

Examples

For examples, see [“IALU Examples” on page 7-43](#).

Store Dreg (Data) Register (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 Load, Store, and Transfer Operations”](#) on page 7-14.

Status Flags

None affected

IALU (Load/Store/Transfer) Instructions

Options

()	Linear addressing
CB	Circular buffer addressing
BR	Bit reversed output

Examples

For examples, see [“IALU Examples” on page 7-43](#).

Transfer Ureg (Universal) Register

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 Single-Instruction, Multiple-Data (SIMD) register transfer inside both compute blocks, use the compute block broadcast in both source and destination *Ureg* groups.

 This instruction is the only instruction that can access groups 63:32. In all other instructions, only register groups 31:0 can be accessed. For information on register groups, see the *ADSP-TS201 TigerSHARC Processor Hardware Reference*.

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 7-42](#).

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.

IALU (Load/Store/Transfer) Instructions

There are special restrictions on using this instruction with the sequencer and debug registers. The following register destination and source combinations are not allowed:

```
Debug_Register = Sequencer_Register ; /* illegal dest<<src */  
Sequencer_Register = Debug_Register ; /* illegal dest<<src */  
Debug_Register = Debug_Register ; /* illegal dest<<src */
```

The sequencer registers are: CJMP, RETI/IB/S, DBGE, LCx, IVSW, FLAGREG/ST/CL, SQCTL/ST/CL, SQSTAT, and SFREG. The debug registers are: WPxCTL, WPxSTAT, WPxL, WPxH, CCNTx, PRFM, PRFCNT, TRCBMASK, TRCBPTR, TRCB31-0, and TRCBVAL.

Also, there are special restrictions on using this instruction with the memory system control registers. For more information, see the CACMDx and CCAIRx register descriptions in [“Memory System Controls and Status” on page 9-19](#).



There is a five cycle stall following any write to the SQCTL register or debug registers.

Exception From IALU Ureg Transfers

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; /* >>> illegal and causes an exception, but */  
R3:0 = R7:4; /* >>> legal */
```

Refer to [“Program Sequencer” on page 1-15](#) and [“IALU Load, Store, and Transfer Operations” on page 7-14](#) for an explanation of how interrupts are introduced with transfer exceptions.

Examples

```
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 *program flow control 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 8-1](#).

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 2-6](#). 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 15- 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 15-bit address (if possible) 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 8-19](#).



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 1-23](#) and [“Instruction Parallelism Rules” on page 1-27](#).

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 15- or 32-bit address). A `CALL` instruction transfers execution to address *Label* (or to an immediate 15- 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 8-19](#).

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.

Examples

```
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 */
```

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 8-19](#).

 If the prediction is true (option `NP` is not set), the call must be absolute (option `ABS` must be set).

Options

<code>NP</code>	Branch prediction is <code>Not Predicted</code>
<code>ABS</code>	Target address is in the immediate field

See section [“Branch Target Buffer \(BTB\)” on page 8-42](#) for more on `NP` and `ABS` options.

Examples

```
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 $\overline{IRQ3-0}$ pins and internal process conditions) and software interrupts (program sets an interrupt's latch bit). Figure 10-53 provides a comparison of interrupt service variations using the RETI and RTI instructions.

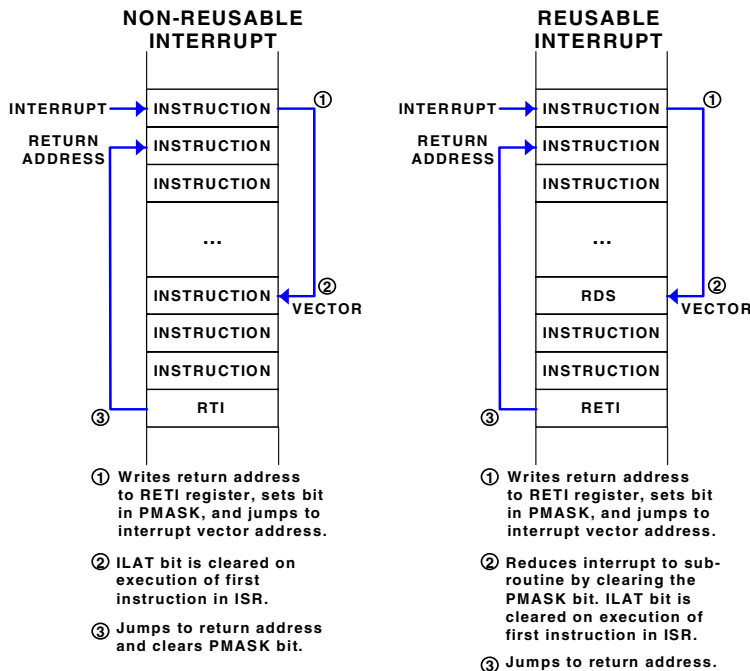


Figure 10-53. Non-Reusable Versus Reusable Interrupt Service

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 10-53 on page 10-238](#) 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 8-26](#).

Options

None

Examples

```
<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 standalone condition. For more information, see [“Conditional Execution” on page 8-14](#).



The NOP, IDLE, BTBINV, BTBEN, BTBDIS, BTBLOCK, BTBELOCK, TRAP, and EMUTRAP instructions may not be conditional. The following instruction line for example is not legal:

```
if aeq; do idle;; /* ILLEGAL! */
```

Examples

```
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 standalone condition. For more information, see [“Conditional Execution” on page 8-14](#).



The NOP, IDLE, BTBINV, BTBEN, BTBDIS, BTBLOCK, BTBELOCK, TRAP, and EMUTRAP instructions may not be conditional. The following instruction line for example is not legal:

```
if aeq; do idle;; /* ILLEGAL! */
```

Examples

```
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.

Load Condition Into Static Condition Flag

Syntax

```
{X|Y|XY}SF1|SF0 = Compute_Cond. ;
{X|Y|XY}SF1|SF0 += AND|OR|XOR Compute_Cond. ;
ISF1|ISF0 = IALU_Cond.|Compute_Cond.|Seq_Cond. ;
ISF1|ISF0 += AND|OR|XOR IALU_Cond.|Compute_Cond.|Seq_Cond. ;
```

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. For more information on the conditions, see [“Sequencer Instruction Summary” on page 8-97](#).

Examples

```
SF0 = XMLT ;; /* Static flag xSF0 is set to XMLT. */
SF1 += OR XAEQ ;; /* Static flag xSF1 ORed with xAEQ. */

ISF0 = JLT ;; /* Static flag ISF0 is set to JLT. */
ISF1 += OR KEQ ;; /* Static flag ISF1 ORed with KEQ. */
```

Sequencer Instructions

Idle

Syntax

```
IDLE ;
```

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.

The `NOP`, `IDLE`, `BTBINV`, `BTBEN`, `BTBDIS`, `BTBLOCK`, `BTBELOCK`, `TRAP`, and `EMUTRAP` instructions may not be conditional. The following instruction line for example is not legal:

```
if aeq; do idle;; /* ILLEGAL! */
```

BTB Enable/Disable

Syntax

```
BTBEN ;
BTBDIS ;
```

Function

These instruction enable or disable the BTB. When `BTBEN` executes, the BTB is enabled and starts full operation, which includes loading jumps with prediction taken and searching for fetch hits to predict program flow changes.

The `NOP`, `IDLE`, `BTBINV`, `BTBEN`, `BTBDIS`, `BTBLOCK`, `BTBELOCK`, `TRAP`, and `EMUTRAP` instructions may not be conditional. The following instruction line for example is not legal:

```
if aeq; do idle;; /* ILLEGAL! */
```



There is an ten cycle latency in the instruction pipeline following the `BTBEN`, `BTBDIS`, `BTBLOCK`, `BTBELOCK`, and `BTBINV` instructions. In pipeline stage PD, these instructions cause a flush of the pipeline and the fetch is resumed when the BTB instruction reaches pipe stage EX2.

Sequencer Instructions

BTB Lock/End Lock

Syntax

```
BTBLOCK ;  
BTBELOCK ;
```

Function

These instructions lock new BTB entries (cannot be replaced based on least recently used (LRU) field or end the lock on new BTB entries).

The `NOP`, `IDLE`, `BTBINV`, `BTBEN`, `BTBDIS`, `BTBLOCK`, `BTBELOCK`, `TRAP`, and `EMUTRAP` instructions may not be conditional. The following instruction line for example is not legal:

```
if aeq; do idle;; /* ILLEGAL! */
```



There is an ten cycle latency in the instruction pipeline following the `BTBEN`, `BTBDIS`, `BTBLOCK`, `BTBELOCK`, and `BTBINV` instructions. In pipeline stage PD, these instructions cause a flush of the pipeline and the fetch is resumed when the BTB instruction reaches pipe stage EX2.

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`, `BTBINV`, `BTBEN`, `BTBDIS`, `BTBLOCK`, `BTBELOCK`, `TRAP`, and `EMUTRAP` instructions may not be conditional. The following instruction line for example is not legal:

```
if aeq; do idle;; /* ILLEGAL! */
```



There is an ten cycle latency in the instruction pipeline following the `BTBEN`, `BTBDIS`, `BTBLOCK`, `BTBELOCK`, and `BTBINV` instructions. In pipeline stage PD, these instructions cause a flush of the pipeline and the fetch is resumed when the BTB instruction reaches pipe stage EX2.

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-TS201 TigerSHARC Processor Hardware Reference*.

The NOP, IDLE, BTBINV, BTBEN, BTBDIS, BTBLOCK, BTBELOCK, TRAP, and EMUTRAP instructions may not be conditional. The following instruction line for example is not legal:

```
if aeq; do idle;; /* 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-TS201 TigerSHARC Processor Hardware Reference*.

The NOP, IDLE, BTBINV, BTBEN, BTBDIS, BTBLOCK, BTBELOCK, TRAP, and EMUTRAP instructions may not be conditional. The following instruction line for example is not legal:

```
if aeq; do idle;; /* ILLEGAL! */
```

Sequencer Instructions

No Operation

Syntax

```
NOP ;
```

Function

No operation – holds an instruction slot.

The `NOP`, `IDLE`, `BTBINV`, `BTBEN`, `BTBDIS`, `BTBLOCK`, `BTBELOCK`, `TRAP`, and `EMUTRAP` instructions may not be conditional. The following instruction line for example is not legal:

```
if aeq; do idle;; /* ILLEGAL! */
```



The `NOP` instruction can be executed with other sequencer instructions in the same instruction line.

A QUICK REFERENCE

This appendix 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 appendix 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:

- [“Processor Architecture” on page 1-7](#)
- [“Instruction Line Syntax and Structure” on page 1-23](#)
- [“Instruction Parallelism Rules” on page 1-27](#)
- [“Register File Registers” on page 2-6](#)
- [“ALU Operations” on page 3-4](#)
- [“CLU Operations” on page 4-4](#)
- [“Multiplier Operations” on page 5-5](#)
- [“Shifter Operations” on page 6-4](#)
- [“IALU Operations” on page 7-6](#)
- [“Sequencer Operations” on page 8-8](#)
- [“Memory Block Accesses” on page 9-16](#)

ALU Quick Reference

For examples using these instructions, see “ALU Examples” on page 3-16.

Listing A-1. ALU Fixed-Point Instructions

```

{X|Y|XY}{S|B}Rs = Rm +|- Rn {{(S|SU){NF}}};1
{X|Y|XY}{L|S|B}Rsd = Rmd +|- Rnd {{(S|SU){NF}}};1
{X|Y|XY}Rs = Rm + CI {-1};
{X|Y|XY}LRsd = Rmd + CI {-1};
{X|Y|XY}Rs = Rm + Rn + CI {{(S|SU){NF}}};1
{X|Y|XY}Rs = Rm - Rn + CI -1 {{(S|SU){NF}}};1
{X|Y|XY}LRsd = Rmd + Rnd + CI {{(S|SU){NF}}};1
{X|Y|XY}LRsd = Rmd - Rnd + CI -1 {{(S|SU){NF}}};1
{X|Y|XY}{S|B}Rs = (Rm +|- Rn)/2 {{(T|U){NF}}};2
{X|Y|XY}{L|S|B}Rsd = (Rmd +|- Rnd)/2 {{(T|U){NF}}};2
{X|Y|XY}{S|B}Rs = ABS Rm {{NF}};
{X|Y|XY}{L|S|B}Rsd = ABS Rmd {{NF}};
{X|Y|XY}{S|B}Rs = ABS (Rm + Rn) {{(X){NF}}};3
{X|Y|XY}{L|S|B}Rsd = ABS (Rmd + Rnd) {{(X){NF}}};3
{X|Y|XY}{S|B}Rs = ABS (Rm - Rn) {{(X|U){NF}}};4
{X|Y|XY}{L|S|B}Rsd = ABS (Rmd - Rnd) {{(X|U){NF}}};4
{X|Y|XY}{S|B}Rs = - Rm {{NF}};
{X|Y|XY}{L|S|B}Rsd = - Rmd {{NF}};
{X|Y|XY}{S|B}Rs = MAX|MIN (Rm, Rn) {{(U){Z}{NF}}};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: (): signed inputs, saturation, (X): extend for ABS uses unsigned saturation

⁴ Options include: (): signed inputs, saturation, (X): extend for ABS uses unsigned saturation, (U): unsigned

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}{L|S|B}Rsd = MAX|MIN (Rmd, Rnd) {{(U){Z}{NF}}};5
{X|Y|XY}S|BRsd = VMAX|VMIN (Rmd, Rnd) {(NF)};
{X|Y|XY}{S|B}Rs = INC|DEC Rm {{(S|SU){NF}}};1
{X|Y|XY}{L|S|B}Rsd = INC|DEC Rmd {{(S|SU){NF}}};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 {(NF)};
{X|Y|XY}{L|S|B}Rsd = CLIP Rmd BY Rnd {(NF)};
{X|Y|XY}Rs = SUM SRm|BRm {{(U){NF}}};1
{X|Y|XY}Rs = SUM SRmd|BRmd {{(U){NF}}};1
{X|Y|XY}Rs = ONES Rm|Rmd {(NF)};
{X|Y|XY}PR1:0 = Rmd {(NF)};
{X|Y|XY}Rsd = PR1:0 {(NF)};
{X|Y|XY}Rs = BFOINC Rmd {(NF)};
{X|Y|XY}PRO|PR1 += ABS (SRmd - SRnd){{(U){NF}}};1
{X|Y|XY}PRO|PR1 += ABS (BRmd - BRnd){{(U){NF}}};1
{X|Y|XY}PRO|PR1 += SUM SRm {{(U){NF}}};1
{X|Y|XY}PRO|PR1 += SUM SRmd {{(U){NF}}};1
{X|Y|XY}PRO|PR1 += SUM BRm {{(U){NF}}};1
{X|Y|XY}PRO|PR1 += SUM BRmd {{(U){NF}}};1
{X|Y|XY}{S|B}Rs = Rm + Rn, Ra = Rm - Rn {(NF)}; (dual op.)
{X|Y|XY}{L|S|B}Rsd = Rmd + Rnd, Rad = Rmd - Rnd {(NF)}; (dual op.)

```

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 {(NF)};
{X|Y|XY}LRsd = Rmd AND|AND NOT|OR|XOR Rnd {(NF)};
{X|Y|XY}Rs = NOT Rm {(NF)};
{X|Y|XY}LRsd = NOT Rmd {(NF)};

```

¹ Options include: (): signed, (U): unsigned

ALU Quick Reference

Listing A-3. ALU Fixed-Point Miscellaneous Instructions

```
{X|Y|XY}Rsd = EXPAND SRm {+|- SRn} {{(I|IU){NF}}} ;1
{X|Y|XY}Rsq = EXPAND SRmd {+|- SRnd} {{(I|IU){NF}}} ;1
{X|Y|XY}SRsd = EXPAND BRm {+|- BRn} {{(I|IU){NF}}} ;1
{X|Y|XY}SRsq = EXPAND BRmd {+|- BRnd} {{(I|IU){NF}}} ;1
{X|Y|XY}SRs = COMPACT Rmd {+|- Rnd} {{(T|I|IS|ISU){NF}}} ;2
{X|Y|XY}BRs = COMPACT SRmd {+|- SRnd} {{(T|I|IS|ISU){NF}}} ;2
{X|Y|XY}Rs = COMPACT LRmd {{(U|IS|ISU){NF}}} ;
{X|Y|XY}LRsd = COMPACT QRmq {{(U|IS|ISU){NF}}} ;
{X|Y|XY}BRsd = MERGE Rm, Rn {(NF)} ;
{X|Y|XY}BRsq = MERGE Rmd, Rnd {(NF)} ;
{X|Y|XY}SRsd = MERGE Rm, Rn {(NF)} ;
{X|Y|XY}SRsq = MERGE Rmd, Rnd {(NF)} ;
{X|Y|XY}Rsd = PERMUTE (Rmd, Rn) {(NF)} ;
{X|Y|XY}Rsq = PERMUTE (Rmd, -Rmd, Rn) {(NF)} ;
```

Listing A-4. Floating-Point ALU Instructions

```
{X|Y|XY}FRs = Rm +|- Rn {{(T){NF}}} ;3
{X|Y|XY}FRsd = Rmd +|- Rnd {{(T){NF}}} ;3
{X|Y|XY}FRs = (Rm +|- Rn)/2 {{(T){NF}}} ;3
{X|Y|XY}FRsd = (Rmd +|- Rnd)/2 {{(T){NF}}} ;3
{X|Y|XY}FRs = MAX|MIN (Rm, Rn) {(NF)} ;4
{X|Y|XY}FRsd = MAX|MIN (Rmd, Rnd) {(NF)} ;4
{X|Y|XY}FRs = ABS Rm {(NF)} ;
{X|Y|XY}FRsd = ABS Rmd {(NF)} ;
{X|Y|XY}FRs = ABS (Rm +|- Rn) {{(T){NF}}} ;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

⁴ Options include: (): round, (T): truncate (MIN only)

```

{X|Y|XY}FRsd = ABS (Rmd +|- Rnd) {{(T){NF}}};3
{X|Y|XY}FRs = - Rm {(NF)} ;
{X|Y|XY}FRsd = - Rmd {(NF)} ;
{X|Y|XY}FCOMP (Rm, Rn) ;
{X|Y|XY}FCOMP (Rmd, Rnd) ;
{X|Y|XY}Rs = FIX FRm|FRmd {BY Rn} {{(T){NF}}};3
{X|Y|XY}FRs|FRsd = FLOAT Rm {BY Rn} {{(T){NF}}};3
{X|Y|XY}FRsd = EXT D Rm {(NF)} ;
{X|Y|XY}FRs = SNGL Rmd {{(T){NF}}};
{X|Y|XY}FRs = CLIP Rm BY Rn {(NF)} ;
{X|Y|XY}FRsd = CLIP Rmd BY Rnd {(NF)} ;
{X|Y|XY}FRs = Rm COPYSIGN Rn {(NF)} ;
{X|Y|XY}FRsd = Rmd COPYSIGN Rnd {(NF)} ;
{X|Y|XY}FRs = SCALB FRm BY Rn {(NF)} ;
{X|Y|XY}FRsd = SCALB FRmd BY Rn {(NF)} ;
{X|Y|XY}FRs = PASS Rm ;
{X|Y|XY}FRsd = PASS Rmd ;
{X|Y|XY}FRs = RECIPS Rm {(NF)} ;
{X|Y|XY}FRsd = RECIPS Rmd {(NF)} ;
{X|Y|XY}FRs = RSQRTS Rm {(NF)} ;
{X|Y|XY}FRsd = RSQRTS Rmd {(NF)} ;
{X|Y|XY}Rs = MANT FRm|FRmd {(NF)} ;
{X|Y|XY}Rs = LOGB FRm|FRmd {{(S){NF}}};1
{X|Y|XY}FRs = Rm + Rn, FRa = Rm - Rn {(NF)} ; (dual op.)
{X|Y|XY}FRsd = Rmd + Rnd, FRad = Rmd - Rnd {(NF)} ; (dual op.)

```

¹ Options include: (): do not saturate, (S): saturate

CLU Quick Reference

For examples using these instructions, see [“CLU Examples” on page 4-43](#).

Listing A-5. CLU Instructions

```
{X|Y|XY}{S}TRsd = TMAX(TRmd + Rmq_h, TRnd + Rmq_l) {(NF)} ;
{X|Y|XY}{S}TRsd = TMAX(TRmd - Rmq_h, TRnd - Rmq_l) {(NF)} ;
{X|Y|XY}{S}Rs = TMAX(TRm, TRn) {(NF)} ;
{X|Y|XY}{S}TRsd = MAX(TRmd + Rmq_h, TRnd + Rmq_l) {(NF)} ;
{X|Y|XY}{S}TRsd = MAX(TRmd - Rmq_h, TRnd - Rmq_l) {(NF)} ;
/* where Rmq_h must be the upper half of a quad register and
Rmq_l must be the lower half of the SAME quad register */

{X|Y|XY}Rs = TRm ;
{X|Y|XY}Rsd = TRmd ;
{X|Y|XY}Rsq = TRmq ;
{X|Y|XY}Rs = THRm ;
{X|Y|XY}Rsd = THRmd ;
{X|Y|XY}Rsq = THRmq ;
{X|Y|XY}Rs = CMCTL ;
/* For TR, THR, and CMCTL register load syntax, see “Shifter Quick Refer-
ence” on page A-9. */

{X|Y|XY}TRs += DESPREAD (Rmq, THRd) ;
{X|Y|XY}Rs = TRs, TRs = DESPREAD (Rmq, THRd) {(NF)} ; (dual op.)
{X|Y|XY}Rsd = TRsd, TRsd = DESPREAD (Rmq, THRd) {(NF)} ; (dual op.)

{X|Y|XY}TRsa = XCORRS (Rmq, THRnq) {(CUT
<Imm>|R)}{(CLR)}{(EXT)}{(NF)} ;
{X|Y|XY}Rsq = TRbq, TRsa = XCORRS (Rmq, THRnq)
{(CUT <Imm>|R)}{(CLR)}{(EXT)}{(NF)} ; (dual op.)
/* where TRsa = TR15:0 or TR31:16 */
```



```
{X|Y|XY}{S}TRsq = ACS (TRmd, TRnd, Rm) {(TMAX)} {(NF)} ;
{X|Y|XY}Rsq = TRaq, {S}TRsq = ACS (TRmd, TRnd, Rm)
{(TMAX)}{(NF)} ; (dual op.)
```

/ For PERMUTE instruction syntax, see “ALU Quick Reference” on page A-2, Listing A-3. */*

Multiplier Quick Reference

For examples using these instructions, see “Multiplier Examples” on page 5-24.

Listing A-6. 32-Bit Fixed-Point Multiplication Instructions

```
{X|Y|XY}Rs = Rm * Rn {{{U|nU}{I|T}{S}{NF}}} ;1
{X|Y|XY}Rsd = Rm * Rn {{{U|nU}{I}{NF}}} ;
{X|Y|XY}MRa += Rm * Rn {{{U}{I}{C|CR}{NF}}} ;2
{X|Y|XY}MRa -= Rm * Rn {{{I}{C|CR}{NF}}} ;
{X|Y|XY}Rs = MRa, MRa += Rm * Rn {{{U}{I}{C|CR}{NF}}} ; (dual op.)
{X|Y|XY}Rsd = MRa, MRa += Rm * Rn {{{U}{I}{C}{NF}}} ; (dual op.)
/* 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}{NF}}} ;
{X|Y|XY}Rsq = Rmd * Rnd {{{U}{I}{NF}}} ;
{X|Y|XY}MR3:0 += Rmd * Rnd {{{U}{I}{C|CR}{NF}}} ;
```

¹ Options include: () : fractional, signed, and 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

Multiplier Quick Reference

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

Listing A-8. 32-Bit Fixed-Point Complex Multiplication Instructions

```
{X|Y|XY}MRa += Rm ** Rn {{{I}{C}{CR}{J}{NF}}} ;  
{X|Y|XY}MRa -= Rm ** Rn {{{I}{C}{CR}{J}{NF}}} ;  
{X|Y|XY}Rs = MRa, MRa += Rm ** Rn {{{I}{C}{CR}{J}{NF}}} ; (dual op.)  
{X|Y|XY}Rsd = MRa, MRa += Rm ** Rn {{{I}{C}{J}{NF}}} ; (dual op.)  
/* 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}{NF}}} ;  
{X|Y|XY}FRsd = Rmd * Rnd {{{T}{NF}}} ;
```

Listing A-10. Multiplier Register Load Instructions

```
{X|Y|XY}{S|L}MRa = Rmd {{{SE|ZE}{NF}}} ;  
{X|Y|XY}MR4 = Rm {(NF)} ;  
{X|Y|XY}{S}Rsd = MRa {{{U}{S}{NF}}} ;  
{X|Y|XY}Rs = MR3:0 {{{U}{S}{NF}}} ;  
{X|Y|XY}Rs = MR4 ;  
/* where MRa is either MR1:0 or MR3:2 */  
  
{X|Y|XY}Rsd = SMRb {{{U}{NF}}} ; /* extract 2 short words */  
{X|Y|XY}LRsd = MRb {{{U}{NF}}} ; /* extract 1 normal word */  
/* where MRb is either MR0, MR1, MR2, or MR3 */  
  
{X|Y|XY}Rs = SMRa {{{U}{NF}}} ; /* extract 4 short words */  
{X|Y|XY}LRs = MRa {{{U}{NF}}} ; /* extract 2 normal words */
```

```

{X|Y|XY}QRsq = LMRa {{(U){NF}}}; /* extract 1 long word from MRa
*/
/* where MRa is either MR1:0 or MR3:2 */

{X|Y|XY}Rs = COMPACT MRa {{(U){I}{S}{NF}}};
{X|Y|XY}SRsd = COMPACT MR3:0 {{(U){I}{S}{NF}}};
/* where MRa is either MR1:0 or MR3:2 */

```

Shifter Quick Reference

For examples using these instructions, see [“Shifter Examples” on page 6-20](#).

Listing A-11. Shifter Instructions

```

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

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

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

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

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

```

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 6-5 on page 6-9](#).

Shifter Quick Reference

```
{X|Y|XY}Rsd = GETBITS Rmq BY Rnd {{{SE}{NF}}} ;
{X|Y|XY}Rsd += PUTBITS Rmd BY Rnd {(NF)} ;
{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> {(NF)} ;
{X|Y|XY}Rsd = BCLR|BSET|BTGL Rmd BY Rn<Imm6> {(NF)} ;
{X|Y|XY}Rs = LDO|LD1 Rm|Rmd {(NF)} ;
{X|Y|XY}Rs = EXP Rm|Rmd {(NF)} ;
{X|Y}STAT = Rm ;
{X|Y}STAT_L = Rm ;
{X|Y}Rs = {X|Y}STAT ;
{X|Y|XY}BKFPTRmd, Rnd ;
{X|Y|XY}Rsd = BFOTMP {(NF)} ;
{X|Y|XY}BFOTMP = Rmd {(NF)} ;
{X|Y|XY}TRs = Rm ;
{X|Y|XY}TRsd = Rmd ;
{X|Y|XY}TRsq = Rmq ;
{X|Y|XY}THRs = Rm ;
{X|Y|XY}THRsd = Rmd {(I)} ;
{X|Y|XY}THRsq = Rmq ;
{X|Y|XY}CMCTL = Rm ;
/* For TR, THR, and CMCTL register transfer syntax, see "CLU Quick Reference" on page A-6. */
```

IALU Quick Reference

For examples using these instructions, see [“IALU Examples” on page 7-43](#).

Listing A-12. IALU Arithmetic, Logical, and Function Instructions

```

Js = Jm +|- Jn|<Imm8>|<Imm32> {({CJMP|CB|BR}{NF})} ;
JB0|JB1|JB2|JB3|JL0|JL1|JL2|JL3 = Jm +|- Jn|<Imm8>|<Imm32> {(NF)} ;
Js = Jm + Jn|<Imm8>|<Imm32> + JC {(NF)} ;
Js = Jm - Jn|<Imm8>|<Imm32> + JC - 1 {(NF)} ;
Js = (Jm +|- Jn|<Imm8>|<Imm32>)/2 {(NF)} ;
COMP(Jm, Jn|<Imm8>|<Imm32>) {(U)} ;
Js = MAX|MIN (Jm, Jn|<Imm8>|<Imm32>) {(NF)} ;
Js = ABS Jm {(NF)} ;
Js = Jm OR|AND|XOR|AND NOT Jn|<Imm8>|<Imm32> {(NF)} ;
Js = NOT Jm {(NF)} ;
Js = ASHIFTR|LSHIFTR Jm {(NF)} ;
Js = ROTR|ROTL Jm {(NF)} ;

Ks = Km +|- Kn|<Imm8>|<Imm32> {({CJMP|CB|BR}{NF})} ;
KB0|KB1|KB2|KB3|KLO|KL1|KL2|KL3 = Km +|- Kn|<Imm8>|<Imm32> {(NF)} ;
Ks = Km + Kn|<Imm8>|<Imm32> + KC {(NF)} ;
Ks = Km - Kn|<Imm8>|<Imm32> + KC - 1 {(NF)} ;
Ks = (Km +|- Kn|<Imm8>|<Imm32>)/2 {(NF)} ;
COMP(Jm, Jn|<Imm8>|<Imm32>) {(U)} ;
Ks = MAX|MIN (Km, Kn|<Imm8>|<Imm32>) {(NF)} ;
Ks = ABS Km {(NF)} ;
Ks = Km OR|AND|XOR|AND NOT Kn|<Imm8>|<Imm32> {(NF)} ;
Ks = NOT Km {(NF)} ;
Ks = ASHIFTR|LSHIFTR Km {(NF)} ;
Ks = ROTR|ROTL Km {(NF)} ;

```

IALU Quick Reference

Listing A-13. IALU Ureg Register Load (Data Addressing) Instructions

```
Ureg_s = {CB|BR} [Jm +|+= Jn|<Imm8>|<Imm32>] ;
Ureg_sd = {CB|BR} L [Jm +|+= Jn|<Imm8>|<Imm32>] ;
Ureg_sq = {CB|BR} Q [Jm +|+= Jn|<Imm8>|<Imm32>] ;

Ureg_s = {CB|BR} [Km +|+= Kn|<Imm8>|<Imm32>] ;
Ureg_sd = {CB|BR} L [Km +|+= Kn|<Imm8>|<Imm32>] ;
Ureg_sq = {CB|BR} 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} 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} 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 */

{XY}Rsq = XDAB Q [Jm += Jn|<Imm8>|<Imm32>] ;
{XY}Rsq = XSDAB Q [Jm += Jn|<Imm8>|<Imm32>] ;
```

```

{XY}Rsq = YDAB Q [Jm += Jn|<Imm8>|<Imm32>] ;
{XY}Rsq = YSDAB Q [Jm += Jn|<Imm8>|<Imm32>] ;

{XY}Rsq = XDAB Q [Km += Kn|<Imm8>|<Imm32>] ;
{XY}Rsq = XSDAB Q [Km += Kn|<Imm8>|<Imm32>] ;
{XY}Rsq = YDAB Q [Km += Kn|<Imm8>|<Imm32>] ;
{XY}Rsq = YSDAB Q [Km += Kn|<Imm8>|<Imm32>] ;

```

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 */

```

Sequencer Quick Reference

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 8-92](#).

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. */1  
  
IF Condition, JUMP|CALL|CJMP|CJMP_CALL ;
```

¹ On a instruction lines beginning with a sequencer instruction (such as a conditional instruction, a conditional sequencer instruction, or an unconditional sequencer instruction), instructions may use the NF (no flag update) option only if ALL instruction slots on that line use the NF option. Instructions that do not support the NF option (such as COMP) may be used.


```

    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. */1

```

```
{X|Y|XY}SF1|SF0 = Compute_Cond. ;
```

```
{X|Y|XY}SF1|SF0 += AND|OR|XOR Compute_Cond. ;
```

```
ISF1|ISF0 = IALU_Cond.|Compute_Cond.|Seq_Cond. ;
```

```
ISF1|ISF0 += AND|OR|XOR IALU_Cond.|Compute_Cond.|Seq_Cond. ;
```

```
IDLE ;
```

```
BTBEN ;
```

```
BTBDIS ;
```

```
BTBLOCK ;
```

```
BTBELOCK ;
```

```
BTBINV ;
```

```
TRAP (<Imm5>) ;;
```

```
EMUTRAP ;;
```

```
NOP ;
```

¹ On a instruction lines beginning with a sequencer instruction (such as a conditional instruction, a conditional sequencer instruction, or an unconditional sequencer instruction), instructions may use the NF (no flag update) option only if ALL instruction slots on that line use the NF option. Instructions that do not support the NF option (such as COMP) may be used.

Memory/Cache Quick Reference

For examples using these instructions, see [“Memory Access Examples” on page 9-53](#).

Listing A-19. Memory System Commands

```
/* These commands use values from the DEFTS201.H file. In all of
these commands, the Js is any IALU data register (J30-0 or K30-
0), and the x in CACMDx or CCAIRx is the block number: 0, 2, 4, 6,
8, 10, or B (for broadcast) */
```

```
Js = (CACMD_REFRESH | Refresh_Rate) ;
```

```
CACMDx = Js ;
```

```
/* where Refresh_Rate is an immediate 13-bit value */
```

```
Js = CACMD_EN ;
```

```
CACMDx = Js ;
```

```
Js = CACMD_DIS ;
```

```
CACMDx = Js ;
```

```
Js = (CACMD_SET_BUS | I_Caching | S_Caching | J_Caching | K_Caching) ;
```

```
CACMDx = Js ;
```

```
/* where _Caching selects caching for a bus's transactions as
none (no transactions cache), read-only, write-only, or
read-write (all transactions cached) */
```

```
Js = CACMD_SLOCK ;
```

```
CACMDx = Js ;
```

```
Js = CACMD_ELOCK ;
```

```
CACMDx = Js ;
```

```
Js = Start_Address ;
```

```
CCAIRx = Js ;
```

```

Js = (CACHE_INIT | (Length << CACMD_LEN_P) {| CACMD_NOSTALL});
CACMDx = Js;
Js = (CACHE_INIT_LOCK | (Length << CACMD_LEN_P) {|
CACMD_NOSTALL});
CACMDx = Js;
/* where Start_Address is a 14-bit address, using bits 16-3 (bits
2-0 are ignored), Length is a value up to the size of the cache,
and Stall_Mode is CACMD_NOSTALL (for STALL) or omitted (STALL) */

Js = Start_Index ;
CCAIRx = Js ;
Js = (CACMD_CB | (Length << CACMD_LEN_P) {| CACMD_NOSTALL} ) ;
CACMDx = Js ;
/* where Start_Index is a 7-bit index, using bits 16-10 (bits 9-0
are ignored), Length is a value ranging from 0 to 511, and
Stall_Mode is CACMD_NOSTALL (no STALL) or omitted (STALL) */

Js = Start_Index ;
CCAIRx = Js ;
Js = (CACMD_INV | (Length << CACHE_LEN_D) {| CACMD_NOSTALL}) ;
CACMDx = Js ;
/* where Start_Index is a 7-bit index, using bits 16-10 (bits 9-0
are ignored), Length is a value ranging from 0xFF8000 to 0x8000,
Stall_Mode is CACMD_NOSTALL (no STALL) or omitted (STALL) */

/* If the CACMDx value and CCAIRx value are written into a double
register (for example, CACMDx in J0 and CCAIRx in J1), a double
register transfer loads both values at once. As in:
Jm = CACMDx_value ; /* low register */
Jn = CCAIRx_value ; /* high register */
CACMDx = Jmd ;      /* long word transfer */

```

Memory/Cache Quick Reference

B REGISTER/BIT DEFINITIONS

When writing DSP programs, it is often necessary to set, clear, or test bits in the processor'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-TS201 TigerSHARC processor. For more information on ADSP-TS201 processor registers, see the "Registers and Memory" chapter in the *ADSP-TS201 TigerSHARC Processor Hardware Reference*.

Listing B-1. DEFTS201.H – Register and Bit #Defines File

```
/*  
/* defTS201.h  
/*  
/* Version 2.1 7/13/04  
/*  
/* Copyright (c) 2002 Analog Devices, Inc., All rights reserved  
/*  
  
#if !defined(__DEFTS201_H_)  
#define __DEFTS201_H_  
  
//  
// Bit Mask 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_AVS_P (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 (0x1E0000)
#define XR1_LOC (0x1E0001)
#define XR2_LOC (0x1E0002)
#define XR3_LOC (0x1E0003)
#define XR4_LOC (0x1E0004)
#define XR5_LOC (0x1E0005)
#define XR6_LOC (0x1E0006)
#define XR7_LOC (0x1E0007)
#define XR8_LOC (0x1E0008)
```

```
#define XR9_LOC (0x1E0009)
#define XR10_LOC (0x1E000A)
#define XR11_LOC (0x1E000B)
#define XR12_LOC (0x1E000C)
#define XR13_LOC (0x1E000D)
#define XR14_LOC (0x1E000E)
#define XR15_LOC (0x1E000F)
#define XR16_LOC (0x1E0010)
#define XR17_LOC (0x1E0011)
#define XR18_LOC (0x1E0012)
#define XR19_LOC (0x1E0013)
#define XR20_LOC (0x1E0014)
#define XR21_LOC (0x1E0015)
#define XR22_LOC (0x1E0016)
#define XR23_LOC (0x1E0017)
#define XR24_LOC (0x1E0018)
#define XR25_LOC (0x1E0019)
#define XR26_LOC (0x1E001A)
#define XR27_LOC (0x1E001B)
#define XR28_LOC (0x1E001C)
#define XR29_LOC (0x1E001D)
#define XR30_LOC (0x1E001E)
#define XR31_LOC (0x1E001F)
```

```
/* Y Comp Block */
```

```
#define YR0_LOC (0x1E0040)
#define YR1_LOC (0x1E0041)
#define YR2_LOC (0x1E0042)
#define YR3_LOC (0x1E0043)
#define YR4_LOC (0x1E0044)
#define YR5_LOC (0x1E0045)
#define YR6_LOC (0x1E0046)
#define YR7_LOC (0x1E0047)
```

```
#define YR8_LOC (0x1E0048)
#define YR9_LOC (0x1E0049)
#define YR10_LOC (0x1E004A)
#define YR11_LOC (0x1E004B)
#define YR12_LOC (0x1E004C)
#define YR13_LOC (0x1E004D)
#define YR14_LOC (0x1E004E)
#define YR15_LOC (0x1E004F)
#define YR16_LOC (0x1E0050)
#define YR17_LOC (0x1E0051)
#define YR18_LOC (0x1E0052)
#define YR19_LOC (0x1E0053)
#define YR20_LOC (0x1E0054)
#define YR21_LOC (0x1E0055)
#define YR22_LOC (0x1E0056)
#define YR23_LOC (0x1E0057)
#define YR24_LOC (0x1E0058)
#define YR25_LOC (0x1E0059)
#define YR26_LOC (0x1E005A)
#define YR27_LOC (0x1E005B)
#define YR28_LOC (0x1E005C)
#define YR29_LOC (0x1E005D)
#define YR30_LOC (0x1E005E)
#define YR31_LOC (0x1E005F)

/* XY Comp Block Merged */

#define XYR0_LOC (0x1E0080)
#define XYR1_LOC (0x1E0081)
#define XYR2_LOC (0x1E0082)
#define XYR3_LOC (0x1E0083)
#define XYR4_LOC (0x1E0084)
#define XYR5_LOC (0x1E0085)
#define XYR6_LOC (0x1E0086)
```

```
#define XYR7_LOC (0x1E0087)
#define XYR8_LOC (0x1E0088)
#define XYR9_LOC (0x1E0089)
#define XYR10_LOC (0x1E008A)
#define XYR11_LOC (0x1E008B)
#define XYR12_LOC (0x1E008C)
#define XYR13_LOC (0x1E008D)
#define XYR14_LOC (0x1E008E)
#define XYR15_LOC (0x1E008F)
#define XYR16_LOC (0x1E0090)
#define XYR17_LOC (0x1E0091)
#define XYR18_LOC (0x1E0092)
#define XYR19_LOC (0x1E0093)
#define XYR20_LOC (0x1E0094)
#define XYR21_LOC (0x1E0095)
#define XYR22_LOC (0x1E0096)
#define XYR23_LOC (0x1E0097)
#define XYR24_LOC (0x1E0098)
#define XYR25_LOC (0x1E0099)
#define XYR26_LOC (0x1E009A)
#define XYR27_LOC (0x1E009B)
#define XYR28_LOC (0x1E009C)
#define XYR29_LOC (0x1E009D)
#define XYR30_LOC (0x1E009E)
#define XYR31_LOC (0x1E009F)
```

```
/* YX Comp Block Merged */
```

```
#define YXR0_LOC (0x1E00C0)
#define YXR1_LOC (0x1E00C1)
#define YXR2_LOC (0x1E00C2)
#define YXR3_LOC (0x1E00C3)
#define YXR4_LOC (0x1E00C4)
#define YXR5_LOC (0x1E00C5)
```

```
#define YXR6_LOC (0x1E00C6)
#define YXR7_LOC (0x1E00C7)
#define YXR8_LOC (0x1E00C8)
#define YXR9_LOC (0x1E00C9)
#define YXR10_LOC (0x1E00CA)
#define YXR11_LOC (0x1E00CB)
#define YXR12_LOC (0x1E00CC)
#define YXR13_LOC (0x1E00CD)
#define YXR14_LOC (0x1E00CE)
#define YXR15_LOC (0x1E00CF)
#define YXR16_LOC (0x1E00D0)
#define YXR17_LOC (0x1E00D1)
#define YXR18_LOC (0x1E00D2)
#define YXR19_LOC (0x1E00D3)
#define YXR20_LOC (0x1E00D4)
#define YXR21_LOC (0x1E00D5)
#define YXR22_LOC (0x1E00D6)
#define YXR23_LOC (0x1E00D7)
#define YXR24_LOC (0x1E00D8)
#define YXR25_LOC (0x1E00D9)
#define YXR26_LOC (0x1E00DA)
#define YXR27_LOC (0x1E00DB)
#define YXR28_LOC (0x1E00DC)
#define YXR29_LOC (0x1E00DD)
#define YXR30_LOC (0x1E00DE)
#define YXR31_LOC (0x1E00DF)

/* XY Comp Block Broadcast */

#define XYBR0_LOC (0x1E0100)
#define XYBR1_LOC (0x1E0101)
#define XYBR2_LOC (0x1E0102)
#define XYBR3_LOC (0x1E0103)
#define XYBR4_LOC (0x1E0104)
```

```
#define XYBR5_LOC (0x1E0105)
#define XYBR6_LOC (0x1E0106)
#define XYBR7_LOC (0x1E0107)
#define XYBR8_LOC (0x1E0108)
#define XYBR9_LOC (0x1E0109)
#define XYBR10_LOC (0x1E010A)
#define XYBR11_LOC (0x1E010B)
#define XYBR12_LOC (0x1E010C)
#define XYBR13_LOC (0x1E010D)
#define XYBR14_LOC (0x1E010E)
#define XYBR15_LOC (0x1E010F)
#define XYBR16_LOC (0x1E0110)
#define XYBR17_LOC (0x1E0111)
#define XYBR18_LOC (0x1E0112)
#define XYBR19_LOC (0x1E0113)
#define XYBR20_LOC (0x1E0114)
#define XYBR21_LOC (0x1E0115)
#define XYBR22_LOC (0x1E0116)
#define XYBR23_LOC (0x1E0117)
#define XYBR24_LOC (0x1E0118)
#define XYBR25_LOC (0x1E0119)
#define XYBR26_LOC (0x1E011A)
#define XYBR27_LOC (0x1E011B)
#define XYBR28_LOC (0x1E011C)
#define XYBR29_LOC (0x1E011D)
#define XYBR30_LOC (0x1E011E)
#define XYBR31_LOC (0x1E011F)
```

```
/* JALU */
```

```
#define J0_LOC (0x1E0180)
#define J1_LOC (0x1E0181)
#define J2_LOC (0x1E0182)
#define J3_LOC (0x1E0183)
```

```
#define J4_LOC (0x1E0184)
#define J5_LOC (0x1E0185)
#define J6_LOC (0x1E0186)
#define J7_LOC (0x1E0187)
#define J8_LOC (0x1E0188)
#define J9_LOC (0x1E0189)
#define J10_LOC (0x1E018A)
#define J11_LOC (0x1E018B)
#define J12_LOC (0x1E018C)
#define J13_LOC (0x1E018D)
#define J14_LOC (0x1E018E)
#define J15_LOC (0x1E018F)
#define J16_LOC (0x1E0190)
#define J17_LOC (0x1E0191)
#define J18_LOC (0x1E0192)
#define J19_LOC (0x1E0193)
#define J20_LOC (0x1E0194)
#define J21_LOC (0x1E0195)
#define J22_LOC (0x1E0196)
#define J23_LOC (0x1E0197)
#define J24_LOC (0x1E0198)
#define J25_LOC (0x1E0199)
#define J26_LOC (0x1E019A)
#define J27_LOC (0x1E019B)
#define J28_LOC (0x1E019C)
#define J29_LOC (0x1E019D)
#define J30_LOC (0x1E019E)
#define J31_LOC (0x1E019F)

/* KALU */

#define K0_LOC (0x1E01A0)
#define K1_LOC (0x1E01A1)
#define K2_LOC (0x1E01A2)
```

```
#define K3_LOC (0x1E01A3)
#define K4_LOC (0x1E01A4)
#define K5_LOC (0x1E01A5)
#define K6_LOC (0x1E01A6)
#define K7_LOC (0x1E01A7)
#define K8_LOC (0x1E01A8)
#define K9_LOC (0x1E01A9)
#define K10_LOC (0x1E01AA)
#define K11_LOC (0x1E01AB)
#define K12_LOC (0x1E01AC)
#define K13_LOC (0x1E01AD)
#define K14_LOC (0x1E01AE)
#define K15_LOC (0x1E01AF)
#define K16_LOC (0x1E01B0)
#define K17_LOC (0x1E01B1)
#define K18_LOC (0x1E01B2)
#define K19_LOC (0x1E01B3)
#define K20_LOC (0x1E01B4)
#define K21_LOC (0x1E01B5)
#define K22_LOC (0x1E01B6)
#define K23_LOC (0x1E01B7)
#define K24_LOC (0x1E01B8)
#define K25_LOC (0x1E01B9)
#define K26_LOC (0x1E01BA)
#define K27_LOC (0x1E01BB)
#define K28_LOC (0x1E01BC)
#define K29_LOC (0x1E01BD)
#define K30_LOC (0x1E01BE)
#define K31_LOC (0x1E01BF)

/* JALU Circular */

#define JB0_LOC (0x1E01C0)
#define JB1_LOC (0x1E01C1)
```



```
#define JB2_LOC (0x1E01C2)
#define JB3_LOC (0x1E01C3)
#define JL0_LOC (0x1E01C4)
#define JL1_LOC (0x1E01C5)
#define JL2_LOC (0x1E01C6)
#define JL3_LOC (0x1E01C7)

/* KALU Circular */

#define KB0_LOC (0x1E01E0)
#define KB1_LOC (0x1E01E1)
#define KB2_LOC (0x1E01E2)
#define KB3_LOC (0x1E01E3)
#define KL0_LOC (0x1E01E4)
#define KL1_LOC (0x1E01E5)
#define KL2_LOC (0x1E01E6)
#define KL3_LOC (0x1E01E7)

/* Sequencer Registers */

#define CJMP_LOC (0x1E0340)
#define RETI_LOC (0x1E0342)
#define RETS_LOC (0x1E0344)
#define DBGE_LOC (0x1E0345)
#define LC0_LOC (0x1E0348)
#define LC1_LOC (0x1E0349)

/*

#define IVSW_LOC (0x1E0350)

/* Flag Control, Set, and Clear Registers with Bit Defines
#define FLAGREG_LOC (0x1E0354)
#define FLAGREGST_LOC (0x1E0355)
```

```

#define FLAGREGCL_LOC (0x1E0356)

// Bit positions
#define FLAGREG_FLAG0_EN_P (0) // FLAG0 output enable is bit 0 in
FLAGREG
#define FLAGREG_FLAG1_EN_P (1) // FLAG1 output enable is bit 1 in
FLAGREG
#define FLAGREG_FLAG2_EN_P (2) // FLAG2 output enable is bit 2 in
FLAGREG
#define FLAGREG_FLAG3_EN_P (3) // FLAG3 output enable is bit 3 in
FLAGREG
#define FLAGREG_FLAG0_OUT_P (4) // FLAG0 out pin is bit 4 in
FLAGREG
#define FLAGREG_FLAG1_OUT_P (5) // FLAG1 out pin is bit 5 in
FLAGREG
#define FLAGREG_FLAG2_OUT_P (6) // FLAG2 out pin is bit 6 in
FLAGREG
#define FLAGREG_FLAG3_OUT_P (7) // FLAG3 out pin is bit 7 in
FLAGREG

// Bit masks
#define FLAGREG_FLAG0_EN MAKE_BITMASK_(FLAGREG_FLAG0_EN_P)
#define FLAGREG_FLAG1_EN MAKE_BITMASK_(FLAGREG_FLAG1_EN_P)
#define FLAGREG_FLAG2_EN MAKE_BITMASK_(FLAGREG_FLAG2_EN_P)
#define FLAGREG_FLAG3_EN MAKE_BITMASK_(FLAGREG_FLAG3_EN_P)
#define FLAGREG_FLAG0_OUT MAKE_BITMASK_(FLAGREG_FLAG0_OUT_P)
#define FLAGREG_FLAG1_OUT MAKE_BITMASK_(FLAGREG_FLAG1_OUT_P)
#define FLAGREG_FLAG2_OUT MAKE_BITMASK_(FLAGREG_FLAG2_OUT_P)
#define FLAGREG_FLAG3_OUT MAKE_BITMASK_(FLAGREG_FLAG3_OUT_P)

/* SQCTL With Bit Defines */
#define SQCTL_LOC (0x1E0358)
#define SQCTLST_LOC (0x1E0359)
#define SQCTLCL_LOC (0x1E035A)

```

```
// Bit positions
#define SQCTL_GIE_P (2)
#define SQCTL_SW_P (3)
#define SQCTL_DBGDSBL_P (8)
#define SQCTL_NMOD_P (9)
#define SQCTL_TRCBEN_P (10)
#define SQCTL_TRCBEXEN_P (11)

// Bit Masks
#define SQCTL_GIE MAKE_BITMASK_(SQCTL_GIE_P)
#define SQCTL_SW MAKE_BITMASK_(SQCTL_SW_P)
#define SQCTL_DBGDSBL MAKE_BITMASK_(SQCTL_DBGDSBL_P)
#define SQCTL_NMOD MAKE_BITMASK_(SQCTL_NMOD_P)
#define SQCTL_TRCBEN MAKE_BITMASK_(SQCTL_TRCBEN_P)
#define SQCTL_TRCBEXEN MAKE_BITMASK_(SQCTL_TRCBEXEN_P)

/* SQSTAT With Bit Defines */

#define SQSTAT_LOC (0x1E035B)

// 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)
#define SQSTAT_GIE_P (20)
#define SQSTAT_SW_P (21)
#define SQSTAT_EMU_P (22)
#define SQSTAT_BT BEN_P (24)
#define SQSTAT_BTBLK_P (25)
```

```

// 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 (0x1E035C)

// 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 )

/* Emulation Registers I */

/* Watchpoint Registers */

#define WPOCTL_LOC (0x1E0360)
#define WP1CTL_LOC (0x1E0361)

```

```
#define WP2CTL_LOC (0x1E0362)

// Bit Masks
// OPMODE
#define WPCTL_DSBL (0x00000000)
#define WPCTL_ADDRESS (0x00000001)
#define WPCTL_RANGE (0x00000002)
#define WPCTL_NOTRANGE (0x00000003)
// EXTYPE
#define WPCTL_NOEXCEPT (0x00000000)
#define WPCTL_EXCEPT (0x00000004)
#define WPCTL_EMUTRAP (0x00000008)
// SSTP, WPOCTL
#define WPCTL_SSTP (0x00000010)
// R/W, WP1CTL and WP2CTL
#define WPCTL_READ (0x00000010)
#define WPCTL_WRITE (0x00000020)
// JK, WP1CTL
#define WPCTL_JK_JBUS (0x00000040)
#define WPCTL_JK_KBUS (0x00000080)
#define WPCTL_JK_BOTH (0x000000C0)

#define WPOSTAT_LOC (0x1E0364)
#define WP1STAT_LOC (0x1E0365)
#define WP2STAT_LOC (0x1E0366)

// 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 (0x1E0368)
#define WP0H_LOC (0x1E0369)
#define WP1L_LOC (0x1E036A)
#define WP1H_LOC (0x1E036B)
#define WP2L_LOC (0x1E036C)
#define WP2H_LOC (0x1E036D)

#define CCNT0_LOC (0x1E0370)
#define CCNT1_LOC (0x1E0371)
#define PRFM_LOC (0x1E0372)

// Bit Masks
// Non Granted Requests
#define PRFM_NGR_SEQ (0)
#define PRFM_NGR_JALU (1)
#define PRFM_NGR_KALU (2)
#define PRFM_NGR_SOC (3)

// Granted Requests
#define PRFM_GR_SEQ (4)
#define PRFM_GR_JALU (5)
#define PRFM_GR_KALU (6)
#define PRFM_GR_SOC (7)

// Module Used
#define PRFM_MODULE_JALU (8)
#define PRFM_MODULE_KALU (9)
#define PRFM_MODULE_CBX (10)
#define PRFM_MODULE_CBY (11)
#define PRFM_MODULE_CTRL (12)

#define PRFM_SCYCLE (16)
#define PRFM_BTBP (17)
#define PRFM_ISL (18)

```

```
#define PRFM_CCYCLE (19)
#define PRFM_SUMEN (31)

#define PRFCNT_LOC (0x1E0373)

#define TRCBMASK_LOC (0x1E0374)
#define TRCBPTR_LOC (0x1E0375)

/* Cache Registers With Bit Defines *
// Command registers
#define CACMD0_LOC (0x1E03C0)
#define CACMD2_LOC (0x1E03C8)
#define CACMD4_LOC (0x1E03D0)
#define CACMD6_LOC (0x1E03D8)
#define CACMD8_LOC (0x1E03E0)
#define CACMD10_LOC (0x1E03E8)
#define CACMDB_LOC (0x1E03FC)

// Bit Masks
#define CACMD_EN (0x00000000)
#define CACMD_DIS (0x04000000)
#define CACMD_SET_BUS (0x1c000000)
#define CACMD_SLOCK (0x08000000)
#define CACMD_ELOCK (0x0c000000)
#define CACMD_CB (0x10000000)
#define CACMD_INV (0x14000000)
#define CACMD_INIT (0x40000000)
#define CACMD_INIT_LOCK (0x44000000)
#define CACMD_REFRESH (0x50000000)
//alternative definitions to keep compatibility - added 7/13/04
#define CACHE_EN (0x00000000)
#define CACHE_DIS (0x04000000)
#define CACHE_SET_BUS (0x1c000000)
#define CACHE_SLOCK (0x08000000)
```

```

#define CACHE_ELOCK      (0x0c000000)
#define CACHE_CB         (0x10000000)
#define CACHE_INV        (0x14000000)
#define CACHE_INIT       (0x40000000)
#define CACHE_INIT_LOCK  (0x44000000)
#define CACHE_REFRESH    (0x50000000)

#define CACMD_NOSTALL    (0x00004000)
//alternative definition to keep compatibility - added 7/13/04
#define CACHE_NOSTALL    (0x00004000)

#define CACMD_K_BUS_N_   (0x000000)
#define CACMD_K_BUS_R_   (0x008000)
#define CACMD_K_BUS_W_   (0x010000)
#define CACMD_K_BUS_RW   (0x018000)
#define CACMD_J_BUS_N_   (0x000000)
#define CACMD_J_BUS_R_   (0x020000)
#define CACMD_J_BUS_W_   (0x040000)
#define CACMD_J_BUS_RW   (0x060000)
#define CACMD_S_BUS_N_   (0x000000)
#define CACMD_S_BUS_R_   (0x080000)
#define CACMD_S_BUS_W_   (0x100000)
#define CACMD_S_BUS_RW   (0x180000)
#define CACMD_I_BUS_N_   (0x000000)
#define CACMD_I_BUS_R_   (0x200000)
//alternative definitions to keep compatibility - added 7/13/04
#define CACHE_K_BUS_N_   (0x000000)
#define CACHE_K_BUS_R_   (0x008000)
#define CACHE_K_BUS_W_   (0x010000)
#define CACHE_K_BUS_RW   (0x018000)
#define CACHE_J_BUS_N_   (0x000000)
#define CACHE_J_BUS_R_   (0x020000)
#define CACHE_J_BUS_W_   (0x040000)
#define CACHE_J_BUS_RW   (0x060000)

```



```
#define CACHE_S_BUS_N_ (0x000000)
#define CACHE_S_BUS_R_ (0x080000)
#define CACHE_S_BUS_W_ (0x100000)
#define CACHE_S_BUS_RW (0x180000)
#define CACHE_I_BUS_N_ (0x000000)
#define CACHE_I_BUS_R_ (0x200000)

// Bit Positions
#define CACMD_LEN_P (15)
//alternative definition to keep compatibility - added 7/13/04
#define CACHE_LEN_P (15)

// Address/Index registers
#define CCAIRO_LOC (0x1E03C1)
#define CCAIR2_LOC (0x1E03C9)
#define CCAIR4_LOC (0x1E03D1)
#define CCAIR6_LOC (0x1E03D9)
#define CCAIR8_LOC (0x1E03E1)
#define CCAIR10_LOC (0x1E03E9)
#define CCAIRB_LOC (0x1E03FD)

// Status registers
#define CASTAT0_LOC (0x1E03C2)
#define CASTAT2_LOC (0x1E03CA)
#define CASTAT4_LOC (0x1E03D2)
#define CASTAT6_LOC (0x1E03DA)
#define CASTAT8_LOC (0x1E03E2)
#define CASTAT10_LOC (0x1E03EA)

// Bit Positions
#define CASTAT_ENBL_P (14)
#define CASTAT_LOCK_P (15)
#define CASTAT_COM_ACTIVE_P (16)
#define CASTAT_COM_ABRTD_P (17)
```

```

#define CASTAT_STL_ACTIVE_P (19)
#define CASTAT_K_CACHING_P (20)
#define CASTAT_J_CACHING_P (22)
#define CASTAT_S_CACHING_P (24)
#define CASTAT_I_CACHING_P (26)

// Bit Masks
#define CASTAT_REFCNTR (0x00003fff)
#define CASTAT_ENBL MAKE_BITMASK_(CASTAT_ENBL_P)
#define CASTAT_LOCK MAKE_BITMASK_(CASTAT_LOCK_P)
#define CASTAT_COM_ACTIVE MAKE_BITMASK_(CASTAT_COM_ACTIVE_P)
#define CASTAT_COM_ABRTD MAKE_BITMASK_(CASTAT_COM_ABRTD_P)

#define CADATA0_LOC (0x1E03C3)
#define CADATA2_LOC (0x1E03CB)
#define CADATA4_LOC (0x1E03D3)
#define CADATA6_LOC (0x1E03DB)
#define CADATA8_LOC (0x1E03E3)
#define CADATA10_LOC (0x1E03EB)
#define CADATAB_LOC (0x1E03FF)

/* Cache Commands Macros */

#if !defined(SET_REFRESH0_)
#define SET_REFRESH0_(x_) () // Make a bit mask from a bit
position
#endif

/* TCBS With Bit Defines */

#define DCS0_LOC (0x1F0000)
#define DCD0_LOC (0x1F0004)
#define DCS1_LOC (0x1F0008)
#define DCD1_LOC (0x1F000c)

```

```
#define DCS2_LOC (0x1F0010)
#define DCD2_LOC (0x1F0014)
#define DCS3_LOC (0x1F0018)
#define DCD3_LOC (0x1F001C)
#define DC4_LOC (0x1F0020)
#define DC5_LOC (0x1F0024)
#define DC6_LOC (0x1F0028)
#define DC7_LOC (0x1F002C)
#define DC8_LOC (0x1F0040)
#define DC9_LOC (0x1F0044)
#define DC10_LOC (0x1F0048)
#define DC11_LOC (0x1F004C)
#define DC12_LOC (0x1F0058)
#define DC13_LOC (0x1F005C)

// TYPES (TY)
#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 (PR)
#define TCB_HPRIORITY (0x10000000)
// 2DDMA
#define TCB_TWODIM (0x08000000)
// OPERAND LENGTH (LEN)
#define TCB_QUAD (0x06000000)
#define TCB_LONG (0x04000000)
#define TCB_NORMAL (0x02000000)
// INTERRUPT (INT)
#define TCB_INT (0x01000000)
// DMA REQUEST (DRQ)
#define TCB_DMAR (0x00800000)
```

```

// CHAINING ENABLE (CHEN)
#define TCB_CHAIN (0x00400000)
// CHAINED CHANNEL (CHTG)
#define TCB_DMA8DEST (0x00000000)
#define TCB_DMA9DEST (0x00080000)
#define TCB_DMA10DEST (0x00100000)
#define TCB_DMA11DEST (0x00180000)
#define TCB_DMA4DEST (0x00200000)
#define TCB_DMA5DEST (0x00280000)
#define TCB_DMA6DEST (0x00300000)
#define TCB_DMA7DEST (0x00380000)

/* DMA Controls With Bit Defines */
#define DCNT_LOC (0x1F0060)
#define DCNTST_LOC (0x1F0064)
#define DCNTCL_LOC (0x1F0068)

// 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 (0x1F006C)
#define DSTATCL_LOC (0x1F0070)

// 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 DSTATH6 (0x1203) // 18th position of length 3
#define DSTATH7 (0x1503) // 21st position of length 3

#define DSTATH_LOC (0x1F006D)
#define DSTATHCH_LOC (0x1F0071)

#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 (0x1F0080)

// 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_MS0_PIPE1 (0x00000000)
#define SYSCON_MS0_PIPE2 (0x00000008)
#define SYSCON_MS0_PIPE3 (0x00000010)
#define SYSCON_MS0_PIPE4 (0x00000018)
#define SYSCON_MS0_SLOW (0x00000020)
#define SYSCON_MS1_IDLE (SYSCON_MS0_IDLE << 6)
#define SYSCON_MS1_WT0 (SYSCON_MS0_WT0 << 6)
#define SYSCON_MS1_WT1 (SYSCON_MS0_WT1 << 6)
#define SYSCON_MS1_WT2 (SYSCON_MS0_WT2 << 6)
#define SYSCON_MS1_WT3 (SYSCON_MS0_WT3 << 6)
#define SYSCON_MS1_PIPE1 (SYSCON_MS0_PIPE1 << 6)
#define SYSCON_MS1_PIPE2 (SYSCON_MS0_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)

/* BUSLOCK register *
#define BUSLOCK_LOC (0x1F0083)

/* SDRCON register With Bit Masks *
#define SDRCON_LOC (0x1F0084)

// 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_REF1100 (0x00000000)
#define SDRCON_REF1850 (0x00000080)
```

```

#define SDRCON_REF2200 (0x00000100)
#define SDRCON_REF3700 (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)
#define SDRCON_EMRS (0x00008000)

/* SYSTAT registers */
#define SYSTAT_LOC (0x1F0086)
#define SYSTATCL_LOC (0x1F0087)

/* BMAX registers */
#define BMAX_LOC (0x1F008C)
#define BMAXC_LOC (0x1F008D)

/* Link Buffer Registers - changed 7/13/04 to meet new spec */
#define LBUFTX0_LOC (0x1F04A0)
#define LBUFRX0_LOC (0x1F04A4)
#define LBUFTX1_LOC (0x1F04A8)
#define LBUFRX1_LOC (0x1F04AC)
#define LBUFTX2_LOC (0x1F04B0)
#define LBUFRX2_LOC (0x1F04B4)
#define LBUFTX3_LOC (0x1F04B8)
#define LBUFRX3_LOC (0x1F04BC)

```



```
/* Link Receive Control Registers with Bit Masks */
#define LRCTL0_LOC (0x1F00E0)
#define LRCTL1_LOC (0x1F00E1)
#define LRCTL2_LOC (0x1F00E2)
#define LRCTL3_LOC (0x1F00E3)

// Bit positions
#define LRCTL_REN_P (0)
#define LRCTL_RVERE_P (1)
#define LRCTL_RTOE_P (2)
#define LRCTL_RBCMPE_P (3)
#define LRCTL_RDSIZE_P (4)
#define LRCTL_ROVRE_P (5)
#define LRCTL_RINIF_P (6)
#define LRCTL_RINIV_P (7)

// Bit Masks
#define LRCTL_REN MAKE_BITMASK_(LRCTL_REN_P)
#define LRCTL_RVERE MAKE_BITMASK_(LRCTL_RVERE_P)
#define LRCTL_RTOE MAKE_BITMASK_(LRCTL_RTOE_P)
#define LRCTL_RBCMPE MAKE_BITMASK_(LRCTL_RBCMPE_P)
#define LRCTL_RDSIZE MAKE_BITMASK_(LRCTL_RDSIZE_P)
#define LRCTL_ROVRE MAKE_BITMASK_(LRCTL_ROVRE_P)

/* Link Transmit Control Registers with Bit Masks */
#define LTCTL0_LOC (0x1F00E4)
#define LTCTL1_LOC (0x1F00E5)
#define LTCTL2_LOC (0x1F00E6)
#define LTCTL3_LOC (0x1F00E7)

// Bit positions
#define LTCTL_TEN_P (0)
#define LTCTL_TVERE_P (1)
#define LTCTL_TTOE_P (2)
```

```

#define LTCTL_TBCMPE_P (3)
#define LTCTL_TDSIZE_P (4)

// Bit Masks
#define LTCTL_TEN MAKE_BITMASK_(LTCTL_TEN_P)
#define LTCTL_TVERE MAKE_BITMASK_(LTCTL_TVERE_P)
#define LTCTL_TTOE MAKE_BITMASK_(LTCTL_TTOE_P)
#define LTCTL_TBCMPE MAKE_BITMASK_(LTCTL_TBCMPE_P)
#define LTCTL_TDSIZE MAKE_BITMASK_(LTCTL_TDSIZE_P)
#define LTCTL_TCLKDIV1 (0x00000000)
#define LTCTL_TCLKDIV1P5 (0x00000020)
#define LTCTL_TCLKDIV2 (0x00000040)
#define LTCTL_TCLKDIV4 (0x00000080)

/* Link Status Registers with Bit Masks */
#define LRSTAT0_LOC (0x1F00F0)
#define LRSTAT1_LOC (0x1F00F1)
#define LRSTAT2_LOC (0x1F00F2)
#define LRSTAT3_LOC (0x1F00F3)

#define LTSTAT0_LOC (0x1F00F4)
#define LTSTAT1_LOC (0x1F00F5)
#define LTSTAT2_LOC (0x1F00F6)
#define LTSTAT3_LOC (0x1F00F7)

#define LRSTATC0_LOC (0x1F00F8)
#define LRSTATC1_LOC (0x1F00F9)
#define LRSTATC2_LOC (0x1F00FA)
#define LRSTATC3_LOC (0x1F00FB)

#define LTSTATC0_LOC (0x1F00FC)
#define LTSTATC1_LOC (0x1F00FD)
#define LTSTATC2_LOC (0x1F00FE)
#define LTSTATC3_LOC (0x1F00FF)

```

```
// Bit positions - Receive Status
#define LRSTAT_RTER_P (2)
#define LRSTAT_RWER_P (3)
#define LRSTAT_RCSE_P (4)
#define LRSTAT_ROVER_P (5)

// Bit positions - Transmit Status
#define LTSTAT_TVACANT_P (0)
#define LTSTAT_TEMP_P (1)
#define LTSTAT_TTER_P (2)
#define LTSTAT_TWER_P (3)

// Bit Masks - Receive Status
#define LRSTAT_RSTAT (0x00000003)
#define LRSTAT_RTER MAKE_BITMASK_(LRSTAT_RTER_P)
#define LRSTAT_RWER MAKE_BITMASK_(LRSTAT_RWER_P)
#define LRSTAT_RCSE MAKE_BITMASK_(LRSTAT_RCSE_P)
#define LRSTAT_ROVER MAKE_BITMASK_(LRSTAT_ROVER_P)

// Bit Masks - Transmit Status
#define LTSTAT_TVACANT MAKE_BITMASK_(LTSTAT_TVACANT_P)
#define LTSTAT_TEMP MAKE_BITMASK_(LTSTAT_TEMP_P)
#define LTSTAT_TTER MAKE_BITMASK_(LTSTAT_TTER_P)
#define LTSTAT_TWER MAKE_BITMASK_(LTSTAT_TWER_P)

/* Interrupt Vectors (except SW interrupt) */

#define IVKERNEL_LOC (0x1F0300)
#define IVTIMER0LP_LOC (0x1F0302)
#define IVTIMER1LP_LOC (0x1F0303)
#define IVLINK0_LOC (0x1F0306)
#define IVLINK1_LOC (0x1F0307)
#define IVLINK2_LOC (0x1F0308)
```

```

#define IVLINK3_LOC (0x1F0309)
#define IVDMA0_LOC (0x1F030E)
#define IVDMA1_LOC (0x1F030F)
#define IVDMA2_LOC (0x1F0310)
#define IVDMA3_LOC (0x1F0311)
#define IVDMA4_LOC (0x1F0316)
#define IVDMA5_LOC (0x1F0317)
#define IVDMA6_LOC (0x1F0318)
#define IVDMA7_LOC (0x1F0319)
#define IVDMA8_LOC (0x1F031D)
#define IVDMA9_LOC (0x1F031E)
#define IVDMA10_LOC (0x1F031F)
#define IVDMA11_LOC (0x1F0320)
#define IVDMA12_LOC (0x1F0325)
#define IVDMA13_LOC (0x1F0326)
#define IVIRQ0_LOC (0x1F0329)
#define IVIRQ1_LOC (0x1F032A)
#define IVIRQ2_LOC (0x1F032B)
#define IVIRQ3_LOC (0x1F032C)
#define VIRPT_LOC (0x1F0330)
#define IVBUSLK_LOC (0x1F0332)
#define IVBUSLOCK_LOC (0x1F0332) // added 5/6/04
per request of VDK to maintain compatibility with TS101
#define IVTIMER0HP_LOC (0x1F0334)
#define IVTIMER1HP_LOC (0x1F0335)
#define IVHW_LOC (0x1F0339)

/* ILAT, IMASK and PMASK with bit defines */

#define ILATL_LOC (0x1F0340)
#define ILATH_LOC (0x1F0341)
#define ILATSTL_LOC (0x1F0342)
#define ILATSTH_LOC (0x1F0343)
#define ILATCLL_LOC (0x1F0344)

```

```
#define ILATCLH_LOC (0x1F0345)
#define PMASKL_LOC (0x1F0346)
#define PMASKH_LOC (0x1F0347)
#define IMASKL_LOC (0x1F0348)
#define IMASKH_LOC (0x1F0349)

// Bit positions
#define INT_KERNEL_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_HWERR_P (25)
#define INT_RES58_P (26)
#define INT_RES59_P (27)
#define INT_RES61_P (29)
```

```
// Bit Masks for C only
#define INT_KERNEL_64 MAKE_LL_BITMASK_(INT_KERNEL_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_HWERR_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_RES61_64 MAKE_LL_BITMASK_(INT_RES61_P + 32)

// Bit Masks
#define INT_KERNEL MAKE_BITMASK_(INT_KERNEL_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_RES61 MAKE_BITMASK_(INT_RES61_P )

/* Interrupt Control Register with Bit Defines */
#define INTCTL_LOC (0x1F034E)

// Bit positions
#define INTCTL_IRQ0_EDGE_P (0)
#define INTCTL_IRQ1_EDGE_P (1)
#define INTCTL_IRQ2_EDGE_P (2)
#define INTCTL_IRQ3_EDGE_P (3)
#define INTCTL_TMRORN_P (4)

```

```
#define INTCTL_TMR1RN_P (5)

// Bit masks
#define INTCTL_IRQ0_EDGE MAKE_BITMASK_(INTCTL_IRQ0_EDGE_P )
#define INTCTL_IRQ1_EDGE MAKE_BITMASK_(INTCTL_IRQ1_EDGE_P )
#define INTCTL_IRQ2_EDGE MAKE_BITMASK_(INTCTL_IRQ2_EDGE_P )
#define INTCTL_IRQ3_EDGE MAKE_BITMASK_(INTCTL_IRQ3_EDGE_P )
#define INTCTL_TMR0RN MAKE_BITMASK_(INTCTL_TMR0RN_P )
#define INTCTL_TMR1RN MAKE_BITMASK_(INTCTL_TMR1RN_P )

/* Timer Registers */
#define TIMER0L_LOC (0x1F0350)
#define TIMER0H_LOC (0x1F0351)
#define TIMER1L_LOC (0x1F0352)
#define TIMER1H_LOC (0x1F0353)
#define TMRIN0L_LOC (0x1F0354)
#define TMRIN0H_LOC (0x1F0355)
#define TMRIN1L_LOC (0x1F0356)
#define TMRIN1H_LOC (0x1F0357)

/* Emulation Registers II */

/* EMUCTL With Bit Defines */
#define EMUCTL_LOC (0x1F03A0)

// Bit positions
#define EMUCTL_EMEN_P (0)
#define EMUCTL_TEME_P (1)
#define EMUCTL_EMUOE_P (2)
#define EMUCTL_SPFDIS_P (3)
#define EMUCTL_SWRST_P (4)
#define EMUCTL_BOOTDSBL_P (5)

// 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 EMUCTL_SWRST MAKE_BITMASK_(EMUCTL_SWRST_P)
#define EMUCTL_BOOTDSBL MAKE_BITMASK_(EMUCTL_BOOTDSBL_P)

/* EMUSTAT With Bit Defines */
#define EMUSTAT_LOC (0x1F03A1)

// Bit positions
#define EMUSTAT_EMUMOD_P (0)
#define EMUSTAT_IRFREE_P (1)
#define EMUSTAT_INRESET_P (2)

// Bit Masks
#define EMUSTAT_EMUMOD MAKE_BITMASK_(EMUSTAT_EMUMOD_P)
#define EMUSTAT_IRFREE MAKE_BITMASK_(EMUSTAT_IRFREE_P)
#define EMUSTAT_INRESET MAKE_BITMASK_(EMUSTAT_INRESET_P)

/*
#define EMUDAT_LOC (0x1F03A2)
#define IDCODE_LOC (0x1F03A3)
#define EMUIR_LOC (0x1F03A4)

/* Trace Buffer Registers */

#define TRCB0_LOC (0x1E0140)
#define TRCB1_LOC (0x1E0141)
#define TRCB2_LOC (0x1E0142)
#define TRCB3_LOC (0x1E0143)
#define TRCB4_LOC (0x1E0144)
#define TRCB5_LOC (0x1E0145)
#define TRCB6_LOC (0x1E0146)

```

```
#define TRCB7_LOC (0x1E0147)
#define TRCB8_LOC (0x1E0148)
#define TRCB9_LOC (0x1E0149)
#define TRCB10_LOC (0x1E014A)
#define TRCB11_LOC (0x1E014B)
#define TRCB12_LOC (0x1E014C)
#define TRCB13_LOC (0x1E014D)
#define TRCB14_LOC (0x1E014E)
#define TRCB15_LOC (0x1E014F)
#define TRCB16_LOC (0x1E0150)
#define TRCB17_LOC (0x1E0151)
#define TRCB18_LOC (0x1E0152)
#define TRCB19_LOC (0x1E0153)
#define TRCB20_LOC (0x1E0154)
#define TRCB21_LOC (0x1E0155)
#define TRCB22_LOC (0x1E0156)
#define TRCB23_LOC (0x1E0157)
#define TRCB24_LOC (0x1E0158)
#define TRCB25_LOC (0x1E0159)
#define TRCB26_LOC (0x1E015A)
#define TRCB27_LOC (0x1E015B)
#define TRCB28_LOC (0x1E015C)
#define TRCB29_LOC (0x1E015D)
#define TRCB30_LOC (0x1E015E)
#define TRCB31_LOC (0x1E015F)

/*

#define AUTODMA0_LOC (0x1F03E0)
#define AUTODMA1_LOC (0x1F03E4)

/* Global memory *

#define BLOCK0_LOC (0x00000000) // Internal memory block 0
```

```

#define BLOCK2_LOC (0x00040000) // Internal memory block 2
#define BLOCK4_LOC (0x00080000) // Internal memory block 4
#define BLOCK6_LOC (0x000C0000) // Internal memory block 6
#define BLOCK8_LOC (0x00100000) // Internal memory block 8
#define BLOCK10_LOC (0x00140000) // Internal memory block 10

#define BCST_OFFSET_LOC (0x0C000000) // Broadcast MP memory
offset
#define P0_OFFSET_LOC (0x10000000) // Processor ID0 MP memory
offset
#define P1_OFFSET_LOC (0x14000000) // Processor ID1 MP memory
offset
#define P2_OFFSET_LOC (0x18000000) // Processor ID2 MP memory
offset
#define P3_OFFSET_LOC (0x1C000000) // Processor ID3 MP memory
offset
#define P4_OFFSET_LOC (0x20000000) // Processor ID4 MP memory
offset
#define P5_OFFSET_LOC (0x24000000) // Processor ID5 MP memory
offset
#define P6_OFFSET_LOC (0x28000000) // Processor ID6 MP memory
offset
#define P7_OFFSET_LOC (0x2C000000) // Processor ID7 MP memory
offset

#endif // !defined(__DEFTS201_H_)

```

C INSTRUCTION DECODE

This appendix identifies operation codes (opcodes) for instructions. Use this appendix 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	11<XY>	Compute block CLU instruction; where 1101 – Y compute CLU, 1110 – X compute CLU, and 1111 – X and Y compute CLU
29–26	1100	Control Flow instructions, Immediate Extension and others.
30	CC/NF	For instruction lines begins with a sequencer condition instruction (if <condition>...), this bit is the CC bit. Otherwise this bit is NF bit. In the case of an instruction line that begins with conditional instruction, the NF bit is in the conditional instruction, common for all 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-40.) If bit 30 is CC (conditional code) bit, when set, conditions the execution of the instruction by the condition instruction in the instruction line. If bit 30 is NF (no flag update) bit, when set, the current instruction does not update flags at the end of the instruction execution.
31	EX	When the EX bit is set, determines the instruction as the last in the instruction line.

Compute Block Instruction Format

The instruction format for all ALU, multiplier, and some shifter instructions is as shown in [Figure C-2](#) and [Table C-2](#).

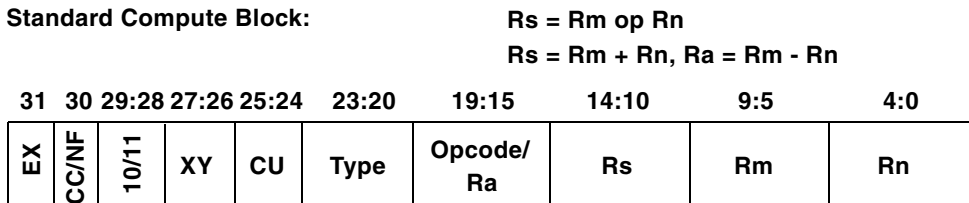


Figure C-2. Standard Compute Block Instruction Format

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/ RA	See Table C-3 on page C-5 , Table C-4 on page C-7 , Table C-5 on page C-10 , Table C-7 on page C-16 , Table C-8 on page C-20 , Table C-9 on page C-21 , Table C-10 on page C-22 , Table C-11 on page C-23 , Table C-12 on page C-25 , Table C-23 on page C-43 , and Table C-24 on page C-44 .
23–20	TYPE	See Table C-3 on page C-5 , Table C-4 on page C-7 , Table C-5 on page C-10 , Table C-7 on page C-16 , Table C-8 on page C-20 , Table C-9 on page C-21 , Table C-10 on page C-22 , Table C-11 on page C-23 , Table C-12 on page C-25 , Table C-23 on page C-43 , and Table C-24 on page C-44 .

Compute Block Instruction Format

Table C-2. Compute Block Instruction Opcode Fields (Cont'd)

Bits	Field	Description
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
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 block's ALU, multiplier, or shifter.
29–28	11	Determine that this is an operation to be executed by the compute block's CLU.
30	CC/NF	For instruction lines begins with a sequencer condition instruction (if <condition>...), this bit is the CC bit. Otherwise this bit is NF bit. In the case of an instruction line that begins with conditional instruction, the NF bit is in the conditional instruction, common for all 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-40.) If bit 30 is CC (conditional code) bit, when set, conditions the execution of the instruction by the condition instruction in the instruction line. If bit 30 is NF (no flag update) bit, when set, the current instruction does not update flags at the end of the instruction execution.
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” on page C-10

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/SRsd 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
$(B/S/L)Rs(d) = ABS (Rm - Rn)$ ⁴		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
$(B/S/L)Rs(d) = Rm + Rn,$ $(B/S/L)Ra(d) = Rm - Rn$		0010-Rs 0101-(L)Rsd ¹ 1000-SRs 1011-BRs 1110-B/SRsd ²	-Ra-

- 1 The LSB of R_n (bit 0 of the instruction) is used to decode operand size, where:
 0 = determines dual normal
 1 = determines long word
- 2 The LSB of R_n (bit 0 of the instruction) is used to decode operand size, where:
 0 = determines octal byte
 1 = determines quad short
- 3 $S_1 S_0$ values are: no saturation () – 00, saturation signed (S) – 01,
 saturation unsigned (SU) – 11
- 4 Option X - extend for ABS
- 5 On instruction $R_s = (R_m +/- R_n) / 2$ 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	
$R_{sd} = \text{EXPAND } SR_m + SR_n$		0000	000	$T_1 T_0^1$
$R_{sq} = \text{EXPAND } SR_{md} +/- SR_{nd}^2$		0000	001	$T_1 T_0^1$
$SR_{sd} = \text{EXPAND } BR_m + BR_n$		0000	010	$T_1 T_0^1$
$SR_{sq} = \text{EXPAND } BR_{md} +/- BR_{nd}^2$		0000	011	$T_1 T_0^1$
$R_{sd} = \text{EXPAND } SR_m - SR_n$		0000	100	$T_1 T_0^1$
$SR_{sd} = \text{EXPAND } BR_m - BR_n$		0000	110	$T_1 T_0^1$

Compute Block Instruction Format

Table C-4. ALU Fixed-Point, Data Conversion Instruction Syntax and Opcodes (Cont'd)

Syntax	CU = 01 Fixed	Type	Opcode	
Rsd = EXPAND SRm	Rn = 00000	0000	111	T1 T0 ¹
Rsq = EXPAND SRmd	Rn = 00001	0000	111	T1 T0 ¹
SRsd = EXPAND BRm	Rn = 00010	0000	111	T1 T0 ¹
SRsq = EXPAND BRmd	Rn = 00011	0000	111	T1 T0 ¹
SRs = COMPACT Rmd +/- Rnd ²		0001	00	C2 C1 C0 ³
BRs = COMPACT SRmd +/- SRnd ²		0001	01	C2 C1 C0 ³
SRs = COMPACT Rmd		0001	10	C2 C1 C0 ³
BRs = COMPACT SRmd		0001	11	C2 C1 C0 ³
Rs =COMPACT LRmd		0011	01	100
LRs =COMPACT QRmd		0011	01	100
Rs = COMPACT LRmd	Rn=00000	0011	011	00
Rs = COMPACT LRmd (U)	Rn=00001	0011	011	00
Rs = COMPACT LRmd (IS)	Rn=00010	0011	011	00
Rs = COMPACT LRmd (ISU)	Rn=00011	0011	011	00
LRsd = COMPACT QRmq	Rn=00100	0011	011	00
LRsd = COMPACT QRmq (U)	Rn=00101	0011	011	00
LRsd = COMPACT QRmq (IS)	Rn=00110	0011	011	00
LRsd = COMPACT QRmq (ISU)	Rn=00111	0011	011	00
BRsd = MERGE Rm, Rn		0010	000	00
BRsq = MERGE Rmd, Rnd		0010	000	01
SRsd = MERGE Rm, Rn		0010	000	10
SRsq = MERGE Rmd, Rnd		0010	000	11
Rs = SUM SRm ⁴	Rn=00000	0010	001	S0 0 ⁵
Rs = SUM SRmd ⁴	Rn=00001	0010	001	S0 0 ⁵

Table C-4. ALU Fixed-Point, Data Conversion Instruction Syntax and Opcodes (Cont'd)

Syntax	CU = 01 Fixed	Type	Opcode	
$R_s = \text{SUM BR}_m^4$	$R_n=00010$	0010	001	$S_0 0^5$
$R_s = \text{SUM BR}_{md}^4$	$R_n=00011$	0010	001	$S_0 0^5$
$R_s = \text{ONES R}_m$	$R_n=00100$	0010	001	00
$R_s = \text{ONES R}_{md}$	$R_n=00101$	0010	001	00
$R_{sd} = \text{PR1:0}$	$R_n=00110$	0010	001	00
$R_s = \text{BFOINC R}_{md}$		0010	010	00
$\text{PR0} += \text{ABS}(\text{SR}_{md} - \text{SR}_{nd})$		0011	000	$S_0 0^5$
$\text{PR0} += \text{ABS}(\text{BR}_{md} - \text{BR}_{nd})$		0011	001	$S_0 0^5$
$\text{PR1} += \text{ABS}(\text{SR}_{md} - \text{SR}_{nd})$		0011	000	$S_0 1^5$
$\text{PR1} += \text{ABS}(\text{BR}_{md} - \text{BR}_{nd})$		0011	001	$S_0 1^5$
$\text{PR0} += \text{SUM SR}_m$	$R_n=00000$	0011	010	$S_0 0^5$
$\text{PR0} += \text{SUM SR}_{md}$	$R_n=00001$	0011	010	$S_0 0^5$
$\text{PR0} += \text{SUM BR}_m$	$R_n=00010$	0011	010	$S_0 0^5$
$\text{PR0} += \text{SUM BR}_{md}$	$R_n=00011$	0011	010	$S_0 0^5$
$\text{PR1} += \text{SUM SR}_m$	$R_n=00100$	0011	010	$S_0 0^5$
$\text{PR1} += \text{SUM SR}_{md}$	$R_n=00101$	0011	010	$S_0 0^5$
$\text{PR1} += \text{SUM BR}_m$	$R_n=00110$	0011	010	$S_0 0^5$
$\text{PR1} += \text{SUM BR}_{md}$	$R_n=00111$	0011	010	$S_0 0^5$
$\text{PR1:0} = \text{R}_{md}$	$R_n=01000$	0011	010	00
Reserved	$R_n>01000$	0011	010	xx
$R_{sq} = \text{Permute}(\text{R}_{md}, -\text{R}_{md}, R_n)$		1100	101	00
$R_{sd} = \text{Permute}(\text{R}_{md}, R_n)$		1100	101	01

1 T1 T0 values are: fractional () – 00, integer signed (I) – 01, integer unsigned (IU) – 11.

Compute Block Instruction Format

- 2 The LSB of R_n in EXPAND & COMPACT is used to decode +/-, where: LSB = 0 determines addition, and LSB = 1 determines subtraction
- 3 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)
- 4 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 PRO/1 SUMs accumulate the results in the PR registers, which are two accumulation registers primarily used for block matching.
- 5 S0 is 0 for signed, and 1 for unsigned.

ALU Floating-Point, Arithmetic and Logical Instructions

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
$FRs(d) = FLOAT Rm$ by Rn		0100	110 T d
$Rs = FIX FRm(d)$ by Rn		0100	111 T d
$FRs(d) = CLIP Rm(d)$ by $Rn(d)$		0101	000 0 d

Table C-5. ALU Floating-Point, Arithmetic Instruction Syntax and Opcodes (Cont'd)

Syntax	(CU = 01 Float)	Type	Opcode
FRs(d) = Rm(d) COPYSIGN Rn(d)		0101	000 1 d
FRs(d) = SCALB FRm(d) 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
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

Compute Block Instruction Format

Table C-5. ALU Floating-Point, Arithmetic Instruction Syntax and Opcodes (Cont'd)

Syntax	(CU = 01 Float)	Type	Opcode
$FRs(d) = \text{MIN}(Rm(d), Rn(d))$		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.

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. Com. Logic Unit (CLU) Instr. Syntax and Opcodes

25:20	19	18:17	16:15	14:12	11	10	9:7	6:5	4:3	2	1	0
CU & type												
Op - Code												
$Rs(d)(q) = TRm(d)(q)$												
00 0000	00000			$Rs(d)(q)$			000	NLQ ¹	$TRm(d)(q)$			
$Rs(d)(q) = THRm(d)(q)$												
00 0000	00001			$Rs(d)(q)$			000	NLQ ¹	$THRm(d)(q)$			

Table C-6. Com. Logic Unit (CLU) Instr. Syntax and Opcodes (Cont'd)

25:20	19	18:17	16:15	14:12	11	10	9: 7	6:5	4:3	2	1	0
CU & type		Op - Code										
Rs = CMCTL												
00 0000	00001			Rs			00000	11111				
(For TR, THR, and CMCTL load instruction opcodes, see Table C-12 on page C-25.)												
(S)TRsd = MAX(TRmd + Rmq_h, TRnd + Rmq_l)												
00 0010	0	TRsd [4:1]	TRnd [4:1]	0	Rmq [4:2]	00	TRmd [4:1]	S ²				
(S)TRsd = MAX(TRmd - Rmq_h, TRnd - Rmq_l)												
00 0010	0	TRsd [4:1]	TRnd [4:1]	1	Rmq [4:2]	00	TRmd [4:1]	S ²				
(S)TRsd = TMAX(TRmd + Rmq_h, TRnd + Rmq_l)												
00 0010	1	TRsd [4:1]	TRnd [4:1]	0	Rmq [4:2]	00	TRmd [4:1]	S ²				
(S)TRsd = TMAX(TRmd - Rmq_h, TRnd - Rmq_l)												
00 0010	1	TRsd [4:1]	TRnd [4:1]	1	Rmq [4:2]	00	TRmd [4:1]	S ²				
(S)Rs = TMAX(TRm, TRn)												
00 0011	0000S			Rs			TRn	TRm				
(S)TRsq = ACS (TRmd, TRnd, Rm) (TMAX) ³												
00 0100	0	TRsq [4:2]	0	TRnd [4:1]	TMAX	Rm	TRmd [4:1]	S ²				
Rsq = TRaq, (S)TRsq = ACS (TRmd, TRnd, Rm) (TMAX) ³												
01 Rsq[4:2] TRAQ[4]	TRsq [4:2]	TRAQ [3:2]	TRnd [4:1]	TMAX	Rm	TRmd [4:1]	S ²					
TRs += DESPREAD (Rmq, THRmd)												
00 1000	TRs [0]	TRs [4:1]	00000			Rmq [4:2]	00	THRmd [4:1]	0			

Compute Block Instruction Format

Table C-6. Com. Logic Unit (CLU) Instr. Syntax and Opcodes (Cont'd)

25:20	19	18:17	16:15	14:12	11	10	9:7	6:5	4:3	2	1	0	
CU & type	Op - Code												
Rs = TRs, TRs = DESPREAD (Rmq, THRmd)													
00 1001	TRs [0]	TRs [4:1]		Rs		Rmq [4:2]	00	THRmd [4:1]		0			
Rsd = TRsd, TRsd = DESPREAD (Rmq, THRmd)													
00 1011	0	TRsd [4:1]		Rsd [4:1]		0	Rmq [4:2]	00	THRmd [4:1]		0		
TR31:16/15:0 = XCORRS (Rmq, THRmq)(cut#) (clr)(ext)													
00 11E0 ⁴	THL	0	0	CUT [5:4]		000	CLR	R ⁵	Rmq [4:2]	CUT [3:2]	THRmq [4:2]	CUT [1:0]	
Rsq = TRsq, TR31:16/15:0 = XCORRS (Rmq,THRmq) (cut imm) (clr) (ext)													
00 11E1 ⁴	THL	TRsq ⁶ [3:2]		CUT [5:4]		Rsq [4:2]		CLR	R ⁵	Rmq [4:2]	CUT [3:2]	THRmq [4:2]	CUT [1:0]
(For permute instruction opcodes, see Table C-4 on page C-7.)													

- 1 Defines the data size : 00 for normal (32 bits), 10 for long (64 bits) and 11 for quad (128 bits).
- 2 Bit S is 0 for short, 1 for word
- 3 Although there are more THR registers, ACS instruction refers only to THR1:0 (as on the AD-SP-TS101).
- 4 Option (ext) E bit (bit 21) is 1, else 0
- 5 R defines if the cut is by register (if 1) or by immediate (if 0). When R is set, the cut field (bits 16, 15, 6, 5, 1 and 0) is zero.
- 6 TRsq [4] = THL, THL = 0 for TR31:16 or 1 for TR15:0

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 – (S)

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

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

Compute Block Instruction Format

Table C-7 summarizes the syntax, type codes, and opcodes for multiplier instructions.

Table C-7. Multiplier Instruction Syntax and Opcodes

Syntax	Rs = ?	Type	Opcode
$R_s = R_m * R_n$		0000	x I S T y ¹
$R_{sd} = R_m * R_n$		0001	x I 1 0 y
$R_{sd} = R_{md} * R_{nd}$		0010	U I S T 0
$R_{sq} = R_{md} * R_{nd}$		0011	U I 1 0 0
$FR_s = R_m * R_n^2$		1111	1 0 0 T 0
$FR_{sd} = R_{md} * R_{nd}$		1111	1 0 0 T 1
$R_s = MR_a, MR_a += R_m * R_n$		0100	U I C R a ³
$R_{sd} = MR_a, MR_a += R_m * R_n$		0101	U I C 0 a
$R_s = MR_a, MR_a += R_m ** R_n$		1000	0 I C R a
$R_{sd} = MR_a, MR_a += R_m ** R_n$		1001	0 I C 0 a
$R_s = MR_a, MR_a += R_m ** R_n (J)$		1010	0 I C R a
$R_{sd} = MR_a, MR_a += R_m ** R_n (J)$		1011	0 I C 0 a
$R_{sd} = MR_a, MR_a += R_{md} * R_{nd}$		1100	U I C 0 a
$R_{sd} = MR_{3:0}, MR_{3:0} += R_{md} * R_{nd} (CR)$		1100	U 0 1 1 1
$R_{sd} = MR_{3:0}, MR_{3:0} += R_{md} * R_{nd} (C)$		1100	U 0 1 1 0
$R_{sd} = MR_{3:0}, MR_{3:0} += R_{md} * R_{nd}$		1100	U 0 0 1 0
$R_{sd} = MR_{3:0}, MR_{3:0} += R_{md} * R_{nd} (I)$		1100	U 1 0 1 0
$R_{sd} = MR_{3:0}, MR_{3:0} += R_{md} * R_{nd} (C)(I)$		1100	U 1 1 1 0

Table C-7. Multiplier Instruction Syntax and Opcodes (Cont'd)

Syntax	Rs = ?	Type	Opcode
MRa += Rm * Rn	Rs = 00000	1101	U I C R a
MRa -= Rm * Rn	Rs = 00001	1101	U I C R a
MRa += Rmd * Rnd	Rs = 00010	1101	U I C 0 a
MR3:0 += Rmd * Rnd	Rs = 00011	1101	U I C R 0
MRa += Rm ** Rn	Rs = 00100	1101	0 I C R a
MRa -= Rm ** Rn	Rs = 00101	1101	0 I C R a
MRa += Rm ** Rn (J)	Rs = 00110	1101	0 I C R a
MRa -= Rm ** Rn (J)	Rs = 00111	1101	0 I C R a
MRa = Rmd	Rs = 01000	1101	0 0 0 0 a
MR4 = Rm	Rs = 01100	1101	0 0 0 0 0
MRa = Rmd (se/ze) ⁴	Rs = 01001	1101	U 0 0 0 a
SMRa = Rmd (se/ze)	Rs = 01010	1101	U 0 0 0 a
LMRa = Rmd (se/ze)	Rs = 01011	1101	U 0 0 0 a

Compute Block Instruction Format

Table C-7. Multiplier Instruction Syntax and Opcodes (Cont'd)

Syntax	R _s = ?	Type	Opcode
Rsd = MRa	Rm = 00000	1110	U 0 S 0 a
SRsd = MRa	Rm = 00001	1110	U 0 S 0 a
Rsq = MR3:0	Rm = 00010	1110	U 0 S 0 0
Rs = MR4	Rm = 00011	1110	0 0 0 0 1
Rs = COMPACT MRa	Rm = 00100	1110	U I S 0 a
SRsd = COMPACT MR3:0	Rm = 00100	1110	U I S 1 0
Rsq = SMRa	Rm = 01000	1110	U 0 0 0 a
LRsq = MRa	Rm = 01001	1110	U 0 0 0 a
QRsq = LMRa	Rm = 01010	1110	U 0 0 0 a
Rsd = SMR3 ⁵	Rm = 01100	1110	0 0 0 1 1
Rsd = SMR2	Rm = 01100	1110	U 0 0 1 0
Rsd = SMR1	Rm = 01100	1110	U 0 0 0 1
Rsd = SMR0	Rm = 01100	1110	U 0 0 0 0
LRsd = MR3	Rm = 01101	1110	U 0 0 1 1
LRsd = MR2	Rm = 01101	1110	U 0 0 1 0
LRsd = MR1	Rm = 01101	1110	U 0 0 0 1
LRsd = MR0	Rm = 01101	1110	U 0 0 0 0

- 1 S bit (for saturation) is always bit 2 of op-code field. Bits 'x' and 'y' define the options (U) and (nU), as follows: xy=00 then both operands signed using syntax () , xy=01 then Rm signed and Rn unsigned using syntax (nU) , or xy=10 then both operands unsigned using syntax (U)
- 2 Floating point instructions – bit 4 of op-code is 1
- 3 The bit 'a' defines the MR pair for the instruction, as follows: a =0 then MR1:0 a= 1 then MR3:2
- 4 MR load with sign or zero extension into MR4 register – defines if zero or sign extension, and bits 1:0 of Rs define data type
- 5 In instructions that refer to a single MR register op-code bits 1:0 selects between MR3 (11), MR2 (10), MR1 (01) and MR0 (00).

Compute Block Instruction Format

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. Single, Normal Word Operands and Single Register

Syntax	Type	Opcode		Comments
$R_s = \text{LSHIFT } R_m \text{ BY } R_n$	0000	000	xx	See ¹
$R_s = \text{LSHIFT } R_m \text{ BY } \langle \text{imm6} \rangle$	0000	001	x_i5^2	
$R_s = \text{ASHIFT } R_m \text{ BY } R_n$	0000	010	xx	No options
$R_s = \text{ASHIFT } R_m \text{ BY } \langle \text{imm6} \rangle$	0000	011	x_i5^2	
$R_s = \text{ROT } R_m \text{ BY } R_n$	0001	000	xx	No options
$R_s = \text{ROT } R_m \text{ BY } \langle \text{imm6} \rangle$	0001	001	0_i5^2	
$R_s = \text{FEXT } R_m \text{ BY } R_n$	0001	010	x S0	S0=1 sign extend (SE)
$R_s = \text{FEXT } R_m \text{ BY } R_{nd}$	0001	011	x S0	
$R_s += \text{FDEP } R_m \text{ BY } R_n$	0010	000	S1 S0	S0=1 sign extend
$R_s += \text{FDEP } R_m \text{ BY } R_{nd}$	0010	001	S1 S0	S1=1 zero fill ZF
$R_s += \text{MASK } R_m \text{ BY } R_n$	0010	010	xx	No options

1 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]

2 LSB of opcode field is bit 5 of the six bits immediate.

Single Long 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 Rn (bit[0] of the instruction) is used to decode operand size, where $LSB = 0$ determines dual normal words and $LSB = 1$ determines long words*

Table C-9. Single Long or Dual Normal Word Operands and Dual Register

Syntax	Type	Opcode	
(L)Rsd = LSHIFT Rmd BY Rn	0100	000	xx
(L)Rsd = LSHIFT Rmd BY <imm7>	0100	001	i6 i5 ¹
(L)Rsd = ASHIFT Rmd BY Rn	0100	010	xx
(L)Rsd = ASHIFT Rmd BY <imm7>	0100	011	i6 i5 ¹
(L)Rsd = ROT Rmd BY Rn	0101	000	xx
(L)Rsd = ROT Rmd BY <imm7>	0101	001	i6 i5 ¹
LRsd = FEXT Rmd BY Rn	0101	010	x S0
LRsd = FEXT Rmd BY Rnd	0101	011	x S0
Rsd = GETBITS Rmq BY Rnd ²	0101	100	x S0
LRsd += FDEP Rmd BY Rn	0110	000	S1 S0
LRsd += FDEP Rmd BY Rnd	0110	001	S1 S0
LRsd += MASK Rmd BY Rnd	0110	010	xx
Rsd += PUTBITS Rmd BY Rnd ²	0110	100	xx

1 Two LSBs of opcode field is bit 6:5 of the seven bits immediate.

2 Uses standard compute instruction format shown in [Figure C-2 on page C-3](#).

Compute Block Instruction Format

Short or Bte 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 Rn (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. Short or Byte Operands and Single or Dual Registers

Syntax	Type	Opcode	S1 S0
$SRs = \text{LSHIFT } Rm \text{ BY } Rn$	1000	000	xx
$SRs = \text{LSHIFT } Rm \text{ BY } \langle \text{imm5} \rangle$	1000	001	
$SRs = \text{ASHIFT } Rm \text{ BY } Rn$	1000	010	xx
$SRs = \text{ASHIFT } Rm \text{ BY } \langle \text{imm5} \rangle$	1000	011	
$BRs = \text{LSHIFT } Rm \text{ BY } Rn$	1001	000	xx
$BRs = \text{LSHIFT } Rm \text{ BY } \langle \text{imm4} \rangle$	1001	001	
$BRs = \text{ASHIFT } Rm \text{ BY } Rn$	1001	010	xx
$BRs = \text{ASHIFT } Rm \text{ BY } \langle \text{imm4} \rangle$	1001	011	
$(S/B)Rsd = \text{LSHIFT } Rmd \text{ BY } Rn$	1010	000	xx
$(S/B)Rsd = \text{LSHIFT } Rmd \text{ BY } \langle \text{imm5} \rangle$	1010	001	

Table C-10. Short or Byte Operands and Single or Dual Registers (Cont'd)

Syntax	Type	Opcode	S1 S0
(S/B)Rsd = ASHIFT Rmd BY Rn	1010	010	xx
(S/B)Rsd = ASHIFT Rmd BY <imm5>	1010	011	

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 Rm BY Rn	1011	1100x
BITEST Rm BY <imm5>	1011	1110x
BITEST Rmd BY Rn	1011	1101x
BITEST Rmd BY <imm6>	1011	1111i5
Rs = BCLR Rm BY Rn	1011	0000x
Rs = BCLR Rm BY <imm5>	1011	0010x
Rs = BSET Rm BY Rn	1011	0100x
Rs = BSET Rm BY <imm5>	1011	0110x

Compute Block Instruction Format

Table C-11. Shifter Instruction Syntax and Opcodes
(Single Operand) (Cont'd)

Syntax	Type	Opcode
$R_s = \text{BTGL } R_m \text{ BY } R_n$	1011	1000x
$R_s = \text{BTGL } R_m \text{ BY } \langle \text{imm5} \rangle$	1011	1010x
$R_{sd} = \text{BCLR } R_{md} \text{ BY } R_n$	1011	0001x
$R_{sd} = \text{BCLR } R_{md} \text{ BY } \langle \text{imm6} \rangle$	1011	0011i5
$R_{sd} = \text{BSET } R_{md} \text{ BY } R_n$	1011	0101x
$R_{sd} = \text{BSET } R_{md} \text{ BY } \langle \text{imm6} \rangle$	1011	0111i5
$R_{sd} = \text{BTGL } R_{md} \text{ BY } R_n$	1011	1001x
$R_{sd} = \text{BTGL } R_{md} \text{ BY } \langle \text{imm6} \rangle$	1011	1011i5
$R_s = \text{LD0 } R_m^1$	1100	00000
$R_s = \text{LD0 } R_{md}^1$	1100	00010
$R_s = \text{LD1 } R_m^1$	1100	00001
$R_s = \text{LD1 } R_{md}^1$	1100	00011
$R_s = \text{EXP } R_m^1$	1100	00100
$R_s = \text{EXP } R_{md}^1$	1100	00110
$X/\text{YSTAT} = R_m^1$	1100	01000
$R_s = X/\text{YSTAT}^1$	1100	01001
$X/\text{YSTATL} = R_m^1$	1100	01110
$\text{BKFP} R_{md}, R_{nd}^1$	1100	01111
$R_{sd} = \text{BFOTMP}^1$	1100	01010
$\text{BFOTMP} = R_{md}^1$	1100	01011

1 Uses standard compute instruction format shown in [Figure C-2 on page C-3](#).

CLU Registers

Opcode encoding for the following CLU register load instructions differs from the rest of the shifter instructions and is specified by:

Bits[2:0] = 000

Bits[4:3] = NLQ; defines the data size as 00 for normal (32 bits), 10 for long (64 bits) and 11 for quad (128 bits)

Bits[9:7] = Rm(d)(q)

Bits[14:10] = Opcode

Bits[18:15] = TRs(d)(q) [4:1], THRs(d)(q) [4:1] *with [4:3] for THRx always equal to 00, or 1111*

Bit[19] = TRs(d)(q) [0], THRs(d)(q) [0], or 1

Table C-11 lists the syntax, type codes, and opcodes for CLU register load shifter instructions.

Table C-12. Shifter Instruction Syntax and Opcodes (CLU Register Load)

Syntax	Type	Opcode
TRs(d)(q) = Rm(d)(q) ¹	1111	00000
THRs(d)(q) = Rm(d)(q)(I) ^{1,2}	1111	0001I
CMCTL = Rm ¹	1111	00010

1 Uses standard compute instruction format shown in [Figure C-2 on page C-3](#).

2 I is for interleave.

IALU Instructions

IALU instruction syntax and opcodes are covered in the following sections:

- “IALU (Integer) Instruction Format” on page C-26
- “IALU Move Instruction Format” on page C-29
- “IALU Load Data Instruction Format” on page C-31
- “IALU Load/Store Instruction Format” on page C-32
- “IALU Immediate Extension Format” on page C-37

IALU (Integer) Instruction Format

The instruction format for regular IALU instructions is as shown in Figure C-4 and Table C-13.

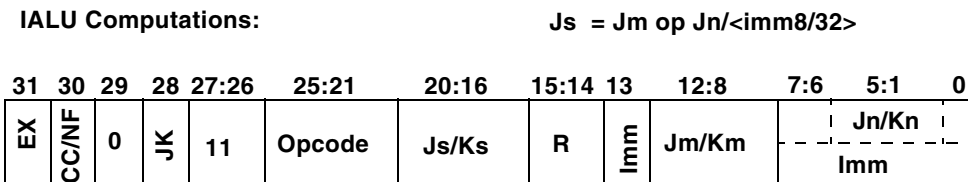


Figure C-4. IALU (Integer) Instruction Format

Table C-13. 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-14 on page C-28 .
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
29	0	Determines that this is an operation to be executed by the IALU.
30	CC/NF	For instruction lines begins with a sequencer condition instruction (if <condition>...), this bit is the CC bit. Otherwise this bit is NF bit. In the case of an instruction line that begins with conditional instruction, the NF bit is in the conditional instruction, common for all 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-40 .) If bit 30 is CC (conditional code) bit, when set, conditions the execution of the instruction by the condition instruction in the instruction line. If bit 30 is NF (no flag update) bit, when set, the current instruction does not update flags at the end of the instruction execution.
31	EX	When set, determines the instruction as the last in the instruction line.

IALU Instructions

Table C-14. Opcodes IALU (Integer) Operations

Syntax	Opcode
$J_s = J_m + J_n$	00000
$J_s = J_m - J_n$	00001
$J_s = J_m + J_n$ (CJMP)	00010
$J_s = J_m - J_n$ (CJMP)	00011
$J_s = J_m + J_n$ (CB)	00100
$J_s = J_m - J_n$ (CB)	00101
$J_s = J_m + J_n$ (BR)	00110
$J_s = J_m - J_n$ (BR)	00111
JB3:0/JL3:0 = $J_m + J_n$	01000
JB3:0/JL3:0 = $J_m - J_n$	01001
$J_s = (J_m + J_n)/2$	01010
$J_s = (j_m - j_n)/2$	01011
COMP(J_m, J_n) signed	01100
COMP(J_m, J_n) unsigned	01101
$J_s = \text{MIN}(J_m, J_n)$	01110
$J_s = \text{MAX}(J_m, J_n)$	01111
$J_s = J_m + J_n + J_c$	10000
$J_s = J_m - J_n + J_c - 1$	10001
$J_s = J_m \text{ OR } J_n$	10010
$J_s = J_m \text{ AND } J_n$	10011
$J_s = J_m \text{ XOR } J_n$	10100
$J_s = J_m \text{ AND NOT } J_n$	10101
$J_s = \text{NOT } J_m$	11000
$J_s = \text{ABS } J_m$	11001

Table C-14. Opcodes IALU (Integer) Operations (Cont'd)

Syntax	Opcode
ASHIFTR = Jm	11010
LSHIFTR = Jm	11011
ROTR = Jm	11100
ROTL = Jm	11101

IALU Move Instruction Format

The instruction format for move register instructions is as shown in [Figure C-5](#) and [Table C-15](#).

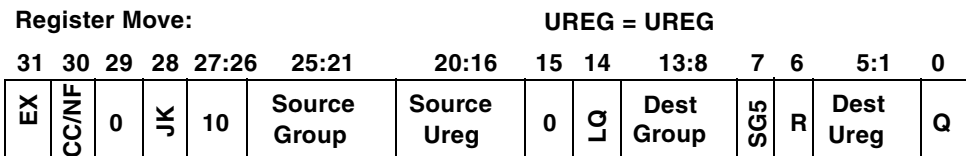


Figure C-5. IALU Move Instruction Format

Table C-15. 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 Instructions

Table C-15. 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/NF	For instruction lines begins with a sequencer condition instruction (if <condition>...), this bit is the CC bit. Otherwise this bit is NF bit. In the case of an instruction line that begins with conditional instruction, the NF bit is in the conditional instruction, common for all 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-40.) If bit 30 is CC (conditional code) bit, when set, conditions the execution of the instruction by the condition instruction in the instruction line. If bit 30 is NF (no flag update) bit, when set, the current instruction does not update flags at the end of the instruction execution.
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-6](#) and [Table C-16](#).

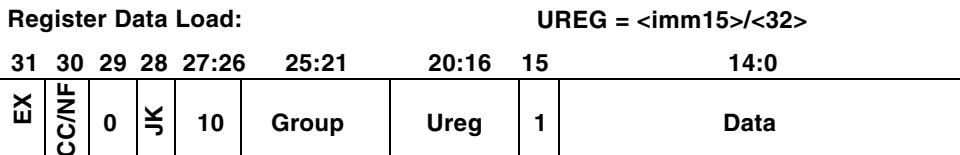


Figure C-6. IALU Load Data Instruction Format

Table C-16. 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.

IALU Instructions

Table C-16. IALU Load Data Instruction Format (Cont'd)

Bits	Field	Description
30	CC/NF	For instruction lines begins with a sequencer condition instruction (if <condition>...), this bit is the CC bit. Otherwise this bit is NF bit. In the case of an instruction line that begins with conditional instruction, the NF bit is in the conditional instruction, common for all 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-40.) If bit 30 is CC (conditional code) bit, when set, conditions the execution of the instruction by the condition instruction in the instruction line. If bit 30 is NF (no flag update) bit, when set, the current instruction does not update flags at the end of the instruction execution.
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-7](#) and [Table C-17](#).

Load with Register Update:

$$\text{UREG} = [\text{Jm} + \text{Jn}/\text{Imm}]$$

$$\text{UREG} = [\text{Jm} += \text{Jn}/\text{Imm}]$$

31	30	29	28	27	26	25:21	20:16	15	14	13	12:8	7:6	5:1	0
EX	CC/NF	0	JK	0	WR=0	Group	Ureg	MOD	LQ	IMM	Jm/Km	R	Jn/Kn	Q
												Imm		Q
												7:1		0

Figure C-7. IALU Load Instruction Format

The instruction format for store register instructions is as shown in [Figure C-8](#) and [Table C-17](#).

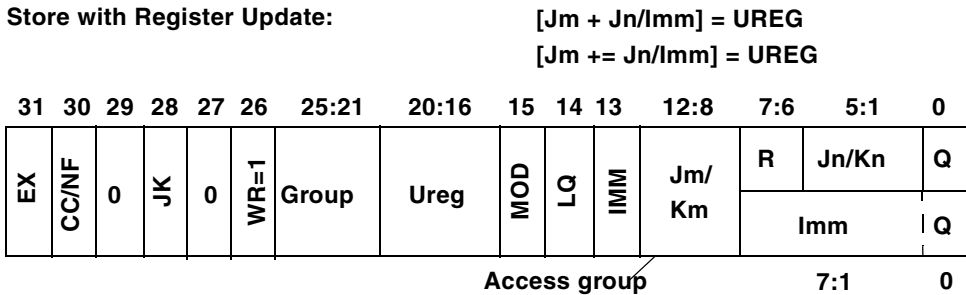


Figure C-8. IALU Store Instructions Format

Table C-17. 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. For alternate access instructions, bits 12–10 select the access group (see description in Table C-18 on page C-36).
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-35 : 1 = Indicates the data to be either long or quad 0 = Indicates the data to be word

IALU Instructions

Table C-17. IALU Load/Store Instruction Opcode Fields (Cont'd)

Bits	Field	Description
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.
30	CC/NF	For instruction lines begins with a sequencer condition instruction (if <condition>...), this bit is the CC bit. Otherwise this bit is NF bit. In the case of an instruction line that begins with conditional instruction, the NF bit is in the conditional instruction, common for all 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-40.) If bit 30 is CC (conditional code) bit, when set, conditions the execution of the instruction by the condition instruction in the instruction line. If bit 30 is NF (no flag update) bit, when set, the current instruction does not update flags at the end of the instruction execution.
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-18](#), and the assembler syntax for the different types of accesses is described in [Table C-19](#).

IALU Instructions

Table C-18. IALU Load/Store with Alternate Access (Using Circular Buffer, Bit Reversed, DAB, or SDAB) Instruction Opcode Fields

Bits	Field	Description
9–8	Jm/Km	Defines register Jm/Km between 0 and 3
12–10	Access Group	<p>Circular Buffer operations can only be used in post-modify address operations. For more information, see “Circular Buffer Addressing” on page 7-33. The access groups are:</p> <p>000 = Determines normal circular buffer (CB) access</p> <p>001 = Sets Bit Reverse option</p> <p>010 = Sets DAB at normal-word misalignment and CB access</p> <p>011 = Sets DAB at short-word misalignment and CB access</p> <p>100 = Set X DAB to shift by one word more than normal-word address misalignment, set Y DAB at normal-word misalignment, and CB access</p> <p>101 = Short DAB, and Set X DAB to shift by one short more than Y address misalignment</p> <p>110 = Set Y DAB to shift by one word more than normal-word address misalignment, set X DAB at normal-word misalignment, and CB access</p> <p>111 = Short DAB, and Set Y DAB to shift by one short more than X address misalignment</p>

Table C-19. IALU Load/Store with Alternate Access (Using Circular Buffer, Bit Reversed, DAB, or SDAB) Instruction Assembler Syntax

Option	Assembler Syntax
Circular buffer	$xyRs q = CB q[Jm += Jn]$
DAB & circular buffer	$xyRs q = DAB q[Jm += Jn]$
Short alignment DAB & circular buffer	$xyRs q = SDAB q[Jm += Jn]$
Bit reverse	$xyRs q = BR q[Jm += Jn]$

IALU Immediate Extension Format

The instruction format for IALU immediate extensions is as shown in Figure C-9, Figure C-10 and Table C-20.

Immediate Extension for IALU

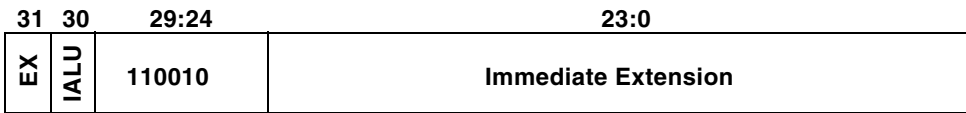


Figure C-9. IALU Immediate Extension Format

Table C-20. 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.

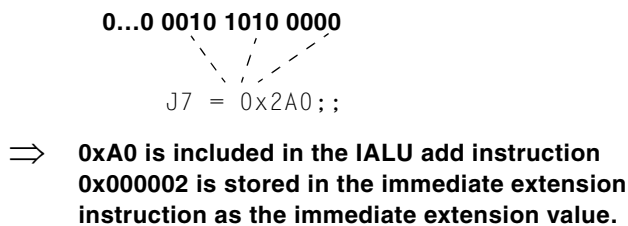


Figure C-10. Examples

Sequencer Instruction Format

Sequencer instruction syntax and opcodes are covered in the following sections:

- “Sequencer Direct Jump/Call Instruction Format” on page C-38
- “Sequencer Indirect Jump Instruction Format” on page C-40
- “Condition Codes” on page C-43
- “Sequencer Immediate Extension Format” on page C-45
- “Miscellaneous Instruction Format” on page C-46

Sequencer Direct Jump/Call Instruction Format

The instruction format for direct jump/call instructions is as shown in [Figure C-11](#) and [Table C-21](#).

Direct Jump/Call: **if cond, jump/call <label> (), (NP), (ABS)**

31	30	29:25	24	23	22	21:16	15	14:0
EX	BP	11000	RL	CL	NC	Condition	NF	Immediate

Figure C-11. Sequencer Direct Jump/Call Instruction Format

Table C-21. Sequencer Direct Jump/Call Instruction Opcode Fields

Bits	Field	Description
14:0	IMM	Specifies the immediate value.
15	NF	Specifies no flag update (for entire instruction line).
21:16	CONDI-TION	Identifies the condition for the branch—see “Condition Codes” on page C-43 .
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 8-8 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 8-8 .
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-12](#) and [Table C-22](#).

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	4:0
EX	BP	1100110	NC	Condition	ZF	INDJ	cond flag op	SCF sel	RTI	CL	Res.

Figure C-12. Sequencer Indirect Jump Instruction Format

Table C-22. Sequencer Indirect Jump Instruction Opcode Fields

Bits	Field	Description
4:0	Reserved	These must be set to 0000.
5	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
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-43 . 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>0 = compute block (X, Y or both) and SCF0 <xy>1 = compute block (X, Y or both) and SCF1

Table C-22. 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
14:13	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
15	NF	Specifies no flag update (for entire instruction line).
21:16	CONDI- TION	Identifies the condition for the branch—see “ Condition Codes ” on page C-43 .
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-43
- “IALU Conditions” on page C-44
- “Sequencer and External Conditions” on page C-45

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-23](#).

Table C-23. 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-24](#).

Table C-24. 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-25](#).

Table C-25. 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 _x _IN (where x = 0 to 3) – external flag pin input 0 to 3

Sequencer Immediate Extension Format

The instruction format for Immediate Extensions for JUMP instructions is as shown in [Figure C-13](#) and [Table C-26](#).

Immediate Extension for Jump

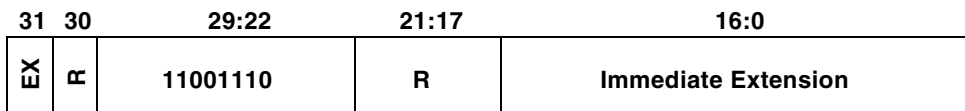


Figure C-13. Immediate Extensions (for JUMP Instruction) Format

Sequencer Instruction Format

Table C-26. Immediate Extensions for JUMP Instruction Opcode Fields

Bits	Field	Description
16:0	IMM EXT	Specifies the value of the immediate extension.
21:17	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-14](#) and [Table C-27](#).

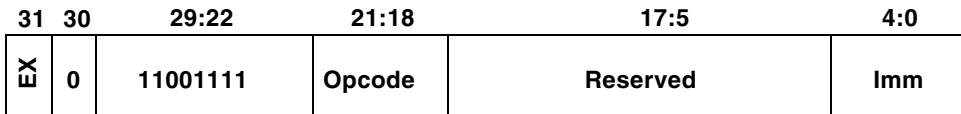


Figure C-14. Miscellaneous Instruction Format

Table C-27. Miscellaneous Instruction Opcode Fields

Bits	Field	Description
4:0	IMM	Immediate – for TRAP instruction
17:5	Reserved	Reserved
21:18	OPCODE	Determines the operation code. See Table C-28 on page C-47 .
29:22	11001111	Determines that this is an Other instruction type.
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-28](#).

Table C-28. Other Instruction Syntax and Opcodes

Instruction Syntax	Type	Opcode		Comments
NOP	11001111		0000	
IDLE	11001111		0001	
BTB Instruction	11001111		0010	A BTB instruction, where bits 4–0 are: 00000 – BTBINV (invalidate) 00010 – BTBEN (enable) 00100 – BTBDIS (disable) 01000 – BTBLOCK (start of lock) 10000 – BTBELOCK (end of lock)
TRAP (spvcmd)	11001111		0100	The “spvcmd” is an immediate indicated by bits 4:0
EMUTRAP	11001111		1000	

Sequencer Instruction Format

G GLOSSARY

These terms are important for understanding processor architecture.

Absolute Address

The optional operation of `CALL` and `JUMP` instructions—using the `(ABS)` option—is to use an absolute address. The 32-bit address in the instruction can address any location in the memory space.

Activate a Memory Page

When a memory transaction requires placing a new page in the sub-array's page buffer, the embedded DRAM system must activate (open) a page and place it in the page buffer.

ALU (Arithmetic Logic Unit)

This part of a compute block performs arithmetic and logic operations on fixed-point and floating-point data.

Base (for circular buffer)

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.

Bit Reverse Addressing

The integer ALU (IALU) provides a bit reversed address during a data move without reversing the stored address.

Block

A memory block is a 4M-bit (128K-word) unit of memory that includes a stand alone block of embedded DRAM with its associated buffers, cache, and controls. A memory block contains two half-blocks, composed of two sub-arrays each.

Cache, Cache Sets, Cache Ways, and Cache Lines

A memory block's 128K bit, 4-way set associative cache is organized as 128 cache sets with each set containing four cache ways—providing a total of 512 ways. Each cache way contains a cache line (data) that is 256 bits wide and is organized as 8 contiguous 32-bit words.

Circular Buffers

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.

CLU (Communications Logic Unit)

The processor includes a special purpose compute unit called the communications logic unit (CLU). The CLU instructions are designed to support different algorithms used for communications applications.

Cluster Multiprocessing

This is a multiprocessing system architecture in which the processor uses its link ports and external port for inter-processor communication.

Compute Block Pipe

The final pipeline stages (EX1 and EX2) are called the compute block pipe.

Compute Blocks

The TigerSHARC processor core contains two computation units called compute blocks. Each compute block contains a register file and four independent computation units—an ALU, a CLU, a multiplier, and a shifter.

Conditional Branches

These are Jump or Call/return instructions whose execution is based on testing an If condition.

Copyback Buffer

A memory block's copyback buffer contains two separate 4096-bit buffers—one for each half-block of the embedded DRAM. Each buffer accommodates sixteen 256-bit entries.

Copyback Buffer Hit

The prefetch buffer can snoop (observe) the contents of the copyback buffer. When the prefetch buffer gets data by snooping the contents of the copyback buffer, the transaction is a copyback buffer hit.

Copyback Replacement (of Cache Contents)

Regular replacement of cache contents differs from copyback replacement—replacement of a cache way that is selected for replacement, has its dirty bit set, and has to have its cache line copied back to embedded DRAM before replacement.

Data Flow Multiprocessing

This is a multiprocessor system architecture in which the processor uses its link ports for inter-processor communication.

Data Register File

This is the set of data registers that transfer data between the data buses and the computation units. These registers also provide local storage for operands and results.

Data Registers (Dreg)

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*).

Dependency Condition

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.

Direct Addressing

Direct addressing uses an index address that is set to zero and uses an immediate value for the address modifier. The index is pre-modified, and the modified address is used for the access.

DMA (Direct Memory Accessing)

The processor's I/O processor supports DMA of data between processor memory and external memory, host, or peripherals through the external and link ports. Each DMA operation transfers an entire block of data.

Double Register

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

Fetch Unit Pipe

The first four stages of the instruction pipeline (F1, F2, F3, and F4) are called the fetch unit pipe.

Half-Block

A memory half-block (64K-word) is an important subdivision of memory because each half-block of memory can support sustained sequential accesses to a single data block. Each memory half-block has its own prefetch buffer, read buffer, and copyback buffer, but each memory block has one cache. Each memory half-block contains two interleaved memory sub-arrays.

Harvard Architecture

Processors performing digital signal processing (DSP) operations require greater data throughput than Von Neumann architecture provides, so many processors use memory architectures that have separate address and data buses for program and data storage. The two sets of buses let the processor fetch a data word and an instruction simultaneously. This type of memory architecture is called Harvard architecture.

Index (for circular buffer)

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.

Indirect Addressing

Indirect addressing uses a non-zero index address and uses either a register or an immediate value for the address modifier.

Instruction Slot and Instruction Line

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 “;;”. Four instructions on an instruction line can be executed in parallel.

Instruction Types

Instruction types are descriptive names for classes of instructions.

Integer ALU Pipe

The second four instruction pipeline stages (PD, D, I, and A) are called the integer ALU pipe.

IALUs (Integer Arithmetic Logic Units)

IALUs provide memory addresses when data is transferred between memory and registers.

Interrupts

These subroutines enable a runtime event (not an instruction) to trigger the execution of the routine.

Interrupts, Masked and Unmasked

In the `IMASK` register, a “1” in an interrupt’s bit means the interrupt is unmasked (processor recognizes and services interrupt). A “0” in the bit means the interrupt is masked (processor does not recognize interrupt).

JTAG port

This port supports the IEEE standard 1149.1 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.

Jumps

Jumps transfer program flow permanently to another part of program memory.

Length (for circular buffer)

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.

Load (a register from memory)

In addition to integer operations, the IALU performs load, store, and transfer data operations. A load operation puts a value from a memory location into a register.

Loop

A loop is one sequence of instructions that executes several times with zero overhead.

Memory Blocks and Banks

The processor's internal memory is divided into blocks of internal embedded DRAM with associated buffer and cache. The processor's external memory spaces are divided into banks of SRAM and SDRAM with associated memory select lines.

Memory Transaction

A memory transaction—a read or write access to a memory block.

Modified Addressing

The integer ALU (IALU) generates an address that is incremented by a value or a register.

Modifier (for circular buffer)

The circular buffer modifier (step size between memory locations) is set by either a general-purpose IALU register or an immediate value. The modifier may not be larger than the length of the circular buffer.

Modify Register

This integer ALU (IALU) register provides the increment or step size by which an index register is pre- or post-modified during a register move.

Multiplier

This part of a compute block does floating-point and fixed-point multiplication and executes fixed-point multiply/add and multiply/subtract operations.

Multiprocessor System

This system comprises multiple processors, with or without a host processor. The processors are connected by the external bus and/or link ports.

Operands

Single, double, and quad register file registers (R_s , R_{sd} , R_{sq}) hold operands (inputs and outputs) for instructions.

Overflows

When the result overflows, the result has crossed the maximum value in which the result format can be represented.

Packed 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.

Page

A memory page is a 2K-bit (64 x 32-bit word) unit of memory.

PC-Relative Address

The default operation of `CALL` and `JUMP` instructions is to assume the address in the instruction is PC-relative. The address in the instruction is combined with the address in the program counter (PC).

Penalties (Memory Access Delays)

When a buffer/cache miss occurs, the prefetch buffer must fetch the required data from embedded DRAM. These unanticipated memory accesses incur penalties—extra clock cycles required to activate and load a page of embedded DRAM. The penalties associated with accessing embedded DRAM are variable and influenced by multiple factors.

Peripherals

Peripherals refer to everything outside the processor core. The processor's peripherals include internal memory, external port, I/O processor, JTAG port, and any external devices that connect to the processor.

Post-Modify With Update Addressing

A 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.

Precharge a Memory Page

When a memory transaction requires that an active page in the sub-array's page buffer must be replaced with a new page from the same sub-array, the embedded DRAM system must precharge (close) the active page and update page data in the sub-array from the buffer.

Precision

The precision of a floating-point number depends on the number of bits after the binary point in the storage format for the number. The processor supports two high precision floating-point formats: 32-bit IEEE single-precision floating-point (which uses 8 bits for the exponent and 24 bits for the mantissa) and a 40-bit extended-precision version of the IEEE format.

Predicted Branch and Not Predicted Branch

A 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 a not predicted branch and is indicated as such using the not predicted (`NP`) option.

Prefetch Buffer

A memory block's prefetch buffer is an 8192-bit buffer organized as four 2048-bit prefetch pages.

Prefetch Buffer Hit

The prefetch buffer also acts as a cache. When a read transaction matches one of the page tags of the four pages of the prefetch buffer and the corresponding valid bits are set, the transaction is a prefetch buffer hit, and the transaction incurs no penalties.

Pre-Modify Without Update Addressing

A 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.

Quad Register

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

Read Buffer

A memory block's read buffer is a 512-bit buffer organized as two 256-bit octal words—one for each half-block.

Read Buffer Hit

The read buffer also works as a cache—a word tag is maintained for each of the 256-bit entries. If a read transaction matches the word tag of a read buffer entry, the transaction is a read buffer hit.

Refresh Embedded DRAM Memory

The embedded DRAM system must refresh the data in every address at least once every 3.2 milliseconds to prevent data loss through process leakage.

Register Relative

Most data references are register relative, which means they allow programs to access data blocks relative to a base register.

Regular Replacement (of Cache Contents)

When there is a cache miss, it may be necessary to replace an existing cache line. Regular replacement of cache contents is the procedure of reassignment of a cache way to a different address of the embedded DRAM. Replacement occurs every time that there is a cache miss, provided there is no free (invalid) way in the cache set that is associated with the miss transaction.

Resource Conflict

A resource conflict is caused by internal bus contention or page cache miss during an internal memory access.

Resources

Resources are portions of the processor architecture that are active during an instruction.

Saturation

When saturation is enabled and the result overflows, the returned result is the extreme value that can be represented in the format of the operation following the direction of the correct result.

Shifter

This part of a compute block completes logical shifts, arithmetic shifts, bit manipulation, field deposit, and field extraction operations on 32-bit operands. Also, the shifter can derive exponents.

Single Register

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

Single-Instruction, Single-Data (SISD) and Single-Instruction, Multiple-Data (SIMD)

Compute block prefixes let 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.

Stalls (Instruction Execution Delays)

A stall is any delay that is caused by a dependency condition or a resource conflict.

Static Superscalar Architecture

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 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.

Store (a register to memory)

In addition to integer operations, the IALU performs load, store, and transfer data operations. A store operation puts a value from a register into a memory location.

Sub-Array

A memory sub-array is a 1M-bit (32K-word) unit of embedded DRAM memory that is interleaved in sets of 2K bit pages with another sub-array to form a half-block. Each sub-array contains 512 memory pages.

Subroutines

These instructions perform a specific task from another part of program memory. They are executed when the processor temporarily interrupts sequential flow of the main routine. When the task is completed, flow returns to the proper place in the main routine.

Super Harvard Architecture (SHARC)

The SHARC processors go a step farther than Harvard architecture by using a Super Harvard architecture. This architecture has program and data buses, but provides a single, unified address space for program and data storage. The data memory bus only carries data, and the program memory bus handles instructions or data.

TigerSHARC Architecture

TigerSHARC processors advance beyond SHARC architecture with a more open access bus structure shown in Figure 9-1. The TigerSHARC processor architecture has four buses, but still provides a single, unified address space for program and data storage. The J-bus and K-bus only carry data, the I-bus handles instructions, and the S-bus handles (external) accesses from external memory and I/O peripherals.

Transfer (a value from register to register)

In addition to integer operations, the IALU performs load, store, and transfer data operations. A transfer (also referred to as a move) operation copies a value from one register to another register.

Tristate Versus Three-state

Analog Devices documentation uses the term “three-state” instead of “tristate” because Tristate™ is a trademarked term, which is owned by National Semiconductor.

Universal Registers (Ureg)

The memory-mapped registers are Universal registers (*Ureg*). These registers are considered universal because *Ureg* registers may be used for many types of operations and may be used in operations outside the portion of the processor core where the *Ureg* register resides.

Von Neumann Architecture

This is the architecture used by most (non-DSP) microprocessors. This architecture uses a single address and data bus for memory access.

I INDEX

Numerics

- 16-bit address, 8-17, 8-19, 8-20, 8-97
- 16-bit short word, 2-22, 2-23
- 32-bit address, 8-17, 8-19, 8-20, 8-97
- 32-bit normal word, 2-24, 2-25
- 32-bit single-precision data, 2-17
- 40-bit extended-precision data, 2-19
- 64-bit long word, 2-25, 2-26
- 8-bit byte word, 2-21, 2-22

A

- ABS (absolute address) option
 - example, 8-20, 8-21
 - example, conditional, 8-21
 - with CJMP/CJMP_CALL, 8-99, 10-236, A-14
 - with JUMP/CALL, 8-20, 8-99, 10-234, A-14
 - with RETI/RTI, 8-99, 10-238, A-14
- ABS (absolute value, floating-point)
 - instruction, 3-25, 10-62, A-4
 - example, 10-64
 - opcode description, C-10, C-11
- ABS (absolute value, integer) instruction, 7-48,
10-211, A-11
 - example, 7-6, 10-211
 - opcode description, C-28
- ABS (absolute value) instruction, 3-1, 3-23,
10-11, A-2
 - See also* X option
 - example, 10-13
 - opcode description, C-5, C-6
- ABS (absolute value with PR accumulate)
 - instruction, 2-5, 3-20, 3-24, 10-33, A-3
 - example, 10-34
 - execution latency, 8-68
 - opcode description, C-9
- absolute, 3-1
- absolute address, 8-97
 - See also* ABS option
 - example, 8-20
 - memory mapped registers, 2-3
 - program branch, 8-20
 - .section assembler directive, 8-17
- absolute value, *See* ABS instruction
- AC (ALU carry) bit, 2-5, 3-15
 - See also* ALU status
 - updated by, 3-12
- accelerator, *See* CLU (communications logic unit)
- access
 - quad data, 1-21, 7-2, 7-4
 - restrictions, 1-44, 1-46
 - types, 7-17

INDEX

- accesses
 - aligned and unaligned, 7-27
 - broadcast, 1-44
 - DAB/SDAB usage, 7-27, 7-29
 - dependency stall, 8-69
 - embedded DRAM read/write, 9-31, 9-32
 - external memory and I/O peripherals, 8-37, 9-1, 9-53
 - memory block, 9-16, 9-17
 - memory pipeline, 9-35
 - merged, 1-44
 - normal, broadcast, and merged examples, 7-19 to 7-25
 - normal, long, and quad word, 7-48
 - penalties, 9-16, 9-37, 9-41, 9-44
 - programming guidelines, 9-46 to 9-50
 - quad word, 1-21, 1-35, 7-3
 - register relative, 1-18
 - repeated series of, 7-26
 - single cycle, dual-data, 9-4
 - stall, 8-81
- access pipe stage, 8-6, 8-7, 8-35
 - and memory pipeline, 9-37
- access time, 1-19
- accumulate, *See* multiply-accumulate
- ACK (acknowledge) pin, 1-5
- ACS (add/compare/select) instruction, 4-3, 4-4, 4-47, 10-105, 10-196, A-7
 - dependency stall, 8-66, 8-67
 - example, 10-106
 - execution latency, 8-66, 8-67
 - metrics, 4-9
 - opcode description, C-13
 - stalls, 8-79
- activate an embedded DRAM page, 9-18
- add compare select, *See* ACS instruction
- add (dual operation) instruction
 - opcode description, C-6
- add (floating-point) instruction
 - example, 10-57
- add instruction
 - and accumulate, *See* SUM instruction
 - example, 3-16, 3-17, 3-18, 10-4
 - example, conditional, 8-15
 - opcode description, C-5
- add (integer) instruction
 - bus usage, 8-81
 - dependency stall, 8-71
 - example, 7-6, 7-8, 8-17, 10-202
 - execution latency, 8-71
 - opcode description, C-28
- addition, reverse carry, 1-14
- additional literature, xxiv
- ADDR31-0 (address bus) pins, 1-5
- ADDRESS() assembler command, 8-93
- address calculation, 8-37
- address cycle in memory pipe, 8-6
- address cycle pipe stage
 - operations, 9-36
- addresses
 - 16- and 32-bit, 8-19, 8-20
 - absolute, 2-3, 8-20, 8-99, 10-234, 10-236, 10-239, A-14
 - branch target, 1-17
 - direct, 7-15, 7-35, C-34, C-36
 - direct and indirect, 10-220, 10-222, 10-223, 10-226, 10-227
 - from register, 1-16
 - immediate, 1-16, 1-47
 - indirect, 1-14, 7-34, 7-35, C-34
 - jump, 1-17
 - memory map, 9-5
 - PC-relative and absolute, 8-20, 8-97
 - quad word, 8-50
 - relative, 1-18
 - return, 1-18
 - unified address space, 9-1
- addresses, PC-relative, 8-17

- address fetch, 8-1, 8-3
 - data flow, 8-4
- address from program label, 8-17, 8-93
- address generation, 8-3
- addressing
 - 32-bit, immediate extension, 7-43
 - See also* bit reversed addressing and circular buffer addressing
 - address output, 7-15
 - base register, 7-35
 - bit reverse addressing set up, 7-38
 - bit reverse example, 7-39
 - circular buffer, 1-14
 - circular buffer, example, 7-36
 - circular buffer, setup, 7-35
 - direct, 1-16
 - immediate value for modifier, 7-15
 - index register, 7-34
 - indirect, 1-16
 - length register, 7-34
 - linear addressing with DAB accesses, 7-27
 - linear (DAB), 7-26
 - modifier, 7-9
 - modifier value, 7-34
 - modulo, 1-14
 - program label, 8-17
- ADDRESS() operator, 7-40
- address output, IALU, 7-15
- address space, unified, 9-1
- add/subtract (dual operation, floating-point) instruction
 - example, 10-95
 - opcode description, C-12
- add/subtract (dual operation) instruction
 - example, 3-17, 10-37
- add with carry (floating-point) instruction
 - opcode description, C-10
- add with carry instruction
 - example, 10-7
 - opcode description, C-6
- add with carry (integer) instruction
 - dependency stall, 8-70, 8-71, 8-72
 - example, 10-204
 - opcode description, C-28
- add with divide by two (floating-point) instruction
 - example, 10-59
- add with divide by two instruction
 - example, 10-10
 - opcode description, C-5
- add with divide by two (integer) instruction
 - example, 7-6, 10-206
 - opcode description, C-28
- AEQ (ALU equal to zero) condition, 8-9
 - See also* ALU conditions
 - condition code, C-43
 - flags set, 3-15
- AI (ALU invalid) bit, 2-5, 3-14
 - See also* shifter status
 - updated by, 3-12
- AIS (ALU invalid, sticky) bit, 2-4
 - See also* ALU status
 - #define, B-2
 - updated by, 3-12
- ALE (ALU less or equal to zero) condition, 8-9
 - condition code, C-43
- ALE (ALU less or equal to zero) condition,
 - See* ALU conditions

INDEX

- algorithms, 4-8
 - CDMA, 4-43
 - digital filters, 1-14
 - division example, 10-86
 - Fourier transforms, 1-14
 - interleaved data, 7-30
 - iterative convergence, 10-86
 - Newton Raphson iteration, 10-88
 - optimization, 2-8, 4-1
 - path search, 1-12
 - turbo code, 1-12, 4-43
 - Viterbi coding, 1-12, 4-43, 10-18, 10-19
- .ALIGN_CODE assembler directive
 - BTB usage, 8-49
 - BTB usage, example, 8-53
 - JUMP/CALL usage, 8-49
 - usage example, 8-53
- aligned DAB accesses, 7-27
- aligned data, 1-46
- aligned quad words, 1-24
 - BTB, 8-49
- alignment and size of data, 7-18
- alignment of words, 1-11
- alignment restrictions, 1-18, 7-19
- ALT (ALU less than zero) condition, 8-9
 - See also* ALU conditions
 - condition code, C-43
 - flags set, 3-15
- alternate registers, 1-18
- ALU (arithmetic logic unit), 1-2, 1-4, 3-1 to 3-26
 - conditional instructions, 8-14
 - conditions, 3-15, 7-13, 8-98, C-43
 - data flow, 3-2, 7-2
 - dependency stall, 8-66, 8-68, 8-69, 8-74, 8-75
 - examples, 3-16, 3-22
 - execution latency, 8-68, 8-69
 - instruction decode, C-5
 - instruction execution, 8-37
- ALU (arithmetic logic unit) *(continued)*
 - instruction parallelism, 1-41
 - instructions, 10-2 to 10-95
 - opcodes, C-5 to C-10
 - operations, 3-4
 - option examples, 3-7
 - options, 3-6
 - quick reference, A-2
 - registers, 2-1
 - resources, 1-37
 - results bus usage, 1-41
 - status, 3-11, 3-13, 3-16, B-2
 - status for parallel operations, 3-18
 - sticky status, 3-13, B-2
- AN (ALU negative) bit, 2-5, 3-13
 - See also* ALU status
 - BFOINC instruction, 6-13
 - updated by, 3-11
- AND instruction, 3-2
 - example, 3-17, 10-40
 - opcode description, C-6
- AND (integer) instruction
 - example, 10-214
 - opcode description, C-28
- AND NOT instruction, 3-2
 - example, 10-40
 - opcode description, C-6
- AND NOT (integer) instruction
 - example, 10-214
 - opcode description, C-28
- AOS (ALU overflow, sticky) bit, 2-4
 - See also* ALU status
 - #define, B-2
 - updated by, 3-12
- arbitration, bus, 1-5, 1-19
- architecture
 - features, 1-6
 - Static Superscalar, 1-8, 1-17
 - system, 1-6

- ARCHITECTURE () linker command,
9-54
- arithmetic logic unit, *See* ALU
- arithmetic operations, 3-1
- ALU instructions, 10-2
- CLU instructions, 10-96
- arithmetic shift
- See also* ASHIFT or ASHIFTR
- instructions
- example, 6-21
- operation, 6-7
- ASHIFT (arithmetic shift) instruction, 6-7,
6-24, 10-171, A-9
- example, 6-7, 6-21, 10-172, 10-173
- opcode description, C-20, C-21, C-22
- ASHIFTR (arithmetic shift right, integer)
instruction
- example, 10-215
- opcode description, C-29
- ASHIFTR (arithmetic shift right)
instruction, 7-49, 10-215, A-11
- assembler, 1-28, 1-38, 1-41
- ADDRESS() operator, 7-40
- .ALIGN_CODE directive, 8-49
- assembly, 1-24
- data types, 2-7
- operand size and format, 2-10
- assembly language, xxiii, A-1
- audience, intended, xxi
- AUS (ALU underflow, sticky) bit, 2-4
- See also* ALU status
- #define, B-2
- updated by, 3-12
- autoDMA registers, 1-44
- AUTODMAx (auto DMA) registers, 1-35
- AV (ALU overflow) bit, 2-5, 3-14
- See also* ALU status
- updated by, 3-11
- AVS (ALU overflow, sticky) bit, 2-4
- See also* ALU status
- #define, B-2
- updated by, 3-12
- AZ (ALU zero) bit, 2-5, 3-13
- See also* ALU status
- updated by, 3-11
- B**
- bank of memory, 9-5
- base registers, 7-35
- circular buffer, 7-5
- DAB accesses, 7-26
- base resources, 1-30
- B (byte word data) option, 2-10
- BCLR (bit clear) instruction, 6-8, 6-24,
10-188, A-10
- example, 6-8, 10-189
- opcode description, C-23, C-24
- BF (block floating-point) bit, 2-5
- See also* shifter status
- updated by, 6-19
- BFOINC (bit FIFO increment)
instruction, 3-1, 3-24, 6-12, 6-13,
10-31, 10-182, A-3
- AN bit, 6-13
- example, 10-32
- opcode description, C-9
- BFOTMP (bit FIFO temp) register, 1-13,
2-4, 2-5, 6-4, 6-12, 6-15, 10-182,
10-184, 10-194
- load, opcode description, C-24
- PUTBITS instruction, 6-5
- transfer, opcode description, C-24
- BFP6 (bit FIFO pointer) field, 6-13
- BFP (bit FIFO pointer), 10-182
- bias, sample, 3-10
- binary point, 2-16, 2-17, 2-19

INDEX

- bit
 - deposit, *See* PUTBITS instruction
 - mask, *See* MASK instruction
 - operations, 1-13, 6-4, 6-8, 6-9, 6-12
- BITEST (bit test) instruction, 6-8, 6-24, 10-187, A-10
 - example, 6-21
 - opcode description, C-23
- bit field
 - manipulation, 1-13, 6-9
 - operations, 6-1, 10-170
 - position and length, 6-9, 6-10, 6-11
- bit FIFO, 1-13
- bit FIFO increment, 3-1, 3-24, 6-12, 6-13, 10-31, 10-182, A-3, C-9
- bit FIFO operations, 6-4
- bit FIFO pointer (BFP), 3-1, 10-182
- bit manipulation, 6-1
 - operations, 6-8
- bit reverse adder, 10-201
- bit reverse addressing, 1-14, 1-47, 7-8, 7-9, 7-38, 7-48, 7-51, 10-201, A-11, A-12, A-13, C-36
 - buffer length, 7-40
 - example, 7-39, 7-41
 - overflow, 7-12
 - set up, 7-38
 - word access order, 7-39
- bit reverse carry, 7-9
- bit set, clear, toggle, and test, 1-13, 6-1
- bit wise barrel shifter, *See* shifter
- bit wise operations, 6-1
 - shifter instructions, 10-170
- BKFPT (block floating-point) instruction, 6-12, 6-25, 10-192, A-10
 - example, 10-193
 - opcode description, C-24
- block floating-point, 6-1
 - See also* BKFPT instruction
- block of memory, 9-2, 9-4, 9-5, 9-13, 9-14, 9-15
 - broadcast, 9-20
- block orientation, 9-52
- BMAX (bus master maximum) register, 1-35
- BM (bus master) condition, 8-9, 8-54, 8-98
 - condition code, C-45
- $\overline{\text{BM}}$ (bus master) pin, 1-5
- $\overline{\text{BMS}}$ (boot memory select) pin, 1-5
- $\overline{\text{BOFF}}$ (backoff) pin, 1-5
- booting, 1-21
- borrow, IALU, 7-12
- $\overline{\text{BR7-0}}$ (bus request) pins, 1-5
- branch, 1-26
 - See also* conditional branch
 - BTB and quad words, 8-49
 - CALL, 1-36, 1-38, 8-99, 10-234, A-14
 - CJMP_CALL (computed jump-call), 10-236
 - CJMP (computed jump), 1-33, 1-34, 8-99, 10-236, A-14
 - conditional, 8-5
 - cost, pipe examples, 8-56 to 8-63
 - cost summary, 8-55
 - execution flow, 8-19
 - incorrect prediction penalties, 8-47
 - in pipeline, 8-53
 - instruction placement, 1-49, 8-49
 - instructions, 8-17
 - JUMP, 10-234
 - PC relative, 1-18
 - penalty, 1-17
 - pipeline effects, 8-53
 - prediction, 1-17, 8-21
 - prediction, aligned quad words and the BTB, 8-50
 - prediction and BTB usage, 8-47
 - prediction and pipeline performance, 8-21

- branch *(continued)*
 - stalls from incorrect prediction, 8-48
 - target address, 1-17
 - unconditional, 8-21
- BR (bit reverse addressing) option, 7-38, 7-49
- BR (bit reverse operation on result) option, 7-7, 7-8, 7-9, 7-10, 10-223, 10-227
- broadcast accesses, 1-44, 7-17
- broadcast load access, 7-17
 - example, 7-20, 7-21, 7-23
 - restrictions, 1-44
- broadcast space, 9-5
- broadcast to all memory blocks, 9-20
- BRST (burst) pin, 1-5
- BSET (bit set) instruction, 6-8, 6-24, 10-188, A-10
 - example, 10-189
 - opcode description, C-23
- BTB (branch target buffer), 1-2, 1-4, 1-8, 1-15, 1-17, 8-1, 8-3, 8-6, 8-53
 - aligned quad words, 8-49
 - and branch prediction, 8-21, 8-47
 - coding techniques, 8-52
 - contents, 8-52
 - data flow, 8-4
 - hit, 8-22, 8-48
 - hit, pipe example, 8-56, 8-60, 8-61
 - internal versus external memory, 8-49
 - invalidating, 10-247
 - miss, 8-45, 8-48
 - miss, pipe example, 8-57, 8-58, 8-59, 8-62, 8-63
 - pipeline interaction, 8-6
 - register groups, 1-46
 - set, 8-44
 - structure, 8-46
 - way, 8-44
- BTBDIS (BTB disable) instruction, 8-9, 8-14, 8-99, 8-100, 10-241, 10-242, 10-244, 10-245, 10-246, 10-247, 10-248, 10-249, 10-250, A-15
 - execution latency, 8-74
- BTBELOCK (BTB end lock) instruction, 8-9, 8-14, 8-99, 8-100, 10-241, 10-242, 10-244, 10-245, 10-246, 10-247, 10-248, 10-249, 10-250, A-15
 - execution latency, 8-74
- BTB enable (BTBEN) bit, 8-13
- BTBEN (BTB enable) bit, 8-13, 8-42, B-15
- BTBEN (BTB enable) instruction, 8-9, 8-14, 8-99, 8-100, 10-241, 10-242, 10-244, 10-245, 10-246, 10-247, 10-248, 10-249, 10-250, A-15
 - execution latency, 8-74
- BTBINV (BTB invalid) instruction, 8-9, 8-14, 8-99, 8-100, 10-241, 10-242, 10-244, 10-245, 10-246, 10-247, 10-248, 10-249, 10-250, A-15, C-47
 - execution latency, 8-74
- BTBLK (BTB lock) bit, 8-13, B-15
- BTB lock (BTBLK) bit, 8-13
- BTBLOCK (BTB lock) instruction, 8-100, 10-246, A-15
- BTBLOCK (BTB start lock) instruction, 8-9, 8-14, 8-99, 10-241, 10-242, 10-244, 10-245, 10-246, 10-247, 10-248, 10-249, 10-250
 - execution latency, 8-74
- BTBLRUxx (BTB least recently used) registers, 8-45
- BTBMOD (BTB mode) bits, 8-10
- BTGL (bit toggle) instruction, 6-8, 6-24, 10-188, A-10
 - example, 6-21, 10-189
 - opcode description, C-24

INDEX

bubbles in pipeline, 8-78, 8-85, 9-38

buffer/cache, 9-2, 9-7

bus, 1-19

arbitration, 1-5, 1-22, 9-33

cacheability, 9-64

clock rate, 1-20

cluster, 1-8, 1-20

conflict penalties, 9-37

controller register groups, 1-46

data, 1-11

data flow, 9-2

deadlock, 8-81

I-, J-, K-, and S-buses, 1-19

master, 8-98

requests, 8-81

width, 1-4

bus arbitration

stall, 8-80

bus contention

pipeline effects, 8-64

buses, 9-1 to 9-81

BUSLK (bus lock) register, 1-35

BUSLOCK (bus lock interrupt) bit, 8-30

BUSLOCK (bus lock) pin, 1-5

bus masters, 2-3

byte word, 1-11, 2-21, 2-22, 6-3

32-, 64-, or 128-bit compaction, 10-46

8- or 16-bit expansion, 10-41

8- or 16-bit transpose, 10-50

add example, 3-18

data, 2-10, 2-13, 2-14, 2-20

multiplier, 5-4

shifter result, 6-5

sideways sum example, 3-19

C

cache, 1-1, 1-3, 9-3, 9-7, 9-11, 9-28

architecture, 9-25

commands, 9-58

copyback, 9-28, 9-74

cache

(continued)

dirty bit, 9-25

disable, 9-50, 9-63

enable, 9-50, 9-62

enable, example, 9-54

enable cache/bus ability, example, 9-54

execution latency, 8-69

hit, 9-12, 9-28, 9-31, 9-45

initialize, 9-70

instruction caching, 9-51

invalidate, 9-78

line, 9-11, 9-13, 9-25

lock, 9-29

lock bit, 9-25

lock end, 9-69

locking data or code into, 9-29

lock start, 9-68

LRU bits, 9-25

miss, 9-28, 9-39, 9-40, 9-44, 9-45

miss transactions, 9-38

operation, 9-25, 9-27

penalty-free transactions, 9-44

pipeline effects, 8-64

programming guidelines, 9-46

regular and copyback replacement, 9-28

replacement policy, 9-11, 9-28, 9-29,
9-66

set, 9-11, 9-13, 9-25, 9-26, 9-27

size, 9-13

tag bits, 9-25

thrashing, 9-47

valid bits, 9-25, 9-28

way, 9-11, 9-25, 9-26

write policy, 9-11

cache address/index (CCAIRx) registers,
9-19

cache command aborted

(CASTAT_COM_ABRTD) bit, 9-22

- cache command active
 - (CASTAT_COM_ACTIVE) bit, [9-22](#)
- cache command (CACMDx) registers, [9-19](#)
 - command opcodes, [9-24](#)
- cache enable (CASTAT_ENBL) bit, [9-23](#)
- CACHE_INIT (cache initialize)
 - command, [9-57](#), [9-70](#), [A-17](#)
 - example, [9-73](#)
- CACHE_INIT_LOCK (cache initialize and lock) command, [9-57](#), [9-70](#), [A-17](#)
 - example, [9-73](#)
- cache lock mode (CASTAT_LOCK) bit, [9-23](#)
- cache status (CASTATx) registers, [9-19](#)
- CACMD_CB (cache copyback) command, [9-57](#), [9-74](#), [A-17](#)
 - CCAIRx register operation, [9-21](#)
 - example, [9-77](#)
 - with local access policy, [9-49](#)
- CACMD_DIS (cache disable) command, [9-56](#), [9-63](#), [9-67](#), [A-16](#), [B-19](#)
 - example, [9-63](#)
- CACMD_ELOCK (cache end lock)
 - command, [9-56](#), [9-69](#), [A-16](#)
 - example, [9-69](#)
- CACMD_EN (cache enable) command, [9-56](#), [9-62](#), [A-16](#)
 - example, [9-62](#)
- CACMD_INIT (cache initialize)
 - command
 - CCAIRx register operation, [9-21](#)
- CACMD_INV (cache invalidate)
 - command
 - with local access policy, [9-50](#)
- CACMD_INV (cache invalid) command, [9-57](#), [9-78](#), [A-17](#)
 - CCAIRx register operation, [9-21](#)
 - example, [9-81](#)
- CACMD_REFRESH (refresh rate)
 - command, [9-56](#), [9-60](#), [A-16](#)
 - example, [9-54](#), [9-61](#)
- CACMD_SET_BUS (set bus/cacheability)
 - command, [9-56](#), [9-64](#), [A-16](#)
 - example, [9-54](#)
- CACMD_SLOCK (cache start lock)
 - command, [9-56](#), [9-68](#), [A-16](#)
 - example, [9-68](#)
- CACMDx (cache command) registers, [9-19](#)
 - dependency stall, [8-69](#)
 - load execution latency, [8-69](#)
 - load/transfer restrictions, [9-20](#)
 - write latency, [9-20](#)
- CADATAx (cache data) registers
 - dependency stall, [8-69](#)
 - load execution latency, [8-69](#)
- call
 - example, [8-21](#)
 - nested and non-nested, [1-18](#)
- CALL instruction, [1-36](#), [1-38](#), [8-19](#), [8-99](#), [10-234](#), [A-14](#)
 - eliminating branch cost, [8-49](#)
 - example, [8-21](#)
- carry, ALU, [3-12](#)
- carry, IALU, [7-11](#), [7-12](#)
- CAS (column address select) pin, [1-5](#)
- CASTAT (cache status) register
 - set cacheability, [9-64](#)
- CASTAT_COM_ABORTD (cache command aborted) bit, [9-22](#)
- CASTAT_COM_ACTIVE (cache command active) bit, [9-22](#)
- CASTAT_ENBL (cache enable) bit, [9-23](#)
- CASTAT_I_CACHING (I-bus cacheability) bits, [9-64](#)
- CASTAT_J_CACHING (J-bus cacheability) bits, [9-64](#)

INDEX

- CASTAT_K_CACHING (K-bus cacheability) bits, [9-64](#)
- CASTAT_LOCK (cache lock mode) bit, [9-23](#)
- CASTAT_REF_ACTIVE (cache refresh active status) bit, [9-22](#)
- CASTAT_REFCNTR (refresh rate) bits, [9-23](#)
- CASTAT_S_CACHING (S-bus cacheability) bits, [9-64](#)
- CASTAT_STL_ACTIVE (cache stall mode active status) bit, [9-22](#)
- CASTATx (cache status) registers, [9-19](#), [9-22](#), [9-23](#)
 - dependency stall, [8-69](#)
 - write latency, [9-20](#)
- CB (circular buffer addressing) option, [1-47](#), [7-7](#), [7-8](#), [7-9](#), [7-10](#), [7-33](#), [7-49](#)
- CB (circular buffer operation on result) option, [10-223](#), [10-227](#)
- CCAIRx (cache address/index) registers, [9-19](#)
 - dependency stall, [8-69](#)
 - load execution latency, [8-69](#)
 - load/transfer restrictions, [9-20](#)
 - write latency, [9-20](#)
- C calls
 - context save/restore, [7-3](#)
- C (clear MR register) option, [5-11](#), [5-17](#)
- CCLK (processor core clock), [8-34](#)
 - and embedded DRAM, [9-18](#)
- CCNTx (cycle counter) registers
 - load/transfer execution latency, [8-76](#), [10-220](#), [10-230](#)
- CDMA algorithm, [4-22](#), [4-43](#), [10-101](#)
 - cross correlations, [4-3](#)
 - despreader, [1-12](#), [4-3](#)
- channels, multiple data, [2-8](#)
- check register dependencies, [1-16](#), [1-18](#)
- circular buffer, [7-33](#)
 - overflow, [1-14](#), [7-9](#)
 - registers, [B-12](#)
 - setup, [7-35](#)
 - wrap, [7-9](#)
- circular buffer addressing, [1-14](#), [1-47](#), [7-8](#), [7-33](#), [7-48](#), [7-51](#), [10-201](#), [A-11](#), [A-12](#), [A-13](#), [C-36](#)
 - base registers, [7-5](#), [7-35](#)
 - DAB, [7-26](#)
 - example, [7-36](#)
 - index registers, [7-5](#), [7-34](#)
 - index value, SDAB, [7-30](#)
 - length registers, [7-5](#), [7-34](#)
 - load Dreg register, dependency stall, [8-71](#)
 - modifier register, [7-5](#)
 - modifier value, SDAB, [7-29](#)
 - register operations, [7-5](#)
 - registers, [7-26](#)
 - setup, [7-9](#)
 - using an immediate extension, [7-44](#)
 - word access order, [7-37](#)
 - wrap around, [7-33](#)
- CJMP_CALL (computed jump/call) instruction, [8-19](#), [8-20](#), [10-236](#)
 - example, [8-93](#), [10-237](#)
- CJMP (computed jump) instruction, [1-17](#), [8-5](#), [8-19](#), [8-99](#), [10-236](#), [A-14](#)
 - example, [10-237](#)
- CJMP (computed jump) option, [8-20](#)
- CJMP (computed jump) register, [1-33](#), [1-34](#), [1-35](#), [1-38](#), [7-5](#), [7-10](#), [7-48](#), [8-19](#), [8-20](#), [10-201](#), [A-11](#), [B-13](#)
 - BTB usage, [8-53](#)
 - load execution latency, [8-76](#)
 - usage, [10-201](#)
- CJMP (load CJMP register from result) option, [7-7](#), [7-8](#), [7-9](#), [7-10](#)

- clear, clear and round, *See* C and CR
 - options
- clear bit, 1-13
 - See also* BCLR instruction
- CLIP instruction, 3-1, 3-23, 3-25, 10-25, 10-77, A-3, A-5
 - example, 10-25, 10-78
 - opcode description, C-6, C-10
- clipping, 3-1
- clock rate, bus, 1-20
- close an embedded DRAM page, 9-18
- CLR (clear trellis register) option, 4-30, 4-41
- CLU (communications logic unit), 1-2, 1-4, 1-12, 4-1 to 4-47
 - conditional instructions, 8-14
 - data flow, 4-2
 - data types and sizes, 4-3, 4-4
 - dependency stall, 8-66, 8-67, 8-68, 8-69, 8-74, 8-75
 - examples, 4-43
 - exceptions, 8-80
 - execution, 8-37
 - execution latency, 8-68, 8-69
 - instruction dependency, 8-78
 - instruction parallelism, 1-41
 - instructions, 10-96 to 10-106, C-10
 - instruction summary, 4-46
 - operations, 4-4
 - register load restrictions, 1-42
 - registers, 2-1
 - resources, 1-36, 1-37
 - results bus usage, 1-41
 - status, 4-42, 4-43
- cluster bus, 1-8, 1-20
- cluster multiprocessing, 1-22, G-2
- CMCTL (communications control)
 - register, 2-4, 2-5, 4-4, 10-104
 - load, execution latency, 8-67
- CMCTL (communications control)
 - register *(continued)*
 - load, opcode description, C-25
 - transfer, opcode description, C-13
- codes, turbo and parity-check, 4-2
- combination constraints, *See* restrictions
- comma delimiters, 1-23
- comma in syntax, 1-25
- communications control (CMCTL)
 - register, *See* CMCTL
 - (communications control) register
- COMPACT instruction, 3-1, 3-10, 3-24, 5-16, 5-28, 10-46, 10-165, A-4, A-9
 - example, 10-48, 10-49
 - opcode description, C-8, C-18
- COMPACT (MR register) instruction
 - example, 10-167, 10-168, 10-169
- comparison, 3-1
- COMP (compare, integer) instruction
 - example, 7-8, 10-208
 - opcode description, C-28
- COMP (compare) instruction, 3-1, 3-23, 7-48, 10-23, 10-207, A-3, A-11
 - example, 10-24
 - opcode description, C-6
- compiler, 1-27
- complement sign (floating-point)
 - instruction, 10-65
 - example, 10-66
 - opcode description, C-11
- complex conjugate operation
 - definition, 5-4
- complex correlation, 4-1
- complex data, 4-3
- complex data type, 2-7
- complex multiply-accumulate
 - definition, 5-4
 - example, 5-25
 - operations, 5-1, 5-10
 - throughput, 4-22

INDEX

- complex multiply-accumulate instruction
 - opcode description, [C-17](#)
- complex multiply-accumulate operations
 - throughput, [4-3](#)
- `+= **` complex multiply-accumulate
 - operator, [5-4](#), [5-10](#), [5-19](#)
- complex multiply-accumulate (short word) instruction
 - example, [10-141](#), [10-142](#)
- complex multiply-accumulate with transfer
 - MR register (dual operation, short word) instruction
 - example, [10-145](#), [10-146](#), [10-147](#), [10-148](#)
- complex multiply operations
 - multiplier instructions, [10-107](#)
- complex numbers, [1-12](#), [2-7](#), [5-4](#)
- COM port, McBSP, *See* link port
- compress, *See* COMPACT instruction
- compute block, [1-4](#), [1-9](#), [1-10](#), [1-19](#)
 - ALU, [3-1](#)
 - arithmetic instruction latency, [7-3](#)
 - CLU, [4-1](#)
 - conditions, [8-9](#), [8-98](#), [C-43](#)
 - data flow, [8-3](#), [8-4](#)
 - dependencies, [8-77](#)
 - execution, [2-1](#)
 - instruction parallelism, [1-41](#)
 - multiplier, [5-1](#)
 - opcodes, [C-37](#)
 - pipeline, [1-16](#), [8-37](#)
 - registers, [2-1](#) to [2-15](#), [B-5](#), [B-6](#), [B-7](#)
 - resources, [1-36](#), [1-37](#)
 - results bus usage, [1-41](#)
 - selection, [2-6](#), [2-7](#), [2-13](#)
 - shifter, [6-1](#)
 - status, [2-4](#), [8-98](#)
 - usage, [1-41](#)
 - compute block pipe, [8-6](#), [8-35](#)
 - computed jump/call, [10-236](#)
 - computed jump (CJMP) register, [7-5](#)
 - compute instructions, [2-1](#)
 - conditional branch, [8-5](#)
 - CALL, [1-36](#), [1-38](#), [8-99](#), [10-234](#), [A-14](#)
 - CJMP_CALL (computed jump/call), [10-236](#)
 - CJMP (computed jump), [1-33](#), [1-34](#), [8-99](#), [10-236](#), [A-14](#)
 - effects on pipeline, [8-53](#)
 - JUMP, [10-234](#)
 - NP (not predicted) branch, [8-21](#), [8-48](#), [8-59](#), [8-99](#), [A-14](#)
 - conditional execution, [1-27](#), [8-14](#)
 - IF...DO instruction, [1-27](#), [5-22](#), [7-13](#), [8-14](#), [8-99](#), [10-241](#), [A-14](#)
 - IF...ELSE instruction, [1-27](#), [8-15](#), [8-100](#), [10-242](#), [A-15](#)
 - conditional instructions, [1-27](#), [3-15](#), [5-22](#), [6-19](#), [7-13](#), [8-9](#), [8-14](#), [8-15](#)
 - ALU condition dependency stall, [8-68](#)
 - compute or IALU, [8-14](#)
 - conditional branch dependency stall, [8-76](#)
 - conditions summary, [8-98](#)
 - IALU condition dependency stall, [8-70](#), [8-71](#), [8-72](#)
 - loop condition dependency stall, [8-74](#)
 - RTI dependency stall, [8-75](#), [8-76](#)
 - static flag condition dependency stall, [8-72](#), [8-73](#), [8-74](#)
 - conditional sequencing
 - IF...ELSE instruction, [10-242](#)
 - condition codes, [C-43](#)
 - pipeline stages, [8-54](#)
 - condition flags, [8-54](#)
 - SFx and ISFx, [10-243](#)

- conditions
 - ALU, 3-15, 7-13
 - IALU, C-44
 - multiplier, 5-22
 - negated, 8-9, 8-98
 - sequencer, C-45
 - shifter, 6-19
 - TRUE, 8-21
 - configurations, multiprocessing, 1-22
 - conflicts, data, 1-27
 - conjugate, J (complex conjugate) option, 5-19
 - conjugate operation
 - definition, 5-4
 - constraints, *See* restrictions
 - context save/restore, 7-3
 - context switching, 1-18
 - control flow, *See* sequencer instructions
 - CONTROLIMP1-0 (control impedance)
 - pins, 1-5
 - control program flow, 8-1
 - control registers, 1-20
 - conventions, xxxii
 - instruction notation, 1-24
 - conversion
 - 32- to 40-bit floating-point, 3-3, 3-25, 10-73, A-5, C-11
 - 40- to 32-bit floating-point, 3-3, 3-25, 10-75, A-5, C-11
 - fixed- to floating-point, 3-2, 3-25, 6-4, 10-71, A-5, C-10, C-11
 - floating- to fixed-point, 3-2, 3-25, 10-69, A-5, C-10, C-11
 - copyback buffer, 9-3, 9-7, 9-11, 9-17
 - hit, 9-12
 - overflow, 9-45
 - set cacheability, 9-66
 - copyback replacement, 9-28
 - cache, 9-28
 - copy sign, 3-2
 - COPYSIGN instruction, 3-2, 3-26, 10-79, A-5
 - example, 10-80
 - opcode description, C-11
 - core clock (CCLK), 8-6, 9-18, 9-40
 - and embedded DRAM, 9-18
 - correlation sequences, 4-30
 - counter, loop
 - decrement, 8-22
 - updates, 1-38
 - counting ones, 10-29
 - CPA (core priority access) pin, 1-5
 - CR (clear MR register and round result)
 - option, 5-11, 5-18
 - operation, 5-17
 - restrictions, 1-42
 - crossbar configuration, 1-22
 - crossbar connect, 9-2, 9-7
 - cross correlations
 - See also* XCORRS instruction
 - CDMA, 4-3
 - customer support, xxiv
 - CUT (cut from XCORRS set) option, 2-5, 4-4, 4-30, 4-41
 - operation, 4-36
- ## D
- DAB (DAB operation) option, 1-31, 1-44, 1-47, 2-1, 7-19, 7-26, 7-50, 10-223, 10-225, A-12, C-35
 - DAB (data alignment buffer), 1-2, 1-21, 1-31, 1-35, 1-44, 1-47, 2-1, 7-19, 7-29, 7-50, 8-3, A-12, C-35
 - accesses, 1-21, 7-26, C-36
 - accesses with offset, 7-30
 - instructions, 7-26
 - linear addressing setup, 7-27
 - load Dreg register, dependency stall, 8-69
 - load restrictions, 1-46

INDEX

- DAB (data alignment buffer) *(continued)*
 - operation, 7-27, 7-49
 - priming, 7-26
 - resources, 1-35
- data
 - See also* immediate value
 - 32-, 64-, or 128-bit compaction, 10-46
 - 32-bit immediate, 7-43
 - 8- or 16-bit expansion, 10-41
 - 8- or 16-bit transpose, 10-50
 - access, 9-18, 9-30, 9-32
 - accesses, 2-1
 - alignment, 1-46, 2-1, 7-18
 - byte word, 2-13, 2-14
 - complex, 4-3
 - DAB accesses, 7-27
 - dependency, 1-27
 - extended word, 2-15
 - fixed/floating-point, 2-13
 - format, 2-7
 - fractional/integer, 5-13
 - immediate, 1-16, 1-26
 - interleaved, 7-30, 9-48, 9-52
 - locking into the cache, 9-29
 - long word, 2-16
 - normal word, 2-14, 2-15
 - placing in memory, 9-2
 - real or imaginary, 7-3
 - short word, 2-14, 2-15
 - signed/unsigned, 5-12, 7-8
 - single/extended-precision, 5-3
 - sizes, 2-9, 7-18
 - types, IEEE, 2-18
- data, cache, 9-11
- data, fixed- and floating-point, G-1
- data, real, 2-7
- DATA63-0 (data bus) pins, 1-5
- data access, 1-21
 - dual-data access, 9-4
- data address generation, 8-3
- data addressing, 7-1
- data addressing and movement operations, 10-218
- data alignment buffer, *See* DAB
- data caching, 9-67
- data compaction operations, 5-1, 5-3
 - multiplier instructions, 10-107
- data conversion operations, 3-1
 - ALU instructions, 10-2
 - shifter instructions, 10-170
- data cycle pipe stage, 8-6
 - operations, 9-36
- data (Dreg) registers, 1-20, 2-3, B-5, B-6, B-7, B-10, B-11
 - load, 7-49
 - register file, 2-6
 - store, 7-51
- data flow, 1-19, 1-40, 2-2, 3-2, 4-2, 5-2, 6-2, 7-2, 8-3
 - embedded DRAM, 9-5
 - embedded DRAM read/write, 9-17
 - instruction and memory pipelines, 9-35
 - memory and buses, 9-2
 - multiplier operations, 5-8, 5-9
- data flow multiprocessing, G-3
- data formats, 1-25, 2-1, 2-17
- data move, 7-1
- data size, 3-4, 4-4
- data structures, 1-14
- data types, 1-11, 1-12, 2-17, 4-5, 5-3
 - assembly, 2-7
 - CLU, 4-3
 - fixed-point, 3-3
 - floating-point, 3-4
 - IALU, 7-8
 - saturation, 5-14
 - shifter, 6-3
- DBGE (debug enable) register, 1-35, 10-249, B-13
- DBGEN (debug enable) bit, B-15

- DCDx (DMA destination TCB) registers, 1-35
- DCLK (embedded DRAM clock), 9-18
- DCNT (DMA control) register, 1-35
- DCSx (DMA source TCB) registers, 1-35
- deadlock, 1-28, 8-81
- debug enable, B-15
- debug register groups, 1-35, 1-46
- DEC (decrement) instruction, 3-1, 3-23, 10-21, A-3
 - example, 10-22
 - opcode description, C-6
- decode pipeline stage, 8-6, 8-7, 8-35, 8-36
- predicted/not predicted branch check, 8-47
- decrement, 3-1
 - See also* DEC instruction
- decrement loop counter, 8-22
- delay lines, 1-14
- delays, correlation, 4-30
- delimiters, instruction slot and line, 1-23
- denormal operands, 2-18
- dependency
 - check, 1-16, 1-18, 8-37
 - compute block, 8-77, 8-83
 - condition, 8-64
 - delays, IALU instructions, 7-3
 - IALU, 8-84, 8-85
 - load, 8-83, 8-84
 - pipeline effects, 8-64
 - resource and pipeline, 8-64
- deposit
 - bits, *See* PUTBITS instruction
 - field, *See* FDEP instruction
- despreader
 - CDMA, 4-3
 - throughput, 4-3
- DESPREAD instruction, 1-12, 2-5, 4-3, 4-5, 4-22, 4-30, 4-43, 4-47, 10-101, 10-196, A-6
 - data flow example, 4-23, 4-24, 4-26, 4-27, 4-28, 4-29
 - dependency stall, 8-66, 8-67
 - execution latency, 8-67
 - function, 4-22
 - opcode description, C-13
- DESPREAD (with TR register transfer, CLU) instruction, 10-101
- destination resources, 1-30
- development enhancements, 1-6
- digital filters, 1-14
- direct addressing, 1-14, 1-38, 7-14, 7-15, 7-34, 7-35, 7-38, 10-220, 10-222, 10-223, 10-226, 10-227, C-34, C-36
- direct calls, 1-16
- direct jumps, 1-16
- dirty bit, cache, 9-25, 9-26
- displace field, memory transaction, 9-27
- division
 - example, 7-6
 - opcode description, C-5
- division (floating-point)
 - example, 10-86
- division seed, 3-2
- DMA13-0 (DMA interrupt) bits, 8-31
- DMA (direct memory access), 1-19, G-4
 - channels, 1-17
 - controllers, 1-1
 - programming guidelines, 9-53
 - register groups, 1-35
 - registers, 1-44
- DMAR3-0 (DMA request) pins, 1-5
- DO instruction, 1-27, 5-22, 7-13, 8-99, A-14
- double (Rmd) registers, 1-10, 1-11, 1-25, 2-9, 2-20, 5-6, 5-7
 - selection, 2-13

INDEX

double sequence flow, 9-52
DPA (DMA priority access) pin, 1-5
DRAM, embedded
 See embedded DRAM
Dreg (data) registers, 1-33, 2-3, 2-5, 7-18
 load, 7-49
 load execution latency, 8-68, 8-69
 store, 7-51
DS2-0 (drive strength) pins, 1-5
DSP
 architecture, 1-7
DSTAT (DMA status) register, 1-35
dual aligned registers, 7-19
 See also double (Rmd) registers
dual-data, single-cycle accesses, 9-4
dual operation
 add and subtract, fixed-point, 10-37
 add and subtract, floating-point, 10-94
dual words, 1-15
.D unit, *See* IALU
D unit, *See* IALU
dynamic restrictions, 1-28
dynamic violations
 See also restrictions

E

edge sensitive interrupts, 8-27, B-38
ELSE instruction, 1-27, 8-15, 8-100, A-15
embedded DRAM, 1-1, 9-1, 9-42
 activate a page, 9-18
 block orientation, 9-52
 blocks and banks, 9-5
 buffer/cache hit, 9-12
 cache, 9-13, 9-50
 close a page, 9-18
 copyback buffer, 9-11, 9-17
 cross page prefetch, 9-42
 data flow, 9-5
 double sequence flow, 9-52
 instruction caching, 9-51

embedded DRAM *(continued)*
 interface, 9-3
 local access policy, 9-46, 9-47, 9-49
 LRU page replacement, 9-42
 open a page, 9-18
 out-of-order accesses, 9-41
 page, 9-16
 page buffer, 9-15, 9-17
 page prefetch, 9-42
 penalty-free transactions, 9-44
 precharge a page, 9-18
 prefetch buffer, 9-8, 9-12, 9-17
 prefetch sequence, 9-43
 read access operation, 9-31
 read buffer, 9-10
 read data flow, 9-17
 refresh rate, 9-19, 9-54, 9-60
 sequential access policy, 9-46, 9-47, 9-48
 sub-array, 9-17, 9-18
 write access operation, 9-32
 write data flow, 9-17
embedded DRAM clock (DCLK), 9-18, 9-40
EMCAUSE (emulation cause) bit, 8-11
EMUCTL (emulation control) register, 8-13
 dependency stall, 8-75
 load execution latency, 8-75
EMUIR (emulator instruction) register, 10-249
emulation, 1-22, 8-11
 memory access penalty analysis, 9-45
emulation cause (EMCAUSE) bit, 8-11
emulation control (EMUCTL) register, 8-13
emulator instruction (EMUIR) register, 10-249
emulator trap (EMUTRAP) instruction, *See* EMUTRAP (emulator trap) instruction

- EMUL (emulation interrupt, pointer) bit, 8-10
- EMUTRAP (emulator trap) instruction, 8-9, 8-14, 8-99, 10-241, 10-242, 10-244, 10-245, 10-246, 10-247, 10-248, 10-249, 10-250, C-47
- EPROM booting, 1-21
- error correcting codes, 4-6
- evaluation of expressions, 7-40
- EXCAUSE (exception cause) bits, 8-11
- EXCEPT (exception pointer) bits, 8-10, 8-13
- exception pointer (EXCEPT) bits, 8-13
- exceptions, 8-10, 8-12
 - arithmetic, 1-17
 - IALU Ureg transfers, 10-230
 - memory system, 9-72, 9-76, 9-80
 - pipeline effects, 8-90
 - pipeline example, 8-86
 - software, 2-4
 - underflow, 2-18
- execute 1,2 pipe stages, 8-6, 8-7, 8-8, 8-35
- and memory pipeline, 9-37
- compute block condition evaluation, 8-54
- execution, 8-34
 - branching, 8-19
 - compute, 2-1
 - compute block instructions, 8-37
 - compute instructions, 8-8
 - flow and branch cost, 8-55
 - IALU instructions, 8-36
 - interrupts, 8-26
 - loop, 8-22
 - management, 8-1
 - memory access instructions, 8-8
 - operand size, 2-11
 - optimization, 2-8, 8-5
 - overlaps, 8-38
 - parallel, 1-26, 8-37
- execution *(continued)*
 - predicated, 1-27
 - rate, 1-16
 - throughput, 8-6
 - time, 1-19
- execution latency, 8-65
- execution status, 8-100, 10-243, A-15
 - ALU, 3-16, B-2, B-4
 - IALU, 7-14
 - multiplier, 5-23, B-2, B-3
 - shifter, 6-20, B-2
- EXPAND instruction, 1-36, 1-37, 3-1, 3-10, 3-24, 10-41, A-4
 - example, 10-42, 10-43, 10-44, 10-45
 - opcode description, C-7
- EXP (exponent extract) instruction, 6-12, 6-24, 10-191, A-10
 - example, 10-191
 - opcode description, C-24
- exponent, 2-16, 2-17, 2-19, 3-3, 5-8, 5-9
- exponent, extract, *See* EXP instruction
- expressions evaluation, 7-40
- EXTD (extend to double, floating-point) instruction, 3-3, 3-25, 10-73, A-5
 - example, 10-74
 - opcode description, C-11
- extended output range, ABS, *See* X option
- extended-precision data, 2-16, 2-19, 5-3
- extended word, 1-11
 - add example, 3-17
 - conversion, 10-73
 - data, 2-15
- external memory
 - access delay, 8-37
 - accesses, 9-1
 - BTB, 8-49
 - fetch, 8-36
 - instruction fetch, 8-40
- external memory space, 9-5

INDEX

external port, 1-3, 1-4, 1-8, 1-19

register groups, 1-35

registers, 1-44

resources, 1-33, 1-34

EXT (extended-precision for XCORRS results) option, 4-30, 4-41

extract

See also EXP, FEXT, MANT, ONES, and LDx instructions

exponent, 6-1, 10-191

field, *See* FEXT instruction

fields, 10-176, 10-182

leading ones/zeros, 10-190

leading zeros, 6-24, 10-190, A-10

mantissa, 10-90

extract leading (LDx) ones/zeros instruction

example, 10-190

extract words from MR register instruction

example, 10-161, 10-162, 10-163, 10-164

EZ-ICE emulator, 1-22

F

factor, scaling block floating-point, 10-192

FCOMP (compare, floating-point)

instruction, 3-25, 10-67, A-5

example, 10-68

opcode description, C-12

FDEP (field deposit) instruction, 1-36,

1-37, 6-3, 6-9, 6-24, 10-178, A-9

bit field length and position, 6-9

example, 6-11, 10-179

opcode description, C-20, C-21

options example, 6-17

restrictions, 1-41

fetch, 8-35

fetch 1,2,3,4 pipe stages, 8-6, 8-35

and memory pipeline, 9-36

fetch unit pipe, 8-6, 8-35

FEXT (field extract) instruction, 6-9, 6-24, 10-176, A-9

bit field length and position, 6-9

example, 6-10, 10-177

opcode description, C-20, C-21

options example, 6-17

F (floating-point data) option, 2-11

FFT algorithms, 1-14

field

See also FDEP, FEXT, and MASK instructions

deposit, 6-1, 10-178

extract, 6-1, 10-176

mask, 10-180

fixed- and floating-point conversion, 3-2, 6-4

fixed-point data, G-1

add example, 3-16, 3-17, 3-18

add example, conditional, 8-15

dual add/subtract example, 3-17

selection, 2-13

sideways sum example, 3-17, 3-19

fixed-point formats, 1-10, 2-19, 5-4

fixed-point fractional and integer, 5-3

fixed-point multiply, 5-6, 5-8, 5-9

fixed- to floating-point conversion, *See*

FLOAT instruction

FIX (fixed-point conversion) instruction, 3-2, 3-25, 10-69, A-5

example, 10-70

opcode description, C-10, C-11

FLAG3-0 (flag I/O) pin, 8-28

FLAG3-0 (flag I/O) pins, 1-5

flag enable (FLAGx_EN) bits, 8-28

flag input (FLGx) bits, 8-13, 8-28

update latency, 8-28

flag output (FLAGx_OUT) bits, 8-28

flag pin/bit updates, 8-13

- FLAGREG (flag regulator) register, 8-13, 8-28, 8-29
 - store dependency stall, 8-75
- FLAGREGST/CL (flag regulator set/clear) register
 - load execution latency, 8-75
- flags
 - conditions, 8-54
 - inexact, 2-18
 - LCxE, 8-54
 - TRUE, 8-54
- flag update, 3-12, 4-42, 5-21, 6-19, 7-11, 8-8
- FLAGx_EN (flag enable) bits, 8-28, 8-29
- FLAGx_IN (flag input) conditions, 8-9, 8-54, 8-98
 - condition code, C-45
- FLAGx_OUT (flag output) bits, 8-28, 8-29
- FLGx (flag input) bits, 8-13, 8-28
 - update latency, 8-28
- FLOAT (floating-point conversion)
 - instruction, 3-2, 3-25, 10-71, A-5
 - example, 10-72
 - opcode description, C-10, C-11
- floating-point data, G-1
 - add example, 3-17
 - conversion, 3-2
 - format, 1-10
 - multiply, 5-8, 5-9
 - selection, 2-13
- floating-point extended to normal word conversion, *See* SNGL instruction
- floating-point normal to extended word conversion, *See* EXTND instruction
- floating- to fixed-point conversion, *See* FIX instruction
- flow control, program, 8-1
- format, 1-12
 - fixed-point, 1-10, 2-13
 - floating-point, 1-6, 1-10, 2-13
 - fractional, 2-16, 2-21, 2-22, 2-23, 2-24, 2-25, 2-26, 5-13
 - IALU data, 7-8
 - instructions, 1-23, 1-25, 1-39
 - integer, 2-16, 2-21, 2-22, 2-23, 2-24, 2-25, 2-26, 5-13
 - saturation, 10-3, 10-6, 10-21
 - selection, 2-7, 2-13, 2-16, 5-13
 - signed, 5-12, 10-3, 10-6, 10-21
 - signed/unsigned maximum/minimum, 3-8, 10-3
 - unsigned, 5-12, 10-3, 10-6, 10-21
- formats
 - 32-, 64-, or 128-bit compaction, 10-46
 - 8- or 16-bit expansion, 10-41
 - 8- or 16-bit transpose, 10-50
 - data, 1-25, 2-1, 2-17
 - data word, 1-11
 - numeric, 2-16 to 2-26
 - shifter, 6-3
 - word size, 2-9
- Fourier transforms, 1-14
- fractional
 - compaction, 10-46
 - data, 2-7, 2-16
 - fixed-point, 5-3
 - format, 2-16, 3-10, 5-13, 7-8
 - multiply-accumulate example, 5-24
 - multiply example, 5-24, 5-25
 - operation, 3-6, 5-11
 - option, 5-13
 - results, 2-19
 - signed format, 2-21, 2-23, 2-24, 2-25
 - unsigned format, 2-22, 2-23, 2-25, 2-26
- full-scale right/left shift, 6-5

INDEX

G

GETBITS instruction, 2-5, 6-3, 6-12, 6-13, 6-24, 10-182, 10-184, A-10
BFP6 and Len7 fields, 6-13
example, 6-14, 10-183
opcode description, C-21
restrictions, 1-41
GIE (global interrupt enable) bit, 8-13, 8-27, B-15
example disable/enable, 8-91
ISR usage, 8-33
global interrupt enable (GIE) bit, *See* GIE (global interrupt enable) bit
global memory, 1-5, 1-22
global memory space, 9-5
global static flags, *See* ISF_x (IALU/global static flag) conditions
GSCF_x (IALU static flag) bits, 8-18

H

half-block of memory, 9-13, 9-14, 9-15
Harvard architecture, 9-1
H_{BG} (host bus grant) pin, 1-5
H_{BR} (host bus request) pin, 1-5
HDQM (SDRAM high word data mask) pin, 1-5
hidden bit, 2-16, 2-17, 2-19
host memory select (MSH) pin, 9-5
host processor, 1-5, 1-22
HW (hardware error interrupt) bit, 8-30

I

IAB (instruction alignment buffer), 1-2, 1-4, 8-1, 8-3, 8-6, 8-35, 8-39
and instruction pipeline, 8-35
data flow, 8-4
structure, 8-39

IALU (integer ALU), 1-2, 1-4, 1-13, 1-19, 7-1 to 7-51
ABS example, 7-6
add example, 7-6, 7-8
addressing examples, 7-16
address output, 7-15
add with divide by two example, 7-6
arithmetic instruction latency, 7-3
bit reverse addressing, 7-38, 7-39
carry, 7-12
circular buffer addressing, 7-5, 7-35, 7-36
COMP example, 7-8
conditions, 7-13, 8-9, 8-98, C-44
data flow, 7-2, 8-3, 8-4
dependency stall, 7-3, 8-69, 8-70, 8-71, 8-72, 8-74, 8-75
examples, 7-43
execution latency, 1-15, 8-69, 8-70
immediate extensions, 1-16, 7-42, 10-229, C-37, C-47
index update, 7-15
indirect addressing, 7-15
instruction encoding, C-26 to C-37
instructions, 10-199 to 10-231
instruction summary, 7-47
integer instructions, 7-1, 7-6, 10-199 to 10-217
load/store/transfer instructions, 7-1, 10-218 to 10-231
memory access, 8-7
opcode descriptions, C-3, C-27, C-29, C-31, C-32, C-33, C-36
operations, 7-1, 7-6, 7-14, 10-199
options, 7-7
overflow, 7-12
pipe, 8-36
quick reference, A-11
registers, 1-14, 1-46, 7-3, B-10, B-11, B-12

- IALU (integer ALU) *(continued)*
 resources, 1-31
 restrictions, 1-43
 result sign bit, 7-12
 status, 7-4, 7-10, 7-14
 subtract example, 7-8
 Ureg transfer exceptions, 10-230
- I (ALU integer operation) option, 3-7
- IALU static flag (ISFx) conditions
 load dependency stall, 8-73
- I-bus
 cacheability, 9-64
- ID, processor, 9-5
- ID2-0 (processor ID) pins, 1-5
- idle flow, 8-5
- IDLE instruction, 8-4, 8-9, 8-14, 8-99,
 8-100, 10-241, 10-242, 10-244,
 10-245, 10-246, 10-247, 10-248,
 10-249, 10-250, A-15, C-47
- IDLE (state) bit, 8-11
- IEEE 1149.1 standard
 JTAG specification, G-6
- IEEE 754/854 standard
 data types, 2-18
 exceptions, 1-22
 formats, 2-16
 rounding methods, 3-9, 5-15
- IF...DO (conditional execution)
 instruction, 1-27, 5-22, 7-13, 8-14,
 8-99, 10-241, A-14
 example, 10-241
- IF...ELSE (conditional sequencing and
 execution) instruction, 1-27, 8-15,
 8-100, 10-242, A-15
 example, 10-242
- I (instruction) bus, 1-2, 1-19, 8-3, 9-1, 9-2
 memory access, 8-6
 memory pipe access, 9-35
- I (integer operation) option, 3-10, 5-11,
 5-13
 restrictions, 1-42
- I (interleave data, CLU) option, 10-197
- ILAT (interrupt latch) register, 1-35, 8-26,
 8-30, 8-86, B-32
- illegal register transfer exception, 10-230
- imaginary numbers, 5-4
 data, 4-3, 7-3
 result, 5-4
- IMASK (interrupt mask) register, 1-35,
 8-26, 8-30, 8-86, 8-90, B-32
- immediate address, 1-16, 1-47
- immediate data, 1-16, 1-26
 operations, 7-1
- immediate extension, 1-15, 1-16, 1-36,
 1-38, 7-14, 7-34, 7-35, 7-38, 8-99,
 A-14, C-34, C-36, C-45
 Dreg register load, 10-223, 10-225
 Dreg register store, 10-227
 format, JUMP (sequencer), C-46
 format (IALU), C-47
 JUMP/CALL instructions, 10-234
 opcode fields (IALU), C-37
 operations, 1-16, 7-42
 restrictions, 1-23, 1-48, 7-43
 Ureg register store, 10-222
 Ureg register transfer, 10-229
 usage with JUMP addresses, 8-17
 with circular buffer addressing, 7-44
- immediate value
 immediate extension, 7-43
 modifier, 7-15
- INC (increment) instruction, 3-1, 3-23,
 10-21, A-3
 example, 10-22
 opcode description, C-6
 increment, *See* INC instruction

INDEX

- index (address), 7-16
 - circular buffer, 7-5
 - DAB accesses, 7-26
 - register, 7-34
 - SDAB accesses, 7-30
 - update, IALU, 7-15
- index field, memory transaction, 9-27
- indirect addressing, 1-14, 1-38, 7-14, 7-15, 7-34, 7-35, 7-38, 10-220, 10-222, 10-223, 10-226, 10-227, C-34, C-36
 - circular buffer setup, 7-35
 - current address, 7-5
 - index registers, 7-5, 7-34
- indirect jump, 1-16
- inexact flags, 2-18
- infinity, representation, 2-17
- input operand, 2-6
- INPUT_SECTION_ALIGN (4) linker command, 9-55
- input sequence, correlation, 4-30
- instruction
 - alignment, 1-18, 1-24
 - branch placement restriction, 1-49
 - caching, 9-51, 9-67
 - combination constraints, 1-16, 1-27
 - decode, 8-6
 - dispatch/decode, *See* sequencer
 - fetch, 8-36, 8-40, 8-49
 - format, 1-23
 - interaction, 1-29
 - line, 1-16, 1-23, 1-27, 2-6, 8-5, C-1
 - line, restrictions, immediate extension, 7-43
 - line, restrictions, results usage, 8-38
 - line delimiters, 1-23
 - line end markers, 1-24
 - notation conventions, 1-24
 - restrictions, 1-23
 - slot, 1-23, 1-27, 1-31, 2-6
 - slot and immediate extension, 7-43
- instruction *(continued)*
 - slot delimiters, 1-23
 - slot restrictions, 1-16
 - text symbols, 1-25
 - types, 1-28
- instruction parallelism, 1-16, 1-23, 1-26
 - CLU, 4-16, 4-20, 4-25
 - CLU instructions, 8-80
 - compute blocks, 1-41
 - results usage, 8-37
 - rules, 1-27
 - shifter restrictions, 6-3
 - single cycle, dual-data accesses, 9-4
 - Ureg load restriction, 1-35
 - Ureg transfer restrictions, 1-35
- instruction parallelism rules, 1-23
- instruction pipeline, 8-5
 - data flow, 9-35
 - operations, 8-34
 - stall example, 8-78, 8-82, 8-83, 8-85, 8-86, 8-88, 8-90
 - timing, 9-36
- instructions
 - add/subtract (integer) dependency stall, 8-70, 8-71, 8-72
 - bit field manipulation, 1-13
 - bit manipulation, 1-13, 6-4
 - branch placement, 8-49
 - combination constraints, 1-27
 - compute, 2-1
 - compute dependency stall, 8-66, 8-68, 8-69, 8-74, 8-75
 - compute execution latency, 8-68, 8-69
 - conditional, 1-27, 3-15, 5-22, 6-19, 7-13, 8-9, 8-14, 8-15
 - conditional (ALU condition) dependency stall, 8-68
 - conditional branch dependency stall, 8-76

- instructions *(continued)*
- conditional (IALU condition)
 - dependency stall, 8-70, 8-71, 8-72
 - conditional (loop condition) dependency stall, 8-74
 - conditional (loop condition) execution latency, 8-74
 - conditional RTI dependency stall, 8-75, 8-76
 - conditional (static flag condition)
 - dependency stall, 8-72, 8-73, 8-74
 - conditions summary, 8-98
 - control flow, 1-15
 - DAB (data alignment buffer), 7-26
 - dependency condition, 8-64
 - execution, 8-37
 - execution of memory access and compute block, 8-8
 - IALU dependency stall, 8-69, 8-70, 8-71, 8-72, 8-74, 8-75
 - IALU execution latency, 8-69, 8-70
 - IALU opcodes, C-26 to C-37
 - immediate extension, 1-15, 1-16
 - memory placement and BTB, 8-52
 - placing in memory, 9-2
 - quad, 1-17, 1-21
 - resource conflicts, 8-64
 - restrictions, 1-38
 - sequencer dependency stall, 8-69, 8-74, 8-75
 - sequencer opcodes, C-38 to C-47
- instruction set, 10-1 to 10-250
- ALU instructions, 10-2 to 10-95
 - CLU instructions, 10-96 to 10-106
 - IALU (integer) instructions, 10-199 to 10-217
 - IALU (load/store/transfer) instructions, 10-218 to 10-231
 - memory system commands, 9-58 to 9-81
- instruction set *(continued)*
- multiplier instructions, 10-107 to 10-169
 - sequencer instructions, 10-232 to 10-250
 - shifter instructions, 10-170 to 10-194
- INTCTL (interrupt control) register, 8-28, 8-29
- integer
- compaction, 10-46
 - data, 2-7, 7-8
 - format, 2-16, 3-10, 5-13, 7-8
 - multiplier, 5-13
 - operation, 3-7, 5-11, 7-7
- integer, fixed-point, 5-3
- integer, signed format, 2-21, 2-22, 2-24, 2-25
- integer, unsigned format, 2-22, 2-23, 2-24, 2-26
- integer ALU pipe, 8-6, 8-35, 8-36
- integer arithmetic logic unit, *See* IALU
- integer pipeline stage, 8-6, 8-7, 8-35
- integer pipe stage
- and memory pipeline, 9-36
 - bus requests, 8-81
- interleaved data, 7-30, 9-48, 9-52
- internal memory and BTB, 8-49
- internal memory space, 9-5
- internal transfer, 1-20
- interrupt, 1-17, 1-18, 8-4
- conditional instruction, 8-88
 - exception, 8-10, 8-12
 - flow, 8-5
 - interrupt disable, 8-90
 - masking, 8-26
 - pipeline, 8-86
 - sensitivity, B-38
 - software interrupt, 8-10, 8-12
- interrupt edge (IRQ_EDGE), 8-28
- interrupt latch (ILAT) register, 8-86, B-32

INDEX

interrupt mask (IMASK) register, 8-86,
8-90, B-32

interrupts, 8-1

- context save/restore, 7-3
- controller registers, 1-44
- data flow, 8-4
- edge and level sensitive, 8-27
- execution flow, 8-26
- latency, 1-17
- masked and unmasked, 8-26
- multiprocessor vector, 1-17
- pipeline example, 8-86
- return from interrupt operation, 10-238
- reusable/non-reusable, 8-32
- service routine steps, 8-33

interrupt service routine (ISR)

- address, 8-26
- reusable/non-reusable interrupts, 8-32
- steps, 8-33

interrupt vector table (IVT), 8-26

invalid exception, NAN, 2-18

invalid operation

- ALU, 3-12
- multiplier, 5-21

$\overline{\text{IOEN}}$ (I/O enable) pin, 1-5

I/O peripherals

- accesses, 9-1
- programming guidelines, 9-53

$\overline{\text{IOR}}$ (I/O read) pin, 1-5

$\overline{\text{IOWR}}$ (I/O write) pin, 1-5

$\overline{\text{IRQ3-0}}$ (external interrupt pin) bits, 8-31

$\overline{\text{IRQ3-0}}$ (external interrupt) pins, 1-5,
1-17, 8-26

$\overline{\text{IRQ_EDGE}}$ (interrupt edge/level
sensitivity select) bits, 8-28, 8-29,
B-38

ISFx (IALU/global static flag) conditions,
7-14, 8-9

- condition code, C-45
- dependency stall, 8-73

iterative convergence algorithm, 10-86

ITYPE (instruction type) field, C-2

IVEN (exception enable on invalid) bit,
2-4, 3-12, 3-14, 5-21, 5-22

J

J30-0 (J-IALU data) registers, 7-3

- load execution latency, 8-71, 8-72
- load/transfer execution latency, 8-71,
8-72
- transfer execution latency, 8-71, 8-72

J3-0 (J-IALU index) registers, 7-5, 7-34

J31/JSTAT (J-IALU status) register, 7-3

- load execution latency, 8-71, 8-72
- load/transfer execution latency, 8-71,
8-72
- operations, 7-4
- transfer execution latency, 8-71, 8-72

Jacobian logarithm, 4-3

JB3-0 (J-IALU base) registers, 7-5, 7-9,
7-35

- load execution latency, 8-71, 8-72
- transfer execution latency, 8-71, 8-72

J-bus

- cacheability, 9-64

JC (J-IALU carry) bit, 7-4

- operation, 7-12
- updated by, 7-11

J (complex conjugate multiply) option, 5-5,
5-12, 5-19

JEQ (J-IALU equal to zero) condition,
7-13, 8-9

- condition code, C-44

J-IALU, *See* IALU

J (J-IALU) bus, 1-2, 1-19, 8-3, 9-1, 9-2

- memory access, 8-6
- memory pipe access, 9-35
- stall, 8-80

- load/transfer execution latency, [8-71](#), [8-72](#)
 - JLE (J-IALU less than or equal to zero)
 - condition, [7-13](#), [8-9](#)
 - condition code, [C-44](#)
 - JLT (J-IALU less than zero) condition,
 - [7-13](#), [8-9](#)
 - condition code, [C-44](#)
 - JN (J-IALU negative) bit, [7-4](#)
 - operation, [7-12](#)
 - updated by, [7-11](#)
 - JSTAT (J-IALU status) register, [7-3](#), [7-11](#)
 - operations, [7-4](#)
 - JTAG (joint test action group) port, [1-3](#), [1-22](#), [G-6](#)
 - registers, [1-44](#)
 - jump
 - example, [8-20](#)
 - example, conditional, [8-16](#), [8-21](#)
 - jump flow, [8-5](#)
 - JUMP instruction, [1-36](#), [8-4](#), [8-5](#), [8-19](#), [8-20](#), [8-99](#), [10-234](#), [A-14](#)
 - `.ALIGN_CODE` assembler directive,
 - BTB example, [8-53](#)
 - conditional, [8-16](#)
 - eliminating branch cost, [8-49](#)
 - example, [8-20](#), [8-92](#), [10-235](#)
 - example, conditional, [8-21](#)
 - immediate extension format, [C-45](#)
 - latency, [1-17](#)
 - JV (J-IALU overflow) bit, [7-4](#)
 - operation, [7-12](#)
 - updated by, [7-11](#)
 - JZ (J-IALU zero) bit, [7-4](#)
 - operation, [7-11](#)
 - updated by, [7-11](#)
- K**
- load execution latency, [8-71](#), [8-72](#)
 - load/transfer execution latency, [8-71](#), [8-72](#)
 - transfer execution latency, [8-71](#), [8-72](#)
 - K3-0 (K-IALU index) registers, [7-5](#), [7-34](#)
 - K31/KSTAT (K-IALU status) register, [7-3](#)
 - load execution latency, [8-71](#), [8-72](#)
 - load/transfer execution latency, [8-71](#), [8-72](#)
 - operations, [7-4](#)
 - transfer execution latency, [8-71](#), [8-72](#)
 - KB3-0 (K-IALU base) registers, [7-5](#), [7-9](#), [7-35](#)
 - load execution latency, [8-71](#), [8-72](#)
 - transfer execution latency, [8-71](#), [8-72](#)
 - K-bus
 - cacheability, [9-64](#)
 - KC (K-IALU carry) bit, [7-4](#)
 - operation, [7-12](#)
 - updated by, [7-11](#)
 - KEQ (K-IALU equal to zero) condition,
 - [7-13](#), [8-9](#)
 - condition code, [C-44](#)
 - K-IALU, *See* IALU
 - K (K-IALU) bus, [1-2](#), [1-19](#), [8-3](#), [9-1](#), [9-2](#)
 - memory access, [8-6](#)
 - memory pipe access, [9-35](#)
 - stall, [8-80](#)
 - KL3-0 (K-IALU length) registers, [7-5](#), [7-9](#), [7-34](#)
 - load/transfer execution latency, [8-71](#), [8-72](#)
 - KLE (K-IALU less than or equal to zero)
 - condition, [7-13](#), [8-9](#)
 - condition code, [C-44](#)
 - KLT (K-IALU less than zero) condition,
 - [7-13](#), [8-9](#)
 - condition code, [C-44](#)

INDEX

- KN (K-IALU negative) bit, [7-4](#)
 - operation, [7-12](#)
 - updated by, [7-11](#)
- KSTAT (K-IALU status) register, [7-3](#), [7-11](#)
 - operations, [7-4](#)
- KV (K-IALU overflow) bit, [7-4](#)
 - operation, [7-12](#)
 - updated by, [7-11](#)
- KZ (K-IALU zero) bit, [7-4](#)
 - operation, [7-11](#)
 - updated by, [7-11](#)
- L**
- label
 - address conversion, [8-17](#)
 - call example, [8-21](#)
 - call example, conditional, [8-21](#)
 - jump example, [8-20](#)
 - jump example, conditional, [8-16](#), [8-21](#)
 - program, [8-5](#), [8-19](#), [8-97](#), [10-234](#)
 - program, with immediate extension, [8-20](#)
- latency
 - compute block versus IALU instruction, [7-3](#)
 - flag pin, [8-13](#)
 - IALU (integer ALU), [1-15](#)
 - interrupt, [1-17](#)
 - jump, [1-17](#)
- LBUFRXx (link port receive buffer)
 - registers, [1-35](#)
- LBUFTXx (link port transmit buffer)
 - registers, [1-35](#)
- LCTLx (link port control) registers, [1-35](#)
- LCxE conditions, [8-54](#)
- LCxE (loop counter expired) condition, [8-9](#), [8-22](#), [8-98](#)
- LCxE (loop counter expired) conditions
 - opcodes, [C-45](#)
- LCx (loop counter) registers, [1-35](#), [B-13](#)
 - load execution latency, [8-74](#)
 - store dependency stall, [8-74](#)
- LDQM (SDRAM low word data mask)
 - pin, [1-5](#)
- LDx (leading ones/zeros) instruction, [6-12](#), [6-24](#), [10-190](#), [A-10](#)
 - example, [6-21](#), [10-190](#)
 - opcode description, [C-24](#)
- leading ones or zeros, [6-1](#)
- least recently used (LRU) algorithm, [8-45](#), [9-25](#), [9-26](#), [9-29](#), [9-42](#)
- left rotate, [10-217](#)
- Len7 (bit field length) field, [6-9](#), [6-10](#), [6-11](#), [6-13](#)
- length, bit reverse addressing buffer, [7-40](#)
- length, circular buffer, [7-5](#)
- length (CACHE_INIT command)
 - parameter, [9-71](#)
- length (CACMD_CB command)
 - parameter, [9-75](#)
- length (CACMD_INV command)
 - parameter, [9-79](#)
- length registers, [7-34](#)
 - DAB accesses, [7-26](#)
- level sensitive interrupts, [8-27](#)
- line, cache, [9-13](#), [9-25](#)
- linear flow, [8-5](#)
- LINK3-0 (link interrupt) bits, [8-31](#)
- linker description file (LDF), [9-53](#)
- link port, [1-1](#), [1-3](#), [1-4](#), [1-8](#), [1-17](#), [1-19](#)
 - booting, [1-21](#)
 - register groups, [1-35](#)
 - registers, [1-44](#)
- literal values, [1-26](#)
- L (long word access) operator, [7-17](#), [7-18](#), [7-48](#), [10-222](#), [10-223](#)
- L (long word access) option, [2-10](#)

- load, [1-14](#), [1-18](#), [1-20](#), [1-29](#), [1-35](#)
 - bus usage, [8-80](#)
 - definition, [G-7](#)
 - example, [3-20](#)
 - example, conditional, [8-15](#)
 - execution latency, [8-68](#)
 - IALU operations, [7-1](#)
 - normal, merged, and broadcast accesses, [7-17](#)
 - opcodes, [C-32](#), [C-33](#), [C-36](#)
 - operations, [7-1](#)
 - post-modify example, [7-16](#)
 - pre-modify example, [7-16](#)
 - register, [C-32](#), [C-33](#), [C-36](#)
 - resources, [1-31](#), [1-32](#), [1-34](#)
 - restrictions, [1-38](#), [1-44](#), [1-45](#), [1-46](#)
- load, store, and transfer operations, *See* IALU instructions
- load BFOTMP register
 - example, [10-194](#)
 - opcode description, [C-24](#)
- load CACMDx registers, restrictions, [9-20](#)
- load CCAIRx registers, restrictions, [9-20](#)
- load CCNTx registers, execution latency, [8-76](#), [10-220](#), [10-230](#)
- load CJMP register, execution latency, [8-76](#)
- load CMCTL register
 - execution latency, [8-67](#)
 - opcode description, [C-25](#)
- load data
 - instruction, [10-229](#)
 - opcodes, [C-31](#)
- load Dreg registers, [7-49](#)
 - execution latency, [8-68](#), [8-69](#)
 - with CB, dependency stall, [8-71](#)
 - with DAB, dependency stall, [8-69](#)
 - with DAB offset operation, [10-225](#)
 - with DAB/SDAB operation, [10-223](#)
- load EMUCTL register, execution latency, [8-75](#)
- load FLAGREGST/CL register, execution latency, [8-75](#)
- load J30-0 registers, execution latency, [8-71](#), [8-72](#)
- load J31/JSTAT register, execution latency, [8-71](#), [8-72](#)
- load JB3-0 registers, execution latency, [8-71](#), [8-72](#)
- load JL3-0 registers, execution latency, [8-71](#), [8-72](#)
- load K30-0 registers, execution latency, [8-71](#), [8-72](#)
- load K31/KSTAT register, execution latency, [8-71](#), [8-72](#)
- load KB3-0 registers, execution latency, [8-71](#), [8-72](#)
- load KL3-0 registers, execution latency, [8-71](#), [8-72](#)
- load LCx registers, execution latency, [8-74](#)
- load memory system registers, execution latency, [8-69](#)
- load MR register
 - example, [10-155](#), [10-156](#), [10-157](#), [10-158](#)
 - opcode description, [C-17](#)
- load PRFCNT register, execution latency, [8-76](#), [10-220](#), [10-230](#)
- load PRFM register, execution latency, [8-76](#), [10-220](#), [10-230](#)
- load PR register, opcode description, [C-9](#)
- load RETI register, execution latency, [8-76](#)
- load SFREG register, execution latency, [8-72](#), [8-73](#), [8-74](#)
- load SFx condition, [10-243](#)
 - dependency stall, [8-73](#), [8-74](#)
 - execution latency, [8-73](#), [8-74](#)
- load SQCTL register, execution latency, [8-75](#)

INDEX

- load THR register, [4-46](#), [4-47](#), [10-99](#), [A-6](#)
 - execution latency, [8-67](#)
 - opcode description, [C-25](#)
 - stall, [8-79](#)
- load TRCB31-0 registers, execution latency, [8-76](#), [10-220](#), [10-230](#)
- load TRCBMASK register, execution latency, [8-76](#), [10-220](#), [10-230](#)
- load TRCBPTR register, execution latency, [8-76](#), [10-220](#), [10-230](#)
- load TRCBVAL register, execution latency, [8-76](#), [10-220](#), [10-230](#)
- load TR register, [4-46](#), [4-47](#), [10-99](#), [A-6](#)
 - execution latency, [8-66](#), [8-67](#)
 - opcode description, [C-25](#)
 - stall, [8-79](#)
- load Ureg register, [7-42](#), [7-49](#)
 - example, [7-44](#)
 - execution latency, [8-70](#), [8-75](#)
- load Ureg usage
 - parallelism restriction, [1-35](#)
- load WPxCTL registers, execution latency, [8-76](#), [10-220](#), [10-230](#)
- load WPxH/L registers, execution latency, [8-76](#), [10-220](#), [10-230](#)
- load WPxSTAT registers, execution latency, [8-76](#), [10-220](#), [10-230](#)
- load X/YSTAT registers, [6-9](#)
 - example, [10-195](#)
 - execution latency, [8-74](#)
 - opcode description, [C-24](#)
- local access policy, [9-46](#), [9-47](#), [9-49](#)
- lock bit, cache, [9-25](#), [9-26](#)
- _LOCK (CACHE_INIT command)
 - parameter, [9-72](#)
- logarithm, [3-3](#), [3-26](#), [10-92](#), [A-5](#), [C-11](#)
 - See also* LOGB instruction
- LOGB (logarithm) instruction, [3-3](#), [3-26](#), [10-92](#), [A-5](#)
 - example, [10-93](#)
 - opcode description, [C-11](#)
- logical AND/AND NOT/OR/XOR/NOT, [10-39](#), [10-213](#)
- logical operations, [3-2](#)
 - ALU, [10-2](#), [10-39](#)
 - CLU, [10-96](#)
 - IALU, [3-2](#), [10-213](#)
 - opcode description, [C-6](#)
- logical shift, [10-171](#), [10-215](#)
 - See also* LSHIFT or LSHIFTR instructions
 - example, [6-21](#)
 - operation, [6-6](#)
- long word, [1-11](#), [2-25](#), [2-26](#), [5-3](#), [6-3](#)
 - 32-, 64-, or 128-bit compaction, [10-46](#)
 - accesses, [7-17](#), [7-48](#)
 - accumulation, [5-19](#)
 - add example, [3-16](#)
 - data, [2-10](#), [2-16](#)
 - move, [7-2](#)
 - shifter operations, [6-8](#)
 - shifter result, [6-5](#)
- loop, [1-26](#), [8-4](#)
 - counter updates, [1-38](#)
 - execution flow, [8-22](#)
 - flow, [8-5](#)
 - set up, [8-22](#)
 - zero-overhead loop and
 - near-zero-overhead example, [8-93](#)
- loop counter expired (LCxE) condition, [8-22](#), [8-98](#)
- loop counter (LCx) register, [B-13](#)
- LSHIFT (logical shift) instruction, [6-6](#), [6-24](#), [10-171](#), [A-9](#)
 - example, [6-6](#), [6-21](#), [8-16](#), [10-172](#)
 - opcode description, [C-20](#), [C-21](#), [C-22](#)

- LSHIFTR (integer) instruction
 opcode description, [C-29](#)
- LSHIFTR (logical shift right) instruction,
[7-49](#), [10-215](#), [A-11](#)
 example, [10-216](#)
- LSTATx (link port status) registers, [1-35](#)
- .L unit, *See* ALU
- L unit, *See* ALU
- LxACKI (link port input acknowledge)
 pins, [1-5](#)
- LxACKO (link port acknowledge) pins, [1-5](#)
- LxBCMPI (link port block input complete)
 pins, [1-5](#)
- LxBCMPO (link port block output
 complete) pins, [1-5](#)
- LxCLKINP/N (link port input clock) pins,
[1-5](#)
- LxCLKOUTP/N (link port output clock)
 pins, [1-5](#)
- LxDATI3-0 (link port input data) pins, [1-5](#)
- LxDATO3-0P/N (link port output data)
 pins, [1-5](#)
- ## M
- MACs, *See* multiply-accumulate
- magnitude, rotate, [10-174](#)
- mantissa, [2-16](#), [2-17](#), [2-19](#), [3-3](#), [5-8](#), [5-9](#)
- MANT (mantissa) instruction, [3-3](#), [3-26](#),
[10-90](#), [A-5](#)
 example, [10-91](#)
 opcode description, [C-11](#)
- manual
 audience, [xxi](#)
 contents, [xxii](#)
 conventions, [xxxii](#)
 new in this edition, [xxiv](#)
 related documents, [xxvii](#)
- mask, field, *See* MASK instruction
- masked bits, shifter, [6-5](#)
- masked interrupt, [8-26](#)
- MASK (field/bit mask) instruction, [1-36](#),
[1-37](#), [6-3](#), [6-9](#), [6-24](#), [10-180](#), [A-9](#)
 example, [10-181](#)
 opcode description, [C-20](#), [C-21](#)
 restrictions, [1-41](#)
- MASK (mask bit field) instruction
 example, [6-12](#)
- maximum, *See* MAX, VMAX, or TMAX
 instructions
- maximum signed number, [10-3](#)
- maximum unsigned number, [10-3](#)
- MAX (maximum, CLU) instruction, [4-9](#),
[4-46](#), [10-98](#), [A-6](#)
 dependency stall, [8-67](#)
 execution latency, [8-67](#)
 opcode description, [C-13](#)
 trellis data flow example, [4-20](#)
- MAX (maximum, floating-point)
 instruction, [3-25](#), [10-60](#), [A-4](#)
 example, [10-61](#)
 opcode description, [C-11](#)
- MAX (maximum, integer) instruction,
[7-48](#), [10-209](#), [A-11](#)
 example, [10-210](#)
 opcode description, [C-28](#)
- MAX (maximum) instruction, [3-1](#), [3-10](#),
[3-23](#), [10-15](#), [A-2](#)
 example, [10-16](#)
 opcode description, [C-5](#)
 trellis data flow example, [4-18](#), [4-19](#),
[4-21](#)
- MAX_SN (maximum signed number),
[10-3](#)
- MAX_UN (maximum unsigned number),
[10-3](#)

INDEX

- memory, 9-1 to 9-81
 - See also* broadcast accesses, merged accesses, normal accesses, and embedded DRAM
 - cache, 9-13
 - data flow, 9-2
 - global, 1-5, 1-22
 - instruction fetch, 8-4
 - instruction placement and BTB, 8-52
 - logical organization, 9-14
 - map, 9-6
 - organization, 1-4
 - physical structure, 9-5
 - placing instructions and data, 9-2
 - shared, 1-5
 - throughput, 1-19
 - wasted space, 8-40
- memory, external, 1-5, 1-44, 9-5
 - access delay, 8-37
 - instruction fetch, 8-36, 8-40
- memory, global, 9-5
- memory, internal, 1-8, 1-19, 9-5
- memory, multiprocessor, 9-5
- memory accesses, 9-16
 - alignment restrictions, 7-19
 - broadcast load, 7-17
 - dependency stall, 8-69
 - examples, 9-53
 - merged read/write, 7-17, 7-18
 - normal read/write, 7-17
 - penalty analysis with simulator/emulator, 9-45
 - penalty summary, 9-44
 - priority, 9-3
 - programming guidelines, 9-46, 9-48, 9-49, 9-50, 9-51, 9-52
 - repeated series, 7-26
 - restrictions, 1-43
- memory bank, 9-5
- memory block, 1-3, 1-8, 1-17, 1-19, 9-2, 9-4, 9-5, 9-7, 9-13, 9-14, 9-15, 9-16, 9-55
 - broadcast, 9-20
- memory bus, 9-2
 - arbitration, 9-33
 - cacheability, 9-64
- memory half-block, 9-7, 9-13, 9-15
- MEMORY { } linker command, 9-54
- memory-mapped I/O, 1-5
- memory-mapped registers, 2-2
- memory page, 9-13, 9-14, 9-16
- memory pipeline, 8-6, 8-8, 9-35
 - stages, 9-36
 - stalls, 9-35
 - timing, 9-36
- memory refresh rate
 - (CASTAT_REFCNTR) bits, 9-23
- memory segments (in LDF), 9-54
- memory select, host ($\overline{\text{MSH}}$) pin, 9-5
- memory select, SDRAM ($\overline{\text{MSSDx}}$) pins, 9-5
- memory select ($\overline{\text{MSx}}$) pins, 9-5
- memory sub-array, 9-13, 9-14, 9-15
 - interrupted access, 9-18
- memory system commands, 9-58 to 9-81
 - summary, 9-56
- memory system exception, 9-72, 9-76, 9-80
- memory transaction, 9-8
 - cache operation, 9-27
 - fields, 9-27
- MEQ (multiplier equal to zero) condition, 5-23, 8-9
 - condition code, C-43
- merge, *See* MERGE instruction
- merged accesses, 1-44, 7-17
 - transfer restrictions, 1-44
- merged read access example, 7-20, 7-22
- merged read/write memory accesses, 7-18
- merged write access example, 7-24, 7-25

- MERGE instruction, 1-36, 1-37, 3-1,
3-25, 10-50, A-4
example, 10-51
opcode description, C-8
- mesh configuration, 1-22
- metrics, Viterbi algorithm, 4-9
- MI (multiplier invalid) bit, 2-5
See also shifter status
updated by, 5-21
- minimum, *See* MIN or VMIN instructions
- minimum signed number, 10-3
- MIN (minimum, floating-point)
instruction
example, 10-61
opcode description, C-12
- MIN (minimum, integer) instruction,
7-48, 10-209, A-11
example, 10-210
opcode description, C-28
- MIN (minimum) instruction, 3-1, 3-10,
3-23, 3-25, 10-15, 10-60, A-2, A-4
example, 10-16
opcode description, C-5
- MIN_SN (minimum signed number),
10-3
- MIS (multiplier invalid, sticky) bit, 2-4
See also multiplier status
updated by, 5-22
- MLE (multiplier less than or equal to zero)
condition, 5-23, 8-9
condition code, C-43
- MLT (multiplier less than zero) condition,
5-23, 8-9
condition code, C-43
- MN (multiplier negative) bit, 2-5
See also multiplier status
updated by, 5-21
- mode, 8-11
See also integer operation, fractional
operation, or rounding operation
bits, 1-25
emulation mode, 8-11
supervisor, ISR usage, 8-33
supervisor mode, 8-11
user mode, 8-11
- MODE (operation mode) bit, 8-11
- modified addressing, G-7
- modified value, 7-9
- modifier, 7-16
addressing, 7-9, 7-15
circular buffer, 7-5
- modifier registers
DAB accesses, 7-26
- modifier value, 7-34
SDAB accesses, 7-29
- modify registers, G-8
- modulo addressing, 1-14
- MOS (multiplier overflow, sticky) bit, 2-4
See also multiplier status
updated by, 5-22
- move, *See* transfer
- MR4 (multiplier result overflow) register,
2-4
- MR (multiplier result) register, 2-4, 2-5,
5-6, 5-14, 5-18, 5-19, 5-28, 10-152,
10-160, A-8
COMPACT, opcode description, C-18
dependency stall, 8-68
execution latency, 8-68
load, opcode description, C-17
transfer, opcode description, C-18
- MS1-0 (memory select) pins, 1-5, 9-5
- MSH (memory select, host) pin, 9-5
- MSSD3-0 (memory select, SDRAM) pins,
1-5, 9-5
- multichannel buffered serial port, McBSP,
See link ports

INDEX

- multiplication
 - normal word, 5-4
 - short word, 5-4
- multiplier, 1-2, 1-4, 5-1 to 5-28, 10-107
 - conditional instructions, 8-14
 - conditions, 5-22, 8-98, C-43
 - data flow, 5-2
 - examples, 5-24
 - fixed-/floating-point data flow, 5-8, 5-9
 - instruction parallelism, 1-41
 - integer, fractional, 5-13
 - normal word, fractional, accumulate
 - example, 5-24
 - normal word, fractional example, 5-24, 5-25
 - operations, 5-5
 - options, 5-9
 - overflow, 5-19
 - quick reference, A-7
 - registers, 2-1
 - short word, complex example, 5-25
 - short word, fractional example, 5-24, 5-25
 - static flags, 5-23
 - status, 5-20, 5-22, B-2
 - sticky status, B-3
- multiplier instructions, 10-107 to 10-169, C-12
 - dependency stall, 8-66, 8-68, 8-69, 8-74, 8-75
 - example, 10-110, 10-126, 10-151
 - execution, 8-37
 - execution latency, 8-68, 8-69
 - opcode description, C-16
 - resources, 1-37
- multiplier results, 2-20, 5-3, 5-8, 5-9
 - See also* MR register
 - bus usage, 1-41
 - latency, 8-37
- multiplier results *(continued)*
 - rounding, 5-16, 5-17
 - saturation, 5-13, 5-14
 - shifting, 2-20
- multiply
 - fixed-point example, 5-6
- multiply-accumulate, complex
 - definition, 5-4
- multiply-accumulate, complex (short word)
 - instruction
 - example, 10-141, 10-142
- multiply-accumulate instructions
 - complex, 10-143
 - example, 10-113, 10-114, 10-130, 10-131, 10-132
 - opcode description, C-17
 - restrictions, 1-42
 - result latency, 8-37
- multiply-accumulate operations, 5-1, 5-3
 - complex, throughput, 4-3
 - multiplier instructions, 10-107
- + = * multiply-accumulate operator
 - dependency stall, 8-68
 - execution latency, 8-68
- = * multiply-accumulate operator
 - dependency stall, 8-68
 - execution latency, 8-68
- multiply-accumulates, complex
 - throughput, 4-22
- multiply-accumulate with transfer MR
 - register, complex (dual operation, short word) instruction
 - example, 10-145, 10-146, 10-147, 10-148
- multiply-accumulate with transfer MR
 - register (dual operation, normal word) instruction
 - example, 10-119, 10-120, 10-121, 10-122
 - opcode description, C-16

- multiply-accumulate with transfer MR
 - register (dual operation, quad-short word) instruction
 - example, [10-135](#), [10-136](#), [10-137](#)
 - multiply-accumulator, *See* multiplier
 - multiply operations, [5-1](#), [5-3](#)
 - multiplier instructions, [10-107](#)
 - * multiply operator, [5-13](#)
 - multiprocessing, [1-22](#)
 - system, [1-6](#)
 - multiprocessor memory space, [2-3](#), [9-5](#)
 - MU (multiplier underflow) bit, [2-5](#)
 - See also* multiplier status
 - updated by, [5-21](#)
 - .M unit, *See* multiplier
 - M unit, *See* multiplier
 - MUS (multiplier underflow, sticky) bit, [2-4](#)
 - See also* multiplier status
 - updated by, [5-22](#)
 - MV (multiplier overflow) bit, [2-5](#)
 - See also* multiplier status
 - updated by, [5-21](#)
 - MVS (multiplier overflow, sticky) bit, [2-4](#)
 - See also* multiplier status
 - updated by, [5-22](#)
 - MZ (multiplier zero) bit, [2-5](#)
 - See also* multiplier status
 - updated by, [5-21](#)
- N**
- NAN (not-a-number) exception, [2-18](#)
 - near-zero-overhead loops, [8-22](#)
 - example, [8-93](#)
 - negated conditions, [8-9](#), [8-98](#)
 - nested calls, [1-18](#)
 - Newton Raphson iteration algorithm, [10-88](#)
 - NF (no flag update) option, [3-7](#), [3-11](#), [4-41](#), [5-12](#), [5-19](#), [6-17](#), [6-18](#), [7-7](#), [7-10](#)
 - conditional instructions, [8-16](#)
 - nibble, [10-52](#)
 - NMOD (normal mode) bit, [8-11](#), [8-13](#), [B-15](#)
 - no flag update (NF) option, [3-7](#), [3-11](#), [5-19](#), [6-18](#), [7-10](#)
 - conditional instructions, [8-16](#)
 - non-nested calls, [1-18](#)
 - non-reusable interrupts, [8-32](#)
 - no operation, *See* NOP instruction
 - NOP (no operation) instruction, [8-9](#), [8-14](#), [8-99](#), [10-241](#), [10-242](#), [10-244](#), [10-245](#), [10-246](#), [10-247](#), [10-248](#), [10-249](#), [10-250](#), [C-47](#)
 - unconditional restriction, [8-9](#)
 - normal accesses, [7-17](#)
 - normal mode (NMOD) bit, [8-11](#), [8-13](#), [B-15](#)
 - normal read access example, [7-19](#), [7-21](#), [7-22](#)
 - normal read/write memory accesses, [7-17](#)
 - normal word, [1-11](#), [2-24](#), [2-25](#), [5-3](#), [6-3](#)
 - 32-, 64-, or 128-bit compaction, [10-46](#)
 - 8- or 16-bit expansion, [10-41](#)
 - ABS (integer) example, [7-6](#)
 - accesses, [7-48](#)
 - accumulation, [5-19](#)
 - add example, [3-17](#), [7-6](#), [7-8](#), [8-15](#)
 - AND example, [3-17](#)
 - COMP (integer) example, [7-8](#)
 - dual add/subtract example, [3-17](#)
 - load example, conditional, [8-15](#)
 - multiplication, [5-4](#)
 - multiply-accumulate example, [5-24](#)
 - multiply example, [5-24](#), [5-25](#)
 - shifter operations, [6-8](#)

INDEX

normal word *(continued)*
 shifter result, 6-5
 subtract (integer) example, 7-8
 transfer, 7-2
normal word data, 2-10, 2-14, 2-15
normal write access example, 7-23, 7-24, 7-25
not-a-number (NaN), handling, 2-17
notation conventions, instruction, 1-24
NOT instruction
 example, 10-40, 10-214
 opcode description, C-6, C-28
not predicted (NP) branch, 8-21
 See also NP option or conditional branch
NP (not predicted CJMP/CJMP_CALL)
 option, 8-99, 10-236, A-14
NP (not predicted JUMP/CALL) option,
 8-21, 8-48, 8-53, 8-59, 8-99, 10-234, A-14
 example, conditional, 8-21
NP (not predicted RETI/RTI) option,
 8-99, 10-238, A-14
numeric formats, 2-16

O

OEN (exception enable on overflow) bit,
 2-4, 3-12, 3-13, 3-14, 5-21, 5-22, B-2
 See also XSTAT register
off-scale left/right, 6-1
one's complement, 3-1
ones, leading, 6-1
ones counting, *See* ONES instruction
ONES (counting) instruction, 3-1, 3-24, 10-29, A-3
 example, 10-29
 opcode description, C-9
ones/zeros extract example, 6-21

opcodes, xxiii
 ALU instructions, C-7, C-10
 constructing, xxiii
 IALU instructions, C-27, C-29, C-33, C-36
 immediate extension instructions, C-37
 miscellaneous instructions, C-46
 other instructions, C-47
 sequencer instructions, C-39, C-40
 shifter instructions, C-23, C-25
operand
 size, 2-10
 size and execution, 2-11
 size selection, 2-7
operands, 2-10
 denormal, 2-18
operands, input and output, 2-6
operation
 selection, 1-24
operation mode, 8-11
 emulation mode, 8-11
 supervisor mode, 8-11
 user mode, 8-11
operations
 add example, 3-16, 3-17, 3-18
 add example, conditional, 8-15
 arithmetic, 3-1
 call example, 8-21
 call example, conditional, 8-21
 data conversion, 3-1
 dual add/subtract example, 3-17
 jump example, 8-20
 jump example, conditional, 8-16, 8-21
 load example, conditional, 8-15
 logical, 3-2
 logical AND example, 3-17
 sideways sum example, 3-19
 sum example, 3-17
option
 selection, 1-25

options
 ALU examples, 3-7
 format, 1-25
 OR instruction, 3-2
 example, 10-40
 opcode description, C-6
 OR (integer) instruction
 example, 10-214
 opcode description, C-28
 out-of-order access, prefetch sequence,
 9-41
 output operand, 2-6
 overflow
See also AV and MV bits
See also XSTAT register
 bit reverse operations, 7-12
 circular buffer, 1-14, 7-9, 7-33
 CLU, 4-5
 fixed-point operations, 2-5
 IALU, 7-12
 multiplier, 5-13, 5-19, 5-21
 saturation, 10-3
 shifter, 6-12
 shifter operations, 2-5
 trellis, 2-5
 overlaps in execution, 8-38

P

packed data formats, 2-20
 page, open an embedded DRAM, 9-18
 page buffer, 9-15, 9-17
 page of memory, 9-13, 9-14, 9-16
 prefetch sequence, 9-42
 parallel execution, 1-26, 8-37
 parallel results, 3-1, 3-4
 parenthesis in syntax, 1-25
 parity-check codes, 4-2
 pass, *See* PASS instruction

PASS instruction, 3-2, 3-24, 3-26, 10-38,
 10-83, A-3, A-5
 example, 10-38, 10-84
 opcode description, C-6, C-11
 path search algorithm, 1-12
 PC (program counter), 1-2, 8-3, 8-20
 data flow, 8-4
 relative address, 8-17, 8-20, 8-97
 relative branch, 1-18
 stack, 1-18
 penalties, 9-45
 bus conflicts and non-sequential accesses,
 9-37
 cache miss transaction, 9-39, 9-40
 definition, 8-68
 local access policy, 9-49
 memory access summary, 9-44
 memory block accesses, 9-16
 sequential access policy, 9-48
See also stall
 penalty cycles
 incorrect branch prediction, 8-48
 instruction fetch across quad word
 boundary, 8-49
 summary, 8-55
 penalty-free transactions, 9-44
 penalty-laden transactions, 9-46
 performance
 block orientation, 9-52
 BTB coding techniques, 8-52
 cache disable, 9-50
 cache enable, 9-50
 disabling interrupts, 8-90
 double sequence flow, 9-52
 eliminating branch cost, 8-49
 instruction caching, 9-51
 local access policy, 9-49
 locking data or code into the cache, 9-29
 memory and bus throughput, 9-3
 penalty-free transactions, 9-44

INDEX

- performance *(continued)*
 - result dependency stall, 8-38
 - results usage, parallel instructions, 8-38
 - sequential access policy, 9-48
 - set cacheability, 9-66
- peripherals, G-9
- PERMUTE (byte word, CLU) instruction,
 - 3-25, 10-54, A-4
 - example, 10-53
- PERMUTE (short word, CLU)
 - instruction, 3-25, 4-3, 10-52, A-4
 - example, 10-55
 - opcode description, C-9
- pipeline, 1-8, 1-9, 1-16, 1-17
 - branches, 8-53
 - bubbles, 8-78, 8-85
 - compute block, 8-37
 - operations, 8-34
 - stages, 8-6
 - stall example, 8-78, 8-82, 8-83, 8-85, 8-86, 8-88, 8-90
 - stalls, 8-64, 8-78, 8-85
- pipeline performance
 - branch prediction, 8-21
- pipeline stages, 8-35
- PMASK (interrupt mask pointer) register,
 - 1-35, 8-26, 8-30
- pointers, stray, 1-28
- pointer wraparound, 1-14
- polynomial reordering, 4-3
- POR_IN (power on reset, DRAM) pin, 1-5
- Pos8 (bit field position) field, 6-9, 6-10, 6-11
- post-modify addressing, 1-14, 1-38, 7-14, 7-34, 7-35, 7-38, 10-220, 10-222, 10-226, 10-227, C-36
 - current address, 7-5
- post-modify and load example, 7-16
- post-modify and update, 7-15
- [+=] post-modify with update operator,
 - 1-14, 7-14
 - bit reverse usage, 7-38
 - circular buffer usage, 7-34, 7-35
 - definition, G-9
 - example, 7-16
 - load Dreg, 10-223, 10-225
 - opcode description, C-36
 - operation restriction, 1-38
 - store Dreg, 10-227
 - store Ureg, 10-222
- precharge an embedded DRAM page, 9-18
- precision, G-10
- predecode pipeline stage, 8-6, 8-7, 8-35
- predicated execution, 1-27
- predicted branch, 8-21
 - correct prediction pipe example, 8-56, 8-57
 - incorrect prediction pipe example, 8-58, 8-59, 8-60, 8-61, 8-62, 8-63
 - stalls from incorrect prediction, 8-48
- predicted branches
 - aligned quad words and the BTB, 8-50
- prefetch buffer, 9-3, 9-7, 9-8, 9-12, 9-17
 - hit, 9-10, 9-12, 9-31, 9-45
 - miss, 9-39, 9-40, 9-45
 - operation sequence, 9-43
 - set cacheability, 9-66
- prefetch mechanism, 9-9
- prefetch pages, 9-8, 9-9
- prefetch sequence
 - cross page, 9-42
 - direction determination, 9-41
 - interrupted, 9-43
- pre-modify addressing, 1-14, 7-14, 7-15, 7-35, 10-220, 10-222, C-34
 - current address, 7-5
 - load example, 7-16

- [+] pre-modify no update operator, 1-14, 7-14
 - definition, G-10
 - example, 7-16
 - not for circular buffer, 7-35
 - opcode description, C-34
 - store Ureg, 10-222
- PRFCNT (performance monitor counter)
 - register
 - load/transfer execution latency, 8-76, 10-220, 10-230
- PRFM (performance monitor mask)
 - register
 - load/transfer execution latency, 8-76, 10-220, 10-230
- prime the DAB, 7-26
- priority access, 9-33
- processor ID, 9-5
- PROCESSOR { } linker command, 9-55
- program
 - locking data or code into the cache, 9-29
- program counter (PC), *See* PC (program counter)
- program fetch, *See* sequencer
- program flow, 8-1, 8-4, 8-5
 - sequencer instructions, 10-232
- program label, 8-17, 8-19, 8-97, 10-234
 - address conversion, 8-17
 - immediate extension, 8-20
- program sequencer, *See* sequencer
- PR (parallel result) register, 2-4, 2-5, 3-4, 3-20, 10-30, 10-33, 10-35
 - usage, 3-20
- PUTBITS instruction, 2-5, 6-3, 6-12, 6-13, 6-15, 6-24, 10-182, 10-184, A-10
 - BFOTMP register, 6-5
 - BFP6 and Len7 fields, 6-13
- PUTBITS instruction *(continued)*
 - example, 6-15, 10-185
 - opcode description, C-21
 - restrictions, 1-41
- Q**
 - Q (quad word) operator, 7-17, 7-18, 7-48, 10-222, 10-223, 10-225
 - quad aligned registers, 1-10, 7-19
 - quad instruction execution, 1-17
 - quad (Rmq) register, 1-25, 2-9, 2-20
 - quad word, 1-11, 1-15
 - 32-, 64-, or 128-bit compaction, 10-46
 - accesses, 1-21, 1-35, 7-2, 7-3, 7-17, 7-26, 7-48
 - address, 8-50
 - boundary, 1-49, 7-26, 8-49
 - data, 7-17
 - fetches, 8-5
 - in memory, BTB, 8-49
 - move, 7-2
 - unaligned, 8-40
- R**
 - $\overline{\text{RAS}}$ (row address select) pin, 1-5
 - $\overline{\text{RD}}$ (read) pin, 1-5
 - RDS (interrupt reduce) instruction, 8-19, 8-32, 8-99, 10-240, A-14
 - example, 10-240
 - ISR usage, 8-34
 - read buffer, 9-3, 9-7, 9-10
 - hit, 9-11, 9-12, 9-45
 - miss, 9-45
 - read embedded DRAM, data flow, 9-17
 - read-modify-write (RMW), 6-5
 - read transaction miss, 9-42
 - read/write normal, merged, and broadcast
 - accesses, 7-17
 - real data, 2-7, 4-3, 7-3

INDEX

- real numbers, [5-4](#)
- real result, [5-4](#)
- reciprocal, *See* RECIPS instruction
- reciprocal square root, [3-3](#)
- reciprocal square root, *See* RSQRTS instruction
- RECIPS (reciprocal) instruction, [3-2](#), [3-26](#), [10-85](#), [A-5](#)
 - example, [10-86](#)
 - opcode description, [C-11](#)
- reduce interrupt to subroutine, *See* RDS instruction
- reference sequence, correlation, [4-30](#)
- refresh embedded DRAM, [9-19](#)
- refresh rate, embedded DRAM, [9-60](#)
 - example, [9-54](#)
 - frequency example, [9-60](#)
 - selections, [9-60](#)
- refresh rate (CASTAT_REFCNTR) bits, [9-23](#)
- register cycle pipe stage, [8-6](#)
 - operations, [9-36](#)
- register file, [1-2](#), [1-4](#), [1-10](#), [1-20](#), [2-1](#), [2-5](#), [3-4](#), [G-4](#)
 - data flow, [1-40](#), [2-2](#)
- register groups, [1-35](#), [1-46](#)
- register relative, [1-18](#)
- registers
 - alternate, [1-18](#)
 - circular buffer addressing, [7-5](#)
 - compute block, [2-1](#)
 - control and status, [1-20](#)
 - dependency, [8-78](#)
 - dependency check, [1-16](#), [1-18](#)
 - double selection, [2-13](#)
 - Dreg (data), [2-3](#)
 - dual aligned, [7-19](#)
 - for addresses, [1-16](#)
 - memory-mapped, [2-2](#)
 - registers *(continued)*
 - name syntax, [1-25](#), [2-1](#), [2-7](#), [2-12](#)
 - quad aligned, [7-19](#)
 - saving and restoring, [1-18](#)
 - SIMD, [1-44](#)
 - single, double, quad, [1-10](#), [1-11](#), [2-20](#)
 - Ureg (universal), [2-3](#)
 - usage, [1-20](#), [1-38](#)
 - width selection, [2-6](#), [2-9](#)
- regular replacement, cache, [9-28](#)
- related documents, [xxvii](#)
- relative addresses, [1-18](#)
- replacement, [9-28](#)
 - cache, regular and copyback, [9-28](#)
 - cache and locked ways, [9-29](#)
 - cache copyback, [9-28](#)
 - LRU page, [9-42](#)
 - regular cache, [9-28](#)
- replacement policy, cache, [9-11](#)
- reset, software, [B-39](#)
- resource conflict, [8-64](#)
 - pipeline effects, [8-64](#)
- resources, [1-28](#)
 - base, [1-29](#)
 - destination, [1-29](#)
 - source, [1-30](#)
- restore context, [7-3](#)
- restrictions, [1-28](#), [1-38](#)
 - access, [1-46](#)
 - alignment, [1-18](#), [7-19](#)
 - branch instruction placement, [1-49](#)
 - broadcast load, [1-44](#)
 - compute block, [1-39](#)
 - dynamic versus static, [1-28](#)
 - IALU instructions, [1-43](#)
 - immediate extension, [1-48](#)
 - instruction combination, [1-27](#)
 - load, [1-38](#), [1-44](#), [1-46](#)
 - memory access, [1-43](#)
 - parallelism rules, [1-27](#)

- restrictions *(continued)*
 - results usage, parallel instructions, 8-38
 - sequencer instructions, 1-16, 1-48
 - shifter instructions, 1-41, 6-3
 - store, 1-38
 - transfer, 1-44
- result
 - availability, 1-16
 - dependency, 1-27, 8-78
 - parallel example, 3-20
 - real/imaginary, 5-4
 - shifter, 6-5
- result flag, 4-5, 4-43
 - See also* ALU, CLU, multiplier, shifter, or IALU status
- results
 - availability order, 8-34
 - bus usage, compute blocks, 1-41
 - dependency stall, 8-38
 - format selection, 5-6, 5-7
 - instruction parallelism guidelines, 8-38
 - max/min signed, max unsigned numbers, 10-3
 - multiplier, 5-3, 5-4, 5-8, 5-9
 - multiplier, rounding, 5-16
 - multiply-accumulate latency, 8-37
 - parallel, 3-4
 - saturated, 5-14
 - sign bit, IALU, 7-12
 - signed/unsigned saturation, 10-3, 10-6, 10-21
 - transfer MR register, example, 10-155, 10-156, 10-157, 10-158
 - unpredictable, 1-28, 8-38
 - unpredictable, BTB, 8-52
- results availability, 8-34
- RETI (return from interrupt) instruction, 1-17, 8-5, 8-19, 8-99, 10-238, A-14
 - example, 10-239
 - ISR usage, 8-34
- RETI (return from interrupt) register, 1-33, 1-34, 1-35, 8-86, B-13
 - BTB usage, 8-53
 - ISR usage, 8-33
 - load execution latency, 8-76
- RETS (return from subroutine) register, 1-33, 1-34, 1-35, B-13
- return addresses, 1-18
- returns zero, 3-10
- reusable interrupts, 8-32
- reverse carry addition, 1-14
- right rotate, 10-217
- rotate, 1-13, 10-174
 - See also* ROT, ROTR, or ROTL, ASHIFTR, and LSHIFTR instructions
 - bit field, 6-1
 - example, 6-21
 - IALU, left, 7-49, 10-217, A-11
 - IALU, right, 7-43, 7-49, 10-217, A-11
- ROTL (rotate left) instruction, 7-49, 10-217, A-11
 - example, 10-217
 - opcode description, C-29
- ROT (rotate) instruction, 6-24, 10-174, A-9
 - example, 6-21, 10-175
 - opcode description, C-20, C-21
- ROTR (rotate right) instruction, 7-43, 7-49, 10-217, A-11
 - example, 10-217
 - opcode description, C-29
- round bit, 5-16
- rounding methods, 2-18, 3-9, 5-17
 - IEEE 754 Standard, 5-15
- round-to- \pm infinity, 2-18
- round-to-nearest, 2-18, 3-9, 5-15
 - even, 5-17
 - multiplier, 5-16
- round-towards-zero, 2-18, 3-9, 5-15

INDEX

RSQRTS (reciprocal square root)
instruction, 3-3, 3-26, 10-87, A-5
example, 10-88
opcode description, C-11
RST_IN (reset input) pin, 1-5
RST_OUT (reset output) pin, 1-5
RTI (return from interrupt) instruction,
1-17, 8-19, 8-99, 10-238, A-14
example, 10-239
ISR usage, 8-33
runtime conditions, 1-28

S

sample bias, 3-10
saturated results, 10-3, 10-6, 10-21
saturation, 3-6, 5-11, 5-13, 5-14
arithmetic, 3-8
enable, 1-25
save context, 7-3
S-bus, 9-2
cacheability, 9-64
scalability and multiprocessing, 1-22
SCALB (scale) instruction, 3-2, 3-26,
10-81, A-5
example, 10-82
opcode description, C-11
scaling, 3-2
scaling factor, block floating-point, *See*
BKFTP instruction
scaling factor identification, 6-1
SCLKRAT2-0 (clock ratio select) pins, 1-5
SCLK (system clock) pin, 1-5, 8-40
SCLK_VREF (clock voltage reference) pin,
1-5
SDA10 (SDRAM address bit 10) pin, 1-5
SDAB (short DAB) option, 1-35, 7-19,
7-26, 7-29, 7-50, 10-223, A-12, C-35,
C-36
resources, 1-35
SDCKE (SDRAM clock enable) pin, 1-5

SDRCON (SDRAM configuration)
register, 1-35
SDWE (SDRAM write enable) pin, 1-5
.SECTION assembler directive
immediate extension usage, 8-20
program label usage, 8-17
SECTIONS { } linker command, 9-55
semaphores, 1-22
semi-colon delimiters, 1-23, 1-25
SEQ (shifter equal to zero) condition, 6-20,
8-9
See also shifter conditions
condition code, C-43
sequencer, 1-2, 1-4, 1-15, 1-19, 8-1 to
8-100
conditions, 8-9, 8-98, C-45
data flow, 8-3, 8-4
examples, 8-92
immediate extensions, 1-16, C-37, C-46
instruction encoding, C-38 to C-47
opcodes, C-39, C-40
operations, 8-1, 8-8
pipeline, 1-16
quick reference, A-14
register groups, 1-46
registers, B-13, B-15, B-16, B-32
resources, 1-36
restrictions, 1-48
status, *See* SQSTAT register
sequencer instructions, 10-232 to 10-250,
C-26, C-38, C-40
conditional, 8-15
dependency stall, 8-69, 8-74, 8-75
examples, 8-92
resources, 1-36
restrictions, 1-16, 1-23
summary, 8-97
sequential access, 9-46
sequential access policy, 9-47, 9-48

- SE (sign extend) option, 6-11, 6-13, 6-17, 6-18
- set, cache, 9-11, 9-13, 9-25, 9-26
- set bits, 1-13, 6-1, 6-8, 6-24, 10-188, A-10, C-23
- sets in BTB, 8-44
- SFREG (static flags) register, 1-35, 3-16, 5-23, 6-20, 7-14, 8-18, 10-243
 - See also* static condition flag
 - load/transfer execution latency, 8-72, 8-73, 8-74
 - store/transfer dependency stall, 8-73
- SF_x (compute blocks static flag) conditions, 5-23
- SF_x (compute block static flag) conditions, 3-16, 5-23, 6-20, 8-9, 8-100, 10-243, A-15
 - dependency stall, 8-73, 8-74
 - loading, 10-243
 - opcodes, C-43
- shared memory, 1-5
- shift, DAB accesses, 7-30
- shift, *See* ASHIFT, LSHIFT, ASHIFTR, and LSHIFTR instructions
- shift bit field, 6-1
- shifter, 1-2, 1-4, 1-13, 6-1 to 6-25
 - arithmetic shift operation, 6-7
 - bit field manipulation operations, 6-9
 - bit manipulation operations, 6-8
 - conditional instructions, 8-14
 - condition codes, C-43
 - conditions, 6-19, 8-98
 - data flow, 6-2
 - data types, 6-3
 - examples, 6-20
 - full-scale shifts, 6-5
 - instruction parallelism, 1-41
 - logical shift operations, 6-6
 - masked bits, 6-5
 - operations, 6-4
- shifter *(continued)*
 - options, 6-17
 - quick reference, A-9
 - read-modify-write (RMW), 6-5
 - registers, 2-1
 - restrictions, 1-41
 - results bus usage, 1-41
 - status, 6-13, 6-18, 6-19, 6-20, B-2
 - status bits, B-2
- shifter instructions, 10-170 to 10-194, C-19, C-20, C-22, C-23
 - dependency stall, 8-66, 8-68, 8-69, 8-74, 8-75
 - execution, 8-37
 - execution latency, 8-68, 8-69
 - opcodes, C-23, C-25
 - resources, 1-36, 1-37
 - summary, 6-22
- shift magnitude, 6-5
- short word, 1-11, 2-22, 2-23, 5-3, 6-3
 - 32-, 64-, or 128-bit compaction, 10-46
 - 8- or 16-bit expansion, 10-41
 - 8- or 16-bit transpose, 10-50
 - accumulation, 5-19
 - add example, 3-17
 - complex multiply example, 5-25
 - multiplication, 5-4
 - multiply example, 5-24, 5-25
 - SDAB accesses, 7-29
 - shifter result, 6-5
 - sideways sum example, 3-17
- short word data, 2-10, 2-14, 2-15, 2-20
- sideways, 3-1
- sideways SUM instruction, 3-1
 - execution latency, 8-68
- sign, copy, *See* COPYSIGN instruction
- sign bit, 2-17, 2-19
 - IALU result, 7-12
- signed data, 3-7, 5-12, 7-8

INDEX

- signed format
 - fractional, [2-21](#), [2-23](#), [2-24](#), [2-25](#)
 - integer, [2-21](#), [2-22](#), [2-24](#), [2-25](#)
 - maximum and minimum, [3-8](#)
- signed-magnitude, [5-12](#)
- signed operation, [3-6](#), [5-11](#), [7-7](#)
- signed results, [10-3](#), [10-6](#), [10-21](#)
- signed saturation, [3-8](#)
- SIMD (single-instruction, multiple-data),
 - [1-9](#), [1-25](#), [1-33](#), [2-8](#)
 - execution, [2-8](#)
 - interleaved data, [7-30](#)
 - latency, [2-8](#)
 - operation selection, [1-39](#)
 - registers, [10-229](#), [B-7](#)
- simulator
 - memory access penalty analysis, [9-45](#)
- single cycle, dual data accesses, [9-4](#)
- single-instruction, multiple-data (SIMD),
 - See* SIMD
- single-instruction, single-data (SISD), *See* SISD
- single-precision data, [2-16](#), [5-3](#)
- single processor system, [1-5](#)
- single (Rm) registers, [1-10](#), [1-11](#), [1-25](#), [2-9](#),
[2-20](#), [5-6](#), [5-7](#)
- single words, [1-15](#)
- SISD (single-instruction, single-data),
 - [1-25](#), [2-8](#)
 - compute block selection, [1-39](#)
 - latency, [2-8](#)
- size and alignment of data, [7-18](#)
- SLT (shifter less than zero) condition, [6-20](#),
[8-9](#)
 - See also* shifter conditions
 - condition code, [C-43](#)
- SNGL (reduce to single, floating-point)
 - instruction, [3-3](#), [3-25](#), [10-75](#), [A-5](#)
 - example, [10-76](#)
 - opcode description, [C-11](#)
- SN (shifter negative) bit, [2-5](#)
 - See also* shifter status
 - updated by, [6-19](#)
- SOC (system-on-chip) interface, [1-3](#), [9-2](#)
 - clock rate, [1-20](#)
 - registers, memory map, [9-5](#)
 - Ureg load restriction, [1-35](#)
 - Ureg transfer restriction, [1-35](#)
- software exceptions, [2-4](#)
- software reset, [B-39](#)
- source resources, [1-30](#)
- SQCTL (sequencer control) register, [1-35](#),
[8-9](#), [8-12](#), [8-13](#), [8-27](#), [B-15](#)
 - example global interrupt disable/enable,
[8-91](#)
 - load/transfer execution latency, [8-75](#)
 - write stall, [8-12](#)
- SQSTAT (sequencer status) register, [1-35](#),
[8-9](#), [8-10](#), [8-13](#)
 - store execution latency, [8-75](#)
- square bracket in syntax, [1-25](#)
- square root, [3-2](#)
- S (saturation) option, [3-6](#), [3-8](#), [5-11](#), [5-13](#)
- S (short word data) option, [2-10](#)
- S (SOC interface) bus, [1-3](#), [1-19](#), [9-1](#)
 - stall, [8-80](#)
- stack, PC, [1-18](#)

- stall, 1-16, 1-18, 1-27
 - See also* penalties
 - bus request, 8-80
 - compute block dependency, 8-65, 8-83
 - display in pipeline timing, 9-38
 - external memory dependency, 8-84
 - IALU load dependency, 8-84, 8-85
 - incorrect branch prediction, 8-48
 - instruction line, 1-24
 - instruction pipeline, 8-38
 - memory access programming guidelines, 9-46
 - memory pipeline, 8-8, 9-35
 - pipeline, 8-78, 8-85
 - pipeline effects, 8-64
 - pipeline example, 8-78, 8-82, 8-83, 8-85, 8-86, 8-88, 8-90
- Stall_Mode (CACHE_INIT command)
 - parameter, 9-71, 9-75, 9-80
- Start_Address (CACHE_INIT command)
 - parameter, 9-71
- Start_Index (CACMD_CB command)
 - parameter, 9-75
- Start_Index (CACMD_INV command)
 - parameter, 9-79
- static condition flag
 - See also* SFREG register
 - load execution latency, 8-73, 8-74
 - loading conditions, 10-243
- static flags, 7-14, 8-100, 10-243, A-15
 - compute block, 3-16, 5-23, 6-20
 - IALU, 7-14
- static flag (SFREG) register, 3-16, 5-23, 6-20, 7-14
- static flag (SFx) conditions
 - load dependency stall, 8-73, 8-74
- static restrictions, 1-28
- Static Superscalar, 1-8, 1-17, 1-27
- status
 - ALU (arithmetic logic unit), 3-11, B-2
 - CLU (communications logic unit), 4-42
 - compute block, 8-54
 - IALU, 7-10
 - multiplier, 5-20, B-2
 - shifter, 6-18, B-2
 - update, 8-8
- status registers, 1-20
- sticky status, 8-37
 - ALU (arithmetic logic unit), 3-12, B-2
 - CLU (communications logic unit), 4-42
 - multiplier, 5-21, B-3
 - pipeline, 8-37
- store, 1-14, 1-18, 1-20, 1-29
 - bus usage, 8-80
 - definition, G-13
 - example, 3-20
 - execution latency, 8-68
 - IALU operations, 7-1
 - normal, merged, and broadcast accesses, 7-17
 - opcodes, C-32, C-33, C-36
 - register, C-32, C-33, C-36
 - resources, 1-32, 1-33
 - restrictions, 1-38, 1-45
- store Dreg register, 7-51, 10-227
 - dependency stall, 8-66, 8-69
- store FLAGREG register, dependency stall, 8-75
- store LCx register, dependency stall, 8-74
- store memory system register, dependency stall, 8-69
- store MR register, 5-25
- store SFREG register, dependency stall, 8-73
- store SQSTAT register, execution latency, 8-75
- store trellis/trellis history registers, 4-46, 4-47, 10-99, A-6

INDEX

- store Ureg (universal) register, [7-42](#), [7-50](#), [10-222](#)
 - execution latency, [8-70](#)
 - store X/YSTAT register, [6-9](#)
 - stray pointers, [1-28](#)
 - sub-array of memory, [9-13](#), [9-14](#), [9-15](#), [9-17](#)
 - interrupted access, [9-18](#)
 - subroutine, [8-4](#)
 - flow, [8-5](#)
 - subtract/add (dual operation) instruction
 - example, [10-37](#)
 - opcode description, [C-6](#)
 - subtract/add (floating-point, dual operation) instruction
 - example, [10-95](#)
 - opcode description, [C-12](#)
 - subtract (floating-point) instruction
 - example, [10-57](#)
 - subtract instruction
 - example, [3-17](#), [10-4](#)
 - opcode description, [C-5](#)
 - subtract (integer) example, [7-8](#)
 - subtract (integer) instruction
 - example, [10-202](#)
 - execution latency, [8-71](#)
 - opcode description, [C-28](#)
 - subtract with borrow (floating-point) instruction
 - opcode description, [C-10](#)
 - subtract with borrow instruction
 - example, [10-8](#)
 - opcode description, [C-6](#)
 - subtract with borrow (integer) instruction
 - example, [10-204](#)
 - opcode description, [C-28](#)
 - subtract with divide by two (floating-point) instruction
 - example, [10-59](#)
 - subtract with divide by two instruction
 - opcode description, [C-5](#)
 - subtract with divide by two (integer) instruction
 - example, [10-206](#)
 - opcode description, [C-28](#)
 - SUM (sideways) instruction, [2-5](#), [3-1](#), [3-4](#), [3-19](#), [3-20](#), [3-24](#), [10-27](#), [10-35](#), [A-3](#)
 - example, [3-17](#), [3-19](#), [10-28](#), [10-36](#)
 - opcode description, [C-8](#)
 - SUM (sideways with parallel accumulate) instruction
 - opcode description, [C-9](#)
 - .S unit, *See* shifter
 - S unit, *See* shifter
 - Super Harvard architecture, [9-1](#)
 - supervisor mode, [B-15](#)
 - ISR usage, [8-33](#)
 - support, technical or customer, [xxiv](#)
 - SWRST (software reset) bit, [8-13](#), [B-39](#)
 - SYSCON (system configuration) register, [1-33](#), [1-34](#), [1-35](#)
 - SYSTAT (system status) register, [1-35](#)
 - system
 - development enhancements, [1-6](#)
 - multiprocessor, [1-6](#)
 - single processor, [1-5](#)
 - system clock (SCLK), [8-40](#)
 - system-on-chip (SOC) interface, [1-1](#), [9-3](#)
 - clock rate, [1-20](#)
 - SZ (shifter zero) bit, [2-5](#)
 - See also* shifter status
 - updated by, [6-19](#)
- ## T
- tag bits, cache, [9-25](#), [9-26](#)
 - tag field, memory transaction, [9-27](#)
 - T (ALU truncate) option, [3-6](#)
 - target address, branch, [1-17](#)
 - technical support, [xxiv](#)

- test, bit, 1-13, 6-1
 - example, 6-21
- THR (trellis history) registers, 2-4, 2-5, 4-1, 4-4
 - dependency stall, 8-68
 - load, opcode description, C-25
 - load execution latency, 8-67
 - load/shift order, 4-24, 4-35
 - transfer, opcode description, C-12
- TigerSHARC architecture, 9-1
- timer run (TMRxRN) bits, 8-28, 8-29, 8-33, B-38
- TIMERxH (timer high priority interrupt) bits, 8-30
- TIMERxL (timer low priority interrupt) bits, 8-31
- TIMERx (timer) registers, 1-35
- TMAX (trellis maximum) instruction, 4-2, 4-3, 4-9, 4-46, 10-97, A-6
 - data flow example, 4-11, 4-12, 4-14, 4-15, 4-17
 - dependency stall, 8-67
 - execution latency, 8-67
 - function, 4-6
 - opcode description, C-13
 - table, 4-6
- TMAX (trellis maximum) option, 4-41
- TMROE (timer enable) pin, 1-5
- TMRIX (timer input) registers, 1-35
- TMRxRN (timer run) bits, 8-28, 8-29, 8-33, B-38
- toggle, bit, 1-13, 6-1
 - See also* BTGL instruction
 - example, 6-21
- torroid configuration, 1-22
- trace back routine, 4-9
- transaction
 - penalty-free transactions, 9-44
 - read miss, 9-42
 - stall, 8-81
- transfer, 1-14, 1-20, 1-26, 1-29
 - bus usage, 8-81
 - definition, G-14
 - Dreg register, dependency stall, 8-69
 - merged access restrictions, 1-44
 - resources, 1-31
 - Ureg register, 7-51
 - exceptions, 10-230
 - WPxH/L registers, execution latency, 8-76
- transfer BFOTMP register
 - example, 10-194
 - opcode description, C-24
- transfer CACMDx register, restrictions, 9-20
- transfer CCAIRx register, restrictions, 9-20
- transfer CCNTx registers, execution latency, 8-76, 10-220, 10-230
- transfer CMCTL register, opcode description, C-13
- transfer Dreg register, dependency stall, 8-66
- transfer J30-0 (data) register, execution latency, 8-71, 8-72
- transfer J31/JSTAT (status) register, execution latency, 8-71, 8-72
- transfer JB3-0 (base) register, execution latency, 8-71, 8-72
- transfer K30-0 (data) register, execution latency, 8-71, 8-72
- transfer K31/KSTAT (status) register, execution latency, 8-71, 8-72
- transfer KB3-0 (base) register, execution latency, 8-71, 8-72
- transfer merged access restrictions, 1-44

INDEX

- transfer MR register
 - dependency stall, 8-68
 - example, 10-155, 10-156, 10-157, 10-158
 - execution latency, 8-68
 - opcode description, C-18
- transfer PRFCNT register, execution latency, 8-76, 10-220, 10-230
- transfer PRFM register, execution latency, 8-76, 10-220, 10-230
- transfer SFREG (static flags) register
 - dependency stall, 8-73
 - execution latency, 8-72, 8-73, 8-74
- transfer SOC Ureg register, execution latency, 8-72
- transfer SQCTL register, execution latency, 8-75
- transfer THR (trellis history) register
 - dependency stall, 8-68
- transfer THR trellis history) registers
 - opcode description, C-12
- transfer TRCB31-0 registers, execution latency, 8-76, 10-220, 10-230
- transfer TRCBMASK register, execution latency, 8-76, 10-220, 10-230
- transfer TRCBPTR register, execution latency, 8-76, 10-220, 10-230
- transfer TRCBVAL register, execution latency, 8-76, 10-220, 10-230
- transfer TR (trellis) registers
 - dependency stall, 8-68
 - opcode description, C-12
- transfer Ureg register, 7-42, 10-229
 - dependency stall, 8-72
 - example, 10-231
 - exception, 10-230
 - parallelism restriction, 1-35
- transfer WPxCTL registers, execution latency, 8-76, 10-220, 10-230
- transfer WPxH/L registers, execution latency, 10-220, 10-230
- transfer WPxSTAT registers, execution latency, 8-76, 10-220, 10-230
- transfer XSTAT register
 - opcode description, C-24
- transfer X/YSTAT registers
 - example, 10-195
 - execution latency, 8-66, 8-73
- transfer YSTAT register
 - opcode description, C-24
- TRAP instruction, 8-9, 8-14, 8-99, 8-100, 10-241, 10-242, 10-244, 10-245, 10-246, 10-247, 10-248, 10-249, 10-250, A-15, C-46, C-47
- TRCB31-0 (trace buffer) registers
 - load/transfer execution latency, 8-76, 10-220, 10-230
- TRCBMASK (trace buffer mask) register
 - load/transfer execution latency, 8-76, 10-220, 10-230
- TRCBPTR (trace buffer pointer) register
 - load/transfer execution latency, 8-76, 10-220, 10-230
- TRCBVAL (trace buffer value) register
 - load/transfer execution latency, 8-76, 10-220, 10-230
- tree configuration, 1-22
- trellis
 - add/compare/select, 4-3
 - calculation, 4-3
 - diagram, 4-8
 - maximum, *See* TMAX instruction
 - overflow, 2-5
- tristate versus three-state, G-14
- TROV (trellis overflow) bit, 2-5
 - See also* CLU status
 - updated by, 4-42

- TRSOV (trellis overflow, sticky) bit, 2-5
See also CLU status
 updated by, 4-42
- TR (trellis) registers, 2-4, 2-5, 4-1, 4-4
 dependency stall, 8-68
 load, execution latency, 8-66, 8-67
 load, opcode description, C-25
 transfer, opcode description, C-12
- TRUE condition, 8-21, 8-54
 condition code, C-45
- truncation, 3-6, 3-9
See also rounding and T option
 enable, 1-25
- T (truncation) option, 3-9, 5-11, 5-15, 5-17
- turbo code algorithm, 1-12, 4-2, 4-6, 4-43
- turbo decoding algorithm, 4-8, 4-9
- two's complement instruction, 10-14
 example, 10-14
 opcode description, C-6
- types of instructions, 1-28
- ## U
- UEN (exception enable on underflow) bit, 2-4, 3-12, 5-21, 5-22, B-2
See also X/YSTAT register
- unaligned DAB accesses, 7-27
- unaligned quad words, 8-40
- unconditional branches, 8-21
- unconditional execution, 1-26
- underflow
See also XSTAT register and ALU, multiplier, shifter, or IALU status
 exception, 2-18
 multiplier, 5-21
 saturation, 10-3
- unified address space, 9-1
 memory map, 9-5
- universal (Ureg) registers, 1-20, 2-3
 dependency stall, 8-72
 load, 7-42, 7-49
 load execution latency, 8-75
 memory map, 9-5
 parallelism restriction, 1-35
 register file, 2-6
 SOC execution latency, 8-72
 store, 7-42, 7-50, 10-222
 transfer, 7-51, 10-229
 transfer merged access restrictions, 1-44
- unmasked interrupt, 8-26
- unpredictable results, 8-38, 8-52
- unsigned data, 3-7, 5-12, 7-8
- unsigned format, 3-8
 fractional, 2-22, 2-23, 2-25, 2-26
 integer, 2-22, 2-23, 2-24
- unsigned operation, 3-6, 7-7
- unsigned results, 10-3, 10-4, 10-6, 10-7, 10-21
- unsigned saturation, 3-8
- Ureg (universal) registers, 1-20, 1-33, 2-3, 2-5, 7-18
 dependency stall, 8-72
 execution latency, 8-70, 8-75
 load, 7-42, 7-49
 memory map, 9-5
 merged access restrictions, 1-44
 parallelism restriction, 1-35
 SOC execution latency, 8-72
 store, 7-42, 7-50, 10-222
 transfer, 7-51, 10-229
- U (unsigned) option, 3-6, 3-7, 5-11, 5-12, 7-7, 7-8
- ## V
- valid bits, cache, 9-25, 9-26, 9-28
- value, modified, 7-9
- values, literal, 1-26
- vector interrupt, 8-5

INDEX

violation, static, 1-28
VIRPT (vector interrupt) bit, 8-30
Viterbi, 3-1
Viterbi algorithm, 1-12, 4-8, 4-43
 metrics, 4-9
Viterbi maximum, 3-1
 See also VMAX instruction
 algorithm, 10-18
Viterbi minimum
 See also VMIN instruction
 algorithm, 10-19
VMAX (Viterbi maximum) instruction,
 2-5, 3-1, 3-4, 3-20
 example, 10-19
 opcode description, C-6
VMIN (Viterbi minimum) instruction,
 2-5, 3-4, 3-20, 3-23, 10-18, A-3
 example, 10-19
 opcode description, C-6
Von Neumann architecture, 9-1
VREF (voltage reference) pin, 1-5

W

wait cycle in memory pipe, 8-6
wait cycle pipe stage
 operations, 9-36
way, cache, 9-11, 9-25, 9-26
ways, BTB, 8-44
ways, cache
 replacement, 9-29
words, single-, dual-, or quad-, 1-15
word size, 1-25, 2-9
word sizes
 and alignment, 1-11
WPxCTL (watchpoint control) registers
 load/transfer execution latency, 8-76,
 10-220, 10-230
WPxH/L (watchpoint address) registers
 load/transfer execution latency, 8-76,
 10-220, 10-230

WPxSTAT (watchpoint status) registers
 load/transfer execution latency, 8-76,
 10-220, 10-230
wrap around, circular buffer, 7-33
WRH/WRL (write) pins, 1-5
write allocate policy, 9-32
write embedded DRAM, data flow, 9-17
write policy, cache, 9-11

X

X (ALU ABS extend) option, 3-6
XCORRS (cross correlation) instruction,
 2-5, 4-3, 4-4, 4-47, 10-103, A-6
 bit placement, 4-32, 4-33
 data flow example, 4-32, 4-33, 4-34,
 4-35
 dependency stall, 8-66, 8-67
 equation, 4-31
 execution latency, 8-67
 function, 4-30
 opcode description, C-14
X (extended output range) option, 3-9
XOR (exclusive OR, integer) instruction
 example, 10-214
 opcode description, C-28
XOR (exclusive OR) instruction, 3-2
 example, 10-40
 opcode description, C-6
XSFCx (X compute block static flag) bits,
 3-16, 5-23, 6-20, 8-18
XSTAT (X compute block status) register,
 2-4, 2-5, 6-9, 10-195
 load, opcode description, C-24
 load execution latency, 8-74
 load/transfer restriction, 3-12
 restrictions, 1-41
 transfer, opcode description, C-24
 transfer execution latency, 8-66, 8-73
X/YSTAT (status register) load/store
 instruction, 1-36, 1-37

Y

YSCFx (Y compute block static flag) bits,
3-16, 5-23, 6-20, 8-18

YSTAT (Y compute block status) register,
2-4, 2-5, 6-9, 10-195

load, opcode description, C-24

load execution latency, 8-74

load/transfer restriction, 3-12

restrictions, 1-41

transfer, opcode description, C-24

transfer execution latency, 8-66, 8-73

Z

Z (ALU MAX/MIN zero return) option,
3-7, 3-10, 10-15

zero, representation, 2-17

zero-overhead loop, 8-22
example, 8-93

zeros, leading, 6-1

zeros/ones extract example, 6-21

ZF (zero-fill on deposited field) option,
6-11, 6-17, 6-18

INDEX