ADSP-219x DSP Instruction Set Reference

Revision 2.0, December 2005

Part Number
82-000390-07

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

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

CONTENTS

PREFACE

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

Data Sheets xxii
Conventions xxiii

INSTRUCTION SET SUMMARY

Core Registers Summary 1-2
Arithmetic Status (ASTAT) Register 1-3
Condition Code (CCODE) Register 1-5
Interrupt Control (ICNTL) Register 1-6
Interrupt Mask (IMASK) Register and Interrupt Latch (IRPTL) Register 1-7
Mode Status (MSTAT) Register 1-8
System Status (SSTAT) Register 1-10
Condition Codes Summary 1-11
Instruction Summary 1-12
ALU Instructions 1-14
Multiplier Instructions 1-15
Shifter Instructions 1-16
Data Move Instructions 1-16
Program Flow Instructions 1-18
Multifunction Instructions 1-19

ALU INSTRUCTIONS

ALU Instruction Conventions 2-1
Input Registers 2-1
Output Registers 2-2
Constants 2-2

ALU Mode Control 2-3
ALU Status Flags 2-4
ALU Instruction Reference 2-4
Add/Add with Carry 2-5
Subtract X–Y/Subtract X–Y with Borrow 2-9
Subtract Y–X/Subtract Y–X with Borrow 2-13
Bitwise Logic: AND, OR, XOR 2-16
Bit Manipulation: TSTBIT, SETBIT, CLRBIT, TGLBIT 2-19
Clear: PASS 2-22
Negate: NOT 2-25
Absolute Value: ABS 2-28
Increment 2-31
Decrement 2-34
Divide Primitives: DIVS and DIVQ 2-37
Generate ALU Status Only: NONE 2-46

MAC INSTRUCTIONS

Multiply Instruction Conventions 3-2
MAC Input Registers 3-2
MAC Output Registers 3-2
Data Format Options 3-3
Rounding Modes 3-4
Numeric Format Modes 3-6
Status Flags 3-7
Multiply 3-8

Multiply with Cumulative Add 3-11
Multiply with Cumulative Subtract 3-14
MAC Clear 3-17
MAC Round/Transfer 3-19
MAC Saturate 3-21
Generate MAC Status Only: NONE 3-24

SHIFTER INSTRUCTIONS

Shifter Operation Conventions 4-2
Shifter Registers 4-2
Shifter Instruction Options 4-3
Shifter Status Flags 4-5
Arithmetic Shift 4-6
Arithmetic Shift Immediate 4-8
Logical Shift 4-10
Logical Shift Immediate 4-12
Normalize 4-14
Normalize Immediate 4-17
Exponent Derive 4-20
Exponent (Block) Adjust 4-23
Denormalization 4-26

MULTIFUNCTION INSTRUCTIONS

Order of Execution of Multifunction Operations 5-2
Multifunction Instruction Reference 5-3

Compute with Dual Memory Read 5-4
Dual Memory Read 5-8
Compute with Memory Read 5-11
Compute with Memory Write 5-15
Compute with Register-to-Register Move 5-19

DATA MOVE INSTRUCTIONS

Core Registers 6-2
PX Register 6-3
DAG Registers 6-5
Address Registers 6-5
DAG Memory Page Registers (DMPGx) 6-6
Secondary DAG Registers 6-7
Register Load Latencies 6-9
Data Addressing Methods 6-11
Direct Addressing 6-11
Indirect Addressing 6-12
Circular Data Buffer Addressing 6-14
Bit-Reversed Addressing 6-16
Data Move Instruction Reference 6-21
Register-to-Register Move 6-22
Direct Memory Read/Write—Immediate Address 6-24
Direct Register Load 6-27
Indirect 16-Bit Memory Read/Write—Postmodify 6-30
Indirect 16-Bit Memory Read/Write—Premodify 6-34

Indirect 24-Bit Memory Read/Write—Postmodify 6-38

Indirect 24-Bit Memory Read/Write—Premodify 6-43

Indirect DAG Register Write (Premodify or Postmodify), with DAG Register Move 6-47

Indirect Memory Read/Write—Immediate Postmodify 6-51

Indirect Memory Read/Write—Immediate Premodify 6-54

Indirect 16-Bit Memory Write—Immediate Data 6-57

Indirect 24-Bit Memory Write—Immediate Data 6-59

External I/O Port Read/Write 6-62

System Control Register Read/Write 6-65

Modify Address Register—Indirect 6-68

Modify Address Register—Direct 6-70

PROGRAM FLOW INSTRUCTIONS

Conditions 7-2

Counter-Based Conditions 7-2

CCODE Register 7-3

Mode Control 7-4

Branch Options 7-4

Addressing Branch Targets 7-6

Stacks 7-7

- PC and Status Stack Operation 7-8
- Loop Stacks Operation 7-10

Stack Status Flags 7-12

Interrupts 7-13

- Enabling Interrupts 7-14
- Switching Contexts 7-16
- Nesting Interrupts 7-16

Application Performance 7-17

- Exiting a Loop 7-18
- Using Long Jumps and Calls 7-20
- Effect Latencies 7-22

Program Flow Instruction Reference 7-23

DO UNTIL (PC Relative) 7-24

Direct JUMP (PC Relative) 7-29

CALL (PC Relative) 7-33

JUMP (PC Relative) 7-37

Long Call (LCALL) 7-40

Long Jump (LJUMP) 7-43

Indirect CALL 7-46

Indirect JUMP 7-50

Return from Interrupt (RTI) 7-53

Return from Subroutine (RTS) 7-57

PUSH or POP Stacks 7-61

FLUSH CACHE 7-67

Set Interrupt (SETINT) 7-69

Clear Interrupt (CLRINT) 7-71

NOP 7-73

IDLE 7-74

Mode Control 7-76

INSTRUCTION OPCODES

Opcode Mnemonics 8-1
ALU or Multiplier Function (AMF) Codes 8-6
Condition Codes 8-8
Constant Codes 8-9
Core Register Codes 8-11
Shift Function (SF) Codes 8-12
Index Register and Modify Register Codes 8-13
DMI, DMM, PMI, and PMM Codes 8-14
IREG/MREG Codes 8-15
XOP and YOP Codes 8-15
Opcode Definitions 8-16
Type 1: Compute | DregX«···DM | DregY«···PM 8-17
Type 3: Dreg/Ireg/Mreg «···» DM/PM 8-18
Type 4: Compute | Dreg «···» DM 8-19
Type 6: Dreg «··· Data16 8-20
Type 7: Reg1/2 «··· Data16 8-21
Type 8: Compute | Dreg1 «··· Dreg2 8-22
Type 9: Compute 8-23
Type 9a: Compute 8-26
Type 10: Direct Jump 8-28
Type 10a: Direct Jump/Call 8-29
Type 11: Do ··· Until 8-30

Type 12: Shift | Dreg «···» DM 8-31
Type 14: Shift | Dreg1 «··· Dreg2 8-32
Type 15: Shift Data8 8-33
Type 16: Shift Reg0 8-34
Type 17: Any Reg «···Any Reg 8-35
Type 18: Mode Change 8-36
Type 19: Indirect Jump/Call 8-37
Type 20: Return 8-38
Type 21: Modify DagI 8-39
Type 21a: Modify DagI 8-40
Type 22: DM «··· Data16 8-41
Type 22a: PM «··· Data24 8-42
Type 23: Divide primitive, DIVQ 8-43
Type 24: Divide primitive, DIVS 8-44
Type 25: Saturate 8-45
Type 26:Push/Pop/Cache 8-46
Type 29: Dreg «···» DM 8-47
Type 30: NOP 8-48
Type 31: Idle 8-49
Type 32: Any Reg «···» PM/DM 8-50
Type 32a: DM«···DAG Reg | DAG Reg«···Ireg 8-51
Type 33: Reg3 «··· Data12 8-52
Type 34: Dreg «···» IOreg 8-53
Type 35: Dreg «···»Sreg 8-54

Type 36: Long Jump/Call 8-55
Type 37: Interrupt 8-56

INDEX

PREFACE

Thank you for purchasing and developing systems using ADSP-219x DSPs from Analog Devices.

Purpose of This Manual

The *ADSP-219x DSP Instruction Set Reference* provides assembly syntax information for ADSP-219x DSPs. The syntax descriptions cover instructions that execute within the DSP's processor core (processing elements, program sequencer, and data address generators). For architecture and design information on the DSP, see the *ADSP-219x/2192 DSP Hardware Reference*.

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

This reference presents instruction information organized by the type of the instruction. Instruction types relate to the machine language opcode for the instruction. On this DSP, the opcodes categorize the instructions by the portions of the DSP architecture that execute the instructions. The following chapters cover the different types of instructions.

- "Instruction Set Summary" on page 1-1—This chapter provides a syntax summary of all instructions and describes the conventions that are used on the instruction reference pages.
- "ALU Instructions" on page 2-1—These instruction specify operations that occur in the DSP's ALU.
- "MAC Instructions" on page 3-1—These instructions specify operations that occur in the DSP's Shifter.
- "Shifter Instructions" on page 4-1—These instructions specify operations that occur in the DSP's Shifter.
- "Multifunction Instructions" on page 5-1—These instructions specify parallel, single-cycle operations.
- "Data Move Instructions" on page 6-1—These instructions specify memory and register access operations.
- "Program Flow Instructions" on page 7-1—These instructions specify program sequencer operations.
- "Instruction Opcodes" on page 8-1—This chapter lists the instruction encoding fields for all instructions.

Each of the DSP's instructions is specified in this text. The reference page for an instruction shows the syntax of the instruction, describes its function, gives one or two assembly-language examples, and identifies fields of

its opcode. The instructions are referred to by type, ranging from 1 to 37. These types correspond to the opcodes that ADSP-219x DSPs recognize, but are for reference only and have no bearing on programming.

Some instructions have more than one syntactical form; for example, the instruction "Type 9: Compute" on page 8-23 has many distinct forms.

Many instructions can be conditional. These instructions are prefaced by `IF COND`; for example:

```
If COND compute;
```

In a conditional instruction, the execution of the entire instruction is based on the specified condition.

What's New in This Manual

Revision 2.0 of the *ADSP-219x DSP Instruction Set Reference* corrects all known document errata issues.

This instruction set reference is a companion document to the *ADSP-219x/2192 DSP Hardware Reference* (Rev 1.1, April 2004).

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 `processor.tools.support@analog.com`

- E-mail processor questions to
  `processor.support@analog.com (World wide support)`
  `processor.europe@analog.com (Europe support)`
  `processor.china@analog.com (China support)`
- Phone questions to **1-800-ANALOGD**
- Contact your Analog Devices, Inc. local sales office or authorized distributor
- Send questions by mail to:

```
Analog Devices, Inc.
One Technology Way
P.O. Box 9106
Norwood, MA 02062-9106
USA
```

Supported Processors

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

TigerSHARC® (ADSP-TSxxx) Processors

The name *TigerSHARC* refers to a family of floating-point and fixed-point [8-bit, 16-bit, and 32-bit] processors. VisualDSP++ currently supports the following TigerSHARC families: ADSP-TS101 and ADSP-TS20x.

SHARC® (ADSP-21xxx) Processors

The name *SHARC* refers to a family of high-performance, 32-bit, floating-point processors that can be used in speech, sound, graphics, and imaging applications. VisualDSP++ currently supports the following SHARC families: ADSP-2106x, ADSP-2116x, ADSP-2126x, and ADSP-2136x.

Blackfin® (ADSP-BFxxx) Processors

The name *Blackfin* refers to a family of 16-bit, embedded processors. VisualDSP++ currently supports the following Blackfin families: ADSP-BF53x and ADSP-BF56x.

ADSP-21xx Processors

The ADSP-21xx processors are high-performance 16-bit DSPs for communications, instrumentation, industrial/control, voice/speech, medical and military applications. The family includes the ADSP-218x, ADSP-219x, and mixed-signal products (ADSP-21990, ADSP-21991, and ADSP-21992).

Product Information

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

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

MyAnalog.com

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

Registration

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

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

Processor Product Information

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

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

- E-mail questions or requests for information to
  processor.support@analog.com (World wide support)
  processor.europe@analog.com (Europe support)
  processor.china@analog.com (China support)
- Fax questions or requests for information to
  1-781-461-3010 (North America)
  +49-89-76903-157 (Europe)
- Access the FTP Web site at
  ftp ftp.analog.com (or ftp 137.71.25.69)
  ftp://ftp.analog.com

Related Documents

The following publications that describe the ADSP-219x processor can be ordered from any Analog Devices sales office:

- *ADSP-219x Processor Data Sheet*
- *ADSP-219x/2192 DSP Hardware Reference*

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

- *VisualDSP++ User's Guide*
- *VisualDSP++ C/C++ Compiler and Library Manual*
- *VisualDSP++ Assembler and Preprocessor Manual*
- *VisualDSP++ Linker and Utilities Manual*
- *VisualDSP++ Kernel (VDK) User's Guide*

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

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

Online Technical Documentation

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

Each documentation file type is described as follows.

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

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

Accessing Documentation From VisualDSP++

From the VisualDSP++ environment:

- Access VisualDSP++ online Help from the Help menu's **Contents**, **Search**, and **Index** commands.
- Open online Help from context-sensitive user interface items (toolbar buttons, menu commands, and windows).

Accessing Documentation From Windows

In addition to any shortcuts you may have constructed, there are many ways to open VisualDSP++ online Help or the supplementary documentation from Windows.

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

Using Windows Explorer

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

Using the Windows Start Button

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

Accessing Documentation From the Web

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

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

Printed Manuals

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

VisualDSP++ Documentation Set

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

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

Hardware Tools Manuals

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

Processor Manuals

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

Data Sheets

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

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

Conventions

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

Example	Description
Close command (**File** menu)	Titles in reference sections indicate the location of an item within the VisualDSP++ environment's menu system (for example, the **Close** command appears on the **File** menu).
`{this | that}`	Alternative items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as `this` or `that`. 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 `this` or `that`.
`[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 `this`.
`.SECTION`	Commands, directives, keywords, and feature names are in text with `letter gothic` font.
filename	Non-keyword placeholders appear in text with italic style format.
	Note: For correct operation, ... A Note: provides supplementary information on a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
	Caution: Incorrect device operation may result if ... **Caution:** Device damage may result if ... A Caution: identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word **Caution** appears instead of this symbol.
	Warning: Injury to device users may result if ... A Warning: identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for devices users. In the online version of this book, the word **Warning** appears instead of this symbol.

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

1 INSTRUCTION SET SUMMARY

This chapter provides a summary of the instructions in the ADSP-219x DSP's instruction set. Chapters 2 through 8 describe these instructions in more detail as follows:

- "ALU Instructions" on page 2-1
- "MAC Instructions" on page 3-1
- "Shifter Instructions" on page 4-1
- "Multifunction Instructions" on page 5-1
- "Data Move Instructions" on page 6-1
- "Program Flow Instructions" on page 7-1
- "Instruction Opcodes" on page 8-1

Also, this chapter identifies mnemonics for using DSP registers, bits, and operating conditions. This information appears in the following summaries:

- "Core Registers Summary" on page 1-2
- "Arithmetic Status (ASTAT) Register" on page 1-3
- "Condition Code (CCODE) Register" on page 1-5
- "Interrupt Control (ICNTL) Register" on page 1-6
- "Interrupt Mask (IMASK) Register and Interrupt Latch (IRPTL) Register" on page 1-7

- “Mode Status (MSTAT) Register” on page 1-8
- “System Status (SSTAT) Register” on page 1-10
- “Condition Codes Summary” on page 1-11

For information on instruction reference notation, see “Conventions” on page xxiii.

Core Registers Summary

The DSP has three categories of registers: core registers, system control registers, and I/O registers. Table 1-1 lists and describes the DSP’s core registers. For information about system control and I/O registers, see the *ADSP-219x/2192 DSP Hardware Reference*.

Table 1-1. Core Registers

<table>
<tr><th>Type</th><th>Registers</th><th>Function</th></tr>
<tr><td>ALU data</td><td>AX0, AX1, AY0, AY1, AR, AF</td><td rowspan="3">16-bit data registers (X and Y) provide input for ALU, multiplier, and shifter operations. AR and AF are ALU result and feedback registers. MR and SR are multiplier result and feedback registers. SR also is the shifter results register.
In this text, Dreg denotes unrestricted use of data registers as a data register file, while “ XOP” and “YOP” denote restricted use.
The data registers (except AF, SE, and SB) serve as a register file for unconditional, single-function instructions.</td></tr>
<tr><td>Multiplier data</td><td>MX0, MX1, MY0, MY1, MR0, MR1, MR2</td></tr>
<tr><td>Shifter data</td><td>SI, SE, SB, SR0, SR1, SR2</td></tr>
<tr><td rowspan="3">DAG address</td><td>I0, I1, I2, I3
I4, I5, I6, I7</td><td>DAG1 index registers
DAG2 index registers</td></tr>
<tr><td>M0, M1, M2, M3
M4, M5, M6, M7</td><td>DAG1 modify registers
DAG2 modify registers</td></tr>
<tr><td>L0, L1, L2, L3
L4, L5, L6, L7</td><td>DAG1 length registers
DAG2 length registers</td></tr>
</table>

Table 1-1. Core Registers (Cont'd)

Type	Registers	Function
System control	B0, B1, B2, B3, B4, B5, B6, B7, SYSCTL, CACTL	DAG1 base address registers (B0-3), DAG2 base address registers (B4-7), System control, and Cache control
Program flow	CCODE LPSTACKA LPSTACKP STACKA STACKP	Software condition register Loop PC stack A register, 16 address LSBs Loop PC stack P register, 8 address MSBs PC stack A register, 16 address LSBs PC stack P register, 8 address MSBs
Interrupt	ICNTL IMASK IRPTL	Interrupt control register Interrupt mask register Interrupt latch register
Status	ASTAT MSTAT SSTAT (read-only)	Arithmetic status flags Mode control and status flags System status
Page	DMPG1 DMPG2 IJPG IOPG	DAG1 page register, 8 address MSBs DAG2 page register, 8 address MSBs Indirect jump page register, 8 address MSBs I/O page register, 8 address MSBs
Bus exchange	PX	Holds eight LSBs of 24-bit memory data for transfers between memory and data registers only.
Shifter	SE SB	Shifter exponent register Shifter block exponent register

Arithmetic Status (ASTAT) Register

The DSP updates the status bits in ASTAT, indicating the status of the most recent ALU, multiplier, or shifter operation.

Table 1-2. ASTAT Register Bit Definitions

Bit	Name	Description
0	AZ	ALU result zero. Logical NOR of all bits written to the ALU result register (AR) or ALU feedback register (AF). 0 =ALU output ≠ 0 1 =ALU output = 0
1	AN	ALU result negative. Sign of the value written to the ALU result register (AR) or ALU feedback register (AF). 0 =ALU output positive (+) 1 =ALU output negative (–)
2	AV	ALU result overflow. 0 =No overflow 1 =Overflow
3	AC	ALU result carry. 0 =No carry 1 =Carry
4	AS	ALU *x* input sign. Sign bit of the ALU *x*-input operand; set by the ABS instruction only. 0 =Positive (+) 1 =Negative (–)
5	AQ	ALU quotient. Sign of the resulting quotient; set by the DIVS or DIVQ instructions. 0 =Positive (+) 1 =Negative (–)
6	MV	Multiplier overflow. Records overflow/underflow condition for MR result register. 0 =No overflow or underflow 1 =Overflow or underflow
7	SS	Shifter input sign. Sign of the shifter input operand. 0 =Positive (+) 1 =Negative (–)
8	SV	Shifter overflow. Records overflow/underflow condition for SR result register. 0 =No overflow or underflow 1 =Overflow or underflow

Condition Code (CCODE) Register

Using the CCODE register (shown in Table 1-3), conditional instructions may base execution on a comparison of the CCODE value (user-selected) and the SWCOND condition (DSP status). The CCODE register holds a value between 0x0 and 0xF, which the instruction tests against when the conditional instruction uses SWCOND or NOT SWCOND. Note that the CCODE register has a one-cycle effect latency.

Table 1-3. CCODE Register Bit Definitions

CCODE	Software Condition	
Value	SWCOND (1010)	NOT SWCOND (1011)
0x00	PF0 pin high	PF0 pin low
0x01	PF1 pin high	PF1 pin low
0x02	PF2 pin high	PF2 pin low
0x03	PF3 pin high	PF3 pin low
0x04	PF4 pin high	PF4 pin low
0x05	PF5 pin high	PF5 pin low
0x06	PF6 pin high	PF6 pin low
0x07	PF7 pin high	PF7 pin low
0x08	AS	NOT AS
0x09	SV	NOT SV
0x0A	PF8 pin high	PF8 pin low
0x0B	PF9 pin high	PF9 pin low
0x0C	PF10 pin high	PF10 pin low
0x0D	PF11 pin high	PF11 pin low
0x0E	PF12 pin high	PF12 pin low
0x0F	PF13 pin high	PF13 pin low

Interrupt Control (ICNTL) Register

Refer to Table 1-4 for ICNTL register bit definitions.

Table 1-4. ICNTL Register Bit Definitions

Bit	Name	Description
0	reserved	write 0
1	reserved	write 0
2	reserved	write 0
3	reserved	write 0
4	INE	Interrupt nesting enable. 0 =Disabled 1 =Enabled
5	GIE	Global interrupt enable. 0 =Disabled 1 =Enabled
6	reserved	write 0
7	BIASRND	MAC biased rounding mode. 0 =Disabled 1 =Enabled
8-9	reserved	write 0
10	PCSTKE	PC stack interrupt enable. 0 =Disabled 1 =Enabled
11	EMUCNTE	Emulator cycle counter interrupt enable. 0 =Disabled 1 =Enabled
12-15	reserved	write 0

Interrupt Mask (IMASK) Register and Interrupt Latch (IRPTL) Register

Refer to Table 1-5 for IMASK register and IRPTL register bit definitions.

Table 1-5. IMASK and IRPTL Register Bit Definitions

Bit	Name	Description
0	EMU	Emulator interrupt mask. Nonmaskable. Highest priority
1	PWDN	Power-down interrupt mask. Maskable only with GIE bit in ICNTL.
2	SSTEP	Single-step interrupt mask (during emulation)
3	STACK	Stack interrupt mask. Generated from any of the following stack status states: (if PCSTKE enabled) PC stack is pushed or popped and hits high-water mark, any stack overflows, or the status or PC stacks underflow.
4	User-defined	
5	User-defined	
6	User-defined	
7	User-defined	
8	User-defined	
9	User-defined	
10	User-defined	
11	User-defined	
12	User-defined	
13	User-defined	
14	User-defined	
15	User-defined	Lowest priority

Mode Status (MSTAT) Register

Refer to Table 1-6 for MSTAT register bit definitions.

Table 1-6. MSTAT Register Bit Definitions

Bit	Name	Description
0	SEC_REG or SR	Secondary data registers enable. Determines which set of data registers is currently active. 0 =Deactivate secondary set of data registers (default). Primary register set (set that is active at reset) enabled and used for normal operation; secondary register set disabled. 1 =Activate secondary set of data registers. Secondary register set enabled and used for alternate DSP context (for example, interrupt servicing); primary register set disabled, current contents preserved. For details, see “Switching Contexts” on page 7-16.
1	BIT_REV or BR	Bit-reversed addressing enable. Enables and disables bit-reversed addressing on DAG1 index registers only. 0 =Disable 1 =Enable For details, see “Bit-Reversed Addressing” on page 6-16.
2	AV_LATCH or OL	ALU overflow latch mode enable. Determines how the ALU overflow flag, AV, gets cleared. 0 =Disable Once an ALU overflow occurs and sets the AV bit in the ASTAT register, the AV bit remains set until explicitly cleared or is cleared by a subsequent ALU operation that does not generate an overflow. 1 =Enable Once an ALU overflow occurs and sets the AV bit in the ASTAT register, the AV bit remains set until the application explicitly clears it. For details on clearing the AV bit, see “Bit Manipulation: TSTBIT, SETBIT, CLRBIT, TGLBIT” on page 2-19 and “Register-to-Register Move” on page 6-22.

Table 1-6. MSTAT Register Bit Definitions (Cont'd)

Bit	Name	Description
3	AR_SAT or AS	ALU saturation mode enable. For signed values, determines whether ALU AR results that overflowed or underflowed are saturated or not. Enables or disables saturation for all subsequent ALU operations. 0 =Disable AR results remain unsaturated and return as is. 1 =Enable AR results saturated according to the state of the AV and AC status flags in ASTAT. **AVACAR register** 00ALU output 01ALU output 100x7FFF 110x8000 Only the results written to the AR register are saturated. If results are written to the AF register, wraparound occurs, but the AV and AC flags reflect the saturated result.
4	M_MODE or MM	MAC result mode. Determines the numeric format of multiplier operands. For all MAC operations, the multiplier adjusts the format of the result according to the selected mode. 0 =Fractional mode, 1.15 format. 1 =Integer mode, 16.0 format. For details, see "Data Format Options" on page 3-3.
5	TIMER or TI	Timer enable. Starts and stops the timer counter. 0 =Stops the timer count. 1 =Starts the timer count. For details on timer operation, see the *ADSP-219x/2192 DSP Hardware Reference*.
6	SEC_DAG or SD	Secondary DAG registers enable. Determines which set of DAG address registers is currently active. 0 =Primary registers. 1 =Secondary registers. For details, see "Secondary DAG Registers" on page 6-7 and "Switching Contexts" on page 7-16.

System Status (SSTAT) Register

Refer to Table 1-7 for SSTAT register bit definitions.

Table 1-7. SSTAT Register Bit Definitions

Bit	Name	Description
0	PCSTKEMPTY or PCE	PC stack empty. 0 =PC stack contains at least one pushed address. 1 =PC stack is empty.
1	PCSTKFULL or PCF	PC stack full. 0 =PC stack contains at least one empty location. 1 =PC stack is full.
2	PCSTKLVL or PCL	PC stack level. 0 =PC stack contains between 3 and 28 pushed addresses. 1 =PC stack is at or above the high-water mark (28 pushed addresses), or it is at or below the low-water mark (3 pushed addresses).
3	Reserved	
4	LPSTKEMPTY or LSE	Loop stack empty. 0 =Loop stack contains at least one pushed address. 1 =Loop stack is empty.
5	LPSTKFULL or LSF	Loop stack full. 0 =Loop stack contains at least one empty location. 1 =Loop stack is full.
6	STSSTKEMPTY or SSE	Status stack empty. 0 =Status stack contains at least one pushed status. 1 =Status stack is empty.
7	STKOVERFLOW or SOV	Stacks overflowed. 0 =Overflow/underflow has not occurred. 1 =At least one of the stacks (PC, loop, counter, status) has overflowed, or the PC or status stack has underflowed. This bit cleared only on reset. Loop stack underflow is not detected because it occurs only as a result of a POP LOOP operation.

Condition Codes Summary

Refer to Table 1-8 for CCODE register bit definitions.

Table 1-8. Condition Codes Summary

Code	Condition	Description
0000	EQ	Equal to zero (= 0).
0001	NE	Not equal to zero (≠ 0).
0010	GT	Greater than zero (> 0).
0011	LE	Less than or equal to zero (≤ 0).
0100	LT	Less than zero (< 0).
0101	GE	Greater than or equal to zero (≥ 0).
0110	AV	ALU overflow.
0111	NOT AV	Not ALU overflow.
1000	AC	ALU carry.
1001	NOT AC	Not ALU carry.
1010	SWCOND	SWCOND (based on CCODE register condition). (For CCODE details, see Table 1-3 on page 1-5.)
1011	NOT SWCOND	Not SWCOND (based on CCODE register condition). (For CCODE details, see Table 1-3 on page 1-5.)
1100	MV	MAC overflow.
1101	NOT MV	Not MAC overflow.
1110	NOT CE	Counter not expired.
1111	TRUE	Always true.

Instruction Summary

The conventions for ADSP-219x instruction syntax descriptions appear in Table 1-9. Other parts of the instruction syntax and opcode information also appear in this section. The following sections provide summaries of the DSP's instruction set:

- "ALU Instructions" on page 1-14
- "Multiplier Instructions" on page 1-15
- "Shifter Instructions" on page 1-16
- "Data Move Instructions" on page 1-16
- "Program Flow Instructions" on page 1-18
- "Multifunction Instructions" on page 1-19

For a list of instructions by types, see "Instruction Opcodes" on page 8-1.

Table 1-9. Instruction Set Notation

Notation	Meaning
UPPERCASE	Explicit syntax—assembler keyword (notation only; the assembler is case-insensitive and lowercase is the preferred programming convention)
;	Semicolon—instruction terminator
,	Comma—separates multiple optional items within vertical bars or separates parallel operations in multifunction instructions
| option1, option2 |	Vertical bars—lists options separated with commas (choose one)
[optional]	Square brackets—encloses optional part of instruction
Compute	ALU, multiplier, shifter or multifunction operation
ALU, MAC, SHIFT	ALU, multiplier, or shifter operation
Cond	Status condition

Table 1-9. Instruction Set Notation (Cont'd)

Notation	Meaning
Term	Loop termination condition
Reg	Any register from register groups Reg0, Reg1, Reg2, or Reg3
Dreg	Data register (register file) registers—subset of Reg0 registers
Ireg	Any DAG I register
Mreg	Any DAG M register
Lreg	Any DAG L register
Ia	I3-I0 (DAG1 index register)
Mb	M3-M0 (DAG1 modify register)
Ic	I7-I4 (DAG2 index register)
Md	M7-M4 (DAG2 modify register)
<Datan>	n-bit immediate data value
<Immn>	n-bit immediate modify value
<Addrn>	n-bit immediate address value
<Reladdrn>	n-bit immediate PC-relative address value
Const	constant value; For valid constant values, see Table 2-1 on page 2-3.
C	carry bit
(DB)	Delayed branch

ALU Instructions

Refer to Table 1-10 for a summary of ALU instructions.

Table 1-10. Summary of ALU Instructions

Instruction	Type	Details
|AR, AF| = Dreg1 + |Dreg2, Dreg2 + C, C |;	9, 9a	on page 2-5
[IF Cond] |AR, AF| = Xop + |Yop, Yop + C, C, Const, Const + C|;	9	on page 2-5
|AR, AF| = Dreg1 – |Dreg2, Dreg2 + C –1, +C –1|;	9, 9a	on page 2-9
[IF Cond] |AR, AF| = Xop – |Yop, Yop+C–1, +C–1, Const, Const +C –1|;	9	on page 2-9
|AR, AF| = Dreg2 – |Dreg1, Dreg1 + C –1|;	9, 9a	on page 2-13
[IF Cond] |AR, AF| = Yop – |Xop, Xop+C–1|;	9	on page 2-13
[IF Cond] |AR, AF| = – |Xop + C –1, Xop + Const, Xop + Const + C –1|;	9	on page 2-13
|AR, AF| = Dreg1 |AND, OR, XOR| Dreg2;	9, 9a	on page 2-16
[IF Cond] |AR, AF| = Xop |AND, OR, XOR| |Yop, Const|;	9	on page 2-16
[IF Cond] |AR, AF| = |TSTBIT, SETBIT, CLRBIT, TGLBIT| n of Xop;	9, 9a	on page 2-19
|AR, AF| = PASS |Dreg1, Dreg2, Const|;	9, 9a	on page 2-22
|AR, AF| = PASS 0;	9, 9a	on page 2-22
[IF Cond] |AR, AF| = PASS |Xop, Yop, Const|;	9	on page 2-22
|AR, AF| = NOT |Dreg|;	9, 9a	on page 2-25
[IF Cond] |AR, AF| = NOT |Xop, Yop|;	9	on page 2-25
|AR, AF| = ABS Dreg;	9, 9a	on page 2-28
[IF Cond] |AR, AF| = ABS Xop;	9	on page 2-28
|AR, AF| = Dreg +1;	9, 9a	on page 2-31
[IF Cond] |AR, AF| = Yop +1;	9	on page 2-31
|AR, AF| = Dreg –1;	9, 9a	on page 2-34

Table 1-10. Summary of ALU Instructions (Cont'd)

Instruction	Type	Details
[IF Cond] |AR, AF| = Yop –1;	9	on page 2-34
DIVS Yop, Xop;	24	on page 2-37
DIVQ Xop;	23	on page 2-37
NONE = ALU (Xop, Yop);	8	on page 2-46

Multiplier Instructions

Refer to Table 1-11 for a summary of multiplier instructions.

Table 1-11. Summary of Multiplier Instructions

Instruction	Type	Details
|MR, SR| = Dreg1 * Dreg2 [(|RND, SS, SU, US, UU|)];	9a	on page 3-8
[IF Cond] |MR, SR| = Xop * Yop [(|RND, SS, SU, US, UU|)];	9	on page 3-8
[IF Cond] |MR, SR| = Yop * Xop [(|RND, SS, SU, US, UU|)];	9	on page 3-8
|MR, SR| = |MR, SR| + Dreg1 * Dreg2 [(|RND, SS, SU, US, UU|)];	9a	on page 3-11
[IF Cond]|MR, SR| = |MR, SR| + Xop * Yop [(|RND, SS, SU, US, UU|)];	9	on page 3-11
[IF Cond] |MR, SR| = |MR, SR| + Yop * Xop [(|RND, SS, SU, US, UU|)];	9	on page 3-11
|MR, SR| = |MR, SR| – Dreg1 * Dreg2 [(|RND, SS, SU, US, UU|)];	9a	on page 3-14
[IF Cond] |MR, SR| = |MR, SR| – Xop * Yop [(|RND, SS, SU, US, UU|)];	9	on page 3-14
[IF Cond] |MR, SR| = |MR, SR| – Yop * Xop [(|RND, SS, SU, US, UU|)];	9	on page 3-14
[IF Cond] |MR, SR| = 0;	9	on page 3-17

Table 1-11. Summary of Multiplier Instructions (Cont'd)

Instruction	Type	Details
[IF Cond] MR = MR [(RND)]; [IF Cond] SR = SR [(RND)];	9	on page 3-19
SAT MR; SAT SR;	25	on page 3-21

Shifter Instructions

Refer to Table 1-12 for a summary of shifter instructions.

Table 1-12. Summary of Shifter Instructions

Instruction	Type	Details
[IF Cond] SR = [SR OR] ASHIFT Dreg [(|HI, LO|)];	16	on page 4-6
SR = [SR OR] ASHIFT BY <Imm8> [(|HI, LO|)];	15	on page 4-8
[IF Cond] SR = [SR OR] LSHIFT Dreg [(|HI, LO|)];	16	on page 4-10
SR = [SR OR] LSHIFT BY <Imm8> [(|HI, LO|)];	15	on page 4-12
[IF Cond] SR = [SR OR] NORM Dreg [(|HI, LO|)];	16	on page 4-14
[IF Cond] SE = EXP Dreg [(|HIX, HI, LO|)];	16	on page 4-20
[IF Cond] SB = EXPADJ Dreg;	16	on page 4-23

Data Move Instructions

Refer to Table 1-13 for a summary of data move instructions.

Table 1-13. Summary of Data Move Instructions

Instruction	Type	Details
Reg = Reg;	17	on page 6-22
|DM(<Addr16>) = |Dreg, Ireg, Mreg|;	3	on page 6-24

Table 1-13. Summary of Data Move Instructions (Cont'd)

Instruction	Type	Details
|Dreg, Ireg, Mreg| = |DM(<Addr16>)|;	3	on page 6-24
|<Dreg>, <Reg1>, <Reg2>| = <Data16>;	6, 7, 7A	on page 6-27
Reg3 = <Data12>;	33	on page 6-27
|DM(Ia += Mb), DM(Ic += Md)| = Reg;	32	on page 6-30
Reg = |DM(Ia += Mb), DM(Ic += Md)|;	32	on page 6-30
|DM(Ia + Mb), DM(Ic + Md)| = Reg;	32	on page 6-34
Reg = |DM (Ia + Mb), DM (Ic + Md)|;	32	on page 6-34
|PM(Ia += Mb), PM(Ic += Md)| = Reg;	32	on page 6-38
Reg = |PM(Ia += Mb), PM(Ic += Md)|;	32	on page 6-38
|PM(Ia + Mb), PM(Ic + Md)| = Reg;	32	on page 6-43
Reg = |PM(Ia + Mb), PM(Ic + Md)|;	32	on page 6-43
DM(Ireg1 += Mreg1) = |Ireg2, Mreg2, Lreg2|, |Ireg2, Mreg2, Lreg2| = Ireg1;	32A	on page 6-47
Dreg = DM(Ireg += <Imm8>);	29	on page 6-51
DM(Ireg += <Imm8>) = Dreg;	29	on page 6-51
Dreg = DM(Ireg + <Imm8>);	29	on page 6-54
DM(Ireg + <Imm8>) = Dreg;	29	on page 6-54
|DM(Ia += Mb), DM (Ic += Md)| = <Data16>;	22	on page 6-57
|PM (Ia += Mb), PM (Ic += Md)| = <Data24>:24;	22A	on page 6-59
IO(<Addr10>) = Dreg;	34	on page 6-62
Dreg = IO (<Addr10>);	34	on page 6-62
REG(<Addr8>) = Dreg;	35	on page 6-65
Dreg = REG(<Addr8>);	35	on page 6-65
|MODIFY (Ia += Mb), MODIFY (Ic += Md)|;	21	on page 6-68
MODIFY (Ireg += <Imm8>);	21A	on page 6-70

Program Flow Instructions

Refer to Table 1-14 for a summary of program flow instructions.

Table 1-14. Summary of Program Flow Instructions

Instruction	Type	Details
DO <Reladdr12> UNTIL [CE, FOREVER];	11	on page 7-24
[IF Cond] JUMP <Reladdr13> [(DB)];	10	on page 7-29
CALL <Reladdr16> [(DB)];	10a	on page 7-33
JUMP <Reladdr16> [(DB)];	10a	on page 7-37
[IF Cond] CALL <Addr24>;	36	on page 7-40
[IF Cond] JUMP <Addr24>;	36	on page 7-43
[IF Cond] CALL <Ireg> [(DB)];	19	on page 7-46
[IF Cond] JUMP <Ireg> [(DB)];	19	on page 7-50
[IF Cond] RTI [(DB)];	20	on page 7-53
[IF Cond] RTS [(DB)];	20	on page 7-57
PUSH |PC, LOOP, STS|;	26	on page 7-61
POP |PC, LOOP, STS|;	26	on page 7-61
FLUSH CACHE;	26	on page 7-67
SETINT <Imm4>;	37	on page 7-69
CLRINT <Imm4>;	37	on page 7-71
NOP;	30	on page 7-73
IDLE;	31	on page 7-74
ENA | TI, MM, AS, OL, BR, SR, BSR, INT | ;	18	on page 7-76
DIS | TI, MM, AS, OL, BR, SR, BSR, INT | ;	18	on page 7-76

Multifunction Instructions

Refer to Table 1-15 for a summary of multifunction instructions.

Table 1-15. Summary of Multifunction Instructions

Instruction	Type	Details
|<ALU>, <MAC>|, Xop = DM(Ia += Mb), Yop = PM(Ic += Md);	1	on page 5-4
Xop = DM(Ia += Mb), Yop = PM(Ic += Md);	1	on page 5-8
|<ALU>, <MAC>,<SHIFT> |, Dreg = DM(Ia += Mb)|;	4, 12	on page 5-11
|<ALU>, <MAC>, <SHIFT>|, DM(Ia += Mb) = Dreg;	4, 12	on page 5-15
|<ALU>, <MAC>, <SHIFT>|, Dreg = Dreg;	8, 14	on page 5-19

2 ALU INSTRUCTIONS

The instruction set provides ALU instructions for performing arithmetic and logical operations on 16- and 24-bit fixed-point data. This chapter includes the following sections:

- "ALU Instruction Conventions" on page 2-1
- "ALU Instruction Reference" on page 2-4

ALU Instruction Conventions

This chapter describes each of the arithmetic instructions and the following related topics:

- "Input Registers" on page 2-1
- "Output Registers" on page 2-2
- "Constants" on page 2-2
- "ALU Mode Control" on page 2-3
- "ALU Status Flags" on page 2-4

Input Registers

The unconditional single-function ALU instructions described in this chapter can use any of the DSP's 16 data registers (`Dregs`) as input operands. The conditional single-function ALU instructions are restricted to

the use of specific data registers for both the *x* and *y* input operands. When restrictions apply, XOP refers to the *x* operand, and YOP refers to the *y* operand.

Output Registers

ALU instructions use one of two output registers:

- AF—ALU Feedback register. Results are directly available for the *y* input only in the next conditional ALU operation.
- AR—ALU Result register. Results output to this register are immediately available as the x-input only in the next conditional ALU, MAC, or shifter operation or as either *x* or *y* input into the next unconditional ALU, MAC, or shifter operation.

Constants

You can use constants in any of the following single-function ALU instructions:

- Add operations
- Subtract operations
- Bitwise logic operations
- PASS operation

Valid constants are those formed from powers of two that fall within the range of – 32768 (0x8000) and +32767 (0x7FFF). Table 2-1 lists the valid constants.

Table 2-1. Valid Constant Values

Positive (+)		Negative (–)	
Decimal	Hexadecimal	Decimal	Hexadecimal
1	0x0001	2	0xFFFE
2	0x0002	3	0xFFFD
4	0x0004	5	0xFFFB
8	0x0008	9	0xFFF7
16	0x0010	17	0xFFEF
32	0x0020	33	0xFFDF
64	0x0040	65	0xFFBF
128	0x0080	129	0xFF7F
256	0x0100	257	0xFEFF
512	0x0200	513	0xFDFF
1024	0x0400	1025	0xFBFF
2048	0x0800	2049	0xF7FF
4096	0x1000	4097	0xEFFF
8192	0x2000	8193	0xDFFF
16384	0x4000	16385	0xBFFF
32767	0x7FFF	32768	0x8000

ALU Mode Control

The `MSTAT` register's `AV_LATCH` bit and `AR_SAT` bit enable and disable two ALU modes: ALU overflow latch mode and ALU saturation mode. For more information on these modes, see the bit descriptions in Table 1-6 on page 1-8.

ALU Status Flags

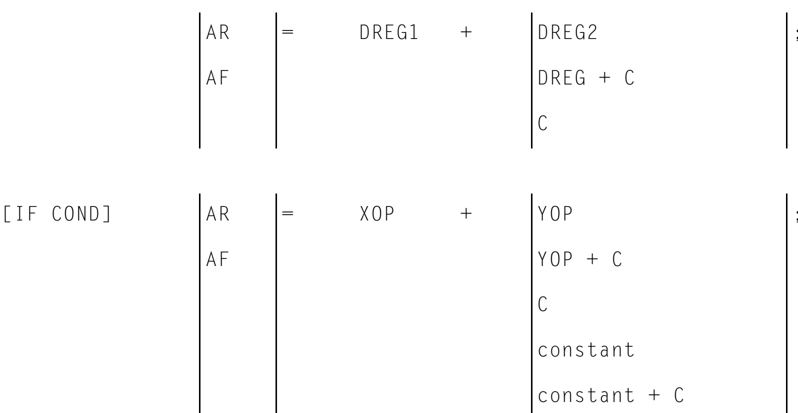
The ASTAT register's AZ, AN, AV, AC, AS, and AQ bits record the status of ALU operations, indicating whether the result of the operation was equal to zero, negative, overflowed, carried, signed, or produced a quotient. For information on these modes, see the bit descriptions in Table 1-2 on page 1-4.

ALU Instruction Reference

ALU instructions include:

- "Add/Add with Carry" on page 2-5
- "Subtract X–Y/Subtract X–Y with Borrow" on page 2-9
- "Subtract Y–X/Subtract Y–X with Borrow" on page 2-13
- "Bitwise Logic: AND, OR, XOR" on page 2-16
- "Bit Manipulation: TSTBIT, SETBIT, CLRBIT, TGLBIT" on page 2-19
- "Clear: PASS" on page 2-22
- "Negate: NOT" on page 2-25
- "Absolute Value: ABS" on page 2-28
- "Increment" on page 2-31
- "Decrement" on page 2-34
- "Divide Primitives: DIVS and DIVQ" on page 2-37
- "Generate ALU Status Only: NONE" on page 2-46

Add/Add with Carry



Function

Adds the input operands and stores the result in the specified result register.

If execution is based on a condition, the ALU performs the addition only if the condition evaluates true, and it performs a NOP operation if the condition evaluates false.

Input

For the unconditional form of this instruction, use any of these data registers for the DREG inputs:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

For the conditional form of this instruction, the input operands are restricted. Valid `XOP` and `YOP` registers are:

Xops	Yops
AX0, AX1, AR, MR0, MR1, MR2, SR0, SR1	AY0, AY1, AF, 0

Output

`AR`	ALU Result register. Results are directly available for *x* input only in the next conditional ALU, MAC, or shifter operation or as either *x* or *y* input in the next unconditional ALU, MAC, or shifter operation.
`AF`	ALU Feedback register. Results are directly available for the *y* input only in the next conditional ALU operation.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
AZ, AN, AV, AC	AS, AQ, MV, SS, SV
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

Omitting the condition forces unconditional execution of the instruction. This instruction uses binary addition to add the *x* and *y* operands and the carry bit, when specified.

The operands are stored in data registers, or, in the case of constants, supplied in the instruction. For the conditional form of this instruction, data registers are restricted.

You can substitute a constant for the *y* operand. For a list of valid constants, see Table 2-1 on page 2-3. To add a negative constant, use either of the following syntaxes:

```
AR = AR - 4097;
AR = AR + 0xEFFF;
```

Carry Option

```
IF AC AR = AX0 + AY0 + C;
```

The above instruction executes if a carry occurs in the previous instruction. The `AR` register receives the result of the addition of the *x* and *y* operands and the carry-in bit from the previous instruction. Otherwise, it performs a `NOP` operation.

The form `XOP + C` is a special case of `XOP + YOP + C` in which `YOP = 0`.

You cannot add or subtract constants in multifunction instructions, and you are restricted to the use of particular data registers in multifunction and conditional instructions.

Examples

```
AR = AX0 + AX1;                 /* add Dregs */
AF = MY0 + MR1 + C;             /* add Dregs and carry */
AR = SR0 + C;                   /* add Dreg and carry */
IF EQ AR = AX0 + AY0;           /* add X and Y ops */
IF LT AF = AX1 + AF + C;        /* add Xop, Yop, and carry */
IF AV AR = SR0 + C;             /* add Xop and carry */
IF AC AR = AR + 1024;           /* add Xop and constant */
IF SWCOND AR = MR0 + 1024 + C;  /* add Xop, constant, */
                                /* and carry */
```

See Also

- "Type 9: Compute" on page 8-23
- "Condition Code (CCODE) Register" on page 1-5
- "Mode Status (MSTAT) Register" on page 1-8

Subtract X–Y/Subtract X–Y with Borrow

```
            | AR | =  DREG1    –  | DREG2              | ;
            | AF |                | DREG + C - 1       |
            |    |                | + C - 1            |

[IF COND]   | AR | =  XOP      –  | YOP                | ;
            | AF |                | YOP + C - 1        |
            |    |                | + C - 1            |
            |    |                | constant           |
            |    |                | constant + C - 1   |
```

Function

Subtracts the input operands and stores the result in the specified result register.

If execution is based on a condition, the ALU performs the subtraction only if the condition evaluates true, and it performs a NOP operation if the condition evaluates false.

Input

For the unconditional form of this instruction, use any of these data registers for the DREG inputs:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

For the conditional form of this instruction, the input operands are restricted. Valid `XOP` and `YOP` registers are:

Xops	Yops
AX0, AX1, AR, MR0, MR1, MR2, SR0, SR1	AY0, AY1, AF, 0

Output

`AR` ALU Result register. Results are directly available for *x* input only in the next conditional ALU, MAC, or shifter operation or as either *x* or *y* input in the next unconditional ALU, MAC, or shifter operation.

`AF` ALU Feedback register. Results are directly available for the *y* input only in the next conditional ALU operation.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
AZ, AN, AV, AC	AS, AQ, MV, SS, SV
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

Omitting the condition forces unconditional execution of the instruction. This instruction uses binary addition to subtract the *y* operand from the *x* operand and then adds the carry bit minus one, when specified. The quantity `C–1` effectively implements a borrow capability for multiprecision subtractions.

The operands are stored in data registers, or, in the case of constants, supplied in the instruction. For the conditional form of this instruction, data registers are restricted.

You can substitute a constant for the *y* operand. For a list of valid constants, see Table 2-1 on page 2-3. To subtract a negative constant, use either of the following syntaxes:

```
AR = AX0 - 4097;
AR = AX0 + 0xEFFF;
```

Using the borrow option for example:

```
IF AC AR = AX0 - AY0 + C - 1;
```

The instruction executes if a carry occurs in the previous instruction. The `AR` register receives the result of the subtraction of the *x* and *y* operands and the carry bit from the previous instruction. Otherwise, it performs a `NOP` operation.

The form `XOP + C - 1` is a special case of `XOP - YOP + C - 1` in which `YOP = 0`.

You cannot add or subtract constants in multifunction instructions, and you are restricted to the use of particular data registers in multifunction and conditional instructions.

Examples

```
AR = AX0 - AX1;                 /* sub Dregs */
AF = MY0 - MR1 + C - 1;         /* sub Dregs and carry */
AR = SR0 + C - 1;               /* sub Dreg and carry */

IF EQ AR = AX0 - AY0;           /* sub X and Y ops */
IF LT AF = AX1 - AF + C - 1;    /* sub Xop, Yop, and carry */
IF AV AR = SR0 + C - 1;         /* sub Xop and carry */
```

```
IF AC AR = AR - 1024;              /* sub Xop and constant */
IF SWCOND AR = MR0 - 1024 + C - 1; /* sub Xop, const, and carry */
```

See Also

- “Type 9: Compute” on page 8-23
- “Condition Code (CCODE) Register” on page 1-5
- “Mode Status (MSTAT) Register” on page 1-8

Subtract Y–X/Subtract Y–X with Borrow

```
                 | AR |  =  DREG2    -  | DREG1          | ;
                 | AF |                 | DREG1 + C -1   |

[IF COND]        | AR |  =  YOP      -  | XOP            | ;
                 | AF |                 | XOP + C -1     |
```

Function

Subtracts the input operands and stores the result in the specified result register.

If execution is based on a condition, the ALU performs the subtraction only if the condition evaluates true, and it performs a `NOP` operation if the condition evaluates false.

Input

For the unconditional form of this instruction, you can use any of these data registers for the `DREG` inputs:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

For the conditional form of this instruction, the input operands are restricted. Valid `XOP` and `YOP` registers are:

Xops	Yops
AX0, AX1, AR, MR0, MR1, MR2, SR0, SR1	AY0, AY1, AF, 0

Output

`AR`	ALU Result register. Results are directly available for *x* input only in the next conditional ALU, MAC, or shifter operation or as either *x* or *y* input in the next unconditional ALU, MAC, or shifter operation.
`AF`	ALU Feedback register. Results are directly available for the *y* input only in the next conditional ALU operation.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
AZ, AN, AV, AC	AS, AQ, MV, SS, SV
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

Omitting the condition forces unconditional execution of the instruction. This instruction uses binary addition to subtract the *x* operand from the *y* operand and then adds the carry bit minus one, when specified. The quantity `C - 1` effectively implements a borrow capability for multiprecision subtractions.

The operands are stored in data registers, or, in the case of constants, supplied in the instruction. For the conditional form of this instruction, data registers are restricted.

You can substitute a constant for the *y* operand. For a list of valid constants, see Table 2-1 on page 2-3. To subtract a negative constant, you use this syntax:

```
AR = -4097 - AR;
AR = 0xEFFF - AR;
```

Using the borrow option for example:

```
IF AC AR = AY0 - AX0 + C - 1;
```

The instruction executes if a carry occurs in the previous instruction. The `AR` register receives the result of the subtraction of the *y* and *x* operands and the carry bit from the previous instruction. Otherwise, it performs a `NOP` operation.

The form `-XOP + C - 1` is a special case of `YOP - XOP + C - 1` in which `YOP = 0`.

You cannot add or subtract constants in multifunction instructions, and you are restricted to the use of particular data registers in multifunction and conditional instructions.

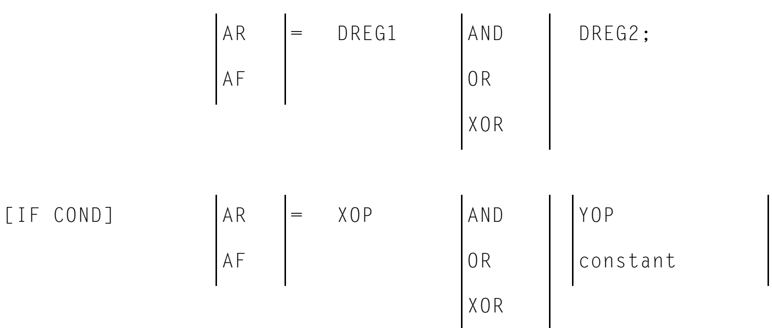
Examples

```
AR = AY0 - AY1;                /* sub Dregs */
AF = MR0 - SR1 + C -1;         /* sub Dregs, add carry */
IF EQ AR = AY0 - AX1;          /* sub Xop from Yop */
IF LT AF = AF - AX0 + C -1;    /* sub Xop from Yop, add carry */
```

See Also

- "Type 9: Compute" on page 8-23
- "Condition Code (CCODE) Register" on page 1-5
- "Mode Status (MSTAT) Register" on page 1-8

Bitwise Logic: AND, OR, XOR



Function

Performs the specified bitwise logical operation (logical AND, inclusive OR, or exclusive XOR) and stores the result in the specified result register.

If execution is based on a condition, the ALU performs the operation only if the condition evaluates true, and it performs a NOP operation if the condition evaluates false.

Only the conditional form of the bitwise operations accept a constant for the *y* operand.

Input

For the unconditional form of this instruction, you can use any of these data registers for the DREG inputs:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

For the conditional form of this instruction, the input operands are restricted. Valid XOP and YOP registers are:

Xops	Yops
AX0, AX1, AR, MR0, MR1, MR2, SR0, SR1	AY0, AY1, AF, 0

Output

`AR` ALU Result register. Results are directly available for *x* input only in the next conditional ALU, MAC, or shifter operation or as either *x* or *y* input in the next unconditional ALU, MAC, or shifter operation.

`AF` ALU Feedback register. Results are directly available for the *y* input only in the next conditional ALU operation.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
AZ, AN, AV (cleared), AC (cleared)	AS, AQ, MV, SS, SV
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

Omitting the condition forces unconditional execution of the instruction.

The operands are stored in data registers, or, in the case of constants, supplied in the instruction. For the conditional form of this instruction, data registers are restricted.

You can substitute a constant for the *y* operand. For a list of valid constants, see Table 2-1 on page 2-3.

Examples

```
AX0 = 0xAAAA;                  /* load 1010 1010 1010 1010 */
AX1 = 0x5555;                  /* load 0101 0101 0101 0101 */
AY0 = 0xAAAA;                  /* load 1010 1010 1010 1010 */
AY1 = 0x5555;                  /* load 0101 0101 0101 0101 */
AR = AX0 AND AX1;              /* AR = 0000 0000 0000 0000 */
AF = AY0 OR AY1;               /* AF = 1111 1111 1111 1111 */
AR = AX0 XOR AY0;              /* AR = 0000 0000 0000 0000 */

IF EQ AR = AX0 AND AY0;        /* AR = 1010 1010 1010 1010 */
IF LT AF = AX1 OR AY0;         /* AF = 1111 1111 1111 1111 */
IF SWCOND AR = AX0 XOR 0x1000; /* AR = 1011 1010 1010 1010 */
```

See Also

- "Type 9: Compute" on page 8-23
- "Condition Code (CCODE) Register" on page 1-5
- "Mode Status (MSTAT) Register" on page 1-8

Bit Manipulation: TSTBIT, SETBIT, CLRBIT, TGLBIT

```
[IF COND]        |AR  |=  |TSTBIT     |n OF XOP        ;
                 |AF  |   |SETBIT     |
                          |CLRBIT     |
                          |TGLBIT     |
```

Function

Performs the specified bit-manipulation operation on the *n* bit of the `X` input operand and stores the result in the specified result register.

- `TSTBIT` Performs an `AND` operation with `1` in the selected bit.
- `SETBIT` Performs an `OR` operation with `1` in the selected bit.
- `CLRBIT` Performs an `AND` operation with `0` in the selected bit.
- `TGLBIT` Performs an `XOR` operation with `1` in the selected bit.

If execution is based on a condition, the ALU performs the operation only if the condition evaluates true, and it performs a `NOP` operation if the condition evaluates false.

Input

For this instruction, the *x* operand is restricted. Valid `XOP` registers are:

Xops
AX0, AX1, AR, MR0, MR1, MR2, SR0, SR1

Output

`AR`	ALU Result register. Results are directly available for *x* input only in the next conditional ALU, MAC, or shifter operation or as either *x* or *y* input in the next unconditional ALU, MAC, or shifter operation.
`AF`	ALU Feedback register. Results are directly available for the *y* input only in the next conditional ALU operation.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
AZ, AN, AV (cleared), AC (cleared)	AS, AQ, MV, SS, SV
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

Omitting the condition forces unconditional execution of the instruction. The operand is stored in an `XOP` data register and the instruction specifies the particular bit within it. You cannot perform any of the bit manipulation operations in multifunction instructions.

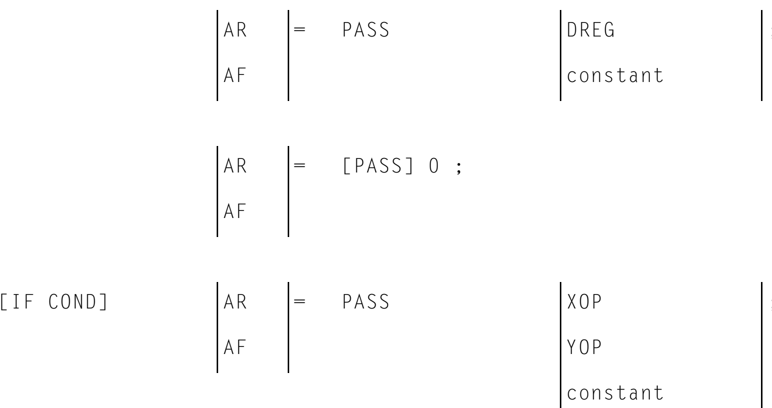
Examples

```
AX0 = 0xAAAA;                  /* load 1010 1010 1010 1010 */
AR = TSTBIT 0x5 OF AX0;        /* AR = 0x0020 */
AF = SETBIT 0x4 OF AX0;        /* AF = 0xAABA */
AF = CLRBIT 0xB OF AX0;        /* AF = 0xA2AA */
AR = TGLBIT 0xF OF AX0;        /* AR = 0x2AAA */
```

See Also

- "Type 9: Compute" on page 8-23
- "Condition Code (CCODE) Register" on page 1-5
- "Mode Status (MSTAT) Register" on page 1-8

Clear: PASS



Function

Passes the source operand unmodified through the ALU unit and stores it in the specified result register. Unlike the move register instruction, this instruction affects the ASTAT status flags.

The PASS 0 operation (the PASS keyword is optional) provides another way to clear the AR register.

If execution is based on a condition, the ALU performs the operation only if the condition evaluates true, and it performs a NOP operation if the condition evaluates false.

Input

For the unconditional form of this instruction, use any of these data registers for the DREG inputs:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

For the conditional form of this instruction, the input operands are restricted. Valid `XOP` and `YOP` registers are:

Xops	Yops
AX0, AX1, AR, MR0, MR1, MR2, SR0, SR1	AY0, AY1, AF, 0

Output

`AR` ALU Result register. Results are directly available for *x* input only in the next conditional ALU, MAC, or shifter operation or as either *x* or *y* input in the next unconditional ALU, MAC, or shifter operation.

`AF` ALU Feedback register. Results are directly available for the *y* input only in the next conditional ALU operation.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
AZ, AN, AV (cleared), AC (cleared)	AS, AQ, MV, SS, SV
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

Omitting the condition forces unconditional execution of the instruction.

The operands are stored in the data registers, or, in the case of constants, supplied in the instruction. For the conditional form of this instruction, data registers are restricted.

You can substitute a constant for the *y* operand. For a list of valid constants, see Table 2-1 on page 2-3.

Combine `PASS 0` with memory read and write operations in multifunction instructions to clear the `AR` register; for example:

```
AR = PASS 0, AX0 = DM(I0, M0), AY0 = PM(I4, M4);
```

You cannot use `DREG` data registers or the `PASS` *`constant`* operation (other than `-1`, `0`, or `1`) in multifunction instructions.

Some forms of the `PASS` instruction result from the special case of the *y* input operand is 0, and other forms of the `PASS` instruction result from the special case of `YOP` is a constant.

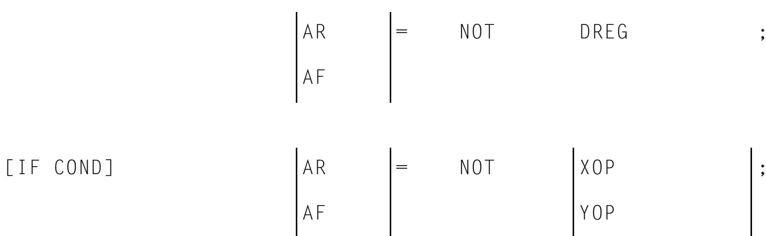
Examples

```
AR = PASS SI;                     /* pass Dreg to AR */
AF = PASS 1024;                   /* pass constant to AF */

AR = PASS 0;                      /* pass 0 to AR */
AF = PASS 0;                      /* pass 0 to AF */

IF EQ AR = PASS AX1;              /* pass Xop to AR */
IF LT AF = PASS AY0;              /* pass Yop to AR */
IF AV AR = PASS 1024;             /* pass constant to AR */
```

See Also

- "Type 9: Compute" on page 8-23
- "Condition Code (CCODE) Register" on page 1-5
- "Mode Status (MSTAT) Register" on page 1-8

Negate: NOT



Function

Performs a logical ones complement operation on the source operand and stores it in the specified result register.

If execution is based on a condition, the ALU performs the operation only if the condition evaluates true, and it performs a `NOP` operation if the condition evaluates false.

Input

For the unconditional form of this instruction, use any of these data registers for the `DREG` inputs:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

For the conditional form of this instruction, the input operands are restricted. Valid `XOP` and `YOP` registers are:

Xops	Yops
AX0, AX1, AR, MR0, MR1, MR2, SR0, SR1	AY0, AY1, AF, 0

Output

`AR`	ALU Result register. Results are directly available for *x* input only in the next conditional ALU, MAC, or shifter operation or as either *x* or *y* input in the next unconditional ALU, MAC, or shifter operation.
`AF`	ALU Feedback register. Results are directly available for the *y* input only in the next conditional ALU operation.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
AZ, AN, AV (cleared), AC (cleared)	AS, AQ, MV, SS, SV
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

Omitting the condition forces unconditional execution of the instruction.

The operands are stored in the data registers. For the conditional form of this instruction or in multifunction instructions, data registers are restricted.

Examples

```
AR = NOT SI;              /* put neg SI in AR  */
AF = NOT AY0;             /* put neg AY0 in AF */

IF EQ AR = NOT AX0;       /* put neg AX0 in AR */
IF LT AF = NOT AF;        /* put neg AF in AR  */
```

See Also

- "Type 9: Compute" on page 8-23
- "Condition Code (CCODE) Register" on page 1-5
- "Mode Status (MSTAT) Register" on page 1-8

Absolute Value: ABS

```
                    | AR   | =   ABS    DREG        ;
                    | AF   |

[IF COND]           | AR   | =   ABS    XOP         ;
                    | AF   |
```

Function

Performs a logical ones complement operation on the *x* input operand and stores it in the specified result register.

If execution is based on a condition, the ALU performs the operation only if the condition evaluates true, and it performs a `NOP` operation if the condition evaluates false.

Input

For the unconditional form of this instruction, use any of these data registers for the `DREG` inputs:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

For this instruction, the *x* operand is restricted. Valid `XOP` registers are:

Xops
AX0, AX1, AR, MR0, MR1, MR2, SR0, SR1

Output

`AR` ALU Result register. Results are directly available for *x* input only in the next conditional ALU, MAC, or shifter operation or as either *x* or *y* input in the next unconditional ALU, MAC, or shifter operation.

`AF` ALU Feedback register. Results are directly available for the *y* input only in the next conditional ALU operation.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
AZ, AN (set if xop is 0x8000), AV (set if xop is 0x8000), AC (cleared), AS (set if xop is negative)	AQ, MV, SS, SV
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

Omitting the condition forces unconditional execution of the instruction.

The operands are stored in the data registers. For the conditional form of this instruction or in multifunction instructions, data registers are restricted.

Examples

```
AR = ABS SI;                    /* put abs SI in AR  */
AF = ABS AY0;                   /* put abs AY0 in AF */

IF EQ AR = ABS AX0;             /* put abs AX0 in AR */
IF LT AF = ABS SR0;             /* put abs SR0 in AF */
```

See Also

- "Type 9: Compute" on page 8-23
- "Condition Code (CCODE) Register" on page 1-5
- "Mode Status (MSTAT) Register" on page 1-8

Increment

```
                    | AR |  =   DREG + 1 ;
                    | AF |

[IF COND]           | AR |  =   YOP  + 1 ;
                    | AF |
```

Function

Increments the *y* input operand by adding `0x0001` to it and stores the value in the specified result register.

If execution is based on a condition, the ALU performs the operation only if the condition evaluates true, and it performs a `NOP` operation if the condition evaluates false.

Input

For the unconditional form of this instruction, use any of these data registers for the `DREG` inputs:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

For this instruction, the *y* operand is restricted. Valid `IOP` registers are:

Yops
AY0, AY1, AF, 0

Output

`AR`	ALU Result register. Results are directly available for *x* input only in the next conditional ALU, MAC, or shifter operation or as either *x* or *y* input in the next unconditional ALU, MAC, or shifter operation.
`AF`	ALU Feedback register. Results are directly available for the *y* input only in the next conditional ALU operation.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
AZ, AN, AV, AC	AS, AQ, MV, SS, SV
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

Omitting the condition forces unconditional execution of the instruction.

The operands are stored in the data registers. For the conditional form of this instruction or in multifunction instructions, data registers are restricted.

Examples

```
AR = SI + 1;              /* inc SI and place in AR  */
AF = AX0 + 1;             /* inc AX0 and place in AF */

IF EQ AR = AY0 + 1;       /* inc AY0 and place in AR */
IF LT AF = AF + 1;        /* inc AF and place in AF  */
```

See Also

- "Type 9: Compute" on page 8-23
- "Condition Code (CCODE) Register" on page 1-5
- "Mode Status (MSTAT) Register" on page 1-8

Decrement

```
                     |AR   |=   DREG - 1 ;
                     |AF   |

[IF COND]            |AR   |=   YOP - 1 ;
                     |AF   |
```

Function

Decrements the *y* operand by subtracting `0x0001` from it and stores the value in the specified result register.

If execution is based on a condition, the ALU performs the operation only if the condition evaluates true, and it performs a `NOP` operation if the condition evaluates false.

Input

For the unconditional form of this instruction, use any of these data registers for the `DREG` inputs:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

For this instruction, the *y* operand is restricted. Valid `IOP` registers are:

Yops
AY0, AY1, AF, 0

Output

`AR`	ALU Result register. Results are directly available for *x* input only in the next conditional ALU, MAC, or shifter operation or as either *x* or *y* input in the next unconditional ALU, MAC, or shifter operation.
`AF`	ALU Feedback register. Results are directly available for the *y* input only in the next conditional ALU operation.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
AZ, AN, AV, AC	AS, AQ, MV, SS, SV
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

Omitting the condition forces unconditional execution of the instruction.

The operands are stored in the data registers. For the conditional form of this instruction or in multifunction instructions, data registers are restricted.

Examples

```
AR = SI - 1;              /* dec SI and place in AR  */
AF = AX0 - 1;             /* dec AX0 and place in AF */

IF EQ AR = AY0 - 1;       /* dec AY0 and place in AR */
IF LT AF = AF - 1;        /* dec AF and place in AF  */
```

See Also

- "Type 9: Compute" on page 8-23
- "Condition Code (CCODE) Register" on page 1-5
- "Mode Status (MSTAT) Register" on page 1-8

Divide Primitives: DIVS and DIVQ

```
DIVS YOP, XOP ;
DIVQ XOP ;
```

Function

The `DIVS` primitive calculates the quotient's sign bit. The `DIVQ` primitive calculates the quotient one bit at a time.

Use both divide primitives to implement a `YOP ÷ XOP` operation on signed (twos complement) numbers. Use the `DIVQ` primitive alone to implement a `YOP ÷ XOP` operation on unsigned (ones complement) numbers.

The divide primitives perform a single-precision divide on a 32-bit numerator by a 16-bit denominator to yield a 16-bit, truncated quotient. Single-precision divides executes in sixteen cycles. Higher precision divides require more cycles since you must execute the `DIVQ` primitive for each bit in the quotient.

The divide operation requires four data registers—one 16-bit data register to hold the 16-bit divisor, two 16-bit data registers to hold the 32-bit dividend, and one 16-bit data register to hold the resulting 16-bit quotient.

Input

Use signed (twos complement) or unsigned (ones complement) operands, but both operands must be the same number format. The resulting quotient has the same number format as its operands.

`XOP` Divisor. Both divide primitives take an *x* input operand. Valid `XOP` registers are:

Xops
AX0, AX1, AR, MR0, MR1, MR2, SR0, SR1

YOP	Dividend. Only the DIVS primitive, which calculates the sign bit of the quotient, takes a *y* input operand. The *y* input operand must contain the upper 16 bits of the dividend. Valid YOP registers are:

Yops
AY1, AF

For unsigned division (DIVQ only), AF must be used as the *y* input operand to hold the upper 16 bits of the dividend. Before issuing the DIVQ primitive, you must explicitly load the lower 16 bits of the dividend into either AY0.

For signed division (DIVS and DIVQ), use AY1 or AF as they input operand to contain the upper 16 bits of the dividend. Before issuing either of the divide primitives, you must explicitly load the lower 16 bits of the dividend into AY0.

Output

AY0	ALU divide result register. At the end of the divide operation, the AY0 data register contains the quotient.
AF	ALU Feedback register. At the end of the divide operation, AF contains the final remainder. This value is incorrect. If you need to use it, correct it before doing so.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
AQ	AZ, AN, AV, AC, AS, MV, SS, SV
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

These instructions implement YOP ÷ XOP. There are two divide primitives, DIVS and DIVQ. A single-precision divide, with a 32-bit numerator and a 16-bit denominator, yielding a 16-bit quotient, executes in 16 cycles. Higher precision divides are also possible.

The division can be signed or unsigned, but both the numerator and denominator must be the same, signed or unsigned. Set up the divide by sorting the upper half of the numerator in any permissible YOP (AY1 or AF), the lower half of the numerator in AY0, and the denominator in any permissible XOP. The divide operation is then executed with the divide primitives, DIVS and DIVQ. Repeated execution of DIVQ implements a non-restoring conditional add-subtract division algorithm. At the conclusion of the divide operation, the quotient will be in AY0.

To implement a signed divide, first execute the DIVS instruction once, which computes the sign of the quotient. Then execute the DIVQ instruction as many times as there are bits remaining in the quotient (for example, for a signed, single-precision divide, execute DIVS once and DIVQ 15 times).

To implement an unsigned divide, place the upper half of the numerator in AF and then set the AQ bit to zero by manually clearing it in the Arithmetic Status register (ASTAT). This indicates that the sign of the quotient is positive. Then execute the DIVQ instruction as many times as there are bits in the quotient (for example, for an unsigned single-precision divide, execute DIVQ 16 times).

The quotient bit generated on each execution of `DIVS` and `DIVQ` is the `AQ` bit, which is written to the `ASTAT` register at the end of each cycle. The final remainder produced by this algorithm (left over in the `AF` register) is not valid and must be corrected if it is needed.

Examples

For example code and code walk-through, see "Division Applications" on page 2-45.

See Also

- For more information, see "Division Theory" on page 2-40, "Division Exceptions" on page 2-43, and "Division Applications" on page 2-45.
- "Type 23: Divide primitive, DIVQ" on page 8-43
- "Type 24: Divide primitive, DIVS" on page 8-44
- "Condition Code (CCODE) Register" on page 1-5
- "Mode Status (MSTAT) Register" on page 1-8

Division Theory

The ADSP-219x DSP family's instruction set contains two instructions for implementing a non-restoring divide algorithm. These instructions take as their operands' twos complement or unsigned numbers, and in 16 cycles produce a truncated quotient of 16 bits. For most numbers and applications, these primitives produce the correct results. However, certain situations produce results that are off by one LSB. This section describes these situations, and presents alternatives for producing the correct results.

Computing a 16-bit fixed-point quotient from two numbers is accomplished by 16 executions of the `DIVQ` instruction for unsigned numbers. Signed division uses the `DIVS` instruction first, followed by fifteen `DIVQ` instructions. Regardless of the division you perform, both input operands must be of the same type (signed or unsigned) and produce a result of the same type.

These two instructions are used to implement a conditional add/subtract, non-restoring division algorithm. As its name implies, the algorithm functions by adding or subtracting the divisor to/from the dividend. The decision as to which operation to perform is based on the previously generated quotient bit. Each add/subtract operation produces a new partial remainder, which is used in the next step.

The term "non-restoring" refers to the fact that the final remainder is not correct. With a restoring algorithm, it is possible, at any step, to take the partial quotient, multiply it by the divisor, and add the partial remainder to recreate the dividend. With this non-restoring algorithm, it is necessary to add two times the divisor to the partial remainder if the previously determined quotient bit is zero. It is easier to compute the remainder using the multiplier than in the ALU.

Signed Division

Signed division is accomplished by first storing the 16-bit divisor in an `XOP` register (`AX0`, `AX1`, `AR`, `MR2`, `MR1`, `MR0`, `SR1`, or `SR0`). The 32-bit dividend must be stored in two separate 16-bit registers. The lower 16-bits must be stored in `AY0`, and the upper 16-bits can be in `AY1` or `AF`.

The `DIVS` primitive is executed once, with the proper operands (such as `DIVS AY1, AX0`) to compute the sign of the quotient. The sign bit of the quotient is determined by XORing (exclusive ORing) the sign bits of each operand. The entire 32-bit dividend is shifted left one bit. The lower 15 bits of the dividend with the recently determined sign bit appended are stored in `AY0`, and the lower 15 bits of the upper word, with the MSB of the lower word appended is stored in `AF`.

To complete the division, 15 DIVQ instructions are executed. Operation of the DIVQ primitive is described below.

Unsigned Division

Computing an unsigned division is done like signed division, except the first instruction is not a DIVS, but another DIVQ. The upper word of the dividend must be stored in AF, and the AQ bit of the ASTAT register must be set to zero before the divide begins.

The DIVQ instruction uses the AQ bit of the ASTAT register to determine whether the dividend should be added to or subtracted from the partial remainder stored in AF and AY0. If AQ is zero, a subtraction occurs. A new value for AQ is determined by XORing the MSB of the divisor with the MSB of the dividend. The 32-bit dividend is shifted left one bit, and the inverted value of AQ is moved into the LSB.

Output Formats

As in multiplication, the format of a division result is based on the format of the input operands. The division logic is designed to work most efficiently with fully fractional numbers—those most commonly used in fixed-point DSP applications. A signed, fully fractional number uses one bit before the binary point as the sign, with 15 bits (or 31 bits in double precision) to the right, for magnitude.

If the dividend is in M.N format (M bits before the binary point, N bits after), and the divisor is in O.P format, the quotient's format will be (M-O+1).(N-P-1). Dividing a 1.31 number by a 1.15 number produces a quotient whose format is (1-1+1).(31-15-1) or 1.15.

Before dividing two numbers, ensure that the format of the quotient will be valid. For example, if you attempt to divide a 32.0 number by a 1.15 number, the result attempts to be in (32-1+1).(0-15-1) or 32.-16 format. This cannot be represented in a 16-bit register.

In addition to proper output format, ensure that a divide overflow does not occur. Even when a division of two numbers produces a valid output format, it is possible that the number will overflow and be unable to fit within the constraints of the output. For example, to divide a 16.16 number by a 1.15 number, the output format would be (16-1+1).(16-15-1) or 16.0, which is valid. Assume you have 16384 (0x4000) as the dividend and .25 (0x2000) as the divisor, the quotient is 65536, which does not fit in 16.0 format. This operation overflows, producing an erroneous result.

Check input operands before division to ensure that an overflow will not result. If the magnitude of the upper 16 bits of the dividend is larger than the magnitude of the divisor, an overflow will result.

Integer Division

One special case of division that deserves special mention is integer division. There may be some cases where you wish to divide two integers, and produce an integer result. It can be seen that an integer-integer division will produce an invalid output format of (32-16+1).(0-0-1), or 17.-1.

To generate an integer quotient, shift the dividend to the left one bit, placing it in 31.1 format. The output format for this division will be (31-16+1).(1-0-1), or 16.0. Ensure that no significant bits are lost during the left shift, or an invalid result will be generated.

Division Exceptions

Although the divide primitives for the ADSP-219x DSP family work correctly in most instances, there are two cases where an invalid or inaccurate result can be generated. The first case involves signed division by a negative number. If you attempt to use a negative number as the divisor, the quotient generated may be one LSB less than the correct result. The other case concerns unsigned division by a divisor greater than `0x7FFF`. If the divisor in an unsigned division exceeds `0x7FFF`, an invalid quotient will be generated.

Negative Divisor Error

The quotient produced by a divide with a negative divisor will generally be one LSB less than the correct result. The divide algorithm implemented on the ADSP-219x DSP family, which does not correctly compensate for the twos complement format of a negative number, causes this inaccuracy.

There is one case where this discrepancy does not occur. When the result of the division operation equals `0x8000`, it is correctly represented, and is not one LSB off.

There are several ways to correct for this error. Before changing any code, however, determine if a one-LSB error in your quotient is a significant problem. In some cases, the LSB is small enough to be insignificant.

If exact results are necessary, two solutions are possible. One is to avoid division by negative numbers. If your divisor is negative, take its absolute value and invert the sign of the quotient after division. This will produce the correct result.

Another technique is to check the result by multiplying the quotient by the divisor. Compare this value with the dividend; if they are off by more than the value of the divisor, increase the quotient by one.

Unsigned Division Error

Unsigned divisions can produce erroneous results if the divisor is greater than `0x7FFF`. Do not attempt to divide two unsigned numbers when the divisor has a one in the MSB. If you must perform such a division, shift both operands right one bit. This will maintain the correct orientation of operands.

Shifting both operands may result in a one LSB error in the quotient. This can be solved by multiplying the quotient by the original (not shifted) divisor. Subtract this value from the original dividend to calculate the error. If the error is greater than the divisor, add one to the quotient; if it is negative, subtract one from the quotient.

Division Applications

Each of the problems mentioned in "Division Exceptions" on page 2-43 can be compensated in software. Listing 2-1 shows the program section *divides*. This code can be used to divide two signed or unsigned numbers to produce the correct quotient, or an error condition.

Listing 2-1. Division Routine Using DIVS and DIVQ

```
/* signed division algorithm with fix for negative division error

inputs:
  AYy1 - 16 MSB of numerator
  AYy0 - 16 LSB of numerator
  AR   - denominator
outputs:
  AR   - corrected quotient

intermediate (scratch) registers:
  MR0, AF  */

signed_div:
  MR0 = AR, AR = ABS AR;
  /*  save copy of denominator, make it positive  */
  DIVS AY1, AR;  DIVQ AR;
  DIVQ AR;  DIVQ AR;
  DIVQ AR;  DIVQ AR;
  DIVQ AR;  DIVQ AR;
  DIVQ AR;  DIVQ AR;
  DIVQ AR;  DIVQ AR;
  DIVQ AR;  DIVQ AR;
  DIVQ AR;  DIVQ AR;
  AR = AY0, AF = PASS MR0;  /*  get sign of denominator  */
  IF LT AR = -AY0;  /*  if neg, invert output, place in ar  */
  RTS;
```

Generate ALU Status Only: NONE

```
NONE = <ALU Operation> ;
```

Function

Performs the indicated unconditional ALU operation but does not load the results into the AR or AF result registers. Generates ALU status flags only. Use this instruction to set ALU status without disturbing the contents of the AR and AF result registers.

Input

XOP — Limits the registers for the *x* input operand. Valid XOP registers are:

Xops
AX0, AX1, AR, MR0, MR1, MR2, SR0, SR1

YOP — Limits the registers for the *x* input operand. Valid YOP registers are:

Yops
AY0, AY1, AF, 0

Output

None. Generates ALU status flags only.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
Depending on ALU operation—AZ, AN, AV, AC, AS, AQ	MV, SS, SV
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

Use any unconditional ALU operation (except ALU operations that use constants) to generate ALU status flags.

The following ALU operations may not appear in the `NONE=` syntax:

- `NONE = <XOP> + <constant>;` For other add operations, see "Add/Add with Carry" on page 2-5.
- `NONE = <XOP> - <constant>;`
  -or-
  `NONE = -<XOP> + <constant>;`
  For other subtract operations, see "Subtract X–Y/Subtract X–Y with Borrow" on page 2-9.
- `NONE = PASS <constant>;` with any constant other than -1, 0, or 1. For other clear operations, see "Clear: PASS" on page 2-22.
- `NONE = <XOP> <AND|OR|XOR> <constant>;`
  For other logical operations, see "Bitwise Logic: AND, OR, XOR" on page 2-16.
- `NONE` with `TSTBIT`, `SETBIT`, `CLRBIT`, or `TGLBIT`.
- `NONE` with the division primitives (`DIVS` or `DIVQ`).

Examples

```
NONE = AX0 - AF;            /* generate status from sub */
NONE = AX1 + AF;            /* generate status from add */
```

```
NONE = AF - AX1;              /* generate status from sub */
NONE = PASS 0;                /* generate status from pass */
NONE = AX1 OR AY0;            /* generate status from or */
```

See Also

- "Type 8: Compute | Dreg1 «··· Dreg2" on page 8-22
- "Condition Code (CCODE) Register" on page 1-5
- "Mode Status (MSTAT) Register" on page 1-8

3 MAC INSTRUCTIONS

The instruction set provides MAC instructions that perform high-speed multiplication and multiply with cumulative add/subtract operations. MAC instructions include:

- "Multiply" on page 3-8
- "Multiply with Cumulative Add" on page 3-11
- "Multiply with Cumulative Subtract" on page 3-14
- "MAC Clear" on page 3-17
- "MAC Round/Transfer" on page 3-19
- "MAC Saturate" on page 3-21
- "Generate MAC Status Only: NONE" on page 3-24

This chapter describes the individual MAC instructions and these related topics:

- "MAC Input Registers" on page 3-2
- "MAC Output Registers" on page 3-2
- "Data Format Options" on page 3-3
- "Status Flags" on page 3-7

For details on condition codes and data input and output registers, see "Condition Codes" on page 8-8 and "Core Register Codes" on page 8-11.

Multiply Instruction Conventions

MAC Input Registers

All unconditional, single-function multiply and multiply with accumulative add or subtract instructions can use any DREG data register for the *x* and *y* input operands (for details, see "Core Register Codes" on page 8-11). A program can use, for example, the ALU registers for the multiplication or shifter operations, without issuing a separate data move instruction. This capability simplifies register allocation in algorithm coding. For example, using the DSP's dual accumulator:

```
SR = SR + MX0 * MY0 (SS);
```

In multifunction operations, you can use only certain registers for the *x*-input operand (AR, MX0, MX1, MR0, MR1, MR2, SR0, SR1) and the *y*-input operand (MY0, MY1, SR1, 0).

All conditional MAC instructions must use the restricted XOP and YOP data registers for the x and y input operands, or an XOP register for the x-input and 0 for the y-input.

MAC Output Registers

All MAC instructions can use the multiplier MR output registers or the shifter SR output registers to receive the result of a multiplier operation. Availability of the shifter SR output registers for multiplier operations provides dual-accumulator functionality.

When MR is the result register, results are directly available from MR0, MR1, or MR2 as the *x*-input operand into the very next multiplier operation.

```
MR = MR + AX0 * AX0 (SS);
```

When SR is the result register, the 16-bit value in SR1 (bits 31:16 of the 40-bit result) is directly available as the *y*-input operand into the very next multiplier operation. This functionality is most useful when shifting the results of a multiply/accumulate operation since it decreases the number of required data moves.

```
SR = SR + AX0 * AY0 (SS);
SR = SR + SR1 * AY0 (SS);
```

Data Format Options

Multiplier operations require the instruction to specify the data format of the input operands (signed or unsigned) or specify that the multiplier rounds (RND) the product of two signed operands.

All data format options, except the round (RND) option, which affects the product stored in the result register, specify the format of both input operands in x/y order. The data format options are:

- (RND) Round value in result register.

  When overflow occurs, rounds the product to the most significant twenty-four bits—SR2/SR1 or MR2/MR1 represent the rounded 24-bit result. Otherwise, rounds bits 31:16 to16 bits—MR1 or SR1 contain the rounded 16-bit result.

  With (RND) selected, the multiplier considers both input operands signed (twos complement). If the DSP is in fractional mode (MSTAT:M_MODE = 0), the multiplier rounds the result after adjusting for fractional data format. For details, see “Numeric Format Modes” on page 3-6.

  The DSP provides two rounding modes (biased and unbiased) to support a variety of application algorithms. For details, see “Rounding Modes” on page 3-4.

- `(SS)` Both input operands are signed numbers. Signed numbers are in twos complement format.

  Use this option to multiply two signed single-precision numbers or to multiply the upper portions of two signed multi-precision numbers.

- `(SU)` X-input operand is signed; y-input operand is unsigned.

  Use this option to multiply a signed single-precision number by an unsigned single-precision number.

- `(US)` X-input operand is unsigned; y-input operand is signed.

  Use this option to multiply an unsigned single-precision number by a signed single-precision number.

- `(UU)` Both input operands are unsigned numbers. Unsigned numbers are in ones complement format.

  Use this option to multiply two unsigned single-precision numbers or to multiply the lower portions of two signed multi-precision numbers.

Rounding Modes

Rounding operates on the boundary between bits 15 and 16 of the 40-bit adder result. The multiplier directs the rounded output to the MR or the `SR` result registers.

ADSP-219x DSPs provide two modes for rounding. The rounding algorithm is the same for both modes, but the final results can differ when the product equals the midway value (`MR0 = 0x8000`).

In both methods, the multiplier adds `1` to the value of bit 15 in the adder chain. But when `MR0 = 0x8000`, the multiplier forces bit 16 in the result output to `0`. Although applied on every rounding operation, the result of this algorithm is evident only when `MR0 = 0x8000` in the adder chain.

The rounding mode determines the final result. The `BIASRND` bit in the `ICNTL` register selects the mode. `BIASRND = 0` selects unbiased rounding, and `BIASRND = 1` selects biased rounding.

- Unbiased rounding. Default mode. Rounds up only when `MR1/SR1` set to an odd value; otherwise, rounds down. Yields a zero large-sample bias.
- Biased rounding. Always rounds up when `MR0/SR0` is set to `0x8000`.

Table 3-1 shows the results of rounding for both modes.

Table 3-1. MR result values

MR Value before RND	Biased RND Result	Unbiased RND Result
00-0000-8000	00-0001-0000	00-0000-0000
00-0001-8000	00-0002-0000	00-0002-0000
00-0000-8001	00-0001-0001	00-0001-0001
00-0001-8001	00-0002-0001	00-0002-0001
00-0000-7FFF	00-0000-FFFF	00-0000-FFFF
00-0001-7FFF	00-0001-FFFF	00-0001- FFFF

Unbiased rounding, which is preferred for most algorithms, yields a zero large-sample bias, assuming uniformly distributed values. Biased rounding supports efficient implementation of bit-specified algorithms, such as GSM speech compression routines.

Numeric Format Modes

The multiplier can operate on integers or fractions. The M_MODE bit in the MSTAT register selects the mode. M_MODE = 0 selects fractional mode, and M_MODE = 1 selects integer mode.

The mode determines whether the multiplier shifts the product before adding or subtracting it from the result register.

Integer mode — 16.0 integer format.

The LSB of the 32-bit product is aligned with the LSB of MR0/SR0.

In multiply and accumulate operations, the multiplier sign-extends the 32-bit product (8 bits) and then adds or subtracts that value from the result register to form the new 40-bit result.

The multiplier sets the MV/SV overflow bit when the result falls outside the range of -1 to $+1-2^{31}$.

Fractional mode — 1.15 fraction format.

Fractions range from -1 to $+1-2^{15}$. The MSB of the product is aligned with the MSB of MR1/SR1. MR1-0/SR1-0 hold a 32-bit fraction (1.31 format) in the range of -1 to $+1-2^{31}$, and MR2/SR2 contains the eight sign-extended bits. In total, the MR/SR registers contains a fraction in 9.31 format.

In multiply and accumulate operations, the multiplier adjusts the format of the 32-bit product before adding or subtracting it from the result register. To do so, the multiplier sign-extends the product

(seven bits), shifts it one bit to the left, and then adds or subtracts that value from the result register to form the new 40-bit result.

The multiplier sets the MV/SV overflow bit when the result falls outside the range of -1 to $+1-2^{31}$.

Status Flags

Two status flags in the ASTAT register record the status of multiplier operations. MV = 1 records an overflow or underflow state when MR is the specified result register, and SV = 1 records an overflow or underflow state when SR is the specified result register.

Multiply

```
             |MR |=  DREG1    *  DREG2    (  |RND |) ;
             |SR |                           |SS  |
                                             |SU  |
                                             |US  |
                                             |UU  |

[IF COND]    |MR  =  XOP      *  YOP    |(   |RND |) ;
             |SR                 XOP    |    |SS  |
                                             |SU  |
                                             |US  |
                                             |UU  |
```

Function

Multiplies the input operands and stores the result in the specified result register. Optionally, inputs may be signed or unsigned, and output may be rounded. For more information on input and output options, see "Data Format Options" on page 3-3.

If execution is based on a condition, the multiplier performs the multiplication only if the condition evaluates true, and it performs a NOP operation if the condition evaluates false. Omitting the condition forces unconditional execution of the instruction.

Input

For the unconditional form of this instruction, use any of these data registers for the DREG inputs:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

For the conditional form of this instruction, the input operands are restricted. Valid XOP and YOP registers are:

Xops	Yops
AR, MX0, MX1, MR0, MR1, MR2, SR0, SR1	MY0, MY1, SR1, 0

Output

MR — Multiplier Result register. Results are directly available for *x* input only in the next conditional ALU, MAC, or shifter operation or as *x* or *y* input in the next unconditional ALU, MAC, or shifter operation.

SR — Multiplier Feedback register. Results are directly available for *x* (SR0 and SR1) or *y* (SR1 only) input in the next conditional ALU, MAC, or shifter operation or as *x* or *y* input in the next unconditional ALU, MAC, or shifter operation.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
MV (if MR used), SV (if SR used)	AZ, AN, AV, AC, AS, AQ, SS
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

This instruction provides a squaring operation (XOP*XOP and DREG1*DREG1) that performs single-cycle X^2 and ΣX^2 functions. In squaring operations, you must use the same register for both x-input operands.

You cannot use DREG form of the multiply instruction in multifunction instructions.

Examples

```
MR = AY0 * SI (RND);            /* mult DREGs, round result */
SR = AX0 * MX1 (SS);            /* mult DREGs, signed inputs */

IF MV MR = MX0 * MY0 (SU);      /* mult signed X, unsigned Y */
CCODE = 0x09; NOP;              /* set CCODE for SV condition */
IF SWCOND SR = MR0 * SR1 (UU);  /* mult unsigned X and Y */
```

See Also

- "Type 9: Compute" on page 8-23
- "Condition Code (CCODE) Register" on page 1-5
- "Mode Status (MSTAT) Register" on page 1-8

Multiply with Cumulative Add

```
           |MR |= |MR |+  DREG1  *  DREG2   (  |RND | ) ;
           |SR |= |SR |                        |SS  |
                                               |SU  |
                                               |US  |
                                               |UU  |

[IF COND]  |MR |= |MR |+ |XOP  |* |YOP  |(  |RND | ) ;
           |SR |= |SR |  |YOP  |  |XOP  |   |SS  |
                                            |SU  |
                                            |US  |
                                            |UU  |
```

Function

Multiplies the input operands, adds the product to the current contents of the MR or SR register, and then stores the sum in the corresponding result register. Optionally, inputs may be signed or unsigned, and output may be rounded. For more information on input and output options, see "Data Format Options" on page 3-3.

If execution is based on a condition, the multiplier performs the operation only when the condition evaluates true, and it performs a NOP operation when the condition evaluates false. Omitting the condition forces unconditional execution of the instruction.

Input

For the unconditional form of this instruction, use any of these data registers for the `DREG` inputs:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

For the conditional form of this instruction, the input operands are restricted. Valid `XOP` and `YOP` registers are:

Xops	Yops
AR, MX0, MX1, MR0, MR1, MR2, SR0, SR1	MY0, MY1, SR1, 0

Output

`MR` — Multiplier Result register. Results are directly available for *x* input only in the next conditional ALU, MAC, or shifter operation or as *x* or *y* input in the next unconditional ALU, MAC, or shifter operation.

`SR` — Multiplier Feedback register. Results are directly available for either *x* (`SR0` and `SR1`) or *y* (`SR1` only) input in the next conditional ALU, MAC, or shifter operation or as *x* or *y* input in the next unconditional ALU, MAC, or shifter operation.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
MV (if MR used), SV (if SR used)	AZ, AN, AV, AC, AS, AQ, SS
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

This instruction provides a squaring operation (`XOP*XOP` and `DREG*DREG`) that performs single-cycle X^2 and ΣX^2 functions. In squaring operations, you must use the same register for both x-input operands.

You cannot use unconditional (`Dreg`) form of the multiply instruction in multifunction instructions.

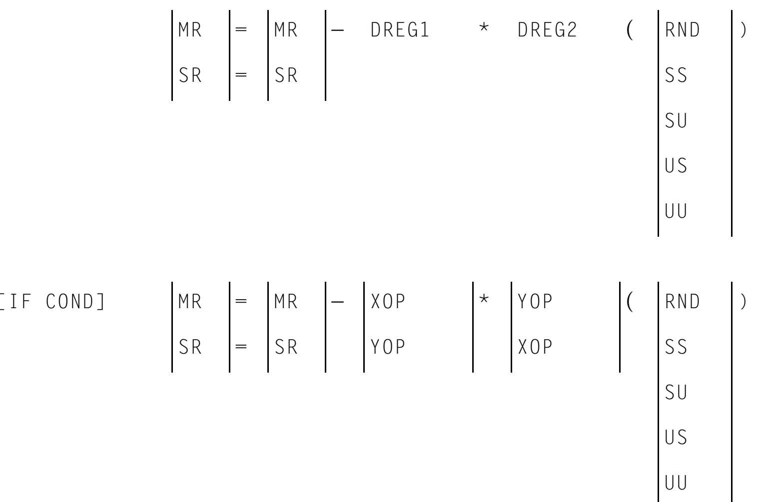
Examples

```
MR = MR + AX0 * SI (RND);      /* mult DREGs, rnd output, sum */
SR = SR + AX1 * MX0 (SS);      /* mult DREGs, sign input, sum */

IF MV MR = MR + MR0 * MY0 (SU);
                               /* mult X/Yops, un/sign in, sum */
IF MV MR = MR + MR2 * MX1 (UU);
                               /* mult X/Yops, unsign in, sum  */
CCODE = 0x09; NOP;             /* set CCODE for SV condition   */
IF SWCOND SR=SR+SR0*MY0 (US);
                               /* mult X/Yops, un/sign in, sum */
```

See Also

- “Type 9: Compute” on page 8-23
- “Condition Code (CCODE) Register” on page 1-5
- “Mode Status (MSTAT) Register” on page 1-8

Multiply with Cumulative Subtract



Function

Multiplies the input operands, subtracts the product from the current contents of the MR or SR register, and then stores the result in the corresponding destination register. Optionally, inputs may be signed or unsigned, and output may be rounded. For more information on input and output options, see "Data Format Options" on page 3-3 .

If execution is based on a condition, the multiplier performs the operation only when the condition evaluates true, and performs a NOP operation when the condition evaluates false. Omitting the condition forces unconditional execution of the instruction.

Input

For the unconditional form of this instruction, you can use any of these data registers for the `DREG` inputs:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

For the conditional form of this instruction, the input operands are restricted. Valid `XOP` and `YOP` registers are:

Xops	Yops
AR, MX0, MX1, MR0, MR1, MR2, SR0, SR1	MY0, MY1, SR1, 0

Output

`MR` Multiplier Result register. Results are directly available for *x* input only in the next conditional ALU, MAC, or shifter operation or as *x* or *y* input in the next unconditional ALU, MAC, or shifter operation.

`SR` Multiplier Feedback register. Results are directly available for *x* (`SR0` and `SR1`) or *y* (`SR1` only) input in the next conditional ALU, MAC, or shifter operation or as *x* or *y* input in the next unconditional ALU, MAC, or shifter operation.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
MV (if MR used), SV (if SR used)	AZ, AN, AV, AC, AS, AQ, SS
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

This instruction provides a squaring operation (XOP*XOP and DREG*DREG) that performs single-cycle X^2 and ΣX^2 functions. In squaring operations, you must use the same register for both *x* input operands.

You cannot use the unconditional (Dreg) form of the multiply instruction in multifunction instructions.

Examples

```
MR = MR - AX0 * SI (RND);      /* mult DREGs, rnd output, sub */
SR = SR - AX1 * MX0 (SS);      /* mult DREGs, sign input, sub */

IF MV MR = MR - MR0 * MY0 (SU); /* mult X/Yops, unsign in, sub */
IF MV MR = MR - MR2 * MX1 (UU); /* mult X/Yops, unsign in, sub */
CCODE = 0x09; NOP;              /* set CCODE for SV condition  */
IF SWCOND SR=SR-SR0*MY0 (US);   /* mult X/Yops, un/sign in, sub*/
```

See Also

- "Type 9: Compute" on page 8-23
- "Condition Code (CCODE) Register" on page 1-5
- "Mode Status (MSTAT) Register" on page 1-8

MAC Clear

```
[IF COND]                        |MR     |= 0 ;
                                 |SR     |
```

Function

Sets the specified register to 0.

If execution is based on a condition, the multiplier performs the operation only when the condition evaluates true, and performs a NOP operation when the condition evaluates false. Omitting the condition forces unconditional execution of the instruction.

Input

This instruction is a special case of XOP * YOP with the y-input operand set to 0.

Output

MR	Multiplier Result register. Results are directly available for *x* input only in the next conditional ALU, MAC, or shifter operation or as *x* or *y* input in the next unconditional ALU, MAC, or shifter operation.
SR	Multiplier Feedback register. Results are directly available for *x* (SR0 and SR1) or *y* (SR1 only) input in the next conditional ALU, MAC, or shifter operation or as *x* or *y* input in the next unconditional ALU, MAC, or shifter operation.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
MV (cleared if MR used), SV (cleared if SR used)	AZ, AN, AV, AC, AS, AQ, SS
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

See description in "Function" on page 3-19.

Examples

```
MR = 0;              /* clears MR */
SR = 0;              /* clears SR */
```

See Also

- "Type 9: Compute" on page 8-23
- "Condition Code (CCODE) Register" on page 1-5
- "Mode Status (MSTAT) Register" on page 1-8

MAC Round/Transfer

```
[IF COND]           MR    =   MR    (RND) ;

[IF COND]           SR    =   SR    (RND) ;
```

Function

Performs a multiply with cumulative add operation in which the *y*-input operand is `0` and the zero-product is added to the specified result register. Rounding `(RND)` directs the multiplier to round the entire 40-bit value stored in the result register (`MR` or `SR`).

If execution is based on a condition, the multiplier performs the operation only if the condition evaluates true, and it performs a `NOP` operation if the condition evaluates false. Omitting the condition forces unconditional execution of the instruction.

Input

This instruction is a special case of `MR|SR + XOP * YOP` with the *y*-input operand set to `0`.

Output

`MR`	Multiplier Result register. Results are directly available for *x* input only in the next conditional ALU, MAC, or shifter operation or as either *x* or *y* input in the next unconditional ALU, MAC, or shifter operation.
`SR`	Multiplier Feedback register. Results are directly available for either *x* (`SR0` and `SR1`) or *y* (`SR1` only) input in the next conditional ALU, MAC, or shifter operation or as either *x* or *y* input in the next unconditional ALU, MAC, or shifter operation.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
MV (if MR used), SV (if SR used)	AZ, AN, AV, AC, AS, AQ, SS
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

The BIASRND bit in the ICNTL register determines the rounding mode. Refer to "Rounding Modes" on page 3-4 for more information. For a complete description of the MAC Round/ Transfer instruction, see "Function" on page 3-19.

Examples

```
MR = MR (RND);                  /* round MR */
IF EQ SR = SR (RND);            /* round SR */
```

See Also

- "Type 9: Compute" on page 8-23
- "Condition Code (CCODE) Register" on page 1-5
- "Mode Status (MSTAT) Register" on page 1-8

MAC Saturate

```
SAT MR ;
SAT SR ;
```

Function

Tests the upper nine bits of the MR or SR register. If all nine bits have the same value, the multiplier performs a NOP operation. Otherwise, bit 23 controls whether the MR2:MR1:MR0 or SR2:SR1:SR0 registers are saturated to 0000:7FFF:FFFF or FFFF:8000:0000. This instruction works independently from the MV and SV bits.

Input

None.

Output

MR	Multiplier Result register. Results are directly available for *x* input only in the next conditional ALU, MAC, or shifter operation or as *x* or *y* input in the next unconditional ALU, MAC, or shifter operation.
SR	Multiplier Feedback register. Results are directly available for *x* (SR0 and SR1) or *y* (SR1 only) input in the next conditional ALU, MAC, or shifter operation or as *x* or *y* input in the next unconditional ALU, MAC, or shifter operation.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
	AZ, AN, AV, AC, AS, AQ, SS, MV, SV
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

The MAC saturation instruction provides control over a multiplication result that has overflowed or underflowed. It saturates the value in the specified register only for the cycle in which it executes. It does not enable a mode that continuously saturates results until disabled, like the ALU. Used at the end of a series of multiply and accumulate operations, the saturation instruction prevents the accumulator from overflowing.

For every operation it performs, the multiplier generates an overflow status signal `MV` (`SV` when `SR` is the specified result register), which is recorded in the `ASTAT` status register. `MV = 1` when the accumulator result, interpreted as a signed (twos complement) number, crosses the 32-bit boundary, spilling over from `MR1` into `MR2`. That is, the multiplier sets `MV = 1` when the upper nine bits in `MR` are anything other than all `0`s or all `1`s. Otherwise, it sets `MV = 0`.

Table 3-2. Saturation Status Bits & Result Registers

MV/SV	MSB of MR2/SR2	MR/SR Results
0	0	No change.
0	1	No change.
1	0	00000000 0111111111111111 1111111111111111
1	1	11111111 1000000000000000 0000000000000000

Do not permit the result to overflow beyond the MSB of MR2. Otherwise, the true sign bit of the result is irretrievably lost, and saturation may not produce a correct result. To reach this state, however, takes more than 255 overflows (MV type).

Examples

```
SAT MR;          /* saturate MR */
SAT SR;          /* saturate SR */
```

See Also

- "Type 9: Compute" on page 8-23
- "Condition Code (CCODE) Register" on page 1-5
- "Mode Status (MSTAT) Register" on page 1-8

Generate MAC Status Only: NONE

```
[NONE =] <MAC Operation> ;
```

Function

Performs the indicated unconditional MAC operation but does not load the results into the MR or SR result registers. Generates MAC status flags only. Use this instruction to set MAC status without disturbing the contents of the MR and SR result registers.

Input

XOP Limits the registers for the *x* input operand. Valid XOP registers are:

Xops
AR, MX0, MX1, MR0, MR1, MR2, SR0, SR1

YOP Limits the registers for the *x* input operand. Valid YOP registers are:

Yops
MY0, MY1, SR1, 0

Output

None. Generates MAC status flags only.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
MV	AZ, AN, AV, AC, AS, AQ, SS, SV
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

You can use any unconditional MAC operation to generate MAC status flags.

Examples

```
MX0 * MY0;       /* generate status from mult */
```

See Also

- "Type 8: Compute | Dreg1 «··· Dreg2" on page 8-22.

4 SHIFTER INSTRUCTIONS

The instruction set provides shifter instructions for performing shift operations on 16-bit input to yield 40-bit output. Combining these functions, programs can efficiently implement numeric format control, including full floating-point representation. Shifter operations include:

- "Arithmetic Shift" on page 4-6
- "Arithmetic Shift Immediate" on page 4-8
- "Logical Shift" on page 4-10
- "Logical Shift Immediate" on page 4-12
- "Normalize" on page 4-14
- "Normalize Immediate" on page 4-17
- "Exponent Derive" on page 4-20
- "Exponent (Block) Adjust" on page 4-23

This chapter describes the individual shifter instructions and the following related topics:

- "Shifter Registers" on page 4-2
- "Shifter Instruction Options" on page 4-3

- "Shifter Status Flags" on page 4-5
- "Denormalization" on page 4-26

For details on condition codes, see "Condition Code (CCODE) Register" on page 1-5.

Shifter Operation Conventions

Shifter Registers

As shown in Table 4-1, the shifter has five registers.

Table 4-1. Summary of Shifter Registers

Name	Size	Description
SR0	16 bits	Shifter Result register (low word). SR denotes SR0, SR1, and SR2 combined, which hold the 40-bit shifter result.
SR1	16 bits	Shifter Result register (middle word). This register also functions as the multiplier's y-input feedback register. SR denotes SR0, SR1, and SR2 combined, which hold the 40-bit shifter result.
SR2	16/8 bits	Shifter Result register (high byte). SR denotes SR0, SR1, and SR2 combined, which hold the 40-bit shifter result. Although this register is 16 bits wide, for shifter operations, only the lower eight bits are used.
SB	16/5 bits	Shifter Block Exponent register. Although this register is 16 bits wide, for shifter operations, only the lower five bits are used. Contains the effective exponent derived from the number with greatest magnitude in a block of numbers. This value provides the shift code for all numbers in the block in subsequent `NORM` or x`SHIFT` instructions. The value in this register is sign-extended to form a 16-bit value when transferred to memory or to other data registers. Non-shifter instructions can use SB for a 16-bit scratch register.

Table 4-1. Summary of Shifter Registers (Cont'd)

Name	Size	Description
SE	16/8 bits	Shifter Exponent register. Although this register is 16 bits wide, for shifter operations, only the lower eight bits are used. Contains the effective exponent derived from the input data. This value provides the shift code for a subsequent NORM or xSHIFT instruction. The value in this register is sign-extended to form a 16-bit value when transferred to memory or to other data registers. Non-shifter instructions can use SE for a 16-bit scratch register.
SI	16 bits	Shifter Input register. Provides single-precision twos complement input to shifter instructions.

When a shifter operation writes data to SR1, it sign-extends the value into SR2, overwriting the previous contents of SR2.

The SB and SE registers are the result registers for the block exponent adjust (EXPADJ) operation and derive exponent (EXP) operation, respectively. The shifter input register (SI) supplies single-precision input data to any shifter operation, except EXPADJ. To input the result from an ALU or MAC operation directly, use the appropriate result register—SR, AR, or MR.

Shifter Instruction Options

Almost all shifter instructions have two to three options: (HI), (LO), and (HIX). Each option enables a different exponent detector mode that operates only while the instruction executes. The shifter interprets and handles the input data according to the selected mode.

For the derive exponent (EXP) and block exponent adjust (EXPADJ) operations, the shifter calculates the shift code—the direction and number of bits to shift—and then stores that value in the SE register. For the ASHIFT, LSHIFT, and NORM operations, supply the value of the shift code directly to the SE or SB registers or use the result of a previous EXP or EXPADJ operation.

For the `ASHIFT`, `LSHIFT`, and `NORM` operations:

`(HI)`	Operation references the lower half of the output field.
`(LO)`	Operation references the upper half of the output field.

For the exponent derive (`EXP`) operation:

`(HIX)`	Use this mode for shifts and normalization of results from ALU operations. Input data is the result of an add or subtract operation that may have overflowed. The shifter examines the ALU overflow bit `AV`. If `AV=1`, the effective exponent of the input is `+1` (this value indicates that overflowed occurred before the `EXP` operation executed). If `AV=0`, no overflow occurred and the shifter performs the same operations as the `(HI)` mode.
`(HI)`	Input data is a single-precision signed number or the upper half of a double-precision signed number. The number of leading sign bits in the input operand, which equals the number of sign bits minus one, determines the shift code. By default, the `EXPADJ` operation always operates in this mode.
`(LO)`	Input data is the lower half of a double-precision signed number. To derive the exponent on a double-precision number, perform the `EXP` operation twice, once on the upper half of the input, and once on the lower half. For details, "Exponent Derive" on page 4-20.

Shifter Status Flags

Two status flags in the ASTAT register (SS and SV) record the status of shifter operations.

SS	Records the sign of the shifter input operand SS = 0 positive (+) input SS = 1 negative (–) input
SV	Records overflow or underflow status SV = 0 no overflow or underflow occurred SV = 1 overflow or underflow occurred

Arithmetic Shift

```
[IF COND]     SR   =  [SR OR]    ASHIFT DREG (   |HI  | ) ;
                                                 |LO  |
```

Function

Arithmetically shifts the bits of the operand by the amount (number of bits) and direction specified by the shift code (value) in the SE register. A positive value produces a left (up) shift, and a negative value produces a right (down) shift. Optionally, the shift can be based on a half of the 32-bit output field being shifted. For more information on output options, see "Shifter Instruction Options" on page 4-3.

If execution is based on a condition, the shifter performs the operation only if the condition evaluates true, and it performs a NOP operation if the condition evaluates false. Omitting the condition forces unconditional execution of the instruction.

With the SR OR option selected, the shifter ORs the shifted output with the current contents of the SR register and stores that value in SR. Otherwise, it overwrites the current contents of the SR register with the shifted output.

Input

The input operand, the value to shift, is supplied in a data register. Use any of these data registers for the DREG inputs:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

Output

SR	Shifter Result register contains 40-bit result.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
SV	AZ, AN, AV, AC, AS, AQ, SS, MV
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

The shifter sign-extends the 40-bit result to the left, replicating the MSB of the input, and zero-fills the 40-bit result from the right. Bits shifted out past either boundary (SR_{39} or SR_0) are dropped.

To shift a double-precision number, shift both halves of the input data separately, using the same shift code value for both halves. ASHIFT the upper half of the input data, but LSHIFT the lower half. The first cycle, ASHIFT the upper half of the input using the (HI) option. The second cycle, LSHIFT the lower half using both the (LO) and SR OR options. Using these options prevents the shifter from sign-extending the MSB of the low word and prevents overwriting the output (upper word) from the previous ASHIFT operation.

Examples

```
AR = 3; SE = AR;        /* shift code, left shift 3 bits */
SI = 0xB6A3;            /* value of upper word of input data */
SR = ASHIFT SI (HI);    /* arithmetically shift high word */
```

See Also

- "Type 16: Shift Reg0" on page 8-34
- "Condition Code (CCODE) Register" on page 1-5
- "Mode Status (MSTAT) Register" on page 1-8

Arithmetic Shift Immediate

```
SR   =  [SR OR]    ASHIFT DREG      BY <imm8> (      |HI  | ) ;
                                                     |LO  |
```

Function

Arithmetically shifts the bits of the operand by the amount (number of bits) and direction specified by the immediate value. Valid immediate values range from −128 to 127. A positive value produces a left (up) shift, and a negative value produces a right (down) shift. Optionally, the shift can be based on a half of the 32-bit output field being shifted. For more information on output options, see "Shifter Instruction Options" on page 4-3.

With the SR OR option selected, the shifter ORs the shifted output with the current contents of the SR register and stores that value in SR. Otherwise, it overwrites the current contents of the SR register with the shifted output.

Input

The input operand (the value to shift) is supplied in a data register. Use any of these data registers for the DREG inputs:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

Output

SR	Shifter Result register contains 40-bit result.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
SV	AZ, AN, AV, AC, AS, AQ, SS, MV
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

The shifter sign-extends the 40-bit result to the left, replicating the MSB of the input, and zero-fills the 40-bit result from the right. Bits shifted out past either boundary (SR_{39} or SR_0) are dropped.

To shift a double-precision number, shift both halves of the input data separately, using the same immediate value for both halves. ASHIFT the upper half of the input data, but LSHIFT the lower half. For the first cycle, ASHIFT the upper half of the input using the (HI) option. For the second cycle, LSHIFT the lower half using both the (LO) and SR OR options. These options prevent the shifter from sign-extending the MSB of the low word and from overwriting the output (upper word) from the previous ASHIFT operation.

Examples

```
SI = 0xB6A3;                 /* upper word of input data */
SR = ASHIFT SI BY 3 (HI);    /* arithmetically shift upper word */
```

See Also

- “Type 15: Shift Data8” on page 8-33
- “Condition Code (CCODE) Register” on page 1-5
- “Mode Status (MSTAT) Register” on page 1-8

Logical Shift

```
[IF COND]    SR  =  [SR OR]    LSHIFT    DREG    ( |HI | )  ;
                                                   |LO |
```

Function

Logically shifts the bits of the operand by the amount (number of bits) and direction specified by the shift code (value) in the SE register. A positive value produces a left (up) shift, and a negative value produces a right (down) shift. Optionally, the shift can be based on a half of the 32-bit output field being shifted. For more information on output options, see "Shifter Instruction Options" on page 4-3.

If execution is based on a condition, the shifter performs the operation only if the condition evaluates true, and it performs a NOP operation if the condition evaluates false. Omitting the condition forces unconditional execution of the instruction.

With the SR OR option selected, the shifter ORs the shifted output with the current contents of the SR register and stores that value in SR. Otherwise, it overwrites the current contents of the SR register with the shifted output.

Input

The input operand (the value to shift) is supplied in a data register. Use any of these data registers for the DREG inputs:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

Output

SR	Shifter Result register contains 40-bit result.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
SV	AZ, AN, AV, AC, AS, AQ, SS, MV
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

For left shifts (positive shift code), the shifter zero-fills the 40-bit result from the right. Bits shifted out past the high-order boundary (SR_{39}) are dropped.

For right shifts (negative shift code), the shifter zero-fills the 40-bit result from the left. Bits shifted out past the low-order boundary (SR_0) are dropped.

To shift a double-precision number, shift both halves of the input data separately, using the same shift code value for both halves. For the first cycle, `LSHIFT` the upper half of the input using the `(HI)` option. For the second cycle, `LSHIFT` the lower half using both the `(LO)` and `SR OR` options. These options prevent the shifter from overwriting the result (upper word) from the previous `LSHIFT` operation.

Examples

```
AR = 3; SE = AR;             /* shift code left shift 3 bits */
AX0 = 0x765D;                /* lower word of input data */
SR = SR OR LSHIFT AX0(LO);   /* logically shift low word */
```

See Also

- "Type 16: Shift Reg0" on page 8-34
- "Condition Code (CCODE) Register" on page 1-5
- "Mode Status (MSTAT) Register" on page 1-8

Logical Shift Immediate

```
SR   =  [SR OR]     LSHIFT BY <imm8>          ( |HI | ) ;
                                                |LO |
```

Function

Logically shifts the bits of the operand by the amount (number of bits) and direction specified by the immediate value. Valid immediate values range from -128 to 127. A positive value produces a left (up) shift, and a negative value produces a right (down) shift. Optionally, the shift can be based on a half of the 32-bit output field being shifted. For more information on output options, see "Shifter Instruction Options" on page 4-3.

With the SR OR option selected, the shifter ORs the shifted output with the current contents of the SR register and stores that value in SR. Otherwise, it overwrites the current contents of the SR register with the shifted output.

Input

The input operand (the value to shift) is supplied in a data register. Use any of these data registers for the DREG inputs:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

Output

SR Shifter Result register contains 40-bit result.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
SV	AZ, AN, AV, AC, AS, AQ, SS, MV
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

For left shifts (positive shift code), the shifter zero-fills the 40-bit result from the right. Bits shifted out past the high-order boundary (SR_{39}) are dropped.

For right shifts (negative shift code), the shifter zero-fills the 40-bit result from the left. Bits shifted out past the low-order boundary (SR_0) are dropped.

To shift a double-precision number, shift both halves of the input data separately, using the same shift code value for both halves. For the first cycle, LSHIFT the upper half of the input using the (HI) option. For the second cycle, LSHIFT the lower half using both the (LO) and SR OR options. These options prevent the shifter from overwriting the result (upper word) from the previous LSHIFT operation.

Examples

```
SI = 0xFF6A;                       /* single-precision input */
SR = SR OR LSHIFTSI BY 3 (LO);     /* logically shift low word */
```

See Also

- “Type 15: Shift Data8” on page 8-33
- “Condition Code (CCODE) Register” on page 1-5
- “Mode Status (MSTAT) Register” on page 1-8

Normalize

```
[IF COND]    SR  =  [SR OR]    NORM DREG    ( |HI| ) ;
                                              |LO|
```

Function

Normalization, in essence, is a fixed- to floating-point conversion operation that produces an exponent and a mantissa. Optionally, the operation can be based on a half of the 32-bit output field being shifted. For more information on output options, see "Shifter Instruction Options" on page 4-3.

Normalization using this instruction is a two-step process that requires:

- The EXP instruction to derive the exponent for the shift code.
- The NORM instruction to shift the twos complement input by the calculated shift code, removing its redundant sign bits and aligning its true sign bit to the high-order bit of the output field.

The EXP operation calculates the number of redundant sign bits in the input and stores the negative of that value in SE. The NORM operation negates the value in SE again to generate a positive shift code, ensuring that the input is shifted left.

If execution is based on a condition, the shifter performs the operation only if the condition evaluates true, and it performs a NOP operation if the condition evaluates false. Omitting the condition forces unconditional execution of the instruction.

With the SR OR option selected, the shifter ORs the shifted output with the current contents of the SR register and stores that value in SR. Otherwise, it overwrites the current contents of the SR register with the shifted output.

Input

The input operand (the value to shift) is supplied in a data register. Use any of these data registers for the DREG inputs:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

Output

SR Shifter Result register contains 40-bit result.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
SV	AZ, AN, AV, AC, AS, AQ, SS, MV
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

The (HI) and (LO) options determine how unused bits in the 40-bit output are filled. When HI is selected, the shifter zero-fills the 40-bit result from the right. When LO is selected, the shifter zero-fills the 40-bit result to the left. Bits shifted out past the high-order boundary (SR_{39}) are dropped.

To normalize a double-precision number, normalize both halves of the input data separately, using the same shift code value for both halves. First, use the EXP instruction to derive the exponent to use for the shift code. Then, in the first normalization cycle, NORM the upper half of the input using the (HI) option. At the next cycle, NORM the lower half using both the (LO) and SR OR options. These options prevent the shifter from overwriting the result (upper word) from the previous NORM operation.

Examples

```
/* Normalize double-precision twos complement data: */
AX1 = 0xF6D4;               /* load hi 2s comp data in dreg */
AX0 = 0x04A2;               /* load lo 2s comp data in dreg */
SE = EXP AX1 (HI);          /* derive exponent on hi word */
SE = EXP AX0 (LO);          /* derive exponent on lo word */
SR = NORM AX1 (HI);         /* normalize hi word */
SR = SR OR NORM AX0 (LO);   /* normalize lo word */
```

See Also

- "Type 16: Shift Reg0" on page 8-34
- "Denormalization" on page 4-26
- "Condition Code (CCODE) Register" on page 1-5
- "Mode Status (MSTAT) Register" on page 1-8

Normalize Immediate

```
SR   =  [SR OR]    NORM DREG BY <imm8>          ( |HI | ) ;
                                                  |LO |
```

Function

Normalization, in essence, is a fixed- to floating-point conversion operation that produces an exponent and a mantissa. Using a positive constant for the shift code, it is a one-step process that shifts the twos complement input left by the specified amount, removing its redundant sign bits and aligning its true sign bit to the high-order bit of the output field. Optionally, the operation can be based on a half of the 32-bit output field being shifted. For more information on output options, see "Shifter Instruction Options" on page 4-3.

With the SR OR option selected, the shifter ORs the shifted output with the current contents of the SR register and stores that value in SR. Otherwise, it overwrites the current contents of the SR register with the shifted output.

Input

The input operand, the value to shift, is supplied in a data register. You can use any of these data registers for the DREG inputs:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

Output

SR — Shifter Result register contains 40-bit result.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
SV	AZ, AN, AV, AC, AS, AQ, SS, MV
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

The (HI) and (LO) options determine how unused bits in the 40-bit output are filled. When HI is selected, the shifter zero-fills the 40-bit result from the right. When LO is selected, the shifter zero-fills the 40-bit result to the left. Bits shifted out past the high-order boundary (SR_{39}) are dropped.

To normalize a double-precision number, normalize both halves of the input data separately, using the immediate value for both halves. In the first normalization cycle, NORM the upper half of the input using the (HI) option. In the next cycle, NORM the lower half using both the (LO) and SR OR options. These options prevent the shifter from overwriting the result (upper word) from the previous NORM operation.

Examples

```
/* Normalize a double-precision, twos complement data: */
AX1 = 0xF6D4;              /* load hi 2s comp data in dreg */
AX0 = 0x04A2;              /* load lo 2s comp data in dreg */
SR = NORM AX1 BY 2 (HI);   /* normalize hi word */
SR = SR OR NORM AX0 BY 2 (LO);      /* normalize lo word */
```

See Also

- "Type 15: Shift Data8" on page 8-33
- "Denormalization" on page 4-26
- "Condition Code (CCODE) Register" on page 1-5
- "Mode Status (MSTAT) Register" on page 1-8

Exponent Derive

```
[IF COND]          SE = EXP DREG            ( |HIX |) ;
                                              |HI  |
                                              |LO  |
```

Function

Derives the effective exponent of the input operand to generate the shift code value for use in a subsequent normalization operation. The instruction option (HIX, HI, or LO) determines the resulting shift code. For more information on output options, see "Shifter Instruction Options" on page 4-3.

If execution is based on a condition, the shifter performs the operation only if the condition evaluates true, and it performs a NOP operation if the condition evaluates false. Omitting the condition forces unconditional execution of the instruction.

Input

The input operand (the value to shift) is supplied in a data register. Use any of these data registers for the DREG inputs:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

Output

SE	Shifter Exponent register contains the 8-bit shift code.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
SS (Affected by operations using the HI and HIX options only. Set by the MSB of the input data when AV = 0. In HIX mode only, set by the inverted MSB of the input data when AV = 1.)	AZ, AN, AV, AC, AS, AQ, MV, SV
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

Use the LO option only to derive the exponent for the low word in a double-precision twos complement number. Before doing so, you must derive the exponent on the high word using either the HI or the HIX option. The result of the EXP operation on the upper half determines the shift code of the lower half. Unless the upper half contains all sign bits, the SE register contains the correct shift code to use for both EXP (HI/HIX) and (LO) operations. If the upper half does contain all sign bits, EXP (LO) totals the number of sign bits in the double-precision word and stores that value in SE.

Examples

```
           /* Normalize double-precision twos complement data: */
AX1 = 0xF6D4;                /* load hi 2s comp data in dreg */
AX0 = 0x04A2;                /* load lo 2s comp data in dreg */
SE = EXP AX1 (HI);           /* derive exponent on hi word */
SE = EXP AX0 (LO);           /* derive exponent on lo word */
SR = NORM AX1 (HI);          /* normalize hi word */
SR = SR OR NORM AX0 (LO);    /* normalize lo word */
```

See Also

- "Type 16: Shift Reg0" on page 8-34
- "Condition Code (CCODE) Register" on page 1-5
- "Mode Status (MSTAT) Register" on page 1-8

Exponent (Block) Adjust

```
[IF COND] SB = EXPADJ DREG ;
```

Function

Derives the effective exponent of the number of largest magnitude in a block of numbers. Use this value for the shift code in subsequent NORM instructions to normalize each number in the block.

If execution is based on a condition, the shifter performs the operation only if the condition evaluates true, and it performs a NOP operation if the condition evaluates false. Omitting the condition forces unconditional execution of the instruction.

Input

The input operand (the value to shift) is supplied in a data register. Use any of these data registers for the DREG inputs:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

Output

SB Shifter Block Exponent register contains the 5-bit exponent value.

You must initialize SB to –16 before issuing the first EXPADJ instruction in the series.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
	AZ, AN, AV, AC, AS, AQ, SS, MV, SV
For information on these status bits in the ASTAT register, see Table 1-2 on page 1-4.	

Details

This instruction operates in `HI` mode to derive the exponent. It works on double-precision, twos complement input only. Possible values for the result of the `EXPADJ` operation range from –`15` to `0`.

To derive the effective exponent for a block of numbers:

1. Initialize the `SB` register to `-16`.

   ```
   SB = -16
   ```

   This value falls below the range of possible exponent values.

   At the end of the last `EXPADJ` operation, `SB` contains the exponent of the number of largest magnitude in the block.

2. For each number in the block, derive the effective exponent.

   ```
   SB = EXPADJ DREGx;
   ```

   For the first operation, the shifter derives the exponent and stores it in `SB`.

   For each subsequent operation, the shifter derives the exponent and compares the new value with the current value of `SB`. If the new value is greater than the current value, the shifter stores the new value in `SB`, overwriting the old value. Otherwise, the shifter discards the new value and the contents of `SB` remain unchanged.

3. Transfer the contents of SB to SE.

   ```
   SE = SB;
   ```

   SE now contains the shift code to use in subsequent NORM operations to normalize each of the numbers in the block. For details, see "Normalize" on page 4-14.

   Alternatively, you can save the exponent in a data register for use later in your program.

Examples

```
/* Normalize double-precision twos complement data: */
AX1 = 0xF6D4;              /* load hi 2s comp data in dreg */
AX0 = 0x04A2;              /* load lo 2s comp data in dreg */
SB = -16;                  /* initialize SB */
SB = EXPADJ AX1;
SB = EXPADJ AX0;
SE = SB;                   /* load block adjusted exp */
SR = NORM AX1 (HI);        /* normalize hi word */
SR = SR OR NORM AX0 (LO);  /* normalize lo word */
```

See Also

- "Type 16: Shift Reg0" on page 8-34
- "Condition Code (CCODE) Register" on page 1-5
- "Mode Status (MSTAT) Register" on page 1-8

Denormalization

Function

Denormalization is a shift function in which a predefined exponent defines the amount and direction of the shift. In essence, denormalization is a floating- to fixed-point conversion operation. It requires a series of shifter operations:

- Use the EXP instruction to derive the exponent the shift code, or SE explicitly loaded with the exponent value. SE must contain the shift value; for denormalization, you cannot use ASHIFT/LSHIFT with an immediate value.
- Use the ASHIFT instruction to shift a single-precision number or the high word of a double-precision number.
- Use the LSHIFT instruction to shift the low word when denormalizing a double-precision number.

Denormalize a double-precision, twos complement number:

```
MX1 = -3;                          /* generate shift code */
SE = MX1;                          /* load value in SE register */
AX1 = 0xB6A3;                      /* load high word of input */
AX0 = 0x765D;                      /* load low word of input */
SR = ASHIFT AX1(HI);               /* arith shift high word */
SR = SR OR LSHIFT AX0(LO);         /* logically shift low word */
```

You can reverse shift order, but you must always arithmetically shift the high word of a double-precision number:

```
MX1 = -3;                          /* generate shift code */
SE = MX1;                          /* load value in SE register */
AX1 = 0xB6A3;                      /* load high word of input */
AX0 = 0x765D;                      /* load low word of input */
```

```
SR = LSHIFT AX0(LO);              /* logically shift low word */
SR = SR OR ASHIFT AX1(HI);        /* arith shift high word */
```

5 MULTIFUNCTION INSTRUCTIONS

The instruction set provides multifunction instructions—multiple instructions within a single instruction cycle. Multifunction instructions can perform (in a single cycle) computations in parallel with data move operations. Multifunction instructions are combinations of single instructions delimited with commas and ended with a semicolon, as in:

```
AR = AX0 - AY0, AX0 = MR1;  /* ALU sub and reg-to-reg move */
```

These operations are the basis for all high-performance DSP functions and take advantage of the DSP's inherent parallelism.

This chapter describes each of the multifunction instructions and the order of execution of multifunction operations.

The multifunction operations include:

- "Order of Execution of Multifunction Operations" on page 5-2
- "Multifunction Instruction Reference" on page 5-3

Multifunction instructions combine compute operations with data move operations. The multifunction combinations have no status flags specifically associated with them, but the DSP does update the status flags for the computations that appear within multifunction instructions. For details, see "Arithmetic Status (ASTAT) Register" on page 1-3.

Order of Execution of Multifunction Operations

The DSP reads registers and memory at the beginning of the processor pipeline and writes them at the end of it. Normal instruction syntax, read from left to right, implies this functional ordering. For example:

a) `MR = MR+MX0*MY0(UU), MX0 = DM(I0+=M0), MY0 = PM(I4+=M4);`

b) `DM(I0+=M0) = AR, AR = AX0+AY0;`

c) `AR = AX0 - AY0, AX0 = MR1;`

For memory reads, the DSP executes the computation first, using the current value of the input data registers, and then transfers new data from memory or from another data register, overwriting the contents of the data registers (a and c).

For memory writes, the DSP transfers the current value from the data register to memory first, and then overwrites the data register with the result of the computation (b).

Even if you alter the order of the operations in your code, execution occurs in the correct order; the assembler issues a warning (if enabled), but results are correct at the opcode level. For example:

```
MX0 = DM(I0+=M0), MY0 = PM(I4+=M4), MR = MR+MX0*MY0(UU);
```

The altered order of operations appears to reverse the order in which the DSP executes the operations, but the DSP always executes instructions using read-first/write-last logic.

The DSP's read-first/write-last logic enables you to use the same data register in more than one multifunction operation. The same data register can serve as an input operand into the computation and as the destination or source register for a data move operation. However, except for the

compute with memory write instruction, the same register cannot serve as destination for more than one multifunction operation. Doing so generates unpredictable and erroneous results.

Multifunction Instruction Reference

The multifunction operations include:

- "Compute with Dual Memory Read" on page 5-4
- "Dual Memory Read" on page 5-8
- "Compute with Memory Read" on page 5-11
- "Compute with Memory Write" on page 5-15
- "Compute with Register-to-Register Move" on page 5-19

Compute with Dual Memory Read

<ALU> <MAC>	,	AX0 AX1 MX0 MX1	= DM(	I0 I1 I2 I3	+=	M0 M1 M2 M3	),	AY0 AY1 MY0 MY1	= PM(	I4 I5 I6 I7	+=	M4 M5 M6 M7	);

Function

Combines an ALU or MAC operation with a read from memory over the 16-bit DM bus and another read from memory over the 24-bit PM bus. The restricted register forms—*using* XOP *and* YOP *registers, not the* DREG *register file*—of all ALU or MAC instructions are supported, except for the MAC saturate instruction (SAT) and the divide primitives (DIVS and DIVQ). Also, the multifunction ALU and MAC instructions may not use conditional (IF) options.

The compute operation executes first, using the current contents of the data registers as input operands. Memory read operations execute next, overwriting the contents of the destination data registers with new data from memory.

The destination of both memory read operations is an ALU or MAC data register. The DM bus read loads an ALU or MAC XOP register, and the PM bus read loads an ALU or MAC YOP register. The memory data is always right-justified in the destination data register.

Input

The input operands for the compute operation are specific to the particular operation. For details, see the compute instruction's individual description.

- For MAC operations, see "MAC Instructions" on page 3-1.
- For ALU operations, see "Multifunction Instructions" on page 5-1.

Both data move operations use two DAG registers, index (Ireg) and modify (Mreg), to generate memory addresses—DAG1 registers for DM bus access, and DAG2 registers for PM bus access. For details on DAG registers and data addressing, see “Data Move Instructions” on page 6-1.

DM/DAG1	I0, I1, I2, or I3 (index registers) M0, M1, M2, or M3 (modify registers)
PM/DAG2	I4, I5, I6, or I7 (index registers) M4, M5, M6, or M7 (modify registers)

You can use any index register with any modify register from the same DAG. You cannot pair a DAG1 register with a DAG2 register.

Output

The result register for the compute operation is always the computation unit’s result registers.

AR	ALU operations
MR	MAC operations

SR is not a MAC result register for this multifunction instruction.

The destination register for both data move operations is an ALU or MAC data register—an XOP register for DM bus access and a YOP register for PM bus access.

XOP register:	AX0, AX1, MX0, or MX1
YOP register:	AY0, AY1, MY0, or MY1

Status Flags

The status flags generated as a result of the computation depend on the compute operation the instruction performs. For more information, see the status flags section of the computation's reference page.

Details

Memory read operations use register indirect addressing with postmodify (`Ireg += Mreg`). For linear indirect addressing, you must initialize the `Lx` register of the corresponding `Ireg` register to `0`. For circular indirect addressing, set the buffer's length and base address with the corresponding `Lreg` and `Breg` registers. For more information on addressing, see the *ADSP-219x/2192 DSP Hardware Reference*.

The `DM()` reference uses the 16-bit DM bus, and the `PM()` reference uses the 24-bit PM bus. For `PM` data moves, the destination data register receives the 16 MSBs from 24-bit memory, and the `PX` register catches the eight LSBs. To use all 24 bits of memory data, transfer the eight LSBs from `PX` to another data register; otherwise, the eight LSBs will be lost.

The address of the access, not the `PM()` or `DM()` reference, selects the memory bank. So, the `DM` reference can access 24-bit memory, and the `PM` reference can access 16-bit memory. `DM` reads of 24-bit memory result in the specified data register receiving bits `23:8` from memory. `PM` reads of 16-bit memory result in the specified data register receiving bits `23:8` from memory. When the `PX` register is loaded using a 16-bit memory access (`PM` reference to 16-bit memory or `DM` reference to 24-bit memory), the DSP clears (=0)the eight LSBs of `PX` register.

This multifunction instruction requires the DSP to fetch three items from memory: the instruction and two data words. The number of cycles required to execute it depends on whether the instruction generates bus conflicts as shown in the following table.

Execution	Conditions
1 cycle	If the instruction is already cached and the data are from different memory banks
2 cycles	If only one bus conflict occurs—data vs. data or instruction vs. data
3 cycles	If two bus conflicts occur—instruction vs. data vs. data

Examples

```
AR = AX0 - AY0,
MX1 = DM(I3 += M0),
MY1 = PM(I5 += M4); /* sub and dual read */

AR = AX0 + AY0,
MX0 = DM(I1 += M0),
MY0 = PM(I4 += M4); /* add and dual read */

MR = MX0 * MY0 (SS),
MX0 = DM(I2 += M2),
MY0 = PM(I7 += M7); /* mult and dual read */
```

See Also

- "Type 1: Compute | DregX«···DM | DregY«···PM" on page 8-17
- "Multifunction Instructions" on page 5-1
- "MAC Instructions" on page 3-1
- "Shifter Instructions" on page 4-1
- "Arithmetic Status (ASTAT) Register" on page 1-3

Dual Memory Read

AX0	=	DM(	I0	+=	M0	),	AY0	=	PM(	I4	+=	M4	);
AX1			I1		M1		AY1			I5		M5	
MX0			I2		M2		MY0			I6		M6	
MX1			I3		M3		MY1			I7		M7	

Function

Performs two memory read operations, one over the 16-bit DM bus and the other over the 24-bit PM bus.

Each read operation moves the contents of the specified memory location to its respective destination register. The destination of both memory read operations is an ALU or MAC data register. The DM bus read loads an ALU or MAC `DREGx` register, and the PM bus read loads an ALU or MAC `DREGy` register. The memory data is always right-justified in the destination data register.

Input

Both data move operations use two DAG registers, index (`Ireg`) and modify (`Mreg`), to generate memory addresses—DAG1 registers for DM bus access, and DAG2 registers for PM bus access. For details on DAG registers and data addressing, see "Data Move Instructions" on page 6-1.

DM/DAG1	`I0`, `I1`, `I2`, or `I3` (index registers) `M0`, `M1`, `M2`, or `M3` (modify registers)
PM/DAG2	`I4`, `I5`, `I6`, or `I7` (index registers) `M4`, `M5`, `M6`, or `M7` (modify registers)

You can use any index register with any modify register from the same DAG. You cannot pair a DAG1 register with a DAG2 register.

Output

The destination register for both data move operations is an ALU or MAC data register—an XOP register for DM bus access and a YOP register for PM bus access.

XOP	AX0, AX1, MX0, or MX1
YOP	AY0, AY1, MY0, or MY1

Status Flags

None affected.

Details

Memory read operations use register indirect addressing with postmodify (Ireg += Mreg). For linear indirect addressing, initialize the Lreg register of the corresponding Ireg register to 0. For circular indirect addressing, set the buffer's length and base address with the corresponding Lreg and Breg registers. For more information on addressing, see the *ADSP-219x/2192 DSP Hardware Reference*.

The DM reference uses the 16-bit DM bus, and the PM reference uses the 24-bit PM bus. For PM data moves, the destination data register receives the 16 MSBs from 24-bit memory, and the PX register catches the eight LSBs. To use all 24 bits of the memory data, transfer the eight LSBs from PX to another data register; otherwise, the eight LSBs will be lost.

The address of the access, not the PM() or DM() reference, selects the memory bank. So, the DM() reference can access 24-bit memory, and the PM() reference can access 16-bit memory. DM reads of 24-bit memory result in the specified data register receiving bits 23:8 from memory. PM reads of 16-bit memory result in the specified data register receiving bits 23:8 from memory. When the PX register is loaded using a 16-bit memory access (PM reference to 16-bit memory or DM reference to 24-bit memory), the DSP clears (=0) the eight LSBs of PX.

This multifunction instruction requires the DSP to fetch three items from memory: the instruction and two data words. The number of cycles required to execute it depends on whether the instruction generates bus conflicts:

Execution	Conditions
1 cycle	If the instruction is already cached and the data are from different memory banks
2 cycles	If only one bus conflict occurs—data vs. data or instruction vs. data
3 cycles	If two bus conflicts occur—instruction vs. data vs. data

Examples

```
MX0 = DM(I0+=M0),
MY0 = PM(I5+=M4);                /* dual read */

AX1 = DM(I3+=M0),
AY1 = PM(I6+=M4);                /* dual read */
```

See Also

- “Type 1: Compute | DregX«···DM | DregY«···PM” on page 8-17

Compute with Memory Read

```
<ALU>      , DREG = DM  ( I0  +=  M0  );
<MAC>                     I1      M1
<SHIFT>                   I2      M2
                          I3      M3
                          I4      M4
                          I5      M5
                          I6      M6
                          I7      M7
```

Function

Combines an ALU, MAC, or shifter operation with a 16-bit read from memory over the DM bus. The restricted register forms—*using* XOP *and* YOP *registers, not the* DREG *register file*—of all ALU, MAC, or shifter instructions are supported, except for the MAC saturate instruction (SAT), the divide primitives (DIVS and DIVQ), and shift immediate. Also, the multifunction ALU, MAC, and shifter instructions may not use conditional (IF) options.

The compute operation executes first, using the current contents of the data registers as input operands. Memory read operation executes next, overwriting the contents of the destination data register with new data from memory.

The read operation moves the contents of the memory location to the specified destination register. The destination of the memory read operation is an ALU, MAC, or shifter data register. The memory data is always right-justified in the destination data register.

Input

Valid input operands for the compute depend on the operation's computation unit. For more information, see the input descriptions in "Multifunction Instructions" on page 5-1, "Shifter Instructions" on page 4-1, and "MAC Instructions" on page 3-1.

The data move operation uses two DAG registers, index (`Ireg`) and modify (`Mreg`), to generate memory addresses. Regardless of the DAG registers used, all accesses occur over the DM bus. For details on DAG registers and data addressing, see "Data Move Instructions" on page 6-1.

DAG1	`I0`, `I1`, `I2`, or `I3` (index registers) `M0`, `M1`, `M2`, or `M3` (modify registers)
DAG2	`I4`, `I5`, `I6`, or `I7` (index registers) `M4`, `M5`, `M6`, or `M7` (modify registers)

You can use any index register with any modify register from the same DAG. You cannot pair a DAG1 register with a DAG2 register.

Output

The result register for the compute operation is always the computation unit's result or feedback register.

`AR/AF`	ALU operations
`MR/SR`	MAC operations
`SR/SE`	Shifter operations

The destination register for the data move operation is any register file data register. Use any of these data registers for the `DREG` destination:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

Status Flags

The status flags generated as a result of the computation depend on the compute operation the instruction performs. For more information, see the status flags section of the computation's reference page.

Details

The memory read operation uses indirect addressing with postmodify (Ireg += Mreg) and always accesses 16-bit data over the DM bus. For linear indirect addressing, initialize the Lx register of the corresponding Ireg register to 0. For circular indirect addressing, set the buffer's length and base address with the corresponding Lreg and Breg registers. For more information on addressing, see the *ADSP-219x/2192 DSP Hardware Reference*.

Since data accesses occur over the DM bus only, PM() and DM() references are semantically identical in this instruction. The address of the access selects the memory bank, so this instruction can access 24-bit memory. If so, the specified data register receives bits 23:8 from memory. Since PM() reference do not activate the PM bus, the PX register is not filled with any data.

This multifunction instruction requires the DSP to fetch two items from memory: the instruction and one data word. The number of cycles required to execute this instruction depends on whether it generates a bus conflict:

Execution	Conditions
1 cycle	If no bus conflict occurs.
2 cycles	If an instruction vs. data conflict occurs on the bus

Examples

```
AR = AX0 - AY1 + C - 1,
AX0 = DM(I1 += M0);             /* ALU operation and mem read */

MR = MX1 * MY0 (SS),
SR1 = PM(I4 += M4);             /* MAC operation and mem read */

AR = 3; SE = AR;                /* shift code, lshift 3 bits */
```

```
SI = 0xB6A3;                    /* value of hi word of input */
SR = ASHIFT SI (HI),
SI = DM(I0 += M0);              /* ashift hi word and mem read */

AR = 3; SE = AR;                /* shift code lshift 3 bits */
SI = 0x765D;                    /* value of lo word of input */
SR = SR OR LSHIFT SI (LO),
SI = DM(I0 += M0);              /* lshift lo word and mem read */
```

See Also

- "Type 4: Compute | Dreg «···» DM" on page 8-19
- "Type 12: Shift | Dreg «···» DM" on page 8-31
- "Multifunction Instructions" on page 5-1
- "MAC Instructions" on page 3-1
- "Shifter Instructions" on page 4-1
- "Arithmetic Status (ASTAT) Register" on page 1-3

Compute with Memory Write

```
DM ( | I0 | += | M0 | )    = DREG ,   | <ALU>    | ;
     | I1 |    | M1 |                 | <MAC>    |
     | I2 |    | M2 |                 | <SHIFT>  |
     | I3 |    | M3 |                 |          |
     | I4 |    | M4 |                 |          |
     | I5 |    | M5 |                 |          |
     | I6 |    | M6 |                 |          |
     | I7 |    | M7 |                 |          |
```

Function

Combines an ALU, MAC, or shifter operation with a 16-bit write to memory over the DM bus. The restricted register forms—*using* XOP *and* YOP *registers, not the* DREG *register file*—of all ALU, MAC, and shifter instructions are supported, except for the MAC saturate instruction (SAT), the divide primitives (DIVS and DIVQ), and shift immediate. Also, the multifunction ALU, MAC, and shifter instructions may not use conditional (IF) options.

The write operation executes first, transferring the current contents of the data register to the specified memory location. The compute operation executes next, overwriting the contents of the destination data register with the result.

The source of data for the memory write operation is an ALU, MAC, or shifter result or feedback register. The data is always right-justified in the destination memory location.

Input

Valid input operands for the compute depend on the operation's computation unit. For more information, see the input descriptions in "Multifunction Instructions" on page 5-1, "Shifter Instructions" on page 4-1, and "MAC Instructions" on page 3-1.

The data move operation uses two DAG registers, index (Ireg) and modify (Mreg), to generate memory addresses. Regardless of the DAG registers used, all accesses occur over the DM bus. For details on DAG registers and data addressing, see "Data Move Instructions" on page 6-1.

DAG1	I0, I1, I2, or I3 (index registers) M0, M1, M2, or M3 (modify registers)
DAG2	I4, I5, I6, or I7 (index registers) M4, M5, M6, or M7 (modify registers)

You can use any index register with any modify register from the same DAG. You cannot pair a DAG1 register with a DAG2 register.

Output

The destination register for the compute operation is always the computation unit's result or feedback register.

AR/AF	ALU operations
MR/SR	MAC operations
SR/SE	Shifter operations

The source register for the data move operation is any register file data register. Use any of these data registers for the DREG source:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

Status Flags

The status flags generated as a result of the computation depend on the compute operation the instruction performs. For more information, see the status flags section of the computation's reference page.

Details

The memory write operation uses indirect addressing with postmodify (`Ireg += Mreg`) and always transfers 16-bit data over the DM bus. For linear indirect addressing, initialize the `Lreg` register of the corresponding `Ireg` register to `0`. For circular indirect addressing, set the buffer's length and base address with the corresponding `Lreg` and `Breg` registers. For more information on addressing, see the *ADSP-219x/2192 DSP Hardware Reference*.

Since transfers occur over the DM bus only, `PM()` and `DM()` references are semantically identical in this instruction. The address of the access selects the memory bank, so this instruction can access 24-bit memory. If so, the operation writes bits `15:0` from the specified data register to bits `23:8` of the specified memory location. Since `PM()` reference do not activate the PM bus, the `PX` register does not supply any data.

This multifunction instruction requires the DSP to fetch one item from memory and write one item to memory: the instruction and one data word. The number of cycles required to execute the instruction depends on whether it generates a bus conflict:

Execution	Conditions
1 cycle	If no bus conflict occurs.
2 cycles	If an instruction vs. data conflict occurs on the bus

Except for `SR2`, you can use the same data register in both the compute and memory write operations—as the result register for the computation and as the source register for the data move operation.

Examples

```
DM(I1 += M0) = AX0,
AR = AX0 - AY1 + C - 1;          /* mem write and ALU operation */

PM(I4 += M4) = SR1,
MR = MX1 * MY0 (SS);             /* mem write and MAC operation */

AR = 3; SE = AR;                 /* shift code, lshift 3 bits */
SI = 0xB6A3;                     /* value of hi word of input */
DM(I0 += M0) = SI,
SR = ASHIFT SI (HI);             /* mem write and ashift hi word */

AR = 3; SE = AR;                 /* shift code lshift 3 bits */
SI = 0x765D;                     /* value of lo word of input */
DM(I0 += M0) = SI,
SR = SR OR LSHIFT SI (LO);       /* mem write and lshift lo word */
```

See Also

- “Type 4: Compute | Dreg «···» DM” on page 8-19
- “Type 12: Shift | Dreg «···» DM” on page 8-31
- “Multifunction Instructions” on page 5-1
- “MAC Instructions” on page 3-1
- “Shifter Instructions” on page 4-1
- “Arithmetic Status (ASTAT) Register” on page 1-3

Compute with Register-to-Register Move

```
<ALU>         , DREG1 = DREG2 ;
<MAC>
<SHIFT>
```

Function

Combines an ALU, MAC, or shifter operation with a register-to-register move. The restricted register forms—*using* XOP *and* YOP *registers, not the* DREG *register file*—of all ALU, MAC, and shifter instructions are supported, except for the MAC saturate instruction (SAT), the divide primitives (DIVS and DIVQ), and shift immediate. Also, the multifunction ALU, MAC, and shifter instructions may not use conditional (IF) options.

The compute operation executes first, using the current contents of the data register. The data move executes next, overwriting the contents of the destination data register with the contents of the source register.

The source and destination of the data move operation is an ALU, MAC, or shifter data register. The transferred data is always right-justified in the destination data register.

Input

Valid input operands for the compute depend on the operation's computation unit. For more information, see the input descriptions in "Multifunction Instructions" on page 5-1, "MAC Instructions" on page 3-1, and "Shifter Instructions" on page 4-1.

Output

The result register for the compute operation is always the computation unit's result or feedback register.

AR/AF	ALU operations
MR/SR	MAC operations
SR/SE	Shifter operations

The source (DREG2) and destination (DREG1) registers for the data move operation are any register file data registers. Use any of these data registers for the DREG source and destination:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

Status Flags

The status flags generated as a result of the computation depend on the compute operation the instruction performs. For more information, see the status flags section of the computation's reference page.

Details

Except for SR2, you can use the same data register in both the compute and data move operations—as the result register for the computation and as the source register for the data move operation.

If you use AR as the source and destination in the data move operation (AR = AR), the compute operation generates status only—no computation results. For more information, see "Generate ALU Status Only: NONE" on page 2-46.

Examples

```
AR = AX1 + AY1,MX0 = AR;          /* add and reg.-to-reg. move */
MR = MX1 * MY0 (US), MY0 = AR;    /* mult and reg-to-reg move */
AR = 3; SE = AR;                  /* shift code, lshift 3 bits */
SI = 0xB6A3;                      /* value of hi word of input */
SR = ASHIFT SI (HI),
SI = MR0;                         /* ashift hi word reg move */
```

See Also

- "Type 8: Compute | Dreg1 «··· Dreg2" on page 8-22
- "Type 14: Shift | Dreg1 «··· Dreg2" on page 8-32
- "Multifunction Instructions" on page 5-1
- "Shifter Instructions" on page 4-1
- "MAC Instructions" on page 3-1
- "Arithmetic Status (ASTAT) Register" on page 1-3

6 DATA MOVE INSTRUCTIONS

The instruction set provides move instructions for transferring data between the DSP's data registers, memory, I/O registers, and system control registers. Transfer operations include reading, writing, loading, and storing data from one location to another.

This chapter describes each of the move instructions ("Data Move Instruction Reference" on page 6-21) and the following related topics:

- "Core Registers" on page 6-2
- "PX Register" on page 6-3
- "DAG Registers" on page 6-5
- "Register Load Latencies" on page 6-9
- "Direct Addressing" on page 6-11
- "Indirect Addressing" on page 6-12
- "Circular Data Buffer Addressing" on page 6-14
- "Bit-Reversed Addressing" on page 6-16

Core Registers

Table 6-1 lists the registers that reside in the DSP's core. Most are 16-bit registers, but some Reg3 registers are shorter—ASTAT[9], MSTAT[7], SSTAT[8], LPCSTACKP[9], CCODE[4], PX[8], DMPG1[8], DMPG2[8], IOPG[8], IJPG[8], and STACKP[8].

Table 6-1. Core Registers

Register Groups			
Reg0 (Dreg)	**Reg1 (G1reg)**	**Reg2 (G2reg)**	**Reg3 (G3reg)**
AX0	I0	I4	ASTAT
AX1	I1	I5	MSTAT
MX0	I2	I6	SSTAT
MX1	I3	I7	LPSTACKP
AY0	M0	M4	CCODE
AY1	M1	M5	SE
MY0	M2	M6	SB
MY1	M3	M7	PX
MR2	L0	L4	DMPG1
SR2	L1	L5	DMPG2
AR	L2	L6	IOPG
SI	L3	L7	IJPG
MR1	IMASK	Reserved	Reserved
SR1	IRPTL	Reserved	Reserved
MR0	ICNTL	CNTR	Reserved
SR0	STACKA	LPSTACKA	STACKP

As shown, registers are grouped along functional lines:

- `Reg0`(`Dreg`) – Consists of data registers.
- `Reg1`(`G1reg`) – Consists of DAG1 addressing registers, interrupt control registers, and the lower part of the PC stack register.
- `Reg2`(`G2reg`) – Consists of DAG2 addressing registers, the loop counter register, and the lower part of the loop PC register.
- `Reg3`(`G3reg`) – Consists of status registers, page registers.

PX Register

The `PX` register, an 8-bit extension register, enables applications to transfer 24-bit data between 24-bit memory and 16-bit data registers. Only 24-bit accesses of 24-bit memory use the `PX` register. (So, a 16-bit read of 24-bit memory does not load the `PX` register, and a 16-bit write fills the lower eight bits in 24-bit memory with zeros (`0`).)

On reads, the `PX` register stores the lower eight bits of the 24-bit data transferring from memory to a destination register. On writes, it supplies them for the data written to 24-bit data space.

Only two instructions use the `PX` register:

- ALU/MAC with dual indirect memory reads (see page "Compute with Dual Memory Read" on page 5-4)
- Indirect 24-bit memory read or write with the premodify addressing option described on page 6-38 or the postmodify addressing option described on page 6-42.

To access 24-bit memory, you typically use the `PM(Ireg += Mreg)` syntax shown here:

```
AX1 = PM(I0 += M2);      /* Read 24 bits, load 16 MSbits in AX1 */
                         /* PX autoloaded w/8 memory LSbits */
AY1 = PX;                /* Load lower 8 bits from PX in AY1 */

PX = MR2;                /* Load lower 8 bits into PX */
PM(I4 += M5) = MR1;      /* Write all 24 bits from MR1 and PX */
```

On data reads using the `PX` register, the DSP transfers the upper 16 bits of the 24-bit data to the destination data register and the lower eight bits to the `PX` register. The data loaded from memory is right-justified in the destination registers.

On data writes using the `PX` register, the DSP transfers the upper 16 bits of the 24-bit data from the bus and the lower eight bits from the `PX` register, except for indirect writes of 24-bit immediate data, in which the instruction supplies the eight LSBs. The data written is right-justified in memory.

`PX` transfers to and from memory occur automatically and transparently to the user, but the user must transfer data between the `PX` register and the data registers.

Because the DSP has a unified memory space, the address, not the syntax, determines whether the reference accesses 16-bit memory or 24-bit memory at run time.

- For 24-bit references that read 16-bit memory, the `PX` register receives whatever data the memory system outputs for the eight LSBs. For internal memory, this value is `0x00`.
- For 24-bit references that write 16-bit memory, the DSP discards the data in the `PX` register.

DAG Registers

DAGs generate memory addresses for data transfers to and from memory. To do so, each DAG uses a set of address registers and a page register. For fast context switching during interrupt servicing, the DAGs provide a secondary set of address registers. This section describes these registers.

Address Registers

Each DAG has a set of address registers that it uses to generate memory addresses for loading or storing data in memory. Each DAG can use its own set of address registers only. DAG1 uses registers `0` through `3`, and DAG2 uses registers 4 through 7.

The DAG address registers are:

- Index (`Ireg`). Pointer to the current memory address. DAG1 (`I0-I3`); DAG2 (`I4-I7`).
- Modify (`Mreg`). Offset (from index) value for pre- or post-modify addressing. DAG1 (`M0-M3`); DAG2 (`M4-M7`).
- Length (`Lreg`). Number of memory locations in a buffer. DAG1 (`L0-L3`); DAG2 (`L4-L7`). For linear buffers, you must explicitly set `Lreg = 0`; for circular buffers, you must explicitly set `Lreg` to the length of the buffer.
- Base (`Breg`). Starting address of a circular buffer. DAG1 (`B0-B3`); DAG2 (`B4-B7`).Used with circular buffering only.

Each base (`Breg`) and length (`Lreg`) register is associated with its specific index (`Ireg`) register—`I0/B0/L0`, `I1/B1/L1`, …, and `I7/B7/L7`. Although you can mix and match any of the index (`Ireg`) and modify (`Mreg`) registers within the same DAG group, you must always use the base (`Breg`) and length (`Lreg`) register that is associated with the particular index (`Ireg`) register you use.

DAG Memory Page Registers (DMPGx)

The DAGs and their associated page registers generate 24-bit addresses for accessing the data needed by instructions. For data accesses, the DSP's unified memory space is organized into 256 pages, with 64K locations per page. Page registers provide the eight MSBs of the 24-bit address, specifying the page on which the data is located. DAGs provide the 16 LSBs of the 24-bit address, specifying the exact location of the data on the page.

- `DMPG1` is associated with DAG1 (registers `I0–I3`) indirect memory accesses as well as immediate, direct memory accesses. It supplies the upper 8 MSBs for direct memory addressed instructions.
- `DMPG2` is associated with DAG2 (registers `I4–I7`) indirect memory accesses.

At power up, the DSP initializes both page registers to `0x0`. Although initializing page registers is unneccessary unless the data is located on other than the current page, good programming practice recommends that you set the corresponding page register whenever you initialize a DAG index register (`Ireg`) to set up a data buffer.

For example,

```
DMPG1 = 0x12;          /* set page register */
    /* or DMPG1 = page(data_buffer); for relative addressing */
I2 = 0x3456;           /* init data buffer; 24b addr=0x123456 */
L2 = 0;                /* define linear buffer */
M2 = 1;                /* increment address by one */
                       /* two stall cycles inserted here */
DM(I2 += M2) = AX0;    /* write data to buffer and update I2 */
```

DAG register `DMPGx`, `(Ireg`, `Mreg`, `Lreg`, `Breg`) loads can incur up to two stall cycles when a memory access based on the initialized register immediately follows the initialization.

To avoid these unproductive stall cycles, you can code the memory access sequence like this:

```
DMPG1 = 0x12;           /* set page register */
    /* or DMPG1 = page(data_buffer); for relative addressing */
I2 = 0x3456;            /* init data buffer; 24b addr=0x123456 */
L2 = 0;                 /* define linear buffer */
M2 = 1;                 /* increment address by one */
AX0 = 0xAAAA;
AR = AX0 - 1;
DM(I2 += M2) = AR;      /* write data to buffer and update I2 */
```

Typically, you load both page registers with the same page value (`0-255`), but you can increase memory flexibility by loading each with a different page value. For example, loading the page registers with different page values allows you to:

- Separate DMA space from the application's data space
- Perform high-speed data transfers between pages

This operation is not automatic and requires explicit programming.

Secondary DAG Registers

The secondary set of DAG address registers (`Ireg`, `Mreg`, `Lreg`, and `Breg`) enable single-cycle context-switching to support real-time control functions and to reduce overhead associated with interrupt servicing.

By default, system power-up and reset enable the primary set of DAG address registers. To enable or disable the secondary address registers, set or clear, respectively, the `SEC_DAG` bit (bit 6) in `MSTAT` (for details, see "Mode Status (MSTAT) Register" on page 1-8). The instruction set provides three methods for doing so. Each method incurs a latency, a

delay between the time the instruction effecting the change executes and the time the change takes effect and is available to other instructions. Table 6-2 on page 6-9 shows the latencies associated with each method.

When switching between primary and secondary DAG registers, applications need to account for the latency associated with the method they use. For example, after the `MSTAT = data12;` instruction, three cycles of latency occur before the mode change takes effect. So, you must issue at least three instructions after `MSTAT = 0x20;` before attempting to use the new set of DAG registers. Otherwise, you will overwrite the primary set and lose data.

The `ENA/DIS` mode instruction is more efficient for enabling and disabling DSP modes since it incurs no cycles of effect latency. For example:

```
CCODE = 0x9; NOP;
IF SWCOND JUMP do_data;       /* Jump to do_data */
do_data:
   ENA SEC_DAG;               /* Switch to 2nd DAGs */
   ENA SEC_REG;               /* Switch to 2nd Dregs */
   AX0 = DM(buffer);          /* if buffer empty, go */
   AR = PASS AX0;             /* right to fill and */
   IF NE JUMP fill;           /* get new data */
   RTI;
fill:                         /* fill routine */
   NOP;
buffer:                       /* buffer data */
   NOP;
```

Register Load Latencies

An effect latency occurs when instructions write or load a value into a register, which changes the value of one or more bits in the register. Effect latency refers to the time it takes after the write or load instruction for the effect of the new value to become available for other instructions to use.

Effect latency values are given in terms of instruction cycles. A latency of zero means that the effect of the new value is available on the next instruction following the write or load instruction. For register changes that have an effect latency greater than 0, do not try to use the register immediately after a write or a new value is loaded. Table 6-2 gives the effect latencies for writes or loads of various interrupt and status registers.

Table 6-2. Effect Latencies for Register Changes

Register	Bits	REG = value	ENA/DIS mode	POP STS	SET/CLR INT
ASTAT	All	1 cycle	NA	0 cycles	NA
CCODE	All	1 cycle	NA	NA	NA
CNTR	All	1 cycle[1]	NA	NA	NA
ICNTL	All	1 cycle	NA	NA	0 cycles
IMASK	All	1 cycle	NA	0 cycles	NA
MSTAT	SEC_REG	1 cycle	0 cycles	1 cycle	NA
	BIT_REV	3 cycles	0 cycles	3 cycles	NA
	AV_LATCH	0 cycles	0 cycles	0 cycles	NA
	AR_SAT	1 cycle	0 cycles	1 cycle	NA
	M_MODE	1 cycle	0 cycles	1 cycle	NA
	TIMER	1 cycle	0 cycles	1 cycle	NA
	SEC_DAG	3 cycles	0 cycles	3 cycles	NA

1 This latency applies only to IF COND instructions, not to the DO UNTIL instruction. Loading the CNTR register has 0 effect latency for the DO UNTIL instruction.

A PUSH or POP PC has one cycle of latency for all SSTAT register bits, but a PUSH or POP LOOP or STS has one cycle of latency only for the STKOVERFLOW bit in the SSTAT register.

When you load some Group 2 and 3 registers (see Table 6-1 on page 6-2), the effect of the new value is not immediately available to subsequent instructions that might use it. For interlocked registers (DAG address and page registers, IOPG, IJPG), the DSP automatically inserts stall cycles as needed. For noninterlocked registers, to accommodate the required latency, insert either the necessary number of NOP instructions or other instructions that are not dependent upon the effect of the new value.

The noninterlocked registers are:

- Status registers (ASTAT and MSTAT)
- Condition code register (CCODE)
- Interrupt control register (ICNTL)

The number of NOP instructions you must insert is specific to the register and the load instruction as shown in Table 6-2. A zero (0) latency indicates that the new value is effective on the next cycle after the load instruction executes. An *n* latency indicates that the effect of the new value is available up to *n* cycles after the load instruction executes. When using a modified register before the required latency, you may get the register's old value.

Since unscheduled or unexpected events (interrupts, DMA operations, and so on) often interrupt normal program flow, do not rely on these load latencies when structuring your program's flow. A delay in executing a subsequent instruction based on a newly loaded register may result in erroneous results—whether the subsequent instruction is based on the effect of the register's new or old value.

Load latency applies only to the time it takes the loaded value to effect the change in operation, not to the number of cycles required to load the new value. A loaded value is always available to a read access on the next instruction cycle.

Data Addressing Methods

The instruction set supports two addressing methods for accessing memory data:

- Direct addressing. The user supplies an explicit address in the instruction.
- Indirect addressing. The DAG address registers generate addresses.

Direct Addressing

Direct addressing is the simplest method to use. An explicit address or a label included in the instruction specifies the address of a memory access. A label is a symbolic name that you assign to an address.

Specify an explicit address or label in a data move instruction like this:

```
DM(I1 += M0) = 0x1234; /* write data 0x1234 and post-modify */
AX0 = DM(0x3333);      /* read location 0x3333, put in AX0 */
DM(port1) = AY1;       /* write value in AY1 to port1 */
port1:                 /* port1 address is in linker ldf */
NOP;
```

When using a label, specify the address that the label references or let the linker assign the label and address. For details on assigning label addresses, see the *VisualDSP++ Linker and Utilities Manual for 16-Bit DSPs*.

Indirect Addressing

Indirect addressing uses a pointer to specify the address of a memory access. The Index (`Ireg`) and Modify (`Mreg`) registers implement address pointers for indirect addressing. The `Ireg` supplies the address value, and the `Mreg` supplies the modify (offset) value, which, when added to the

address value, forms the address of the next memory location. The instruction set provides two address modification options—premodify with no update and postmodify with update.

- Premodify addressing—no update

  Premodify addressing does not permanently change the value of the Index register (`Ireg`). In premodify operations, the sum of the `Ireg` and `Mreg` register values provides the address of the memory access. After the access, the `Ireg` register retains its original value.

  For example, the following sets up a DAG1 linear data buffer using the premodify option:

```
#define buffer1 0x2
DMPG1 = page(buffer1);
I0 = buffer1;
M0 = 0x0007;
L0 = 0;              /* Unless L = 0 buffer is circular */
AX0 = DM(I0 + M0);   /* AX0 receives data @ I0+M0 */
                     /* I0 retains original value */
```

  For example, this sets up a DAG2 linear data buffer premodified with a constant:

```
#define buffer2 0x3
DMPG2 = page(buffer2);
I4 = buffer2;
L4 = 0;               /* Unless L=0, buffer is circular */
AX0 = DM(I4 + 0x0007); /* AX0 receives data @ I4+0x0007 */
                      /* I4 retains original value */
```

- Postmodify addressing—with update

Postmodify addressing permanently changes the value in the Index (`Ireg`) register. In postmodify operations, the current value in `Ireg` is used for the memory access. After the access, the DSP adds the modify value in `Mreg` to the address value in `Ireg` and overwrites the contents in `Ireg` with the result.

For example, the following sets up a DAG1 linear data buffer using the postmodify option:

```
#define buffer3 0x2
DMPG1 = page(buffer3);
I0 = buffer3;
M0 = 0x0007;
L0 = 0;               /* Unless L = 0 buffer is circular */
AX0 = DM(I0 += M0);   /* AX0 receives data @ I0+M0 */
                      /* updated with sum of (I0+M0) */
```

For example, this sets up a a DAG1 linear data buffer postmodified with a constant:

```
#define buffer4 0x3
DMPG1 = page(buffer4);
I0 = buffer4;
L0 = 0;                   /* Unless L=0, buffer is circular*/
AX0 = DM(I0 += 0x0003);  /* AX0 receives data @ I0+0x0003 */
                          /* I0 updated w/sum (I0+0x0003) */
```

Circular buffers work with postmodify addressing only.

To set up data buffers, you can mix and match any of the Index (`Ireg`) and modify (`Mreg`) registers within the same DAG group (DAG1 or DAG2), but not between DAG groups.

Length (Lreg) and base address (Breg) registers, when used, must always match their corresponding Ireg. For example, the following code is valid, because it uses corresponding Ireg and Lreg registers:

```
DMPG1 = page(data_in);
I3 = data_in;
M1 = 0x0007;
L3 = 0;            /* Unless Lreg = 0 buffer is circular */
AX0 = DM(I3 += M1);
data_in:           /* data_in location could elsewhere */

NOP;
```

Circular Data Buffer Addressing

Circular data buffers enable applications to reuse the same data buffer; for example, to store the filter coefficients for a FIR or IIR filter or to act as a delay line for the convolution of an input signal.

A circular data buffer is a set of memory locations used for storing a set of data. A circular data buffer is defined by a set of DAG address registers:

- Base (B0-B7) Starting address.
  These registers are off core, so you must use the data (Dreg) registers and this syntax to access them: REG(Breg) = Dreg;
- Index (I0-I7) Current address.
  The Ireg points to the current address within the buffer. After the access, the Ireg is postmodified with the address of next access.
- Modify (M0-M7) Number of locations offset from the current address. To calculate the address of the next access, the offset value in Mreg is added to the current Ireg value, and the result is written to Ireg.
- Length (L0-L7) Number of memory locations in the buffer.

An index pointer (Ireg) steps through the data buffer, forwards or backwards, in programmable increments as determined by a modifier (Mreg) value. The base address (Breg) and the buffer's length (Lreg) keep the pointer within the range of the buffer's memory locations. When the index pointer steps outside the buffer's address range, the logic adds or subtracts the buffer's length from the index value to wrap the pointer back to the top or bottom of the buffer, as appropriate.

For example, the following code sets up a circular data buffer:

```
.section/dm seg_data;
.VAR coeff_buffer[13] = 0,1,2,3,4,5,6,7,8,9,10,11,12;
.section/pm seg_code;
DMPG2 = page(coeff_buffer);      /* Set the memory page */
I4 = coeff_buffer;               /* Set the current addr */
M5 = 5;                          /* Set the modify value */
L4 = LENGTH(coeff_buffer);       /* If L=0, buffer is linear */
AX0 = I4;                        /* Copy base addr into AX0 */
REG(B4) = AX0;                   /* Set buffer's base addr */
AR = AX1 AND AY0;
AR = DM(I4 += M5);               /* Read 1st buffer location */
```

Figure 6-1, using this code example, demonstrates how the index pointer steps through the circular buffer.

Circular data buffers work with the postmodify addressing option only. You must initialize the length (Lreg) register to the length of the buffer. Positive modify values increment the index register, and negative modify values decrement it.

Do not place the index pointer for a circular buffer such that it crosses a memory page boundary during post-modify addressing. All memory locations in a circular buffer must reside on the same memory page.

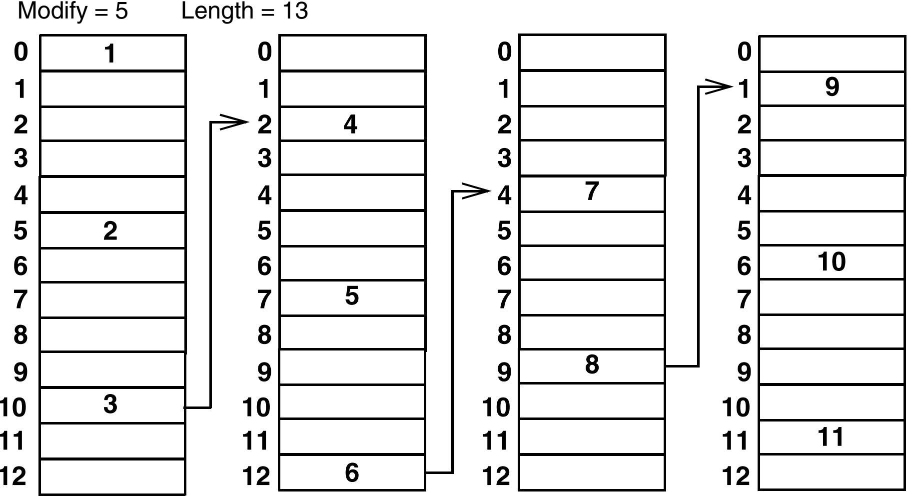


Figure 6-1. Stepping Through a Circular Data Buffer

Bit-Reversed Addressing

Bit-reversed addressing is frequently used in FFT calculations to obtain results in sequential order. Because FFT operations repeatedly subdivide data sequences, the data or twiddle factors may be scrambled, loaded or stored in bit-reversed order.

For performing FFT operations, you can reverse the order in which DAG1 outputs its address bits. DAG2 always outputs its address bits in normal, big endian format. Since the two DAGs operate independently, you can use them in tandem, with one generating sequentially ordered addresses and the other generating bit-reversed addresses, to perform memory reads and writes of the same FFT data.

To use bit-reversed addressing, set bit 1 in the `MSTAT` register (`ENA BIT_REV`). When enabled, DAG1 outputs all addresses generated by its Index registers (`I0–I3`) in bit-reversed order. The reversal applies to the address value DAG1 outputs only, not to the address value stored in the

Index (Ireg) register, so the Ireg value is stored in big endian format. Bit-reversed mode remains in effect until you clear bit 1 in the MSTAT register (DIS BIT_REV).

Bit reversal operates on the binary number that represents the position of a sample within an array of samples. Using 3-bit addresses, Table 6-3 shows the position of each sample within an array before and after the bit-reverse operation. For example, sample x4 occupies position 0b100 in sequential order, but occupies position 0b001 in bit-reversed order. Bit reversing transposes the bits of a binary number about its midpoint, so 0b001 becomes 0b100, 0b011 becomes 0b110, and so on. Some numbers, like 0b000, 0b111, and 0b101, remain unchanged and retain their original position within the array.

Table 6-3. Eight-Point Array Sequence Before and After Bit Reversal

Sequential Order		Bit-Reversed Order	
Sample	**Binary**	**Binary**	**Sample**
x0	000	000	x0
x1	001	100	x4
x2	010	010	x2
x3	011	110	x6
x4	100	001	x1
x5	101	101	x5
x6	110	011	x3
x7	111	111	x7

Bit-reversing the samples in a sequentially ordered array scrambles their positions within the array. Bit-reversing the samples in a scrambled array restores their sequential order within the array.

In full 16-bit reversed addressing, bits 7 and 8 of the 16-bit address are the pivot points for the reversal:

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

FFT operations often need only a few address bits reversed; for example, a a 16-point sequence requires four reversed bits, and a 1024-bit sequence requires ten reversed bits. You can bit-reverse address values less than 16-bits—which reverses a specified number of LSBs only. Bit-reversing less than the full 16-bit Index register value requires that you add the correct modify value to the index pointer after each memory access to generate the correct bit-reversed addresses.

To set up bit-reversed addressing for address values less than bits, you need to determine:

- The number of bits to reverse (`N`)

  Use this value to calculate the modify value.

- The starting address of the linear data buffer

  The starting address of an array that the program accesses with bit-reversed addressing must be zero or an integer multiple of the number of bits to reverse (starting address = `0, N, 2N, …`).

- The first bit-reversed address that the DAG will output

  This value is the buffer's starting address, but with the `N` LSBs bit-reversed.

- The initialization value for the index (Ireg)

  Initialize the Index (Ireg) register with the bit-reversed value of the first bit-reversed address the DAG will output.

- The correct modify value (`Mreg`) with which to update the index pointer after each memory access

  Use this formula to calculate the modify value:
  `Mreg = 2`$^{(16-N)}$.

As an example, the following code sets up bit-reversed addressing that reverses the eight address LSBs (`N = 8`) of a data buffer with a starting address of `0x0020` (`4N`).

We need to determine the:

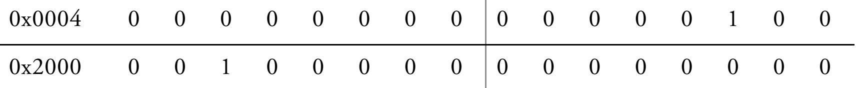
- First bit-reversed address that DAG1 will output

  This value is the buffer's starting address (`0x0020`) with bits[7:0] reversed: `0x0004`.

0x0020	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0x0004	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

- Initialization value for the index (Ireg) register

  This is first bit-reversed address DAG1 will output (`0x0004`) with bits[15:0] reversed: `0x2000`.



0x0004	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0x2000	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

  Correct modify value for `Mreg`

  This is `2`$^{(16-N)}$ which evaluates to 2^8 or `0x0100`.

Listing 6-1 shows the code for this example.

Listing 6-1. Bit-Reversed Addressing, 8 LSBs

```
br_adds:
   I4 = read_in;            /* DAG2 pointer to input samples */
   I0 = 0x0200;             /* Base address of bit_rev output */
   M4 = 1;                  /* DAG2 increment by 1 */
   M0 = 0x0100;             /* DAG1 increment for 8-bit rev. */
   L4 = 0;                  /* Linear data buffer */
   L0 = 0;                  /* Linear data buffer */
   CNTR = 8;                /* 8 samples */
   ENA BIT_REV;             /* Enable DAG1 bit reverse mode */
   DO brev UNTIL CE;
      AY1 = DM(I4+=M4);     /* Read samples sequentially */
      brev: DM(I0+=M0) = AY1; /* Write results nonsequentially */
   DIS BIT_REV;             /* Disable DAG1 bit reverse mode */
   RTS;                     /* Return to calling routine */
read_in:                    /* input buffer, could be .extern */
   NOP;
```

Data Move Instruction Reference

Data move operations include:

- "Register-to-Register Move" on page 6-22
- "Direct Memory Read/Write—Immediate Address" on page 6-24
- "Direct Register Load" on page 6-27
- "Indirect 16-Bit Memory Read/Write—Postmodify" on page 6-30
- "Indirect 16-Bit Memory Read/Write—Premodify" on page 6-34
- "Indirect 24-Bit Memory Read/Write—Postmodify" on page 6-38
- "Indirect 24-Bit Memory Read/Write—Premodify" on page 6-42
- "Indirect DAG Register Write (Premodify or Postmodify), with DAG Register Move" on page 6-46
- "Indirect Memory Read/Write—Immediate Postmodify" on page 6-50
- "Indirect Memory Read/Write—Immediate Premodify" on page 6-53
- "Indirect 16-Bit Memory Write—Immediate Data" on page 6-56
- "Indirect 24-Bit Memory Write—Immediate Data" on page 6-58
- "External I/O Port Read/Write" on page 6-61
- "System Control Register Read/Write" on page 6-64
- "Modify Address Register—Indirect" on page 6-67
- "Modify Address Register—Direct" on page 6-69

Register-to-Register Move

Dreg1	=	Dreg2	;
G1reg1		G1reg2	
G2reg1		G2reg2	
G3reg1		G3reg2	

Function

Moves the contents in the source register to the destination register. The contents in the source register are right-justified in the destination register.

Source

REG2 can be any core register listed in Table 6-1 on page 6-2.

Destination

REG1 can be any core register listed in Table 6-1 on page 6-2.

Details

For transfers in which the destination register is MR1 or SR2, this operation sign-extends into MR2 or SR2, respectively, the most significant bit of the value stored in MR1 or SR1.

For transfers in which the destination register is smaller than the source register, this operation right-justifies the value in the destination register, such that bit 0 maps to bit 0, and truncates the extraneous high-order bits.

For transfers in which the source register is smaller than the destination register, the value is sign-extended.

When loading the CCODE, ASTAT, or MSTAT register, the effect of the new value is not available immediately to subsequent instructions that are based on it. You must insert the required number of NOP instructions before using the modified register, or your instruction may execute based the old value. For more information, see "Register Load Latencies" on page 6-9).

SSTAT is a read-only register, so SSTAT = reg; is invalid instruction syntax.

Examples

```
I0 = I4;                    /* load I0 from I4 */
CCODE = AY0;                /* load CCODE from AY0 */
DMPG1 = DMPG2;              /* load DMPG1 from DMPG2 */
```

See Also

- "Type 17: Any Reg «···Any Reg" on page 8-35
- "Core Registers" on page 6-2
- "PX Register" on page 6-3
- "DAG Registers" on page 6-5
- "Register Load Latencies" on page 6-9
- "Direct Register Load" on page 6-27

Direct Memory Read/Write—Immediate Address

```
| Dreg |  =    DM(<Imm16>)   ;
| Ireg |
| Mreg |

DM(<Imm16>)   =   | Dreg | ;
                  | Ireg |
                  | Mreg |
```

Function

The memory read operation moves the contents of the memory location specified by an immediate 16-bit value or label into the destination register.

The memory write operation moves the contents of the source register into the memory location specified by an immediate 16-bit value or label.

Source

Reads — The data comes from the memory location specified by an immediate 16-bit value or label.

Writes — The data comes from any register file data (`Dreg`) register or Index (`Ireg`) or Modify (`Mreg`) register:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

DAG1/DAG2 Index and Modify Registers
I0, I1, I2, I3, I3, I4, I6, I7, M0, M1, M2, M3, M4, M5, M6, M7

Destination

Reads	The data goes to any register file data (`Dreg`) register or Index (`Ireg`) or Modify (`Mreg`) register.
Writes	The data goes to the memory location specified by an immediate 16-bit value or label.

Details

This instruction is typically used by memory-intensive applications that must make highly efficient use of memory. Applications that use absolute memory locations need to configure and use them with care. For information on using absolute memory locations, see the *VisualDSP++ Linker and Utilities Manual for 16-Bit DSPs* and the *VisualDSP++ Assembler Manual for 16-Bit DSPs*.

This instruction transfers 16-bit data only over the DM bus. It does not write or read from the `PX` register.

When you load 16-bit data into `MR1` or `SR1`, it is sign-extended into `MR2` or `SR2`, respectively.

`DMPG1` provides the eight MSBs of the address. For details, see "DAG Memory Page Registers (DMPGx)" on page 6-6.

When loading a DAG address or page register, the new value is not available immediately to subsequent instructions that use the register for a memory access. The DAG address registers have a two-cycle latency.

Because DAG registers are interlocked, the DSP automatically inserts up to two stall cycles, as needed, to ensure that subsequent instructions use the new address value.

For efficient programming, insert two instructions that do not use the modified register in the two instruction lines immediately following the load instruction. For example, separate the `DMPG1 load` and the memory access with two other DAG register loads:

```
I0 = buffer;           /* Ireg load, data_in defined */
NOP;                   /* any two non-DAG1 instructions */
NOP;                   /* can execute here without latencies */
AX0 = DM(I0 + 0);      /* memory access */
```

Examples

```
SI = DM(data_in);      /* Dreg load, label defined */
I4 = DM(coeff_buffer);
                       /* Ireg load, label defined */
M5 = DM(coeff_buffer);
                       /* Mreg load, label defined */
DM(coeff_buffer) = AX1;
                       /* Dreg load, label defined */
DM(data_in) = I0;      /* Ireg load, label defined */
DM(data_in) = M1;      /* Mreg load, label defined */
```

See Also

- "Type 3: Dreg/Ireg/Mreg «···» DM/PM" on page 8-18
- "Direct Addressing" on page 6-11
- "Core Registers" on page 6-2
- "DAG Registers" on page 6-5
- "Secondary DAG Registers" on page 6-7
- "Register Load Latencies" on page 6-9

Direct Register Load

```
|Dreg        | = <Data16> ;
|G1reg       |
|G2reg       |

 G3reg         = <Data12> ;
```

Function

Loads the destination register with immediate data supplied in the instruction. The data is right-justified in the destination register.

Use the `<data16>` instruction to load a value into the data registers, to initialize DAG address registers, to enable or disable interrupts, and to load certain stack registers.

Use the `<data12>` instruction to load `G3reg` registers that are less than 16 bits wide. Load the short registers to set or clear one or more bits in the status registers, to set up flag conditions, to set the various page registers, and to load certain stack registers. For a list of the core registers, see Table 6-1 on page 6-2.

Source

The `<data16>` instruction accepts a 16-bit immediate value, a pointer to a 16-bit variable, or a `LENGTH(16-bit variable)`.

The `<data12>` instruction accepts only an immediate value less than or equal to 12 bits.

Destination

The <data16> instruction places the data in any register group 0, 1, or 2 register:

Register Group 0 (Dreg), 1 (G1reg), & 2 (G2reg) Registers
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI, I0, I1, I2, I3, I3, I4, I6, I7, M0, M1, M2, M3, M4, M5, M6, M7, L0, L1, L2, L3, L4, L5, L6, L7, IMASK, IRPTL, ICNTL, STACKA, CNTR, LPCSTACKA, SB, SE

The <data12> instruction places the data in any register group 3 register:

Register Group 3 (G3reg) Registers (writable)
ASTAT, MSTAT, LPCSTACKP, CCODE, SE, SB, PX, DMPG1, DMPG2, IOPG, IJPG, STACKP

SSTAT is a read-only register.

Details

When you load 16-bit data into MR1 or SR1, it is sign-extended into MR2 or SR2, respectively.

When using the <data12> instruction to load a 16-bit register (SE, or SB), the destination register's MSBs are filled with zeros (0).

When loading certain registers (CCODE, ASTAT, MSTAT, IJPG, Ireg, or DMPGx), the new value is not available immediately to subsequent instructions. For information on register latencies, see "Register Load Latencies" on page 6-9.

Examples

```
 /* Loading 16-bit registers with 16-bit values: */
AR = 0x5409;      /* Dreg put data */
I2 = ;coeff_buffer
   /* Ireg put addr, label defined  */  see definition on page 6-15

M0 = 0x1234;      /* Mreg put data */
L3 = LENGTH(coeff_buffer);
/* put length, label defined */  see definition on page 6-15

/* Loading 12-bit Reg3 registers with short constants */
STACKP = 0;
MSTAT = 0x4;      /* Enable AV_latch */
```

See Also

- "Type 6: Dreg «··· Data16" on page 8-20
- "Type 7: Reg1/2 «··· Data16" on page 8-21
- "Type 33: Reg3 «··· Data12" on page 8-52
- "Core Registers" on page 6-2
- "DAG Registers" on page 6-5
- "Secondary DAG Registers" on page 6-7
- "Register Load Latencies" on page 6-9
- "System Control Register Read/Write" on page 6-64

Indirect 16-Bit Memory Read/Write—Postmodify

```
| Dreg     | = DM(Ireg += Mreg) ;
| G1reg    |
| G2reg    |
| G3reg    |
```

```
DM(Ireg += Mreg) =        | Dreg     | ;
                          | G1reg    |
                          | G2reg    |
                          | G3reg    |
```

Function

Transfers 16-bit data between memory and any of the core registers (Dreg, G1reg, G2reg, or G3reg) over the DM bus. The current value in Ireg provides the address for the memory access. After the access, Ireg is updated with the sum of its current value and the value in Mreg.

Source

Reads — The 16-bit data comes from the memory location pointed to by the address in the Ireg, which is modified after the access by the value in the Mreg:

- DM/DAG1—I0, I1, I2, or I3 (index registers) M0, M1, M2, or M3 (modify registers)
- PM/DAG2—I4, I5, I6, or I7 (index registers) M4, M5, M6, or M7 (modify registers)

You can use any index register with any modify register from the same DAG. You cannot pair a DAG1 register with a DAG2 register.

Writes	The 16-bit data comes from any core register, except `SSTAT`, which is a read-only register. For information on core registers, see Table 6-1 on page 6-2.

Destination

Reads	The 16-bit data goes to any core register. For information on core registers, see Table 6-1 on page 6-2.
Writes	The 16-bit data goes to the memory location pointed to by the address in the `Ireg`, which is modified after the access by the value in the `Mreg`.

Details

On reads and writes, the data is right-justified in the destination location (bit0 of the transfer data maps to bit0 of the destination). If the width of the destination register is less than 16 bits, the extraneous MSBs of the data are discarded. On writes from source registers less than 16 bits, the missing high-order bits are zero-filled in the memory location.

As shown in Figure 6-2, if this instruction actually references 24-bit data space at runtime, a read operation loads bits 23:8 from the memory location into bits 15:0 of a 16-bit register (or bits 23:16 into bits 7:0 of an 8-bit register, and so on). The low-order bits of the memory location are ignored. Conversely, a write operation stores bits 15:0 from a 16-bit source register into bits 23:8 of the 24-bit memory location and zero-fills the low-order bits 7:0.

To implement a linear data buffer, initialize the `Ireg`'s corresponding `Lreg` to `0`. For details, see "DAG Registers" on page 6-5.

To implement circular buffer addressing, initialize the `Ireg`'s corresponding `Lreg` to the length of the buffer and its corresponding `Breg` with the base address of the buffer. For details, see "Circular Data Buffer Addressing" on page 6-14.

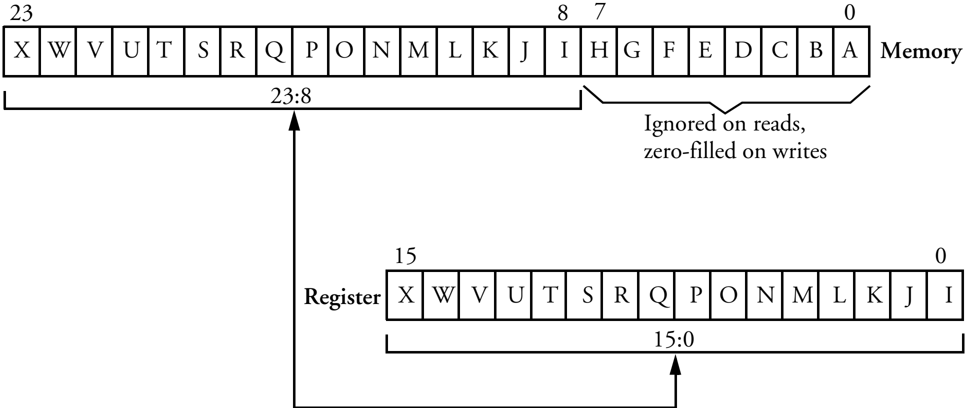


Figure 6-2. 24-bit DM Bus Transactions

To perform bit-reversed addressing, use DAG1 address registers. For details, see “Bit-Reversed Addressing” on page 6-16.

A DAG page register, `DMPG1` (`I3-I0`) or `DMPG2` (`I7-I4`), provide the eight MSBs of the memory address. For details, see “DAG Memory Page Registers (DMPGx)” on page 6-6.

When loading certain registers (`CCODE`, `ASTAT`, `MSTAT`, `IJPG`, `Ireg`, or `DMPGx`), the new value is not available immediately to subsequent instructions. For information on register latencies, see “Register Load Latencies” on page 6-9.

Examples

```
/* This code segment demonstrates indirect 16-bit Memory Reads /*
/* and Writes with postmodify and incurs no stall cycles */
#define taps 10
.SECTION/DM seg_data;
.VAR signal_buffer[taps];
.VAR coeffs[taps];
```

```
.SECTION/PM seg_code;
init:
   I0 = coeff_buffer;
   /* Ireg put addr, label defined */see definition on page 6-15
   I5 = coeffs;
   M0 = 1;
   M5 = 1;
   L0 = LENGTH(coeff_buffer);
   L5 = LENGTH(coeffs);
   AX0 = I0;
   AX1 = I5;
   REG(B0) = AX0;
   REG(B5) = AX1;
   DMPG1 = 0x0;
   DMPG2 = 0x0;
   CNTR = taps;
   DO clear UNTIL CE;
   DM(I5 += M5) = 0;
clear:
   DM(I0 += M0) = 0;
```

See Also

- “Type 32: Any Reg «···» PM/DM” on page 8-50
- “Core Registers” on page 6-2
- “DAG Registers” on page 6-5
- “Secondary DAG Registers” on page 6-7
- “Register Load Latencies” on page 6-9

Indirect 16-Bit Memory Read/Write—Premodify

```
| Dreg   | = DM(Ireg + Mreg) ;
| G1reg  |
| G2reg  |
| G3reg  |
```

```
DM(Ireg + Mreg) =    | Dreg   | ;
                     | G1reg  |
                     | G2reg  |
                     | G3reg  |
```

Function

Transfers 16-bit data between memory and any of the core registers (Dreg, G1reg, G2reg, or G3reg) over the DM bus. The value in Mreg added to the value in Ireg provides the address for the memory access. No update occurs after the access, so Ireg retains its original value.

Source

Reads — The 16-bit data comes from the memory location addressed by the Ireg plus Mreg; the Ireg retains its original value:

- DM/DAG1—I0, I1, I2, or I3 (index registers) M0, M1, M2, or M3 (modify registers)
- PM/DAG2—I4, I5, I6, or I7 (index registers) M4, M5, M6, or M7 (modify registers)

You can use any index register with any modify register from the same DAG. You cannot pair a DAG1 register with a DAG2 register.

Writes — The 16-bit data comes from any core register, except `SSTAT`, which is a read-only register. For information on core registers, see Table 6-1 on page 6-2.

Destination

Reads — The 16-bit data goes to any core register. For information on core registers, see Table 6-1 on page 6-2.

Writes — The 16-bit data goes to the memory location addressed by the `Ireg` plus `Mreg`; the `Ireg` retains its original value.

Details

On reads and writes, the data is right-justified in the destination location (bit0 of the transfer data maps to bit0 of the destination). If the width of the destination register is less than 16 bits, the overflow MSBs of the data are discarded. On writes from source registers less than 16 bits, the missing high-order bits are zero-filled in the memory location.

If this instruction actually references 24-bit data space at runtime, a read operation loads bits 23:8 from the memory location into bits 15:0 of a 16-bit register (or bits 23:16 into bits 7:0 of an 8-bit register, and so on). The low-order bits of the memory location are ignored. Conversely, a write operation stores bits 15:0 from a 16-bit source register into bits 23:8 of the 24-bit memory location and zero-fills the low-order bits 7:0. For details, see Figure 6-2 on page 6-32.

A DAG page register, `DMPG1` (`I3-I0`) or `DMPG2` (`I7-I4`), provides the eight MSBs of the memory address. For details, see "DAG Memory Page Registers (DMPGx)" on page 6-6.

When loading certain registers (CCODE, ASTAT, MSTAT, IJPG, Ireg, or DMPGx), the new value is not available immediately to subsequent instructions. For information on register latencies, see "Register Load Latencies" on page 6-9.

You cannot use circular buffering with this instruction, so you must initialize the Ireg's corresponding Lreg to 0. For details, see "Indirect Addressing" on page 6-12.

To perform bit-reversed addressing, you must use DAG1 address registers. For details, see "Bit-Reversed Addressing" on page 6-16.

Examples

```
.SECTION/DM seg_data;
.VAR look_tbl[3] = 0x0, 0x1, 0x2;

.SECTION/PM seg_code;
cases:
   DMPG1 = 0x1;
   I0 = look_tbl;
   M0 = 0;
   M1 = 1;
   M2 = 2;
   L0 = 0;
   AR = AX0 + AX1;
   IF EQ JUMP cases_end;
case1:
   AY0 = DM(I0 + M0);      /* read from premodified location */
   IF GT JUMP cases_end;
case2:
   DM(I0 + M1) = AY0;      /* write to premodified location */
cases_end:
   NOP;
```

See Also

- "Type 32: Any Reg «···» PM/DM" on page 8-50
- "Core Registers" on page 6-2
- "DAG Registers" on page 6-5
- "Secondary DAG Registers" on page 6-7
- "Register Load Latencies" on page 6-9

Indirect 24-Bit Memory Read/Write—Postmodify

```
| Dreg  | = PM(Ireg += Mreg) ;
| G1reg |
| G2reg |
| G3reg |
```

```
PM(Ireg += Mreg) =     | Dreg  | ;
                       | G1reg |
                       | G2reg |
                       | G3reg |
```

Function

Transfers 24-bit data between memory and any of the core registers (`Dreg`, `G1reg`, `G2reg`, or `G3reg`) over the PM bus. Employs the `PX` register to hold the low-order bits 7:0 while it transfers the high-order bits 23:8 directly between memory and the destination register.

The current value in `Ireg` provides the address for the memory access. After the access, `Ireg` is updated with the sum of its current value and the value in `Mreg`.

Source

Reads — The 24-bit data comes from the memory location pointed to by the address in the `Ireg`, which is modified after the access by the value in the `Mreg`:

- DM/DAG1—`I0`, `I1`, `I2`, or `I3` (index registers) `M0`, `M1`, `M2`, or `M3` (modify registers)
- PM/DAG2—`I4`, `I5`, `I6`, or `I7` (index registers) `M4`, `M5`, `M6`, or `M7` (modify registers)

You can use any index register with any modify register from the same DAG. You cannot pair a DAG1 register with a DAG2 register.

Writes — The 24-bit data comes from any core register, except `SSTAT`, which is a read-only register. For information on core registers, see Table 6-1 on page 6-2.

Destination

Reads — The 24-bit data goes to any core register. For information on core registers, see Table 6-1 on page 6-2.

Writes — The 24-bit data goes to the memory location pointed to by the address in the `Ireg`, which is modified after the access by the value in the `Mreg`.

Details

Unless this instruction is already in the instruction cache, it causes a one-cycle stall.

The 8-bit `PX` register holds the eight low-order bits of 24-bit data transferring between memory and a register. On reads, it automatically stores these bits; on writes, it supplies them. On reads, you must explicitly move the contents of `PX` into a data register; on writes, you must explicitly load the `PX` register with the value of the low-order bits. For details, see "PX Register" on page 6-3 .

On reads, the high-order bits 23:8 of the memory location are right-justified in the destination register (bit8 of the transfer data maps to bit0 of the destination register). If the width of the destination register is less than 16 bits, the overflow MSBs of the data are discarded. For details, see Figure 6-2 on page 6-32.

On writes, bits 15:0 of the source register are right-justified in the memory location (bit0 of the transfer data maps to bit8 of the memory location). If the width of the source register is less than 16 bits, the missing high-order bits of the memory location are zero-filled.

If a PM memory read instruction references 16-bit data space at runtime, the PX register is zeroed.

A DAG page register, DMPG1 (I3-I0) or DMPG2 (I7-I4), provides the eight MSBs of the memory address. For details, see "DAG Memory Page Registers (DMPGx)" on page 6-6.

When you load certain registers (CCODE, ASTAT, MSTAT, IJPG, Ireg, or DMPGx), the new value is not available immediately to subsequent instructions. For information on register latencies, see "Register Load Latencies" on page 6-9.

To implement a linear data buffer, you must initialize the Ireg's corresponding Lreg to 0. For details, see "Indirect Addressing" on page 6-12.

To implement circular buffer addressing, you must initialize the Ireg's corresponding Lreg to the length of the buffer and its corresponding Breg with the base address of the buffer. For details, see "Circular Data Buffer Addressing" on page 6-14.

To perform bit-reversed addressing, you must use DAG1 address registers. For details, see "Bit-Reversed Addressing" on page 6-16.

Examples

```
#define more_taps 10
.SECTION/DM seg_dmda;
.VAR dmdata_buffer[more_taps];
.SECTION/PM seg_pmda;
.VAR pmdata_coeffs[more_taps];
.SECTION/PM seg_code;
more_init:
```

```
    I0 = dmdata_buffer;       /* dmdag Ireg write/output address */
    I5 = pmdata_coeffs;       /* pmdag Ireg read/input address */
    M0 = 1;
    M5 = 1;
    L0 = LENGTH(dmdata_buffer);
    L5 = LENGTH(pmdata_coeffs);
    AX0 = I0;
    AX1 = I5;
    REG(B0) = AX0;
    REG(B5) = AX1;
    DMPG1 = 0x0;
    DMPG2 = 0x0;
    CNTR = taps;
    SI = 0xB6A3;             /* shifter input word */
    DO clear UNTIL CE;
    SR1 = PM(I5 += M5);      /* read upper 16-bits & post modify */
    SR0 = PX;                /* read lower 8-bits from PX */
    SR = SR OR ASHIFT SI BY 3 (HI);    /* ashift upper word */
more_clear:
    DM(I0 += M0) = SR0;
                        /* 16-bit write SR0 & post modify address */
```

See Also

- “Type 32: Any Reg «···» PM/DM” on page 8-50
- “Core Registers” on page 6-2
- “DAG Registers” on page 6-5
- “Secondary DAG Registers” on page 6-7
- “Register Load Latencies” on page 6-9

Indirect 24-Bit Memory Read/Write—Premodify

```
| Dreg     | = PM(Ireg + Mreg) ;
| G1reg    |
| G2reg    |
| G3reg    |
```

```
PM(Ireg + Mreg) =       | Dreg     | ;
                        | G1reg    |
                        | G2reg    |
                        | G3reg    |
```

Function

Transfers 24-bit data between memory and any of the core registers (Dreg, G1reg, G2reg, or G3reg) over the PM bus. Employs the PX register to hold the low-order bits 7:0 while it transfers the high-order bits 23:8 directly between memory and the destination register. The value in Mreg added to the value in Ireg provides the address for the memory access. No update occurs after the access, so Ireg retains its original value.

Source

Reads — The 24-bit data comes from the memory location addressed by the Ireg plus Mreg; the Ireg retains its original value:

- DM/DAG1—I0, I1, I2, or I3 (index registers) M0, M1, M2, or M3 (modify registers)
- PM/DAG2—I4, I5, I6, or I7 (index registers) M4, M5, M6, or M7 (modify registers)

You can use any index register with any modify register from the same DAG. You cannot pair a DAG1 register with a DAG2 register.

Writes — The 24-bit data comes from any core register, except SSTAT, which is a read-only register. For information on core registers, see Table 6-1 on page 6-2.

Destination

Reads — The 24-bit data goes to any core register. For information on core registers, see Table 6-1 on page 6-2.

Writes — The 24-bit data goes to the memory location addressed by the Ireg plus Mreg; the Ireg retains its original value

Details

Unless this instruction is already in the instruction cache, it causes a one-cycle stall.

The 8-bit PX register holds the eight low-order bits of 24-bit data transferring between memory and a register. On reads, it automatically stores these bits, and on writes, it supplies them. On reads, you must explicitly move the contents of PX into a data register, and on writes, you must explicitly load the PX register with the value of the low-order bits. For details, see "PX Register" on page 6-3.

On reads, the high-order bits 23:8 of the memory location are right-justified in the destination register (bit8 of the transfer data maps to bit0 of the destination register). If the width of the destination register is less than 16 bits, the overflow MSBs of the data are discarded. For details, see Figure 6-2 on page 6-32.

On writes, bits 15:0 of the source register are right-justified in the memory location (bit0 of the transfer data maps to bit8 of the memory location). If the width of the source register is less than 16 bits, the missing high-order bits of the memory location are zero-filled.

A DAG page register, DMPG1 (I3-I0) or DMPG2 (I7-I4), provides the eight MSBs of the memory address. For details, see "DAG Memory Page Registers (DMPGx)" on page 6-6.

When you load certain registers (CCODE, ASTAT, MSTAT, IJPG, Ireg, or DMPGx), the new value is not available immediately to subsequent instructions. For information on register latencies, see "Register Load Latencies" on page 6-9.

You cannot use circular buffering with this instruction, so you must initialize the Ireg's corresponding Lreg to 0. For details, see "Indirect Addressing" on page 6-12.

To perform bit-reversed addressing, you must use DAG1 address registers. For details, see "Bit-Reversed Addressing" on page 6-16.

Examples

```
.SECTION/DM seg_data;
.VAR lookup_tbl[3] = 0x0, 0x1, 0x2;

.SECTION/PM seg_code;
pmrw_cases:
   DMPG1 = 0x1;
   I0 = lookup_tbl;
   M0 = 0;
   M1 = 1;
   M2 = 2;
   L0 = 0;
   AR = AX0 + AX1;
   IF EQ JUMP cases_end;
```

```
pmrw_case1:
   SR1 = PM(I0 + M1);     /* premodify and read upper 16-bits */
   SR0 = PX;              /* read lower 8-bits from PX */
   SR = SR OR ASHIFT SI BY 3 (HI);   /* ashift upper word */
   IF GT JUMP cases_end;
pmrw_case2:
   PX = SR1;              /* Load lower 8 bits into PX */
   PM(I0 + M2) = SR0;     /* Write all 24 bits from SR0 and PX */
pmrw_cases_end:
   NOP;
```

See Also

- "Type 32: Any Reg «···» PM/DM" on page 8-50
- "Core Registers" on page 6-2
- "DAG Registers" on page 6-5
- "Secondary DAG Registers" on page 6-7
- "Register Load Latencies" on page 6-9

Indirect DAG Register Write (Premodify or Postmodify), with DAG Register Move

```
| DM(Ireg1 | +  | Mreg1) = | Ireg2 |  ,  | Ireg2 | = Ireg1 ;
|          | += |          | Mreg2 |     | Mreg2 |
|          |    |          | Lreg2 |     | Lreg2 |
```

Function

Writes the contents of a source address register to memory and loads the same source address register with a new value—the effective address written to memory. Register usage within this instruction has the following restrictions (shown below graphically):

- The `Ireg1` registers must be the same register.
- The `Mreg1` register must come from the same DAG as `Ireg1`.
- The `Ireg2`, `Mreg2`, or `Lreg2` registers be the same register.
- The `Ireg2`, `Mreg2`, or `Lreg2` registers must come from the same DAG as `Ireg1`, but may not be `Ireg1` (`Ireg1` 1/4 `Ireg2`).

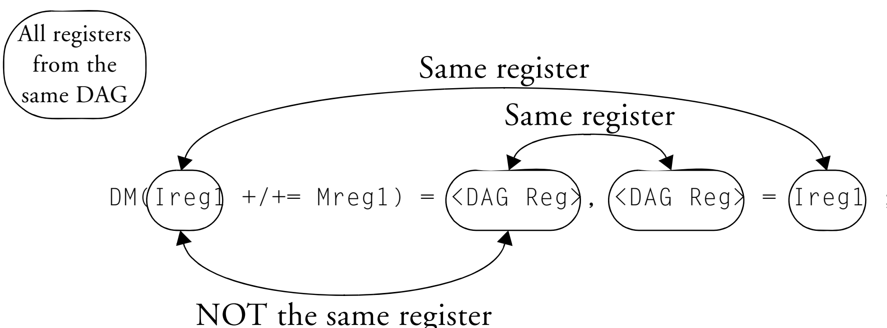


Figure 6-3. Register Sources

If the premodify (+) addressing option is used, after the memory access, the instruction loads the source address register with the modified value of Ireg (Ireg + Mreg). If the postmodify (+=) addressing option is used, the instruction loads the source address register with the unmodified value of Ireg. For details on the indirect addressing options, see "Indirect Addressing" on page 6-12.

Source

Memory write	The 16-bit data comes from any DAG (Ireg, Mreg, or Lreg) register in the same DAG as Ireg1. The source register may not be the Ireg1 register. For information on core registers, see Table 6-1 on page 6-2.
Register load	The 16-bit data comes from the Ireg1 register.

Destination

Memory write

For the post-modified access (+=), the 16-bit data goes to the memory location pointed to by the address in the Ireg, which is modified after the access by the value in the Mreg.

For the pre-modified access (+), the 16-bit data goes to the memory location addressed by the Ireg plus Mreg; the Ireg retains its original value:

- DM/DAG1—I0, I1, I2, or I3 (index registers) M0, M1, M2, or M3 (modify registers)
- PM/DAG2—I4, I5, I6, or I7 (index registers) M4, M5, M6, or M7 (modify registers)

You can use any index register with any modify register from the same DAG. You cannot pair a DAG1 register with a DAG2 register.

Register load	The 16-bit data goes to any DAG (Ireg, Mreg, or Lreg) register in the same DAG as Ireg1. The destination register may not be the Ireg1 register. For information on core registers, see Table 6-1 on page 6-2.

Details

On reads and writes, the data is right-justified in the destination location (bit0 of the transfer data maps to bit0 of the destination). If the width of the destination register is less than 16 bits, the overflow MSBs of the data are discarded. On writes from source registers less than 16 bits, the missing high-order bits are zero-filled in the memory location.

If this instruction actually references 24-bit data space at runtime, a read operation loads bits 23:8 from the memory location into bits 15:0 of a 16-bit register (or bits 23:16 into bits 7:0 of an 8-bit register, and so on). The low-order bits of the memory location are ignored. Conversely, a write operation stores bits 15:0 from a 16-bit source register into bits 23:8 of the 24-bit memory location and zero-fills the low-order bits 7:0. For details, see Figure 6-2 on page 6-32.

A DAG page register, DMPG1 (I3-I0) or DMPG2 (I7-I4), provides the eight MSBs of the memory address. For details, see "DAG Memory Page Registers (DMPGx)" on page 6-6.

When you load certain registers (CCODE, ASTAT, MSTAT, IJPG, Ireg, or DMPGx), the new value is not available immediately to subsequent instructions. For information on register latencies, see "Register Load Latencies" on page 6-9.

You cannot use circular buffering with the pre-modify addressing (+) form of this instruction, so you must initialize the Ireg's corresponding Lreg to 0. For details, see "Indirect Addressing" on page 6-12.

To perform bit-reversed addressing, you must use DAG1 address registers. For details, see "Bit-Reversed Addressing" on page 6-16.

Examples

```
/* This routine uses the "type 32a" instruction to save */
/* the previous frame pointer and allocate a new frame pointer */
/* before calling C-callable subroutine. */

_memalloc:
DM(I4 += M5)=I5, I5=I4;  /* save old FP and allocate new FP */
AX1 = DM(I5 + 1);        /* read a 16 bit parameter from stack */
I2 = 0xFFFF;             /* load preserved register I2 */
I6 = 0xFFFF;             /* load scratch register I6 */
DM(I4 += M5) = AX1;      /* put argument on stack for call */
I7 = I6;                 /* save scratch register I6 */
CALL _malloc;            /* call C function malloc */

_malloc:
   NOP;                  /* _malloc code here */
```

See Also

- "Type 32a: DM«···DAG Reg | DAG Reg«···Ireg" on page 8-51
- "Core Registers" on page 6-2
- "DAG Registers" on page 6-5
- "Secondary DAG Registers" on page 6-7
- "Register Load Latencies" on page 6-9

Indirect Memory Read/Write—Immediate Postmodify

```
Dreg = DM(Ireg += <Imm8>) ;
```

```
DM(Ireg += <Imm8>) = Dreg ;
```

Function

Transfers 16-bit data between memory and a data register over the DM bus. The current value in `Ireg` provides the address for the memory access. After the access, `Ireg` is updated with the sum of its current value and the immediate 8-bit, twos complement value supplied in the instruction.

Source

Reads — The contents of a memory location in data space pointed to by `Ireg` (`I0-I7`).

Writes — Any of these data registers:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

Destination

Reads — A data register (same as data registers—write).

Writes — The contents of a memory location in data space pointed to by `Ireg` (same as read source registers).

Details

The immediate value supplied in the instruction is an 8-bit two's-complement number. Valid values range from –`128` through `127`.

On reads and writes, the data is right-justified in the destination location (bit0 of the transfer data maps to bit0 of the destination).

If this instruction actually references 24-bit data space at runtime, a read operation loads bits 23:8 from the memory location into bits 15:0 of a 16-bit register (or bits 23:16 into bits 7:0 of an 8-bit register, and so on). The low-order bits of the memory location are ignored. Conversely, a write operation stores bits 15:0 from a 16-bit source register into bits 23:8 of the 24-bit memory location and zero-fills the low-order bits 7:0. For details, see Figure 6-2 on page 6-32.

A DAG page register, DMPG1 (I3-I0) or DMPG2 (I7-I4), provides the eight MSBs of the memory address. For details, see "DAG Memory Page Registers (DMPGx)" on page 6-6.

When you load certain registers (CCODE, ASTAT, MSTAT, IJPG, Ireg, or DMPGx), the new value is not available immediately to subsequent instructions. For information on register latencies, see "Register Load Latencies" on page 6-9.

To implement a linear data buffer, you must initialize the Ireg's corresponding Lreg to 0. For details, see "Indirect Addressing" on page 6-12.

To implement circular buffer addressing, you must initialize the Ireg's corresponding Lreg with the length of the buffer and its corresponding Breg with the base address of the buffer. For details, see "Circular Data Buffer Addressing" on page 6-14.

To perform bit-reversed addressing, you must use DAG1 address registers. For details, see "Bit-Reversed Addressing" on page 6-16.

Examples

```
AX0 = DM(I0 += 0x11);
DM(I6 += 0x08) = MR1;
DM(I2 += -3) = SI;
```

See Also

- "Type 29: Dreg «···» DM" on page 8-47
- "Core Registers" on page 6-2
- "DAG Registers" on page 6-5
- "Secondary DAG Registers" on page 6-7
- "Register Load Latencies" on page 6-9

Indirect Memory Read/Write—Immediate Premodify

```
Dreg = DM(Ireg + <Imm8>) ;
```

```
DM(Ireg + <Imm8>) = Dreg ;
```

Function

Transfers 16-bit data between memory and a data register over the DM bus. The immediate 8-bit, twos complement value supplied in the instruction added to the current value in `Ireg` provides the address for the memory access. No update occurs after the access, so `Ireg` retains its original value.

Source

Reads — The contents of a memory location in data space, accessed indirectly using an `Ireg` (`I0-I7`) and an immediate 8-bit, twos complement value supplied in the instruction.

Writes — Any of these data registers:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

Destination

Reads — A data register (same as source data registers).

Writes — The contents of a memory location in data space, accessed indirectly with an `Ireg` (same as source address registers) and immediate 8-bit, twos complement value supplied in the instruction.

Details

The immediate value supplied in the instruction is an 8-bit twos complement number. Valid values range from –128 through 127.

On reads and writes, the data is right-justified in the destination location (bit0 of the transfer data maps to bit0 of the destination).

If this instruction actually references 24-bit data space at runtime, a read operation loads bits 23:8 from the memory location into bits 15:0 of a 16-bit register (or bits 23:16 into bits 7:0 of an 8-bit register, and so on). The low-order bits of the memory location are ignored. Conversely, a write operation stores bits 15:0 from a 16-bit source register into bits 23:8 of the 24-bit memory location and zero-fills the low-order bits 7:0. For details, see Figure 6-2 on page 6-32.

A DAG page register, DMPG1 (I3-I0) or DMPG2 (I7-I4), provides the eight MSBs of the memory address. For details, see “DAG Memory Page Registers (DMPGx)” on page 6-6.

When you load certain registers (CCODE, ASTAT, MSTAT, IJPG, Ireg, or DMPGx), the new value is not available immediately to subsequent instructions. For information on register latencies, see “Register Load Latencies” on page 6-9.

You cannot use circular buffering with this instruction, so you must initialize the Ireg’s corresponding Lreg to 0. For details, see “Indirect Addressing” on page 6-12.

To perform bit-reversed addressing, you must use DAG1 address registers. For details, see “Bit-Reversed Addressing” on page 6-16.

Examples

```
AX0 = DM(I0 + 0x11);
DM(I6 + 0x08) = MR1;
DM(I2 + -3) = SI;
```

See Also

- "Type 29: Dreg «…» DM" on page 8-47
- "Core Registers" on page 6-2
- "DAG Registers" on page 6-5
- "Secondary DAG Registers" on page 6-7
- "Register Load Latencies" on page 6-9

Indirect 16-Bit Memory Write—Immediate Data

```
DM(Ireg += Mreg) = <Data16> ;
```

Function

Writes a 16-bit data value supplied in the instruction to a memory location over the DM bus. The current value in `Ireg` provides the address for the memory access. After the memory access, `Ireg` is updated with the sum of its current value and the value in `Mreg`.

This instruction is a two-word instruction and requires (at minimum) two cycles to execute. For more information, see “Type 22: DM «··· Data16” on page 8-41.

Source

`<data16>`	The data comes from a 16-bit number supplied in the instruction.

Destination

Memory write — The 16-bit data goes to the memory location pointed to by the address in the `Ireg`, which is modified after the access by the value in the `Mreg`:

- DM/DAG1—`I0`, `I1`, `I2`, or `I3` (index registers) `M0`, `M1`, `M2`, or `M3` (modify registers)
- PM/DAG2—`I4`, `I5`, `I6`, or `I7` (index registers) `M4`, `M5`, `M6`, or `M7` (modify registers)

You can use any index register with any modify register from the same DAG. You cannot pair a DAG1 register with a DAG2 register.

Details

The data transferred is right-justified in the memory location (bit0 of the data maps to bit0 of the location). If this instruction actually accesses 24-bit data space at runtime, the write operation stores bits 15:0 in bits 15:0 of the 24-bit memory location and zero-fills the high-order bits 23:16.

A DAG page register, DMPG1 (I3-I0) or DMPG2 (I7-I4), provides the eight MSBs of the memory address. For details, see "DAG Memory Page Registers (DMPGx)" on page 6-6.

When you load certain registers (CCODE, ASTAT, MSTAT, IJPG, Ireg, or DMPGx), the new value is not available immediately to subsequent instructions. For information on register latencies, see "Register Load Latencies" on page 6-9.

Setting up a data buffer requires initializing one or more additional address registers. For details, see "Indirect Addressing" on page 6-12.

To perform bit-reversed addressing, you must use DAG1 address registers. For details, see "Bit-Reversed Addressing" on page 6-16.

Examples

```
DMPG1 = page(0x0);                /* selects internal memory */
I3 = 0x8100;                      /* selects 16-bit, block 1 */
M2 = 1;
L3 = 0;
DM(I3 += M2) = 0x7743;            /* write data16 to memory */
```

See Also

- "Type 22: DM «··· Data16" on page 8-41
- "Core Registers" on page 6-2
- "DAG Registers" on page 6-5

Indirect 24-Bit Memory Write—Immediate Data

```
PM(Ireg += Mreg) = <Data24>:24 ;
```

Function

Writes a 24-bit data value supplied in the instruction to a memory location over the PM bus. The current value in `Ireg` provides the address for the memory access. After the memory access, `Ireg` is updated with the sum of its current value and the value in `Mreg`.

This instruction is a two-word instruction and requires (at minimum) two cycles to execute. For more information, see “Type 22: DM «··· Data16” on page 8-41.

Source

`<data24>`	The 24-bit data comes from a 24-bit number supplied in the instruction. The `:24` after the data directs the assembler to handle 24-bit data.

Destination

Memory write — The 24-bit data goes to the memory location pointed to by the address in the `Ireg`, which is modified after the access by the value in the `Mreg`:

- DM/DAG1—`I0`, `I1`, `I2`, or `I3` (index registers) `M0`, `M1`, `M2`, or `M3` (modify registers)
- PM/DAG2—`I4`, `I5`, `I6`, or `I7` (index registers) `M4`, `M5`, `M6`, or `M7` (modify registers)

You can use any index register with any modify register from the same DAG. You cannot pair a DAG1 register with a DAG2 register.

Details

The data transferred is right-justified in the memory location (bit0 of the data maps to bit0 of the location).

If this instruction actually accesses 16-bit data space at runtime, the write operation stores bits 23:8 in bits 15:0 of the 16-bit memory location and discards the data's low-order bits 7:0. For details, see Figure 6-2 on page 6-32.

A DAG page register, DMPG1 (I3-I0) or DMPG2 (I7-I4), provides the eight MSBs of the memory address. For details, see "DAG Memory Page Registers (DMPGx)" on page 6-6.

When you load certain registers (CCODE, ASTAT, MSTAT, IJPG, Ireg, or DMPGx), the new value is not available immediately to subsequent instructions. For information on register latencies, see "Register Load Latencies" on page 6-9.

Setting up a data buffer requires initializing one or more additional address registers. For details, see "Indirect Addressing" on page 6-12.

To perform bit-reversed addressing, you must use DAG1 address registers. For details, see "Bit-Reversed Addressing" on page 6-16.

Examples

```
DMPG2 = page(0x0);             /* selects internal memory */
I4 = 0x1000;                   /* selects 24-bit, block 0 */
M5 = 1;
L4 = 0;
PM(I4 += M5) = 0x3F4512:24;    /* write 24-bit data to memory */
```

See Also

- “Type 22: DM «··· Data16” on page 8-41
- “Core Registers” on page 6-2
- “DAG Registers” on page 6-5

External I/O Port Read/Write

```
Dreg = IO(<Imm10>) ;

IO(<Imm10>) = Dreg ;
```

Function

Transfers data between I/O memory space and a data register.

Source

Reads — The contents of a location in I/O memory space specified by the 10-bit immediate value (or memory-mapped register name) supplied in the instruction.

Writes — Any of these data registers:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

Destination

Reads — Any of the data registers (same as write source registers).

Writes — The contents of a location in I/O memory space specified by the 10-bit immediate value (or memory-mapped register name) supplied in the instruction.

Details

The I/O Page (IOPG) register, with the 10-bit immediate value supplied in the instruction, generates the 18-bit address required for accessing I/O memory space. Valid page values (IOPG) are 0-255. Valid location values (10-bit immediate) are 0-1023. The arrangement of these values to make an address appears in Figure 6-4.

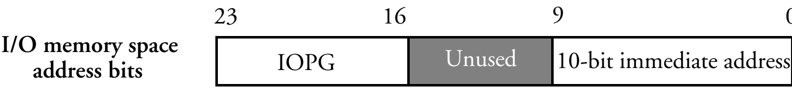


Figure 6-4. Addressing I/O Memory Space and Peripherals

At power-up, the DSP initializes the IOPG register to 0x0. Although initializing IOPG is unneccessary unless the data to access is located on a different page, good programming practice recommends that you do so whenever you access an external I/O port. To do so, you use the direct register load instruction (for details, see "Direct Register Load" on page 6-27). Then you can read or write the external I/O port.

When loading certain registers (CCODE, ASTAT, MSTAT, IJPG, Ireg, DMPGx, or IOPG), the new value is not available immediately to subsequent instructions. For information on register latencies, see "Register Load Latencies" on page 6-9.

Examples

```
#define io_address1 0x1FF       /* define 10-bit IO address */
IOPG = 0x01;                    /* set IOPG to page 1 */
AX0 = 0xaaff;                   /* load AX0 with 16-bit data */
IO(io_address1) = AX0;          /* write data to io_address1 */
```

See Also

- "Type 34: Dreg «···» IOreg" on page 8-53
- The I/O registers are specific to each ADSP-219x DSP. For register list, see the *ADSP-219x/2191 DSP Hardware Reference*.

System Control Register Read/Write

```
Dreg = REG(<Imm8>) ;

REG(<Imm8>) = Dreg ;
```

Function

Transfers data between an internal system control register and a data register.

Source

Reads — The contents of a location in system control memory space specified by the 8-bit immediate value, or register name, supplied in the instruction:

- DAG `Breg`—`B0`, `B1`, `B2`, or `B3` (DAG1 base registers) `B4`, `B5`, `B6`, or `B7` (DAG2 base registers)
- Sequencer—`SYSCTL` (system control register)
- Cache control—`CACTL` (cache control register)

Except for the DAG base address registers (`B0`–`B7`), `SYSCTL`, and `CACTL`, the system control registers are specific to each ADSP-219x product. Refer to the *ADSP-219x/2192 DSP Hardware Reference* for a complete list of these registers and their addresses.

Writes — Any of these data registers:

Register File
AX0, AX1, AY0, AY1, AR, MX0, MX1, MY0, MY1, MR0, MR1, MR2, SR0, SR1, SR2, SI

Destination

Reads	Any of the data registers (same as write source registers).
Writes	The contents of a location in system control memory space specified by the 8-bit immediate value, or register mnemonic, supplied in the instruction (same as source registers).

Details

System control memory space consists of 256 locations. These locations are reserved for core-based controls or for peripherals, such as DMA or serial ports, that interface with the core. For more information, see the *ADSP-219x/2192 DSP Hardware Reference*.

You cannot write the system control registers directly. Instead, you must load a data register, then load the system register from the data register.

When you access a DAG base address register (B0-B7), whether you access the primary or secondary set depends on bit 6 (SEC_DAG) in the MSTAT register. For details, see "Secondary DAG Registers" on page 6-7.

When you load certain registers (CCODE, ASTAT, MSTAT, IJPG, Ireg, or DMPGx), the new value is not available immediately to subsequent instructions. For information on register latencies, see "Register Load Latencies" on page 6-9.

Examples

```
AX0 = 0x0800;                /* load data into AX0 */
REG(B0) = AX0;               /* load AX0 into B0   */
REG(0x0) = AX0;              /* same as above      */
```

See Also

- "Type 35: Dreg «···»Sreg" on page 8-54
- The system registers are specific to each ADSP-219x DSP. For register list, see the *ADSP-219x/2192 DSP Hardware Reference*.

Modify Address Register—Indirect

```
MODIFY(Ireg += Mreg) ;
```

Function

Updates the value of an Index register without performing a memory access.

Sums the value in `Ireg` with the value in `Mreg` and writes the result to `Ireg`. If you set up circular buffering, this instruction also performs that logic operation.

Source

Update	The DAG `Ireg` and `Mreg` registers specified in the instruction: • DM/DAG1—`I0`, `I1`, `I2`, or `I3` (index registers) `M0`, `M1`, `M2`, or `M3` (modify registers) • PM/DAG2—`I4`, `I5`, `I6`, or `I7` (index registers) `M4`, `M5`, `M6`, or `M7` (modify registers)

You can use any index register with any modify register from the same DAG. You cannot pair a DAG1 register with a DAG2 register.

Destination

Update	The DAG `Ireg` specified in the instruction.

Details

For linear data buffers, you must initialize the `Ireg`'s corresponding `Lreg` to `0`. For details, see "Indirect Addressing" on page 6-12.

For circular buffers, you must initialize the `Ireg`'s corresponding `Lreg` with the length of the buffer and its corresponding `Breg` with the base address of the buffer. For details, see "Circular Data Buffer Addressing" on page 6-14.

When loading certain registers (`CCODE`, `ASTAT`, `MSTAT`, `IJPG`, `Ireg`, `DMPGx`, or `IOPG`), the new value is not available immediately to subsequent instructions. For information on register latencies, see "Register Load Latencies" on page 6-9.

Examples

```
I2 = 0x2000;
M1 = 1;
L2 = 0;
MODIFY(I2 += M1);          /* updates I2 with value from M1 */
                            /* I2 = 0x2001 */
```

See Also

- "Type 21: Modify DagI" on page 8-39
- "Core Registers" on page 6-2
- "DAG Registers" on page 6-5

Modify Address Register—Direct

```
MODIFY(Ireg += <Imm8>) ;
```

Function

Updates the value of an index register without performing a memory access.

Sums the value in `Ireg` with the 8-bit, twos complement immediate value supplied in the instruction and writes the result to `Ireg`. If you set up circular buffering, this instruction also performs that logic operation.

Source

The DAG `Ireg` register and the `Imm8` data specified in the instruction:

- DM/DAG1—`I0`, `I1`, `I2`, or `I3` (index registers)
- PM/DAG2—`I4`, `I5`, `I6`, or `I7` (index registers)

Destination

The DAG `Ireg` specified in the instruction.

Details

For linear data buffers, you must initialize the `Ireg`'s corresponding `Lreg` to `0`. For details, see "Indirect Addressing" on page 6-12.

For circular buffers, you must initialize the `Ireg`'s corresponding `Lreg` to the length of the buffer and its corresponding `Breg` with the base address of the buffer. For details, see "Circular Data Buffer Addressing" on page 6-14.

When loading certain registers (CCODE, ASTAT, MSTAT, IJPG, Ireg, DMPGx, or IOPG), the new value is not available immediately to subsequent instructions. For information on register latencies, see "Register Load Latencies" on page 6-9.

Examples

```
I2 = 0x2000;
#define mod_val 0x1
Nop;
L2 = 0;
MODIFY(I2 += mod_val);      /* updates I2 with mod_val */
                            /* I2 = 0x2001 */
```

See Also

- "Type 21a: Modify DagI" on page 8-40
- "Core Registers" on page 6-2
- "DAG Registers" on page 6-5

7 PROGRAM FLOW INSTRUCTIONS

The instruction set provides program flow instructions for controlling the sequence in which the DSP executes instructions. Generally, the instructions in a program execute sequentially, one after another, unless otherwise directed by various program structures—branches, loops, subroutines, or interrupts—that intervene and temporarily or permanently redirect this linear flow. These program structures enable an application to respond to events or conditions as they occur.

This chapter describes each of the instructions ("Program Flow Instruction Reference" on page 7-23) and the following related topics:

- "Conditions" on page 7-2
- "Counter-Based Conditions" on page 7-2
- "CCODE Register" on page 7-3
- "Mode Control" on page 7-4
- "Branch Options" on page 7-4
- "Addressing Branch Targets" on page 7-6
- "Stacks" on page 7-7
- "Stack Status Flags" on page 7-12
- "Interrupts" on page 7-13
- "Application Performance" on page 7-17

Conditions

Table 1-8 on page 1-11 lists the conditions used in conditional (IF COND) instructions and their opcodes. Besides these conditions (which mainly relate to the status of the ALU, multiplier, and counter) it is possible to use the SWCOND condition and the value in the CCODE register to test for other DSP status conditions. For more information, see "Condition Code (CCODE) Register" on page 1-5 Also, you can test for bit states to generate conditions using the TSTBIT instruction. For more information, see the section "Bit Manipulation: TSTBIT, SETBIT, CLRBIT, TGLBIT" on page 2-19.

Counter-Based Conditions

Both IF Condition (conditional) instructions and DO UNTIL (loop) instructions can base execution on the NOT CE condition. Although the DO UNTIL instruction uses the CE syntax only, the condition actually tested is NOT CE—counter not expired.

To use a counter condition with either instruction type, load the CNTR register with an initial counter value (>1) before issuing the instruction that uses the counter condition.

There are important differences between how conditional and loop instructions implement (decrement and test) the counter condition:

- To implement the NOT CE condition in an IF Condition (conditional) instruction, the DSP decrements and tests the value loaded in the CNTR register before executing the conditional instruction. For a conditional instruction based on NOT CE, the DSP tests whether the CNTR register contains a value >1.
- To implement the CE condition in a DO UNTIL (loop) instruction, the DSP loads the loop counter stack from the CNTR register at the start of the loop, then decrements and tests the counter value in the

loop counter stack (not the CNTR register) at the end of each pass through the loop. For a loop instruction based on CE, the DSP tests whether the loop counter stack's counter value >0. For more information, see "Loop Stacks Operation" on page 7-10.

CCODE Register

Table 1-3 on page 1-5 lists the CCODE register values used to test the SWCOND and NOT SWCOND software conditions. Although the source each value tests is specific to each DSP in the ADSP-219x family, these values (except 0x08 and 0x09) map to software interrupt bits in the IMASK and IRPTL registers.

To test for a software condition, load the CCODE register with the value of the source you want to test and test for the true or false state. For example, 0x08 represents ALU saturation status, you might code this sequence:

```
CCODE = 0x08;                /* ALU Saturated (AR_SAT) cond */
AR = AX0 + AY1;
IF SWCOND JUMP fix_data;     /* Jump to fix_data if AR_SAT */

fix_data:
   NOP;                      /* code to fix data ALU_SAT */
```

Or, to test for a shifter overflowed result:

```
CCODE = 0x09;               /* Shifter Overflowed (SV) cond */
AR = 3; SE = AR;            /* shift code, left shift 3 bits */
SI = 0xB6A3;       /* value of hi word of input data */
IF NOT SWCOND SR = ASHIFT SI (HI);/* ASHIFT high word if SV */
```

A value written to CCODE is not available on the next cycle, so you must insert at least one instruction between the write to CCODE and the conditional instruction that tests the software condition. Otherwise, the conditional instruction tests the previous value of CCODE.

Mode Control

As shown in Table 1-6 on page 1-8, bits 0 through 7 of the MSTAT register control various DSP modes. These modes determine some conditions for how status flags are set.

Branch Options

All of the DSP's branch instructions (except DO UNTIL and LJUMP/LCALL), support two branch options: immediate and delayed. These options determine whether the DSP executes the first two instructions following the branch instruction before executing the instruction at the branch target address. Because of the instruction pipeline, a number (usually four) of latency cycles occur between execution of the branch instruction and execution of the branch target instruction.

By default, branch instructions perform an immediate branch, which means that the next instruction the DSP executes after the branch instruction is the instruction at the branch target address, but only after a number of NOP cycles. The delayed branch option allows you to salvage two of the NOP cycles and perform useful work. To do so, include the delayed branch (DB) option in the branch instruction and code the two delay slots directly following the branch instruction the two instructions that you want executed before the branch target instruction.

You cannot insert `JUMP` or `CALL` instructions in delay branch slots. You can insert one two-word instruction only, and it must occupy the first delay branch slot.

When the DSP executes an `RTI` or `RTS` instruction to return to the main program, it returns to execute the first or third instruction after the branch instruction, depending on whether the branch is immediate or delayed.

Return from immediate branch:

```
IF AV CALL immediate_pump;    /* immed branches may be cond */
   NOP;                   /* RTS returns program flow here */
   NOP;

immediate_pump:
   NOP;
   RTS;
```

Return from delayed branch:

```
CALL delayed_pump (DB);  /* delayed branches must be uncond */
   NOP;                   /* 1st_delay_instruction */
   NOP;                   /* 2nd_delay_instruction */
   NOP;                   /* RTS returns program flow here */
   NOP;

delayed_pump:
   NOP;
   RTS;
```

Addressing Branch Targets

When issuing a JUMP or CALL instruction, specify the address of the instruction to branch to in one of three ways:

- PC relative (Offset from the current PC)
  The immediate value specified in the instruction is added to the PC of the branch instruction to form the address of the branch target location. For example, the CALL in the following code goes to PC-relative address (find_me):

```
.EXTERN find_me;   /* matches .global in other file */
CALL find_me (DB);

NOP;
```

- Far absolute
  The full 24-bit address of the branch target location is specified in the instruction. You can program this instruction explicitly in an LJUMP/LCALL instruction. The assembler substitutes this instruction automatically when the actual address assembled from a PC relative address is insufficient.

- Indirect
  The address of the branch target location is specified using a DAG Index register (I0–I7) and the Indirect jump memory page (IJPG) register. For example, the CALL in the following code goes to an address using the indirect address from the I0 register:

```
.EXTERN find_me_too;/* matches .global in other file */
IJPG = 0x0;             /* set memory page for CALL */
I0 = find_me_too;       /* loads I0 with address */
NOP;
NOP;
CALL (I0) (DB);
NOP;
```

Stacks

Loops and other branch instructions use the DSP's stacks to implement their respective operations.

- PC stack (33 words × 24 bits)

  Holds the address of the next instruction to execute on return from a called subroutine. Only the `CALL`, `RTI`, `RTS`, and `PUSH/POP PC` instructions use this stack.

- Loop begin stack (8 words × 24 bits)

  Holds the address of the first instruction in a loop. Only the `DO UNTL` and `PUSH/POP LOOP` instructions use this stack.

- Loop end stack (8 words × 24 bits)

  Holds the address of the last instruction in a loop. Only the `DO UNTL` and `PUSH/POP LOOP` instructions use this stack.

- Loop counter stack (8 words × 16 bits)

  Holds the current value of the loop counter that is loaded from the `CNTR` register. This value—not the value in the `CNTR` register—is tested and decremented at the end of each pass through the loop. Only the `DO UNTL` and `PUSH/POP LOOP` instructions use this stack.

- Status stack (16 words × 32 bits)

  Holds the current value of the `ASTAT` and `MSTAT` registers. Only the `RTI` and `PUSH/POP STS` instructions use this stack. (When globally enabled and unmasked interrupts occur, the DSP automatically saves the two status registers to this stack.)

PC and Status Stack Operation

Applications use these stacks to implement function calls and interrupt service routines (ISRs).

Function Calls. When a CALL instruction executes, it automatically pushes onto the PC stack the address of the next instruction to execute upon returning from the subroutine. The CALL instruction does not push the status registers onto the status stack.

The RTS instruction, executed at the end of the subroutine, returns program execution to the first or third instruction following the CALL instruction, depending on whether the CALL was immediate or delayed, respectively.

ISRs. When interrupts are globally enabled, an unmasked interrupt causes the DSP to automatically save its current state before entering the interrupt's ISR. To do so, the DSP:

- Pushes onto the PC stack the address of the next instruction to execute upon returning from the ISR.

  If the interrupt is higher than the core's current level of operation, the DSP pushes the address of the current instruction onto the PC stack and branches immediately to the interrupt's ISR.

  If the interrupt is lower than the core's current level of operation, the DSP finishes the current operation, pushes the address of the next sequential instruction onto the PC stack, and then branches to the interrupt's ISR.

- Pushes onto the status stack, in order, the ASTAT and MSTAT registers.

The RTI instruction, executed at the end of the ISR, pops the PC stack returning program execution to the instruction at the retrieved address. It also pops the status stack, restoring the ASTAT and MSTAT registers to their previous values. So, if the ISR enables any of the MSTAT mode bits, the RTI operation automatically disables them.

PUSH/POP PC/STS. You can explicitly push and pop the PC and status stacks as needed. The DSP automatically performs these operations when using nested interrupts.

Pushing (PUSH STS) and popping (POP STS) the status stack automatically saves or restores the ASTAT and MSTAT registers. But, pushing (PUSH PC) and popping (POP PC) the PC stack requires a few more steps that involve the STACKA and STACKP registers.

For PUSH/POP PC operations, the 16-bit STACKA register supplies or receives, respectively, the 16 LSBs of an instruction's 24-bit address, and the 8-bit STACKP register supplies or receives the eight MSBs. Before issuing a PUSH PC instruction, load the STACKA and STACKP registers with the appropriate values:

```
STACKA = 0x3521;
STACKP = 0x02;
PUSH PC;
```

Likewise, after popping the PC stack, you can check the contents of the STACKA and STACKP registers:

```
POP PC;
AX0 = STACKA;
AY0 = STACKP;
```

A PUSH PC or POP PC instruction has one cycle of latency for all SSTAT register bits, but a PUSH/POP LOOP or STS has one cycle of latency only for the STKOVERFLOW bit in the SSTAT register.

Loop Stacks Operation

Applications use this stack to implement loop operations.

When a DO UNTIL instruction executes, it automatically pushes data onto the three loop stacks:

- Loop begin stack. Receives the loop start address (current PC).
- Loop end stack. Receives the loop end address.
- Counter stack. Receives the counter value from the CNTR register for finite loops. If the loop is infinite, the counter stack receives the counter value and the DSP decrements this value on the stack, but ignores the result.

Finite Loops (DO <loop> UNTIL CE). The CE terminator specifies a finite loop. There is an effect in the number of cycles executed, depending on the number of instructions in the loop. A minimum of three instructions after the DO UNTIL instruction are needed to avoid this added latency. When the DO UNTIL instruction executes, it automatically pushes the CNTR value onto the counter stack. (The CNTR register retains the original value, until explicitly changed with a data move or POP LOOP instruction.)

The loop mechanism decrements and tests the value at the top of the counter stack at the end of each pass through the loop. The loop ends when the counter expires (decrements to 1). At loop end, program execution automatically continues with the instruction directly following the end of the loop.

Infinite Loops (DO <loop> [UNTIL FOREVER]). To end an infinite loop, the loop must contain an explicit JUMP to an instruction outside the loop to exit and end it. The JUMP is based on a condition created inside the loop and typically branches to a POP LOOP instruction to recover the loop stacks—the goal is to adjust the loop stack pointers, not retrieve the loop start and end addresses. After that, another JUMP instruction returns program execution to the next sequential instruction following the loop's end.

PUSH/POP LOOP. You can explicitly push and pop the loop stacks. These operations are necessary to recover and maintain the loop stacks when you abort a loop.

PUSH/POP LOOP instructions operate on all three loop stacks in parallel. Both operations involve the STACKA, STACKP, LPSTACKA, LPSTACKP, and CNTR registers. Before issuing a PUSH LOOP instruction, load the STACKA, STACKP, LPSTACKA, and LPSTACKP registers with appropriate values.

- The 16-bit STACKA and 8-bit STACKP register supply or receive the loop start address from the loop begin stack.

  STACKA holds the 16 LSBs of the 24-bit, loop start address, and STACKP holds the eight MSBs.

- The 16-bit LPSTACKA and 16-bit LPSTACKP register supply or receive the loop end address from the loop end stack. (Only bit 15 and bits 7:0 of LPSTACKP are valid—bits 14:8 should always be zero.)

  LPSTACKA holds the 16 LSBs of the 24-bit, loop end address, and LPSTACKP holds the eight MSBs in bits 7:0 and the loop terminator condition, CE or FOREVER, in bit 15. When the FOREVER bit is set, the loop logic ignores the loop counter value.

- The 16-bit CNTR register supplies or receives the counter value from the counter stack.

  On a pop, the CNTR register receives the value at the top of the counter stack. For finite loops, since the value in the counter stack is decremented at the end of each pass through the loop, a POP loads CNTR with a new value, overwriting the original count value, unless the POP occurs before the first pass through the loop.

  For infinite loops, the PUSH LOOP instruction pushes the current value of the CNTR register onto the loop counter stack. Although this value is irrelevant, pushing it maintains the pointer's correct position in the loop counter stack.

A PUSH PC or POP PC instruction has one cycle of latency for all SSTAT register bits, but a PUSH/POP LOOP or STS has one cycle of latency only for the STKOVERFLOW bit in the SSTAT register.

Stack Status Flags

As shown in Table 1-7 on page 1-10, bits 0 through 7 of the SSTAT register record the status of the DSP's stacks. This information is useful for managing the stack and servicing stack interrupts.

The stack interrupt is always generated by a stack overflow condition, but can also be generated by ORing together the stack overflow status (STKOVERFLOW) bit and stack high/low level status (PCSTKLVL) bit. The level bit is set when:

- The PC stack is pushed and the resulting level is at the high-water mark.
- The PC stack is popped and the resulting level is at the low-water mark.

Spill-fill mode (using the stack to generate a stack interrupt) is disabled on reset. Two bits in the ICNTL register (bit 10 —PC stack interrupt enable) can be used to enable interrupts for the three corresponding stacks.

When switching on spill-fill mode, a spurious low-water mark interrupt may occur (depending on the level of the stack). In this case, the interrupt handler should push values on the stack to raise the level above the low-water mark level.

Interrupts

The DSP uses interrupts to communicate with the outside world. The DSP's core generates internal interrupts, the peripherals generate external interrupts, and software can generate software interrupts.

When an interrupt occurs, the DSP suspends its current operation, saving the ASTAT and MSTAT registers, and jumps to the location in memory of the interrupt's service routine (ISR) and begins executing that program code. When it has completed the interrupt's ISR, an RTI instruction at the end of the routine forces program flow to return to the suspended operation and continue executing code at the location where it left off, after the DSP restores the ASTAT and MSTAT registers.

Each interrupt has a priority rank and a vector address. The interrupt's vector address specifies the location in memory of its service routine. Its priority determines the order in which the interrupt is serviced relative to the other interrupts. An higher priority interrupt is serviced before one with lower priority. As shown in Table 1-5 on page 1-7, the lower the interrupt's position in the IMASK/IRPTL register, the higher its priority.

To implement and use interrupts, your software must perform these tasks:

- Globally enable interrupts.
- Individually enable the particular interrupt.
- At the beginning of the ISR, switch context to secondary registers and perform the necessary tasks to handle the interrupt condition. For details, see "Switching Contexts" on page 7-16.

  If your program requires nested interrupts, extra tasks within each interrupt's ISR may be required. For details, see "Nesting Interrupts" on page 7-16.

- At the end of the ISR, insert an RTI instruction to branch back (return) to the suspended operation. The RTI instruction automatically switches context back to the primary register sets.
- Continue executing program code at the return address.

Enabling Interrupts

When an interrupt occurs, the DSP services it only when all interrupts are globally enabled and the particular interrupt is individually enabled. Typically, you enable interrupts both globally and individually in your main program and at the appropriate place wait for an interrupt to occur.

Global Interrupts. Use these instructions to enable and disable interrupts globally:

```
ENA INT;             /* Enable interrupts globally */
DIS INT;             /* Disable interrupts globally */
```

With interrupts globally disabled, the DSP does not recognize or latch interrupts that occur and so cannot service them.

Individual Interrupts. You can enable and disable interrupts individually using the register load instruction (for details, see, “Direct Register Load” on page 6-27).

For example, to enable (unmask) interrupts 3, 5, 7, and 8, set them to 1, as follows:

```
IMASK = 0x01A8;  /* Enable interrupts 8, 7, 5, & 3 only */
```

Interrupt 0 is nonmaskable in IMASK and cannot be enabled or disabled globally.

When interrupts are globally enabled and individual interrupts are enabled in IMASK, the DSP services them automatically when it detects their respective bits set in IRPTL.

Interrupt latch bits of the IRPTL register can be controlled individually with the following instructions:

```
SETINT n;
CLRINT n;
```

Read-modify-write operations are not recommended on the IRPTL register.

When interrupts are globally enabled and individual interrupts are disabled in IMASK, when they occur and are latched in IRPTL, you can choose to unmask their respective bits in IRPTL and service them or to clear their bits and reject them. For example:

```
   ENA INT;                    /* globally enable ints */
   IMASK = 0x0000;             /* individually disable all ints */
   ...
   AX0 = IRPTL;
   AF = TSTBIT 8 OF AX0;
   IF EQ JUMP normal;          /* Note the “EQ” */
service:
   CLRINT 8;
   ...
normal:
   ...
```

IMASK is the interrupt mask register, and IRPTL is the interrupt latch register. As shown in Table 1-5 on page 1-7, the IMASK and the IRPTL registers match each other bit for bit.

Switching Contexts

The DSP has two sets of DAG address registers and two sets of data registers that enable you to quickly switch between the context of normal processing and the context of interrupt processing as needed. Secondary register sets eliminate the need to save the state of the data and address registers before processing an interrupt and reduces interrupt latency. (For details on DSP modes, see "Mode Control" on page 7-4.)

Typically, you switch from primary to secondary registers at the beginning of the interrupt's ISR. To do so, you use the following instructions:

```
ENA SEC_REG;     /* enable secondary data registers */
ENA SEC_DAG;     /* enable secondary DAG address registers */
```

Use the RTI instruction at the end of the routine to return program flow to the main program. This instruction automatically switches context back to the primary registers when it restores the ASTAT, MSTAT, and SSTAT registers.

An interrupt service routine might look like this:

```
service_interrupt:
    ENA SEC_REG, ENA SEC_DAG;  /* enable secondary registers */
    NOP;
    NOP;                        /* ISR code */
    NOP;
    RTI;                        /* return from interrupt and */
                                /* enable primary registers */
```

Nesting Interrupts

Nested interrupts enable the DSP to respond to more than one interrupt at a time. A higher priority interrupt suspends a lower priority interrupt's routine. After the higher priority interrupt's RTI executes, the lower priority interrupt's routine continues to execute.

Without nested interrupts, one interrupt at a time gets serviced, so other interrupts remain pending until the RTI of the current routine executes. Then, the pending interrupt with highest priority is serviced.

To use nested interrupts, enable them in the ICNTL register. To do so, explicitly set bit 4 of INCTL:

```
ICNTL = 0x0010;
```

Once enabled, an interrupt with higher priority than the currently executing ISR suspends that ISR's execution. Table 1-4 on page 1-6 lists and describes the bits of the ICNTL register.

The DSP supports up to 16 nested interrupts, but has one set of secondary data and DAG address registers only. If your application uses deeply nested interrupts, you may need to manually save the state of the data registers and DAG address registers to memory in your ISR routines.

To do so:

- Set up a segment in memory to save the current state of the data and DAG address registers.
- In the ISR, save to memory the state of the data registers and the state of the DAG address registers that you intend to use.

Application Performance

The ADSP-219x's instruction set provides many ways to optimize code to accommodate particular applications. This section discusses optimization strategies for these topics:

- Exiting a loop
- Using long jumps and calls
- Effect latencies

Exiting a Loop

When you exit an infinite loop or abort a finite loop prematurely, the loop hardware fixes and restores the loop stacks before the POP LOOP instruction executes. With few restrictions, you can branch out of a loop from almost any location, regardless of the length of the loop. For optimal performance, consider these scenarios:

- JUMPs or CALLs near loop ends may add extra cycles of loop stack clean-up when the jump or call is taken.

```
CNTR = 5;
MX0 = 0xFF;
MY0 = 0xFF;
DO mac_loop UNTIL CE;        /* start of mac_loop */
   NOP;
   NOP;
   MR = MX0 * MY0 (SS);
   IF MV JUMP abort_loop;
mac_loop:
NOP;                         /* end of mac_loop */
NOP;                         /* 1st instr after mac_loop */
NOP;                         /* 2nd instr after mac_loop */
NOP;                         /* 3rd instr after mac_loop */
abort_loop:                  /* loop exit routine */
   POP LOOP;
   JUMP mac_loop + 1;
```

  The jump to abort_loop takes 1, 2, or 3 extra cycles, depending on whether the first, second, and third instruction after the end of the mac_loop are also loop ends, to clean up the loop stacks before the POP LOOP instruction executes. Impact on performance is minimal if the POP occurs only once.

- Jumps or calls near loop ends add 1, 2, or 3 extra cycles each time the branch is taken.

```
CNTR = 5;
DO little_loop UNTIL CE;
   NOP;            /* 1st instr. of little_loop */
   NOP;            /* 2nd instr. of little_loop */
   NOP;            /* 3rd instr. of little_loop */
   IF MV CALL fix_my_data;
little_loop:
   NOP;           /* end of little_loop */
   NOP;           /* 1st instr. after little_loop */
   NOP;           /* 2nd instr. after little_loop */
   NOP;           /* 3rd instr. after little_loop */
   NOP;           /* 4th instr. after little_loop */

fix_my_data:
   NOP;
   NOP;
   RTS;
```

The call to `fix_my_data` takes 1, 2, or 3 extra cycles, depending on whether the first, second, and third instructions after the end of `little_loop` are also loop ends, to clean up the loop stacks. To avoid the degradation in performance caused by this construct, move the `CALL` instruction further up in the loop or insert the called subroutine in the loop.

- Because the loop begin stack and PC stack are separate and distinct, this loop construct causes a loop to fall gracefully through the next loop end.

```
CNTR = 5;
DO bigger_loop UNTIL CE;
   NOP;            /* 1st instr. of bigger_loop */
   NOP;            /* 2nd instr. of bigger_loop */
```

```
    NOP;                /* 3rd instr. of bigger_loop */
    IF MV CALL fix_bigger_data;
    NOP;
    NOP;
bigger_loop:
    NOP;                /* end of bigger_loop */
    NOP;                /* 1st instr. after bigger_loop */
    NOP;                /* 2nd instr. after bigger_loop */
    NOP;                /* 3rd instr. after bigger_loop */

fix_bigger_data:
    NOP;
    NOP;
    RTS;
```

Unless the loop is aborted, the call to `fix_bigger_data` takes no extra cycles. One abort routine can serve all loops in nearby code space since the routine is identical for each. Even after the loop is aborted, the end of the `bigger_loop` still executes, and the loop falls out gracefully.

Using Long Jumps and Calls

The instruction set provides several jump/call instructions that support different address ranges for addressing branch targets:

- `-4096` to `+4095` "Direct JUMP (PC Relative)" on page 7-29
- `-32768` to `+32767` "JUMP (PC Relative)" on page 7-37
- `-32768` to `+32767` "CALL (PC Relative)" on page 7-33
- `16777216` to `+16777215` "Long Call (LCALL)" on page 7-40
- `-16777216` to `+16777215`"Long Jump (LJUMP)" on page 7-43

Usually, programmers must determine in advance the offset of the target from the branch and use the appropriate branch instruction, ensuring that the address of the branch target falls within the address range of the branch instruction.

However, by combining an option provided in the assembler and in the linker with any of the PC relative branch instructions, you can allow the development tools to determine and select which branch instruction to use based on the actual address of the branch target. To do so, encode PC relative branch instructions and use the assembler's and linker's `-jcs21` option, which directs the tools to substitute, during linking, `LJUMP or LCALL` for any particular PC relative branch instruction as appropriate. For details, see the *VisualDSP++ Assembler and Preprocessor Manual for 16-Bit Processors* and the *VisualDSP++ Linker and Utilities Manual for 16-Bit Processors*.

When using the linker's `-jcs21` option, you need to understand how it alters the linker's operation to fine-tune your code accordingly. When the linker substitutes `LJUMP or LCALL` for a corresponding PC relative branch instruction:

- It substitutes an absolute address for the PC relative address.
- If it encounters the (`DB`) option in a PC relative instruction, it moves the 48 bits (either two one-word instructions or one two-word instruction) from the two delay slots following the PC relative instruction and inserts them directly in front of the `LCALL` or `LJUMP` instruction.

  For conditional PC relative instructions, this procedure could change the condition code upon which the branch instruction is predicated. To avoid this potential bug, base (`DB`) branch instructions on negated conditions (`IF NOT COND`), not positive conditions (`IF COND`).
- For unconditional PC relative instructions, it always encodes the `TRUE` condition.

Effect Latencies

An effect latency occurs when instructions write or load a value into a register, changing the value of one or more bits in the register. Effect latency refers to the time it takes after the write or load instruction for the effect of the new value to become available for other instructions to use. For more information, see "Register Load Latencies" on page 6-8.

Program Flow Instruction Reference

Program flow control instructions include:

- "DO UNTIL (PC Relative)" on page 7-24
- "Direct JUMP (PC Relative)" on page 7-29
- "CALL (PC Relative)" on page 7-33
- "JUMP (PC Relative)" on page 7-37
- "Long Call (LCALL)" on page 7-40
- "Long Jump (LJUMP)" on page 7-43
- "Indirect CALL" on page 7-46
- "Indirect JUMP" on page 7-50
- "Return from Interrupt (RTI)" on page 7-53
- "Return from Subroutine (RTS)" on page 7-57
- "PUSH or POP Stacks" on page 7-61
- "FLUSH CACHE" on page 7-67
- "Set Interrupt (SETINT)" on page 7-69
- "Clear Interrupt (CLRINT)" on page 7-71
- "NOP" on page 7-73
- "IDLE" on page 7-74
- "Mode Control" on page 7-76

DO UNTIL (PC Relative)

```
DO <Imm12> [UNTIL <Term>] ;
```

Function

Sets up the looping circuitry for zero-overhead looping. The DO UNTIL instruction uses the current PC (Program Counter) as the basis for determining the beginning of the loop and the PC-relative 12-bit offset value (Imm12) as the end of a loop.

Input

Imm12	12-bit, positive offset value added to the address (PC) of the DO UNTIL instruction. Valid values range from 1 to 4095. (For good programming practice, use declared labels.)
Term	Loop terminator. Valid loop termination conditions are FOREVER and CE.

Output

None.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
LPSTKEMPTY (always cleared), LPSTK-FULL, STKOVERFLOW	PCSTKEMPTY, PCSTKFULL, PCSTKLVL, STSSTKEMPTY
For information on these status bits in the SSTAT register, see Table 1-2 on page 1-4.	

Details

The loop begins at the instruction directly following the DO UNTIL instruction (PC + 1) and ends at the instruction located at the offset address specified in the DO UNTIL instruction—(PC + <imm12>).

When using the FOREVER (infinite loop) termination condition, explicitly exit the loop by generating a status condition on which to base a jump to a location outside the loop. If you omit a terminator (DO <loop>), the instruction defaults to FOREVER.

When using the CE (counter expired) termination condition, before entering the loop, load the CNTR register with the number of times to execute the loop. Each pass through the loop decrements and tests the counter value in the loop counter stack, not the CNTR register (for details, see "Loop Stacks Operation" on page 7-10). When the counter expires, looping terminates.

If using CE termination, you must load a value >1 in the CNTR register.

Finite loops (CE terminator) execute repeatedly until the loop terminator occurs. Infinite loops (FOREVER) execute repeatedly until a condition occurs that invokes an explicit jump to the address of an instruction outside the loop.

At execution, the DO UNTIL instruction pushes:

- The address of the loop start instruction (PC + 1) onto the loop begin stack.
- The address of the loop end instruction (PC + <imm12>) and the code of the loop terminator onto the loop end stack.
- The contents of the CNTR register onto the loop counter stack.

During execution of a finite loop, the DSP tests and decrements the loop counter value stored in the loop counter stack—not the value in the CNTR register—at the end of each pass of the loop.

The CNTR register retains the original loop counter value until you load it with a new value, either explicitly with a load instruction or with a POP LOOP instruction.

To test the current value of the decrementing loop counter, pop the value off the loop counter stack into the CNTR register, move the CNTR contents into a data register, and then push the CNTR contents back onto the stack.

During execution of an infinite loop, the DSP pushes the current value of the CNTR register and the FOREVER bit onto the loop counter stack. When the FOREVER bit is set, the loop logic ignores the loop counter value. If you set up an infinite loop with the PUSH LOOP instruction instead of the DO UNTIL instruction, you must set the FOREVER bit of LPSTACKP (bit 15). (For details, see "Loop Stacks Operation" on page 7-10 and "PUSH or POP Stacks" on page 7-61.)

You can nest up to eight loops because each of the loop stacks has eight locations. The DSP pushes the loop begin stack, loop end stack, and loop counter stack for each level of nesting.

Follow these guidelines when coding loops:

- For nested loops, set up a separate counter for each loop, and end each loop with a separate instruction.
- Do not use an RTI instruction or RTS instruction inside a loop.
- Do not use a PUSH or POP instruction in the last seven lines of a loop. Avoid using PUSH or POP instructions within loops.

- Do not use a CALL instruction in the last line of a loop because the return address then resides outside of the loop, which causes incorrect sequencing.
- A JUMP instruction may be used in the last line of an infinite loop.
- If you use a JUMP or CALL instruction to abort a loop, ensure that you handle the loop stacks properly (POP LOOP). POP LOOP pops each of the loop stacks automatically. For details, see "Stacks" on page 7-7 and "PUSH or POP Stacks" on page 7-61.

Examples

```
                     /* finite loop example */
CNTR = 0xF;
IOPG = 0x1;
SI = AX0;
DM(I0 += M0) = SI;
MR = 0, MX0 = DM(I0+=M0), MY0 = PM(I4+=M4);
DO a_finite_loop UNTIL CE;
   MR = MR+MX0*MY0(SS), MX0 = DM(I0+=M0), MY0=PM(I4+=M4);
   MR = MR+MX0*MY0(RND);
a_finite_loop:
   IO(0xFF) = MR1;

                     /* infinite loop example */
IOPG = 0x1;
I0 = 0x1000;
M0 = 1;
L0 = 0;
DO an_infinite_loop;
   AX0 = DM(I0+=M0);
   AR = AX0 + AY0;
   IF AV JUMP exit_an_infinite_loop;
an_infinite_loop:
```

```
   DM(I0 + 100) = AR;
NOP;             /* 1st instruction after an infinite loop */
NOP;
NOP;             /* any number of instructions */
NOP;
exit_an_infinite_loop:
   POP LOOP;
   JUMP an_infinite_loop +1;

                     /* nested loop example */
AX0 = DM(I0 += M0), AY0 = PM(I4 += M4);
CNTR = 10;
DO outer_nested_loop UNTIL CE;
   CNTR = 20;
   DO middle_nested_loop UNTIL CE;
      CNTR = 30;
      DO inner_nested_loop UNTIL CE;
         AR =AX0 + AY0, AX0=DM(I0 += M0), AY0=PM(I4 += M4);
      inner_nested_loop:
         DM(I2 += M2) = AR;
   middle_nested_loop:
      NOP;
outer_nested_loop:
   NOP;
```

See Also

- “Type 11: Do … Until” on page 8-30
- “Conditions” on page 7-2
- “Counter-Based Conditions” on page 7-2
- “Stacks” on page 7-7

Direct JUMP (PC Relative)

```
[IF COND] JUMP <Imm13> [(DB)] ;
```

Function

This branch instruction causes program execution to continue at the offset address specified in the instruction. The offset address is the sum of the PC of the JUMP instruction and the 13-bit immediate value supplied in the instruction (PC + <imm13>).

If execution is based on an optional condition, the JUMP instruction executes only if the condition evaluates true and a NOP operation executes if the condition evaluates false. Omitting the condition forces unconditional execution of the loop. For a list of valid conditions, see "Conditions" on page 7-2.

The branch can be immediate or delayed (using the optional (DB)). If immediate, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency of four NOP cycles.

If using the optional delayed branch ((DB)) syntax, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency of four cycles. The two instructions directly following the JUMP instruction execute in sequence during the first two latency cycles if the branch is taken. Even if the branch is not taken, the instructions occupying the two branch delay slots still execute.

Input

`Imm13`	A 13-bit, twos complement offset value added to the address (`PC`) of the `JUMP` instruction. Valid values range from –`4096` to +`4095`. For good programming practice, always use a label, rather than a numeric value, since a label is relocatable.

Output

None.

Status Flags

None.

Details

When using the `(DB)` option, do not insert the following instructions after the `JUMP` instruction, in the two delayed branch slots:

- Stack manipulation instructions—`PUSH/POP`
- Branch instructions—`JUMP`, `CALL`, `RTI`, `RTS`
- Loop instruction—`DO UNTIL`

You can use the indirect 16-bit memory write—immediate data instruction. For details, see "Indirect 16-Bit Memory Write—Immediate Data" on page 6-56. Because it is a double-word instruction, you must place it in the first delay branch slot, right after the `CALL` instruction.

The number of cycles required to perform a `JUMP` operation depends on whether the branch is taken or not. With the immediate branch option, when the branch is taken, the DSP flushes the instruction pipeline except for the `JUMP` instruction and inserts four `NOP` cycles. As shown in Table 7-1

on page 7-31, when you use the (DB) option, the operation still takes five cycles (JUMP instruction + four cycles of latency), but the DSP executes in sequence the two instructions following the JUMP instruction, flushing only the top of the instruction pipeline.

Table 7-1. Branch (JUMP) Execution Cycles

Branch Case	Time to Execute	Delayed Branch Fills	Delayed Branch NOPs
Taken	5 cycles	2 cycles	2 cycles
Not Taken	1 cycle	0 cycles	0 cycles

If the address range of this instruction is inadequate, you can use the LJUMP instruction, which loses use of the (DB) option, or you can retain the (DB) option and allow the development tools to determine during assembly/linking whether to use this instruction or substitute the LJUMP instruction. For details see “Using Long Jumps and Calls” on page 7-20.

Examples

```
   JUMP first_branch_target;   /* immediate branch jump */
   NOP;                        /* any number of instructions */
first_branch_target:
   NOP;                        /* any number of instructions */
   NOP;
   Jump second_branch_target (DB);   /* delayed branch jump */
   AR = PASS 0;
   AR = AX0 + AY0;       /* these 2 instr. after jump execute */
   NOP;                  /* any number of instructions */
   NOP;
second_branch_target:
   NOP;
   NOP;                  /* any number of instructions */
   IF NE JUMP third_branch_target (DB);
      MR = 0;        /* these 2 instr. after (DB) jump execute */
```

```
      AR = PASS 0;   /* whether or not cond branch is taken */
    NOP;
    NOP;                    /* any number of instructions */
  third_branch_target:
    NOP;
    NOP;                    /* any number of instructions */
```

See Also

- "Type 10: Direct Jump" on page 8-28
- "Branch Options" on page 7-4
- "Addressing Branch Targets" on page 7-6

CALL (PC Relative)

```
CALL <Imm16> [(DB)] ;
```

Function

This instruction causes the Program Sequencer to branch to the offset address specified in the instruction and execute the subroutine at that location. The offset address is the sum of the PC of the CALL instruction and the 16-bit immediate value supplied in the instruction (PC + <imm16>).

The branch can be immediate or delayed (using the optional ((DB)). If immediate, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency of four NOP cycles.

If using the optional delayed branch ((DB)) syntax, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency of four cycles. The two instructions directly following the CALL instruction execute in sequence during the first two latency cycles of the branch.

Input

imm16	16-bit, twos complement offset value added to the address (PC) of the CALL instruction or a declared label. Valid values range from –32768 to +32767. For good programming practice, always use a label, rather than a numeric value, since a label is relocatable.

Output

None.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
PCSTKFULL, PCSTKLVL, STKOVER-FLOW, PCSTKEMPTY	LPSTKEMPTY, LPSTKFULL, STSST-KEMPTY
For information on these status bits in the SSTAT register, see Table 1-2 on page 1-4.	

Details

Before branching, the program sequencer pushes onto the PC stack the return address of the next instruction to execute after returning from the called subroutine. The next instruction to execute is:

- Immediate `CALL`—The first instruction following the `CALL` instruction.
- Delayed `CALL`—The third instruction following the `CALL` instruction.

To return from the subroutine, you must explicitly issue an `RTS` instruction. For details, see "Return from Subroutine (RTS)" on page 7-57.

When using the `(DB)` option, do not insert the following instructions after the `CALL` instruction, in the two delay branch slots:

- Stack manipulation instructions—`PUSH/POP`
- Branch instructions—`JUMP`, `CALL`, `RTI`, `RTS`
- Loop instruction—`DO UNTIL`

You can use the indirect 16-bit memory write—immediate data instruction. For details, see "Indirect 16-Bit Memory Write—Immediate Data" on page 6-56. Because it is a double-word instruction, you must place it in the first delay branch slot, right after the `CALL` instruction.

The number of cycles required to perform a CALL operation depends on whether the branch is taken or not. With the immediate branch option, when the branch is taken, the DSP flushes the instruction pipeline except for the CALL instruction and inserts four NOP cycles. As shown in Table 7-2, when you use the (DB) option, the operation still takes five cycles (CALL instruction + four cycles of latency), but the DSP executes in sequence the two instructions following the CALL instruction, flushing only the top of the instruction pipeline.

Table 7-2. Branch (CALL) Execution Cycles

Branch Case	Time to Execute	Delayed Branch Fills	Delayed Branch NOPs
Taken	5 cycles	2 cycles	2 cycles
Not Taken	1 cycle	0 cycles	0 cycles

If the address range of this instruction is inadequate, you can use the LCALL instruction, losing use of the (DB) option, or you can retain the (DB) option and let the development tools to determine during assembly/linking whether to use this instruction or substitute the LCALL instruction. For details see "Using Long Jumps and Calls" on page 7-20 and "Long Call (LCALL)" on page 7-40.

Examples

```
CALL data_shift_subroutine (DB);
   AX0 = DM(I0 += M1), AY0 = PM(I4 += M5);   /* these two instr.
*/
   AR = PASS 0;           /* execute before (DB) branch starts */
DM(I1 += M1) = SR0;       /* RTS returns here */
NOP;
NOP;                      /* any number of instructions */
NOP;
data_shift_subroutine:
   AR = AX0 + AY0;
```

```
AX1 = 3; SE = AR;
SR = ASHIFT SI (HI);
RTS;                    /* returns operation */
```

See Also

- “Type 10: Direct Jump” on page 8-28
- “Branch Options” on page 7-4
- “Addressing Branch Targets” on page 7-6

JUMP (PC Relative)

```
JUMP <Imm16> [(DB)] ;
```

Function

This branch instruction causes program execution to continue at the offset address specified in the instruction. The offset address is the sum of the PC of the JUMP instruction and the 16-bit immediate value supplied in the instruction (PC + <imm16>).

The branch can be immediate or delayed (using the optional ((DB)). If immediate, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency of four NOP cycles.

If using the optional delayed branch ((DB)) syntax, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency equal to four cycles. The two instructions directly following the JUMP instruction execute in sequence during the first two latency cycles of the branch.

Input

imm16	16-bit, twos complement offset value added to the address (PC) of the JUMP instruction or a declared label. Valid values range from −32768 to +32767. For good programming practice, always use a label, rather than a numeric value, since a label is relocatable.

Output

None.

Status Flags

None.

Details

When using the (DB) option, you cannot insert the following instructions in the two delay branch slots directly after the JUMP instruction:

- Stack manipulation instructions—PUSH/POP
- Branch instructions—JUMP, CALL, RTI, RTS
- Loop instruction—DO UNTIL

You can use the Indirect 16-bit Memory Write—Immediate Data instruction. For details, see "Indirect 16-Bit Memory Write—Immediate Data" on page 6-56. Because it is a double-word instruction, you must place it in the first delay branch slot, right after the CALL instruction.

The number of cycles required to perform a JUMP operation depends on whether the branch is taken or not. With the immediate branch option, when the branch is taken, the DSP flushes the instruction pipeline except for the JUMP instruction and inserts four NOP cycles. As shown in Table 7-1 on page 7-31, when you use the (DB) option, the operation still takes five cycles (JUMP instruction + four cycles of latency), but the DSP executes in sequence the two instructions following the JUMP instruction, flushing only the top of the instruction pipeline.

If the address range of this instruction is inadequate, you can use the LJUMP instruction, but you lose use of the (DB) option, or you can retain the (DB) option and let the tools determine during assembly/linking whether to use this instruction or substitute the LJUMP instruction. For details see "Using Long Jumps and Calls" on page 7-20 and "Long Jump (LJUMP)" on page 7-43.

Examples

```
.SECTION/PM seg_code;
   JUMP my_cod2_label;     /* jumps to 16-bit relative address */
   NOP;
   NOP;                    /* any number of instructions */
   NOP;
my_code_exit_label:
   NOP;                    /* jump from seg_cod2 comes here */
   NOP;
   NOP;                    /* any number of instructions */
   NOP;
.SECTION/PM seg_cod2;
my_cod2_label:
   NOP;
   NOP;                    /* any number of instructions */
   NOP;
   JUMP my_code_exit_label;
```

See Also

- "Type 10: Direct Jump" on page 8-28
- "Branch Options" on page 7-4
- "Addressing Branch Targets" on page 7-6

Long Call (LCALL)

```
[IF COND] LCALL <Imm24> ;
```

Function

This instruction causes the program sequencer to branch to the absolute address specified in the instruction and execute the subroutine at that location. The absolute address is the 24-bit immediate value supplied in the instruction. The 24-bit immediate value enables programs to access any location in program memory address space.

If execution is based on a condition, the `JUMP` instruction executes only if the condition evaluates true and a `NOP` operation executes if the condition evaluates false. Omitting the condition forces unconditional execution of the loop. For a list of valid conditions, see "Conditions" on page 7-2.

Input

`Imm24`	24-bit, twos complement value or a declared label. Values range from –16777216 to +16777215. Always use a label, rather than a numeric value, since a label is relocatable.

Output

None.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
PCSTKFULL, PCSTKLVL, STKOVERFLOW, PCSTKEMPTY	LPSTKEMPTY, LPSTKFULL, STSSTKEMPTY
For information on these status bits in the SSTAT register, see Table 1-2 on page 1-4.	

Details

For details on using the assembler's and linker's `-jcs21` option to direct the tools to determine when to replace PC relative `CALL` instructions with this instruction, see "Using Long Jumps and Calls" on page 7-20.

This is a double-word instruction, so it executes in six (2 instruction + 4 latency) cycles.

Before branching, the Program Sequencer automatically pushes onto the PC stack the return address of the next instruction to execute after returning from the called subroutine. The next instruction to execute is the first instruction following the `LCALL` instruction.

To return from the subroutine, you must explicitly issue an `RTS` instruction. For details, see "Return from Subroutine (RTS)" on page 7-57.

Examples

```
.SECTION/PM seg_code;
IF EQ LCALL my_faraway_routine;
   NOP;                        /* execution returns here */
   NOP;
   NOP;                        /* any number of instructions */
   NOP;

.SECTION/PM seg_cod2;
my_faraway_routine:
   NOP;
   NOP;                       /* any number of instructions */
   NOP;
   RTS;
```

See Also

- "Type 36: Long Jump/Call" on page 8-55
- "Branch Options" on page 7-4
- "Addressing Branch Targets" on page 7-6
- "Using Long Jumps and Calls" on page 7-20

Long Jump (LJUMP)

```
[IF COND] LJUMP <Imm24> ;
```

Function

This branch instruction causes program execution to continue at the absolute address specified in the instruction. The absolute address is the 24-bit immediate value supplied in the instruction. The 24-bit immediate value enables programs to access any location in program memory address space.

If execution is based on a condition, the JUMP instruction executes only if the condition evaluates true and a NOP operation executes if the condition evaluates false. Omitting the condition forces unconditional execution of the loop. For a list of valid conditions, see “Conditions” on page 7-2.

This instruction is a two-word instruction and requires (at minimum) six cycles to execute. For more information, see “Type 36: Long Jump/Call” on page 8-55.

Input

Imm24	24-bit, twos complement value or a declared variable. Values range from –16777216 to +16777215. For good programming practice, always use a label, rather than a numeric value, since a label is relocatable.

Output

None.

Status Flags

None.

Details

For details on using the ADSP-219x assembler's `-jcs21` (convert Jump/Call Short to Long) option to direct the tools to determine when to replace PC relative `JUMP` instructions with `LJUMP` instructions, see "Using Long Jumps and Calls" on page 7-20.

Examples

```
/* Long JUMP example nearby half */

.SECTION/PM seg_code;
.GLOBAL my_local_exit_label;
.EXTERN my_faraway_label;
   LJUMP my_faraway_label;   /* jumps to 24-bit relative addr */
   NOP;
   NOP;                      /* any number of instructions */
   NOP;
 my_local_exit_label:
   NOP;                      /* jump from seg_cod2 comes here */
   NOP;
   NOP;                      /* any number of instructions */

/* Long JUMP example faraway half */

.SECTION/PM seg_xpmc;
.GLOBAL my_faraway_label;
.EXTERN my_local_exit_label;
my_faraway_label:
   NOP;                      /* any number of instructions */
   NOP;
   LJUMP my_local_exit_label;
```

See Also

- "Type 36: Long Jump/Call" on page 8-55
- "Branch Options" on page 7-4
- "Addressing Branch Targets" on page 7-6
- "Using Long Jumps and Calls" on page 7-20

Indirect CALL

[IF COND] CALL (<Ireg>) [(DB)] ;

Function

This instruction causes the program sequencer to branch to the address pointed to by the DAG index register (Ireg). The Ireg supplies the 16 LSBs of the 24-bit address, and the IJPG register supplies the eight MSBs (page address) of the 24-bit address. You must explicitly load the IJPG register with the eight MSBs of the address before executing this instruction (for details, see "Data Move Instructions" on page 6-1).

If execution is based on a condition, the CALL instruction executes only if the condition evaluates true, and a NOP operation executes if the condition evaluates false. Omitting the condition forces unconditional execution of the loop. For a list of valid conditions, see "Conditions" on page 7-2.

The branch can be immediate or delayed (using the optional ((DB)). If immediate, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency of four NOP cycles.

If using the optional delayed branch ((DB)) syntax, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency equal to four cycles. The two instructions directly following the CALL instruction execute in sequence during the first two latency cycles if the branch is taken. Even if the branch is not taken, the instructions occupying the two branch delay slots still execute.

Input

Ireg	I0–I3 (DAG1 index registers) or I4–I7 (DAG2 index registers)

Output

None.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
PCSTKFULL, PCSTKLVL, STKOVERFLOW, PCSTKEMPTY	LPSTKEMPTY, LPSTKFULL, STSST-KEMPTY
For information on these status bits in the SSTAT register, see Table 1-2 on page 1-4.	

Details

Before branching, the program sequencer automatically pushes onto the PC stack the return address of the next instruction to execute after returning from the called subroutine. The next instruction to execute is:

Immediate CALL	The first instruction following the CALL instruction.
Delayed CALL	The third instruction following the CALL instruction.

To return from the subroutine, explicitly issue an RTS instruction. For details, see "Return from Subroutine (RTS)" on page 7-57.

When using the (DB) option, do not insert the following instructions after the CALL instruction, in the two delay branch slots:

- Stack manipulation instructions—PUSH/POP
- Branch instructions—JUMP, CALL, RTI, RTS
- Loop instruction—DO UNTIL

You can use the indirect 16-bit memory write—immediate data instruction (for details, see page 6-56), but because it is a double-word instruction, you must place it in the first delay branch slot, right after the CALL instruction.

The number of cycles required to perform a CALL operation depends on whether the branch is taken or not. With the immediate branch option, when the branch is taken, the DSP flushes the instruction pipeline except for the CALL instruction and inserts four NOP cycles. As shown in Table 7-2 on page 7-35, when you use the (DB) option, the operation still takes five cycles (CALL instruction + four cycles of latency), but the DSP executes in sequence the two instructions following the CALL instruction, flushing only the top of the instruction pipeline.

Examples

```
I5 = sampling_routine;
AR = AR + AX0;
IF EQ CALL (I5) (DB);
   DM(I0 += M1) = AR;           /* these two instr. execute */
   AR = 0;              /* whether or not the branch is taken */
AR = PASS 0;            /* RTS returns execution to this instr. */
NOP;
NOP;                    /* any number of instructions */
NOP;

sampling_routine:
   MX0 = DM(I0+=M1);
   MR = MX0 * MY0 (SS);
   NOP;
   NOP;                 /* any number of instructions */
   NOP;
   RTS;
```

See Also

- "Type 19: Indirect Jump/Call" on page 8-37
- "Branch Options" on page 7-4
- "Addressing Branch Targets" on page 7-6

Indirect JUMP

```
[IF COND] JUMP (<Ireg>) [(DB)] ;
```

Function

This branch instruction causes program execution to continue at the address pointed to by the DAG index register (Ireg). The Ireg supplies the 16 LSBs of the 24-bit address, and the IJPG register supplies the eight MSBs (page address) of the 24-bit address. You must explicitly load the IJPG register with the eight MSBs of the address before executing this instruction (for details, see "Data Move Instructions" on page 6-1).

If execution is based on a condition, the JUMP instruction executes only if the condition evaluates true, and a NOP operation executes if the condition evaluates false. Omitting the condition forces unconditional execution of the loop. For a list of valid conditions, see "Conditions" on page 7-2.

The branch can be immediate or delayed (using the optional ((DB)). If immediate, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency of four NOP cycles.

If using the optional delayed branch ((DB)) syntax, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency equal to four cycles. The two instructions directly following the JUMP instruction execute in sequence during the first two latency cycles if the branch is taken. Even if the branch is not taken, the instructions occupying the two branch delay slots still execute.

Input

Ireg	I0–I3 (DAG1 index registers) or I4–I7 (DAG2 index registers)

Output

None.

Status Flags

None.

Details

Loading the IJPG register or an Ireg has a zero (0) effect latency for this instruction, so the new value is available on the next instruction cycle.

When using the (DB) option, do not insert the following instructions after the JUMP instruction, in the two delay branch slots:

- Stack manipulation instructions—PUSH/POP
- Branch instructions—JUMP, CALL, RTI, RTS
- Loop instruction—DO UNTIL

You can use the indirect 16-bit memory write—immediate data instruction. For details, see "Indirect 16-Bit Memory Write—Immediate Data" on page 6-56. Because it is a double-word instruction, you must place it in the first delay branch slot, right after the JUMP instruction.

The number of cycles required to perform a JUMP operation depends on whether the branch is taken. With the immediate branch option, when the branch is taken, the DSP flushes the instruction pipeline except for the JUMP instruction and inserts four NOP cycles. As shown in Table 7-1 on page 7-31, when using the (DB) option, the operation still takes five cycles (JUMP instruction + four cycles of latency), but the DSP executes in sequence the two instructions following the JUMP instruction, flushing only the top of the instruction pipeline.

Examples

```
I4 = sampling;
I5 = next_sample;

sampling:
AR = AR + AX0;
IF EQ JUMP (I5) (DB);
   DM(I0 += M1) = AR;          /* these two instr. execute */
   AR = 0;              /* whether or not the branch is taken */
   JUMP (I4) (DB);
      AX0 = DM(I0 += M1);      /* these two instr. execute */
      AR = AX0;                /* before the branch starts */

next_sample:
   MX0 = DM(I0 += M1);
   MR = MX0 * MY0 (SS);
   NOP;
   NOP;                        /* any number of instructions */
   NOP;
   JUMP (I4);                  /* goes back to sampling */
```

See Also

- "Type 19: Indirect Jump/Call" on page 8-37
- "Branch Options" on page 7-4
- "Addressing Branch Targets" on page 7-6

Return from Interrupt (RTI)

```
[IF COND] RTI [(DB)] [(SS)] ;
```

Function

This instruction executes a return from an interrupt service routine (ISR). It returns program execution to the address of either the first or third instruction following the branch instruction that launched the ISR.

If execution is based on a condition, the `RTI` instruction executes only if the condition evaluates true, and a `NOP` operation executes if the condition evaluates false. Omitting the condition forces unconditional execution of the loop. For a list of valid conditions, see "Conditions" on page 7-2.

The branch can be immediate or delayed (using the optional (`(DB)`). If immediate, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency of four `NOP` cycles.

If using the optional delayed branch (`(DB)`) syntax, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency equal to four cycles. The two instructions directly following the `RTI` instruction execute in sequence during the first two latency cycles if the branch is taken. Even if the branch is not taken, the instructions occupying the two branch delay slots still execute.

For emulation, the `RTI` instruction supports an additional option, the single-step `(SS)` return interrupt. This option causes the instruction at the return address to generate an interrupt when it executes during emulation.

Input

None.

Output

None.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
PCSTKFULL, PCSTKEMPTY, PCSTKLVL	LPSTKEMPTY, LPSTKFULL, STKOVERFLOW, STSSTKEMPTY
For information on these status bits in the SSTAT register, see Table 1-2 on page 1-4.	

Details

This instruction pops and uses the address at top of the PC stack for the return address. It also pops the value at the top of the status stack and loads it into the Arithmetic Status (ASTAT) register and the Mode Status (MSTAT) register. If the ISR enabled secondary registers or changed other DSP modes in MSTAT, the RTI instruction disables them automatically when it executes.

Do not use an RTI instruction inside a loop without explicitly performing stack maintenance.

When using the (DB) option, do not insert the following instructions after the RTI instruction, in the two delay branch slots:

- Stack manipulation instructions—PUSH/POP
- Branch instructions—JUMP, CALL, RTI, RTS
- Loop instruction—DO UNTIL

You can use the indirect 16-bit memory write—immediate data instruction. For details, see "Indirect 16-Bit Memory Write—Immediate Data" on page 6-56. Because it is a double-word instruction, you must place it in the first delay branch slot, directly following the RTI instruction.

The number of cycles required to perform an RTI depends on whether the branch is taken. With the immediate branch option, when the branch is taken, the DSP flushes the instruction pipeline except for the RTI instruction and inserts four NOP cycles. As shown in Table 7-3, when you use the (DB) option, the operation still takes five cycles (RTI instruction + four cycles of latency), but the DSP executes in sequence the two instructions following the RTI instruction, flushing only the top of the instruction pipeline.

Table 7-3. Branch (RTI) Execution Cycles

Branch Case	Time to Execute	Delayed Branch Fills	Delayed Branch NOPs
Taken	5 cycles	2 cycles	2 cycles
Not Taken	2 cycle	0 cycles	1 cycle

Examples

```
interrupt_setup:
/* define addr. of inter. priority registers in IO() memory */
#define IPR0 0x203
#define IPR1 0x204
#define IPR2 0x205
#define IPR3 0x206
/* load interrupt priorities into IPR registers */
AX0 = 0x3210;
IO(IPR0) = AX0;       /* set priorities for peripherals 3-0 */
AX0 = 0x7654;
IO(IPR1) = AX0;       /* set priorities for peripherals 7-4 */
AX0 = 0xBA98;
IO(IPR2) = AX0;       /* set priorities for peripherals 11-8 */
AX0 = 0x0BBB;
IO(IPR3) = AX0;       /* set priorities for peripherals 14-12 */

ICNTL = 0x0010;       /* set GIE global interrupt enable bit */
```

```
IMASK = 0x4000;        /* unmask interrupt ID 14, which is
                       /* assigned to Timer Interrupt A by IPR2 */
ENA INT;               /* enable interrupts */

wait_here_for_interrupt:       /* loop waiting for interrupt */
   NOP;
   NOP;                /* any number of instructions */
   NOP;
   JUMP wait_here_for_interrupt;

.SECTION/PM irq_14;    /* map this ISR to addr. 0x01C0 with LDF */
timer_a_int:
   ENA SEC_REG, ENA SEC_DAG;
   NOP;
   NOP;                /* up to 32 instructions */
   NOP;
   RTI;
```

See Also

- "Type 20: Return" on page 8-38
- "Interrupts" on page 7-13
- "Set Interrupt (SETINT)" on page 7-69
- "Clear Interrupt (CLRINT)" on page 7-71

Return from Subroutine (RTS)

[IF COND] RTS [(DB)] ;

Function

This instruction executes a return from a subroutine. It returns program execution to the address of either the first or third instruction following the branch instruction that called the subroutine.

If execution is based on a condition, the `RTS` instruction executes only if the condition evaluates true, and a `NOP` operation executes if the condition evaluates false. Omitting the condition forces unconditional execution of the branch. For a list of valid conditions, see "Conditions" on page 7-2.

The branch can be immediate or delayed (using the optional (`(DB)`). If immediate, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency of four `NOP` cycles.

If using the optional delayed branch (`(DB)`) syntax, the branch instruction executes immediately, but the instruction at the offset address (branch target) executes after a latency of four cycles. The two instructions directly following the `RTS` instruction execute in sequence during the first two latency cycles if the branch is taken. Even if the branch is not taken, the instructions occupying the two branch delay slots still execute.

The two instructions following the `RTS(DB);` command are atomic. Interrupts are delayed until the two instruction are executed.

Do not use an `RTS` instruction inside a loop without explicitly performing stack maintenance. For details, see begin stack.

Input

None.

Output

None.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
PCSTKFULL, PCSTKEMPTY, PCSTKLVL	LPSTKEMPTY, LPSTKFULL, STKOVERFLOW, STSSTKEMPTY
For information on these status bits in the SSTAT register, see Table 1-2 on page 1-4.	

Details

This instruction pops and uses the address at top of the PC stack for the return address.

When using the (DB) option, do not insert the following instructions after the RTS instruction, in the two delay branch slots:

- Stack manipulation instructions—PUSH/POP
- Branch instructions—JUMP, CALL, RTI, RTS
- Loop instruction—DO UNTIL

You can use the indirect 16-bit memory write—immediate data instruction (for details, see page 6-56), but because it is a double-word instruction, you must place it in the first delay branch slot, right after the RTI instruction.

The number of cycles required to perform an RTS depends on whether the branch is taken. With the immediate branch option, when the branch is taken, the DSP flushes the instruction pipeline except for the RTS instruction and inserts four NOP cycles. As shown in Table 7-4 on page 7-59, when you use the (DB) option, the operation still takes five cycles

(RTS instruction + four cycles of latency), but the DSP executes in sequence the two instructions following the RTS instruction, flushing only the top of the instruction pipeline.

Table 7-4. Branch (RTS) Execution Cycles

Branch Case	Time to Execute	Delayed Branch Fills	Delayed Branch NOPs
Taken	5 cycles	2 cycles	2 cycles
Not Taken	1 cycle	0 cycles	0 cycle

Examples

```
I5 = sample_routine;
AR = AR + AX0;
IF EQ CALL (I5) (DB);
   DM(I0 += M1) = AR;  /* these two instr. execute */
   AR = 0;             /* whether or not the branch is taken */
AR = PASS 0;           /* RTS returns execution to this instr. */
NOP;
NOP;                   /* any number of instructions */
NOP;

sample_routine:
   MX0 = DM(I0+=M1);
   MR = MX0 * MY0 (SS);
   NOP;
   NOP;                /* any number of instructions */
   NOP;
   RTS;
```

See Also

- "Type 20: Return" on page 8-38
- "Branch Options" on page 7-4
- "Addressing Branch Targets" on page 7-6

PUSH or POP Stacks

PUSH POP	PC LOOP STS	;

Function

This instruction PUSHes (stores) or POPs (retrieves) a value from the top of the specified stack: PC, LOOP, or STS.

- PC

  On a PUSH, stores onto the top of the PC stack a 24-bit address value assembled from the STACKA and STACKP registers. STACKA provides the 16 LSBs of the address, and STACKP provides the eight MSBs of the address.

  On a POP, retrieves the most recently stacked 24-bit address value from the top of the PC stack into the STACKA and STACKP registers. STACKA receives the 16 LSBs of the address, and STACKP receives the eight MSBs of the address.

- LOOP

  On a PUSH, stores onto the top of the loop begin stack the 24-bit loop start address assembled from the STACKA and STACKP registers, pushes onto the top of the loop end stack the 24-bit loop end address assembled from the LPSTACKA and LPSTACKP registers, and pushes onto the top of the loop counter stack the current loop counter value from the CNTR register.

  On a POP, retrieves the most recently stacked 24-bit loop start address from the top of the loop begin stack into the STACKA and STACKP registers, pops the most recently stacked 24-bit loop end address from the top of the loop end stack into the LPSTACKA and LPSTACKP registers, and pops the current loop counter value from the top of the loop counter stack into the CNTR register.

- STS

  On a PUSH, stores the current values of the ASTAT and MSTAT registers onto the status stack. After each push, the status stack pointer increments by one to access the next available location in the stack.

  On a POP, retrieves the most recently stacked 16-bit value of the ASTAT and MSTAT registers from the top of the status stack. After each individual pop, the status stack pointer decrements by one to access the next lowest location (next register value) in the stack.

A PUSH or POP PC has one cycle of latency for all SSTAT register bits, but a PUSH or POP LOOP or STS has one cycle of latency only for the STKOVERFLOW bit in the SSTAT register.

Input

None.

Output

None.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
PCSTKEMPTY (affected on POP), PCSTKFULL, PCSTKLVL (affected on POP), LPSTKEMPTY, LPSTKFULL, STSSTKEMPTY (affected on POP), STKOVERFLOW	(none)
For information on these status bits in the SSTAT register, see Table 1-2 on page 1-4.	

Details (Push)

You can push up to two stacks in parallel by issuing two PUSH instructions on the same instruction line, pushing either:

```
PUSH PC,   PUSH STS;
PUSH LOOP, PUSH STS;
```

Do not push the PC and LOOP stacks in parallel (PUSH PC, PUSH LOOP;).

If you push the PC and loop stacks in parallel, you push the same address value onto both the PC stack and the loop begin stack. This occurs because STACKA and STACKP serve as the source registers for both stacks.

Regardless of the number of stacks pushed, this instruction always executes in a single cycle.

Subroutines, loops, and interrupts automatically push certain stacks:

- Calls to subroutines and entry into interrupt service routines automatically push the PC stack.
- Execution of the DO UNTIL instruction pushes the loop begin stack, the loop end stack, and the loop counter stack.

Do not use this instruction in either of the two slots directly following a delayed branch instruction.

Do not use this instruction inside a loop without explicitly performing stack maintenance. For details, see "PUSH or POP Stacks" on page 7-61

If you set up an infinite loop with the PUSH LOOP instruction, set bit 15 of LPSTACKP to indicate FOREVER. Although the LPSTACKP register has 16 bits, only bit 15 and bits 7:0 are valid. When the FOREVER bit is set (bit 15 = 1), the loop logic ignores the loop

counter value. When the FOREVER bit is cleared (bit 15 = 0), CE is the loop terminator condition, and the loop logic decrements the loop counter value.

Details (Pop)

You can pop up to two stacks in parallel by issuing two POP instructions on the same instruction line, popping either:

```
POP PC,   POP STS;
POP LOOP, POP STS;
```

Do not pop the PC and loop stacks in parallel (POP PC, POP LOOP;).

If you pop the PC and loop stacks in parallel, you lose the loop start address retrieved from the loop begin stack. This occurs because STACKA and STACKP serve as the destination registers for values popped from both the PC stack and the loop begin stack. In this case, STACKA and STACKP receive the most recently stacked PC value.

Regardless of the number of stacks popped, this instruction always executes in a single cycle.

Subroutines, loops, and interrupts automatically pop certain stacks:

- Upon exiting, RTS and RTI instructions automatically pop the PC stack.
- Loop termination automatically pops the loop begin stack, the loop end stack, and the loop counter stack

Do not use this instruction in either of the two slots directly following a delayed branch instruction.

Do not use this instruction inside a loop without explicitly performing stack maintenance. For details, see "PUSH or POP Stacks" on page 7-61.

Examples (PUSH)

```
/* Pushing an infinite loop—loop terminator */
/* condition = FOREVER: */

STACKA = 0x0045;
STACKP = 0x03;
LPSTACKA = 0x004C;
LPSTACKP = 0x03;
PUSH LOOP;

/* Saving the DSP's current state: */

STACKA = 0x0022;
STACKP = 0x05;
PUSH PC, PUSH STS;
```

Examples (POP)

```
/* Restoring the DSP's current state: */

POP PC, POP STS;
AR = TSTBIT 6 OF AX0;
IF EQ CALL primary;
AX1 = STACKA;
AY1 = STACKP;
IJPG = AY1;
primary: DIS SEC_DAG, DIS SEC_REG;
         RTS;

/* Aborting a loop: */

CNTR = 10;
MX0 = DM(I2 += M2),
MY0 = PM(I5 += M5);
```

```
DO mac UNTIL CE;
   MR = MR + MX0 * MY0 (SS),
   MX0 = DM(I2 += M2),
   MY0 = PM(I5 += M5);
   IF MV JUMP abort;
mac:
   DM(I0 += M1) = MR0;
NOP;
NOP;                /* any number of instructions */
NOP;
abort:
   POP LOOP;
   JUMP mac + 1;
```

See Also

- "Type 26:Push/Pop/Cache" on page 8-46
- "Mode Control" on page 7-4
- "Stacks" on page 7-7
- "PC and Status Stack Operation" on page 7-8
- "Loop Stacks Operation" on page 7-10

FLUSH CACHE

```
FLUSH CACHE ;
```

Function

This instruction flushes the instruction cache, invalidating all instructions currently cached, so the next instruction fetch results in a memory access.

Use this instruction when program memory changes to resynchronize the instruction cache with program memory.

Input

None.

Output

None.

Status Flags

None.

Details

This operation may require up to six cycles to take effect.

Do not use this instruction in either of the two slots directly following a delayed branch instruction.

Examples

```
FLUSH CACHE;
```

See Also

- "Type 26:Push/Pop/Cache" on page 8-46

Set Interrupt (SETINT)

```
SETINT n    ;
```

Function

This instruction sets bit *n* (`n = 1`) in the interrupt latch (`IRPTL`) register and its associated interrupt. If the specified interrupt is unmasked, its corresponding bit in the `IMASK` register is set, program flow immediately branches to and executes the interrupt's service routine. Otherwise, the interrupt request remains latched but ignored until the program unmasks it or clears it. If unmasked, the interrupt's ISR executes; if cleared, the interrupt request is rejected.

Input

`n`	Specifies which bit (and interrupt) in the `IRPTL` register to set. Valid values range from `0–15`. The mapping of bits to interrupts is specific to particular DSPs in the ADSP-219x family. For details, see the *ADSP-219x/2192 DSP Hardware Reference*.

Output

None.

Options

None.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
PCSTKFULL, PCSTKEMPTY, PCSTKLVL, STSSTKEMPTY, STKOVERFLOW	LPSTKEMPTY, LPSTKFULL
For information on these status bits in the SSTAT register, see Table 1-2 on page 1-4.	

Details

This instruction has no associated effect latency.

Examples

```
MR = MR+MX0*MY0(SS), MX0 = DM(I0+=M0), MY0 = PM(I4+=M4);
IF MV SAT MR;
IF LT JUMP adjust;

adjust: SETINT 12;
```

See Also

- “Stacks” on page 7-7
- “Interrupts” on page 7-13
- “Clear Interrupt (CLRINT)” on page 7-71
- “Type 37: Interrupt” on page 8-56

Clear Interrupt (CLRINT)

```
CLRINT n    ;
```

Function

This instruction clears bit *n* (`n = 0`) in the interrupt latch (`IRPTL`) register and its associated interrupt.

This instruction clears a pending interrupt. It is used, in an ISR for example, to clear a pending interrupt that has been detected, but not yet been serviced.

Input

`n`	Specifies which bit (and interrupt) in the `IRPTL` register to clear. Valid values range from `0–15`. The mapping of bits to interrupts is specific to particular DSPs in the ADSP-219x family. For details, see the *ADSP-219x/2192 DSP Hardware Reference*.

Output

None.

Options

None.

Status Flags

Affected Flags–set or cleared by the operation	Unaffected Flags
	PCSTKFULL, PCSTKEMPTY, PCSTKLVL, STSSTKEMPTY, STKOVERFLOW, LPSTKEMPTY, LPSTKFULL
For information on these status bits in the SSTAT register, see Table 1-2 on page 1-4.	

Details

This instruction has no associated latency.

Examples

```
AX0 = IRPTL;
AR = TSTBIT 12 of AX0;
IF EQ JUMP clear;

clear: CLRINT 12;
```

See Also

- "Stacks" on page 7-7
- "Interrupts" on page 7-13
- "Return from Interrupt (RTI)" on page 7-53
- "Set Interrupt (SETINT)" on page 7-69
- "Type 37: Interrupt" on page 8-56

NOP

```
NOP  ;
```

Function

This instruction causes the DSP's core to perform no operation for one cycle. Program execution continues with the instruction directly following the `NOP` instruction.

Input

None.

Output

None.

Status Flags

None.

Details

Only the core ceases operation for one cycle; the on-chip peripherals continue their respective operations.

Examples

```
NOP;    /* no operation */
```

See Also

- "Type 30: NOP" on page 8-48

IDLE

```
IDLE ;
```

```
IDLE 0 ;
```

Function

This instruction directs the DSP's core to wait indefinitely in a low-power state until an interrupt occurs. When an interrupt occurs, the DSP's core exits the low-power state, services the interrupt, and then continues program execution at the instruction directly following the `IDLE` instruction.

Input

None.

Output

None.

Status Flags

None.

Details

Applications typically use this instruction to wait for an interrupt:

```
standby: IDLE;
```

In idle mode, the DSP's response time to incoming interrupts is one cycle.

Some DSPs in the ADSP-219x family also support a *sleep mode*, in which the DSP's core and all of its on-chip peripherals enter idle mode. To invoke sleep mode, the application must program the appropriate bits in the PLL control and I/O clock control registers and use the standard `IDLE` instruction.

Examples

```
IDLE;     /* Idle at internal clock's rate */
```

See Also

- "NOP" on page 7-73
- "Type 31: Idle" on page 8-49

Mode Control

```
| ENA | | SEC_REG  | ;
| DIS | | BIT_REV  |
|     | | AV_LATCH |
|     | | AR_SAT   |
|     | | M_MODE   |
|     | | TIMER    |
|     | | SEC_DAG  |
|     | | INT      |
```

Function

This instruction enables (ENA) or disables (DIS) from one to seven DSP modes in parallel. To enable or disable a mode, this instruction sets (1) or clears (0), respectively, the mode's bit in the Mode Status, MSTAT register (refer to "Mode Control" on page 7-4). The DSP modes are:

- SEC_REG Secondary computation register bank (MSTAT[0]).
- BIT_REV Bit-reversed addressing mode (MSTAT[1]).
- AV_LATCH ALU overflow latch mode enable (MSTAT[2]).
- AR_SAT ALU AR register saturation mode (MSTAT[3]).
- M_MODE MAC integer operand format mode (MSTAT[4]).
- TIMER Timer enable (MSTAT[5]).
- SEC_DAG Secondary DAG address register bank (MSTAT[6]).
- INT Global interrupts

Input

SEC_REG, BIT_REV, AV_LATCH, AR_SAT, M_MODE, TIMER, SEC_DAG, INT

Output

None.

Status Flags

None.

Details

You can enable or disable one or more modes in parallel by issuing multiple DIS or ENA instructions on the same instruction line, as in:

```
ENA AR_SAT, ENA M_MODE, ENA AV_LATCH, ENA SEC_REG;
```

You cannot issue both DIS and ENA instructions on the same instruction line to enable and disable modes in parallel.

For example, the following is not allowed:
ENA AR_SAT, DIS AV_LATCH, ENA SEC_DAG;

As shown in Table 6-2 on page 6-9, changing modes using this instruction (as opposed to register writes or popping the status stack) does not incur any cycles of latency. Latency is the delay, in number of instruction cycles, between the time the mode change instruction executes and the time when the mode change takes effect, such that other instructions can execute operations based on the new value. A latency of 0 means that mode change is available to the instruction directly following the mode change instruction.

ENA INT or DIS INT sets or clears bit 5 in the ICNTL register. The write takes effect on the next instruction cycle.

Examples

```
/* Switching contexts during an ISR: */

ENA INT;
IMASK = 0x21A0;
ENA SEC_REG, ENA SEC_DAG;
AY0 = DM(I0 += M0);
RTI (DB);
   AR = AX0 + AY0;
   DM(I0 += M0) = AR;

/* Bit-reversing DAG1 output to memory: */

ENA BIT_REV;
AY0 = DM(I0 += M0);
AR = AX0 + AY0;
DM(I0 += M0) = AR;
DIS BIT_REV;
```

See Also

- "Type 18: Mode Change" on page 8-36
- "Mode Control" on page 7-4
- "Enabling Interrupts" on page 7-14
- "Effect Latencies" on page 7-22

8 INSTRUCTION OPCODES

This chapter lists and describes the opcodes that defines each of the instructions in the ADSP-219x's instruction set. This information is useful for debugging programs.

This chapter covers the following topics:

- "Opcode Mnemonics" on page 8-1
- "Opcode Definitions" on page 8-16

Opcode Mnemonics

This section lists, describes, and gives the numeric value for each opcode mnemonic. See Table 8-1.

Table 8-1. Opcode Mnemonics

Mnemonic	Description	Details
AMF	Specifies an ALU or multiplier operation.	on page 8-6
AS	Specifies whether ALU saturation mode is 0 = disabled 1 = enabled	on page 8-36
B	Specifies whether branch is 0 = immediate 1 = delayed	on page 8-23 on page 8-36 on page 8-38
BIT	Specifies which interrupt to enable or disable (0–15).	on page 8-56

Table 8-1. Opcode Mnemonics (Cont'd)

Mnemonic	Description	Details
BO	Specifies whether the supplied 4-bit constant in a type 9 instruction is 01 = as is 11 = negated	on page 8-9 on page 8-23
BR	Specifies whether bit-reverse addressing on DAG1 is 0 = disabled 1 = enabled	on page 8-36
BSR	Specifies whether the secondary DAG address registers are 0 = disabled 1 = enabled	on page 8-36
C	Specifies whether a software interrupt is 0 = set 1 = cleared	on page 8-56
CC	Specifies the two LSBs of a 4-bit constant value in a type 9 instruction.	on page 8-9 on page 8-23
CF	Specifies whether to flush the instruction cache 0 = No flush 1 = flush	on page 8-46
COND	Specifies one of the condition codes on which to base execution of the instruction.	on page 8-8
D	Specifies the direction of a data move. 0 = read 1 = write	on page 8-18 on page 8-31 on page 8-47 on page 8-50 on page 8-53 on page 8-54
DD	Specifies a destination data register for a DM bus transfer. 00 = AX0 01 = AX1 10 = MX0 11 = MX1	on page 8-17
DDREG	Specifies a destination register for a register-to-register move operation.	on page 8-11

Table 8-1. Opcode Mnemonics (Cont'd)

Mnemonic	Description	Details
DREG	Specifies an unrestricted data register (REG0 only).	on page 8-11
DMI	Specifies a DAG index address register (I0–I3) for a DM bus transfer.	on page 8-14 on page 8-17
DMM	Specifies a DAG modify address register (M0–M3) for a DM bus transfer.	on page 8-14 on page 8-17
DRGP	Specifies a destination register group. 00 = REG0 01 = REG1 10 = REG2 11 = REG3	on page 8-35
DRL	Specifies two MSBs of DREG data register address.	on page 8-11
DRU	Specifies two LSBs of DREG data register address.	on page 8-11
Exponent	Specifies an 8-bit, twos complement shift value.	on page 8-33
G	Specifies a DAG register group. 0 = DAG1 1 = DAG2	on page 8-13
IREG/ MREG	Specifies DAG index and modify registers (I0–I7, M0–M7).	on page 8-15
I	Specifies DAG index register (I0–I7).	on page 8-13
Idle Value	Specifies a 4-bit value that defines an internal clock divisor.	on page 8-49
INT	Specifies whether interrupts are globally 0 = disabled 1 = enabled	on page 8-36
LPP	Specifies push/pop of the loop stacks. 0 = disabled 1 = enabled	on page 8-46
M	Specifies a DAG modify register.	on page 8-13
MM	Specifies whether MAC integer mode is 0 = disabled 1 = enabled	on page 8-36

Table 8-1. Opcode Mnemonics (Cont'd)

Mnemonic	Description	Details
MOD DATA	Specifies an 8-bit, two's-complement immediate data value.	on page 8-39 on page 8-47
MS	Specifies memory bus for a memory data transfer 0 = 16-bit DM bus 1 = 24-bit PM bus	on page 8-50
OL	Specifies whether ALU overflow mode is 0 = disabled 1 = enabled	on page 8-36
PD	Specifies a destination data register for a PM bus transfer. 00 = AY0 01 = AY1 10 = MY0 11 = MY1	on page 8-17
PMI	Specifies a DAG index address register (I4–I7) for a PM bus transfer.	on page 8-14
PMM	Specifies a DAG modify address register (M4–M7) for a PM bus transfer.	on page 8-14
PPP	Specifies push/pop of the PC stack. 0 = disabled 1 = enabled	on page 8-46
Q	Specifies the RTI mode. 0 = normal 1 = single-step	on page 8-38
R	Specifies a result register. 0 = MR register 1 = SR register	on page 8-45
REG	Specifies a core register of RGPx.	on page 8-11
REG1	Specifies a register group 1 register	on page 8-11 on page 8-21
REG2	Specifies a register group 2 register	on page 8-11 on page 8-21

Table 8-1. Opcode Mnemonics (Cont'd)

Mnemonic	Description	Details
REG3	Specifies a register group 3 register	on page 8-11 on page 8-51
RGP	Specifies a register group. 00 = REG0 01 = REG1 10 = REG2 11 = REG3.	on page 8-11
S	Specifies the branch type. 0 = jump 1 = call	on page 8-28 on page 8-37
SDREG	Specifies the source data register for a data move operation.	on page 8-11
SF	Specifies a shift function.	on page 8-12
SPP	Specifies push/pop of the status stack. 0 = disabled 1 = enabled	on page 8-46
SR	Specifies whether the secondary data registers are 0 = disabled 1 = enabled	on page 8-36
SRGP	Specifies a source register group for a data move operation. 00 = REG0 01 = REG1 10 = REG2 11 = REG3	on page 8-35
SWCD	Specifies a 4-bit nonfunctional value used by ADI tools only.	on page 8-48
T	Specifies the return type. 0 = RTS 1 = RTI	on page 8-38
TERM	Specifies the terminating condition for the type 11 instruction. 1110 = NOT CE 1111 = TRUE	on page 8-30
TI	Specifies whether the timer is 0 = disabled 1 = enabled	on page 8-36

Table 8-1. Opcode Mnemonics (Cont'd)

Mnemonic	Description	Details
U	Specifies whether the DAG index register is 0 = premodified with no update 1 = postmodified with update	on page 8-50
XOP	Specifies a restricted data register used to supply the x operand value in a multifunction or conditional instruction.	on page 8-15
XREG	Specifies the source register (REG0) in a shift function.	on page 8-11
Y0	Specifies whether the source of the y operand is 0 = data register 1 = 0 (explicit value)	on page 8-8
YOP	Specifies a restricted data register used to supply the y operand value in a multifunction or conditional instruction.	on page 8-15
YREG	Specifies the destination register (REG0) in a shift function.	on page 8-11 on page 8-8
YY	Specifies the two MSBs of a 4-bit constant value in a type 9 instruction.	on page 8-9 on page 8-23
Z	Specifies a result or feedback register 0 = result register 1 = feedback register	on page 8-19 on page 8-22 on page 8-23 on page 8-8

ALU or Multiplier Function (AMF) Codes

Table 8-2 lists the AMF codes used by these instruction types:

- "Type 1: Compute | DregX«···DM | DregY«···PM" on page 8-17
- "Type 4: Compute | Dreg «···» DM" on page 8-19
- "Type 8: Compute | Dreg1 «··· Dreg2" on page 8-22
- "Type 9: Compute" on page 8-23

Table 8-2. ALU/Multiplier Function (AMF) Codes

Code	Function	Description
Multiplier functions		
00000	NOP	No operation
00001	X * Y (RND)	Multiply
00010	MR + X * Y (RND)	Multiply and accumulate
00011	MR – X * Y (RND)	Multiply and subtract
00100	X * Y (SS)	Multiply
00101	X * Y (SU)	Multiply
00110	X * Y (US)	Multiply
00111	X * Y (UU)	Multiply
01000	MR + X * Y (SS)	Multiply and accumulate
01001	MR + X * Y (SU)	Multiply and accumulate
01010	MR + X * Y (US)	Multiply and accumulate
01011	MR + X * Y (UU)	Multiply and accumulate
01100	MR – X * Y (SS)	Multiply and subtract
01101	MR – X * Y (SU)	Multiply and subtract
01110	MR – X * Y (US)	Multiply and subtract
01111	MR – X * Y (UU)	Multiply and subtract
ALU functions		
10000	Y	PASS/CLEAR
10001	Y + 1	PASS
10010	X + Y + C	Add with carry
10011	X + Y	Add
10100	NOT Y	Negate
(RND) = round results; (SS) = both operands signed; (SU) = x operand signed, y operand unsigned; (US) =x operand unsigned, y operand signed; (UU) = both operands unsigned		

Table 8-2. ALU/Multiplier Function (AMF) Codes (Cont'd)

Code	Function	Description
10101	– Y	PASS
10110	X – Y + C – 1	Subtract (X–Y) with borrow
10111	X – Y	Subtract
11000	Y – 1	PASS
11001	Y – X	Subtract
11010	Y – X + C – 1	Subtract (Y–X) with borrow
11011	NOT X	Negate
11100	X AND Y	AND/test bit,clear bit
11101	X OR Y	OR/set bit
11110	X XOR Y	XOR/toggle bit
11111	ABS X	Absolute value
(RND) = round results; (SS) = both operands signed; (SU) = x operand signed, y operand unsigned; (US) =x operand unsigned, y operand signed; (UU) = both operands unsigned		

Condition Codes

Table 8-3 lists the condition codes used by these instruction types:

- "Type 9: Compute" on page 8-23
- "Type 10: Direct Jump" on page 8-28
- "Type 11: Do … Until" on page 8-30 uses `NOT CE` and `TRUE` only for the terminating condition.
- "Type 16: Shift Reg0" on page 8-34
- "Type 19: Indirect Jump/Call" on page 8-37

- "Type 20: Return" on page 8-38
- "Type 36: Long Jump/Call" on page 8-55

Table 8-3. Condition Codes

Code	Condition	Description
0000	EQ	Equal to 0 (= 0)
0001	NE	Not equal to 0 (≠ 0)
0010	GT	Greater than 0 (>0)
0011	LE	Less than or equal to 0 (≤0)
0100	LT	Less than 0 (<0)
0101	GE	Greater than or equal to 0 (≥0)
0110	AV	ALU overflow
0111	NOT AV	Not ALU overflow
1000	AC	ALU carry
1001	NOT AC	Not ALU carry
1010	SWCOND	CCODE register condition
1011	NOT SWCOND	Not CCODE register condition
1100	MV	MAC overflow
1101	NOT MV	Not MAC overflow
1110	NOT CE	Counter not expired
1111	TRUE	Always true

Constant Codes

Table 8-4 lists the valid constants used by "Type 9: Compute" on page 8-23. As shown, the `YY/CC` bits determine the constant value and the `BO` bits determine the sign of the value.

Table 8-4. Constants

Code		Decimal / Hex	Decimal / Hex
YY	**CC**	**BO = 01**	**BO = 11**
00	00	1 / 0x0001	–2 / 0xFFFE
00	01	2 / 0x0002	–3 / 0xFFFD
00	10	4 / 0x0004	–5 / 0xFFFB
00	11	8 / 0x0008	–9 / 0xFFF7
01	00	16 / 0x0010	–17 / 0xFFEF
01	01	32 / 0x0020	–33 / 0xFFDF
01	10	64 / 0x0040	–65 / 0xFFBF
01	11	128 / 0x0080	–129 / 0xFF7F
10	00	256 / 0x0100	–257 / 0xFEFF
10	01	512 / 0x0200	–513 / 0xFDFF
10	10	1024 / 0x0400	–1025 / 0xFBFF
10	11	2048 / 0x0800	–2049 / 0xF7FF
11	00	4096 / 0x1000	–4097 / 0xEFFF
11	01	8192 / 0x2000	–8193 / 0xDFFF
11	10	16384 / 0x4000	–16385 / 0xBFFF
11	11	–32768 / 0x8000	+32767 / 0x7FFF

Core Register Codes

Table 8-5 lists the core registers and their addresses. The complete address of any individual register is formed by appending the register's address bits to its RGP bits, so, for example, the address of the I2 register is 010010. The opcode mnemonics DREG, DDREG, SDREG, XREG, and YREG and the following instruction types reference these registers by their address bits:

- "Type 3: Dreg/Ireg/Mreg «···» DM/PM" on page 8-18
- "Type 4: Compute | Dreg «···» DM" on page 8-19
- "Type 6: Dreg «··· Data16" on page 8-20
- "Type 8: Compute | Dreg1 «··· Dreg2" on page 8-22
- "Type 9: Compute" on page 8-23
- "Type 12: Shift | Dreg «···» DM" on page 8-31
- "Type 14: Shift | Dreg1 «··· Dreg2" on page 8-32
- "Type 15: Shift Data8" on page 8-33
- "Type 16: Shift Reg0" on page 8-34
- "Type 17: Any Reg «···Any Reg" on page 8-35
- "Type 34: Dreg «···» IOreg" on page 8-53
- "Type 35: Dreg «···»Sreg" on page 8-54

Table 8-5. Core Registers

RGP/Address	Register Groups (RGP)			
Address	00 (REG0)	01 (REG1)	10 (REG2)	11 (REG3)
0000	AX0	I0	I4	ASTAT
0001	AX1	I1	I5	MSTAT

Table 8-5. Core Registers (Cont'd)

RGP/Address	Register Groups (RGP)			
Address	00 (REG0)	01 (REG1)	10 (REG2)	11 (REG3)
0010	MX0	I2	I6	SSTAT
0011	MX1	I3	I7	LPSTACKP
0100	AY0	M0	M4	CCODE
0101	AY1	M1	M5	SE
0110	MY0	M2	M6	SB
0111	MY1	M3	M7	PX
1000	MR2	L0	L4	DMPG1
1001	SR2	L1	L5	DMPG2
1010	AR	L2	L6	IOPG
1011	SI	L3	L7	IJPG
1100	MR1	IMASK	Reserved	Reserved
1101	SR1	IRPTL	Reserved	Reserved
1110	MR0	ICNTL	CNTR	Reserved
1111	SR0	STACKA	LPSTACKA	STACKP

Shift Function (SF) Codes

Table 8-6 lists the shift function (SF) codes used by these instruction types:

- "Type 12: Shift | Dreg «···» DM" on page 8-31
- "Type 14: Shift | Dreg1 «··· Dreg2" on page 8-32
- "Type 15: Shift Data8" on page 8-33—shift functions (codes `0000-0111`) only

- "Type 16: Shift Reg0" on page 8-34

Table 8-6. SF Codes

Code	Function
0000	LSHIFT (HI)
0001	LSHIFT (HI, OR)
0010	LSHIFT (LO)
0011	LSHIFT (LO, OR)
0100	ASHIFT (HI)
0101	ASHIFT (HI, OR)
0110	ASHIFT (LO)
0111	ASHIFT (LO, OR)
1000	NORM (HI)
1001	NORM (HI, OR)
1010	NORM (LO)
1011	NORM (LO, OR)
1100	EXP (HI)
1101	EXP (HIX)
1110	EXP (LO)
1111	Derive Block Exponent

Index Register and Modify Register Codes

Table 8-7 lists the DAG index and modify register codes used by the following instruction types. The `G` bit (`DAG1/DAG2`) determines which group of `I` (index) and `M` (modify) registers.

- "Type 4: Compute | Dreg «···» DM" on page 8-19
- "Type 12: Shift | Dreg «···» DM" on page 8-31

- "Type 19: Indirect Jump/Call" on page 8-37
- "Type 21: Modify DagI" on page 8-39
- "Type 21a: Modify DagI" on page 8-40
- "Type 22: DM «··· Data16" on page 8-41
- "Type 29: Dreg «···» DM" on page 8-47
- "Type 32: Any Reg «···» PM/DM" on page 8-50

Table 8-7. I and M Codes

	DAG1 (G=0)		DAG2 (G=1)	
Code	**I**	**M**	**I**	**M**
00	I0	M0	I4	M4
01	I1	M1	I5	M5
10	I2	M2	I6	M6
11	I3	M3	I7	M7

DMI, DMM, PMI, and PMM Codes

Table 8-8 lists the DAG index and modify register codes used by "Type 1: Compute | DregX«···DM | DregY«···PM" on page 8-17.

Table 8-8. DMI, DMM, PMI, and PMM Codes

Code	DMI	DMM	PMI	PMM
00	I0	M0	I4	M4
01	I1	M1	I5	M5
10	I2	M2	I6	M6
11	I3	M3	I7	M7

IREG/MREG Codes

Table 8-9 lists the Ireg and Mreg codes used by "Type 3: Dreg/Ireg/Mreg «···» DM/PM" on page 8-18 to specify a DAG index or modify register.

Table 8-9. Ireg and Mreg Codes

Code	Register	Code	Register
0000	I0	1000	M0
0001	I1	1001	M1
0010	I2	1010	M2
0011	I3	1011	M3
0100	I4	1100	M4
0101	I5	1101	M5
0110	I6	1110	M6
0111	I7	1111	M7

XOP and YOP Codes

Table 8-10 lists the XOP and YOP codes used by these instructions:

- "Type 1: Compute | DregX«···DM | DregY«···PM" on page 8-17
- "Type 4: Compute | Dreg «···» DM" on page 8-19
- "Type 8: Compute | Dreg1 «··· Dreg2" on page 8-22
- "Type 9: Compute" on page 8-23
- "Type 12: Shift | Dreg «···» DM" on page 8-31
- "Type 23: Divide primitive, DIVQ" on page 8-43
- "Type 24: Divide primitive, DIVS" on page 8-44

Table 8-10. XOP and YOP Codes

Code	XOP			Code	YOP	
	ALU	MAC	Shift		ALU	MAC
000	AX0	MX0	SI	00	AY0	MY0
001	AX1	MX1	SR2	01	AY1	MY1
010	AR	AR	AR	10	AF	SR1
011	MR0	MR0	MR0	11	0	0
100	MR1	MR1	MR1			
101	MR2	MR2	MR2			
110	SR0	SR0	SR0			
111	SR1	SR1	SR1			

Opcode Definitions

For each instruction opcode, this section provides the following information:

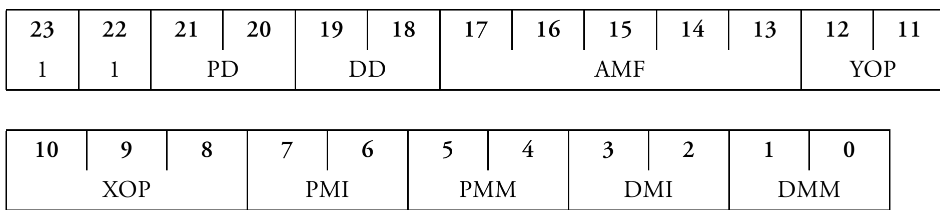
- Opcode bits
- Syntax
- See also (related instruction reference pages)

For mnemonics definitions, see “Opcode Mnemonics” on page 8-1.

Type 1: Compute | DregX«···DM | DregY«···PM

Multifunction ALU/MAC with DM and PM dual read with DAG1 and DAG2 postmodify

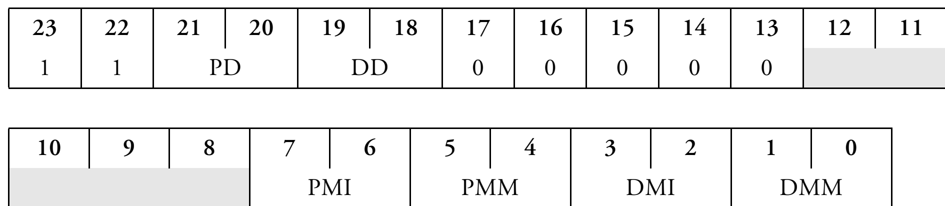
Opcode Bits



23	22	21	20	19	18	17	16	15	14	13	12	11
1	1	PD		DD		AMF					YOP	

10	9	8	7	6	5	4	3	2	1	0
XOP			PMI		PMM		DMI		DMM	

or for NOP only:



23	22	21	20	19	18	17	16	15	14	13	12	11
1	1	PD		DD		0	0	0	0	0		

10	9	8	7	6	5	4	3	2	1	0
			PMI		PMM		DMI		DMM	

Syntax

```
|<ALU>, <MAC>|, Xop = DM(Ia += Mb), Yop = PM(Ic += Md);
```

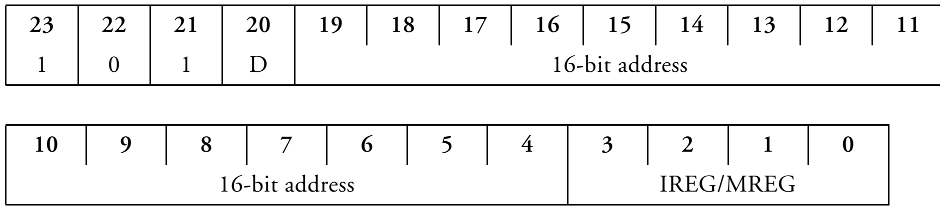
See Also

- “Opcode Mnemonics” on page 8-1
- “Compute with Dual Memory Read” on page 5-4
- “Dual Memory Read” on page 5-8

Type 3: Dreg/Ireg/Mreg «··» DM/PM

Register read/write to immediate 16-bit address

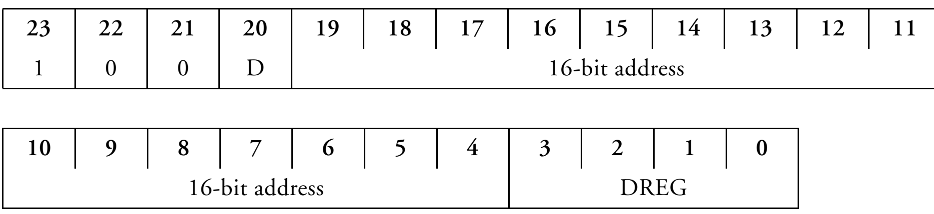
Opcode Bits



23	22	21	20	19	18	17	16	15	14	13	12	11
1	0	1	D	16-bit address								

10	9	8	7	6	5	4	3	2	1	0
16-bit address							IREG/MREG			

or:



23	22	21	20	19	18	17	16	15	14	13	12	11
1	0	0	D	16-bit address								

10	9	8	7	6	5	4	3	2	1	0
16-bit address							DREG			

Syntax

```
|DM(<Addr16>| = |Dreg, Ireg, Mreg|;
|Dreg, Ireg, Mreg| = |DM(<Addr16>)|;
```

See Also

- “Opcode Mnemonics” on page 8-1
- “Direct Memory Read/Write—Immediate Address” on page 6-24

Type 4: Compute | Dreg «··» DM

Multifunction ALU/MAC with memory read or write using DAG postmodify

23	22	21	20	19	18	17	16	15	14	13	12	11
0	1	1	G	D	Z	AMF					YOP	

Opcode Bits

10	9	8	7	6	5	4	3	2	1	0
XOP			DREG				I		M	

Syntax

```
|<ALU>, <MAC> |, Dreg = DM(Ia += Mb);
|<ALU>, <MAC> |, DM(Ia += Mb) = Dreg;
```

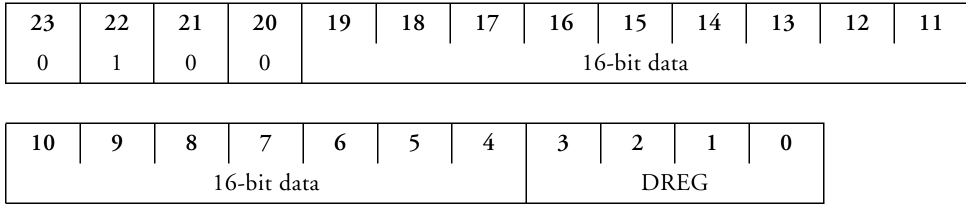
See Also

- "Opcode Mnemonics" on page 8-1
- "Compute with Dual Memory Read" on page 5-4
- "Compute with Memory Write" on page 5-15

Type 6: Dreg «··· Data16

Immediate register group 0 (`Dreg`) register load

Opcode Bits



23	22	21	20	19	18	17	16	15	14	13	12	11
0	1	0	0	16-bit data								

10	9	8	7	6	5	4	3	2	1	0
16-bit data							DREG			

Syntax

```
<Dreg> = <Data16>;
```

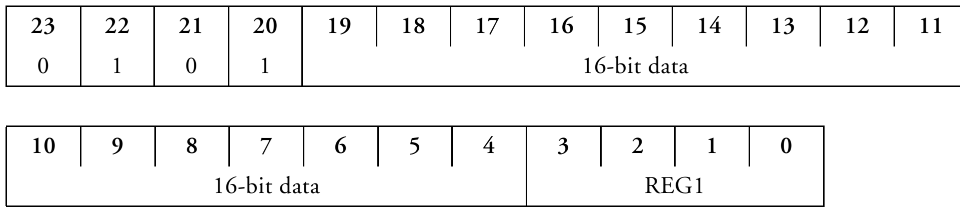
See Also

- "Opcode Mnemonics" on page 8-1
- "Direct Register Load" on page 6-27

Type 7: Reg1/2 «··· Data16

Immediate register group 1 or 2 (`Ireg`, `Mreg`, `Lreg`, `IMASK`, `IRPTL`, `ICNTL`, `CNTR`, `STACKA`, `LPCSTACKA`) register load

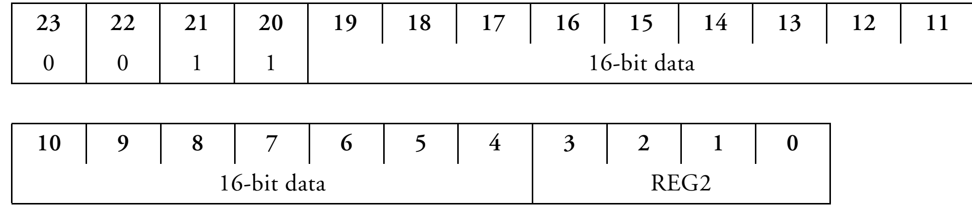
Opcode Bits



23	22	21	20	19	18	17	16	15	14	13	12	11
0	1	0	1	16-bit data								

10	9	8	7	6	5	4	3	2	1	0
16-bit data							REG1			

or:



23	22	21	20	19	18	17	16	15	14	13	12	11
0	0	1	1	16-bit data								

10	9	8	7	6	5	4	3	2	1	0
16-bit data							REG2			

Syntax

```
| <Reg1>, <Reg2> | = <Data16>;
```

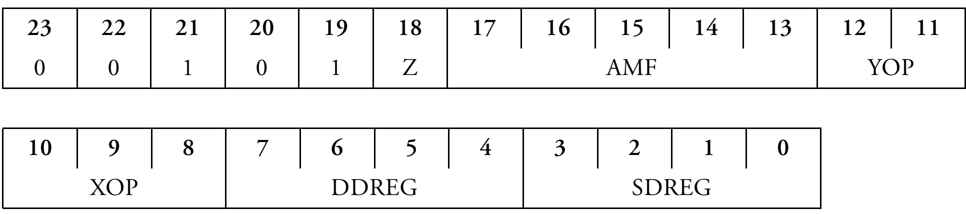
See Also

- “Opcode Mnemonics” on page 8-1
- “Direct Register Load” on page 6-27

Type 8: Compute | Dreg1 «··· Dreg2

ALU/MAC with data register move

Opcode Bits



23	22	21	20	19	18	17	16	15	14	13	12	11
0	0	1	0	1	Z	AMF					YOP	

10	9	8	7	6	5	4	3	2	1	0
XOP			DDREG				SDREG			

or, generate ALU/MAC status only

23	22	21	20	19	18	17	16	15	14	13	12	11
0	0	1	0	1	Z	AMF					YOP	

10	9	8	7	6	5	4	3	2	1	0
XOP			1	0	1	0	1	0	1	0

Syntax

```
| <ALU>, <MAC> |, Dreg = Dreg;
NONE = ALU (Xop, Yop);
```

See Also

- “Opcode Mnemonics” on page 8-1
- “Compute with Register-to-Register Move” on page 5-19
- “Generate ALU Status Only: NONE” on page 2-46

Type 9: Compute

Conditional ALU/MAC

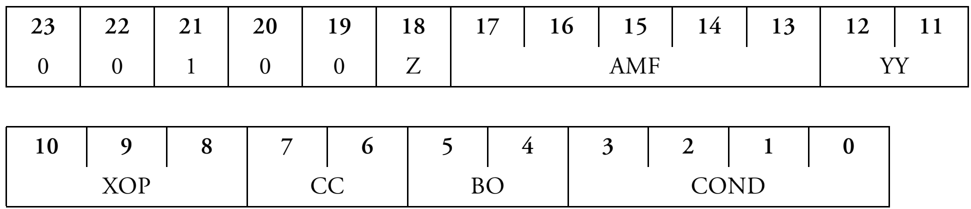
Opcode

Conditional ALU/MAC

23	22	21	20	19	18	17	16	15	14	13	12	11
0	0	1	0	0	Z	AMF					YOP	

10	9	8	7	6	5	4	3	2	1	0
XOP			0	0	0	0	COND			

Conditional ALU/MAC operations using constant YOP



23	22	21	20	19	18	17	16	15	14	13	12	11
0	0	1	0	0	Z	AMF					YY	

10	9	8	7	6	5	4	3	2	1	0
XOP			CC		BO		COND			

Conditional ALU/MAC operations with YOP=0

23	22	21	20	19	18	17	16	15	14	13	12	11
0	0	1	0	0	Z	AMF					1	1

10	9	8	7	6	5	4	3	2	1	0
XOP			0	0	0	0	COND			

Conditional MAC squaring operations only

23	22	21	20	19	18	17	16	15	14	13	12	11
0	0	1	0	0	Z	AMF					0	0

10	9	8	7	6	5	4	3	2	1	0
XOP			0	0	0	1	COND			

Syntax

```
[IF Cond] |AR, AF| = Xop + |Yop, Yop + C, C, Const, Const + C|;
[IF Cond] |AR, AF| = Xop – |Yop,Yop+C-1,+C-1,Const, Const+C-1|;
[IF Cond] |AR, AF| = Yop – |Xop, Xop+C-1|;
[IF Cond] |AR, AF| =  – |Xop+C-1, Xop+Const, Xop+Const+C-1|;
[IF Cond] |AR, AF| = Xop |AND, OR, XOR| |Yop, Const|;
[IF Cond] |AR, AF| = PASS |Xop, Yop, Const|;
[IF Cond] |AR, AF| = NOT |Xop, Yop|;
[IF Cond] |AR, AF| = ABS Xop;
[IF Cond] |AR, AF| = Yop +1;
[IF Cond] |AR, AF| = Yop –1;
[IF Cond] |MR, SR| = Xop * Yop [(|RND, SS, SU, US, UU|)];
[IF Cond] |MR, SR| = Yop * Xop [(|RND, SS, SU, US, UU|)];
[IF Cond] |MR, SR| = |MR, SR| + Xop * Yop [(|RND,SS,SU,US,UU|)];
[IF Cond] |MR, SR| = |MR, SR| + Yop * Xop [(|RND,SS,SU,US,UU|)];
[IF Cond] |MR, SR| = |MR, SR| – Xop * Yop [(|RND,SS,SU,US,UU|)];
[IF Cond] |MR, SR| = |MR, SR| – Yop * Xop [(|RND,SS,SU,US,UU|)];
[IF Cond] |MR, SR| = 0;
[IF Cond] MR = MR [(RND)];
[IF Cond] SR = SR [(RND)];
```

See Also

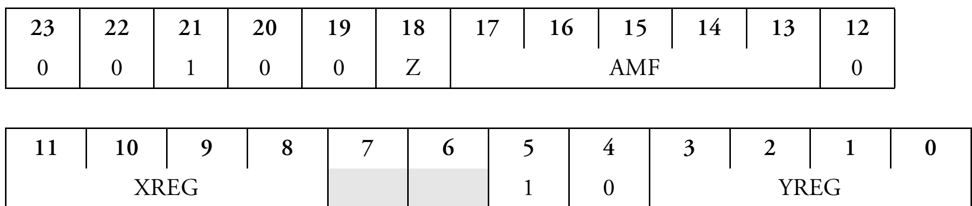
- "Opcode Mnemonics" on page 8-1
- "Add/Add with Carry" on page 2-5
- "Subtract X–Y/Subtract X–Y with Borrow" on page 2-9
- "Subtract Y–X/Subtract Y–X with Borrow" on page 2-13
- "Bitwise Logic: AND, OR, XOR" on page 2-16
- "Bit Manipulation: TSTBIT, SETBIT, CLRBIT, TGLBIT" on page 2-19
- "Clear: PASS" on page 2-22
- "Negate: NOT" on page 2-25
- "Absolute Value: ABS" on page 2-28
- "Increment" on page 2-31
- "Decrement" on page 2-34
- "Multiply" on page 3-8
- "Multiply with Cumulative Add" on page 3-11
- "Multiply with Cumulative Subtract" on page 3-14
- "MAC Clear" on page 3-17
- "MAC Round/Transfer" on page 3-19

Type 9a: Compute

Unconditional ALU/MAC

Opcode

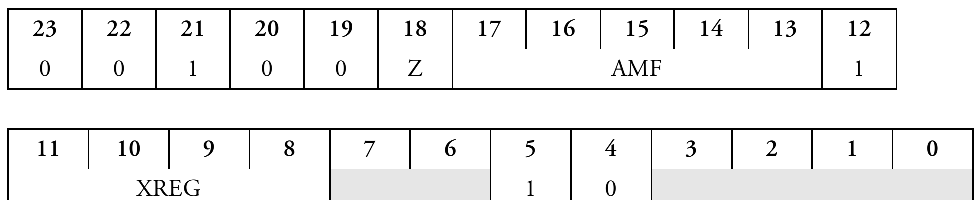
Register file ALU/MAC



23	22	21	20	19	18	17	16	15	14	13	12
0	0	1	0	0	Z	AMF					0

11	10	9	8	7	6	5	4	3	2	1	0
XREG						1	0	YREG			

Register file ALU/MAC with YREG=0



23	22	21	20	19	18	17	16	15	14	13	12
0	0	1	0	0	Z	AMF					1

11	10	9	8	7	6	5	4	3	2	1	0
XREG						1	0				

Syntax

```
|AR, AF| = Dreg1 + |Dreg2, Dreg2 + C, C |;
|AR, AF| = Dreg1 – |Dreg2, Dreg2 + C =1, +C –1|;
|AR, AF| = Dreg2 – |Dreg1, Dreg1 + C -1|;
|AR, AF| = Dreg1 |AND, OR, XOR| Dreg2;
|AR, AF| = PASS |Dreg1, Dreg2, Const|;
|AR, AF| = PASS 0;
|AR, AF| = NOT |Dreg|;
|AR, AF| = ABS Dreg;
|AR, AF| = Dreg +1;
|AR, AF| = Dreg –1;
```

```
|MR, SR| = Dreg1 * Dreg2 [(|RND, SS, SU, US, UU|)];
|MR, SR| = |MR, SR| + Dreg1 * Dreg2 [(|RND, SS, SU, US, UU|)];
|MR, SR| = |MR, SR| – Dreg1 * Dreg2 [(|RND, SS, SU, US, UU|)];
```

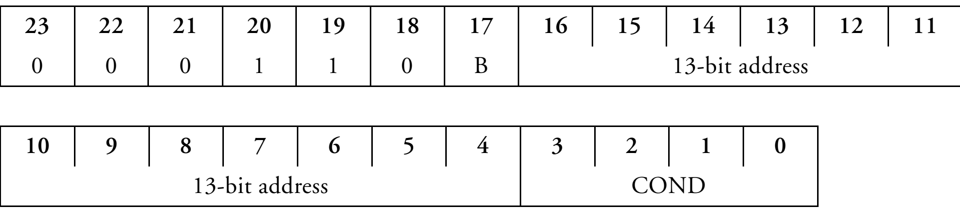
See Also

- “Opcode Mnemonics” on page 8-1
- “Add/Add with Carry” on page 2-5
- “Subtract X–Y/Subtract X–Y with Borrow” on page 2-9
- “Subtract Y–X/Subtract Y–X with Borrow” on page 2-13
- “Bitwise Logic: AND, OR, XOR” on page 2-16
- “Clear: PASS” on page 2-22
- “Negate: NOT” on page 2-25
- “Absolute Value: ABS” on page 2-28
- “Increment” on page 2-31
- “Decrement” on page 2-34
- “Multiply” on page 3-8
- “Multiply with Cumulative Add” on page 3-11
- “Multiply with Cumulative Subtract” on page 3-14

Type 10: Direct Jump

13-bit relative conditional/unconditional jump with delayed branch option

Opcode



23	22	21	20	19	18	17	16	15	14	13	12	11
0	0	0	1	1	0	B	13-bit address					

10	9	8	7	6	5	4	3	2	1	0
13-bit address							COND			

Syntax

```
[IF Cond] JUMP <Reladdr13> [(DB)];
```

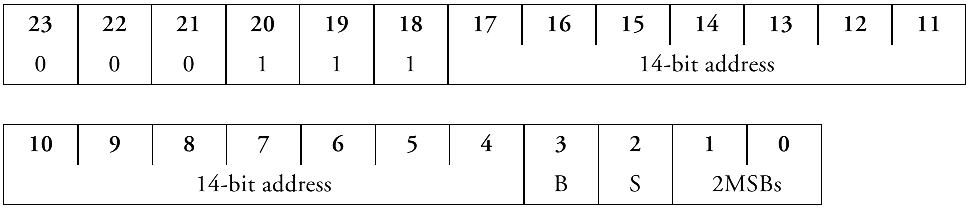
See Also

- “Opcode Mnemonics” on page 8-1
- “Direct JUMP (PC Relative)” on page 7-29

Type 10a: Direct Jump/Call

16-bit relative conditional/unconditional jump with delayed branch option

Opcode



23	22	21	20	19	18	17	16	15	14	13	12	11
0	0	0	1	1	1	14-bit address						

10	9	8	7	6	5	4	3	2	1	0
14-bit address							B	S	2MSBs	

Syntax

```
CALL <Reladdr16> [(DB)];
JUMP <Reladdr16> [(DB)];
```

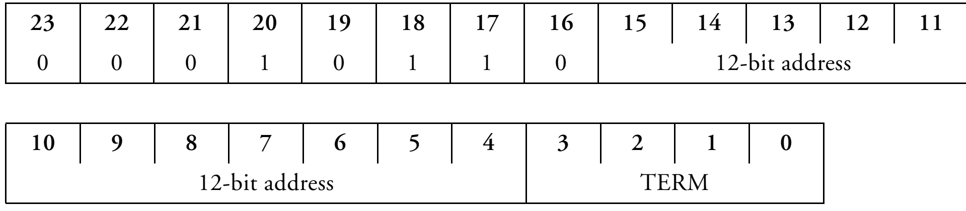
See Also

- “Opcode Mnemonics” on page 8-1
- “CALL (PC Relative)” on page 7-33
- “JUMP (PC Relative)” on page 7-37

Type 11: Do ···Until

12-bit relative conditional DO

Opcode Bits



23	22	21	20	19	18	17	16	15	14	13	12	11
0	0	0	1	0	1	1	0	12-bit address				

10	9	8	7	6	5	4	3	2	1	0
12-bit address							TERM			

Syntax

```
DO <Reladdr12> UNTIL [CE, FOREVER];
```

See Also

- “Opcode Mnemonics” on page 8-1
- “DO UNTIL (PC Relative)” on page 7-24

Type 12: Shift | Dreg «··» DM

Shift with memory read/write using DAG postmodify

Opcode Bits

23	22	21	20	19	18	17	16	15	14	13	12	11
0	0	0	1	0	0	1	G	SF				D

10	9	8	7	6	5	4	3	2	1	0
XOP			DREG				I		M	

Syntax

```
<SHIFT> , Dreg = DM(Ia += Mb);
<SHIFT> , DM(Ia += Mb) = Dreg;
```

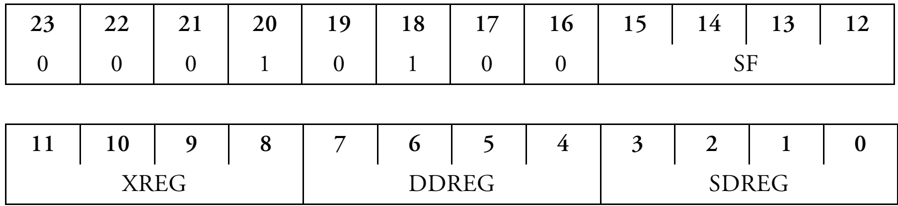
See Also

- "Opcode Mnemonics" on page 8-1
- "Compute with Memory Read" on page 5-11
- "Compute with Memory Write" on page 5-15
- "XOP and YOP Codes" on page 8-15

Type 14: Shift | Dreg1 «··· Dreg2

Register file shift with data register move

Opcode Bits



23	22	21	20	19	18	17	16	15	14	13	12
0	0	0	1	0	1	0	0	SF			

11	10	9	8	7	6	5	4	3	2	1	0
XREG				DDREG				SDREG			

Syntax

```
<SHIFT>, Dreg = Dreg;
```

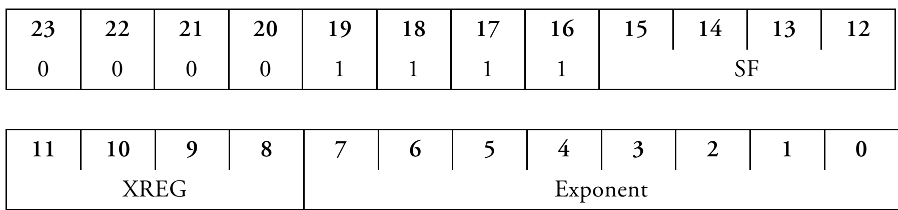
See Also

- “Opcode Mnemonics” on page 8-1
- “Compute with Register-to-Register Move” on page 5-19

Type 15: Shift Data8

Immediate register file shift

Opcode Bits



23	22	21	20	19	18	17	16	15	14	13	12
0	0	0	0	1	1	1	1	SF			

11	10	9	8	7	6	5	4	3	2	1	0
XREG				Exponent							

Syntax

```
SR = [SR OR] ASHIFT BY <Imm8> [(|HI, LO|)];
SR = [SR OR] LSHIFT BY <Imm8> [(|HI, LO|)];
```

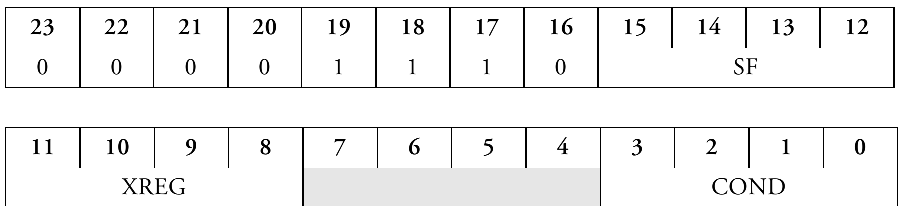
See Also

- "Opcode Mnemonics" on page 8-1
- "Arithmetic Shift Immediate" on page 4-8
- "Logical Shift Immediate" on page 4-12

Type 16: Shift Reg0

Conditional register file shift

Opcode Bits



23	22	21	20	19	18	17	16	15	14	13	12
0	0	0	0	1	1	1	0	SF			

11	10	9	8	7	6	5	4	3	2	1	0
XREG								COND			

Syntax

```
[IF Cond] SR = [SR OR] ASHIFT Dreg [(|HI, LO|)];
[IF Cond] SR = [SR OR] LSHIFT Dreg [(|HI, LO|)];
[IF Cond] SR = [SR OR] NORM Dreg [(|HI, LO|)];
[IF Cond] SE = [SR OR] EXP Dreg [(|HIX, HI, LO|)];
[IF Cond] SB = [SR OR] EXPADJ Dreg;
```

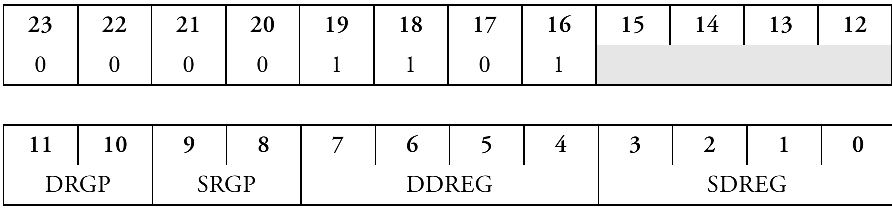
See Also

- “Opcode Mnemonics” on page 8-1
- “Arithmetic Shift” on page 4-6
- “Logical Shift” on page 4-10
- “Normalize” on page 4-14
- “Exponent Derive” on page 4-20
- “Exponent (Block) Adjust” on page 4-23

Type 17: Any Reg «···Any Reg

General register move

Opcode Bits



23	22	21	20	19	18	17	16	15	14	13	12
0	0	0	0	1	1	0	1				

11	10	9	8	7	6	5	4	3	2	1	0
DRGP		SRGP		DDREG				SDREG			

Syntax

```
Reg = Reg;
```

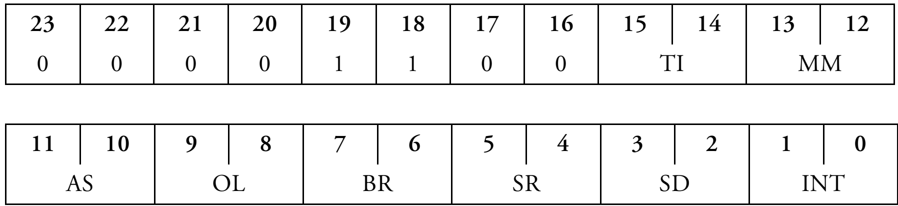
See Also

- "Opcode Mnemonics" on page 8-1
- "Register-to-Register Move" on page 6-22

Type 18: Mode Change

Mode control

Opcode Bits



23	22	21	20	19	18	17	16	15	14	13	12
0	0	0	0	1	1	0	0	TI		MM	

11	10	9	8	7	6	5	4	3	2	1	0
AS		OL		BR		SR		SD		INT	

Syntax

```
ENA | TI, MM, AS, OL, BR, SR, SD, INT |;
DIS | TI, MM, AS, OL, BR, SR, SD, INT |;
```

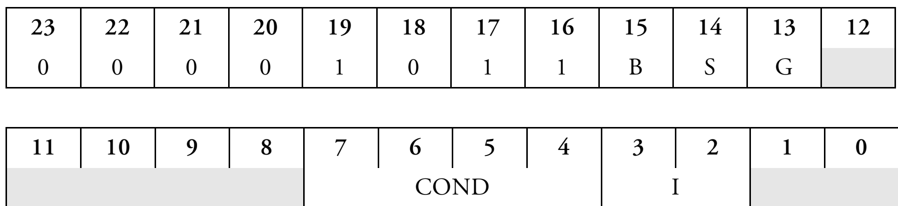
See Also

- “Opcode Mnemonics” on page 8-1
- “Mode Control” on page 7-76

Type 19: Indirect Jump/Call

Conditional indirect jump/call with delayed branch option

Opcode Bits



23	22	21	20	19	18	17	16	15	14	13	12
0	0	0	0	1	0	1	1	B	S	G	

11	10	9	8	7	6	5	4	3	2	1	0
				COND				I			

Syntax

```
[IF Cond] CALL <Ireg> [(DB)];
[IF Cond] JUMP <Ireg> [(DB)];
```

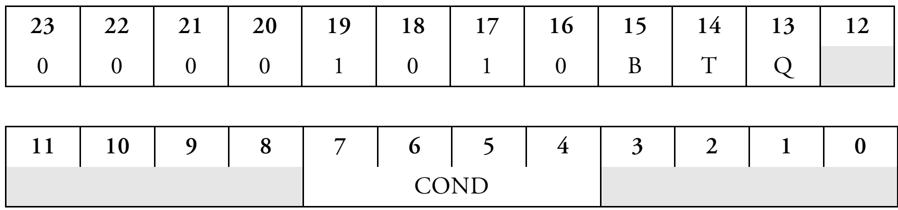
See Also

- "Opcode Mnemonics" on page 8-1
- "Indirect CALL" on page 7-46
- "Indirect JUMP" on page 7-50

Type 20: Return

Conditional return from interrupt/return from subroutine with delayed branch option

Opcode Bits



23	22	21	20	19	18	17	16	15	14	13	12
0	0	0	0	1	0	1	0	B	T	Q	

11	10	9	8	7	6	5	4	3	2	1	0
				COND							

Syntax

```
[IF Cond] RTI [(DB)];
[IF Cond] RTS [(DB)];
```

See Also

- "Opcode Mnemonics" on page 8-1
- "Return from Interrupt (RTI)" on page 7-53
- "Return from Subroutine (RTS)" on page 7-57

Type 21: Modify DagI

DAG modify

Opcode Bits

23	22	21	20	19	18	17	16	15	14	13	12
0	0	0	0	0	0	0	1	1		G	

11	10	9	8	7	6	5	4	3	2	1	0
								I		M	

Syntax

```
|MODIFY (Ia += Mb), MODIFY (Ic += Md)|;
```

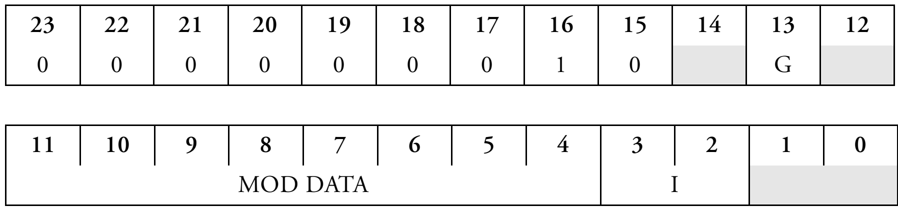
See Also

- “Opcode Mnemonics” on page 8-1
- “Modify Address Register—Indirect” on page 6-67

Type 21a: Modify DagI

DAG modify immediate value

Opcode Bits



23	22	21	20	19	18	17	16	15	14	13	12
0	0	0	0	0	0	0	1	0		G	

11	10	9	8	7	6	5	4	3	2	1	0
MOD DATA								I			

Syntax

```
MODIFY (Ireg += <Imm8>);
```

See Also

- "Opcode Mnemonics" on page 8-1
- "Modify Address Register—Indirect" on page 6-67

Type 22: DM «··· Data16

16-bit immediate data indirect memory write (two-word instruction) using DAG postmodify addressing

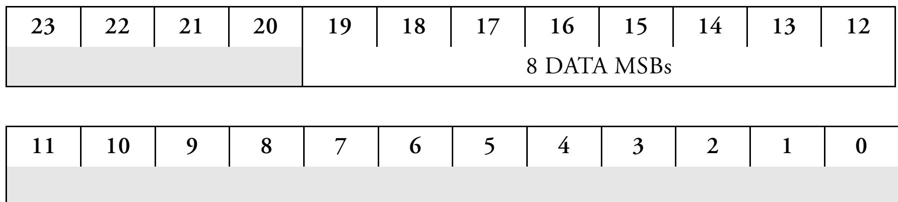
Opcode

First word: 16-bit data write

23	22	21	20	19	18	17	16	15	14	13	12
0	0	0	0	0	1	1	1	1	0	G	

11	10	9	8	7	6	5	4	3	2	1	0
8 DATA LSBs								I		M	

Second word: 16-bit data write



23	22	21	20	19	18	17	16	15	14	13	12
				8 DATA MSBs							

11	10	9	8	7	6	5	4	3	2	1	0

Syntax

```
|DM(Ia += Mb), DM (Ic += Md)| = <Data16>;
```

See Also

- “Opcode Mnemonics” on page 8-1
- “Indirect 16-Bit Memory Write—Immediate Data” on page 6-56

Type 22a: PM «··· Data24

24-bit immediate data indirect memory write (two-word instruction) using DAG postmodify addressing

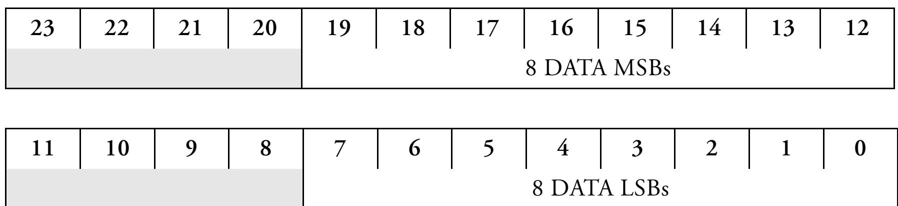
Opcode

First word: 24-bit data write

23	22	21	20	19	18	17	16	15	14	13	12
0	0	0	0	0	1	1	1	1	1	G	

11	10	9	8	7	6	5	4	3	2	1	0
8 DATA MidSBs								I		M	

Second word: 24-bit data write

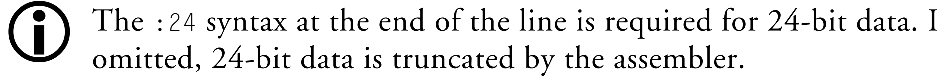


23	22	21	20	19	18	17	16	15	14	13	12
				8 DATA MSBs							

11	10	9	8	7	6	5	4	3	2	1	0
				8 DATA LSBs							

Syntax

```
|PM (Ia += Mb), PM (Ic += Md)| = <Data24>:24;
```



The `:24` syntax at the end of the line is required for 24-bit data. If omitted, 24-bit data is truncated by the assembler.

See Also

- “Opcode Mnemonics” on page 8-1
- “Indirect 24-Bit Memory Write—Immediate Data” on page 6-58

Type 23: Divide primitive, DIVQ

DIVQ divide primitive

Opcode Bits

23	22	21	20	19	18	17	16	15	14	13	12
0	0	0	0	0	0	1	1	1	1	0	1

11	10	9	8	7	6	5	4	3	2	1	0
0	XOP										

Syntax

```
DIVQ Xop;
```

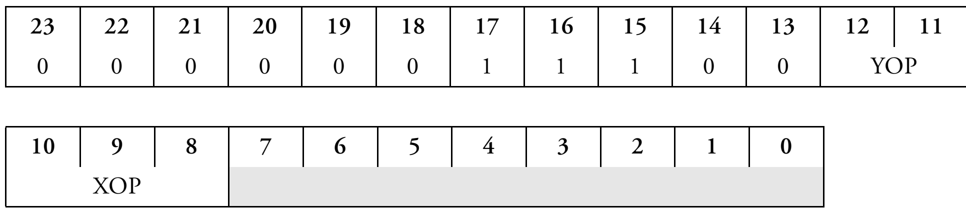
See Also

- “Opcode Mnemonics” on page 8-1
- “Divide Primitives: DIVS and DIVQ” on page 2-37

Type 24: Divide primitive, DIVS

DIVS divide primitive

Opcode Bits



23	22	21	20	19	18	17	16	15	14	13	12	11
0	0	0	0	0	0	1	1	1	0	0	YOP	

10	9	8	7	6	5	4	3	2	1	0
XOP										

Syntax

```
DIVS Yop, Xop;
```

See Also

- “Opcode Mnemonics” on page 8-1
- “Divide Primitives: DIVS and DIVQ” on page 2-37

Type 25: Saturate

Saturate MR/SR on overflow

Opcode Bits

23	22	21	20	19	18	17	16	15	14	13	12
0	0	0	0	0	0	1	1	0	R		

11	10	9	8	7	6	5	4	3	2	1	0

Syntax

```
SAT MR;
SAT SR;
```

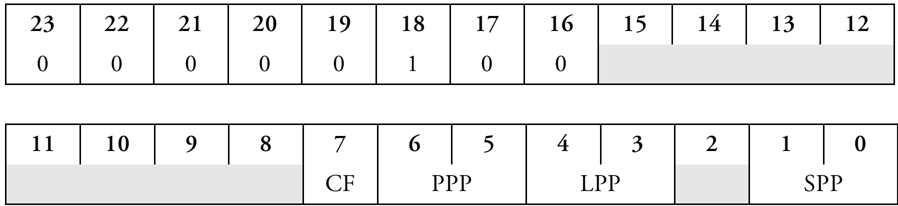
See Also

- “Opcode Mnemonics” on page 8-1
- “MAC Saturate” on page 3-21

Type 26:Push/Pop/Cache

Stack control

Opcode Bits



23	22	21	20	19	18	17	16	15	14	13	12
0	0	0	0	0	1	0	0				

11	10	9	8	7	6	5	4	3	2	1	0
				CF	PPP		LPP			SPP	

Syntax

```
PUSH |PC, LOOP, STS|;
POP  |PC, LOOP, STS|;
FLUSH CACHE;
```

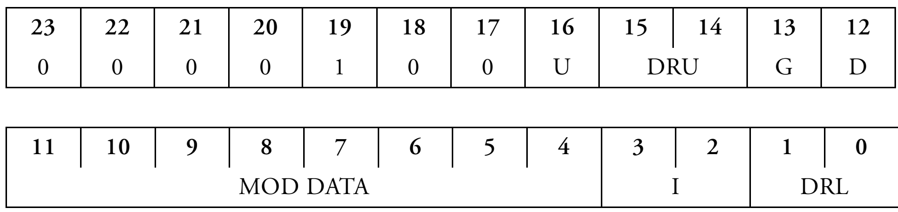
See Also

- "Opcode Mnemonics" on page 8-1
- "PUSH or POP Stacks" on page 7-61
- "FLUSH CACHE" on page 7-67

Type 29: Dreg «··» DM

Memory read/write with immediate modify (postmodify with update or premodify offset)

Opcode Bits



23	22	21	20	19	18	17	16	15	14	13	12
0	0	0	0	1	0	0	U	DRU		G	D

11	10	9	8	7	6	5	4	3	2	1	0
MOD DATA								I		DRL	

Syntax

```
Dreg = DM(Ireg += <Imm8>);        /* postmodify read */
DM(Ireg += <Imm8>) = Dreg;        /* postmodify write */
Dreg = DM(Ireg + <Imm8>);         /* premodify read */
DM(Ireg + <Imm8>) = Dreg;         /* premodify write */
```

See Also

- “Opcode Mnemonics” on page 8-1
- “Indirect Memory Read/Write—Immediate Postmodify” on page 6-50
- “Indirect Memory Read/Write—Immediate Premodify” on page 6-53

Type 30: NOP

No operation

Opcode Bits

23	22	21	20	19	18	17	16	15	14	13	12
0	0	0	0	0	0	0	0				

11	10	9	8	7	6	5	4	3	2	1	0
								SWCD			

Syntax

```
NOP;
```

See Also

- "Opcode Mnemonics" on page 8-1
- "NOP" on page 7-73

Type 31: Idle

Idle

Opcode Bits

23	22	21	20	19	18	17	16	15	14	13	12
0	0	0	0	0	0	1	0	0			

11	10	9	8	7	6	5	4	3	2	1	0
								IDLE VALUE			

Syntax

```
IDLE;
```

See Also

- “Opcode Mnemonics” on page 8-1
- “IDLE” on page 7-74

Type 32: Any Reg «··» PM/DM

DAG memory read/write with premodify offset or postmodify update

Opcode

23	22	21	20	19	18	17	16	15	14	13	12	11
0	0	0	1	0	1	0	1	MS	U	G	D	0

10	9	8	7	6	5	4	3	2	1	0
	RGP		REG				I		M	

Syntax

```
|DM(Ia += Mb), DM(Ic += Md)| = Reg;  /* postmodify write,16-bit*/
Reg = |DM(Ia += Mb), DM(Ic += Md)|;  /* premodify read,16-bit*/
|DM(Ia + Mb), DM(Ic + Md)| = Reg;    /* premodify write,16-bit*/
Reg = |DM (Ia + Mb), DM (Ic + Md)|;  /* postmodify read,16-bit */
|PM(Ia += Mb), PM(Ic += Md)| = Reg;  /* postmodify write,24-bit*/
Reg = |PM(Ia += Mb), PM(Ic += Md)|;  /* premodify read,24-bit*/
|PM(Ia + Mb), PM(Ic + Md)| = Reg;    /* premodify write,24-bit*/
Reg = |PM(Ia + Mb), PM(Ic + Md)|;    /* postmodify read,24-bit*/
```

See Also

- “Opcode Mnemonics” on page 8-1
- “Indirect 16-Bit Memory Read/Write—Postmodify” on page 6-30
- “Indirect 16-Bit Memory Read/Write—Premodify” on page 6-34
- “Indirect 24-Bit Memory Read/Write—Postmodify” on page 6-38
- “Indirect 24-Bit Memory Read/Write—Premodify” on page 6-42

Type 32a: DM«···DAG Reg | DAG Reg«···Ireg

DAG register store with register transfer

Opcode

23	22	21	20	19	18	17	16	15	14	13	12	11
0	0	0	1	0	1	0	1	0	U	G	1	1

10	9	8	7	6	5	4	3	2	1	0
0	RGP		DAG REG				I		M	

Syntax

```
DM(Ireg1 += Mreg1) = |Ireg2, Mreg2, Lreg2|,
    |Ireg2, Mreg2, Lreg2|= Ireg1 ;

DM(Ireg1 += Mreg1) = |Ireg2, Mreg2, Lreg2|,
   |Ireg2, Mreg2, Lreg2| = Ireg1 ;
```

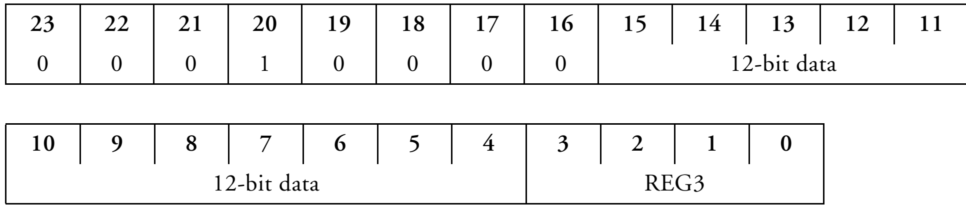
See Also

- "Opcode Mnemonics" on page 8-1
- "Indirect DAG Register Write (Premodify or Postmodify), with DAG Register Move" on page 6-46

Type 33: Reg3 «··· Data12

Load short constants

Opcode Bits



23	22	21	20	19	18	17	16	15	14	13	12	11
0	0	0	1	0	0	0	0	12-bit data				

10	9	8	7	6	5	4	3	2	1	0
12-bit data							REG3			

Syntax

```
Reg3 = <Data12>;
```

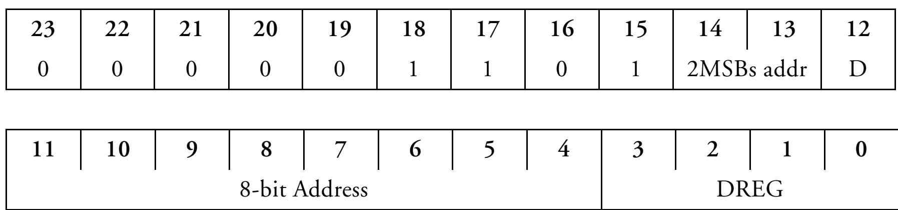
See Also

- “Opcode Mnemonics” on page 8-1
- “Direct Register Load” on page 6-27

Type 34: Dreg «··» IOreg

I/O register read/write

Opcode Bits



23	22	21	20	19	18	17	16	15	14	13	12
0	0	0	0	0	1	1	0	1	2MSBs addr		D

11	10	9	8	7	6	5	4	3	2	1	0
8-bit Address								DREG			

Syntax

```
IO(<Addr10>) = Dreg;
Dreg = IO (<Addr10>);
```

See Also

- “Opcode Mnemonics” on page 8-1
- “External I/O Port Read/Write” on page 6-61

Type 35: Dreg «··»Sreg

System control register read/write

Opcode Bits

23	22	21	20	19	18	17	16	15	14	13	12
0	0	0	0	0	1	1	0	0			D

11	10	9	8	7	6	5	4	3	2	1	0
8-bit Address								DREG			

Syntax

```
REG(<Addr8>) = Dreg;
Dreg = REG(<Addr8>);
```

See Also

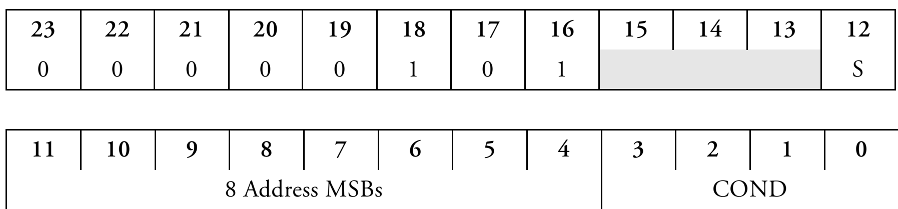
- "Opcode Mnemonics" on page 8-1
- "System Control Register Read/Write" on page 6-64

Type 36: Long Jump/Call

Conditional long jump/call (two-word instruction)

Opcode Bits

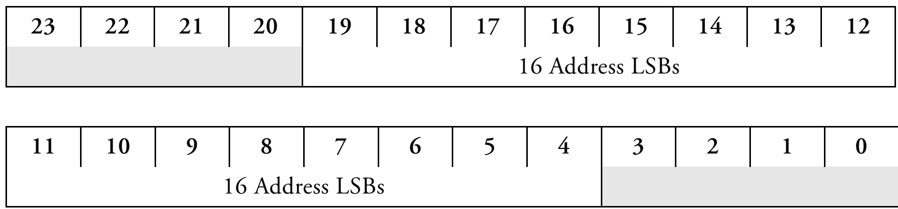
- First word



23	22	21	20	19	18	17	16	15	14	13	12
0	0	0	0	0	1	0	1				S

11	10	9	8	7	6	5	4	3	2	1	0
8 Address MSBs								COND			

- Second word



23	22	21	20	19	18	17	16	15	14	13	12
				16 Address LSBs							

11	10	9	8	7	6	5	4	3	2	1	0
16 Address LSBs											

Syntax

```
[IF Cond] LJUMP <Addr24>;
[IF Cond] LCALL <Addr24>;
```

See Also

- "Opcode Mnemonics" on page 8-1
- "Long Call (LCALL)" on page 7-40
- "Long Jump (LJUMP)" on page 7-43

Type 37: Interrupt

Software interrupt

Opcode Bits

23	22	21	20	19	18	17	16	15	14	13	12
0	0	0	0	0	1	1	1	0			

11	10	9	8	7	6	5	4	3	2	1	0
						C		BIT			

Syntax

```
SETINT <Imm4>;
CLRINT <Imm4>;
```

See Also

- “Opcode Mnemonics” on page 8-1
- “Set Interrupt (SETINT)” on page 7-69
- “Clear Interrupt (CLRINT)” on page 7-71

I INDEX

A

ABS instruction 2-28
absolute value instruction 2-28
AC bit 1-4
AC condition 1-11
add/add with carry 2-5
addressing
  branch targets 7-6
alternate registers (*see secondary registers*)
ALU
  data registers 1-2
  instructions summary 1-14
ALU carry (AC) condition 1-11
ALU mode control 2-3
ALU or multiplier function (AMF) 8-1
  codes 8-6
ALU overflow (AV) bit 1-8
ALU overflow (AV) condition 1-11
ALU overflow latch mode enable (AV_LATCH) bit 1-8, 7-76
ALU overflow latch mode enable (OL) bit 1-8
ALU quotient (AQ) bit 1-4
ALU result carry (AC) bit 1-4
ALU result negative (AN) bit 1-4
ALU result overflow (AV) bit 1-4
ALU saturation mode enable (AR_SAT) bit 1-9, 7-76
ALU saturation mode enable (AS) bit 1-9
ALU signed (AS) bit 8-1
ALU status flags 2-4
ALU x- and y-input (AX AY) registers 8-2
ALU x-input sign (AS) bit 1-4
ALU zero (AZ) bit 1-4
always true (TRUE) condition 1-11
AN bit 1-4
AND operator 2-16
AQ bit 1-4
AR_SAT bit 1-9, 7-76
arithmetic shift (ASHIFT) instruction 4-3, 4-6
arithmetic shift immediate 4-8
Arithmetic Status (ASTAT) register 1-3, 1-8, 6-2, 7-7
arithmetic status bits 1-4
AS bit 1-4, 1-9
ASHIFT instruction 4-3, 4-6, 4-26
ASTAT register 1-3, 6-2
AV bit 1-4, 1-8
AV condition 1-11

AV_LATCH bit 1-8, 7-76
AZ bit 1-4

B

background registers (*see secondary registers*)
Base (B0-7) registers 1-3, 6-5, 6-14
bias rounding enable (BIASRND) bit 3-5
BIASRND bit 1-6, 3-5
bit manipulation instructions 2-19
BIT_REV bit 1-8, 6-16, 7-76
bit-reversed addressing
  about 6-16
bit-reversed addressing enable (BIT_REV) bit 1-8, 6-16, 7-76
bit-reversed addressing enable (BR) bit 1-8
bitwise logic instructions 2-16
BR bit 1-8
branch options 7-4
branching
  preparation 7-34

C

CALL (PC relative) 7-33
CALL instruction 7-5, 7-6, 7-7, 7-8, 7-19, 7-20, 7-34, 7-41, 7-46
calls 7-20
CCODE register 1-5, 6-2
  about 7-3
CE condition 1-11, 7-10
circular buffer addressing 5-9, 5-17
  restrictions 6-15
circular data buffers
  addressing 6-14
clear (PASS) instruction 2-22
clear bit (CLRBIT) instruction 2-19
clear interrupt (CLRINT) instruction 8-56
  about 7-71
clear multiplier instruction 3-17
CLRBIT instruction 2-19
CLRINT instruction 8-56
  about 7-71
compute with dual memory read 5-4
compute with memory read 5-11
compute with memory write 5-15
compute with register-to-register move 5-19
Condition Code (CCODE) register 1-5, 6-2
  about 7-3
condition codes 1-11, 8-8
  summary 1-11
conditional instruction (COND) codes 1-11
Conditional Instructions xv
conditions 7-2
  counter-based 7-2
  loop termination 7-24
constant
  codes 8-9
constants 2-2
context switching 7-16
Conventions xxiii
core register

codes 8-11
core registers 6-2
summary 1-2
Counter (CNTR) register 7-2, 7-7, 7-25
operation 7-26
counter expired (CE) condition 1-11, 7-2, 7-10, 7-24, 7-25, 8-5, 8-8
counter-based conditions 7-2
customer support xv

D

DAG registers
about 6-5
DAG1 (G1reg) registers 6-3
DAG2 (G2reg) registers 6-3
Data Address Generator (DAG) registers 6-5
restrictions 6-13
data addressing
methods 6-11
data format options 3-3
Data Memory Page (DMPGx) registers 6-2, 6-32
about 6-6
data move instructions 1-16
DB option 7-21
decrement instruction 2-34
delayed branch (DB) option 7-4, 7-21, 7-31
latency 7-35, 7-48
restrictions 7-5, 7-34
denormalization 4-26
direct addressing 6-11
direct JUMP (PC relative) 7-29
direct memory read/write—immediate address 6-24
direct register load 6-27
DIS instruction 6-8, 7-76, 7-77
disable (DIS) instruction 7-76
divide primitives (DIVS DIVQ) 2-37, 5-4, 5-11, 5-15, 5-19
division
applications 2-45
code example 2-45
exceptions 2-43
integer 2-43
signed 2-39
theory 2-40
unsigned 2-39
DIVQ instruction 2-37, 5-4, 5-11, 5-15, 5-19
DIVS instruction 2-37, 5-4, 5-11, 5-15, 5-19
DMI
codes 8-14
DMM
codes 8-14
DMPGx registers 6-2, 6-32
about 6-6
DO UNTIL instruction 7-2, 7-4, 7-7, 7-10, 7-24, 7-34
dual memory read 5-8

E

effect latencies
  about 7-22
  registers 6-9
EMU bit 1-7
EMUCNTE bit 1-6
emulation
  single-step option 7-53
emulator cycle counter interrupt enable (EMUCNTE) bit 1-6
emulator interrupt mask (EMU) bit 1-7
ENA instruction 6-8, 7-77
enable (ENA) instruction 7-76
ending loops
  termination 7-24
EQ condition 1-11
equal to zero (EQ) condition 1-11
execution order
  multifunction operations 5-2
exiting
  loops 7-18
EXP instruction 4-3, 4-20
EXPADJ instruction 4-3, 4-23
exponent adjust (EXPADJ) instruction 4-3
exponent block adjust (EXPADJ) instruction 4-23
exponent derive (EXP) instruction 4-3, 4-14, 4-20, 4-26
external I/O port read/write 6-62

F

far absolute branches 7-6
finite loops 7-10
FLUSH CACHE instruction
  about 7-67
FOREVER condition 7-11, 7-24, 7-25, 7-63
fractional mode (*see* multiplier results mode) 1-9

G

G1reg 6-3
G2reg 6-3
G3reg 6-3
GE condition 1-11
generate ALU status only (NONE) instruction 2-46
generate MAC status only (NONE) instruction 3-24
GIE bit 1-6
global interrupt enable (GIE) bit 1-6
global interrupts
  enabling 7-14
greater than (GT) condition 1-11
greater than or equal to zero (GE) condition 1-11
Greg
  codes 8-15
GT condition 1-11

H

HI option 4-3
high half output shift (HI) option
4-3
high half output shift unless
overflow (HIX) option 4-3
HIX option 4-3

I

I/O Memory Page (IOPG) register
6-2, 6-63
ICNTL register 1-6
IDLE instruction 7-74
about 7-74
IJPG register 6-2, 7-46
IMASK register 1-7, 7-14
increment instruction 2-31
Index (I0-7) registers 5-5, 5-8, 5-12,
5-16, 6-5, 6-6, 6-14, 6-16, 8-15
index (Ix) registers
codes 8-13
indirect 16-bit memory
read/write—premodify 6-34
indirect 16-bit memory write—
immediate data 6-57
indirect 24-bit memory
read/write—postmodify 6-38
indirect 24-bit memory
read/write—premodify 6-43
indirect 24-bit memory write—
immediate data 6-59
indirect addressing 6-11
indirect branches 7-6
indirect call (CALL) instruction
7-46
indirect DAG register write
(premodify or postmodify) with
DAG register move 6-47
indirect jump (JUMP) instruction
7-50
Indirect Jump Memory Page (IJPG)
register 6-2, 7-46
indirect memory read/write—
immediate postmodify 6-51
indirect memory read/write—
immediate premodify 6-54
INE bit 1-6
infinite loops 7-10, 7-26, 7-63
input registers 2-1
multiplier 3-2
instruction
(Type 1 compute | regX«•••DM
| DregY«•••PM 8-17
(Type 3) Dreg/Greg
«•••»DM/PMType 3
Dreg/Greg «…» DM/PM 8-18
(Type 4) compute | Dreg
«•••»DM 8-19
(Type 6) Dreg «•••Data16 8-20
(Type 7) Reg1/2 «•••Data16
8-21
(Type 8) compute | Dreg1
«•••Dreg2 8-22
(Type 9) compute 8-23
(Type 9a) compute 8-26
(Type 10) direct jump/call 8-28
(Type 10a) direct jump/call 8-29

(Type 11) Do ••• Until 8-30
(Type 12) Shift | Dreg «•••» DM 8-31
(Type 14) Shift | Dreg1 «••• Dreg2 8-32
(Type 15) shift data8 8-33
(Type 16) shift Reg0 8-34
(Type 17) Dreg1 «••• Dreg2 8-35
(Type 18) mode change 8-36
(Type 19) indirect jump/call 8-37
(Type 20) return 8-38
(Type 21) modify DagI 8-39
(Type 22) DM «••• data16 8-41
(Type 22a) PM «••• data24 8-42
(Type 23) divide primitive, DIVQ 8-43
(Type 24) divide primitive, DIVS 8-44
(Type 25) saturate 8-45
(Type 26) push/pop/cache 8-46
(Type 29) Dreg «•••» DM 8-47
(Type 30) NOP 8-48
(Type 31) idle 8-49
(Type 32) Reg «•••» PM/DM 8-50
(Type 32a) Reg «•••» DM 8-51
(Type 33) Reg «••• data12 8-52
(Type 34) Dreg «•••» oreg 8-53
(Type 35) Dreg «•••» Sreg 8-54
(Type 36) long jump/call 8-55
(Type 37) interrupt 8-56

instruction pipeline 7-35, 7-48
instruction set
  notation 1-12
instructions
  ALU 1-14
  conditional 1-11
  multifunction 1-19, 5-1
  multiplier 1-15
  program flow 1-18
  shifter 1-16
  summary 1-12
INT bit 7-76
integer division 2-43
Interrupt Control (ICNTL) register 1-6
Interrupt Latch (IRPTL) register 1-7, 7-14
Interrupt Mask (IMASK) register 1-7, 7-14
interrupt nesting enable (INE) bit 1-6
interrupt service routines (ISRs) 7-13, 7-53
interrupts
  about 7-13
  enabling 7-14
  nesting 7-16
interrupts enable (INT) bit 7-76
IOPG register 6-2, 6-63
IRPTL register 1-7, 7-14
ISRs 7-13, 7-53

J

JUMP (PC relative) instruction 7-37
JUMP instruction 7-5, 7-6, 7-29, 7-34
  long jumps 7-20

L

latency
  about 7-22
  delayed branch 7-35
  registers 6-9
LCALL instruction 7-21, 7-35, 7-40, 7-41
LE condition 1-11
Length (L0-7) registers 5-9, 5-13, 5-17, 6-5, 6-14
less than or equal to zero (LE) condition 1-11
less that zero (LT) condition 1-11
linear indirect addressing 5-17
LJUMP instruction 7-21, 7-31, 7-43, 7-44
LO option 4-3
logical shift (LSHIFT) instruction 4-3, 4-7, 4-10
logical shift immediate 4-12
long call (LCALL) instruction
  about 7-40
long jump (LJUMP) instruction
  about 7-43
long jumps 7-20
loop begin stack 7-7
loop counter stack 7-7
loop end stack 7-7
Loop PC Stack Pointer (LPCSTACKP) register 6-2
loop stack empty (LPSTKEMPTY) bit 1-10
loop stack empty (LSE) bit 1-10
loop stack full (LPSTKFULL) bit 1-10
loop stack full (LSF) bit 1-10
loop stacks
  operation 7-10
  pushing and popping 7-11
loop termination 7-64
  conditions 7-24
loops
  exiting 7-18
  finite 7-10
  infinite 7-10
  PUSH/POP 7-11
  restrictions 7-26
low half output shift (LO) option 4-3
LPCSTACKP register 6-2
LPSTKEMPTY bit 1-10
LPSTKFULL bit 1-10
LSE bit 1-10
LSF bit 1-10
LSHIFT instruction 4-3, 4-7, 4-10, 4-26
LT condition 1-11

M

M_MODE bit 1-9, 7-76
MAC biased rounding mode (BIASRND) bit 1-6
MAC clear instruction 3-17
MAC input registers 3-2
MAC output registers 3-2
MAC overflow (MV) condition 1-11
MAC result mode (M_MODE) bit 1-9
MAC result mode (MM) bit 1-9
MAC round/transfer (RND) instruction 3-19
MAC saturate (SAT) instruction 3-21
Manual
  audience xiii
  contents description xiv
  conventions xxiii
  new in this edition xv
manual
  related documents xix
MM bit 1-9
mode control 7-4, 7-76
Mode Status (MSTAT) register 1-8, 6-2, 6-7, 6-16, 7-7
Modify (M0-7) registers 1-2, 5-5, 5-8, 5-12, 5-16, 6-5, 6-14
modify (Mx) registers
  codes 8-13
MODIFY instruction
  direct 6-70
  indirect 6-68
Mreg
  codes 8-15
MSTAT register 1-8, 6-2, 6-7, 6-16, 7-4
multifunction instruction
  stall cycles 5-6, 5-10, 5-13, 5-17
multifunction instructions 5-1
  execution order 5-2
  summary 1-19
multiplier clear instruction 3-17
multiplier input registers 3-2
multiplier instructions
  summary 1-15
multiplier overflow (MV) bit 1-4, 3-6, 3-7
multiplier results mode selection (M_MODE) bit 3-6, 7-76, 8-3
multiplier round/transfer (RND) instruction 3-19
Multiplier x- and y-input (MX MY) registers 8-2
multiply instruction 3-8
multiply with cumulative add 3-11
multiply with cumulative subtract 3-14
MV bit 1-4
MV condition 1-11

N

NE condition 1-11
negate instruction
  NOT 2-25
nested interrupts 7-16

nesting
  interrupts 7-16
no operation (NOP) instruction
  about 7-73
NONE instruction 2-46, 3-24
NOP instruction
  about 7-73
NORM instruction 4-3, 4-14
normalize (NORM) instruction
    4-2, 4-3, 4-23
normalize immediate instruction
    4-17
normalize instruction 4-14
NOT CE condition 1-11
not equal to zero (NE) condition
    1-11
numeric format modes 3-6

O

OL bit 1-8
ones complement 2-25, 2-28
opcode
  definitions 8-16
  mnemonics 8-1
optimizing
  performance 7-17
OR operator 2-16
output registers 2-2
  MAC 3-2

P

PASS instruction 2-22
PC register 7-24
PC relative branches 7-6
PC stack 7-7
  operation 7-8
  restrictions 7-7
PC stack empty (PCE) bit 1-10
PC stack empty (PCSTKEMPTY)
    bit 1-10
PC stack full (PCF) bit 1-10
PC stack full (PCSTKFULL) bit
    1-10
PC stack interrupt enable
    (PCSTKE) bit 1-6
PC stack level (PCL) bit 1-10
PC stack level (PCSTKLVL) bit
    1-10
PC stack level status (PCSTKLVL)
    bit 7-12
PCE bit 1-10
PCF bit 1-10
PCL bit 1-10
PCSTKE bit 1-6
PCSTKEMPTY bit 1-10
PCSTKFULL bit 1-10
PCSTKLVL bit 1-10, 7-12
performance
  optimizing 7-17
pipeline 7-35, 7-48
PMI
  codes 8-14
PMM
  codes 8-14
POP instruction 7-7, 7-11, 7-18,
    7-34, 7-64
POP LOOP instruction 7-9, 7-26
POP PC instruction 7-9

POP stacks 7-61
POP STS instruction 7-9
postmodify addressing 5-13, 5-17, 6-48
power-down interrupt mask (PWDN) bit 1-7
premodify addressing 6-12, 6-48
program counter
  pushing and popping 7-9
Program Counter (PC) register 7-24
program flow instructions
  summary 1-18
Program Memory Bus Exchange (PX) register 5-6, 5-9, 5-13, 5-17, 6-2, 6-38
  about 6-3
Programming information xiii
PUSH instruction 7-7, 7-11, 7-34, 7-63
PUSH LOOP instruction 7-9, 7-26
PUSH PC instruction 7-9
PUSH stacks 7-61
PUSH STS instruction 7-9
PUSH/POP loops 7-11
PWDN bit 1-7
PX register 5-6, 5-9, 5-13, 5-17, 6-2, 6-38
  about 6-3

Q

quotient (division) 2-37

R

registers
  background 1-8
  core 1-2, 6-2
  input 2-1
  load latencies 6-9
  output 2-2
  register groups (Reg0-3) 6-2
  shifter 4-2
register-to-register move 6-22
related documents xix
result registers 1-2
return from interrupt (RTI) instruction 7-7, 7-13, 7-34
  about 7-53
return from subroutine (RTS) instruction 7-7, 7-34
  about 7-57
  restrictions 7-57
RND instruction 3-19
round/transfer (RND) multiplier instruction 3-19
rounding (RND) multiplier option 3-3
rounding modes 3-4
RTI instruction 7-5, 7-7, 7-13
  about 7-53
  single-step (SS) RTI option 7-53
RTS instruction 7-5, 7-7, 7-8
  about 7-57

S

SAT instruction 3-21
saturation
  testing for 3-21
SB register 4-2
SD bit 1-9
SE register 4-3
SEC_DAG bit 1-9, 6-7, 7-76
SEC_REG bit 1-8, 7-76
secondary DAG registers
  about 6-7
secondary DAG registers enable (SD) bit 1-9
secondary DAG registers enable (SEC_DAG) bit 1-9, 6-7, 7-76
secondary DAG registers enable (SEC_REG) bit 1-8, 7-76
secondary DAG registers enable (SR) bit 1-8
set bit (SETBIT) instruction 2-19
set interrupt (SETINT) instruction 8-56
  about 7-69
SETBIT instruction 2-19
SETINT instruction 8-56
  about 7-69
shift function
  codes 8-12
shifter
  options 4-3
  status flags 4-5
Shifter Block Exponent (SB) register 4-2
Shifter Exponent (SE) register 4-3, 4-26
Shifter Input (SI) register 4-3
shifter input sign (SS) bit 1-4
shifter instructions
  options 4-3
  summary 1-16
shifter overflow (SV) bit 1-4, 3-6, 3-7
shifter registers 4-2
Shifter Result (SR0-2) registers 1-2, 4-2
shifter signed (SS) bit 4-21
SI register 4-3
signed division 2-39
signed input (S) multiplier option 3-3
single-step interrupt mask (SSTEP) bit 1-7
single-step option 7-53
sleep mode 7-74
software condition (SWCOND) bits 1-5
software condition true (SWCOND) condition 1-11, 7-3
SOV bit 1-10
spill-fill mode 7-12
SR bit 1-8
SR0 register 4-2
SR1 register 4-2
SR2 register 4-2
SS bit 1-4, 3-4, 4-21, 7-53
SSE bit 1-10

SSTAT register 1-10, 6-2, 7-12
SSTEP bit 1-7
stack
  status flags 7-12
STACK bit 1-7
stack interrupt mask (STACK) bit 1-7
stack overflow status (STKOVERFLOW) bit 7-12
Stack Pointer (STACKP) register 6-2
STACKP register 6-2
stacks
  about 7-7
  loop begin 7-7
  loop counter 7-7
  loop end 7-7
  PC 7-7
  POP 7-61
  PUSH 7-61
  status 7-7
stacks overflowed (SOV) bit 1-10
stacks overflowed (STKOVERFLOW) bit 1-10
status flags 3-7
  ALU 2-4
  shifter 4-5
  stack 7-12
status stack 7-7
  operation 7-8
  pushing and popping 7-9
status stack empty (SSE) bit 1-10
status stack empty (STSSTKEMPTY) bit 1-10
Status/Control (G3reg) registers 6-3
STKOVERFLOW bit 1-10, 7-12
STSSTKEMPTY bit 1-10
SU bit 3-4
subtract X-Y 2-9
subtract X-Y with borrow 2-9
subtract Y-X 2-13
subtract Y-X with borrow 2-13
support, technical or customer xv
SV bit 1-4
SWCOND bit 1-5, 1-11, 7-3
switching
  context 7-16
system control register read/write 6-65
System Status (SSTAT) register 1-10, 6-2, 7-12
  latency 6-10, 7-9, 7-62

T

technical support xv
termination conditions
  loops 7-24
test bit (TSTBIT) instruction 2-19
TESTBIT instruction 2-19
TGLBIT instruction 2-19
TI bit 1-9
TIMER bit 1-9, 7-76
timer enable (TI) bit 1-9
timer enable (TIMER) bit 1-9, 7-76, 8-5
toggle bit (TGLBIT) instruction 2-19
TRUE condition 1-11, 8-5, 8-8

U

unsigned division 2-39
unsigned input (U) multiplier
option 3-3
US bit 3-4
UU bit 3-4

X

X-input operand (XOP) 2-2
XOP
codes 8-15
XOR operator 2-16

Y

Y-input operand (YOP) 2-2
YOP
codes 8-15