Copyright Information

© 2013 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, SHARC, TigerSHARC, CrossCore, VisualDSP++, and EZ-KIT Lite 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 .............................................................. xi
Intended Audience ................................................................. xii
Manual Contents ....................................................................... xii
What’s New in This Manual ....................................................... xiii
Technical Support ...................................................................... xiv
Supported Processors ............................................................... xv
Product Information ............................................................... xv
  Analog Devices Web Site ......................................................... xvi
  EngineerZone ............................................................................ xvi
Notation Conventions ............................................................. xvii
Register Diagram Conventions ................................................. xviii

INSTRUCTION SUMMARY

Chapter Overview ...................................................................... 1-1
Development Tools .................................................................... 1-2
Compute and Move/Modify Summary ...................................... 1-3
Program Flow Control Summary ............................................. 1-5
Immediate Move Summary ..................................................... 1-7
Miscellaneous Operations Summary .............................................. 1-9
Register Types Summary ............................................................ 1-10
Memory Addressing Summary ..................................................... 1-16
Instruction Set Notation Summary .............................................. 1-17
Conditional Execution Codes Summary ...................................... 1-20
SISD/SIMD Conditional Testing Summary .................................. 1-22
Instruction Opcode Acronym Summary ...................................... 1-23
Universal Register Codes ............................................................. 1-28
ADSP-21160 Instruction Opcode Map ....................................... 1-33

**COMPUTE AND MOVE**

Group I Instructions ..................................................................... 2-1
Type 1: Compute, Dreg«···»DM | Dreg«···»PM .............................. 2-3
Type 2: Compute .......................................................................... 2-7
Type 3: Compute, ureg«···»DM | PM, register modify ............... 2-9
Type 4: Compute, dreg«···»DM | PM, data modify .................... 2-14
Type 5: Compute, ureg«···»ureg | Xdreg<-»Ydreg .................... 2-19
Type 6: Immediate Shift, dreg«···»DM | PM ............................. 2-23
Type 7: Compute, modify .......................................................... 2-28

**PROGRAM FLOW CONTROL**

Group II Instructions ................................................................. 3-1
Type 8: Direct Jump | Call ............................................................ 3-3
Type 9: Indirect Jump | Call, Compute ......................................... 3-8
Type 10: Indirect Jump | Compute, dreg«···»DM .......................... 3-14
Type 11: Return From Subroutine | Interrupt, Compute .............. 3-19
Type 12: Do Until Counter Expired ................................................ 3-24
Type 13: Do Until ....................................................................... 3-26

IMMEDIATE MOVE INSTRUCTIONS

Group III Instructions .................................................................. 4-1
Type 14: Ureg«···»DM | PM (direct addressing) ......................... 4-2
Type 15: Ureg«···»DM | PM (indirect addressing) ......................... 4-5
Type 16: Immediate data···»DM | PM ........................................... 4-9
Type 17: Immediate data···»Ureg .................................................. 4-12

MISCELLANEOUS OPERATIONS

Group IV Instructions .................................................................. 5-1
Type 18: System Register Bit Manipulation ................................. 5-2
Type 19: I Register Modify | Bit-Reverse ..................................... 5-6
Type 20: Push, Pop Stacks, Flush Cache .................................... 5-9
Type 21: Nop .............................................................................. 5-11
Type 22: Idle ............................................................................... 5-12
Type 25: Cjump/Rframe ............................................................. 5-13

COMPUTATIONS REFERENCE

Compute Field .............................................................................. 6-1
ALU Operations ........................................................................... 6-3
    Fixed-Point ALU Operations .................................................. 6-4
    Floating-Point ALU Operations ............................................. 6-5
Rn = Rx + Ry ................................................................. 6-7
Rn = Rx – Ry ................................................................. 6-8
Rn = Rx + Ry + CI .......................................................... 6-9
Rn = Rx – Ry + CI – 1 .................................................... 6-10
Rn = (Rx + Ry)/2 ........................................................... 6-11
COMP(Rx, Ry) .............................................................. 6-12
COMPU(Rx, Ry) ............................................................ 6-13
Rn = Rx + CI ............................................................... 6-14
Rn = Rx + CI – 1 ........................................................... 6-15
Rn = Rx + 1 ................................................................. 6-16
Rn = Rx – 1 ................................................................. 6-17
Rn = –Rx ................................................................. 6-18
Rn = ABS Rx .............................................................. 6-19
Rn = PASS Rx .............................................................. 6-20
Rn = Rx AND Ry ........................................................... 6-21
Rn = Rx OR Ry ............................................................. 6-22
Rn = Rx XOR Ry ........................................................... 6-23
Rn = NOT Rx ............................................................. 6-24
Rn = MIN(Rx, Ry) ......................................................... 6-25
Rn = MAX(Rx, Ry) ......................................................... 6-26
Rn = CLIP Rx BY Ry ...................................................... 6-27
Fn = Fx + Fy ............................................................. 6-28
Fn = Fx – Fy ............................................................. 6-29
Fn = ABS (Fx + Fy) ...................................................... 6-30
Fn = ABS (Fx – Fy) ................................................................. 6-31
Fn = (Fx + Fy)/2 ................................................................. 6-32
COMP(Fx, Fy) ................................................................. 6-33
Fn = –Fx ................................................................. 6-34
Fn = ABS Fx ................................................................. 6-35
Fn = PASS Fx ................................................................. 6-36
Fn = RND Fx ................................................................. 6-37
Fn = SCALB Fx BY Ry ..................................................... 6-38
Rn = MANT Fx ................................................................. 6-39
Rn = LOGB Fx ................................................................. 6-40
Rn = FIX Fx
  Rn = TRUNC Fx
  Rn = FIX Fx BY Ry
  Rn = TRUNC Fx BY Ry ................................................. 6-41
Fn = FLOAT Rx BY Ry
  Fn = FLOAT Rx ................................................................. 6-43
Fn = RECIPS Fx ................................................................. 6-44
Fn = RSQRTS Fx ................................................................. 6-46
Fn = Fx COPYSIGN Fy ............................................................. 6-48
Fn = MIN(Fx, Fy) ................................................................. 6-49
Fn = MAX(Fx, Fy) ................................................................. 6-50
Fn = CLIP Fx BY Fy ................................................................. 6-51
Multiplier Operations ................................................................. 6-52
Multiplier Fixed-Point Operations ............................................ 6-53
Multiplier Floating-Point Operations ..................................... 6-54
Mod1 and Mod2 Modifiers ................................................... 6-54

Rn = Rx * Ry mod2
MRF = Rx * Ry mod2
MRB = Rx * Ry mod2 ......................................................... 6-56

Rn = MRF + Rx * Ry mod2
Rn = MRB + Rx * Ry mod2
MRF = MRF + Rx * Ry mod2
MRB = MRB + Rx * Ry mod2 ................................................. 6-57

Rn = MRF – Rx * Ry mod2
Rn = MRB – Rx * Ry mod2
MRF = MRF – Rx * Ry mod2
MRB = MRB – Rx * Ry mod2 ................................................. 6-58

Rn = SAT MRF mod1
Rn = SAT MRB mod1
MRF = SAT MRF mod1
MRB = SAT MRB mod1 ...................................................... 6-59

Rn = RND MRF mod1
Rn = RND MRB mod1
MRF = RND MRF mod1
MRB = RND MRB mod1 ..................................................... 6-60

MRF = 0
MRB = 0 ................................................................................. 6-61

MRxF/B = Rn/Rn = MRxF/B .................................................. 6-62

Fn = Fx * Fy ............................................................................ 6-64
Shifter Operations ................................................................. 6-64

Shifter Opcodes ................................................................. 6-64

Rn = LSHIFT Rx BY Ry
    Rn = LSHIFT Rx BY <data8> ........................................... 6-66

Rn = Rn OR LSHIFT Rx BY Ry
    Rn = Rn OR LSHIFT Rx BY <data8> ................................. 6-67

Rn = ASHIFT Rx BY Ry
    Rn = ASHIFT Rx BY <data8> .......................................... 6-68

Rn = Rn OR ASHIFT Rx BY Ry
    Rn = Rn OR ASHIFT Rx BY <data8> ................................. 6-69

Rn = ROT Rx BY Ry
    Rn = ROT Rx BY <data8> ................................................ 6-70

Rn = BCLR Rx BY Ry
    Rn = BCLR Rx BY <data8> ............................................. 6-71

Rn = BSET Rx BY Ry
    Rn = BSET Rx BY <data8> ............................................. 6-72

Rn = BTGL Rx BY Ry
    Rn = BTGL Rx BY <data8> ............................................. 6-73

BTST Rx BY Ry
    BTST Rx BY <data8> .................................................. 6-74

Rn = FDEP Rx BY Ry
    Rn = FDEP Rx BY <bit6>:<len6> .................................... 6-75

Rn = Rn OR FDEP Rx BY Ry
    Rn = Rn OR FDEP Rx BY <bit6>:<len6> ............................ 6-77

Rn = FDEP Rx BY Ry (SE)
    Rn = FDEP Rx BY <bit6>:<len6> (SE) ......................... 6-79

Rn = Rn OR FDEP Rx BY Ry (SE)
    Rn = Rn OR FDEP Rx BY <bit6>:<len6> (SE) .................... 6-81
Rn = FEXT Rx BY Ry
     Rn = FEXT Rx BY <bit6>:<len6> .............................................  6-83
Rn = FEXT Rx BY Ry (SE)
     Rn = FEXT Rx BY <bit6>:<len6> (SE) .............................................  6-85
Rn = EXP Rx ..............................................................................  6-87
Rn = EXP Rx (EX) .....................................................................  6-88
Rn = LEFTZ Rx .........................................................................  6-89
Rn = LEFTO Rx .........................................................................  6-90
Rn = FPACK Fx .........................................................................  6-91
Fn = FUNPACK Rx ...................................................................  6-92
Multifunction Computations ......................................................  6-93
     Operand Constraints .............................................................  6-93
Parallel Add and Subtract .............................................................  6-95
Parallel Multiplier and ALU ........................................................  6-98
Parallel Multiplier With Add and Subtract ................................. 6-101

INDEX
Thank you for purchasing Analog Devices SHARC® digital signal processor (DSP).

Purpose of This Manual

The ADSP-21160 SHARC DSP Instruction Set Reference provides assembly syntax information for the ADSP-21160 Super Harvard Architecture (SHARC) Digital Signal Processor (DSP). 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 ADSP-21160 SHARC DSP Hardware Reference.

Intended Audience

The primary audience for this manual is a programmer who is familiar with Analog Devices processors. The manual assumes 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 hardware and programming reference manuals that describe their 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 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.
- “Compute and Move” on page 2-1 – These instruction specify a compute operation in parallel with one or two data moves or an index register modify.
- “Program Flow Control” on page 3-1 – These instructions specify various types of branches, calls, returns, and loops. Some may also specify a compute operation and/or a data move.
- “Immediate Move Instructions” on page 4-1 – These instructions use immediate instruction fields as operators for addressing.
- “Miscellaneous Operations” on page 5-1 – These instructions include bit modify, bit test, no operation, idle, and cache manipulation.
- “Computations Reference” on page 6-1 – This chapter describes computation and multifunction computation operations that are available within many instructions’ opcodes through a COMPUTE field that specifies a compute operation using the ALU, multiplier, or shifter.

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 25.
These types correspond to the opcodes that ADSP-21160 DSPs recognize, but are for reference only and have no bearing on programming.

Some instructions have more than one syntactical form; for example, instruction “Type 4: Compute, dreg«···»DM | PM, data modify” on page 2-14 has four distinct forms.

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

If COND compute, |DM(Ia,Mb)| = ureg:

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

What’s New in This Manual

This manual is Revision 4.1 of ADSP-21160 SHARC DSP Instruction Set Reference. This revision corrects minor typographical errors and the following issues:

- Active low signals represented correctly in equations for ALU conditions in Chapter 1, “Instruction Summary”.

- AU flag removed and descriptions of the AV and AI flags corrected for the Rn = MANT Fx instruction in Chapter 6, “Computations Reference”.
Technical Support

You can reach Analog Devices processors and DSP technical support in the following ways:

- Post your questions in the processors and DSP support community at EngineerZone®:
  http://ez.analog.com/community/dsp

- Submit your questions to technical support directly at:
  http://www.analog.com/support

- E-mail your questions about processors, DSPs, and tools development software from CrossCore® Embedded Studio or VisualDSP++®:
  Choose Help > Email Support. This creates an e-mail to processor.tools.support@analog.com and automatically attaches your CrossCore Embedded Studio or VisualDSP++ version information and license.dat file.

- E-mail your questions about processors and processor applications to:
  processor.support@analog.com or processor.china@analog.com (Greater China support)

- In the USA only, call 1-800-ANALOGD (1-800-262-5643)

- Contact your Analog Devices sales office or authorized distributor. Locate one at:
  www.analog.com/adi-sales
Send questions by mail to:
Processors and DSP Technical Support
Analog Devices, Inc.
Three Technology Way
P.O. Box 9106
Norwood, MA 02062-9106
USA

Supported Processors

The name “SHARC” refers to a family of high-performance, floating-point embedded processors. Refer to the CCES or VisualDSP++ online help for a complete list of supported processors.

Product Information

Product information can be obtained from the Analog Devices Web site and the CCES or VisualDSP++ online help.
Product Information

Analog Devices Web Site


To access a complete technical library for each processor family, go to http://www.analog.com/processors/technical_library. The manuals selection opens a list of current manuals related to the product as well as a link to the previous revisions of the manuals. When locating your manual title, note a possible errata check mark next to the title that leads to the current correction report against the manual.

Also note, myAnalog is a free feature of the Analog Devices Web site that allows customization of a Web page to display only the latest information about products you are interested in. You can choose to receive weekly e-mail notifications containing updates to the Web pages that meet your interests, including documentation errata against all manuals. myAnalog provides access to books, application notes, data sheets, code examples, and more.

Visit myAnalog to sign up. If you are a registered user, just log on. Your user name is your e-mail address.

EngineerZone

EngineerZone is a technical support forum from Analog Devices, Inc. It allows you direct access to ADI technical support engineers. You can search FAQs and technical information to get quick answers to your embedded processing and DSP design questions.

Use EngineerZone to connect with other DSP developers who face similar design challenges. You can also use this open forum to share knowledge and collaborate with the ADI support team and your peers. Visit http://ez.analog.com to sign up.
# Notation Conventions

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

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File &gt; Close</td>
<td>Titles in reference sections indicate the location of an item within the IDE environment’s menu system (for example, the Close command appears on the File menu).</td>
</tr>
<tr>
<td>{this</td>
<td>that}</td>
</tr>
<tr>
<td>[this</td>
<td>that]</td>
</tr>
<tr>
<td>[this,...]</td>
<td>Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipsis; read the example as an optional comma-separated list of this.</td>
</tr>
<tr>
<td>.SECTION</td>
<td>Commands, directives, keywords, and feature names are in text with letter gothic font.</td>
</tr>
<tr>
<td>filename</td>
<td>Non-keyword placeholders appear in text with italic style format.</td>
</tr>
</tbody>
</table>
| ![Note](image) | **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](image) | **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](image) | **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. |
Register Diagram Conventions

Register diagrams use the following conventions:

- The descriptive name of the register appears at the top, followed by the short form of the name in parentheses.
- If the register is read-only (RO), write-1-to-set (W1S), or write-1-to-clear (W1C), this information appears under the name. Read/write is the default and is not noted. Additional descriptive text may follow.
- If any bits in the register do not follow the overall read/write convention, this is noted in the bit description after the bit name.
- If a bit has a short name, the short name appears first in the bit description, followed by the long name in parentheses.
- The reset value appears in binary in the individual bits and in hexadecimal to the right of the register.
- Bits marked x have an unknown reset value. Consequently, the reset value of registers that contain such bits is undefined or dependent on pin values at reset.
- Shaded bits are reserved.

To ensure upward compatibility with future implementations, write back the value that is read for reserved bits in a register, unless otherwise specified.
The following figure shows an example of these conventions.

### Timer Configuration Registers (TIMERx_CONFIG)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-1</td>
<td>ERR_TYP[1:0] (Error Type) - RO</td>
</tr>
<tr>
<td>00</td>
<td>No error</td>
</tr>
<tr>
<td>01</td>
<td>Counter overflow error</td>
</tr>
<tr>
<td>10</td>
<td>Period register programming error</td>
</tr>
<tr>
<td>11</td>
<td>Pulse width register programming error</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>14-0</td>
<td>TMODE[1:0] (Timer Mode)</td>
</tr>
<tr>
<td>00</td>
<td>Reset state - unused</td>
</tr>
<tr>
<td>01</td>
<td>PWM_OUT mode</td>
</tr>
<tr>
<td>10</td>
<td>WDTH_CAP mode</td>
</tr>
<tr>
<td>11</td>
<td>EXT_CLK mode</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>TOGGLE_HI (PWM_OUT PULSE_HI Toggle Mode)</td>
</tr>
<tr>
<td>0</td>
<td>Negative action pulse</td>
</tr>
<tr>
<td>1</td>
<td>Positive action pulse</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>PERIOD_CNT (Period Count)</td>
</tr>
<tr>
<td>0</td>
<td>Count to end of width</td>
</tr>
<tr>
<td>1</td>
<td>Count to end of period</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>IRQ_ENA (Interrupt Request Enable)</td>
</tr>
<tr>
<td>0</td>
<td>Interrupt request disable</td>
</tr>
<tr>
<td>1</td>
<td>Interrupt request enable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>TIN_SEL (Timer Input Select)</td>
</tr>
<tr>
<td>0</td>
<td>Sample TMRx pin or PF1 pin</td>
</tr>
<tr>
<td>1</td>
<td>Sample UART RX pin or PPI_CLK pin</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>OUT_DIS (Output Pad Disable)</td>
</tr>
<tr>
<td>0</td>
<td>Enable pad in PWM_OUT mode</td>
</tr>
<tr>
<td>1</td>
<td>Disable pad in PWM_OUT mode</td>
</tr>
</tbody>
</table>

**Figure 1. Register Diagram Example**
Register Diagram Conventions
1 INSTRUCTION SUMMARY

This instruction set summary provides a syntax summary for each instruction and includes a cross reference to each instruction’s reference page.

Chapter Overview

The following summary topics appear in this chapter.

- “Development Tools” on page 1-2
- “Compute and Move/Modify Summary” on page 1-3
- “Program Flow Control Summary” on page 1-5
- “Immediate Move Summary” on page 1-7
- “Miscellaneous Operations Summary” on page 1-8
- “Register Types Summary” on page 1-10
- “Memory Addressing Summary” on page 1-16
- “Instruction Set Notation Summary” on page 1-17
- “Conditional Execution Codes Summary” on page 1-20
- “SISD/SIMD Conditional Testing Summary” on page 1-22
- “Instruction Opcode Acronym Summary” on page 1-23
Development Tools

The processor is supported by a complete set of software and hardware development tools, including Analog Devices’ emulators and the Cross-Core Embedded Studio or VisualDSP++ development environment. (The emulator hardware that supports other Analog Devices processors also emulates the processor.)

The development environments support advanced application code development and debug with features such as:

- Create, compile, assemble, and link application programs written in C++, C, and assembly
- Load, run, step, halt, and set breakpoints in application programs
- Read and write data and program memory
- Read and write core and peripheral registers
- Plot memory

Analog Devices DSP emulators use the IEEE 1149.1 JTAG test access port to monitor and control the target board processor during emulation. The emulator provides full speed emulation, allowing inspection and modification of memory, registers, and processor stacks. Nonintrusive in-circuit emulation is assured by the use of the processor JTAG interface—the emulator does not affect target system loading or timing.
Software tools also include Board Support Packages (BSPs). Hardware tools also include standalone evaluation systems (boards and extenders). In addition to the software and hardware development tools available from Analog Devices, third parties provide a wide range of tools supporting the Blackfin processors. Third party software tools include DSP libraries, real-time operating systems, and block diagram design tools.

**Compute and Move/Modify Summary**

Compute and move/modify instructions are classed as Group I instructions, and they provide math, conditional, memory/register access services. The series of tables that follow summarize the Group I instructions. For a complete description of these instructions, see the noted pages.

“Type 1: Compute, Dreg«⋯»DM | Dreg«⋯»PM” on page 2-3

```
compute , DM(Ia, Mb) = dreg1
, dreg1 = DM(Ia, Mb)
, PM(Ic, Md) = dreg2
, dreg2 = PM(Ic, Md)
```

“Type 2: Compute” on page 2-7

```
IF COND compute ;
```
Compute and Move/Modify Summary

“Type 3: Compute, ureg«⋅⋅⋅»DM | PM, register modify” on page 2-9

IF COND compute
, DM(Ia, Mb) = ureg (LW);
, PM(Ic, Md)

, DM(Mb, Ia) = ureg (LW);
, PM(Md, Ic)

, ureg = DM(Ia, Mb) (LW);
, PM(Ic, Md) (LW);

, ureg = DM(Mb, Ia) (LW);
, PM(Md, Ic) (LW);

“Type 4: Compute, dreg«⋅⋅⋅»DM | PM, data modify” on page 2-14

IF COND compute
, DM(Ia, <data6>) = dreg;
, PM(Ic, <data6>)

, DM(<data6>, Ia) = dreg;
, PM(<data6>, Ic)

, dreg = DM(Ia, <data6>) ;
, PM(Ic, <data6>) ;

, dreg = DM(<data6>, Ia) ;
, PM(<data6>, Ic) ;
“Type 5: Compute, ureg«···»ureg | Xdreg<->Ydreg” on page 2-19

IF COND compute, ureg1 = ureg2 ;

X dreg <-> Y dreg

“Type 6: Immediate Shift, dreg«···»DM | PM” on page 2-23

IF COND shiftimm , DM(Ia, Mb) = dreg ;

, PM(Ic, Md)

, dreg = DM(Ia, Mb) ;

PM(Ic, Md) ;

“Type 7: Compute, modify” on page 2-28

IF COND compute , MODIFY (Ia, Mb) ;

(Ic, Md) ;

Program Flow Control Summary

Program flow control instructions are classed as Group II instructions, and they let you control program execution flow. The series of tables that follow summarize the Group II instructions. For a complete description of these instructions, see the noted pages.
“Type 8: Direct Jump | Call” on page 3-3

IF COND JUMP

\[
\begin{align*}
\text{<addr24>} & \quad \text{DB} \\
\text{(PC, <reladdr24>)} & \quad \text{LA} \\
\text{} & \quad \text{CI} \\
\text{} & \quad \text{DB, LA} \\
\text{} & \quad \text{DB, CI} \\
\end{align*}
\]

IF COND CALL

\[
\begin{align*}
\text{<addr24>} & \quad \text{DB} \\
\text{(PC, <reladdr24>)} & \quad \text{LA} \\
\text{} & \quad \text{CI} \\
\text{} & \quad \text{DB, LA} \\
\text{} & \quad \text{DB, CI} \\
\end{align*}
\]

“Type 9: Indirect Jump | Call, Compute” on page 3-8

IF COND JUMP

\[
\begin{align*}
\text{(Md, Ic)} & \quad \text{DB} \\
\text{(PC, <reladdr6>)} & \quad \text{LA} \\
\text{} & \quad \text{CI} \\
\text{} & \quad \text{DB, LA} \\
\text{} & \quad \text{DB, CI} \\
\end{align*}
, compute
, ELSE compute

IF COND CALL

\[
\begin{align*}
\text{(Md, Ic)} & \quad \text{DB} \\
\text{(PC, <reladdr6>)} & \quad \text{LA} \\
\text{} & \quad \text{CI} \\
\text{} & \quad \text{DB, LA} \\
\text{} & \quad \text{DB, CI} \\
\end{align*}
, compute
, ELSE compute

“Type 10: Indirect Jump | Compute, dreg<---DM” on page 3-14

IF COND Jump

\[
\begin{align*}
\text{(Md, Ic)} & \quad \text{DB} \\
\text{(PC, <reladdr6>)} & \quad \text{LA} \\
\text{} & \quad \text{CI} \\
\text{} & \quad \text{DB, LA} \\
\text{} & \quad \text{DB, CI} \\
\end{align*}
,Else
\begin{align*}
\text{compute, DM(1a, Mb) = dreg} \\
\text{compute, dreg = DM(1a, Mb)}
\end{align*}
\]
“Type 11: Return From Subroutine | Interrupt, Compute” on page 3-19

IF COND RTS

| (DB) | compute | ; |
| (LR) | , ELSE compute |
| (DB, LR) |

IF COND RTI

| (DB) | compute | ; |
| , ELSE compute |

“Type 12: Do Until Counter Expired” on page 3-24

LCNTR = <data16>, DO

| ureg | <addr24> | UNTIL LCE; |
| (PC, <reladdr24>) |

“Type 13: Do Until” on page 3-26

DO

| <addr24> | UNTIL termination ; |
| (PC, <reladdr24>) |

Immediate Move Summary

Immediate move instructions are classed as Group III instructions, and they provide memory/register access services. The series of tables that follow summarize the Group III instructions. For a complete description of these instructions, see the noted pages.
“Type 14: Ureg«···»DM | PM (direct addressing)” on page 4-2

\[
\begin{align*}
\text{DM}(\text{<addr32>}) & = \text{ureg (LW)}; \\
\text{PM}(\text{<addr32>}) & \\
\text{ureg} & = \\
\text{DM}(\text{<addr32>}) (\text{LW}); \\
\text{PM}(\text{<addr32>}) (\text{LW});
\end{align*}
\]

“Type 15: Ureg«···»DM | PM (indirect addressing)” on page 4-5

\[
\begin{align*}
\text{DM}(\text{<data32>, Ia}) & = \text{ureg (LW)}; \\
\text{PM}(\text{<data32>, Ic}) & \\
\text{ureg} & = \\
\text{DM}(\text{<data32>, Ia}) (\text{LW}); \\
\text{PM}(\text{<data32>, Ic})
\end{align*}
\]

“Type 16: Immediate data···»DM | PM” on page 4-9

\[
\begin{align*}
\text{DM}(&, \text{Mb}) & = \text{<data32> } ; \\
\text{PM}(&, \text{Md}) & \\
\text{ureg} & = \text{<data32> } ;
\end{align*}
\]

“Type 17: Immediate data···»Ureg” on page 4-12

\[
\begin{align*}
\text{ureg} & = \text{<data32> } ;
\end{align*}
\]

Miscellaneous operations are classed as Group IV instructions, and they provide system register, bit manipulation, and low power services. The
series of tables that follow summarize the Group IV instructions. For a complete description of these instructions, see the noted pages.

“Type 18: System Register Bit Manipulation” on page 5-2

BIT

SET sreg <data32> ;
CLR
TGL
TST
XOR

“Type 19: I Register Modify | Bit-Reverse” on page 5-6

MODIFY

(Ia, <data32>) ;
(Ic, <data32>)

BITREV

(Ia, <data32>) ;
(Ic, <data32>)

“Type 20: Push, Pop Stacks, Flush Cache” on page 5-9

PUSH LOOP

PUSH STS

PUSH PCSTK, FLUSH CACHE ;

POP

“Type 21: Nop” on page 5-11

NOP ;
Register Types Summary

“Type 22: Idle” on page 5-12

IDLE ;

“Type 25: Cjump/Rframe” on page 5-13

CJUMP function (DB) ;

(PC, <reladdr24>)

RFRAME ;

Register Types Summary

Table 1-1 and Table 1-2 list ADSP-21160 DSP registers. The registers in Table 1-1 are in the core processor portion of the processor. The registers in Table 1-2 are in the integrated I/O processor and external port sections of the DSP.
### Table 1-1. Universal Registers (Ureg)

<table>
<thead>
<tr>
<th>Register Type</th>
<th>Register(s)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register File (ureg &amp; dreg)</td>
<td>R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15</td>
<td>Processing element X register file locations, fixed-point</td>
</tr>
<tr>
<td></td>
<td>F0, F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, F13, F14, F15</td>
<td>Processing element X register file locations, floating-point</td>
</tr>
<tr>
<td></td>
<td>S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15</td>
<td>Processing element Y register file locations, fixed-point</td>
</tr>
<tr>
<td></td>
<td>SF0, SF1, SF2, SF3, SF4, SF5, SF6, SF7, SF8, SF9, SF10, SF11, SF12, SF13, SF14, SF15</td>
<td>Processing element Y register file locations, floating-point</td>
</tr>
<tr>
<td>Program Sequencer</td>
<td>PC</td>
<td>Program counter (read-only)</td>
</tr>
<tr>
<td></td>
<td>PCSTK</td>
<td>Top of PC stack</td>
</tr>
<tr>
<td></td>
<td>PCSTKP</td>
<td>PC stack pointer</td>
</tr>
<tr>
<td></td>
<td>FADDR</td>
<td>Fetch address (read-only)</td>
</tr>
<tr>
<td></td>
<td>DADDR</td>
<td>Decode address (read-only)</td>
</tr>
<tr>
<td></td>
<td>LADDR</td>
<td>Loop termination address, code; top of loop address stack</td>
</tr>
<tr>
<td></td>
<td>CURLCNTR</td>
<td>Current loop counter; top of loop count stack</td>
</tr>
<tr>
<td></td>
<td>LCNTR</td>
<td>Loop count for next nested counter-controlled loop</td>
</tr>
</tbody>
</table>
Table 1-1. Universal Registers (Ureg) (Cont’d)

<table>
<thead>
<tr>
<th>Register Type</th>
<th>Register(s)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Address Generators</td>
<td>I0, I1, I2, I3, I4, I5, I6, I7</td>
<td>DAG1 index registers</td>
</tr>
<tr>
<td></td>
<td>M0, M1, M2, M3, M4, M5, M6, M7</td>
<td>DAG1 modify registers</td>
</tr>
<tr>
<td></td>
<td>L0, L1, L2, L3, L4, L5, L6, L7</td>
<td>DAG1 length registers</td>
</tr>
<tr>
<td></td>
<td>B0, B1, B2, B3, B4, B5, B6, B7</td>
<td>DAG1 base registers</td>
</tr>
<tr>
<td></td>
<td>I8, I9, I10, I11, I12, I13, I14, I15</td>
<td>DAG2 index registers</td>
</tr>
<tr>
<td></td>
<td>M8, M9, M10, M11, M12, M13, M14, M15</td>
<td>DAG2 modify registers</td>
</tr>
<tr>
<td></td>
<td>L8, L9, L10, L11, L12, L13, L14, L15</td>
<td>DAG2 length registers</td>
</tr>
<tr>
<td></td>
<td>B8, B9, B10, B11, B12, B13, B14, B15</td>
<td>DAG2 base registers</td>
</tr>
<tr>
<td>Bus Exchange</td>
<td>PX1</td>
<td>PMD-DMD bus exchange 1 (32 bits)</td>
</tr>
<tr>
<td></td>
<td>PX2</td>
<td>PMD-DMD bus exchange 2 (32 bits)</td>
</tr>
<tr>
<td></td>
<td>PX</td>
<td>64-bit combination of PX1 and PX2</td>
</tr>
<tr>
<td>Timer</td>
<td>TPERIOD</td>
<td>Timer period</td>
</tr>
<tr>
<td></td>
<td>TCOUNT</td>
<td>Timer counter</td>
</tr>
</tbody>
</table>
### System Registers (sreg & ureg)

- **MODE1**: Mode control & status
- **MODE2**: Mode control & status
- **IRPTL**: Interrupt latch
- **IMASK**: Interrupt mask
- **IMASKP**: Interrupt mask pointer (for nesting)
- **MMASK**: Mode mask
- **FLAGS**: Flag pins input/output state
- **LIRPTL**: Link Port interrupt latch, mask, and pointer
- **ASTATx**: Element x arithmetic status flags, bit test flag, etc.
- **ASTATy**: Element y arithmetic status flags, bit test flag, etc.
- **STKYx**: Element x sticky arithmetic status flags, stack status flags, etc.
- **STKYy**: Element y sticky arithmetic status flags, stack status flags, etc.
- **USTAT1**: User status register 1
- **USTAT2**: User status register 2
- **USTAT3**: User status register 3
- **USTAT4**: User status register 4

---

**Table 1-1. Universal Registers (Ureg) (Cont’d)**

<table>
<thead>
<tr>
<th>Register Type</th>
<th>Register(s)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Registers</td>
<td>MODE1</td>
<td>Mode control &amp; status</td>
</tr>
<tr>
<td></td>
<td>MODE2</td>
<td>Mode control &amp; status</td>
</tr>
<tr>
<td></td>
<td>IRPTL</td>
<td>Interrupt latch</td>
</tr>
<tr>
<td></td>
<td>IMASK</td>
<td>Interrupt mask</td>
</tr>
<tr>
<td></td>
<td>IMASKP</td>
<td>Interrupt mask pointer (for nesting)</td>
</tr>
<tr>
<td></td>
<td>MMASK</td>
<td>Mode mask</td>
</tr>
<tr>
<td></td>
<td>FLAGS</td>
<td>Flag pins input/output state</td>
</tr>
<tr>
<td></td>
<td>LIRPTL</td>
<td>Link Port interrupt latch, mask, and pointer</td>
</tr>
<tr>
<td></td>
<td>ASTATx</td>
<td>Element x arithmetic status flags, bit test flag, etc.</td>
</tr>
<tr>
<td></td>
<td>ASTATy</td>
<td>Element y arithmetic status flags, bit test flag, etc.</td>
</tr>
<tr>
<td></td>
<td>STKYx</td>
<td>Element x sticky arithmetic status flags, stack status flags, etc.</td>
</tr>
<tr>
<td></td>
<td>STKYy</td>
<td>Element y sticky arithmetic status flags, stack status flags, etc.</td>
</tr>
<tr>
<td></td>
<td>USTAT1</td>
<td>User status register 1</td>
</tr>
<tr>
<td></td>
<td>USTAT2</td>
<td>User status register 2</td>
</tr>
<tr>
<td></td>
<td>USTAT3</td>
<td>User status register 3</td>
</tr>
<tr>
<td></td>
<td>USTAT4</td>
<td>User status register 4</td>
</tr>
</tbody>
</table>
Table 1-2. I/O and Multiplier Registers

<table>
<thead>
<tr>
<th>Register Type</th>
<th>Register(s)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOP registers (system control)</td>
<td>SYSCON</td>
<td>System control</td>
</tr>
<tr>
<td></td>
<td>SYSTAT</td>
<td>System status</td>
</tr>
<tr>
<td></td>
<td>WAIT</td>
<td>Memory wait states</td>
</tr>
<tr>
<td></td>
<td>VIRPT</td>
<td>Multiprocessor IRQ</td>
</tr>
<tr>
<td>IOP registers (system control)</td>
<td>MSGR0, MSGR1, MSGR2, MSGR3, MSGR4, MSGR5, MSGR6, MSGR7</td>
<td>Message registers</td>
</tr>
<tr>
<td></td>
<td>BMAX</td>
<td>Bus timeout max</td>
</tr>
<tr>
<td></td>
<td>BCNT</td>
<td>Bus timeout count</td>
</tr>
<tr>
<td></td>
<td>ELAST</td>
<td>External address last</td>
</tr>
<tr>
<td>IOP registers (DMA)</td>
<td>EPB0, EPB1, EPB2, EPB3</td>
<td>External port FIFO buffers</td>
</tr>
<tr>
<td></td>
<td>DMAC10, DMAC11, DMAC12, DMAC13</td>
<td>DMA controls (EPB0-3)</td>
</tr>
<tr>
<td></td>
<td>DMASTAT</td>
<td>DMA status</td>
</tr>
<tr>
<td></td>
<td>II0, IM0, C0, CP0, GP0, DB0, DA0</td>
<td>DMA 0 parameters (SPORT0 RX)</td>
</tr>
<tr>
<td></td>
<td>II1, IM1, C1, CP1, GP1, DB1, DA1</td>
<td>DMA 1 parameters (SPORT1 RX)</td>
</tr>
<tr>
<td></td>
<td>II2, IM2, C2, CP2, GP2, DB2, DA2</td>
<td>DMA 2 parameters (SPORT0 TX)</td>
</tr>
<tr>
<td></td>
<td>II3, IM3, C3, CP3, GP3, DB3, DA3</td>
<td>DMA 3 parameters (SPORT1 TX)</td>
</tr>
</tbody>
</table>
## Table 1-2. I/O and Multiplier Registers (Cont’d)

<table>
<thead>
<tr>
<th>Register Type</th>
<th>Register(s)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOP registers</td>
<td>II4, IM4, C4, CP4, GP4, DB4, DA4</td>
<td>DMA 4 parameters (LBUF0)</td>
</tr>
<tr>
<td></td>
<td>II5, IM5, C5, CP5, GP5, DB5, DA5</td>
<td>DMA 5 parameters (LBUF1)</td>
</tr>
<tr>
<td></td>
<td>II6, IM6, C6, CP6, GP6, DB6, DA6</td>
<td>DMA 6 parameters (LBUF2)</td>
</tr>
<tr>
<td></td>
<td>II7, IM7, C7, CP7, GP7, DB7, DA7</td>
<td>DMA 7 parameters (LBUF3)</td>
</tr>
<tr>
<td></td>
<td>II8, IM8, C8, CP8, GP8, DB8, DA8</td>
<td>DMA 8 parameters (LBUF4)</td>
</tr>
<tr>
<td></td>
<td>II9, IM9, C9, CP9, GP9, DB9, DA9</td>
<td>DMA 9 parameters (LBUF5)</td>
</tr>
<tr>
<td></td>
<td>II10, IM10, C10, CP10, GP10, EI10, EM10, EC10</td>
<td>DMA 10 parameters (EPB0)</td>
</tr>
<tr>
<td></td>
<td>II11, IM11, C11, CP11, GP11, EI11, EM11, EC11</td>
<td>DMA 11 parameters (EPB1)</td>
</tr>
<tr>
<td></td>
<td>II12, IM12, C12, CP12, GP12, EI12, EM12, EC12</td>
<td>DMA 12 parameters (EPB2)</td>
</tr>
<tr>
<td></td>
<td>II13, IM13, C13, CP13, GP13, EI13, EM13, EC13</td>
<td>DMA 7 parameters (EPB3)</td>
</tr>
<tr>
<td>IOP registers</td>
<td>LBUF0, LBUF1, LBUF2, LBUF3, LBUF4, LBUF5</td>
<td>Link port buffers</td>
</tr>
<tr>
<td>(Link ports)</td>
<td>LCTL0, LCTL1</td>
<td>Link buffer control</td>
</tr>
<tr>
<td></td>
<td>LCOM</td>
<td>Link common control</td>
</tr>
<tr>
<td></td>
<td>LAR</td>
<td>Link assignment</td>
</tr>
<tr>
<td></td>
<td>LSRQ</td>
<td>Link service request</td>
</tr>
<tr>
<td></td>
<td>LPATH1, LPATH2, LPATH3</td>
<td>Link path (mesh)</td>
</tr>
<tr>
<td></td>
<td>LPCNT</td>
<td>Link path count (mesh)</td>
</tr>
<tr>
<td></td>
<td>CNST1, CNST2</td>
<td>Link constant (mesh)</td>
</tr>
</tbody>
</table>
Memory Addressing Summary

Table 1-2. I/O and Multiplier Registers (Cont’d)

<table>
<thead>
<tr>
<th>Register Type</th>
<th>Register(s)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOP registers</td>
<td>STCTL0, SRCTL0, TX0, RX0, TDIV0, RDIV0, MTCS0, MRCS0, MTCCS0, MRCCS0, SATH0, KEYWD0, KEYMASK0</td>
<td>SPORT 0 registers</td>
</tr>
<tr>
<td></td>
<td>STCTL1, SRCTL1, TX1, RX1, TDIV1, RDIV1, MTCS, MRCS1, MTCCS1, MRCCS1, SATH1, KEYWD1, KEYMASK1</td>
<td>SPORT 1 registers</td>
</tr>
<tr>
<td>Multiplier registers</td>
<td>MR, MR0, MR1, MR2,</td>
<td>Multiplier results</td>
</tr>
<tr>
<td></td>
<td>MRF, MR0F, MR1F, MR2F</td>
<td>Multiplier results, foreground</td>
</tr>
<tr>
<td></td>
<td>MRB, MR0B, MR1B, MR2B</td>
<td>Multiplier results, background</td>
</tr>
</tbody>
</table>

Memory Addressing Summary

ADSP-21160 processors support the following types of addressing.

Direct Addressing

- **Absolute address** (Instruction Types 8, 12, 13, 14)

  \[
  \text{dm}(0x000015F0) = \text{astat};
  \]

  if ne jump label2;   \{'label2' is an address label}\]

- **PC-relative address** (Instruction Types 8, 9, 10, 12, 13)

  \[
  \text{call}(pc,10), \text{r0}=\text{r6}+\text{r3};
  \]

  \[
  \text{do}(pc,\text{length}) \text{ until sz}; \quad \{'\text{length}' \text{ is a variable}\} \]
Indirect Addressing (using DAG registers):

- **Post-modify with M register, update I register** (Instruction Types 1, 3, 6, 16)
  
  \[ f_{5} = pm(i_{9}, m_{12}); \]
  \[ dm(i_{0}, m_{3}) = r_{3}, r_{1} = pm(i_{15}, m_{10}); \]

- **Pre-modify with M register, no update** (Instruction Types 3, 9, 10)
  
  \[ r_{1} = pm(m_{10}, i_{15}); \]
  \[ jum p(m_{13}, i_{11}); \]

- **Post-modify with immediate value, update I register** (Instruction Type 4)
  
  \[ f_{15} = dm(i_{0}, 6); \]
  \[ if \ av \ r_{1} = pm(i_{15}, 0x11); \]

- **Pre-modify with immediate value, no update** (Instruction Types 4, 15)
  
  \[ if \ av \ r_{1} = pm(0x11, i_{15}); \]
  \[ dm(127, i_{5}) = laddr; \]

**Instruction Set Notation Summary**

The conventions for ADSP-210xx instruction syntax descriptions appear in Table 1-3 on page 1-18. Other parts of the instruction syntax and opcode information also appear in this section.
Table 1-3. Instruction Set Notation

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Explicit syntax— assembler keyword (notation only; assembler is case-insensitive and lowercase is the preferred programming convention)</td>
</tr>
<tr>
<td>;</td>
<td>Semicolon (instruction terminator)</td>
</tr>
<tr>
<td>,</td>
<td>Comma (separates parallel operations in an instruction)</td>
</tr>
<tr>
<td>italics</td>
<td>Optional part of instruction</td>
</tr>
<tr>
<td></td>
<td>List of options between vertical bars (choose one)</td>
</tr>
<tr>
<td>compute</td>
<td>ALU, multiplier, shifter or multifunction operation (see the chapter “Computations Reference”).</td>
</tr>
<tr>
<td>shiftimm</td>
<td>Shifter immediate operation (see the chapter “Computations Reference”).</td>
</tr>
<tr>
<td>cond</td>
<td>Status condition (see condition codes in Table 1-4 on page 1-20)</td>
</tr>
<tr>
<td>termination</td>
<td>Loop termination condition (see condition codes in Table 1-4 on page 1-20)</td>
</tr>
<tr>
<td>ureg</td>
<td>Universal register</td>
</tr>
<tr>
<td>cureg</td>
<td>Complementary universal register (see Table 1-10 on page 1-30)</td>
</tr>
<tr>
<td>sreg</td>
<td>System register</td>
</tr>
<tr>
<td>csreg</td>
<td>Complementary system register (see Table 1-10 on page 1-30)</td>
</tr>
<tr>
<td>dreg</td>
<td>Data register (register file): R15-R0 or F15-F0</td>
</tr>
<tr>
<td>cdreg</td>
<td>Complementary data register (register file): R15-R0 or F15-F0 (see Table 1-10 on page 1-30)</td>
</tr>
<tr>
<td>creg</td>
<td>One of 32 cache entries, an entry consisting of a CH, CL, &amp; CA</td>
</tr>
<tr>
<td>Ia</td>
<td>I7-I0 (DAG1 index register)</td>
</tr>
<tr>
<td>Mb</td>
<td>M7-M0 (DAG1 modify register)</td>
</tr>
<tr>
<td>Ic</td>
<td>I15-I8 (DAG2 index register)</td>
</tr>
<tr>
<td>Md</td>
<td>M15-M8 (DAG2 modify register)</td>
</tr>
<tr>
<td>&lt;datan&gt;</td>
<td>n-bit immediate data value</td>
</tr>
<tr>
<td>Notation</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td>&lt;addrn&gt;</td>
<td>n-bit immediate address value</td>
</tr>
<tr>
<td>&lt;reladdrn&gt;</td>
<td>n-bit immediate PC-relative address value</td>
</tr>
<tr>
<td>+1</td>
<td>the incremented data, address or register value</td>
</tr>
<tr>
<td>(DB)</td>
<td>Delayed branch</td>
</tr>
<tr>
<td>(LA)</td>
<td>Loop abort (pop loop and PC stacks on branch)</td>
</tr>
<tr>
<td>(CI)</td>
<td>Clear interrupt</td>
</tr>
<tr>
<td>(LR)</td>
<td>Loop reentry</td>
</tr>
<tr>
<td>(LW)</td>
<td>Long Word (forces Long word access in Normal word range)</td>
</tr>
</tbody>
</table>
Conditional Execution Codes Summary

In a conditional instruction, execution of the entire instruction depends on the specified condition (cond or terminate). Table 1-4 lists the codes that you can use in conditionals (IF and DO UNTIL).

Table 1-4. IF Condition and Do/Until Termination Mnemonics

<table>
<thead>
<tr>
<th>Condition From</th>
<th>Description</th>
<th>True if…</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>ALU = 0</td>
<td>AZ = 1</td>
<td>EQ</td>
</tr>
<tr>
<td></td>
<td>ALU ≠ 0</td>
<td>AZ = 0</td>
<td>NE</td>
</tr>
<tr>
<td></td>
<td>ALU &gt; 0</td>
<td>footnote¹</td>
<td>GT</td>
</tr>
<tr>
<td></td>
<td>ALU &lt; zero</td>
<td>footnote²</td>
<td>LT</td>
</tr>
<tr>
<td></td>
<td>ALU ≥ 0</td>
<td>footnote³</td>
<td>GE</td>
</tr>
<tr>
<td></td>
<td>ALU ≤ 0</td>
<td>footnote⁴</td>
<td>LE</td>
</tr>
<tr>
<td></td>
<td>ALU carry</td>
<td>AC = 1</td>
<td>AC</td>
</tr>
<tr>
<td></td>
<td>ALU not carry</td>
<td>AC = 0</td>
<td>NOT AC</td>
</tr>
<tr>
<td></td>
<td>ALU overflow</td>
<td>AV = 1</td>
<td>AV</td>
</tr>
<tr>
<td></td>
<td>ALU not overflow</td>
<td>AV = 0</td>
<td>NOT AV</td>
</tr>
<tr>
<td>Multiplier</td>
<td>Multiplier overflow</td>
<td>MV = 1</td>
<td>MV</td>
</tr>
<tr>
<td></td>
<td>Multiplier not overflow</td>
<td>MV = 0</td>
<td>NOT MV</td>
</tr>
<tr>
<td></td>
<td>Multiplier sign</td>
<td>MN = 1</td>
<td>MS</td>
</tr>
<tr>
<td></td>
<td>Multiplier not sign</td>
<td>MN = 0</td>
<td>NOT MS</td>
</tr>
<tr>
<td>Shifter</td>
<td>Shifter overflow</td>
<td>SV = 1</td>
<td>SV</td>
</tr>
<tr>
<td></td>
<td>Shifter not overflow</td>
<td>SV = 0</td>
<td>NOT SV</td>
</tr>
<tr>
<td></td>
<td>Shifter zero</td>
<td>SZ = 1</td>
<td>SZ</td>
</tr>
<tr>
<td></td>
<td>Shifter not zero</td>
<td>SZ = 0</td>
<td>NOT SZ</td>
</tr>
<tr>
<td>Bit Test</td>
<td>Bit test flag true</td>
<td>BTF = 1</td>
<td>TF</td>
</tr>
<tr>
<td></td>
<td>Bit test flag false</td>
<td>BTF = 0</td>
<td>NOT TF</td>
</tr>
</tbody>
</table>
### Table 1-4. IF Condition and Do/Until Termination Mnemonics (Cont’d)

<table>
<thead>
<tr>
<th>Condition From</th>
<th>Description</th>
<th>True if…</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag Input</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Flag0 asserted</td>
<td>F10 = 1</td>
<td>FLAG0_IN</td>
</tr>
<tr>
<td></td>
<td>Flag0 not asserted</td>
<td>F10 = 0</td>
<td>NOT FLAG0_IN</td>
</tr>
<tr>
<td></td>
<td>Flag1 asserted</td>
<td>F11 = 1</td>
<td>FLAG1_IN</td>
</tr>
<tr>
<td></td>
<td>Flag1 not asserted</td>
<td>F11 = 0</td>
<td>NOT FLAG1_IN</td>
</tr>
<tr>
<td></td>
<td>Flag2 asserted</td>
<td>F12 = 1</td>
<td>FLAG2_IN</td>
</tr>
<tr>
<td></td>
<td>Flag2 not asserted</td>
<td>F12 = 0</td>
<td>NOT FLAG2_IN</td>
</tr>
<tr>
<td></td>
<td>Flag3 asserted</td>
<td>F13 = 1</td>
<td>FLAG3_IN</td>
</tr>
<tr>
<td></td>
<td>Flag3 not asserted</td>
<td>F13 = 0</td>
<td>NOT FLAG3_IN</td>
</tr>
<tr>
<td>Mode</td>
<td>Bus master true</td>
<td></td>
<td>BM</td>
</tr>
<tr>
<td></td>
<td>Bus master false</td>
<td></td>
<td>NOT BM</td>
</tr>
<tr>
<td>Sequencer</td>
<td>Loop counter expired (Do)</td>
<td>CURLCNTR = 1</td>
<td>LCE</td>
</tr>
<tr>
<td></td>
<td>Loop counter not expired (If)</td>
<td>CURLCNTR ≠ 1</td>
<td>NOT ICE</td>
</tr>
<tr>
<td></td>
<td>Always false (Do)</td>
<td>Always</td>
<td>FOREVER</td>
</tr>
<tr>
<td></td>
<td>Always true (If)</td>
<td>Always</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

1. ALU greater than (GT) is true if: [$AF$ and ($AN$ xor ($AV$ and $\overline{ALUSAT}$)) or ($AF$ and $AN$)] or $AZ = 0$
2. ALU less than (LT) is true if: [$AF$ and ($AN$ xor ($AV$ and $\overline{ALUSAT}$)) or ($AF$ and $AN$ and $AZ$)] = 1
3. ALU greater equal (GE) is true if: [$AF$ and ($AN$ xor ($AV$ and $\overline{ALUSAT}$)) or ($AF$ and $AN$ and $AZ$)] = 0
4. ALU lesser or equal (LE) is true if: [$AF$ and ($AN$ xor ($AV$ and $\overline{ALUSAT}$)) or ($AF$ and $AN$)] or $AZ = 1$
SISD/SIMD Conditional Testing Summary

The processor handles conditional execution differently in SISD versus SIMD mode. There are three ways that conditionals differ in SIMD mode:

- In conditional computation (If ... Compute) instructions, each processing element executes the computation based on evaluating the condition in that processing element.

- In conditional program control (If ... Jump/Call) instructions, the program sequencer executes the Jump/Call based on a logical AND of the conditions in both processing elements.

- In conditional computation instructions with an Else clause, each processing element executes the Else computation based on evaluating the inverse of the condition (Not Cond) in that processing element.

Table 1-5 on page 1-22 and Table 1-6 on page 1-23 compare SISD and SIMD If-Else conditional execution, which are available in the Type 9, 10, and 11 instructions.

Table 1-5. SISD Mode Conditional Execution

<table>
<thead>
<tr>
<th>Conditional test</th>
<th>ELSE modifier</th>
<th>Results for Type 11 (RTS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (false)</td>
<td>0 (without else)</td>
<td>rts nops, compute nops</td>
</tr>
<tr>
<td>0 (false)</td>
<td>1 (else)</td>
<td>rts nops, compute executes</td>
</tr>
<tr>
<td>1 (true)</td>
<td>0 (without else)</td>
<td>rts executes, compute executes</td>
</tr>
<tr>
<td>1 (true)</td>
<td>1 (else)</td>
<td>rts executes, compute nops</td>
</tr>
</tbody>
</table>
Table 1-6. SIMD Mode Conditional Execution

<table>
<thead>
<tr>
<th>Conditional test</th>
<th>Else modifier</th>
<th>Results for Type 11 (RTS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEx</td>
<td>PEy</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

For more information and examples, see the following instruction reference pages.

- “Type 9: Indirect Jump | Call, Compute” on page 3-8
- “Type 10: Indirect Jump | Compute, dreg«…»DM” on page 3-14
- “Type 11: Return From Subroutine | Interrupt, Compute” on page 3-19

Instruction Opcode Acronym Summary

In ADSP-21160 DSP opcodes, some bits are explicitly defined to be zeros or ones. The values of other bits or fields set various parameters for the instruction. The terms in Table 1-7 define these opcode bits and fields. Unspecified bits are ignored when the processor decodes the instruction, but are reserved for future use.
## Instruction Opcode Acronym Summary

### Table 1-7. Opcode Acronyms

<table>
<thead>
<tr>
<th>Bit/Field</th>
<th>Description</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Loop abort code</td>
<td>0: Do not pop loop, PC stacks on branch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Pop loop, PC stacks on branch</td>
</tr>
<tr>
<td>ADDR</td>
<td>Immediate address field</td>
<td></td>
</tr>
<tr>
<td>AI</td>
<td>Computation unit register</td>
<td>0000 MR0F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0001 MR1F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0010 MR2F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0100 MR0B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0101 MR1B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0110 MR2B</td>
</tr>
<tr>
<td>B</td>
<td>Branch type</td>
<td>0: Jump</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Call</td>
</tr>
<tr>
<td>BOP</td>
<td>Bit Operation select codes</td>
<td>000  Set</td>
</tr>
<tr>
<td></td>
<td></td>
<td>001  Clear</td>
</tr>
<tr>
<td></td>
<td></td>
<td>010  Toggle</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100  Test</td>
</tr>
<tr>
<td></td>
<td></td>
<td>101  XOR</td>
</tr>
<tr>
<td>COMPUTE</td>
<td>Compute operation field (see “Compute operations Reference” on page 6-1)</td>
<td></td>
</tr>
<tr>
<td>COND</td>
<td>Status Condition codes</td>
<td>0–31</td>
</tr>
<tr>
<td>CI</td>
<td>Clear interrupt code</td>
<td>0: Do not clear current interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Clear current interrupt</td>
</tr>
<tr>
<td>CREG</td>
<td>Instruction cache entry</td>
<td>0–31</td>
</tr>
</tbody>
</table>
Table 1-7. Opcode Acronyms (Cont’d)

<table>
<thead>
<tr>
<th>Bit/Field</th>
<th>Description</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>Instruction cache register select code</td>
<td>00  Lower half of instruction RAM entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01  Upper half of instruction RAM entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11  Address CAM entry</td>
</tr>
<tr>
<td>CU</td>
<td>Computation unit select codes</td>
<td>00  ALU</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01  Multiplier</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10  Shifter</td>
</tr>
<tr>
<td>DATA</td>
<td>Immediate data field</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>Counter decrement code</td>
<td>0   No counter decrement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1   Counter decrement</td>
</tr>
<tr>
<td>DMD</td>
<td>Memory access direction</td>
<td>0   Read</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1   Write</td>
</tr>
<tr>
<td>DMI</td>
<td>Index (I) register numbers, DAG1</td>
<td>0–7</td>
</tr>
<tr>
<td>DMM</td>
<td>Modify (M) register numbers, DAG1</td>
<td>0–7</td>
</tr>
<tr>
<td>DREG</td>
<td>Register file locations</td>
<td>0–15</td>
</tr>
<tr>
<td>E</td>
<td>ELSE clause code</td>
<td>0   No ELSE clause</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1   ELSE clause</td>
</tr>
<tr>
<td>FC</td>
<td>Flush cache code</td>
<td>0   No cache flush</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1   Cache flush</td>
</tr>
<tr>
<td>G</td>
<td>DAG/Memory select</td>
<td>0   DAG1 or Data Memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1   DAG2 or Program Memory</td>
</tr>
<tr>
<td>INC</td>
<td>Counter increment code</td>
<td>0   No counter increment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1   Counter increment</td>
</tr>
</tbody>
</table>
Table 1-7. Opcode Acronyms (Cont’d)

<table>
<thead>
<tr>
<th>Bit/Field</th>
<th>Description</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>Jump Type</td>
<td>0: Non-delayed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Delayed</td>
</tr>
<tr>
<td>L</td>
<td>Long Word memory address</td>
<td>0: Access size based on memory map</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Long word (64-bit) access size</td>
</tr>
<tr>
<td>LPO</td>
<td>Loop stack pop code</td>
<td>0: No stack pop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Stack pop</td>
</tr>
<tr>
<td>LPU</td>
<td>Loop stack push code</td>
<td>0: No stack push</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Stack push</td>
</tr>
<tr>
<td>LR</td>
<td>Loop reentry code</td>
<td>0: No loop reentry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Loop reentry</td>
</tr>
<tr>
<td>NUM</td>
<td>Interrupt vector</td>
<td>0–7</td>
</tr>
<tr>
<td>PMD</td>
<td>Memory access direction</td>
<td>0: Read</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Write</td>
</tr>
<tr>
<td>PMI</td>
<td>Index (I) register numbers, DAG2</td>
<td>8–15</td>
</tr>
<tr>
<td>PMM</td>
<td>Modify (M) register numbers, DAG2</td>
<td>8–15</td>
</tr>
<tr>
<td>PPO</td>
<td>PC stack pop code</td>
<td>0: No stack pop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Stack pop</td>
</tr>
<tr>
<td>PPU</td>
<td>PC stack push code</td>
<td>0: No stack push</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Stack push</td>
</tr>
<tr>
<td>RELADDR</td>
<td>PC-relative address field</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>UREG transfer/instruction cache</td>
<td>0: instruction cache read-load</td>
</tr>
<tr>
<td></td>
<td>read-load select</td>
<td>1: ureg transfer</td>
</tr>
</tbody>
</table>
### Table 1-7. Opcode Acronyms (Cont’d)

<table>
<thead>
<tr>
<th>Bit/Field</th>
<th>Description</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPO</td>
<td>Status stack pop code</td>
<td>0: No stack pop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Stack pop</td>
</tr>
<tr>
<td>SPU</td>
<td>Status stack push code</td>
<td>0: No stack push</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Stack push</td>
</tr>
<tr>
<td>SREG</td>
<td>System Register code</td>
<td>0–15 (see “Universal Register Codes” on page 1-28)</td>
</tr>
<tr>
<td>TERM</td>
<td>Termination Condition codes</td>
<td>0–31</td>
</tr>
<tr>
<td>U</td>
<td>Update, index (I) register</td>
<td>0: Pre-modify, no update</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Post-modify with update</td>
</tr>
<tr>
<td>UREG</td>
<td>Universal Register code</td>
<td>0–256 (see “Universal Register Codes” on page 1-28)</td>
</tr>
<tr>
<td>RA, RM, RN, RS, RX, RY</td>
<td>Register file locations for compute operands and results</td>
<td>0–15</td>
</tr>
<tr>
<td>RXA</td>
<td>ALU x-operand register file location for multifunction operations</td>
<td>8–11</td>
</tr>
<tr>
<td>RXM</td>
<td>Multiplier x-operand register file location for multifunction operations</td>
<td>0–3</td>
</tr>
<tr>
<td>RYA</td>
<td>ALU y-operand register file location for multifunction operations</td>
<td>12–15</td>
</tr>
<tr>
<td>RYM</td>
<td>Multiplier y-operand register file location for multifunction operations</td>
<td>4–7</td>
</tr>
</tbody>
</table>
Table 1-8, Table 1-9 on page 1-29, Table 1-10 on page 1-30, and Table 1-11 on page 1-31 in this section list the bit codes for register that appear within opcode fields.

Table 1-8. Universal Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>program counter</td>
</tr>
<tr>
<td>PCSTK</td>
<td>top of PC stack</td>
</tr>
<tr>
<td>PCSTKP</td>
<td>PC stack pointer</td>
</tr>
<tr>
<td>FADDR</td>
<td>fetch address</td>
</tr>
<tr>
<td>DADDR</td>
<td>decode address</td>
</tr>
<tr>
<td>LADDR</td>
<td>loop termination address</td>
</tr>
<tr>
<td>CURLCNTR</td>
<td>current loop counter</td>
</tr>
<tr>
<td>LCNTR</td>
<td>loop counter</td>
</tr>
<tr>
<td>R15–R0</td>
<td>X element register file locations</td>
</tr>
<tr>
<td>S15–S0</td>
<td>Y element register file locations</td>
</tr>
<tr>
<td>I15–I0</td>
<td>DAG1 and DAG2 index registers</td>
</tr>
<tr>
<td>M15–M0</td>
<td>DAG1 and DAG2 modify registers</td>
</tr>
<tr>
<td>L15–L0</td>
<td>DAG1 and DAG2 length registers</td>
</tr>
<tr>
<td>B15–B0</td>
<td>DAG1 and DAG2 base registers</td>
</tr>
<tr>
<td>PX</td>
<td>48-bit PX1 and PX2 combination</td>
</tr>
<tr>
<td>PX1</td>
<td>bus exchange 1 (16 bits)</td>
</tr>
<tr>
<td>PX2</td>
<td>bus exchange 2 (32 bits)</td>
</tr>
<tr>
<td>TPERIOD</td>
<td>timer period</td>
</tr>
<tr>
<td>TCOUNT</td>
<td>timer counter</td>
</tr>
</tbody>
</table>
Table 1-9. Universal and System Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODE1</td>
<td>mode control 1</td>
</tr>
<tr>
<td>MODE2</td>
<td>mode control 2</td>
</tr>
<tr>
<td>IRPTL</td>
<td>interrupt latch</td>
</tr>
<tr>
<td>IMASK</td>
<td>interrupt mask</td>
</tr>
<tr>
<td>IMASKP</td>
<td>interrupt mask pointer</td>
</tr>
<tr>
<td>MMASK</td>
<td>Mode mask</td>
</tr>
<tr>
<td>FLAGS</td>
<td>Flag pins input/output state</td>
</tr>
<tr>
<td>ASTATx</td>
<td>X element arithmetic status</td>
</tr>
<tr>
<td>STKYx</td>
<td>X element sticky status</td>
</tr>
<tr>
<td>ASTATy</td>
<td>Y element arithmetic status</td>
</tr>
<tr>
<td>STKYy</td>
<td>Y element sticky status</td>
</tr>
<tr>
<td>USTAT1</td>
<td>user status reg 1</td>
</tr>
<tr>
<td>USTAT2</td>
<td>user status reg 2</td>
</tr>
<tr>
<td>USTAT3</td>
<td>user status reg 3</td>
</tr>
<tr>
<td>USTAT4</td>
<td>user status reg 4</td>
</tr>
</tbody>
</table>
## Table 1-10. Complementary Registers (Ureg–Cureg)

<table>
<thead>
<tr>
<th>Register Type</th>
<th>SIMD Mode Complementary Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data register (dreg &amp; ureg)</td>
<td>R0–S0</td>
</tr>
<tr>
<td></td>
<td>R1–S1</td>
</tr>
<tr>
<td></td>
<td>R2–S2</td>
</tr>
<tr>
<td></td>
<td>R3–S3</td>
</tr>
<tr>
<td></td>
<td>R4–S4</td>
</tr>
<tr>
<td></td>
<td>R5–S5</td>
</tr>
<tr>
<td></td>
<td>R6–S6</td>
</tr>
<tr>
<td></td>
<td>R7–S7</td>
</tr>
<tr>
<td></td>
<td>R8–S8</td>
</tr>
<tr>
<td></td>
<td>R9–S9</td>
</tr>
<tr>
<td></td>
<td>R10–S10</td>
</tr>
<tr>
<td></td>
<td>R11–S11</td>
</tr>
<tr>
<td></td>
<td>R12–S12</td>
</tr>
<tr>
<td></td>
<td>R13–S13</td>
</tr>
<tr>
<td></td>
<td>R14–S14</td>
</tr>
<tr>
<td></td>
<td>R15–S15</td>
</tr>
<tr>
<td>System register (sreg &amp; ureg)</td>
<td>USTAT1–USTAT2</td>
</tr>
<tr>
<td></td>
<td>USTAT3–USTAT4</td>
</tr>
<tr>
<td></td>
<td>ASTATx–ASTATy</td>
</tr>
<tr>
<td></td>
<td>STKYx–STKYy</td>
</tr>
<tr>
<td>Bus exchange register (ureg)</td>
<td>PX1–PX2</td>
</tr>
</tbody>
</table>
Table 1-11 shows how Ureg register codes appear to PEx.

Table 1-11. Processing Element X Universal Register Codes (SISD/SIMD)

<table>
<thead>
<tr>
<th>Bits: (3210)</th>
<th>Bits: (7654)</th>
<th>0000</th>
<th>0001</th>
<th>0010</th>
<th>0011</th>
<th>0100</th>
<th>0101</th>
<th>0110</th>
<th>0111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>R0</td>
<td>I0</td>
<td>M0</td>
<td>L0</td>
<td>B0</td>
<td>S0</td>
<td>FADDR</td>
<td>USTAT1</td>
</tr>
<tr>
<td>0001</td>
<td>0001</td>
<td>R1</td>
<td>I1</td>
<td>M1</td>
<td>L1</td>
<td>B1</td>
<td>S1</td>
<td>DADDR</td>
<td>USTAT2</td>
</tr>
<tr>
<td>0010</td>
<td>0010</td>
<td>R2</td>
<td>I2</td>
<td>M2</td>
<td>L2</td>
<td>B2</td>
<td>S2</td>
<td>MODE1</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>0011</td>
<td>R3</td>
<td>I3</td>
<td>M3</td>
<td>L3</td>
<td>B3</td>
<td>S3</td>
<td>PC</td>
<td>MMASK</td>
</tr>
<tr>
<td>0100</td>
<td>0100</td>
<td>R4</td>
<td>I4</td>
<td>M4</td>
<td>L4</td>
<td>B4</td>
<td>S4</td>
<td>PCSTK</td>
<td>MODE2</td>
</tr>
<tr>
<td>0101</td>
<td>0101</td>
<td>R5</td>
<td>I5</td>
<td>M5</td>
<td>L5</td>
<td>B5</td>
<td>S5</td>
<td>PCSTKP</td>
<td>FLAGS</td>
</tr>
<tr>
<td>0110</td>
<td>0110</td>
<td>R6</td>
<td>I6</td>
<td>M6</td>
<td>L6</td>
<td>B6</td>
<td>S6</td>
<td>LADDR</td>
<td>ASTATx</td>
</tr>
<tr>
<td>0111</td>
<td>0111</td>
<td>R7</td>
<td>I7</td>
<td>M7</td>
<td>L7</td>
<td>B7</td>
<td>S7</td>
<td>CURLCNT</td>
<td>ASTATy</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>R8</td>
<td>I8</td>
<td>M8</td>
<td>L8</td>
<td>B8</td>
<td>S8</td>
<td>LCNTR</td>
<td>STKYx</td>
</tr>
<tr>
<td>1001</td>
<td>1001</td>
<td>R9</td>
<td>I9</td>
<td>M9</td>
<td>L9</td>
<td>B9</td>
<td>S9</td>
<td>EMUCLK</td>
<td>STKYy</td>
</tr>
<tr>
<td>1010</td>
<td>1010</td>
<td>R10</td>
<td>I10</td>
<td>M10</td>
<td>L10</td>
<td>B10</td>
<td>S10</td>
<td>EMUCLK2</td>
<td>IRPTL</td>
</tr>
<tr>
<td>1011</td>
<td>1011</td>
<td>R11</td>
<td>I11</td>
<td>M11</td>
<td>L11</td>
<td>B11</td>
<td>S11</td>
<td>PX</td>
<td>IMASK</td>
</tr>
<tr>
<td>1100</td>
<td>1100</td>
<td>R12</td>
<td>I12</td>
<td>M12</td>
<td>L12</td>
<td>B12</td>
<td>S12</td>
<td>PX1</td>
<td>IMASKP</td>
</tr>
<tr>
<td>1101</td>
<td>1101</td>
<td>R13</td>
<td>I13</td>
<td>M13</td>
<td>L13</td>
<td>B13</td>
<td>S13</td>
<td>PX2</td>
<td>LRPTL</td>
</tr>
<tr>
<td>1110</td>
<td>1110</td>
<td>R14</td>
<td>I14</td>
<td>M14</td>
<td>L14</td>
<td>B14</td>
<td>S14</td>
<td>TPERIOD</td>
<td>USTAT3</td>
</tr>
<tr>
<td>1111</td>
<td>1111</td>
<td>R15</td>
<td>I15</td>
<td>M15</td>
<td>L15</td>
<td>B15</td>
<td>S15</td>
<td>TCOUNT</td>
<td>USTAT4</td>
</tr>
</tbody>
</table>
Table 1-12 shows how Ureg register codes appear to PEy.

Table 1-12. Processing Element Y Universal Register Codes (SIMD)

<table>
<thead>
<tr>
<th>Bits: 3210</th>
<th>Bits: 7654</th>
<th>0000</th>
<th>0001</th>
<th>0010</th>
<th>0011</th>
<th>0100</th>
<th>0101</th>
<th>0110</th>
<th>0111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>S0</td>
<td>I0</td>
<td>M0</td>
<td>L0</td>
<td>B0</td>
<td>R0</td>
<td>FADDR</td>
<td>USTAT2</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>S1</td>
<td>I1</td>
<td>M1</td>
<td>L1</td>
<td>B1</td>
<td>R1</td>
<td>DADDR</td>
<td>USTAT1</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>S2</td>
<td>I2</td>
<td>M2</td>
<td>L2</td>
<td>B2</td>
<td>R2</td>
<td>MODE1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>S3</td>
<td>I3</td>
<td>M3</td>
<td>L3</td>
<td>B3</td>
<td>R3</td>
<td>PC</td>
<td>MMASK</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>S4</td>
<td>I4</td>
<td>M4</td>
<td>L4</td>
<td>B4</td>
<td>R4</td>
<td>PCSTK</td>
<td>MODE2</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>S5</td>
<td>I5</td>
<td>M5</td>
<td>L5</td>
<td>B5</td>
<td>R5</td>
<td>PCSTKP</td>
<td>FLAGS</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>S6</td>
<td>I6</td>
<td>M6</td>
<td>L6</td>
<td>B6</td>
<td>R6</td>
<td>LADDR</td>
<td>ASTATy</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>S7</td>
<td>I7</td>
<td>M7</td>
<td>L7</td>
<td>B7</td>
<td>R7</td>
<td>CURLCNT</td>
<td>RASTATx</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>S8</td>
<td>I8</td>
<td>M8</td>
<td>L8</td>
<td>B8</td>
<td>R8</td>
<td>LCNTR</td>
<td>STKYy</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>S9</td>
<td>I9</td>
<td>M9</td>
<td>L9</td>
<td>B9</td>
<td>R9</td>
<td>EMUCLK</td>
<td>STKYx</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>S10</td>
<td>I10</td>
<td>M10</td>
<td>L10</td>
<td>B10</td>
<td>R10</td>
<td>EMUCLK2</td>
<td>IRPTL</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>S11</td>
<td>I11</td>
<td>M11</td>
<td>L11</td>
<td>B11</td>
<td>R11</td>
<td>PX</td>
<td>IMASK</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>S12</td>
<td>I12</td>
<td>M12</td>
<td>L12</td>
<td>B12</td>
<td>R12</td>
<td>PX2</td>
<td>IMASKP</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>S13</td>
<td>I13</td>
<td>M13</td>
<td>L13</td>
<td>B13</td>
<td>R13</td>
<td>PX1</td>
<td>LRPTL</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>S14</td>
<td>I14</td>
<td>M14</td>
<td>L14</td>
<td>B14</td>
<td>R14</td>
<td>TPERIOD</td>
<td>USTAT4</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>S15</td>
<td>I15</td>
<td>M15</td>
<td>L15</td>
<td>B15</td>
<td>R15</td>
<td>TCOUNT</td>
<td>USTAT3</td>
<td></td>
</tr>
</tbody>
</table>
## ADSP-21160 Instruction Opcode Map

Table 1-13. ADSP-21160 DSP Opcodes (Bits 47–27)

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>43</th>
<th>42</th>
<th>41</th>
<th>40</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Type 1: Compute, Dreg«···»DM</td>
<td>Dreg«···»PM”</td>
<td>001</td>
<td>D</td>
<td>M</td>
<td>D</td>
<td>DMI</td>
<td>DMM</td>
<td>P</td>
<td>M</td>
<td>D</td>
<td>DM DREG</td>
<td>PMI</td>
<td>PMM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>“Type 2: Compute”</td>
<td>000</td>
<td>00001</td>
<td>COND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>“Type 3: Compute, ureg«···»DM</td>
<td>PM, register modify”</td>
<td>010</td>
<td>U</td>
<td>I</td>
<td>M</td>
<td>COND</td>
<td>G</td>
<td>D</td>
<td>L</td>
<td>UREG</td>
<td>DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>“Type 4: Compute, dreg«···»DM</td>
<td>PM, data modify”</td>
<td>011</td>
<td>0</td>
<td>I</td>
<td>G</td>
<td>D</td>
<td>U</td>
<td>COND</td>
<td>DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a) “Type 5: Immediate Shift, dreg«···»DM</td>
<td>PM”</td>
<td>011</td>
<td>1</td>
<td>0</td>
<td>SRC UREG</td>
<td>COND</td>
<td>SU</td>
<td>DEST</td>
<td>UREG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b) “Type 5: Compute, ureg«···»ureg</td>
<td>Xdreg&lt;&gt;Ydreg”</td>
<td>011</td>
<td>1</td>
<td>1</td>
<td>Y DREG</td>
<td>COND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a) “Type 6: Immediate Shift, dreg«···»DM</td>
<td>PM”</td>
<td>100</td>
<td>0</td>
<td>I</td>
<td>M</td>
<td>COND</td>
<td>G</td>
<td>D</td>
<td>DATAEX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Summary

ADSP-21160 SHARC DSP Instruction Set Reference for ADSP-21160 SHARC DSPs

1-33
### Table 1-14. ADSP-21160 DSP Opcodes (Bits 26–0)

<table>
<thead>
<tr>
<th>Bit 26</th>
<th>Bit 25</th>
<th>Bit 24</th>
<th>Bit 23</th>
<th>Bit 22</th>
<th>Bit 21</th>
<th>Bit 20</th>
<th>Bit 19</th>
<th>Bit 18</th>
<th>Bit 17</th>
<th>Bit 16</th>
<th>Bit 15</th>
<th>Bit 14</th>
<th>Bit 13</th>
<th>Bit 12</th>
<th>Bit 11</th>
<th>Bit 10</th>
<th>Bit 9</th>
<th>Bit 8</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM DREG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;UREG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DREG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;DEST UREG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X DREG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The table shows the opcode bits 26–0 for the ADSP-21160 SHARC DSPs, with specific bit assignments for different registers and operations.
### Table 1-15. ADSP-21160 DSP Opcodes (Bits 47–27)

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>43</th>
<th>42</th>
<th>41</th>
<th>40</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b) “Type 6: Immediate Shift, dreg«…»DM</td>
<td>PM”</td>
<td>000</td>
<td>000010</td>
<td>COND</td>
<td>DATAEX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>“Type 7: Compute, modify”</td>
<td>000</td>
<td>00100</td>
<td>G</td>
<td>COND</td>
<td>I</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a) “Type 8: Direct Jump</td>
<td>Call”</td>
<td>000</td>
<td>00110</td>
<td>B</td>
<td>A</td>
<td>COND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b) “Type 8: Direct Jump</td>
<td>Call”</td>
<td>000</td>
<td>00111</td>
<td>B</td>
<td>A</td>
<td>COND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a) “Type 9: Indirect Jump</td>
<td>Call, Compute”</td>
<td>000</td>
<td>01000</td>
<td>B</td>
<td>A</td>
<td>COND</td>
<td>PMI</td>
<td>PMM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b) “Type 9: Indirect Jump</td>
<td>Call, Compute”</td>
<td>000</td>
<td>01001</td>
<td>B</td>
<td>A</td>
<td>COND</td>
<td>RELADDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a) “Type 10: Indirect Jump</td>
<td>Compute, dreg«…»DM”</td>
<td>110</td>
<td>D</td>
<td>DMI</td>
<td>DMM</td>
<td>COND</td>
<td>PMI</td>
<td>PMM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>43</th>
<th>42</th>
<th>41</th>
<th>40</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 6: Immediate Shift, dreg«…»DM</td>
<td>PM”</td>
<td>000</td>
<td>000010</td>
<td>COND</td>
<td>DATAEX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 1-16. ADSP-21160 DSP Opcodes (Bits 26–0)

| 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    | SHIFTOP | DATA | RN | RX |
| J | CI |    | ADDR |
| J | CI |    | RELADDR |
| J | E | CI | COMPUTE |
| J | E | CI | COMPUTE |
| DREG | COMPUTE |

| 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
### Table 1-17. ADSP-21160 DSP Opcodes (Bits 47–27)

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>43</th>
<th>42</th>
<th>41</th>
<th>40</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b) “Type 10: Indirect Jump</td>
<td>Compute, dreg«···»DM”</td>
<td>111</td>
<td>D</td>
<td>DMI</td>
<td>DMM</td>
<td>COND</td>
<td>RELADDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a) “Type 11: Return From Subroutine</td>
<td>Interrupt, Compute”</td>
<td>000</td>
<td></td>
<td></td>
<td>01010</td>
<td>COND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b) “Type 11: Return From Subroutine</td>
<td>Interrupt, Compute”</td>
<td>000</td>
<td></td>
<td></td>
<td>01011</td>
<td>COND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a) “Type 12: Do Until Counter Expired”</td>
<td>000</td>
<td></td>
<td></td>
<td>01100</td>
<td>DATA&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b) “Type 12: Do Until Counter Expired”</td>
<td>000</td>
<td></td>
<td></td>
<td>01101</td>
<td>0</td>
<td>UREG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>“Type 13: Do Until”</td>
<td>000</td>
<td></td>
<td></td>
<td>01110</td>
<td>TERM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>“Type 14: Ureg«···»DM</td>
<td>PM (direct addressing)”</td>
<td>000</td>
<td>100</td>
<td>G</td>
<td>D</td>
<td>L</td>
<td>UREG</td>
<td>ADDR (upper 5 bits)</td>
<td>Instruction Type</td>
<td>47</td>
<td>46</td>
<td>45</td>
<td>44</td>
<td>43</td>
<td>42</td>
<td>41</td>
<td>40</td>
<td>39</td>
<td>38</td>
<td>37</td>
<td>36</td>
</tr>
</tbody>
</table>
Table 1-18. ADSP-21160 DSP Opcodes (Bits 26–0) (Cont’d)

| 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| DREG | COMPUTE |
| J | E | L | R | COMPUTE |
| J | E | COMPUTE |
| <DATA | RELADDR |
| RELADDR |
| RELADDR |
| ADDR (lower 27 bits) |

| 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
### Table 1-19. ADSP-21160 DSP Opcodes (Bits 47–27)

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>43</th>
<th>42</th>
<th>41</th>
<th>40</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Type 15: Ureg&lt;--&gt;DM</td>
<td>PM (indirect addressing)”</td>
<td>101</td>
<td>G</td>
<td>I</td>
<td>D</td>
<td>L</td>
<td>UREG</td>
<td>DATA (upper 5 bits)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>“Type 16: Immediate data&lt;--&gt;DM</td>
<td>PM”</td>
<td>100</td>
<td>1</td>
<td>I</td>
<td>M</td>
<td>G</td>
<td>DATA (upper 5 bits)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>“Type 17: Immediate data&lt;--&gt;Ureg”</td>
<td>000</td>
<td>01111</td>
<td>0</td>
<td>UREG</td>
<td>DATA (upper 5 bits)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>“Type 18: System Register Bit Manipulation”</td>
<td>000</td>
<td>10100</td>
<td>BOP</td>
<td>SREG</td>
<td>DATA (upper 5 bits)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a) “Type 19: I Register Modify</td>
<td>Bit-Reverse”</td>
<td>000</td>
<td>10110</td>
<td>0</td>
<td>G</td>
<td>I</td>
<td>DATA (upper 5 bits)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b) “Type 19: I Register Modify</td>
<td>Bit-Reverse”</td>
<td>000</td>
<td>10110</td>
<td>1</td>
<td>G</td>
<td>I</td>
<td>DATA (upper 5 bits)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>“Type 20: Push, Pop Stacks, Flush Cache”</td>
<td>000</td>
<td>10111</td>
<td>L</td>
<td>P</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>S</td>
<td>P</td>
<td>O</td>
<td>S</td>
<td>P</td>
<td>O</td>
<td>P</td>
<td>P</td>
<td>O</td>
<td>F</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ADSP-21160 SHARC DSP Instruction Set Reference for ADSP-21160 SHARC DSPs
Table 1-20. ADSP-21160 DSP Opcodes (Bits 26–0) (Cont’d)

| 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| DATA (lower 27 bits) |
| DATA (lower 27 bits) |
| DATA (lower 27 bits) |
| DATA (lower 27 bits) |
| DATA (lower 27 bits) |
| DATA (lower 27 bits) |
### Table 1-21. ADSP-21160 DSP Opcodes (Bits 47–27)

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>43</th>
<th>42</th>
<th>41</th>
<th>40</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Type 21: Nop”</td>
<td>000</td>
<td>00000</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>“Type 22: Idle”</td>
<td>000</td>
<td>00000</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type 23: Idle16</td>
<td>Not supported on ADSP-21160</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type 24: creg«···»ureg</td>
<td>Not documented on ADSP-21160</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a) “Type 25: Cjump/Rframe”</td>
<td>0001</td>
<td>1000</td>
<td>0000</td>
<td>0100</td>
<td>0000</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b) “Type 25: Cjump/Rframe”</td>
<td>0001</td>
<td>1000</td>
<td>0100</td>
<td>0100</td>
<td>0000</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(c) “Type 25: Cjump/Rframe”</td>
<td>0001</td>
<td>1001</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 1-22. ADSP-21160 DSP Opcodes (Bits 26–0)

<p>| | | | | | | | | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>ADDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RELADDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
| 26    | 25    | 24    | 23    | 22    | 21    | 20    | 19    | 18    | 17    | 16    | 15    | 14    | 13    | 12    | 11    | 10    |  9   |  8   |  7   |  6   |  5   |  4   |  3   |  2   |  1   |  0   |
2 COMPUTE AND MOVE

The compute and move instructions in the Group I set of instructions specify a compute operation in parallel with one or two data moves or an index register modify.

Group I Instructions

The instructions in this group contain a compute field that specifies a compute operation using the ALU, multiplier, or shifter. Because there are a large number of options available for computations, these operations are described separately in the “Computations Reference” on page 6-1. Note that data moves between the MR registers and the register file are considered multiplier operations and are covered in the “Computations Reference” on page 6-1. Group I instructions include the following.

- “Type 1: Compute, Dreg«···»DM | Dreg«···»PM” on page 2-3
- Parallel data memory and program memory transfers with register file, optional compute operation
- “Type 2: Compute” on page 2-7
- Compute operation, optional condition
- “Type 3: Compute, ureg«···»DM | PM, register modify” on page 2-9
- Transfer between data or program memory and universal register, optional condition, optional compute operation
- “Type 4: Compute, dreg«•••»DM | PM, data modify” on page 2-14
  - PC-relative transfer between data or program memory and register file, optional condition, optional compute operation

- “Type 5: Compute, ureg«•••»ureg | Xdreg<->Ydreg” on page 2-19
  - Transfer between two universal registers, optional condition, optional compute operation

- “Type 6: Immediate Shift, dreg«•••»DM | PM” on page 2-23
  - Immediate shift operation, optional condition, optional transfer between data or program memory and register file

- “Type 7: Compute, modify” on page 2-28
  - Index register modify, optional condition, optional compute operation
**Type 1: Compute, Dreg«···»DM | Dreg«···»PM**

Parallel data memory and program memory transfers with register file, option compute operation

**Syntax**

```
compute , DM(Ia, Mb) = dreg1
, dreg1 = DM(Ia, Mb)
, PM(Ic, Md) = dreg2
, dreg2 = PM(Ic, Md)
```

**Function (SISD)**

In SISD mode, the Type 1 instruction provides parallel accesses to data and program memory from the register file. The specified I registers address data and program memory. The I values are post-modified and updated by the specified M registers. Pre-modify offset addressing is not supported. For more information on register restrictions, see the “Data Address Generators” chapter of the *ADSP-21160 SHARC DSP Hardware Reference*.

**Function (SIMD)**

In SIMD mode, the Type 1 instruction provides the same parallel accesses to data and program memory from the register file as are available in SISD mode, but provides these operations simultaneously for the X and Y processing elements.

The X element uses the specified I registers to address data and program memory, and the Y element adds one to the specified I registers to address data and program memory. If the broadcast read bits—\texttt{BDCST1} (for \texttt{I1}) or \texttt{BDCST9} (for \texttt{I9})—are set, the Y element uses the specified I register without adding one.

The I values are post-modified and updated by the specified M registers. Pre-modify offset addressing is not supported. For more information on
Type 1: Compute, Dreg«···»DM | Dreg«···»PM

register restrictions, see the “Data Address Generators” chapter of the 
*ADSP-21160 SHARC DSP Hardware Reference*.

The X element uses the specified Dreg registers, and the Y element uses 
the complementary registers (Cdreg) that correspond to the Dreg registers. 
For a list of complementary registers, see Table 1-10 on page 1-30.

The following pseudo code compares the Type 1 instruction’s explicit and 
implicit operations in SIMD mode.

**SIMD Explicit Operation (PEx Operation Stated in the Instruction Syntax)**

| compute | , DM(Ia, Mb) = dreg1 |
|         | , dreg1 = DM(Ia, Mb) |
|         | , PM(Ic, Md) = dreg2 |
|         | , dreg2 = PM(Ic, Md) |

**SIMD Implicit Operation (PEy Operation Implied by the Instruction Syntax)**

| compute | , DM(Ia+1, 0) = cdreg1 |
|         | , cdreg1 = DM(Ia+1, 0) |
|         | , PM(Ic+1, 0) = cdreg2 |
|         | , cdreg2 = PM(Ic+1, 0) |

ℹ️ Do not use the pseudo code above as instruction syntax.

**Examples**

R7=BSET R6 BY R0, DM(I0,M3)=R5, PM(I11,M15)=R4;
R8=DM(I4,M1), PM(I12 M12)=R0;

When the ADSP-21160 processor is in SISD, the first instruction in this 
example performs a computation along with two memory writes. DAG1 is 
used to write to DM and DAG2 is used to write to PM. In the second 
instruction, a read from data memory to register R8 and a write to program 
memory from register R0 are performed.

When the ADSP-21160 DSP is in SIMD, the first instruction in this 
example performs the same computation and performs two writes in
parallel on both PEx and PEy. The \( R_7 \) register on PEx and \( S_7 \) on PEy both store the results of the Bset computations. Also, simultaneous dual memory writes occur with DM and PM, writing in values from \( R_5, S_5 \) (DM) and \( R_4, S_4 \) (PM) respectively. In the second instruction, values are simultaneously read from data memory to registers \( R_8 \) and \( S_8 \) and written to program memory from registers \( R_0 \) and \( S_0 \).

\[
R_0 = DM(I_1, M_1);
\]

When the ADSP-21160 processor is in broadcast from the \( BDCST1 \) bit being set in the \( MODE1 \) system register, the \( R_0 \) (PEx) data register in this example is loaded with the value from data memory utilizing the \( I_1 \) register from DAG1, and \( S_0 \) (PEy) is loaded with the same value.

**Type 1 Opcode**

<table>
<thead>
<tr>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>43</th>
<th>42</th>
<th>41</th>
<th>40</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>D</td>
<td>M</td>
<td>D</td>
<td>DMI</td>
<td>DMM</td>
<td>P</td>
<td>M</td>
<td>D</td>
<td>DM</td>
<td>DREG</td>
<td>PMI</td>
<td>PMM</td>
<td>PM</td>
<td>DREG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |

**COMPUTE**
### Type 1: Compute, Dreg\(\cdots\)DM | Dreg\(\cdots\)PM

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMD, PMD</td>
<td>Select the access types (read or write)</td>
</tr>
<tr>
<td>DM DREG, PM DREG</td>
<td>Specify register file location.</td>
</tr>
<tr>
<td>DMI, PMI</td>
<td>Specify I registers for data and program memory</td>
</tr>
<tr>
<td>DMM, PMM</td>
<td>Specify M registers used to update the I registers</td>
</tr>
<tr>
<td>COMPUTE</td>
<td>Defines a compute operation to be performed in parallel with the data accesses; if omitted, this is a NOP</td>
</tr>
</tbody>
</table>
**Type 2: Compute**

Compute operation, optional condition

**Syntax**

```plaintext
IF COND compute ;
```

**Function (SISD)**

In SISD mode, the Type 2 instruction provides a conditional `compute` instruction. The instruction is executed if the specified `condition` tests true.

**Function (SIMD)**

In SIMD mode, the Type 2 instruction provides the same conditional `compute` instruction as is available in SISD mode, but provides the operation simultaneously for the X and Y processing elements. The instruction is executed in a processing element if the specified `condition` tests true in that element independent of the `condition` result for the other element.

The following pseudo code compares the Type 2 instruction’s explicit and implicit operations in SIMD mode.

**SIMD Explicit Operation (PEx Operation Stated in the Instruction Syntax)**

```plaintext
IF PEx COND compute ;
```

**SIMD Implicit Operation (PEy Operation Implied by the Instruction Syntax)**

```plaintext
IF PEy COND compute ;
```

ℹ️ Do not use the pseudo code above as instruction syntax.
Type 2: Compute

Examples

IF MV R6=SAT MRF (UI);

When the ADSP-21160 DSP is in SISD, the condition is evaluated in the PEx processing element. If the condition is true, the computation is performed and the result is stored in register $R6$.

When the ADSP-21160 DSP is in SIMD, the condition is evaluated on each processing element, PEx and PEy, independently. The computation executes on both PE’s, either one PE, or neither PE dependent on the outcome of the condition. If the condition is true in PEx, the computation is performed and the result is stored in register $R6$. If the condition is true in PEy, the computation is performed and the result is stored in register $S6$.

Type 2 Opcode

<table>
<thead>
<tr>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>43</th>
<th>42</th>
<th>41</th>
<th>40</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>00001</td>
<td>COND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| COMPUTE |

Bits | Description |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>COND</td>
<td>Selects whether the operation specified in the COMPUTE field is executed. If the COND is true, the compute is executed. If no condition is specified, COND is TRUE condition, and the compute is executed.</td>
</tr>
</tbody>
</table>
Type 3: Compute, ureg«···»DM | PM, register modify

Transfer operation between data or program memory and universal register, optional condition, optional compute operation

Syntax

IF COND compute

, DM(Ia, Mb) = ureg (LW);
, PM(Ic, Md)

, DM(Mb, Ia) = ureg (LW);
, PM(Md, Ic)

, ureg = DM(Ia, Mb) (LW);
, PM(Ic, Md) (LW);

Function (SISD)

In SISD mode, the Type 3 instruction provides access between data or program memory and a universal register. The specified I register addresses data or program memory. The I value is either pre-modified (M, I order) or post-modified (I, M order) by the specified M register. If it is post-modified, the I register is updated with the modified value. If a compute operation is specified, it is performed in parallel with the data access. The optional (LW) in this syntax lets you specify Long Word addressing, overriding default addressing from the memory map. If a condition is specified, it affects the entire instruction. Note that the Ureg may not be from the same DAG (that is, DAG1 or DAG2) as Ia/Mb or Ic/Md. For
more information on register restrictions, see the “Data Address Generators” chapter of the ADSP-21160 SHARC DSP Hardware Reference.

Function (SIMD)

In SIMD mode, the Type 3 instruction provides the same access between data or program memory and a universal register as is available in SISD mode, but provides this operation simultaneously for the X and Y processing elements.

The X element uses the specified I register to address data or program memory. The I value is either pre-modified (M, I order) or post-modified (I, M order) by the specified M register. The Y element adds one to the specified I register (before pre-modify or post-modify) to address data or program memory. If the broadcast read bits—BDGST1 (for I1) or BDGST9 (for I9)—are set, the Y element uses the specified I and M registers without adding one. If the I value post-modified, the I register is updated with the modified value from the specified M register. The optional (LW) in this syntax lets you specify Long Word addressing, overriding default addressing from the memory map.

For the universal register, the X element uses the specified Ureg register, and the Y element uses the corresponding complementary register (Cureg). For a list of complementary registers, see Table 1-10 on page 1-30. Note that the Ureg may not be from the same DAG (DAG1 or DAG2) as Ia/Mb or Ic/Md.

If a compute operation is specified, it is performed simultaneously on the X and Y processing elements in parallel with the data access. If a condition is specified, it affects the entire instruction. The instruction is executed in a processing element if the specified condition tests true in that element independent of the condition result for the other element.

The following pseudo code compares the Type 3 instruction’s explicit and implicit operations in SIMD mode.
Compute and Move

SIMD **Explicit** Operation (PEx Operation **Stated** in the Instruction Syntax)

\[
\text{IF PEx COND compute} \quad \begin{align*}
\text{DM}(Ia, Mb) \quad &= \text{ureg (LW);} \\
\text{PM}(Ic, Md) \\
\text{DM}(Mb, Ia) \quad &= \text{ureg (LW);} \\
\text{PM}(Md, Ic) \quad &= \text{ureg (LW);} \\
\text{ureg} &= \quad \begin{align*}
\text{DM}(Ia, Mb) (LW); \\
\text{PM}(Ic, Md) (LW); \\
\text{DM}(Mb, Ia) (LW); \\
\text{PM}(Md, Ic) (LW); \\
\end{align*}
\]

SIMD **Implicit** Operation (PEy Operation **Implied** by the Instruction Syntax)

\[
\text{IF PEy COND compute} \quad \begin{align*}
\text{DM}(Ia+1, 0) \quad &= \text{cureg (LW);} \\
\text{PM}(Ic+1, 0) \\
\text{DM}(Mb+1, Ia) \quad &= \text{cureg (LW);} \\
\text{PM}(Md+1, Ic) \quad &= \text{cureg (LW);} \\
\text{cureg} &= \quad \begin{align*}
\text{DM}(Ia+1, 0) (LW); \\
\text{PM}(Ic+, 0) (LW); \\
\text{DM}(Mb+1, Ia) (LW); \\
\text{PM}(Md+1, Ic) (LW); \\
\end{align*}
\]

_Do not use the pseudo code above as instruction syntax._
Type 3: Compute, ureg «···» DM | PM, register modify

Examples

\[ R6=R3-R11, \ \text{DM}(i0,m1)\text{=ASTAT}_x; \]
\[ \text{IF NOT SV } F8=\text{CLIP} \ F2 \ \text{BY} \ F14, \ F7=\text{PM}(i12,m12); \]

When the ADSP-21160 processor is in SISD, the computation and a data memory write in the first instruction are performed in PEx. The second instruction stores the result of the computation in \( F8 \), and the result of the program memory read into \( F7 \) if the condition’s outcome is true.

When the ADSP-21160 processor is in SIMD, the result of the computation in PEx in the first instruction is stored in \( R6 \), and the result of the parallel computation in PEy is stored in \( S6 \). In addition, there is a simultaneous data memory write of the values stored in \( \text{ASTAT}_x \) and \( \text{ASTAT}_y \). The condition is evaluated on each processing element, PEx and PEy, independently. The computation executes on both PE’s, either one PE, or neither PE, dependent on the outcome of the condition. If the condition is true in PEx, the computation is performed, the result is stored in register \( F8 \) and the result of the program memory read is stored in \( F7 \). If the condition is true in PEy, the computation is performed, the result is stored in register \( SF8 \), and the result of the program memory read is stored in \( SF7 \).

\[ \text{IF NOT SV } F8=\text{CLIP} \ F2 \ \text{BY} \ F14, \ F7=\text{PM}(i9,m12); \]

When the ADSP-21160 DSP is in broadcast from the \( \text{BDCST}_9 \) bit being set in the \( \text{MODE1} \) system register and the condition tests true, the computation is performed and the result is stored in register \( F8 \). Also, the result of the program memory read via the \( I9 \) register from \( \text{DAG2} \) is stored in \( F7 \). The \( SF7 \) register is loaded with the same value from program memory as \( F7 \).
# Type 3 Opcode

| 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 |
| 010 | U | I | M | COND | G | D | L | UREG |

**Bits** | **Description**
--- | ---
COND | Specifies the test condition; if omitted, COND is TRUE
D | Selects the access Type (read or write)
G | Selects data memory or program memory
L | Forces a long word (LW) access when address is in normal word address range
UREG | Specifies the universal register
I | Specifies the I register
M | Specifies the M register
U | Selects either update (post-modify) or no update (pre-modify)
COMPUTE | Defines a compute operation to be performed in parallel with the data access; if omitted, this is a NOP
Type 4: Compute, dreg «••» DM | PM, data modify

Type 4: Compute, dreg «••» DM | PM, data modify

PC-relative transfer between data or program memory and register file, optional condition, optional compute operation

Syntax

<table>
<thead>
<tr>
<th>IF COND compute</th>
<th>DM(Ia, &lt;data6&gt;) = dreg ;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PM(Ic, &lt;data6&gt;)</td>
</tr>
<tr>
<td></td>
<td>DM(&lt;data6&gt;, Ia) = dreg ;</td>
</tr>
<tr>
<td></td>
<td>PM(&lt;data6&gt;, Ic)</td>
</tr>
<tr>
<td></td>
<td>, dreg = DM(Ia, &lt;data6&gt;) ;</td>
</tr>
<tr>
<td></td>
<td>PM(Ic, &lt;data6&gt;) ;</td>
</tr>
<tr>
<td></td>
<td>, dreg = DM(&lt;data6&gt;, Ia) ;</td>
</tr>
<tr>
<td></td>
<td>PM(&lt;data6&gt;, Ic) ;</td>
</tr>
</tbody>
</table>

Function (SISD)

In SISD mode, the Type 4 instruction provides access between data or program memory and the register file. The specified I register addresses data or program memory. The I value is either pre-modified (data order, I) or post-modified (I, data order) by the specified immediate data. If it is post-modified, the I register is updated with the modified value. If a compute operation is specified, it is performed in parallel with the data access. If a condition is specified, it affects the entire instruction. For more information on register restrictions, see the “Data Address Generators” chapter of the ADSP-21160 SHARC DSP Hardware Reference.
Function (SIMD)

In SIMD mode, the Type 4 instruction provides the same access between data or program memory and the register file as is available in SISD mode, but provides the operation simultaneously for the X and Y processing elements.

The X element uses the specified I register to address data or program memory. The I value is either pre-modified (data, I order) or post-modified (I, data order) by the specified immediate data. The Y element adds one to the specified I register (before pre-modify or post-modify) to address data or program memory. If the broadcast read bits—BDCST1 (for I1) or BDCST9 (for I9)—are set, the Y element uses the specified I and M registers without adding one. If the I value post-modified, the I register is updated with the modified value from the specified M register. The optional (LW) in this syntax lets you specify Long Word addressing, overriding default addressing from the memory map.

For the data register, the X element uses the specified Dreg register, and the Y element uses the corresponding complementary register (Cdreg). For a list of complementary registers, see Table 1-10 on page 1-30.

If a compute operation is specified, it is performed simultaneously on the X and Y processing elements in parallel with the data access. If a condition is specified, it affects the entire instruction, not just the computation. The instruction is executed in a processing element if the specified condition tests true in that element independent of the condition result for the other element.

The following pseudo code compares the Type 4 instruction’s explicit and implicit operations in SIMD mode.

Examples

```
IF FLAG0_IN F1=F5*F12, F11=PM(I10,6);
R12=R3 AND R1, DM(6,11)=R6;
```
### Type 4: Compute, dreg«···»DM | PM, data modify

**SIMD Explicit Operation (PEx Operation Stated in the Instruction Syntax)**

<table>
<thead>
<tr>
<th>IF PEx COND compute</th>
<th>DM(Ia, &lt;data6&gt;)</th>
<th>PM(Ic, &lt;data6&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DM(&lt;data6&gt;, Ia)</td>
<td>PM(&lt;data6&gt;, Ic)</td>
</tr>
<tr>
<td>, dreg =</td>
<td>DM(Ia, &lt;data6&gt;)</td>
<td>PM(Ic, &lt;data6&gt;)</td>
</tr>
<tr>
<td>, dreg =</td>
<td>DM(&lt;data6&gt;, Ia)</td>
<td>PM(&lt;data6&gt;, Ic)</td>
</tr>
</tbody>
</table>

**SIMD Implicit Operation (PEy Operation Implied by the Instruction Syntax)**

<table>
<thead>
<tr>
<th>IF PEy COND compute</th>
<th>DM(Ia+1, 0)</th>
<th>PM(Ic+1, 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DM(&lt;data6&gt;+1, Ia)</td>
<td>PM(&lt;data6&gt;+1, Ic)</td>
</tr>
<tr>
<td>, cdreg =</td>
<td>DM(Ia+1, 0)</td>
<td>PM(Ic+1, 0)</td>
</tr>
<tr>
<td>, cdreg =</td>
<td>DM(&lt;data6&gt;+1, Ia)</td>
<td>PM(&lt;data6&gt;+1, Ic)</td>
</tr>
</tbody>
</table>

ℹ️ Do not use the pseudo code above as instruction syntax.
When the ADSP-21160 is in SISD, the computation and program memory read in the first instruction are performed in PEx if the condition’s outcome is true. The second instruction stores the result of the logical AND in R12 and writes the value within R6 into data memory.

When the ADSP-21160 is in SIMD, the condition is evaluated on each processing element, PEx and PEy, independently. The computation and program memory read execute on both PE’s, either one PE, or neither PE dependent on the outcome of the condition. If the condition is true in PEx, the computation is performed, and the result is stored in register F1, and the program memory value is read into register F11. If the condition is true in PEy, the computation is performed, the result is stored in register SF1, and the program memory value is read into register SF11.

If FLAG0_IN F1=F5*F12, F11=PM(I9,3);

When the ADSP-21160 is in broadcast from the BDCST9 bit is set in the MODE1 system register and the condition tests true, the computation is performed, the result is stored in register F1, and the program memory value is read into register F11 via the I9 register from DAG2. The SF11 register is also loaded with the same value from program memory as F11.

Type 4 Opcode

```
| 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 011| 0  | I  | G  | D  | U  | COND| DATA| DREG|
| 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|    |    |    |    |    |    | COMPUTE|
```
### Type 4: Compute, dreg\(\ldots\)DM | PM, data modify

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COND</td>
<td>Specifies the test condition; if omitted, COND is TRUE</td>
</tr>
<tr>
<td>D</td>
<td>Selects the access Type (read or write)</td>
</tr>
<tr>
<td>G</td>
<td>Selects data memory or program memory</td>
</tr>
<tr>
<td>DREG</td>
<td>Specifies the register file location</td>
</tr>
<tr>
<td>I</td>
<td>Specifies the I register</td>
</tr>
<tr>
<td>DATA</td>
<td>Specifies a 6-bit, twos-complement modify value</td>
</tr>
<tr>
<td>U</td>
<td>Selects either pre-modify without update or post-modify with update</td>
</tr>
<tr>
<td>COMPUTE</td>
<td>Defines a compute operation to be performed in parallel with the data access; if omitted, this is a NOP</td>
</tr>
</tbody>
</table>
Type 5: Compute, ureg1 → ureg2 | Xdreg <-> Ydreg

Transfer between two universal registers or swap between a data register in each processing element, optional condition, optional compute operation

Syntax

IF COND compute, ureg1 = ureg2 ;

Xdreg <-> Ydreg

Function (SISD)

In SISD mode, the Type 5 instruction provides transfer (=) from one universal register to another or provides a swap (<>->) between a data register in the X processing element and a data register in the Y processing element. If a compute operation is specified, it is performed in parallel with the data access. If a condition is specified, it affects the entire instruction.

Function (SIMD)

In SIMD mode, the Type 5 instruction provides the same transfer (=) from one register to another as is available in SISD mode, but provides this operation simultaneously for the X and Y processing elements. The swap (<>->) operation does the same operation in SISD and SIMD modes; no extra swap operation occurs in SIMD mode.

In the transfer (=), the X element transfers between the universal registers Ureg1 and Ureg2, and the Y element transfers between the complementary universal registers Cureg1 and Cureg2. For a list of complementary registers, see Table 1-10 on page 1-30.

If a compute operation is specified, it is performed simultaneously on the X and Y processing elements in parallel with the transfer. If a condition is specified, it affects the entire instruction. The instruction is executed in a
processing element if the specified condition tests true in that element independent of the condition result for the other element.

The following pseudo code compares the Type 5 instruction’s explicit and implicit operations in SIMD mode.

SIMD Explicit Operation (PEx Operation Stated in the Instruction Syntax)

IF PEx COND compute,       
ureg1 = ureg2  ;
X dreg <-> Y dreg

SIMD Implicit Operation (PEy Operation Implied by the Instruction Syntax)

IF PEy COND compute,       
cureg1 = cureg2  ;
{no implicit operation}

Do not use the pseudo code above as instruction syntax.

Examples

IF TF MRF=R2*R6(SSFR), M4=R0;  
LCNTR=L7;  
R0 <-> S1;

When the ADSP-21160 processor is in SISD, the condition in the first instruction is evaluated in the PEx processing element. If the condition is true, MRF is loaded with the result of the computation and a register transfer occurs between R0 and M4. The second instruction initializes the loop counter independent of the outcome of the first instruction’s condition. The third instruction swaps the register contents between R0 and S1.

When the ADSP-21160 DSP is in SIMD, the condition is evaluated on each processing element, PEx and PEy, independently. The computation
executes on both PE’s, either one PE, or neither PE dependent on the outcome of the condition. For the register transfer to complete, the condition must be satisfied in both PEx and PEy. The second instruction initializes the loop counter independent of the outcome of the first instruction’s condition. The third instruction swaps the register contents between $R_0$ and $S_1$—the SISD and SIMD swap operation is the same.

**Type 5 Opcode (Ureg = Ureg transfer)**

<table>
<thead>
<tr>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>43</th>
<th>42</th>
<th>41</th>
<th>40</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>011</td>
<td>1</td>
<td>0</td>
<td>SRC UREG</td>
<td>COND</td>
<td>SU</td>
<td>DEST UREG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Type 5 Opcode (X Dreg <-> Y Dreg swap)**

<table>
<thead>
<tr>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>43</th>
<th>42</th>
<th>41</th>
<th>40</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>011</td>
<td>1</td>
<td>1</td>
<td>Y DREG</td>
<td>COND</td>
<td></td>
<td>X DREG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

**COMPUTE**
### Type 5: Compute, ureg≪···≫ureg | Xdreg<->Ydreg

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COND</td>
<td>Specifies the test condition; if omitted, COND is TRUE</td>
</tr>
<tr>
<td>SRC UREG</td>
<td>Identifies the universal register source. (highest 5 bits of register code)</td>
</tr>
<tr>
<td>SU</td>
<td>Identifies the universal register source. (lowest 2 bits of register code)</td>
</tr>
<tr>
<td>DEST UREG</td>
<td>Identifies the universal register destination</td>
</tr>
<tr>
<td>Y DREG</td>
<td>Identifies the PEy data registers for swap (must appear to right of swap operator)</td>
</tr>
<tr>
<td>X DREG</td>
<td>Identifies the PEx data register for swap (must appear to left of swap operator)</td>
</tr>
<tr>
<td>COMPUTE</td>
<td>Defines a compute operation to be performed in parallel with the data transfer; if omitted, this is a NOP</td>
</tr>
</tbody>
</table>
Type 6: Immediate Shift, dreg«···»DM | PM

Immediate shift operation, optional condition, optional transfer between data or program memory and register file

Syntax

\[
\text{IF COND shiftimm, } DM(I_a, M_b), PM(I_c, M_d) = \text{dreg; }
\]

Function (SISD)

In SISD mode, the Type 6 instruction provides an immediate shift, which is a shifter operation that takes immediate data as its Y-operand. The immediate data is one 8-bit value or two 6-bit values, depending on the operation. The X-operand and the result are register file locations.

For more information on shifter operations, see “Shifter Operations” on page 6-64. For more information on register restrictions, see the “Data Address Generators” chapter of the ADSP-21160 SHARC DSP Hardware Reference.

If an access to data or program memory from the register file is specified, it is performed in parallel with the shifter operation. The I register addresses data or program memory. The I value is post-modified by the specified M register and updated with the modified value. If a condition is specified, it affects the entire instruction.
Function (SIMD)

In SIMD mode, the Type 6 instruction provides the same immediate shift operation as is available in SISD mode, but provides this operation simultaneously for the X and Y processing elements.

If an access to data or program memory from the register file is specified, it is performed simultaneously on the X and Y processing elements in parallel with the shifter operation.

The X element uses the specified I register to address data or program memory. The I value is post-modified by the specified M register and updated with the modified value. The Y element adds one to the specified I register to address data or program memory. If the broadcast read bits—BDCST1 (for I1) or BDCST9 (for I9)—are set, the Y element uses the specified I and M registers without adding one.

If a condition is specified, it affects the entire instruction. The instruction is executed in a processing element if the specified condition tests true in that element independent of the condition result for the other element.

The following pseudo code compares the Type 6 instruction’s explicit and implicit operations in SIMD mode.

SIMD Explicit Operation (PEx Operation Stated in the Instruction Syntax)

```plaintext
IF PEx COND shiftimm
   DM(Ia, Mb) = dreg ;
   PM(Ic, Md) ;
   , dreg = DM(Ia, Mb) ;
   PM(Ic, Md) ;
```

Do not use the pseudo code above as instruction syntax.
Examples

IF GT R2 = LSHIFT R6 BY 0x4, DM(I4, M4) = R0;
IF NOT SZ R3 = FEXT R1 BY 8:4;

When the ADSP-21160 processor is in SISD, the computation and data memory write in the first instruction are performed in PEx if the condition’s outcome is true. In the second instruction, register R3 is loaded with the result of the computation if the outcome of the condition is true.

When the ADSP-21160 processor is in SIMD, the condition is evaluated on each processing element, PEx and PEy, independently. The computation and data memory write executes on both PE’s, either one PE, or neither PE dependent on the outcome of the condition. If the condition is true in PEx, the computation is performed, the result is stored in register R2, and the data memory value is written from register R0. If the condition is true in PEy, the computation is performed, the result is stored in register S2, and the value within S0 is written into data memory. The second instruction’s condition is also evaluated on each processing element, PEx and PEy, independently. If the outcome of the condition is true, register R3 is loaded with the result of the computation on PEx, and register S3 is loaded with the result of the computation on PEy.

R2 = LSHIFT R6 BY 0x4, F3 = DM(I1, M3);
When the ADSP-21160 DSP is in broadcast from the BDCST1 bit being set in the MODE1 system register, the computation is performed, the result is stored in R2, and the data memory value is read into register F3 via the I1 register from DAG1. The SF3 register is also loaded with the same value from data memory as F3.

### Type 6 Opcode (with data access)

| 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 100 | 0 | I | M | COND | G | D | DATAEX | DREG |
| 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | SHIFTOP | DATA | RN | RX |

### Type 6 Opcode (without data access)

| 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 000 | 00010 | COND | DATAEX |
| 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | SHIFTOP | DATA | RN | RX |
## Compute and Move

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COND</td>
<td>Specifies the test condition; if omitted, COND is TRUE</td>
</tr>
<tr>
<td>SHIFTOP</td>
<td>Specifies the shifter operation. For more information, see “Shifter Operations” on page 6-64</td>
</tr>
<tr>
<td>DATA</td>
<td>Specifies an 8-bit immediate shift value. For shifter operations requiring two 6-bit values (a shift value and a length value), the DATAEX field adds 4 MSBs to the DATA field, creating a 12-bit immediate value. The six LSBs are the shift value, and the six MSBs are the length value.</td>
</tr>
<tr>
<td>D</td>
<td>Selects the access Type (read or write) if a memory access is specified</td>
</tr>
<tr>
<td>G</td>
<td>Selects data memory or program memory</td>
</tr>
<tr>
<td>DREG</td>
<td>Specifies the register file location</td>
</tr>
<tr>
<td>I</td>
<td>Specifies the I register, which is post-modified and updated by the M register</td>
</tr>
<tr>
<td>M</td>
<td>Identifies the M register for post-modify</td>
</tr>
</tbody>
</table>
Type 7: Compute, modify

Index register modify, optional condition, optional compute operation

Syntax

```plaintext
IF COND compute , MODIFY
   (Ia, Mb) ;
   (Ic, Md) ;
```

Function (SISD)

In SISD mode, the Type 7 instruction provides an update of the specified I register by the specified M register. If a compute operation is specified, it is performed in parallel with the data access. If a condition is specified, it affects the entire instruction. For more information on register restrictions, see the “Data Address Generators” chapter of the ADSP-21160 SHARC DSP Hardware Reference.

If the DAG’s Lx and Bx registers that correspond to Ia or Ic are set up for circular buffering, the Modify operation always executes circular buffer wrap around, independent of the state of the CBUFEN bit.

Function (SIMD)

In SIMD mode, the Type 7 instruction provides the same update of the specified I register by the specified M register as is available in SISD mode, but provides additional features for the optional compute operation.

If a compute operation is specified, it is performed simultaneously on the X and Y processing elements in parallel with the transfer. If a condition is specified, it affects the entire instruction. The instruction is executed in a processing element if the specified condition tests true in that element independent of the condition result for the other element.
The following pseudo code compares the Type 7 instruction’s explicit and implicit operations in SIMD mode.

**SIMD Explicit Operation (PEx Operation Stated in the Instruction Syntax)**

IF PEx COND compute, MODIFY (Ia, Mb);

IF PEx COND compute, MODIFY (Ic, Md);

**SIMD Implicit Operation (PEy Operation Implied by the Instruction Syntax)**

IF PEy COND compute {no implied MODIFY operation} 

Do not use the pseudo code above as instruction syntax.

**Examples**

IF NOT FLAG2_IN R4=R6*R12(SUF), MODIFY(I10,M8);

IF NOT LCE MODIFY(I3,M1);

When the ADSP-21160 processor is in SISD, the computation and index register modify in the first instruction are performed in PEx if the condition’s outcome is true. In the second instruction, an index register modification occurs if the outcome of the condition is true.

When the ADSP-21160 processor is in SIMD, the condition in the first instruction is evaluated on each processing element, PEx and PEy, independently. The computation executes on both PE’s, either PE, or neither PE dependent on the outcome of the condition. If the condition is true in PEx, the computation is performed, and the result is stored in R4. If the condition is true in PEy, the computation is performed, and the result is stored in S4. The index register modify operation occurs based on the logical OR’ing of the outcome of the conditions tested on both PE’s. In the second instruction, the index register modify also occurs based on the logical OR’ing of the outcomes of the conditions tested on both PE’s. Because both threads of a SIMD sequence may be dependent on a single
### Type 7: Compute, modify

DAG index value, either thread needs to be able to cause a modify of the index.

#### Type 7 Opcode

- **Bits Description**
  - **COND**: Specifies the test condition; if omitted, COND is TRUE
  - **G**: Selects DAG1 or DAG2
  - **I**: Specifies the I register
  - **M**: Specifies the M register
  - **COMPUTE**: Defines a compute operation to be performed in parallel with the data access; if omitted, this is a NOP

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COND</td>
<td>Specifies the test condition; if omitted, COND is TRUE</td>
</tr>
<tr>
<td>G</td>
<td>Selects DAG1 or DAG2</td>
</tr>
<tr>
<td>I</td>
<td>Specifies the I register</td>
</tr>
<tr>
<td>M</td>
<td>Specifies the M register</td>
</tr>
<tr>
<td>COMPUTE</td>
<td>Defines a compute operation to be performed in parallel with the data access; if omitted, this is a NOP</td>
</tr>
</tbody>
</table>
3  PROGRAM FLOW CONTROL

The program control instructions in the Group II set of instructions specify a program flow operation in parallel with a compute.

Group II Instructions

The instructions in this group contain a \texttt{compute} field that specifies a compute operation using the ALU, multiplier, or shifter. Because there are a large number of options available for computations, these operations are described separately in the “Computations Reference” on page 6-1. Note that data moves between the MR registers and the register file are considered multiplier operations and are covered in the “Computations Reference” on page 6-1. Group II instructions include the following.

- “Type 8: Direct Jump | Call” on page 3-3
- Direct (or PC-relative) jump/call, optional condition
- “Type 9: Indirect Jump | Call, Compute” on page 3-8
- Indirect (or PC-relative) jump/call, optional condition, optional compute operation
- “Type 10: Indirect Jump | Compute, dreg\texttt{\ldots}DM” on page 3-14
- Indirect (or PC-relative) jump or optional compute operation with transfer between data memory and register file
• “Type 11: Return From Subroutine | Interrupt, Compute” on page 3-19

• Return from subroutine or interrupt, optional condition, optional compute operation

• “Type 12: Do Until Counter Expired” on page 3-24

• Load loop counter, do loop until loop counter expired

• “Type 13: Do Until” on page 3-26

• Do until termination
Program Flow Control

Type 8: Direct Jump | Call

Direct (or PC-relative) jump/call, optional condition

Syntax

IF COND JUMP
<addr24>
(PC, <reladdr24>)
(DB)
(LA)
(CI)
(DB, LA)
(DB, CI)

IF COND CALL
<addr24>
(PC, <reladdr24>)
(DB)

Function (SISD)

In SISD mode, the Type 8 instruction provides a jump or call to the specified address or PC-relative address. The PC-relative address is a 24-bit, two’s-complement value. The Type 8 instruction supports the following modifiers.

- (DB)—delayed branch—starts a delayed branch
- (LA)—loop abort—causes the loop stacks and PC stack to be popped when the jump is executed. Use the (LA) modifier if the jump transfers program execution outside of a loop. Do not use (LA) if there is no loop or if the jump address is within the loop.
- (CI)—clear interrupt—lets you reuse an interrupt while it is being serviced

Normally, the ADSP-21160 processor ignores and does not latch an interrupt that reoccurs while its service routine is already executing. Jump (CI)
clears the status of the current interrupt without leaving the interrupt service routine. This feature reduces the interrupt routine to a normal subroutine and allows the interrupt to occur again, as a result of a different event or task in the ADSP-21160 DSP system. The Jump (CI) instruction should be located within the interrupt service routine. For more information on interrupts, see the “Program Sequencer” chapter of the ADSP-21160 SHARC DSP Hardware Reference.

To reduce the interrupt service routine to a normal subroutine, the Jump (CI) instruction clears the appropriate bit in the interrupt latch register (IRPTL) and interrupt mask pointer (IMASKP). The ADSP-21160 processor then allows the interrupt to occur again.

When returning from a reduced subroutine, you must use the (LR) modifier of the RTS if the interrupt occurs during the last two instructions of a loop. For related information, see “Type 11: Return From Subroutine | Interrupt, Compute” on page 3-19.

**Function (SIMD)**

In SIMD mode, the Type 8 instruction provides the same Jump or Call operation as in SISD mode, but provides additional features for handling the optional condition.

If a condition is specified, the Jump or Call is executed if the specified condition tests true in both the X and Y processing elements.

The following pseudo code compares the Type 8 instruction’s explicit and implicit operations in SIMD mode.

SIMD Explicit Operation (Program Sequencer Operation Stated in the Instruction Syntax)
Program Flow Control

IF (PEx AND PEy COND) JUMP
<addr24>
(PC, <reladdr24>)
(DB) ;

IF (PEx AND PEy COND) CALL
<addr24>
(PC, <reladdr24>)
(DB) ;

SIMD Explicit Operation (PEx Operation Stated in the Instruction Syntax)
{No explicit PEx operation}

SIMD Implicit Operation (PEy Operation Implied by the Instruction Syntax)
{No implicit PEy operation}

Do not use the pseudo code above as instruction syntax.

Examples

IF AV JUMP(PC,0x00A4) (LA);
CALL init (DB);  {init is a program label}
JUMP (PC,2) (DB,CI);  {clear current int. for reuse}

When the ADSP-21160 processor is in SISD, the first instruction performs a jump to the PC-relative address depending on the outcome of the condition tested in PEx. In the second instruction, a jump to the program label init occurs. A PC-relative jump takes place in the third instruction.

When the ADSP-21160 processor is in SIMD, the first instruction performs a jump to the PC-relative address depending on the logical AND’ing of the outcomes of the conditions tested in both PE’s. In SIMD
mode, the second and third instructions operate the same as in SISD mode. In the second instruction, a jump to the program label \textit{init} occurs. A PC-relative jump takes place in the third instruction.

**Type 8 Opcode (with direct branch)**

<table>
<thead>
<tr>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>43</th>
<th>42</th>
<th>41</th>
<th>40</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>00110</td>
<td>B</td>
<td>A</td>
<td>COND</td>
<td>J</td>
<td>CI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

** ADDR **

**Type 8 Opcode (with PC-relative branch)**

<table>
<thead>
<tr>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>43</th>
<th>42</th>
<th>41</th>
<th>40</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>00111</td>
<td>B</td>
<td>A</td>
<td>COND</td>
<td>J</td>
<td>CI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

** RELADDR **
### Program Flow Control

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COND</td>
<td>Specifies the test condition; if omitted, COND is TRUE</td>
</tr>
<tr>
<td>B</td>
<td>Selects the branch type, jump or call. For calls, A and CI are ignored</td>
</tr>
<tr>
<td>J</td>
<td>Determines whether the branch is delayed or non-delayed</td>
</tr>
<tr>
<td>ADDR</td>
<td>Specifies a 24-bit program memory address</td>
</tr>
<tr>
<td>A</td>
<td>Activates loop abort</td>
</tr>
<tr>
<td>CI</td>
<td>Activates clear interrupt</td>
</tr>
<tr>
<td>RELADDR</td>
<td>Holds a 24-bit, twos-complement value that is added to the current PC value to generate the branch address</td>
</tr>
</tbody>
</table>
Type 9: Indirect Jump | Call, Compute

Indirect (or PC-relative) jump/call, optional condition, optional compute operation

Syntax

IF COND JUMP
(Md, Ic)
(PC, <reladdr6>)
(DB)
(LA)
(CI)
(DB, LA)
(DB, CI)
, compute
, ELSE compute
;

IF COND CALL
(Md, Ic)
(PC, <reladdr6>)
(DB)
, compute
, ELSE compute
;

Function (SISD)

In SISD mode, the Type 9 instruction provides a Jump or Call to the specified PC-relative address or pre-modified I register value. The PC-relative address is a 6-bit, two's-complement value. If an I register is specified, it is modified by the specified M register to generate the branch address. The I register is not affected by the modify operation. The Type 9 instruction supports the following modifiers:

- (DB)—delayed branch—starts a delayed branch
- (LA)—loop abort—causes the loop stacks and PC stack to be popped when the jump is executed. Use the (LA) modifier if the jump transfers program execution outside of a loop. Do not use (LA) if there is no loop or if the jump address is within the loop.
- (CI)—clear interrupt—lets you reuse an interrupt while it is being serviced
Program Flow Control

Normally, the ADSP-21160 processor ignores and does not latch an interrupt that reoccurs while its service routine is already executing. Jump (CI) clears the status of the current interrupt without leaving the interrupt service routine. This feature reduces the interrupt routine to a normal subroutine and allows the interrupt to occur again, as a result of a different event or task in the ADSP-21160 DSP system. The Jump (CI) instruction should be located within the interrupt service routine. For more information on interrupts, see the “Program Sequencer” chapter of the ADSP-21160 SHARC DSP Hardware Reference.

To reduce an interrupt service routine to a normal subroutine, the Jump (CI) instruction clears the appropriate bit in the interrupt latch register (IRPTL) and interrupt mask pointer (IMASKP). The ADSP-21160 processor then allows the interrupt to occur again.

When returning from a reduced subroutine, you must use the (LR) modifier of the RTS instruction if the interrupt occurs during the last two instructions of a loop. For related information, see “Type 11: Return From Subroutine | Interrupt, Compute” on page 3-19.

The Jump or Call is executed if the optional specified condition is true or if no condition is specified. If a compute operation is specified without the ELSE, it is performed in parallel with the Jump or Call. If a compute operation is specified with the Else, it is performed only if the condition specified is false. Note that a condition must be specified if an Else compute clause is specified.

Function (SIMD)

In SIMD mode, the Type 9 instruction provides the same Jump or Call operation as is available in SISD mode, but provides additional features for the optional condition.

If a condition is specified, the Jump or Call is executed if the specified condition tests true in both the X and Y processing elements.
If a compute operation is specified without the Else, it is performed by the processing element(s) in which the condition test true in parallel with the Jump or Call. If a compute operation is specified with the Else, it is performed in an element when the condition tests false in that element. Note that a condition must be specified if an Else compute clause is specified.

Note that for the compute, the X element uses the specified registers and the Y element uses the complementary registers. For a list of complementary registers, see Table 1-10 on page 1-30.

The following pseudo code compares the Type 9 instruction’s explicit and implicit operations in SIMD mode.

Examples

```
JUMP(M8,I12), R6=R6-1;
IF EQ CALL(PC,17)(DB), ELSE R6=R6-1;
```

When the ADSP-21160 processor is in SISD, the indirect jump and compute in the first instruction are performed in parallel. In the second instruction, a call occurs if the condition is true, otherwise the computation is performed.

When the ADSP-21160 processor is in SIMD, the indirect jump in the first instruction occurs in parallel with both processing elements executing computations. In PEx, R6 stores the result, and S6 stores the result in PEy. In the second instruction, the condition is evaluated independently on each processing element, PEx and PEy. The Call executes based on the logical AND'ing of the PEx and PEy conditional tests. So, the Call executes if the condition tests true in both PEx and PEy. Because the Else inverts the conditional test, the computation is performed independently on either PEx or PEy based on the negative evaluation of the condition code seen by that processing element. If the computation is executed, R6
SIMD Explicit Operation (PEx Operation Stated in the Instruction Syntax)

IF (PEx AND PEy COND) JUMP

(Md, Ic) (DB) , (if PEx COND) compute
(PC, <reladdr6>) (LA) , ELSE (if NOT PEx) compute
(CI) (DB, LA)
(DB, CI)

IF (PEx AND PEy COND) CALL

(Md, Ic) (DB) , (if PEx COND) compute
(PC, <reladdr6>) , ELSE (if NOT PEx) compute

SIMD Implicit Operation (PEy Operation Implied by the Instruction Syntax)

IF (PEx AND PEy COND) JUMP

(Md, Ic) (DB) , (if PEy COND) compute
(PC, <reladdr6>) (LA) , ELSE (if NOT PEy) compute
(CI) (DB, LA)
(DB, CI)

IF (PEx AND PEy COND) CALL

(Md, Ic) (DB) , (if PEy COND) compute
(PC, <reladdr6>) , ELSE (if NOT PEy) compute

Do not use the pseudo code above as instruction syntax.
stores the result of the computation in PEx, and $s_6$ stores the result of the computation in PEy.

For a summary of SISD/SIMD conditional testing, see “SISD/SIMD Conditional Testing Summary” on page 1-22.

### Type 9 Opcode (with indirect branch)

| 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 000 | 01000 | B | A | COND | PMI | PMM | J | E | CI |

Computes:

### Type 9 Opcode (with PC-relative branch)

| 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 000 | 01001 | B | A | COND | RELADDR | J | E | CI |

Computes:
## Program Flow Control

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COND</td>
<td>Specifies the test condition; if omitted, COND is true</td>
</tr>
<tr>
<td>E</td>
<td>Specifies whether or not an ELSE clause is used</td>
</tr>
<tr>
<td>B</td>
<td>Selects the branch type, jump or call. For calls, A and CI are ignored.</td>
</tr>
<tr>
<td>J</td>
<td>Selects delayed or non-delayed branch</td>
</tr>
<tr>
<td>A</td>
<td>Activates loop abort</td>
</tr>
<tr>
<td>CI</td>
<td>Activates clear interrupt</td>
</tr>
<tr>
<td>COMPUTE</td>
<td>Defines a compute operation to be performed in parallel with the data access; if omitted, this is a NOP</td>
</tr>
<tr>
<td>RELADDR</td>
<td>Holds a 6-bit, two's-complement value that is added to the current PC value to generate the branch address</td>
</tr>
<tr>
<td>PMI</td>
<td>Specifies the I register for indirect branches. The I register is pre-modified but not updated by the M register.</td>
</tr>
<tr>
<td>PMM</td>
<td>Specifies the M register for pre-modifies</td>
</tr>
</tbody>
</table>
Type 10: Indirect Jump | Compute, dreg ← DM

Indirect (or PC-relative) jump or optional compute operation with transfer between data memory and register file

Syntax

\[
\text{IF COND Jump } (\text{Md}, \text{Ic}) \quad \text{,Else}
\quad \text{compute, } DM(\text{Ia}, \text{Mb}) = \text{dreg} \\
\quad \text{(PC, <reladdr6>) compute, } \text{dreg} = DM(\text{Ia}, \text{Mb})
\]

Function (SISD)

In SISD mode, the Type 10 instruction provides a conditional Jump to either specified PC-relative address or pre-modified I register value. In parallel with the Jump, this instruction also provides a transfer between data memory and a data register with optional parallel compute operation. For this instruction, the If condition and Else keywords are not optional and must be used. If the specified condition is true, the Jump is executed. If the specified condition is false, the data memory transfer and optional compute operation are performed in parallel. Only the compute operation is optional in this instruction.

The PC-relative address for the Jump is a 6-bit, two’s-complement value. If an I register is specified (Ic), it is modified by the specified M register (Md) to generate the branch address. The I register is not affected by the modify operation. For this Jump, you may not use the delay branch (DB), loop abort (LA), or clear interrupt (CI) modifiers.

For the data memory access, the I register (Ia) provides the address. The I register value is post-modified by the specified M register (Mb) and is updated with the modified value. Pre-modify addressing is not available for this data memory access.
Function (SIMD)

In SIMD mode, the Type 10 instruction provides the same conditional Jump as is available in SISD mode, but the Jump is executed if the specified condition tests true in both the X or Y processing elements.

In parallel with the Jump, this instruction also provides a transfer between data memory and a data register in the X and Y processing elements. An optional parallel compute operation for the X and Y processing elements is also available.

For this instruction, the If condition and Else keywords are not optional and must be used. If the specified condition is true in both processing elements, the Jump is executed. The data memory transfer and optional compute operation specified with the Else are performed in an element when the condition tests false in that element.

Note that for the compute, the X element uses the specified Dreg register and the Y element uses the complementary Cdreg register. For a list of complementary registers, see Table 1-10 on page 1-30. Only the compute operation is optional in this instruction.

The addressing for the Jump is the same in SISD and SIMD modes, but addressing for the data memory access differs slightly. For the data memory access in SIMD mode, X processing element uses the specified I register \((I_a)\) to address memory. The I register value is post-modified by the specified M register \((M_b)\) and is updated with the modified value. The Y element adds one to the specified I register to address memory.

Pre-modify addressing is not available for this data memory access.

The following pseudo code compares the Type 10 instruction’s explicit and implicit operations in SIMD mode.

Examples

```plaintext
IF TF JUMP(M8, I8),
ELSE R6=DM(I6, M1);
```
When the ADSP-21160 processor is in SISD, the indirect jump in the first instruction is performed if the condition tests true. Otherwise, R6 stores the value of a data memory read. The second instruction is much like the first, however, it also includes an optional compute, which is performed in parallel with the data memory read.

When the ADSP-21160 processor is in SIMD, the indirect Jump in the first instruction executes depending on the outcome of the conditional in both processing element. The condition is evaluated independently on each processing element, PEx and PEy. The indirect Jump executes based on the logical AND’ing of the PEx and PEy conditional tests. So, the indirect Jump executes if the condition tests true in both PEx and PEy. The data memory read is performed independently on either PEx or PEy based on the negative evaluation of the condition code seen by that PE.

The second instruction is much like the first instruction. The second instruction, however, includes an optional compute also performed in parallel with the data memory read independently on either PEx or PEy and
Program Flow Control

based on the negative evaluation of the condition code seen by that processing element.

For a summary of SISD/SIMD conditional testing, see “SISD/SIMD Conditional Testing Summary” on page 1-22.

IF TF JUMP(M8,I8), ELSE R6=DM(I1,M1);

When the ADSP-21160 processor is in broadcast from the BDCST1 bit being set in the MODE1 system register, the instruction performs an indirect jump if the condition tests true. Otherwise, R6 stores the value of a data memory read via the I1 register from DAG1. The S6 register is also loaded with the same value from data memory as R6.

Type 10 Opcode (with indirect jump)

| 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 110| D  | DMI| DMM| COND| PMI| PMM| DREG|

22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

COMPUTE

Type 10 Opcode (with PC-relative jump)

| 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 111| D  | DMI| DMM| COND| RELADDR| DREG|

22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

COMPUTE
<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COND</td>
<td>Specifies the condition to test; not optional</td>
</tr>
<tr>
<td>PMI</td>
<td>Specifies the I register for indirect branches. The I register is premodified, but not updated by the M register.</td>
</tr>
<tr>
<td>PMM</td>
<td>Specifies the M register for pre-modifies</td>
</tr>
<tr>
<td>D</td>
<td>Selects the data memory access Type (read or write)</td>
</tr>
<tr>
<td>DREG</td>
<td>Specifies the register file location</td>
</tr>
<tr>
<td>DMI</td>
<td>Specifies the I register that is post-modified and updated by the M register</td>
</tr>
<tr>
<td>DMM</td>
<td>Identifies the M register for post-modifies</td>
</tr>
<tr>
<td>COMPUTE</td>
<td>Defines a compute operation to be performed in parallel with the data access; if omitted, this is a NOP</td>
</tr>
<tr>
<td>RELADDR</td>
<td>Holds a 6-bit, twos-complement value that is added to the current PC value to generate the branch address</td>
</tr>
</tbody>
</table>
Program Flow Control

Type 11: Return From Subroutine | Interrupt, Compute

Indirect (or PC-relative) jump or optional compute operation with transfer between data memory and register file

Syntax

IF COND RTS (DB) , compute (LR) , ELSE compute (DB, LR)

IF COND RTI (DB) , compute , ELSE compute

Function (SISD)

In SISD mode, the Type 11 instruction provides a return from a subroutine (RTS) or return from an interrupt service routine (RTI). A return causes the processor to branch to the address stored at the top of the PC stack. The difference between RTS and RTI is that the RTS instruction only pops the return address off the PC stack, while the RTI does that plus:

- Pops status stack if the ASTAT and MODE1 status registers have been pushed—if the interrupt was IRQ2-0, the timer interrupt, or the VIRPT vector interrupt

- Clears the appropriate bit in the interrupt latch register (IRPTL) and the interrupt mask pointer (IMASKP)

The return executes when the optional If condition is true (or if no condition is specified). If a compute operation is specified without the Else, it is performed in parallel with the return. If a compute operation is specified
with the Else, it is performed only when the If condition is false. Note that a condition must be specified if an Else compute clause is specified.

RTS supports two modifiers (DB) and (LR); RTI supports one modifier, (DB). If the delayed branch (DB) modifier is specified, the return is delayed; otherwise, it is non-delayed.

If the return is not a delayed branch and occurs as one of the last three instructions of a loop, you must use the loop reentry (LR) modifier with the subroutine’s RTS instruction. The (LR) modifier assures proper reentry into the loop. For example, the DSP checks the termination condition in counter-based loops by decrementing the current loop counter (CURLCNTR) during execution of the instruction two locations before the end of the loop. In this case, the RTS (LR) instruction prevents the loop counter from being decremented again, avoiding the error of decrementing twice for the same loop iteration.

You must also use the (LR) modifier for RTS when returning from a subroutine that has been reduced from an interrupt service routine with a Jump (CI) instruction. This case occurs when the interrupt occurs during the last two instructions of a loop. For a description of the Jump (CI) instruction, see “Type 8: Direct Jump | Call” on page 3-3 or “Type 9: Indirect Jump | Call, Compute” on page 3-8.

Function (SIMD)

In SIMD mode, the Type 11 instruction provides the same return operations as are available in SISD mode, except that the return is executed if the specified condition tests true in both the X and Y processing elements.

In parallel with the return, this instruction also provides a parallel compute or Else compute operation for the X and Y processing elements. If a condition is specified, the optional compute is executed in a processing element if the specified condition tests true in that processing element. If a
compute operation is specified with the Else, it is performed in an element when the condition tests false in that element.

Note that for the compute, the X element uses the specified registers, and the Y element uses the complementary registers. For a list of complementary registers, see Table 1-10 on page 1-30.

The following pseudo code compares the Type 11 instruction’s explicit and implicit operations in SIMD mode.

**SIMD Explicit Operation (PEx Operation Stated in the Instruction Syntax)**

<table>
<thead>
<tr>
<th>IF (PEx AND PEy COND) RTS</th>
<th>(DB)</th>
<th>, (if PEx COND) compute</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(LR)</td>
<td>, ELSE (if NOT PEx) compute</td>
</tr>
<tr>
<td></td>
<td>(DB, LR)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IF (PEx AND PEy COND) RTI</th>
<th>(DB)</th>
<th>, (if PEx COND) compute</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(LR)</td>
<td>, ELSE (if NOT PEx) compute</td>
</tr>
<tr>
<td></td>
<td>(DB, LR)</td>
<td></td>
</tr>
</tbody>
</table>

**SIMD Implicit Operation (PEy Operation Implied by the Instruction Syntax)**

<table>
<thead>
<tr>
<th>IF (PEx AND PEy COND) RTS</th>
<th>(DB)</th>
<th>, (if PEy COND) compute</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(LR)</td>
<td>, ELSE (if NOT PEy) compute</td>
</tr>
<tr>
<td></td>
<td>(DB, LR)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IF (PEx AND PEy COND) RTI</th>
<th>(DB)</th>
<th>, (if PEy COND) compute</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(LR)</td>
<td>, ELSE (if NOT PEy) compute</td>
</tr>
<tr>
<td></td>
<td>(DB, LR)</td>
<td></td>
</tr>
</tbody>
</table>

Do not use the pseudo code above as instruction syntax.
Examples

RTI, R6=R5 XOR R1;
IF le RTS(DB);
IF sz RTS, ELSE R0=LSHIFT R1 BY R15;

When the ADSP-21160 processor is in SISD, the first instruction performs a return from interrupt and a computation in parallel. The second instruction performs a return from subroutine only if the condition is true. In the third instruction, a return from subroutine is executed if the condition is true. Otherwise, the computation executes.

When the ADSP-21160 processor is in SIMD, the first instruction performs a return from interrupt and both processing elements execute the computation in parallel. The result from PEx is placed in R6, and the result from PEy is placed in S6. The second instruction performs a return from subroutine (RTS) if the condition tests true in both PEx or PEy. In the third instruction, the condition is evaluated independently on each processing element, PEx and PEy. The RTS executes based on the logical AND’ing of the PEx and PEy conditional tests. So, the RTS executes if the condition tests true in both PEx and PEy. Because the Else inverts the conditional test, the computation is performed independently on either PEx or PEy based on the negative evaluation of the condition code seen by that processing element. The R0 register stores the result in PEx, and S0 stores the result in PEy if the computations are executed.

For a summary of SISD/SIMD conditional testing, see “SISD/SIMD Conditional Testing Summary” on page 1-22.
Program Flow Control

Type 11 Opcode (return from subroutine)

<table>
<thead>
<tr>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>43</th>
<th>42</th>
<th>41</th>
<th>40</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>01010</td>
<td>COND</td>
<td>J</td>
<td>E</td>
<td>LR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

COMPUTE

Type 11 Opcode (return from interrupt)

<table>
<thead>
<tr>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>43</th>
<th>42</th>
<th>41</th>
<th>40</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>01011</td>
<td>COND</td>
<td>J</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

COMPUTE

Bits | Description
--- | ---
COND | Specifies the test condition; if omitted, COND is true
J | Determines whether the return is delayed or non-delayed
E | Specifies whether an ELSE clause is used
COMPUTE | Defines the compute operation to be performed; if omitted, this is a NOP
LR | Specifies whether or not the loop reentry modifier is specified
Type 12: Do Until Counter Expired

Load loop counter, do loop until loop counter expired

Syntax

\[
\text{LCNTR} = \begin{cases} <\text{data16}> \\ \text{ureg} \end{cases}, \quad \text{DO} \quad \begin{cases} <\text{addr24}> \\ (\text{PC}, <\text{reladdr24}>) \end{cases} \quad \text{UNTIL LCE};
\]

Function (SISD and SIMD)

In SISD or SIMD modes, the Type 12 instruction sets up a counter-based program loop. The loop counter \texttt{LCNTR} is loaded with 16-bit immediate data or from a universal register. The loop start address is pushed on the \texttt{PC} stack. The loop end address and the \texttt{LCE} termination condition are pushed on the loop address stack. The end address can be either a label for an absolute 24-bit program memory address, or a \texttt{PC}-relative 24-bit twos-complement address. The \texttt{LCNTR} is pushed on the loop counter stack and becomes the \texttt{CURLCNTR} value. The loop executes until the \texttt{CURLCNTR} reaches zero.

Examples

\[
\begin{align*}
\text{LCNTR} &= 100, \quad \text{DO fmax UNTIL LCE}; \quad \{\text{fmax is a program label}\} \\
\text{LCNTR} &= \text{R12}, \quad \text{DO (PC,16) UNTIL LCE};
\end{align*}
\]

The ADSP-21160 processor (in SISD or SIMD) executes the action at the indicated address for the duration of the loop.
Program Flow Control

Type 12 Opcode (with immediate loop counter load)

| 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 000 | 01100 | DATA |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----| 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |

RELADDR

Type 12 Opcode (with loop counter load from a Ureg)

| 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 000 | 01101 | 0 | UREG |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----| 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |

RELADDR

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RELADDR</td>
<td>Specifies the end-of-loop address relative to the DO LOOP instruction address. The assembler also accepts an absolute address and converts the absolute address to the equivalent relative address for coding.</td>
</tr>
<tr>
<td>DATA</td>
<td>Specifies a 16-bit value to load into the loop counter (LCNTR) for an immediate load.</td>
</tr>
<tr>
<td>UREG</td>
<td>Specifies a register containing a 16-bit value to load into the loop counter (LCNTR) for a load from an universal register.</td>
</tr>
</tbody>
</table>
Type 13: Do Until

Do until termination

Syntax

```
DO <addr24> UNTIL termination ;
(PC, <reladdr24>)
```

Function (SISD)

In SISD mode, the Type 13 instruction sets up a conditional program loop. The loop start address is pushed on the PC stack. The loop end address and the termination condition are pushed on the loop stack. The end address can be either a label for an absolute 24-bit program memory address or a PC-relative, 24-bit two’s-complement address. The loop executes until the termination condition tests true.

Function (SIMD)

In SIMD mode, the Type 13 instruction provides the same conditional program loop as is available in SISD mode, except that in SIMD mode the loop executes until the termination condition tests true in both the X and Y processing elements.

The following pseudo code compares the Type 13 instruction’s explicit and implicit operations in SIMD mode.

Examples

```
DO end UNTIL FLAG1_IN;  {end is a program label}
DO (PC,7) UNTIL AC;
```

When the ADSP-21160 processor is in SISD, the end program label in the first instruction specifies the start address for the loop, and the loop is executed until the instruction’s condition tests true. In the second
Program Flow Control

SIMD **Explicit** Operation (Program Sequencer Operation **Stated** in the Instruction Syntax)

```
DO <addr24> UNTIL (PEx AND PEy) termination ;
    (PC, <reladdr24>)
```

SIMD **Explicit** Operation (PEx Operation **Stated** in the Instruction Syntax)

{No explicit PEx operation}

SIMD **Implicit** Operation (PEy Operation **Implied** by the Instruction Syntax)

{No implicit PEy operation}

⚠️ Do not use the pseudo code above as instruction syntax.

When the ADSP-21160 processor is in SIMD, the end program label in the first instruction specifies the start address for the loop, and the loop is executed until the instruction’s condition tests true in both PEx or PEy. In the second instruction, the start address is given in the form of a PC-relative address. The loop executes until the instruction’s condition tests true in both PEx or PEy.
### Type 13 Opcode (relative addressing)

<table>
<thead>
<tr>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>43</th>
<th>42</th>
<th>41</th>
<th>40</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>01110</td>
<td>TERM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### RELADDR

<table>
<thead>
<tr>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

**Bits** | **Description**
---|---
RELADDR | Specifies the end-of-loop address relative to the DO LOOP instruction address. The assembler accepts an absolute address as well and converts the absolute address to the equivalent relative address for coding.
TERM | Specifies the termination condition.
4 IMMEDIATE MOVE INSTRUCTIONS

The immediate move instructions in the Group III set of instructions specify register-to-memory data moves.

Group III Instructions

Group III instructions include the following.

- “Type 14: Ureg«⋯»DM | PM (direct addressing)” on page 4-2
- Transfer between data or program memory and universal register, direct addressing, immediate address
- “Type 15: Ureg«⋯»DM | PM (indirect addressing)” on page 4-5
- Transfer between data or program memory and universal register, indirect addressing, immediate modifier
- “Type 16: Immediate data⋯»DM | PM” on page 4-9
- Immediate data write to data or program memory
- “Type 17: Immediate data⋯»Ureg” on page 4-12
- Immediate data write to universal register
Type 14: Ureg«···»DM | PM (direct addressing)

Transfer between data or program memory and universal register, direct addressing, immediate address

Syntax

```
DM(<addr32>) = ureg (LW);
PM(<addr32>) = ureg (LW);
```

Function (SISD)

In SISD mode, the Type 14 instruction sets up an access between data or program memory and a universal register, with direct addressing. The entire data or program memory address is specified in the instruction. Addresses are 32 bits wide (0 to $2^{32} - 1$). The optional (LW) in this syntax lets you specify Long Word addressing, overriding default addressing from the memory map.

Function (SIMD)

In SIMD mode, the Type 14 instruction provides the same access between data or program memory and a universal register, with direct addressing, as is available in SISD mode, except that addressing differs slightly, and the transfer occurs in parallel for the X and Y processing elements.

For the memory access in SIMD mode, the X processing element uses the specified 32-bit address to address memory. The Y element adds one to the specified 32-bit address to address memory.
Immediate Move Instructions

For the universal register, the X element uses the specified Ureg, and the Y element uses the complementary register (Cureg) that corresponds to the Ureg register specified in the instruction. For a list of complementary registers, see Table 1-10 on page 1-30. Note that only the Cureg subset registers which have complimentary registers are effected by SIMD mode.

The following pseudo code compares the Type 14 instruction’s explicit and implicit operations in SIMD mode.

**SIMD Explicit Operation (PEx Operation Stated in the Instruction Syntax)**

```
DM(<addr32>)
PM(<addr32>)
= ureg (LW);
```

ureg =
```
DM(<addr32>) (LW);
PM(<addr32>) (LW);
```

**SIMD Implicit Operation (PEy Operation Implied by the Instruction Syntax)**

```
DM(<addr32>+1)
PM(<addr32>+1)
= cureg (LW);
```

cureg =
```
DM(<addr32>+1) (LW);
PM(<addr32>+1) (LW);
```

Do not use the pseudo code above as instruction syntax.

**Examples**

```
DM(temp)=MODE1;   {temp is a program label}
WAIT=PM(0x489060);
```
When the ADSP-21160 processor is in SISD, the first instruction performs a direct memory write of the value in the MODE1 register into data memory with the data memory destination address specified by the program label, temp. The second instruction initializes the WAIT register with the value found in the specified address in program memory.

Because of the register selections in this example, these two instructions operate the same in SIMD and SISD mode. The MODE1 (SYSCON) and WAIT (IOP) registers are not included in the Cureg subset, so they do not operate differently in SIMD mode.

### Type 14 Opcode

| 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 000 | 100 | G | D | L | UREG | ADDR (upper 8 bits) |

| 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| ADDR (lower 24 bits) |

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Selects the access Type (read or write)</td>
</tr>
<tr>
<td>G</td>
<td>Selects the memory Type (data or program)</td>
</tr>
<tr>
<td>L</td>
<td>Forces a long word (LW) access when address is in normal word address range</td>
</tr>
<tr>
<td>UREG</td>
<td>Specifies the number of a universal register</td>
</tr>
<tr>
<td>ADDR</td>
<td>Contains the immediate address value</td>
</tr>
</tbody>
</table>
Immediate Move Instructions

Type 15: Ureg«···»DM | PM (indirect addressing)

Transfer between data or program memory and universal register, indirect addressing, immediate modifier

Syntax

\[
\begin{align*}
\text{DM(<data32>, Ia)} & = \text{ureg} \\
\text{PM(<data32>, Ic)} & = \text{ureg}
\end{align*}
\]

Function (SISD)

In SISD mode, the Type 15 instruction sets up an access between data or program memory and a universal register, with indirect addressing using I registers. The I register is pre-modified with an immediate value specified in the instruction. The I register is not updated. Address modifiers are 32 bits wide (0 to \(2^{32}-1\)). The Ureg may not be from the same DAG (that is, DAG1 or DAG2) as Ia/Mb or Ic/Md. For more information on register restrictions, see the “Data Address Generators” chapter of the ADSP-21160 SHARC DSP Hardware Reference. The optional (LW) in this syntax lets you specify Long Word addressing, overriding default addressing from the memory map.

Function (SIMD)

In SIMD mode, the Type 15 instruction provides the same access between data or program memory and a universal register, with indirect addressing using I registers, as is available in SISD mode, except that addressing differs slightly, and the transfer occurs in parallel for the X and Y processing elements.
The X processing element uses the specified I register—pre-modified with an immediate value—to address memory. The Y processing element adds one to the pre-modified I value to address memory. The I register is not updated.

The Ureg specified in the instruction is used for the X processing element transfer and may not be from the same DAG (that is, DAG1 or DAG2) as \( I_a/M_b \) or \( I_c/M_d \). The Y element uses the complementary register (Cureg) that correspond to the Ureg register specified in the instruction. For a list of complementary registers, see Table 1-10 on page 1-30. Note that only the Cureg subset registers which have complimentary registers are effected by SIMD mode. For more information on register restrictions, see the “Data Address Generators” chapter of the ADSP-21160 SHARC DSP Hardware Reference.

The following pseudo code compares the Type 15 instruction’s explicit and implicit operations in SIMD mode.

SIMD Explicit Operation (PEx Operation Stated in the Instruction Syntax)

\[
\begin{align*}
DM(<\text{data32}>, I_a) & \quad \text{= ureg (LW);} \\
PM(<\text{data32}>, I_c) &
\end{align*}
\]

ureg = \[
\begin{align*}
DM(<\text{data32}>, I_a) & \quad \text{(LW);} \\
PM(<\text{data32}>, I_c)
\end{align*}
\]

Do not use the pseudo code above as instruction syntax.
Immediate Move Instructions

SIMD Implicit Operation (PEy Operation Implied by the Instruction Syntax)

\[
\begin{align*}
\text{DM}(<\text{data32}>+1, Ia) & = \text{cureg} \quad \text{(LW);} \\
\text{PM}(<\text{data32}>+1, Ic) & \\
\text{cureg} & = \begin{array}{c}
\text{DM}(<\text{data32}>+1, Ia) \\
\text{PM}(<\text{data32}>+1, Ic)
\end{array} \quad \text{(LW);} \\
\end{align*}
\]

Do not use the pseudo code above as instruction syntax.

Examples

\[
\begin{align*}
\text{DM}(24, I5) & = \text{Tcount}; \\
\text{Ustat}1 & = \text{PM}(\text{offs}, I13); \quad \text{"offs" is a user-defined constant}
\end{align*}
\]

When the ADSP-21160 processor is in SISD, the first instruction performs a data memory write, using indirect addressing and the Ureg timer register, TCOUNT. The DAG1 register I5 is pre-modified with the immediate value of 24. The I5 register is not updated after the memory access occurs. The second instruction performs a program memory read, using indirect addressing and the system register, USTAT1. The DAG2 register I13 is pre-modified with the immediate value of the defined constant, offs. The I13 register is not updated after the memory access occurs.

Because of the register selections in this example, the first instruction in this example operates the same in SIMD and SISD mode. The TCOUNT (timer) register is not included in the Cureg subset, and therefore the first instruction operates the same in SIMD and SISD mode.

The second instruction operates differently in SIMD. The USTAT1 (system) register is included in the Cureg subset. Therefore, a program memory read—using indirect addressing and the system register, USTAT1 and its complimentary register USTAT2—is performed in parallel on PE0.
and PEy respectively. The DAG2 register I13 is pre-modified with the immediate value of the defined constant, offs, to address memory on PEx. This same pre-modified value in I13 is skewed by 1 to address memory on PEy. The I13 register is not updated after the memory access occurs in SIMD mode.

**Type 15 Opcode**

<table>
<thead>
<tr>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>43</th>
<th>42</th>
<th>41</th>
<th>40</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>G</td>
<td>I</td>
<td>D</td>
<td>L</td>
<td>UREG</td>
<td>DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(upper 8 bits)

<table>
<thead>
<tr>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(lower 24 bits)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Selects the access Type (read or write)</td>
</tr>
<tr>
<td>G</td>
<td>Selects the memory Type (data or program)</td>
</tr>
<tr>
<td>L</td>
<td>Forces a long word (LW) access when address is in normal word address range</td>
</tr>
<tr>
<td>UREG</td>
<td>Specifies the number of a universal register</td>
</tr>
<tr>
<td>DATA</td>
<td>Specifies the immediate modify value for the I register</td>
</tr>
</tbody>
</table>
Immediate Move Instructions

Type 16: Immediate data \(\rightarrow\) DM | PM

Immediate data write to data or program memory

Syntax

Function (SISD)

In SISD mode, the Type 16 instruction sets up a write of 32-bit immediate data to data or program memory, with indirect addressing. The data is placed in the most significant 32 bits of the 40-bit memory word. The least significant 8 bits are loaded with 0s. The I register is post-modified and updated by the specified M register.

Function (SIMD)

In SIMD mode, the Type 16 instruction provides the same write of 32-bit immediate data to data or program memory, with indirect addressing, as is available in SISD mode, except that addressing differs slightly, and the transfer occurs in parallel for the X and Y processing elements.

The X processing element uses the specified I register to address memory. The Y processing element adds one to the I register to address memory. The I register is post-modified and updated by the specified M register.

The following pseudo code compares the Type 16 instruction’s explicit and implicit operations in SIMD mode.

Examples

\[
\begin{align*}
DM(I4, M0) &= 19304; \\
PM(I14, M11) &= \text{count}; \text{ (count is user-defined constant)}
\end{align*}
\]
When the ADSP-21160 processor is in SISD, the two immediate memory writes are performed on PEx. The first instruction writes to data memory and the second instruction writes to program memory. DAG1 and DAG2 are used to indirectly address the locations in memory to which values are written. The \( I_4 \) and \( I_{14} \) registers are post-modified and updated by \( M_0 \) and \( M_{11} \) respectively.

When the ADSP-21160 processor is in SIMD, the two immediate memory writes are performed in parallel on PEx and PEy. The first instruction writes to data memory and the second instruction writes to program memory. DAG1 and DAG2 are used to indirectly address the locations in memory to which values are written. The \( I_4 \) and \( I_{14} \) registers are post-modified and updated by \( M_0 \) and \( M_{11} \) respectively.
### Immediate Move Instructions

#### Type 16 Opcode

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>I</td>
</tr>
<tr>
<td>46</td>
<td>M</td>
</tr>
<tr>
<td>45</td>
<td>I</td>
</tr>
<tr>
<td>44</td>
<td>M</td>
</tr>
<tr>
<td>43</td>
<td>G</td>
</tr>
<tr>
<td>42</td>
<td>G</td>
</tr>
<tr>
<td>41</td>
<td>G</td>
</tr>
<tr>
<td>40</td>
<td>G</td>
</tr>
<tr>
<td>39</td>
<td>G</td>
</tr>
<tr>
<td>38</td>
<td>G</td>
</tr>
<tr>
<td>37</td>
<td>G</td>
</tr>
<tr>
<td>36</td>
<td>G</td>
</tr>
<tr>
<td>35</td>
<td>G</td>
</tr>
<tr>
<td>34</td>
<td>G</td>
</tr>
<tr>
<td>33</td>
<td>G</td>
</tr>
<tr>
<td>32</td>
<td>G</td>
</tr>
<tr>
<td>31</td>
<td>G</td>
</tr>
<tr>
<td>30</td>
<td>G</td>
</tr>
<tr>
<td>29</td>
<td>G</td>
</tr>
<tr>
<td>28</td>
<td>G</td>
</tr>
<tr>
<td>27</td>
<td>G</td>
</tr>
<tr>
<td>26</td>
<td>G</td>
</tr>
<tr>
<td>25</td>
<td>G</td>
</tr>
<tr>
<td>24</td>
<td>G</td>
</tr>
<tr>
<td>23</td>
<td>DATA (upper 8 bits)</td>
</tr>
<tr>
<td>22</td>
<td>DATA (lower 24 bits)</td>
</tr>
<tr>
<td>21</td>
<td>DATA (lower 24 bits)</td>
</tr>
<tr>
<td>20</td>
<td>DATA (lower 24 bits)</td>
</tr>
<tr>
<td>19</td>
<td>DATA (lower 24 bits)</td>
</tr>
<tr>
<td>18</td>
<td>DATA (lower 24 bits)</td>
</tr>
<tr>
<td>17</td>
<td>DATA (lower 24 bits)</td>
</tr>
<tr>
<td>16</td>
<td>DATA (lower 24 bits)</td>
</tr>
<tr>
<td>15</td>
<td>DATA (lower 24 bits)</td>
</tr>
<tr>
<td>14</td>
<td>DATA (lower 24 bits)</td>
</tr>
<tr>
<td>13</td>
<td>DATA (lower 24 bits)</td>
</tr>
<tr>
<td>12</td>
<td>DATA (lower 24 bits)</td>
</tr>
<tr>
<td>11</td>
<td>DATA (lower 24 bits)</td>
</tr>
<tr>
<td>10</td>
<td>DATA (lower 24 bits)</td>
</tr>
<tr>
<td>9</td>
<td>DATA (lower 24 bits)</td>
</tr>
<tr>
<td>8</td>
<td>DATA (lower 24 bits)</td>
</tr>
<tr>
<td>7</td>
<td>DATA (lower 24 bits)</td>
</tr>
<tr>
<td>6</td>
<td>DATA (lower 24 bits)</td>
</tr>
<tr>
<td>5</td>
<td>DATA (lower 24 bits)</td>
</tr>
<tr>
<td>4</td>
<td>DATA (lower 24 bits)</td>
</tr>
<tr>
<td>3</td>
<td>DATA (lower 24 bits)</td>
</tr>
<tr>
<td>2</td>
<td>DATA (lower 24 bits)</td>
</tr>
<tr>
<td>1</td>
<td>DATA (lower 24 bits)</td>
</tr>
<tr>
<td>0</td>
<td>DATA (lower 24 bits)</td>
</tr>
</tbody>
</table>

- **I**: Selects the I register
- **M**: Selects the M register
- **G**: Selects the memory (data or program)
- **DATA**: Specifies the 32-bit immediate data
Type 17: Immediate data \( \rightarrow \text{Ureg} \)

Immediate data write to universal register

Syntax

\[
\text{ureg} = <\text{data32}>;
\]

Function (SISD)

In SISD mode, the Type 17 instruction writes 32-bit immediate data to a universal register. If the register is 40 bits wide, the data is placed in the most significant 32 bits, and the least significant 8 bits are loaded with 0s.

Function (SIMD)

In SIMD mode, the Type 17 instruction provides the same write of 32-bit immediate data to universal register as is available in SISD mode, but provides parallel writes for the X and Y processing elements.

The X element uses the specified Ureg, and the Y element uses the complementary Cureg. Note that only the Cureg subset registers which have complimentary registers are effected by SIMD mode. For a list of complimentary registers, see Table 1-10 on page 1-30.

The following pseudo code compares the Type 17 instruction’s explicit and implicit operations in SIMD mode.

Examples

\[
\text{ASTATx}=0\times0; \\
\text{M15} = \text{mod1}; \quad \{\text{mod1 is user-defined constant}\}
\]

When the ADSP-21160 processor is in SISD, the two instructions load immediate values into the specified registers.
Immediate Move Instructions

SIMD Explicit Operation (PEx Operation Stated in the Instruction Syntax)

ureg = <data32> ;

SIMD Implicit Operation (PEy Operation Implied by the Instruction Syntax)

cureg = <data32> ;

Do not use the pseudo code above as instruction syntax.

Because of the register selections in this example, the second instruction in
this example operates the same in SIMD and SISD mode. The ASTATx
(system) register is included in the Cureg subset. In the first instruction,
the immediate data write to the system register ASTATx and its compli-
mentary register ASTATy are performed in parallel on PEx and PEy
respectively. In the second instruction, the M15 register is not included in
the Cureg subset. Therefore, the second instruction operates the same in
SIMD and SISD mode.
### Type 17 Opcode

<table>
<thead>
<tr>
<th></th>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>43</th>
<th>42</th>
<th>41</th>
<th>40</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0111</td>
<td></td>
<td>UREG</td>
<td></td>
<td>DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Bits Description

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UREG</td>
<td>Specifies the number of a universal register.</td>
</tr>
<tr>
<td>DATA</td>
<td>Specifies the immediate modify value for the I register.</td>
</tr>
</tbody>
</table>
5 MISCELLANEOUS OPERATIONS

The miscellaneous operation instructions in the Group IV set of instructions specify system operations.

Group IV Instructions

Group IV instructions include the following.

- “Type 18: System Register Bit Manipulation” on page 5-2
- System register bit manipulation
- “Type 19: I Register Modify | Bit-Reverse” on page 5-6
- Immediate I register modify, with or without bit-reverse
- “Type 20: Push, Pop Stacks, Flush Cache” on page 5-9
- Push or Pop of loop and/or status stacks
- “Type 21: Nop” on page 5-11
- No Operation (NOP)
- “Type 22: Idle” on page 5-12
- Idle
- “Type 25: Cjump/Rframe” on page 5-13
- CJUMP/RFRAME (Compiler-generated instruction)
Type 18: System Register Bit Manipulation

System register bit manipulation

Syntax

BIT | SET | sreg <data32> ;
CLR
TGL
TST
XOR

Function (SISD)

In SISD mode, the Type 18 instruction provides a bit manipulation operation on a system register. This instruction can set, clear, toggle or test specified bits, or compare (XOR) the system register with a specified data value. In the first four operations, the immediate data value is a mask.

The set operation sets all the bits in the specified system register that are also set in the specified data value. The clear operation clears all the bits that are set in the data value. The toggle operation toggles all the bits that are set in the data value. The test operation sets the bit test flag (BTF in ASTATx/y) if all the bits that are set in the data value are also set in the system register. The XOR operation sets the bit test flag (BTF in ASTATx/y) if the system register value is the same as the data value.

For more information on shifter operations, see “Computations Reference” on page 6-1. For more information on system registers, see the “Registers” appendix of the ADSP-21160 SHARC DSP Hardware Reference.
Function (SIMD)

In SIMD mode, the Type 18 instruction provides the same bit manipulation operations as are available in SISD mode, but provides them in parallel for the X and Y processing elements.

The X element operation uses the specified Sreg, and the Y element operations uses the complementary Csreg. For a list of complementary registers, see Table 1-10 on page 1-30.

The following pseudo code compares the Type 18 instruction’s explicit and implicit operations in SIMD mode.

<table>
<thead>
<tr>
<th>SIMD Explicit Operation (PEx Operation Stated in the Instruction Syntax)</th>
<th>SIMD Implicit Operation (PEy Operation Implied by the Instruction Syntax)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>SET</td>
</tr>
<tr>
<td>CLR</td>
<td></td>
</tr>
<tr>
<td>TGL</td>
<td></td>
</tr>
<tr>
<td>TST</td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>csreg &lt;data32&gt; ;</td>
</tr>
</tbody>
</table>

Do not use the pseudo code above as instruction syntax.
Type 18: System Register Bit Manipulation

Examples

BIT SET MODE2 0x00000070;
BIT TST ASTATx 0x00002000;

When the ADSP-21160 processor is in SISD, the first instruction sets all of the bits in the MODE2 register that are also set in the data value, bits 4, 5, and 6 in this case. The second instruction sets the bit test flag (BTF in ASTATx) if all the bits set in the data value, just bit 13 in this case, are also set in the system register.

Because of the register selections in this example, the first instruction operates the same in SISD and SIMD, but the second instruction operates differently in SIMD. Only the Cureg subset registers which have complimentary registers are affected in SIMD mode. The ASTATx (system) register is included in the Cureg subset, so the bit test operations are performed independently on each processing element in parallel using these complimentary registers. The BTF is set on both PE’s (ASTATx and ASTATy), either one PE (ASTATx or ASTATy), or neither PE dependent on the outcome of the bit test operation.
### Type 18 Opcode

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>47-40</td>
<td>DATA (upper 8 bits)</td>
</tr>
<tr>
<td>39-32</td>
<td>SREG</td>
</tr>
<tr>
<td>31</td>
<td>BOP</td>
</tr>
<tr>
<td>00</td>
<td>10100</td>
</tr>
</tbody>
</table>

#### Bits Description

- **BOP**: Selects one of the five bit operations.
- **SREG**: Specifies the system register.
- **DATA**: Specifies the data value.
Type 19: I Register Modify | Bit-Reverse

Immediate I register modify, with or without bit-reverse

Syntax

```
MODIFY  (Ia, <data32>)  ;
            (Ic, <data32>)

BITREV   (Ia, <data32>)  ;
            (Ic, <data32>)
```

Function (SISD & SIMD)

In SISD and SIMD modes, the Type 19 instruction modifies and updates the specified I register by an immediate 32-bit data value. If the address is to be bit-reversed, you must specify a DAG1 Ia register (10–17) or DAG2 Ic register (18–115), and the modified value is bit-reversed before being written back to the I register. No address is output in either case. For more information on register restrictions, see the “Data Address Generators” chapter of the ADSP-21160 SHARC DSP Hardware Reference.

If the DAG’s Lx and Bx registers that correspond to Ia or Ic are set up for circular bufferring, the Modify operation always executes circular buffer wrap around, independent of the state of the CBUFEN bit.
Examples

MODIFY (I4,304);
BITREV (I7,space);  {space is a user-defined constant}

In SISD and SIMD, the first instruction modifies and updates the I4 register by the immediate value of 304. The second instruction utilizes the DAG1 register I7. The value originally stored in I7 is modified by the defined constant, space, and is then bit-reversed before being written back to the I7 register.

Type 19 Opcode (without bit-reverse)

\[
\begin{array}{cccccccccccccccccccc}
\hline
000 & 10110 & 0 & G & I & \text{DATA} \ (\text{upper 8 bits}) \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccccccc}
23 & 22 & 21 & 20 & 19 & 18 & 17 & 16 & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
\text{DATA} \ (\text{lower 24 bits}) \\
\end{array}
\]

Type 19 Opcode (with bit-reverse)

\[
\begin{array}{cccccccccccccccccccc}
\hline
000 & 10110 & 1 & G & I & \text{DATA} \ (\text{upper 8 bits}) \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccccccc}
23 & 22 & 21 & 20 & 19 & 18 & 17 & 16 & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
\text{DATA} \ (\text{lower 24 bits}) \\
\end{array}
\]
### Type 19: I Register Modify | Bit-Reverse

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
</table>
| G    | Selects the data address generator:  
      | G=0 for DAG1  
      | G=1 for DAG2  |
| I    | Selects the I register:  
      | I=0–7 for I0–I7 (for DAG1)  
      | I=0–7 for I8–I15 (for DAG2)  |
| DATA | Specifies the immediate modifier. |
Type 20: Push, Pop Stacks, Flush Cache

Push or Pop of loop and/or status stacks

Syntax

```
PUSH LOOP, PUSH STS, PUSH PCSTK, FLUSH CACHE;
```

Function (SISD and SIMD)

In SISD and SIMD modes, the Type 20 instruction pushes or pops the loop address and loop counter stacks, the status stack, and/or the PC stack, and/or clear the instruction cache. Any of set of Pushes (Push Loop, Push Sts, Push Pcstk) or Pops (Pop Loop, Pop Sts, Pop Pcstk) may be combined in a single instruction, but a Push may not be combined with a Pop.

Flushing the instruction cache invalidates all entries in the cache, with no latency—the cache is cleared at the end of the cycle.

Examples

```
PUSH LOOP, PUSH STS;
PUSH PCSTK, FLUSH CACHE;
```

In SISD and SIMD, the first instruction pushes the loop stack and status stack. The second instruction pops the PC stack and flushes the cache.

When a PUSH LOOP instruction is executed, the top of the loop address stack is filled with zeros. After executing the PUSH LOOP instruction, LADDR must be written with the content to be pushed on the loop address stack. A write to the LADDR will update the top of the loop address stack.
Type 20: Push, Pop Stacks, Flush Cache

Type 20 Opcode

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPU</td>
<td>Pushes the loop stacks</td>
</tr>
<tr>
<td>LPO</td>
<td>Pops the loop stacks</td>
</tr>
<tr>
<td>SPU</td>
<td>Pushes the status stack</td>
</tr>
<tr>
<td>SPO</td>
<td>Pops the status stack</td>
</tr>
<tr>
<td>PPU</td>
<td>Pushes the PC stack</td>
</tr>
<tr>
<td>PPO</td>
<td>Pops the PC stack</td>
</tr>
<tr>
<td>FC</td>
<td>Causes a cache flush</td>
</tr>
</tbody>
</table>

5-10 ADSP-21160 SHARC DSP Instruction Set Reference for ADSP-21160 SHARC DSPs
Type 21: Nop

No Operation (NOP)

Syntax

NOP ;

Function (SISD and SIMD)

In SISD and SIMD modes, the Type 21 instruction provides a null operation; it increments only the fetch address.

Type 21 Opcode

| 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 000| 0000| 0 |

| 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
Type 22: Idle

Idle

Syntax

IDLE ;

Function (SISD and SIMD)

In SISD and SIMD modes, the Type 22 instruction executes a Nop and puts the processor in a low power state. The processor remains in the low power state until an interrupt occurs. On return from the interrupt, execution continues at the instruction following the Idle instruction.

Type 22 Opcode

| 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 000 | 00000 | 1 |

| 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
Type 25: Cjump/Rframe

Cjump/Rframe (Compiler-generated instruction)

Syntax

```
CJUMP function (DB) ;
(PC, <reladdr24>)
```

```
RFRAME ;
```

Function (SISD and SIMD)

In SISD mode, the Type 25 instruction (Cjump) combines a direct or PC-relative jump with register transfer operations that save the frame and stack pointers. The instruction (Rframe) also reverses the register transfers to restore the frame and stack pointers.

The Type 25 instruction is only intended for use by a C (or other high-level language) compiler. Do not use Cjump or Rframe in your assembly programs.

The different forms of this instruction perform the operations listed in Table 5-1.
### Table 5-1. Operations Done by Forms of the Type 25 Instruction

<table>
<thead>
<tr>
<th>Compiler-Generated Instruction</th>
<th>Operations Performed in SISD Mode</th>
<th>Operations Performed in SIMD Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>CJUMP label (DB);</td>
<td>JUMP label (DB), R2=I6, I6=I7;</td>
<td>JUMP label (DB), R2=I6, S2=I6, I6=I7;</td>
</tr>
<tr>
<td>CJUMP (PC,raddr)(DB);</td>
<td>JUMP (PC,raddr) (DB), R2=I6, I6=I7;</td>
<td>JUMP (PC,raddr) (DB), R2=I6, S2=I6, I6=I7;</td>
</tr>
<tr>
<td>RFRAME;</td>
<td>I7=I6, I6=DM(0,I6);</td>
<td>I7=I6, I6=DM(0,I6), I6=DM(1,I6);</td>
</tr>
</tbody>
</table>

1 In this table, raddr indicates a relative 24-bit address.
Type 25a Opcode (with direct branch)

<table>
<thead>
<tr>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>43</th>
<th>42</th>
<th>41</th>
<th>40</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>1000</td>
<td>0000</td>
<td>0100</td>
<td>0000</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

ADDR

Type 25b Opcode (with PC-relative branch)

<table>
<thead>
<tr>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>43</th>
<th>42</th>
<th>41</th>
<th>40</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>1000</td>
<td>0100</td>
<td>0100</td>
<td>0000</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

RELADDR

Bits | Description
--- | ---
ADDR | Specifies a 24-bit program memory address for “function”
RELADDR | Specifies a 24-bit, twos-complement value added to the current PC value to generate the branch address
Type 25c Opcode (RFRAME)

<table>
<thead>
<tr>
<th>47</th>
<th>46</th>
<th>45</th>
<th>44</th>
<th>43</th>
<th>42</th>
<th>41</th>
<th>40</th>
<th>39</th>
<th>38</th>
<th>37</th>
<th>36</th>
<th>35</th>
<th>34</th>
<th>33</th>
<th>32</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>1001</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6 COMPUTATIONS REFERENCE

This chapter describes each compute operation in detail, including its assembly language syntax and opcode field. Compute operations execute in the multiplier, the ALU, and the shifter.

Compute Field

The 23-bit compute field is a mini instruction within the ADSP-21000 instruction. You can specify a value in this field for a variety of compute operations, which include the following.

- Single-function operations involve a single computation unit.
- Multifunction operations specify parallel operation of the multiplier and the ALU or two operations in the ALU.
- The MR register transfer is a special type of compute operation used to access the fixed-point accumulator in the multiplier.

For each operation, the assembly language syntax, the function, and the opcode format and contents are specified. For an explanation of the notation and abbreviations, see Chapter 2, “Instruction Summary.”
Compute Field

In single-function operations, the compute field of a single-function operation is made up of the following bit fields.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CU</td>
<td>Specifies the computation unit for the compute operation, where: 00=ALU, 01=Multiplier, and 10=Shifter</td>
</tr>
<tr>
<td>Opcode</td>
<td>Specifies the compute operation</td>
</tr>
<tr>
<td>Rn</td>
<td>Specifies register for the compute result</td>
</tr>
<tr>
<td>Rx</td>
<td>Specifies register for the compute’s x operand</td>
</tr>
<tr>
<td>Ry</td>
<td>Specifies register for the compute’s y operand</td>
</tr>
</tbody>
</table>

The compute operation (Opcode) is executed in the computation unit (CU). The x operand and y operand are input from data registers (Rx and Ry). The compute result goes to a data register (Rn). Note that in some shifter operations, the result register (Rn) serves as a result destination and as source for a third input operand.

The available compute operations (Opcode) appear in Table 6-1 on page 6-4, Table 6-2 on page 6-5, Table 6-3 on page 6-53, Table 6-4 on page 6-54, and Table 6-8 on page 6-65. These tables are organized by computation unit: “ALU Operations” on page 6-3, “Multiplier Operations” on page 6-51, and “Shifter Operations” on page 6-64. Following each table, each compute operation is described in detail.
ALU Operations

This section describes the ALU operations. Table 6-1 and Table 6-2 summarize the syntax and opcodes for the fixed-point and floating-point ALU operations, respectively.
## Fixed-Point ALU Operations

### Table 6-1. Fixed-Point ALU Operations

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Opcode</th>
<th>Reference page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn = Rx + Ry</td>
<td>0000 0001</td>
<td>on page 6-7</td>
</tr>
<tr>
<td>Rn = Rx – Ry</td>
<td>0000 0010</td>
<td>on page 6-8</td>
</tr>
<tr>
<td>Rn = Rx + Ry + CI</td>
<td>0000 0101</td>
<td>on page 6-9</td>
</tr>
<tr>
<td>Rn = Rx – Ry + CI – 1</td>
<td>0000 0110</td>
<td>on page 6-10</td>
</tr>
<tr>
<td>Rn = (Rx + Ry)/2</td>
<td>0000 1001</td>
<td>on page 6-11</td>
</tr>
<tr>
<td>COMP(Rx, Ry)</td>
<td>0000 1010</td>
<td>on page 6-12</td>
</tr>
<tr>
<td>COMPU(Rx, Ry)</td>
<td>0000 1011</td>
<td>on page 6-13</td>
</tr>
<tr>
<td>Rn = Rx + CI</td>
<td>0010 0101</td>
<td>on page 6-14</td>
</tr>
<tr>
<td>Rn = Rx + CI – 1</td>
<td>0010 0110</td>
<td>on page 6-15</td>
</tr>
<tr>
<td>Rn = Rx + 1</td>
<td>0010 1001</td>
<td>on page 6-16</td>
</tr>
<tr>
<td>Rn = Rx – 1</td>
<td>0010 1010</td>
<td>on page 6-17</td>
</tr>
<tr>
<td>Rn = – Rx</td>
<td>0010 0010</td>
<td>on page 6-18</td>
</tr>
<tr>
<td>Rn = ABS Rx</td>
<td>0011 0000</td>
<td>on page 6-19</td>
</tr>
<tr>
<td>Rn = PASS Rx</td>
<td>0010 0001</td>
<td>on page 6-20</td>
</tr>
<tr>
<td>Rn = Rx AND Ry</td>
<td>0100 0000</td>
<td>on page 6-21</td>
</tr>
<tr>
<td>Rn = Rx OR Ry</td>
<td>0100 0001</td>
<td>on page 6-22</td>
</tr>
<tr>
<td>Rn = Rx XOR Ry</td>
<td>0100 0010</td>
<td>on page 6-23</td>
</tr>
<tr>
<td>Rn = NOT Rx</td>
<td>0100 0011</td>
<td>on page 6-24</td>
</tr>
<tr>
<td>Rn = MIN(Rx, Ry)</td>
<td>0110 0001</td>
<td>on page 6-25</td>
</tr>
<tr>
<td>Rn = MAX(Rx, Ry)</td>
<td>0110 0010</td>
<td>on page 6-26</td>
</tr>
<tr>
<td>Rn = CLIP Rx BY Ry</td>
<td>0110 0011</td>
<td>on page 6-27</td>
</tr>
</tbody>
</table>
Floating-Point ALU Operations

Table 6-2. Floating-Point ALU Operations

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Opcode</th>
<th>Reference page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fn = Fx + Fy</td>
<td>1000 0001</td>
<td>on page 6-28</td>
</tr>
<tr>
<td>Fn = Fx – Fy</td>
<td>1000 0010</td>
<td>on page 6-29</td>
</tr>
<tr>
<td>Fn = ABS (Fx + Fy)</td>
<td>1001 0001</td>
<td>on page 6-30</td>
</tr>
<tr>
<td>Fn = ABS (Fx – Fy)</td>
<td>1001 0010</td>
<td>on page 6-31</td>
</tr>
<tr>
<td>Fn = (Fx + Fy)/2</td>
<td>1000 1001</td>
<td>on page 6-32</td>
</tr>
<tr>
<td>Fn = COMP(Fx, Fy)</td>
<td>1000 1010</td>
<td>on page 6-33</td>
</tr>
<tr>
<td>Fn = –Fx</td>
<td>1010 0010</td>
<td>on page 6-34</td>
</tr>
<tr>
<td>Fn = ABS Fx</td>
<td>1011 0000</td>
<td>on page 6-35</td>
</tr>
<tr>
<td>Fn = PASS Fx</td>
<td>1010 0001</td>
<td>on page 6-36</td>
</tr>
<tr>
<td>Fn = RND Fx</td>
<td>1010 0101</td>
<td>on page 6-37</td>
</tr>
<tr>
<td>Fn = SCALB Fx BY Ry</td>
<td>1011 1101</td>
<td>on page 6-38</td>
</tr>
<tr>
<td>Rn = MANT Fx</td>
<td>1010 1101</td>
<td>on page 6-39</td>
</tr>
<tr>
<td>Rn = LOGB Fx</td>
<td>1100 0001</td>
<td>on page 6-40</td>
</tr>
<tr>
<td>Rn = FIX Fx BY Ry</td>
<td>1101 1001</td>
<td>on page 6-41</td>
</tr>
<tr>
<td>Rn = FIX Fx</td>
<td>1100 1001</td>
<td>on page 6-41</td>
</tr>
<tr>
<td>Rn = TRUNC Fx BY Ry</td>
<td>1101 1101</td>
<td>on page 6-41</td>
</tr>
<tr>
<td>Rn = TRUNC Fx</td>
<td>1100 1101</td>
<td>on page 6-41</td>
</tr>
<tr>
<td>Fn = FLOAT Rx BY Ry</td>
<td>1101 1010</td>
<td>on page 6-43</td>
</tr>
<tr>
<td>Fn = FLOAT Rx</td>
<td>1100 1010</td>
<td>on page 6-43</td>
</tr>
<tr>
<td>Fn = RECIPS Fx</td>
<td>1100 0100</td>
<td>on page 6-44</td>
</tr>
<tr>
<td>Fn = RSQRTS Fx</td>
<td>1100 0101</td>
<td>on page 6-46</td>
</tr>
<tr>
<td>Fn = Fx COPYSIGN Fy</td>
<td>1110 0000</td>
<td>on page 6-48</td>
</tr>
<tr>
<td>Fn = MIN(Fx, Fy)</td>
<td>1110 0001</td>
<td>on page 6-49</td>
</tr>
</tbody>
</table>
Table 6-2. Floating-Point ALU Operations (Cont’d)

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Opcode</th>
<th>Reference page</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F_n = \text{MAX}(F_x, F_y) )</td>
<td>1110 0010</td>
<td>on page 6-50</td>
</tr>
<tr>
<td>( F_n = \text{CLIP } F_x \text{ BY } F_y )</td>
<td>1110 0011</td>
<td>on page 6-51</td>
</tr>
</tbody>
</table>
**Rn = Rx + Ry**

**Function**

Adds the fixed-point fields in registers Rx and Ry. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set), positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

**Status Flags**

- **AZ**: Set if the fixed-point output is all 0s, otherwise cleared
- **AU**: Cleared
- **AN**: Set if the most significant output bit is 1, otherwise cleared
- **AV**: Set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
- **AC**: Set if the carry from the most significant adder stage is 1, otherwise cleared
- **AS**: Cleared
- **AI**: Cleared
Compute Field

Rn = Rx – Ry

Function

Subtracts the fixed-point field in register Ry from the fixed-point field in register Rx. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in \texttt{MODE1} set), positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

Status Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZ</td>
<td>Set if the fixed-point output is all 0s, otherwise cleared</td>
</tr>
<tr>
<td>AU</td>
<td>Cleared</td>
</tr>
<tr>
<td>AN</td>
<td>Set if the most significant output bit is 1, otherwise cleared</td>
</tr>
<tr>
<td>AV</td>
<td>Set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared</td>
</tr>
<tr>
<td>AC</td>
<td>Set if the carry from the most significant adder stage is 1, otherwise cleared</td>
</tr>
<tr>
<td>AS</td>
<td>Cleared</td>
</tr>
<tr>
<td>AI</td>
<td>Cleared</td>
</tr>
</tbody>
</table>
Computations Reference

**Rn = Rx + Ry + Cl**

**Function**

Adds with carry (AC from ASTAT) the fixed-point fields in registers Rx and Ry. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set), positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

**Status Flags**

- **AZ** Set if the fixed-point output is all 0s, otherwise cleared
- **AU** Cleared
- **AN** Set if the most significant output bit is 1, otherwise cleared
- **AV** Set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
- **AC** Set if the carry from the most significant adder stage is 1, otherwise cleared
- **AS** Cleared
- **AI** Cleared
Compute Field

\[ R_n = R_x - R_y + C_i - 1 \]

**Function**

Subtracts with borrow \((AC - 1 from ASTAT)\) the fixed-point field in register \(R_y\) from the fixed-point field in register \(R_x\). The result is placed in the fixed-point field in register \(R_n\). The floating-point extension field in \(R_n\) is set to all 0s. In saturation mode (the ALU saturation mode bit in \(MODE1\) set) positive overflows return the maximum positive number \((0x7FFF FFFF)\), and negative overflows return the minimum negative number \((0x8000 0000)\).

**Status Flags**

- **AZ**: Set if the fixed-point output is all 0s, otherwise cleared
- **AU**: Cleared
- **AN**: Set if the most significant output bit is 1, otherwise cleared
- **AV**: Set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
- **AC**: Set if the carry from the most significant adder stage is 1, otherwise cleared
- **AS**: Cleared
- **AI**: Cleared
Rn = (Rx + Ry)/2

Function

Adds the fixed-point fields in registers Rx and Ry and divides the result by 2. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. Rounding is to nearest (IEEE) or by truncation, as defined by the rounding mode bit in the MODE1 register.

Status Flags

- AZ: Set if the fixed-point output is all 0s, otherwise cleared
- AU: Cleared
- AN: Set if the most significant output bit is 1, otherwise cleared
- AV: Cleared
- AC: Set if the carry from the most significant adder stage is 1, otherwise cleared
- AS: Cleared
- AI: Cleared
**Compute Field**

**COMP(Rx, Ry)**

**Function**

Compares the fixed-point field in register Rx with the fixed-point field in register Ry. Sets the $AZ$ flag if the two operands are equal, and the $AN$ flag if the operand in register Rx is smaller than the operand in register Ry.

The ASTAT register stores the results of the previous eight ALU compare operations in bits 24–31. These bits are shifted right (bit 24 is overwritten) whenever a fixed-point or floating-point compare instruction is executed. The MSB of $\text{ASTAT}$ is set if the X operand is greater than the Y operand (its value is the AND of $AZ$ and $AN$); it is otherwise cleared.

**Status Flags**

- **AZ**: Set if the operands in registers Rx and Ry are equal, otherwise cleared
- **AU**: Cleared
- **AN**: Set if the operand in the Rx register is smaller than the operand in the Ry register, otherwise cleared
- **AV**: Cleared
- **AC**: Cleared
- **AS**: Cleared
- **AI**: Cleared
COMPU(Rx, Ry)

Function

Compares the fixed-point field in register Rx with the fixed-point field in register Ry, Sets the AZ flag if the two operands are equal, and the AN flag if the operand in register Rx is smaller than the operand in register Ry. This operation performs a magnitude comparison of the fixed-point contents of Rx and Ry.

The ASTAT register stores the results of the previous eight ALU compare operations in bits 24–31. These bits are shifted right (bit 24 is overwritten) whenever a fixed-point or floating-point compare instruction is executed. The MSB of ASTAT is set if the X operand is greater than the Y operand (its value is the AND of AZ and AN); it is otherwise cleared.

Status Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZ</td>
<td>Is set if the operands in registers Rx and Ry are equal, otherwise cleared</td>
</tr>
<tr>
<td>AU</td>
<td>Is cleared</td>
</tr>
<tr>
<td>AN</td>
<td>Is set if the operand in the Rx register is smaller than the operand in the Ry register, otherwise cleared</td>
</tr>
<tr>
<td>AV</td>
<td>Is cleared</td>
</tr>
<tr>
<td>AC</td>
<td>Is cleared</td>
</tr>
<tr>
<td>AS</td>
<td>Is cleared</td>
</tr>
<tr>
<td>AI</td>
<td>Is cleared</td>
</tr>
</tbody>
</table>
Compute Field

\[ R_n = R_x + C_l \]

Function

Adds the fixed-point field in register \( R_x \) with the carry flag from the \( \text{ASTAT} \) register \((\text{AC})\). The result is placed in the fixed-point field in register \( R_n \). The floating-point extension field in \( R_n \) is set to all 0s. In saturation mode (the ALU saturation mode bit in \( \text{MODE1} \) set) positive overflows return the maximum positive number (0x7FFF FFFF).

Status Flags

- **AZ**: Set if the fixed-point output is all 0s, otherwise cleared
- **AU**: Cleared
- **AN**: Set if the most significant output bit is 1, otherwise cleared
- **AV**: Set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
- **AC**: Set if the carry from the most significant adder stage is 1, otherwise cleared
- **AS**: Cleared
- **AI**: Cleared
\[ R_n = R_x + C_l - 1 \]

**Function**

Adds the fixed-point field in register \( R_x \) with the borrow from the \( A_{STAT} \) register (\( A_C - 1 \)). The result is placed in the fixed-point field in register \( R_n \). The floating-point extension field in \( R_n \) is set to all 0s. In saturation mode (the ALU saturation mode bit in \( M_{ODE1} \) set) positive overflows return the maximum positive number (0x7FFF FFFF).

**Status Flags**

- **AZ**: Set if the fixed-point output is all 0s, otherwise cleared
- **AU**: Cleared
- **AN**: Set if the most significant output bit is 1, otherwise cleared
- **AV**: Set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
- **AC**: Set if the carry from the most significant adder stage is 1, otherwise cleared
- **AS**: Cleared
- **AI**: Cleared
Rn = Rx + 1

Function

Increments the fixed-point operand in register Rx. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set), overflow causes the maximum positive number (0x7FFF FFFF) to be returned.

Status Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZ</td>
<td>Set if the fixed-point output is all 0s, otherwise cleared</td>
</tr>
<tr>
<td>AU</td>
<td>Cleared</td>
</tr>
<tr>
<td>AN</td>
<td>Set if the most significant output bit is 1, otherwise cleared</td>
</tr>
<tr>
<td>AV</td>
<td>Set if the XOR of the carries of the two most significant adder, stages is 1, otherwise cleared</td>
</tr>
<tr>
<td>AC</td>
<td>Set if the carry from the most significant adder stage is 1, otherwise cleared</td>
</tr>
<tr>
<td>AS</td>
<td>Cleared</td>
</tr>
<tr>
<td>AI</td>
<td>Cleared</td>
</tr>
</tbody>
</table>
Rn = Rx – 1

Function

Decrement the fixed-point operand in register Rx. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set), underflow causes the minimum negative number (0x8000 0000) to be returned.

Status Flags

<table>
<thead>
<tr>
<th>AZ</th>
<th>Set if the fixed-point output is all 0s, otherwise cleared</th>
</tr>
</thead>
<tbody>
<tr>
<td>AU</td>
<td>Cleared</td>
</tr>
<tr>
<td>AN</td>
<td>Set if the most significant output bit is 1, otherwise cleared</td>
</tr>
<tr>
<td>AV</td>
<td>Set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared</td>
</tr>
<tr>
<td>AC</td>
<td>Set if the carry from the most significant adder stage is 1, otherwise cleared</td>
</tr>
<tr>
<td>AS</td>
<td>Cleared</td>
</tr>
<tr>
<td>AI</td>
<td>Cleared</td>
</tr>
</tbody>
</table>
Compute Field

\[ Rn = -Rx \]

Function

Negates the fixed-point operand in Rx by two’s-complement. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. Negation of the minimum negative number (0x8000 0000) causes an overflow. In saturation mode (the ALU saturation mode bit in MODE1 set), overflow causes the maximum positive number (0x7FFF FFFF) to be returned.

Status Flags

- **AZ**: Set if the fixed-point output is all 0s
- **AU**: Cleared
- **AN**: Set if the most significant output bit is 1
- **AV**: Set if the XOR of the carries of the two most significant adder stages is 1
- **AC**: Set if the carry from the most significant adder stage is 1, otherwise cleared
- **AS**: Cleared
- **AI**: Cleared
**Rn = ABS Rx**

**Function**

Determines the absolute value of the fixed-point operand in Rx. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. The ABS of the minimum negative number (0x8000 0000) causes an overflow. In saturation mode (the ALU saturation mode bit in `MODE1` set), overflow causes the maximum positive number (0x7FFF FFFF) to be returned.

**Status Flags**

- **AZ**: Set if the fixed-point output is all 0s, otherwise cleared
- **AU**: Cleared
- **AN**: Set if the most significant output bit is 1, otherwise cleared
- **AV**: Set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
- **AC**: Set if the carry from the most significant adder stage is 1, otherwise cleared
- **AS**: Set if the fixed-point operand in Rx is negative, otherwise cleared
- **AI**: Cleared
Rn = PASS Rx

Function

Passes the fixed-point operand in Rx through the ALU to the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s.

Status Flags

AZ  Set if the fixed-point output is all 0s, otherwise cleared
AU  Cleared
AN  Set if the most significant output bit is 1, otherwise cleared
AV  Cleared
AC  Cleared
AS  Cleared
AI  Cleared
**Rn = Rx AND Ry**

**Function**

Logically ANDs the fixed-point operands in Rx and Ry. The result is placed in the fixed-point field in Rn. The floating-point extension field in Rn is set to all 0s.

**Status Flags**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZ</td>
<td>Set if the fixed-point output is all 0s, otherwise cleared</td>
</tr>
<tr>
<td>AU</td>
<td>Cleared</td>
</tr>
<tr>
<td>AN</td>
<td>Set if the most significant output bit is 1, otherwise cleared</td>
</tr>
<tr>
<td>AV</td>
<td>Cleared</td>
</tr>
<tr>
<td>AC</td>
<td>Cleared</td>
</tr>
<tr>
<td>AS</td>
<td>Cleared</td>
</tr>
<tr>
<td>AI</td>
<td>Cleared</td>
</tr>
</tbody>
</table>
Compute Field

\[ \text{Rn} = \text{Rx OR Ry} \]

Function

Logically ORs the fixed-point operands in Rx and Ry. The result is placed in the fixed-point field in Rn. The floating-point extension field in Rn is set to all 0s.

Status Flags

- **AZ**: Set if the fixed-point output is all 0s, otherwise cleared
- **AU**: Cleared
- **AN**: Set if the most significant output bit is 1, otherwise cleared
- **AV**: Cleared
- **AC**: Cleared
- **AS**: Cleared
- **AI**: Cleared
**Rn = Rx XOR Ry**

**Function**

Logically XORs the fixed-point operands in Rx and Ry. The result is placed in the fixed-point field in Rn. The floating-point extension field in Rn is set to all 0s.

**Status Flags**

- **AZ**
  - Set if the fixed-point output is all 0s, otherwise cleared
- **AU**
  - Cleared
- **AN**
  - Set if the most significant output bit is 1, otherwise cleared
- **AV**
  - Cleared
- **AC**
  - Cleared
- **AS**
  - Cleared
- **AI**
  - Cleared
Compute Field

Rn = NOT Rx

Function

Logically complements the fixed-point operand in Rx. The result is placed in the fixed-point field in Rn. The floating-point extension field in Rn is set to all 0s.

Status Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZ</td>
<td>Set if the fixed-point output is all 0s, otherwise cleared</td>
</tr>
<tr>
<td>AU</td>
<td>Cleared</td>
</tr>
<tr>
<td>AN</td>
<td>Set if the most significant output bit is 1, otherwise cleared</td>
</tr>
<tr>
<td>AV</td>
<td>Cleared</td>
</tr>
<tr>
<td>AC</td>
<td>Cleared</td>
</tr>
<tr>
<td>AS</td>
<td>Cleared</td>
</tr>
<tr>
<td>AI</td>
<td>Cleared</td>
</tr>
</tbody>
</table>
Rn = MIN(Rx, Ry)

Function

Returns the smaller of the two fixed-point operands in Rx and Ry. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s.

Status Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZ</td>
<td>Set if the fixed-point output is all 0s, otherwise cleared</td>
</tr>
<tr>
<td>AU</td>
<td>Cleared</td>
</tr>
<tr>
<td>AN</td>
<td>Set if the most significant output bit is 1, otherwise cleared</td>
</tr>
<tr>
<td>AV</td>
<td>Cleared</td>
</tr>
<tr>
<td>AC</td>
<td>Cleared</td>
</tr>
<tr>
<td>AS</td>
<td>Cleared</td>
</tr>
<tr>
<td>AI</td>
<td>Cleared</td>
</tr>
</tbody>
</table>
Compute Field

\( Rn = \text{MAX}(Rx, Ry) \)

**Function**

Returns the larger of the two fixed-point operands in Rx and Ry. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s.

**Status Flags**

- **AZ**: Set if the fixed-point output is all 0s, otherwise cleared
- **AU**: Cleared
- **AN**: Set if the most significant output bit is 1, otherwise cleared
- **AV**: Cleared
- **AC**: Cleared
- **AS**: Cleared
- **AI**: Cleared
**Rn = CLIP Rx BY Ry**

**Function**

Returns the fixed-point operand in Rx if the absolute value of the operand in Rx is less than the absolute value of the fixed-point operand in Ry. Otherwise, returns $|\text{Ry}|$ if Rx is positive, and $-|\text{Ry}|$ if Rx is negative. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s.

**Status Flags**

- **AZ** Set if the fixed-point output is all 0s, otherwise cleared
- **AU** Cleared
- **AN** Set if the most significant output bit is 1, otherwise cleared
- **AV** Cleared
- **AC** Cleared
- **AS** Cleared
- **AI** Cleared
**Fn = Fx + Fy**

**Function**

Adds the floating-point operands in registers Fx and Fy. The normalized result is placed in register Fn. Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in MODE1. Post-rounded overflow returns ±Infinity (round-to-nearest) or ±NORM.MAX (round-to-zero).
Post-rounded denormal returns ±Zero. Denormal inputs are flushed to ±Zero. A NAN input returns an all 1s result.

**Status Flags**

- **AZ** Set if the post-rounded result is a denormal (unbiased exponent < −126) or zero, otherwise cleared
- **AU** Set if the post-rounded result is a denormal, otherwise cleared
- **AN** Set if the floating-point result is negative, otherwise cleared
- **AV** Set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared
- **AC** Cleared
- **AS** Cleared
- **AI** Set if either of the input operands is a NAN, or if they are opposite-signed Infinities, otherwise cleared
**Fn = Fx – Fy**

**Function**

Subtracts the floating-point operand in register Fy from the floating-point operand in register Fx. The normalized result is placed in register Fn. Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in MODE1. Post-rounded overflow returns ±Infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). Post-rounded denormal returns ±Zero. Denormal inputs are flushed to ±Zero. A NAN input returns an all 1s result.

**Status Flags**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZ</td>
<td>Set if the post-rounded result is a denormal (unbiased exponent &lt; −126) or zero, otherwise cleared</td>
</tr>
<tr>
<td>AU</td>
<td>Set if the post-rounded result is a denormal, otherwise cleared</td>
</tr>
<tr>
<td>AN</td>
<td>Set if the floating-point result is negative, otherwise cleared</td>
</tr>
<tr>
<td>AV</td>
<td>Set if the post-rounded result overflows (unbiased exponent &gt; +127), otherwise cleared</td>
</tr>
<tr>
<td>AC</td>
<td>Cleared</td>
</tr>
<tr>
<td>AS</td>
<td>Cleared</td>
</tr>
<tr>
<td>AI</td>
<td>Set if either of the input operands is a NAN, or if they are like-signed Infinities, otherwise cleared</td>
</tr>
</tbody>
</table>
Compute Field

\[ \text{Fn} = \text{ABS}(\text{Fx} + \text{Fy}) \]

Function

Add the floating-point operands in registers \( \text{Fx} \) and \( \text{Fy} \), and places the absolute value of the normalized result in register \( \text{Fn} \). Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in \text{MODE1}.

Post-rounded overflow returns +Infinity (round-to-nearest) or +NORM.MAX (round-to-zero). Post-rounded denormal returns +Zero. Denormal inputs are flushed to ±Zero. A NAN input returns an all 1s result.

Status Flags

- **AZ**: Set if the post-rounded result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
- **AU**: Set if the post-rounded result is a denormal, otherwise cleared
- **AN**: Cleared
- **AV**: Set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared
- **AC**: Cleared
- **AS**: Cleared
- **AI**: Set if either of the input operands is a NAN, or if they are opposite-signed Infinities, otherwise cleared
Pressure = \text{ABS} (F_x - F_y)

Function

Subtracts the floating-point operand in \( F_y \) from the floating-point operand in \( F_x \) and places the absolute value of the normalized result in register \( F_n \). Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in \( \text{MODE1} \). Post-rounded overflow returns +\text{Infinity} (round-to-nearest) or +\text{NORM.MAX} (round-to-zero). Post-rounded denormal returns +\text{Zero}. Denormal inputs are flushed to ±\text{Zero}. A \text{NAN} input returns an all 1s result.

Status Flags

- **AZ**: Set if the post-rounded result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
- **AU**: Set if the post-rounded result is a denormal, otherwise cleared
- **AN**: Cleared
- **AV**: Set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared
- **AC**: Cleared
- **AS**: Cleared
- **AI**: Set if either of the input operands is a \text{NAN}, or if they are like-signed \text{Infinities}, otherwise cleared
Compute Field

\[ Fn = (F_x + F_y)/2 \]

**Function**

Adds the floating-point operands in registers \( F_x \) and \( F_y \) and divides the result by 2, by decrementing the exponent of the sum before rounding. The normalized result is placed in register \( F_n \). Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in `MODE1`. Post-rounded overflow returns \( \pm \text{Infinity} \) (round-to-nearest) or \( \pm \text{NORM.MAX} \) (round-to-zero). Post-rounded denormal results return \( \pm \text{Zero} \). A denormal input is flushed to \( \pm \text{Zero} \). A NAN input returns an all 1s result.

**Status Flags**

- **AZ**: Set if the post-rounded result is a denormal (unbiased exponent < \(-126\)) or zero, otherwise cleared
- **AU**: Set if the post-rounded result is a denormal, otherwise cleared
- **AN**: Set if the floating-point result is negative, otherwise cleared
- **AV**: Set if the post-rounded result overflows (unbiased exponent > \(+127\)), otherwise cleared
- **AC**: Cleared
- **AS**: Cleared
- **AI**: Set if either of the input operands is a NAN, or if they are opposite-signed Infinities, otherwise cleared
**COMP(Fx, Fy)**

**Function**

Compares the floating-point operand in register Fx with the floating-point operand in register Fy. Sets the $AZ$ flag if the two operands are equal, and the $AN$ flag if the operand in register Fx is smaller than the operand in register Fy.

The $ASTAT$ register stores the results of the previous eight ALU compare operations in bits 24-31. These bits are shifted right (bit 24 is overwritten) whenever a fixed-point or floating-point compare instruction is executed. The MSB of $ASTAT$ is set if the X operand is greater than the Y operand (its value is the AND of $AZ$ and $AN$); it is otherwise cleared.

**Status Flags**

- **AZ** Set if the operands in registers Fx and Fy are equal, otherwise cleared
- **AU** Cleared
- **AN** Set if the operand in the Fx register is smaller than the operand in the Fy register, otherwise cleared
- **AV** Cleared
- **AC** Cleared
- **AS** Cleared
- **AI** Set if either of the input operands is a NaN, otherwise cleared
Compute Field

\( Fn = \overline{Fx} \)

Function

Complements the sign bit of the floating-point operand in \( Fx \). The complemented result is placed in register \( Fn \). A denormal input is flushed to \( \pm \)Zero. A NAN input returns an all 1s result.

Status Flags

- **AZ**: Set if the result operand is a \( \pm \)Zero, otherwise cleared
- **AU**: Cleared
- **AN**: Set if the floating-point result is negative, otherwise cleared
- **AV**: Cleared
- **AC**: Cleared
- **AS**: Cleared
- **AI**: Set if the input operand is a NAN, otherwise cleared
**Fn = ABS Fx**

**Function**

Returns the absolute value of the floating-point operand in register Fx by setting the sign bit of the operand to 0. Denormal inputs are flushed to +Zero. A NaN input returns an all 1s result.

**Status Flags**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZ</td>
<td>Set if the result operand is +Zero, otherwise cleared</td>
</tr>
<tr>
<td>AU</td>
<td>Cleared</td>
</tr>
<tr>
<td>AN</td>
<td>Cleared</td>
</tr>
<tr>
<td>AV</td>
<td>Cleared</td>
</tr>
<tr>
<td>AC</td>
<td>Cleared</td>
</tr>
<tr>
<td>AS</td>
<td>Set if the input operand is negative, otherwise cleared</td>
</tr>
<tr>
<td>AI</td>
<td>Set if the input operand is a NaN, otherwise cleared</td>
</tr>
</tbody>
</table>
Compute Field

**Fn = PASS Fx**

**Function**

Passes the floating-point operand in Fx through the ALU to the floating-point field in register Fn. Denormal inputs are flushed to ±Zero. A NAN input returns an all 1s result.

**Status Flags**

- **AZ** Set if the result operand is a ±Zero, otherwise cleared
- **AU** Cleared
- **AN** Set if the floating-point result is negative, otherwise cleared
- **AV** Cleared
- **AC** Cleared
- **AS** Cleared
- **AI** Set if the input operand is a NAN, otherwise cleared
**Fn = RND Fx**

**Function**

Rounds the floating-point operand in register Fx to a 32 bit boundary. Rounding is to nearest (IEEE) or by truncation, as defined by the rounding mode bit in `MODE1`. Post-rounded overflow returns ±\(\text{Infinity} \) (round-to-nearest) or ±\(\text{NORM.MAX} \) (round-to-zero). A denormal input is flushed to ±\(\text{Zero} \). A NAN input returns an all 1s result.

**Status Flags**

- **AZ**
  - Set if the result operand is a ±\(\text{Zero} \), otherwise cleared

- **AU**
  - Cleared

- **AN**
  - Set if the floating-point result is negative, otherwise cleared

- **AV**
  - Set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared

- **AC**
  - Cleared

- **AS**
  - Cleared

- **AI**
  - Set if the input operand is a NAN, otherwise cleared
Compute Field

\[ \text{Fn} = \text{SCALB} \ Fx \ BY \ Ry \]

Function

Scales the exponent of the floating-point operand in \( Fx \) by adding to it the fixed-point twos-complement integer in \( Ry \). The scaled floating-point result is placed in register \( Fn \). Overflow returns \( \pm\text{Infinity} \) (round-to-nearest) or \( \pm\text{NORM.MAX} \) (round-to-zero). Denormal returns \( \pm\text{Zero} \). Denormal inputs are flushed to \( \pm\text{Zero} \). A NAN input returns an all 1s result.

Status Flags

- **AZ**: Set if the result is a denormal (unbiased exponent < \(-126\)) or zero, otherwise cleared
- **AU**: Set if the post-rounded result is a denormal, otherwise cleared
- **AN**: Set if the floating-point result is negative, otherwise cleared
- **AV**: Set if the result overflows (unbiased exponent > \(+127\)), otherwise cleared
- **AC**: Cleared
- **AS**: Cleared
- **AI**: Set if the input is a NAN, an otherwise cleared
Rn = MANT Fx

Function

Extracts the mantissa (fraction bits with explicit hidden bit, excluding the sign bit) from the floating-point operand in Fx. The unsigned-magnitude result is left-justified (1.31 format) in the fixed-point field in Rn. Rounding modes are ignored and no rounding is performed because all results are inherently exact. Denormal inputs are flushed to ±Zero. A NAN or an Infinity input returns an all 1s result (–1 in signed fixed-point format).

Status Flags

AZ Set if the result is zero, otherwise cleared
AN Cleared
AV Set if the input operand is an infinity, otherwise cleared.
AC Cleared
AS Set if the input is negative, otherwise cleared
AI Set if the input operand is a NAN, otherwise cleared
Rn = LOGB Fx

Function

Converts the exponent of the floating-point operand in register Fx to an unbiased two’s-complement fixed-point integer. The result is placed in the fixed-point field in register Rn. Unbiasing is done by subtracting 127 from the floating-point exponent in Fx. If saturation mode is not set, a ±Infinity input returns a floating-point +Infinity and a ±Zero input returns a floating-point –Infinity. If saturation mode is set, a ±Infinity input returns the maximum positive value (0x7FFF FFFF), and a ±Zero input returns the maximum negative value (0x8000 0000). Denormal inputs are flushed to ±Zero. A NAN input returns an all 1s result.

Status Flags

AZ  Set if the fixed-point result is zero, otherwise cleared
AU  Cleared
AN  Set if the result is negative, otherwise cleared
AV  Set if the input operand is an Infinity or a Zero, otherwise cleared
AC  Cleared
AS  Cleared
AI  Set if the input is a NAN, otherwise cleared
Function

Converts the floating-point operand in Fx to a two’s-complement 32-bit fixed-point integer result.

If the \texttt{MODE1} register \texttt{TRUNC} bit=1, the Fix operation truncates the mantissa towards –Infinity. If the \texttt{TRUNC} bit=0, the Fix operation rounds the mantissa towards the nearest integer.

The Trunc operation always truncates toward 0. The \texttt{TRUNC} bit does not influence operation of the Trunc instruction.

If a scaling factor (Ry) is specified, the fixed-point two’s-complement integer in Ry is added to the exponent of the floating-point operand in Fx before the conversion.

The result of the conversion is right-justified (32.0 format) in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s.

In saturation mode (the ALU saturation mode bit in \texttt{MODE1} set) positive overflows and +Infinity return the maximum positive number (0x7FFF FFFF), and negative overflows and –Infinity return the minimum negative number (0x8000 0000).

For the Fix operation, rounding is to nearest (IEEE) or by truncation, as defined by the rounding mode bit in \texttt{MODE1}. A NAN input returns a floating-point all 1s result. If saturation mode is not set, an Infinity input or a result that overflows returns a floating-point result of all 1s.
Compute Field

All positive underflows return zero. Negative underflows that are rounded-to-nearest return zero, and negative underflows that are rounded by truncation return –1 (0xFF FFFF FF00).

**Status Flags**

- **AZ**  Set if the fixed-point result is Zero, otherwise cleared
- **AU**  Set if the pre-rounded result is a denormal, otherwise cleared
- **AN**  Set if the fixed-point result is negative, otherwise cleared
- **AV**  Set if the conversion causes the floating-point mantissa to be shifted left, that is, if the floating-point exponent + scale bias is >157 (127 + 31 – 1) or if the input is ±Infinity, otherwise cleared
- **AC**  Cleared
- **AS**  Cleared
- **AI**  Set if the input operand is a NAN or, when saturation mode is not set, either input is an Infinity or the result overflows, otherwise cleared
**Fn = FLOAT Rx BY Ry**

**Fn = FLOAT Rx**

**Function**

Converts the fixed-point operand in Rx to a floating-point result. If a scaling factor (Ry) is specified, the fixed-point twos-complement integer in Ry is added to the exponent of the floating-point result. The final result is placed in register Fn. Rounding is to nearest (IEEE) or by truncation, as defined by the rounding mode, to a 40-bit boundary, regardless of the values of the rounding boundary bits in MODE1. The exponent scale bias may cause a floating-point overflow or a floating-point underflow. Overflow generates a return of ±Infinity (round-to-nearest) or ±NORM.MAX (round-to-zero); underflow generates a return of ±Zero.

**Status Flags**

- **AZ** Set if the result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
- **AU** Set if the post-rounded result is a denormal, otherwise cleared
- **AN** Set if the floating-point result is negative, otherwise cleared
- **AV** Set if the result overflows (unbiased exponent >127)
- **AC** Cleared
- **AS** Cleared
- **AI** Cleared
Compute Field

\[ \text{Fn} = \text{RECIPS} \ Fx \]

Function

Creates an 8-bit accurate seed for \(1/Fx\), the reciprocal of \(Fx\). The mantissa of the seed is determined from a ROM table using the 7 MSBs (excluding the hidden bit) of the \(Fx\) mantissa as an index. The unbiased exponent of the seed is calculated as the twos complement of the unbiased \(Fx\) exponent, decremented by one; that is, if \(e\) is the unbiased exponent of \(Fx\), then the unbiased exponent of \(Fn = -e - 1\). The sign of the seed is the sign of the input. A \(\pm\)Zero returns \(\pm\)Infinity and sets the overflow flag. If the unbiased exponent of \(Fx\) is greater than \(+125\), the result is \(\pm\)Zero. A NAN input returns an all 1s result.

The following code performs floating-point division using an iterative convergence algorithm.\(^1\) The result is accurate to one LSB in whichever format mode, 32-bit or 40-bit, is set (32-bit only for ADSP-21010). The following inputs are required: \(F0=\)numerator, \(F12=\)denominator, \(F11=2.0\). The quotient is returned in \(F0\). (The two highlighted instructions can be removed if only a \(\pm1\) LSB accurate single-precision result is necessary.)

\[
\begin{align*}
F0 &= \text{RECIPS} \ F12, \ F7 = F0; \quad (\text{Get 8 bit seed } R0=1/D) \\
F12 &= F0*F12; \quad (D' = D*R0) \\
F7 &= F0*F7, \ F0 = F11 - F12; \quad (F0=R1=2-D', \ F7=N*R0) \\
F12 &= F0*F12; \quad (F12=D'-D'*R1) \\
F7 &= F0*F7, \ F0 = F11 - F12; \quad (F7=N*R0*R1, \ F0=R2=2-D') \\
F12 &= F0*F12; \quad (F12=D'-D'*R2) \\
F7 &= F0*F7, \ F0 = F11 - F12; \quad (F7=N*R0*R1*R2, \ F0=R3=2-D') \\
F0 &= F0*F7; \quad (F7=N*R0*R1*R2*R3)
\end{align*}
\]

To make this code segment a subroutine, add an RTS(DB) clause to the third-to-last instruction. The 2nd and 3rd last instructions can be removed if only a ±1 LSB accurate single-precision result is necessary.

**Status Flags**

| AZ  | Set if the floating-point result is ±Zero (unbiased exponent of Fx is greater than +125), otherwise cleared |
| AU  | Cleared |
| AN  | Set if the input operand is negative, otherwise cleared |
| AV  | Set if the input operand is ±Zero, otherwise cleared |
| AC  | Cleared |
| AS  | Cleared |
| AI  | Set if the input operand is a NAN, otherwise cleared |
Fn = RSQRTS Fx

Function

Creates a 4-bit accurate seed for $1/(Fx)^{1/2}$, the reciprocal square root of Fx.

The mantissa of the seed is determined from a ROM table, using the LSB of the biased exponent of Fx concatenated with the six MSBs (excluding the hidden bit of the mantissa) of Fx’s an index.

The unbiased exponent of the seed is calculated as the twos complement of the unbiased Fx exponent, shifted right by one bit and decremented by one; that is, if $e$ is the unbiased exponent of Fx, then the unbiased exponent of $Fn = -\text{INT}[e/2] - 1$.

The sign of the seed is the sign of the input. The input ±Zero returns ±Infinity and sets the overflow flag. The input +Infinity returns +Zero. A NAN input or a negative nonzero input returns a result of all 1s.

The following code calculates a floating-point reciprocal square root $(1/(x)^{1/2})$ using a Newton-Raphson iteration algorithm.\(^1\) The result is accurate to one LSB in whichever format mode, 32-bit or 40-bit, is set (32-bit only for ADSP-21010).

To calculate the square root, simply multiply the result by the original input. The following inputs are required: $F0=\text{input}$, $F8=3.0$, $F1=0.5$. The result is returned in $F4$. (The four highlighted instructions can be removed if only a ±1 LSB accurate single-precision result is necessary.)

\[
\begin{align*}
F4 &= \text{RSQRTS } F0; & \text{(Fetch 4-bit seed)} \\
F12 &= F4 \times F4; & \text{($F12=X0^2$)} \\
F12 &= F12 \times F0; & \text{($F12=C \times X0^2$)} \\
F4 &= F1 \times F4, F12=F8-F12; & \text{($F4=0.5 \times X0$, $F12=3-C \times X0^2$)} \\
F4 &= F4 \times F12; & \text{($F4=X1=0.5 \times X0(3-C \times X0^2)$)}
\end{align*}
\]

F12=F4*F4;   \( F12=X_1^2 \)
F12=F12*F0;   \( F12=C*X_1^2 \)
F4=F1*F4, F12=F8-F12;   \( F4=.5*X_1, F12=3-C*X_1^2 \)
F4=F4*F12;   \( F4=X_2=.5*X_1(3-C*X_1^2) \)
F12=F4*F4;   \( F12=X_2^2 \)
F12=F12*F0;   \( F12=C*X_2^2 \)
F4=F1*F4, F12=F8-F12;   \( F4=.5*X_2, F12=3-C*X_2^2 \)
F4=F4*F12;   \( F4=X_3=.5*X_2(3-C*X_2^2) \)

Note that this code segment can be made into a subroutine by adding an RTS(DB) clause to the third-to-last instruction. Also, the 2\textsuperscript{nd}, 3\textsuperscript{rd}, 4\textsuperscript{th}, and 5\textsuperscript{th} last instructions can be removed if only a ±1 LSB accurate single-precision result is necessary.

**Status Flags**

- **AZ**: Set if the floating-point result is +Zero (Fx = +Infinity), otherwise cleared
- **AU**: Cleared
- **AN**: Set if the input operand is –Zero, otherwise cleared
- **AV**: Set if the input operand is ±Zero, otherwise cleared
- **AC**: Cleared
- **AS**: Cleared
- **AI**: Set if the input operand is negative and nonzero, or a NAN, otherwise cleared
Compute Field

Fn = Fx COPYSIGN Fy

Function

Copies the sign of the floating-point operand in register Fy to the floating-point operand from register Fx without changing the exponent or the mantissa. The result is placed in register Fn. A denormal input is flushed to ±Zero. A NAN input returns an all 1s result.

Status Flags

- **AZ**: Set if the floating-point result is ±Zero, otherwise cleared
- **AU**: Cleared
- **AN**: Set if the floating-point result is negative, otherwise cleared
- **AV**: Cleared
- **AC**: Cleared
- **AS**: Cleared
- **AI**: Set if either of the input operands is a NAN, otherwise cleared
\[ F_n = \text{MIN}(F_x, F_y) \]

**Function**

Returns the smaller of the floating-point operands in register \( F_x \) and \( F_y \). A \( \text{NAN} \) input returns an all 1s result. The \( \text{MIN} \) of \( +\text{Zero} \) and \( -\text{Zero} \) returns \( -\text{Zero} \). Denormal inputs are flushed to ±\( \text{Zero} \).

**Status Flags**

- **AZ** Set if the floating-point result is ±\( \text{Zero} \), otherwise cleared
- **AU** Cleared
- **AN** Set if the floating-point result is negative, otherwise cleared
- **AV** Cleared
- **AC** Cleared
- **AS** Cleared
- **AI** Set if either of the input operands is a \( \text{NAN} \), otherwise cleared
$$Fn = \text{MAX}(Fx, Fy)$$

Function

Returns the larger of the floating-point operands in registers $Fx$ and $Fy$. A $\text{NAN}$ input returns an all 1s result. The $\text{MAX}$ of $+\text{Zero}$ and $-\text{Zero}$ returns $+\text{Zero}$. Denormal inputs are flushed to $\pm\text{Zero}$.

Status Flags

| AZ  | Set if the floating-point result is $\pm\text{Zero}$, otherwise cleared |
| AU  | Cleared |
| AN  | Set if the floating-point result is negative, otherwise cleared |
| AV  | Cleared |
| AC  | Cleared |
| AS  | Cleared |
| AI  | Set if either of the input operands is a $\text{NAN}$, otherwise cleared |
**Fn = CLIP Fx BY Fy**

**Function**

Returns the floating-point operand in Fx if the absolute value of the operand in Fx is less than the absolute value of the floating-point operand in Fy. Else, returns $|Fy|$ if Fx is positive, and $-|Fy|$ if Fx is negative. A NAN input returns an all 1s result. Denormal inputs are flushed to ±Zero.

**Status Flags**

- **AZ**: Set if the floating-point result is ±Zero, otherwise cleared
- **AU**: Cleared
- **AN**: Set if the floating-point result is negative, otherwise cleared
- **AV**: Cleared
- **AC**: Cleared
- **AS**: Cleared
- **AI**: Set if either of the input operands is a NAN, otherwise cleared

**Multiplier Operations**

This section describes the multiplier operations. Table 6-3 and Table 6-4 on page 6-54 summarize the syntax and opcodes for the fixed-point and floating-point multiplier operations, respectively. These tables use the following symbols to indicate location of operands and other features:

- $y = y\text{-input (1 = signed, 0 = unsigned)}$
- $x = x\text{-input (1 = signed, 0 = unsigned)}$
Compute Field

- \( f \) = format (1 = fractional, 0 = integer)
- \( r \) = rounding (1 = yes, 0 = no)
## Multiplier Fixed-Point Operations

Table 6-3. Fixed-Point Multiplier Operations

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Opcode</th>
<th>Reference Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn = Rx*Ry mod2</td>
<td>01yx f00r</td>
<td>on page 6-56</td>
</tr>
<tr>
<td>MRF = Rx*Ry mod2</td>
<td>01yx f10r</td>
<td>on page 6-56</td>
</tr>
<tr>
<td>MRB = Rx*Ry mod2</td>
<td>01yx f11r</td>
<td>on page 6-56</td>
</tr>
<tr>
<td>Rn = MRF +Rx*Ry mod2</td>
<td>10yx f00r</td>
<td>on page 6-57</td>
</tr>
<tr>
<td>Rn = MRB +Rx*Ry mod2</td>
<td>10yx f01r</td>
<td>on page 6-57</td>
</tr>
<tr>
<td>MRF = MRF +Rx*Ry mod2</td>
<td>10yx f10r</td>
<td>on page 6-57</td>
</tr>
<tr>
<td>MRB = MRB +Rx*Ry mod2</td>
<td>10yx f11r</td>
<td>on page 6-57</td>
</tr>
<tr>
<td>Rn = MRF –Rx*Ry mod2</td>
<td>11yx f00r</td>
<td>on page 6-58</td>
</tr>
<tr>
<td>Rn = MRB –Rx*Ry mod2</td>
<td>11yx f01r</td>
<td>on page 6-58</td>
</tr>
<tr>
<td>MRF = MRF –Rx*Ry mod2</td>
<td>11yx f10r</td>
<td>on page 6-58</td>
</tr>
<tr>
<td>MRB = MRB –Rx*Ry mod2</td>
<td>11yx f11r</td>
<td>on page 6-58</td>
</tr>
<tr>
<td>Rn = SAT MRF mod1</td>
<td>0000 f00x</td>
<td>on page 6-59</td>
</tr>
<tr>
<td>Rn = SAT MRB mod1</td>
<td>0000 f01x</td>
<td>on page 6-59</td>
</tr>
<tr>
<td>MRF = SAT MRF mod1</td>
<td>0000 f10x</td>
<td>on page 6-59</td>
</tr>
<tr>
<td>MRB = SAT MRB mod1</td>
<td>0000 f11x</td>
<td>on page 6-59</td>
</tr>
<tr>
<td>Rn =RND MRF mod1</td>
<td>0001 100x</td>
<td>on page 6-60</td>
</tr>
<tr>
<td>Rn = RND MRB mod1</td>
<td>0001 101x</td>
<td>on page 6-60</td>
</tr>
<tr>
<td>MRF = RND MRF mod1</td>
<td>0001 110x</td>
<td>on page 6-60</td>
</tr>
<tr>
<td>MRB = RND MRB mod1</td>
<td>0001 111x</td>
<td>on page 6-60</td>
</tr>
<tr>
<td>MRF = 0</td>
<td>0001 0100</td>
<td>on page 6-61</td>
</tr>
<tr>
<td>MRB = 0</td>
<td>0001 0110r</td>
<td>on page 6-61</td>
</tr>
<tr>
<td>MR = Rn</td>
<td></td>
<td>on page 6-62</td>
</tr>
<tr>
<td>Rn = MR</td>
<td></td>
<td>on page 6-62</td>
</tr>
</tbody>
</table>
Multiplier Floating-Point Operations

Table 6-4. Floating-Point Multiplier Operations

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Opcode</th>
<th>Reference Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_n = F_x \times F_y$</td>
<td>0011 0000</td>
<td>on page 6-64</td>
</tr>
</tbody>
</table>

Mod1 and Mod2 Modifiers

Mod2 in Table 6-3 on page 6-53 is an optional modifier. It is enclosed in parentheses and consists of three or four letters that indicate whether:

- The x-input is signed (S) or unsigned (U)
- The y-input is signed or unsigned
- The inputs are in integer (I) or fractional (F) format
- The result written to the register file will be rounded-to-nearest (R).

Table 6-5 lists the options for mod2 and the corresponding opcode values.

Table 6-5. Mod2 Options and Opcodes

<table>
<thead>
<tr>
<th>Option</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SSI)</td>
<td><em>.11 0</em>.0</td>
</tr>
<tr>
<td>(SUI)</td>
<td><em>.01 0</em>.0</td>
</tr>
<tr>
<td>(USI)</td>
<td><em>.10 0</em>.0</td>
</tr>
<tr>
<td>(UUI)</td>
<td><em>.00 0</em>.0</td>
</tr>
<tr>
<td>(SSF)</td>
<td><em>.11 1</em>.0</td>
</tr>
<tr>
<td>(SUF)</td>
<td><em>.01 1</em>.0</td>
</tr>
<tr>
<td>(USF)</td>
<td><em>.10 1</em>.0</td>
</tr>
<tr>
<td>(UUF)</td>
<td><em>.00 1</em>.0</td>
</tr>
</tbody>
</table>
Similarly, mod1 in Table 6-3 on page 6-53 is an optional modifier, enclosed in parentheses, consisting of two letters that indicate whether the input is signed (S) or unsigned (U) and whether the input is in integer (I) or fractional (F) format. The options for mod1 and the corresponding opcode values are listed in Table 6-6.

Table 6-6. Mod1 Options and Opcodes

<table>
<thead>
<tr>
<th>Option</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SI) (for SAT only)</td>
<td>_ _ _ _ 0 _ _ 1</td>
</tr>
<tr>
<td>(UI) (for SAT only)</td>
<td>_ _ _ _ 0 _ _ 0</td>
</tr>
<tr>
<td>(SF)</td>
<td>_ _ _ _ 1 _ _ 1</td>
</tr>
<tr>
<td>(UF)</td>
<td>_ _ _ _ 1 _ _ 0</td>
</tr>
</tbody>
</table>

Similarly, mod1 in Table 6-3 on page 6-53 is an optional modifier, enclosed in parentheses, consisting of two letters that indicate whether the input is signed (S) or unsigned (U) and whether the input is in integer (I) or fractional (F) format. The options for mod1 and the corresponding opcode values are listed in Table 6-6.

Table 6-6. Mod1 Options and Opcodes

<table>
<thead>
<tr>
<th>Option</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SI) (for SAT only)</td>
<td>_ _ _ _ 0 _ _ 1</td>
</tr>
<tr>
<td>(UI) (for SAT only)</td>
<td>_ _ _ _ 0 _ _ 0</td>
</tr>
<tr>
<td>(SF)</td>
<td>_ _ _ _ 1 _ _ 1</td>
</tr>
<tr>
<td>(UF)</td>
<td>_ _ _ _ 1 _ _ 0</td>
</tr>
</tbody>
</table>
Compute Field

\[ Rn = Rx \times Ry \mod 2 \]
\[ MRF = Rx \times Ry \mod 2 \]
\[ MRB \times Ry \mod 2 \]

Function

Multiplies the fixed-point fields in registers Rx and Ry.

If rounding is specified (fractional data only), the result is rounded. The result is placed either in the fixed-point field in register Rn or one of the accumulation registers.

If Rn is specified, only the portion of the result that has the same format as the inputs is transferred (bits 31–0 for integers, bits 63–32 for fractional). The floating-point extension field in Rn is set to all 0s. If MRF or MRB is specified, the entire 80-bit result is placed in MRF or MRB.

Status Flags

- **MN**: Set if the result is negative, otherwise cleared
- **MV**: Set if the upper bits are not all zeros (signed or unsigned result) or ones (signed result); Number of upper bits depends on format; For a signed result, fractional=33, integer=49; For an unsigned result, fractional=32, integer=48
- **MU**: Set if the upper 48 bits of a fractional result are all zeros (signed or unsigned result) or ones (signed result) and the lower 32 bits are not all zeros; Integer results do not underflow
- **MI**: Cleared
Rn = MRF + Rx * Ry mod2
Rn = MRB + Rx * Ry mod2
MRF = MRF + Rx * Ry mod2
MRB = MRB + Rx * Ry mod2

Function

Multiplies the fixed-point fields in registers Rx and Ry, and adds the product to the specified MR register value. If rounding is specified (fractional data only), the result is rounded. The result is placed either in the fixed-point field in register Rn or one of the MR accumulation registers, which must be the same MR register that provided the input. If Rn is specified, only the portion of the result that has the same format as the inputs is transferred (bits 31–0 for integers, bits 63–32 for fractional). The floating-point extension field in Rn is set to all 0s. If MRF or MRB is specified, the entire 80-bit result is placed in MRF or MRB.

Status Flags

- **MN**: Set if the result is negative, otherwise cleared
- **MV**: Set if the upper bits are not all zeros (signed or unsigned result) or ones (signed result); Number of upper bits depends on format; For a signed result, fractional=33, integer=49; For an unsigned result, fractional=32, integer=48
- **MU**: Set if the upper 48 bits of a fractional result are all zeros (signed or unsigned result) or ones (signed result) and the lower 32 bits are not all zeros; Integer results do not underflow
- **MI**: Cleared
Compute Field

\[ R_n = MRF - R_x \times R_y \mod 2 \]
\[ R_n = MRB - R_x \times R_y \mod 2 \]
\[ MRF = MRF - R_x \times R_y \mod 2 \]
\[ MRB = MRB - R_x \times R_y \mod 2 \]

Function

Multiplies the fixed-point fields in registers \( R_x \) and \( R_y \), and subtracts the product from the specified \( MR \) register value. If rounding is specified (fractional data only), the result is rounded. The result is placed either in the fixed-point field in register \( R_n \) or in one of the \( MR \) accumulation registers, which must be the same \( MR \) register that provided the input. If \( R_n \) is specified, only the portion of the result that has the same format as the inputs is transferred (bits 31–0 for integers, bits 63–32 for fractional). The floating-point extension field in \( R_n \) is set to all 0s. If \( MRF \) or \( MRB \) is specified, the entire 80-bit result is placed in \( MRF \) or \( MRB \).

Status Flags

- **MN**: Set if the result is negative, otherwise cleared
- **MV**: Set if the upper bits are not all zeros (signed or unsigned result) or ones (signed result); Number of upper bits depends on format; For a signed result, fractional=33, integer=49; For an unsigned result, fractional=32, integer=48
- **MU**: Set if the upper 48 bits of a fractional result are all zeros (signed or unsigned result) or ones (signed result) and the lower 32 bits are not all zeros; Integer results do not underflow
- **MI**: Cleared
Function

If the value of the specified \texttt{MR} register is greater than the maximum value for the specified data format, the multiplier sets the result to the maximum value. Otherwise, the \texttt{MR} value is unaffected. The result is placed either in the fixed-point field in register \texttt{Rn} or one of the \texttt{MR} accumulation registers, which must be the same \texttt{MR} register that provided the input. If \texttt{Rn} is specified, only the portion of the result that has the same format as the inputs is transferred (bits 31–0 for integers, bits 63–32 for fractional). The floating-point extension field in \texttt{Rn} is set to all 0s. If \texttt{MRF} or \texttt{MRB} is specified, the entire 80-bit result is placed in \texttt{MRF} or \texttt{MRB}.

Status Flags

\begin{itemize}
  \item \texttt{MN} \hspace{1cm} Set if the result is negative, otherwise cleared
  \item \texttt{MV} \hspace{1cm} Cleared
  \item \texttt{MU} \hspace{1cm} Set if the upper 48 bits of a fractional result are all zeros (signed or unsigned result) or ones (signed result) and the lower 32 bits are not all zeros; Integer results do not underflow
  \item \texttt{MI} \hspace{1cm} Cleared
\end{itemize}
Compute Field

\[
\begin{align*}
Rn &= \text{RND MRF mod1} \\
Rn &= \text{RND MRB mod1} \\
\text{MRF} &= \text{RND MRF mod1} \\
\text{MRB} &= \text{RND MRB mod1}
\end{align*}
\]

Function

Rounds the specified MR value to nearest at bit 32 (the \( \text{MR}^{1}-\text{MR}^{0} \) boundary). The result is placed either in the fixed-point field in register \( Rn \) or one of the MR accumulation registers, which must be the same MR register that provided the input. If \( Rn \) is specified, only the portion of the result that has the same format as the inputs is transferred (bits 31–0 for integers, bits 63–32 for fractional). The floating-point extension field in \( Rn \) is set to all 0s. If \( \text{MRF} \) or \( \text{MRB} \) is specified, the entire 80-bit result is placed in \( \text{MRF} \) or \( \text{MRB} \).

Status Flags

- **MN** Set if the result is negative, otherwise cleared
- **MV** Set if the upper bits are not all zeros (signed or unsigned result) or ones (signed result); Number of upper bits depends on format; For a signed result, fractional=33, integer=49; For an unsigned result, fractional=32, integer=48
- **MU** Set if the upper 48 bits of a fractional result are all zeros (signed or unsigned result) or ones (signed result) and the lower 32 bits are not all zeros; Integer results do not underflow
- **MI** Not affected
**MRF = 0**  
**MRB = 0**

Function

Sets the value of the specified MR register to zero. All 80 bits (MR2, MR1, MR0) are cleared.

Status Flags

| MN | Not affected |
| MV | Not affected |
| MU | Not affected |
| MI | Not affected |
Compute Field

\[ \text{MRxF/B} = \text{Rn/Rn} = \text{MRxF/B} \]

Function

A transfer to an \( \text{MR} \) register places the fixed-point field of register \( \text{Rn} \) in the specified \( \text{MR} \) register. The floating-point extension field in \( \text{Rn} \) is ignored. A transfer from an \( \text{MR} \) register places the specified \( \text{MR} \) register in the fixed-point field in register \( \text{Rn} \). The floating-point extension field in \( \text{Rn} \) is set to all 0s.

Syntax Variations

\[
\begin{array}{ccc}
\text{MR0F} &=& \text{Rn} \\
\text{MR1F} &=& \text{Rn} \\
\text{MR2F} &=& \text{Rn} \\
\text{MR0B} &=& \text{Rn} \\
\text{MR1B} &=& \text{Rn} \\
\text{MR2B} &=& \text{Rn}
\end{array}
\]

Table 6-7 indicates how \( \text{Ai} \) specifies the \( \text{MR} \) register, and \( \text{Rk} \) specifies the data register. The \( T \) determines the direction of the transfer (0=to register file, 1=to \( \text{MR} \) register).

Table 6-7. Ai Values and MR Registers

<table>
<thead>
<tr>
<th>Ai</th>
<th>MR Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>MR0F</td>
</tr>
<tr>
<td>0001</td>
<td>MR1F</td>
</tr>
</tbody>
</table>
## Table 6-7. Ai Values and MR Registers (Cont’d)

<table>
<thead>
<tr>
<th>Ai</th>
<th>MR Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>MR2F</td>
</tr>
<tr>
<td>0100</td>
<td>MR0B</td>
</tr>
<tr>
<td>0101</td>
<td>MR1B</td>
</tr>
<tr>
<td>0110</td>
<td>MR2B</td>
</tr>
</tbody>
</table>

### Status Flags

- MN: Cleared
- MV: Cleared
- MU: Cleared
- MI: Cleared
**Function**

Multiplies the floating-point operands in registers Fx and Fy and places the result in the register Fn.

**Status Flags**

- **MN**  Set if the result is negative, otherwise cleared
- **MV**  Set if the unbiased exponent of the result is greater than 127, otherwise cleared
- **MU**  Set if the unbiased exponent of the result is less than –126, otherwise cleared
- **MI**  Set if either input is a NAN or if the inputs are ±Infinity and ±Zero, otherwise cleared

**Shifter Operations**

Shifter operations are described in this section. Table 6-8 lists the syntax and opcodes for the shifter operations. The succeeding pages provide detailed descriptions of each operation. Some of the instructions in Table 6-8 accept the following modifiers.

- (SE) = Sign extension of deposited or extracted field
- (EX) = Extended exponent extract

**Shifter Opcodes**

The shifter operates on the register file’s 32-bit fixed-point fields (bits 38-9). Two-input shifter operations can take their y input from the register file or from immediate data provided in the instruction. Either form uses the same opcode. However, the latter case, called an immediate shift or shifter immediate operation, is allowed only with instruction type
6, which has an immediate data field in its opcode for this purpose. All other instruction types must obtain the input from the register file when the compute operation is a two-input shifter operation.

Table 6-8. Shifter Operations

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Opcode</th>
<th>Reference Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn = LSHIFT Rx BY Ry</td>
<td>&lt;data8&gt;</td>
<td>0000 0000</td>
</tr>
<tr>
<td>Rn = Rn OR LSHIFT Rx BY Ry</td>
<td>&lt;data8&gt;</td>
<td>0010 0000</td>
</tr>
<tr>
<td>Rn = ASHIFT Rx BY Ry</td>
<td>&lt;data8&gt;</td>
<td>0000 0100</td>
</tr>
<tr>
<td>Rn = Rn OR ASHIFT Rx BY Ry</td>
<td>&lt;data8&gt;</td>
<td>0010 0100</td>
</tr>
<tr>
<td>Rn = ROT Rx BY Ry</td>
<td>&lt;data8&gt;</td>
<td>0000 1000</td>
</tr>
<tr>
<td>Rn = BCLR Rx BY Ry</td>
<td>&lt;data8&gt;</td>
<td>1100 0100</td>
</tr>
<tr>
<td>Rn = BSET Rx BY Ry</td>
<td>&lt;data8&gt;</td>
<td>1100 0000</td>
</tr>
<tr>
<td>Rn = BTGL Rx BY Ry</td>
<td>&lt;data8&gt;</td>
<td>1100 1000</td>
</tr>
<tr>
<td>BTST Rx BY Ry</td>
<td>&lt;data8&gt;</td>
<td>1100 1100</td>
</tr>
<tr>
<td>Rn = FDEP Rx BY Ry</td>
<td>&lt;bit6&gt;:&lt;len6&gt;</td>
<td>0100 0100</td>
</tr>
<tr>
<td>Rn = Rn OR FDEP Rx BY Ry</td>
<td>&lt;bit6&gt;:&lt;len6&gt;</td>
<td>0110 0100</td>
</tr>
<tr>
<td>Rn = FDEP Rx BY Ry</td>
<td>&lt;bit6&gt;:&lt;len6&gt; (SE)</td>
<td>0100 1100</td>
</tr>
<tr>
<td>Rn = Rn OR FDEP Rx BY Ry</td>
<td>&lt;bit6&gt;:&lt;len6&gt;(SE)</td>
<td>0110 1100</td>
</tr>
<tr>
<td>Rn = FEXT RX BY Ry</td>
<td>&lt;bit6&gt;:&lt;len6&gt;</td>
<td>0100 0000</td>
</tr>
<tr>
<td>Rn = FEXT Rx BY Ry</td>
<td>&lt;bit6&gt;:&lt;len6&gt; (SE)</td>
<td>0100 1000</td>
</tr>
<tr>
<td>Rn = EXP Rx</td>
<td></td>
<td>1000 0000</td>
</tr>
<tr>
<td>Rn = EXP Rx (EX)</td>
<td></td>
<td>1000 0100</td>
</tr>
<tr>
<td>Rn = LEFTZ Rx</td>
<td></td>
<td>1000 1000</td>
</tr>
<tr>
<td>Rn = LEFTO Rx</td>
<td></td>
<td>1000 1100</td>
</tr>
<tr>
<td>Rn = FPACK Fx</td>
<td></td>
<td>1001 0000</td>
</tr>
<tr>
<td>Fn = FUNPACK Rx</td>
<td></td>
<td>1001 0100</td>
</tr>
</tbody>
</table>
Compute Field

\[ Rn = \text{LSHIFT Rx BY Ry} \]
\[ Rn = \text{LSHIFT Rx BY <data8>} \]

**Function**

Logically shifts the fixed-point operand in register Rx by the 32-bit value in register Ry or by the 8-bit immediate value in the instruction. The shifted result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The shift values are two's-complement numbers. Positive values select a left shift, negative values select a right shift. The 8-bit immediate data can take values between \(-128\) and \(127\) inclusive, allowing for a shift of a 32-bit field from off-scale right to off-scale left.

**Status Flags**

- **SZ** Set if the shifted result is zero, otherwise cleared
- **SV** Set if the input is shifted to the left by more than 0, otherwise cleared
- **SS** Cleared
**Rn = Rn OR LSHIFT Rx BY Ry**

**Rn = Rn OR LSHIFT Rx BY <data8>**

**Function**

Logically shifts the fixed-point operand in register Rx by the 32-bit value in register Ry or by the 8-bit immediate value in the instruction. The shifted result is logically ORed with the fixed-point field of register Rn and then written back to register Rn. The floating-point extension field of Rn is set to all 0s. The shift values are twos-complement numbers. Positive values select a left shift, negative values select a right shift. The 8-bit immediate data can take values between –128 and 127 inclusive, allowing for a shift of a 32-bit field from off-scale right to off-scale left.

**Status Flags**

- **SZ** Set if the shifted result is zero, otherwise cleared
- **SV** Set if the input is shifted left by more than 0, otherwise cleared
- **SS** Cleared
Rn = ASHIFT Rx BY Ry
Rn = ASHIFT Rx BY <data8>

Function

Arithmetically shifts the fixed-point operand in register Rx by the 32-bit value in register Ry or by the 8-bit immediate value in the instruction. The shifted result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The shift values are two's-complement numbers. Positive values select a left shift, negative values select a right shift. The 8-bit immediate data can take values between –128 and 127 inclusive, allowing for a shift of a 32-bit field from off-scale right to off-scale left.

Status Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SZ</td>
<td>Set if the shifted result is zero, otherwise cleared</td>
</tr>
<tr>
<td>SV</td>
<td>Set if the input is shifted left by more than 0, otherwise cleared</td>
</tr>
<tr>
<td>SS</td>
<td>Cleared</td>
</tr>
</tbody>
</table>
Rn = Rn OR ASHIFT Rx BY Ry
Rn = Rn OR ASHIFT Rx BY <data8>

Function

Arithmetically shifts the fixed-point operand in register Rx by the 32-bit value in register Ry or by the 8-bit immediate value in the instruction. The shifted result is logically ORed with the fixed-point field of register Rn and then written back to register Rn. The floating-point extension field of Rn is set to all 0s. The shift values are twos-complement numbers. Positive values select a left shift, negative values select a right shift. The 8-bit immediate data can take values between –128 and 127 inclusive, allowing for a shift of a 32-bit field from off-scale right to off-scale left.

Status Flags

SZ Set if the shifted result is zero, otherwise cleared
SV Set if the input is shifted left by more than 0, otherwise cleared
SS Cleared
Compute Field

\[ Rn = \text{ROT Rx BY Ry} \]
\[ Rn = \text{ROT Rx BY <data8>} \]

**Function**

Rotates the fixed-point operand in register Rx by the 32-bit value in register Ry or by the 8-bit immediate value in the instruction. The rotated result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The shift values are two's-complement numbers. Positive values select a rotate left; negative values select a rotate right. The 8-bit immediate data can take values between –128 and 127 inclusive, allowing for a rotate of a 32-bit field from full right wrap around to full left wrap around.

**Status Flags**

- **SZ**: Set if the rotated result is zero, otherwise cleared
- **SV**: Cleared
- **SS**: Cleared
Rn = BCLR Rx BY Ry
Rn = BCLR Rx BY <data8>

Function

Clears a bit in the fixed-point operand in register Rx. The result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The position of the bit is the 32-bit value in register Ry or the 8-bit immediate value in the instruction. The 8-bit immediate data can take values between 31 and 0 inclusive, allowing for any bit within a 32-bit field to be cleared. If the bit position value is greater than 31 or less than 0, no bits are cleared.

Status Flags

SZ Set if the output operand is 0, otherwise cleared
SV Set if the bit position is greater than 31, otherwise cleared
SS Cleared

This compute operation affects a bit in a register file location. There is also a bit manipulation instruction that affects one or more bits in a system register. This Bclr instruction should not be confused with the Bclr shifter operation. For more information on Bit Clr, see “Type 18: System Register Bit Manipulation” on page 5-2.
Compute Field

\[ Rn = \text{BSET Rx BY Ry} \]
\[ Rn = \text{BSET Rx BY } <\text{data8}> \]

Function

Sets a bit in the fixed-point operand in register Rx. The result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The position of the bit is the 32-bit value in register Ry or the 8-bit immediate value in the instruction. The 8-bit immediate data can take values between 31 and 0 inclusive, allowing for any bit within a 32-bit field to be set. If the bit position value is greater than 31 or less than 0, no bits are set.

Status Flags

\[ \begin{align*}
\text{SZ} & \quad \text{Set if the output operand is 0, otherwise cleared} \\
\text{SV} & \quad \text{Set if the bit position is greater than 31, otherwise cleared} \\
\text{SS} & \quad \text{Cleared}
\end{align*} \]

This compute operation affects a bit in a register file location. There is also a bit manipulation instruction that affects one or more bits in a system register. This Bit Set instruction should not be confused with the Bset shifter operation. For more information on Bit Set, see “Type 18: System Register Bit Manipulation” on page 5-2.
**Rn = BTGL Rx BY Ry**  
**Rn = BTGL Rx BY <data8>**

**Function**

Toggles a bit in the fixed-point operand in register Rx. The result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The position of the bit is the 32-bit value in register Ry or the 8-bit immediate value in the instruction. The 8-bit immediate data can take values between 31 and 0 inclusive, allowing for any bit within a 32-bit field to be toggled. If the bit position value is greater than 31 or less than 0, no bits are toggled.

**Status Flags**

- **SZ**  
  Set if the output operand is 0, otherwise cleared
- **SV**  
  Set if the bit position is greater than 31, otherwise cleared
- **SS**  
  Cleared

This compute operation affects a bit in a register file location. There is also a bit manipulation instruction that affects one or more bits in a system register. This **Bit Tgl** instruction should not be confused with the **Btgl** shifter operation. For more information on **Bit Tgl**, see “Type 18: System Register Bit Manipulation” on page 5-2.
Compute Field

**BTST Rx BY Ry**
**BTST Rx BY <data8>**

**Function**

Tests a bit in the fixed-point operand in register Rx. The $SZ$ flag is set if the bit is a 0 and cleared if the bit is a 1. The position of the bit is the 32-bit value in register Ry or the 8-bit immediate value in the instruction. The 8-bit immediate data can take values between 31 and 0 inclusive, allowing for any bit within a 32-bit field to be tested. If the bit position value is greater than 31 or less than 0, no bits are tested.

**Status Flags**

- **SZ**: Cleared if the tested bit is a 1, is set if the tested bit is a 0 or if the bit position is greater than 31
- **SV**: Set if the bit position is greater than 31, otherwise cleared
- **SS**: Cleared

This compute operation tests a bit in a register file location. There is also a bit manipulation instruction that tests one or more bits in a system register. This Bit Tst instruction should not be confused with the Btst shifter operation.

For more information on Bit Tst, see “Type 18: System Register Bit Manipulation” on page 5-2.
**Computations Reference**

### Rn = FDEP Rx BY Ry
### Rn = FDEP Rx BY <bit6>:<len6>

**Function**

Deposits a field from register Rx to register Rn. The input field is right-aligned within the fixed-point field of Rx. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction. The field is deposited in the fixed-point field of Rn, starting from a bit position determined by the bit6 field in register Ry or by the immediate bit6 field in the instruction. Bits to the left and to the right of the deposited field are set to 0. The floating-point extension field of Rn (bits 7–0 of the 40-bit word) is set to all 0s. Bit6 and len6 can take values between 0 and 63 inclusive, allowing for deposit of fields ranging in length from 0 to 32 bits, and to bit positions ranging from 0 to off-scale left.

![Figure 6-1. Field Alignment](image-url)

- **len6** = number of bits to take from Rx, starting from LSB of 32-bit field
- **bit6** = starting bit position for deposit, referenced from the LSB of the 32-bit field
Compute Field

Example

If \( \text{len6}=14 \) and \( \text{bit6}=13 \), then the 14 bits of \( Rx \) are deposited in \( Rn \) bits 34-21 (of the 40-bit word).

\[
\begin{array}{cccccccc}
39 & 31 & 23 & 15 & 7 & 0 \\
\hline
|--------|--------|--abcdef|ghijklmn|--| Ry \\
/--------/--------/--------/--------/--------/  \\
14 bits
\end{array}
\]

\[
\begin{array}{cccccccc}
39 & 31 & 23 & 15 & 7 & 0 \\
\hline
|000000abc|defghijk|lmn00000|00000000|00000000|--| Rn \\
/--------/--------/--------/--------/--------/------/  \\
|  \\
bit position 13 (from reference point)
\end{array}
\]

Status Flags

\( SZ \)  
Set if the output operand is 0, otherwise cleared

\( SV \)  
Set if any bits are deposited to the left of the 32-bit fixed-point output field (that is, if \( \text{len6} + \text{bit6} > 32 \)), otherwise cleared

\( SS \)  
Cleared
Rn = Rn OR FDEP Rx BY Ry
Rn = Rn OR FDEP Rx BY <bit6>:<len6>

**Function**

Deposits a field from register Rx to register Rn. The field value is logically ORed bitwise with the specified field of register Rn and the new value is written back to register Rn. The input field is right-aligned within the fixed-point field of Rx. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction.

The field is deposited in the fixed-point field of Rn, starting from a bit position determined by the bit6 field in register Ry or by the immediate bit6 field in the instruction. Bit6 and len6 can take values between 0 and 63 inclusive, allowing for deposit of fields ranging in length from 0 to 32 bits, and to bit positions ranging from 0 to off-scale left.

**Example**

```
39 31 23 15 7 0
|--------|--------|--abcdef|ghijklmn|--|
|--------|
Rx
--------/
len6 bits

39 31 23 15 7 0
|abcdefgh|ijklmnop|qrstuvwxy|yzabcdef|ghijklmn|
|--------|
Rn old
--------/

| bit position bit6 (from reference point)

39 31 23 15 7 0
|abcdeopqrstuvwxyzabtuvwxy|yzabcdef|ghijklmn|
|--------|
Rn new
--------/

OR result
```
Compute Field

Status Flags

SZ  Set if the output operand is 0, otherwise cleared

SV  Set if any bits are deposited to the left of the 32-bit fixed-point output field (that is, if len6 + bit6 > 32), otherwise cleared

SS  Cleared
Function

Deposits and sign-extends a field from register Rx to register Rn. The input field is right-aligned within the fixed-point field of Rx. Its length is determined by the `len6` field in register Ry or by the immediate `len6` field in the instruction. The field is deposited in the fixed-point field of Rn, starting from a bit position determined by the `bit6` field in register Ry or by the immediate `bit6` field in the instruction. The MSBs of Rn are sign-extended by the MSB of the deposited field, unless the MSB of the deposited field is off-scale left. Bits to the right of the deposited field are set to 0. The floating-point extension field of Rn (bits 7–0 of the 40-bit word) is set to all 0s. `Bit6` and `len6` can take values between 0 and 63 inclusive, allowing for deposit of fields ranging in length from 0 to 32 bits into bit positions ranging from 0 to off-scale left.

![Field Alignment Diagram]

Figure 6-2. Field Alignment
Compute Field

Example

```
39 31 23 15 7 0
|--------|--------|--abcdef|ghijklmn|--------| Rx
\-----------------/          
  len6 bits

39 31 23 15 7 0
|aaaaaabc|defghijk|lmn00000|00000000|00000000| Rn
\----/\--------------/
  sign                
  extension            bit position bit6
       (from reference point)
```

Status Flags

- **SZ** Set if the output operand is 0, otherwise cleared
- **SV** Set if any bits are deposited to the left of the 32-bit fixed-point output field (that is, if len6 + bit6 > 32), otherwise cleared
- **SS** Cleared
Function

Deposits and sign-extends a field from register Rx to register Rn. The sign-extended field value is logically ORed bitwise with the value of register Rn and the new value is written back to register Rn. The input field is right-aligned within the fixed-point field of Rx. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction. The field is deposited in the fixed-point field of Rn, starting from a bit position determined by the bit6 field in register Ry.

The bit position can also be determined by the immediate bit6 field in the instruction. Bit6 and len6 can take values between 0 and 63 inclusive to allow the deposit of fields ranging in length from 0 to 32 bits into bit positions ranging from 0 to off-scale left.
Compute Field

Example

```
39  31  23  15  7  0
|--------|--------|--abcdef|ghijklmn|--------|
\-------------/
len6 bits

39  31  23  15  7  0
|aaaaaabc|defghijk|lmn00000|00000000|00000000|
\-----/\-------------/
  sign  
  extension bit position bit6
       (from reference point)

39  31  23  15  7  0
|abcdefghijklmnopqrstuvwxyzabcdefghijklmn| Rn old

39  31  23  15  7  0
|vwxyzabc|defghijk|lmntuvwxyzabcdefghijklmnopqrstuvwxyzghijklmn| Rn new
  |
OR result
```

Status Flags

- **SZ**: Set if the output operand is 0, otherwise cleared
- **SV**: Set if any bits are deposited to the left of the 32-bit fixed-point output field (that is, if len6 + bit6 > 32), otherwise cleared
- **SS**: Cleared
Rn = FEXT Rx BY Ry
Rn = FEXT Rx BY <bit6>:<len6>

Function

Extracts a field from register Rx to register Rn. The output field is placed right-aligned in the fixed-point field of Rn. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction. The field is extracted from the fixed-point field of Rx starting from a bit position determined by the bit6 field in register Ry or by the immediate bit6 field in the instruction. Bits to the left of the extracted field are set to 0 in register Rn. The floating-point extension field of Rn (bits 7–0 of the 40-bit word) is set to all 0s. Bit6 and len6 can take values between 0 and 63 inclusive, allowing for extraction of fields ranging in length from 0 to 32 bits, and from bit positions ranging from 0 to off-scale left.

Figure 6-3. Field Alignment
Compute Field

Example

\[
\begin{array}{cccccc}
39 & 31 & 23 & 15 & 7 & 0 \\
\hline
\text{len6 bits} & \text{bit position bit6} & \text{(from reference point)} \\
\end{array}
\]

Status Flags

- **SZ**: Set if the output operand is 0, otherwise cleared
- **SV**: Set if any bits are extracted from the left of the 32-bit fixed-point, input field (that is, if \(\text{len6} + \text{bit6} > 32\)), otherwise cleared
- **SS**: Cleared
**Rn = FEXT Rx BY Ry (SE)**

**Rn = FEXT Rx BY <bit6>:<len6> (SE)**

**Function**

Extracts and sign-extends a field from register Rx to register Rn. The output field is placed right-aligned in the fixed-point field of Rn. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction. The field is extracted from the fixed-point field of Rx starting from a bit position determined by the bit6 field in register Ry or by the immediate bit6 field in the instruction. The MSBs of Rn are sign-extended by the MSB of the extracted field, unless the MSB is extracted from off-scale left.

The floating-point extension field of Rn (bits 7–0 of the 40-bit word) is set to all 0s. Bit6 and len6 can take values between 0 and 63 inclusive, allowing for extraction of fields ranging in length from 0 to 32 bits and from bit positions ranging from 0 to off-scale left.

**Example**

```
39 31 23 15 7 0
|-----abc|defghijk|lmn-----|--------|--------| Rx
\---------------/
len6 bits

bit position bit6
(from reference point)

39 31 23 15 7 0
|aaaaaaa|aaaaaaa|aaabcdef|ghijklmn|00000000| Rn
\-------------------/
sign extension
```
Compute Field

Status Flags

SZ   Set if the output operand is 0, otherwise cleared

SV   Set if any bits are extracted from the left of the 32-bit fixed-point input field (that is, if len6 + bit6 > 32), otherwise cleared

SS   Cleared
**Rn = EXP Rx**

**Function**

Extracts the exponent of the fixed-point operand in Rx. The exponent is placed in the shf8 field in register Rn. The exponent is calculated as the two’s-complement of:

# leading sign bits in Rx – 1

**Status Flags**

- **SZ**: Set if the extracted exponent is 0, otherwise cleared
- **SV**: Cleared
- **SS**: Set if the fixed-point operand in Rx is negative (bit 31 is a 1), otherwise cleared
**Compute Field**

\[ R_n = \text{EXP } R_x (\text{EX}) \]

**Function**

Extracts the exponent of the fixed-point operand in \( R_x \), assuming that the operand is the result of an ALU operation. The exponent is placed in the shf8 field in register \( R_n \). If the \( AV \) status bit is set, a value of +1 is placed in the shf8 field to indicate an extra bit (the ALU overflow bit). If the \( AV \) status bit is not set, the exponent is calculated as the twos-complement of:

\[
\# \text{ leading sign bits in } R_x - 1
\]

**Status Flags**

- **SZ**: Set if the extracted exponent is 0, otherwise cleared
- **SV**: Cleared
- **SS**: Set if the exclusive OR of the \( AV \) status bit and the sign bit (bit 31) of the fixed-point operand in \( R_x \) is equal to 1, otherwise cleared
Rn = LEFTZ Rx

Function

Extracts the number of leading 0s from the fixed-point operand in Rx. The extracted number is placed in the bit6 field in Rn.

Status Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SZ</td>
<td>Set if the MSB of Rx is 1, otherwise cleared</td>
</tr>
<tr>
<td>SV</td>
<td>Set if the result is 32, otherwise cleared</td>
</tr>
<tr>
<td>SS</td>
<td>Cleared</td>
</tr>
</tbody>
</table>
**Compute Field**

*Rn = LEFTO Rx*

**Function**

Extracts the number of leading 1s from the fixed-point operand in Rx. The extracted number is placed in the bit6 field in Rn.

**Status Flags**

- **SZ**  
  Set if the MSB of Rx is 0, otherwise cleared

- **SV**  
  Set if the result is 32, otherwise cleared

- **SS**  
  Cleared
Rn = FPACK Fx

Function

Converts the IEEE 32-bit floating-point value in Fx to a 16-bit floating-point value stored in Rn. The short float data format has an 11-bit mantissa with a four-bit exponent plus sign bit. The 16-bit floating-point numbers reside in the lower 16 bits of the 32-bit floating-point field.

The result of the FPACK operation is:

135 < exp\(^1\) Largest magnitude representation

120 < exp \(\leq\) 135 Exponent is MSB of source exponent concatenated with the three LSBs of source exponent. The packed fraction is the rounded upper 11 bits of the source fraction.

109 < exp \(\leq\) 120 Exponent=0. Packed fraction is the upper bits (source exponent – 110) of the source fraction prefixed by zeros and the “hidden” 1. The packed fraction is rounded.

exp < 110 Packed word is all zeros.

\(^1\) exp = source exponent sign bit remains the same in all cases

The short float type supports gradual underflow. This method sacrifices precision for dynamic range. When packing a number which would have underflowed, the exponent is set to zero and the mantissa (including “hidden” 1) is right-shifted the appropriate amount. The packed result is a denormal which can be unpacked into a normal IEEE floating-point number.

Status Flags

SZ Cleared

SV Set if overflow occurs, cleared otherwise

SS Cleared
Compute Field

**Fn = FUNPACK Rx**

**Function**

Converts the 16-bit floating-point value in Rx to an IEEE 32-bit floating-point value stored in Fx.

**Result**

\[ 0 < \exp^1 \leq 15 \]

Exponent is the three LSBs of the source exponent prefixed by the MSB of the source exponent and four copies of the complement of the MSB. The unpacked fraction is the source fraction with 12 zeros appended.

\[ \exp = 0 \]

Exponent is \((120 - N)\) where \(N\) is the number of leading zeros in the source fraction. The unpacked fraction is the remainder of the source fraction with zeros appended to pad it and the “hidden” 1 stripped away.

1 \ exp = source exponent sign bit remains the same in all cases

The short float type supports gradual underflow. This method sacrifices precision for dynamic range. When packing a number that would have underflowed, the exponent is set to 0 and the mantissa (including “hidden” 1) is right-shifted the appropriate amount. The packed result is a denormal, which can be unpacked into a normal IEEE floating-point number.
Status Flags

SZ    Cleared
SV    Cleared
SS    Cleared

Multifunction Computations

Multifunction computations are operations that occur simultaneously within the DSP’s computational unit. The syntax for these operations consists of combinations of instructions, delimited with commas and ended with a semicolon. The three types of multifunction computations appear below. Each type has a different format for the compute field.

- “Parallel Add and Subtract” on page 6-95
- “Parallel Multiplier and ALU” on page 6-98
- “Parallel Multiplier With Add and Subtract” on page 6-101

Operand Constraints

Each of the four input operands for multifunction computations are constrained to a different set of four register file locations, as shown in Figure 6-4. For example, the x-input to the ALU must be R8, R9, R10, or R11. In all other compute operations, the input operands can be any register file location.
Figure 6-4. Permitted Input Registers for Multifunction Computations
Parallel Add and Subtract

Function (Fixed-Point)

Completes a dual add/subtract of the fixed-point fields in registers Rx and Ry. The sum is placed in the fixed-point field of register Ra and the difference in the fixed-point field of Rs. The floating-point extension fields of Ra and Rs are set to all 0s. In saturation mode (the ALU saturation mode bit in \texttt{MODE1} set) positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

Function (Floating-Point)

Completes a dual add/subtract of the floating-point operands in registers Fx and Fy. The normalized results are placed in registers Fa and Fs: the sum in Fa and the difference in Fs. Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in \texttt{MODE1}. Post-rounded overflow returns ±Infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). Post-rounded denormal returns ±Zero. Denormal inputs are flushed to ±Zero. A NAN input returns an all 1s result.

Syntax

Table 6-9 shows the fixed-point and floating-point syntax for multifunction add and subtract instructions.

Table 6-9. Multifunction, Parallel Add and Subtract

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Opcode (bits 19–16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ra = Rx + Ry, Rs = Rx – Ry</td>
<td>0111</td>
</tr>
<tr>
<td>Fa = Fx + Fy, Fs = Fx – Fy</td>
<td>1111</td>
</tr>
</tbody>
</table>
**Compute Field**

**Compute Field (Fixed-Point)**

| 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 00 | 0111 | Rs | Ra | Rx | Ry |

- **AZ**: Set if an output is 0s, otherwise cleared
- **AU**: Cleared
- **AN**: Set if the most significant output bit is 1 for either of the outputs, otherwise cleared
- **AV**: Set if the XOR of the carries of the two most significant adder stages of either of the outputs is 1, otherwise cleared
- **AC**: Set if the carry from the most significant adder stage for either of the outputs is 1, otherwise cleared
- **AS**: Cleared
- **AI**: Cleared
## Compute Field (Fixed-Point)

<table>
<thead>
<tr>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>1111</td>
<td>Fs</td>
<td>Fa</td>
<td>Fx</td>
<td>Fy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **AZ**: Set if either of the post-rounded results is a denormal (unbiased exponent < –126) or zero, otherwise cleared.
- **AU**: Set if either post-rounded result is a denormal, otherwise cleared.
- **AN**: Set if either of the floating-point results is negative, otherwise cleared.
- **AV**: Set if a post-rounded result overflows (unbiased exponent > +127), otherwise cleared.
- **AC**: Cleared.
- **AS**: Cleared.
- **AI**: Set if an input is a NAN or if both inputs are Infinities, otherwise cleared.
Parallel Multiplier and ALU

Function

The parallel multiplier/ALU operation performs a multiply or multiply/accumulate and one of the following ALU operations: Add, Subtract, Average, Fixed-point to floating-point conversion or floating-point to fixed-point conversion, and/or Floating-point Abs, Min, or Max.

The multiplier and ALU operations are determined by \texttt{OPCODE}. The selections for the 6-bit \texttt{OPCODE} field are listed in Table 6-11 on page 6-99. The multiplier x and y operands are received from data registers RXM (FXM) and RYM (FYM). The multiplier result operand is returned to data register RM (FM). The ALU x and y operands are received from data registers RXA (FXA) and RYA (FYA). The ALU result operand is returned to data register RA (FA).

The result operands can be returned to any registers within the register file. Each of the four input operands is restricted to a particular set of four data registers.

Table 6-10. Valid Data Registers for Input Operands

<table>
<thead>
<tr>
<th>Input</th>
<th>Valid Sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier X</td>
<td>R3-R0 (F3-F0)</td>
</tr>
<tr>
<td>Multiplier Y</td>
<td>R7-R4 (F7-F4)</td>
</tr>
<tr>
<td>ALU X</td>
<td>R11-R8 (F11-F8)</td>
</tr>
<tr>
<td>ALU Y</td>
<td>R15-R12 (F15-F12)</td>
</tr>
</tbody>
</table>
Syntax

Table 6-11 provides the syntax and opcode for each of the parallel multiplier and ALU instructions for both fixed-point and floating-point versions.

Table 6-11. Multifunction, Multiplier and ALU

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Opcode (bits 22–16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm = R3-0 * R7-4 (SSFR), Ra = R11-8 + R15-12</td>
<td>1000100</td>
</tr>
<tr>
<td>Rm = R3-0 * R7-4 (SSFR), Ra = R11-8 – R15-12</td>
<td>1000101</td>
</tr>
<tr>
<td>Rm = R3-0 * R7-4 (SSFR), Ra = (R11-8 + R15-12)/2</td>
<td>1000110</td>
</tr>
<tr>
<td>MRF = MRF + R3-0 * R7-4 (SSF), Ra = R11-8 + R15-12</td>
<td>1001000</td>
</tr>
<tr>
<td>MRF = MRF + R3-0 * R7-4 (SSF), Ra = R11-8 – R15-12</td>
<td>1001001</td>
</tr>
<tr>
<td>MRF = MRF + R3-0 * R7-4 (SSF), Ra = (R11-8 + R15-12)/2</td>
<td>1001010</td>
</tr>
<tr>
<td>Rm = MRF + R3-0 * R7-4 (SSFR), Ra = R11-8 + R15-12</td>
<td>1001100</td>
</tr>
<tr>
<td>Rm = MRF + R3-0 * R7-4 (SSFR), Ra = R11-8 – R15-12</td>
<td>1001101</td>
</tr>
<tr>
<td>Rm = MRF + R3-0 * R7-4 (SSFR), Ra = (R11-8 + R15-12)/2</td>
<td>1001110</td>
</tr>
<tr>
<td>MRF = MRF – R3-0 * R7-4 (SSF), Ra = R11-8 + R15-12</td>
<td>1010000</td>
</tr>
<tr>
<td>MRF = MRF – R3-0 * R7-4 (SSF), Ra = R11-8 – R15-12</td>
<td>1010001</td>
</tr>
<tr>
<td>MRF = MRF – R3-0 * R7-4 (SSF), Ra = (R11-8 + R15-12)/2</td>
<td>1010010</td>
</tr>
<tr>
<td>Rm = MRF – R3-0 * R7-4 (SSFR), Ra = R11-8 + R15-12</td>
<td>1010100</td>
</tr>
<tr>
<td>Rm = MRF – R3-0 * R7-4 (SSFR), Ra = R11-8 – R15-12</td>
<td>1010101</td>
</tr>
<tr>
<td>Rm = MRF – R3-0 * R7-4 (SSFR), Ra = (R11-8 + R15-12)/2</td>
<td>1010110</td>
</tr>
<tr>
<td>Fm = F3-0 * F7-4, Fa = F11-8 + F15-12</td>
<td>1011000</td>
</tr>
<tr>
<td>Fm = F3-0 * F7-4, Fa = F11-8 – F15-12</td>
<td>1011001</td>
</tr>
<tr>
<td>Fm = F3-0 * F7-4, Fa = FLOAT R11-8 by R15-12</td>
<td>1011010</td>
</tr>
<tr>
<td>Fm = F3-0 * F7-4, Fa = FIX F11-8 by R15-122</td>
<td>1011011</td>
</tr>
<tr>
<td>Fm = F3-0 * F7-4, Fa = ABS F11-8</td>
<td>1011101</td>
</tr>
</tbody>
</table>
### Compute Field

#### Table 6-11. Multifunction, Multiplier and ALU (Cont’d)

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Opcode (bits 22–16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fm = F3-0 * F7-4, Fa = MAX (F11-8, F15-12)</td>
<td>1011110</td>
</tr>
<tr>
<td>Fm = F3-0 * F7-4, Fa = MIN (F11-8, F15-12)</td>
<td>1011111</td>
</tr>
</tbody>
</table>

**Compute Field (Fixed-Point)**

<table>
<thead>
<tr>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Opcode (Table 6-11)</td>
<td>Rs</td>
<td>Ra</td>
<td>Rxm</td>
<td>Rym</td>
<td>Rxa</td>
<td>Rya</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Compute Field (Floating-Point)**

<table>
<thead>
<tr>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Opcode (Table 6-11)</td>
<td>Fs</td>
<td>Fa</td>
<td>Fxm</td>
<td>Fym</td>
<td>Fxa</td>
<td>Fya</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Parallel Multiplier With Add and Subtract

Function

The parallel multiplier and dual add/subtract operation performs a multiply or multiply/accumulate and computes the sum and the difference of the ALU inputs.

The multiplier x and y operands are received from data registers RXM (FXM) and RYM (FYM). The multiplier result operand is returned to data register RM (FM). The ALU x and y operands are received from data registers RXA (FXA) and RYA (FYA). The ALU result operands are returned to data register RA (FA) and RS (FS).

The result operands can be returned to any registers within the register file. Each of the four input operands is restricted to a different set of four data registers.

Table 6-12. Valid Sources of the Input Operands

<table>
<thead>
<tr>
<th>Input</th>
<th>Valid Sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier X</td>
<td>R3-R0 (F3-F0)</td>
</tr>
<tr>
<td>Multiplier Y</td>
<td>R7-R4 (f7-f4)</td>
</tr>
<tr>
<td>ALU X</td>
<td>R11-R8 (F11-F8)</td>
</tr>
<tr>
<td>ALU Y</td>
<td>R15-R12 (F15-F12)</td>
</tr>
</tbody>
</table>
Syntax

Table 6-13 provides the syntax and opcode for each of the parallel multiplier and add/subtract instructions for both fixed-point and floating-point versions.

Table 6-13. Multifunction, Multiplier and Dual Add and Subtract

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Opcode (bits 22–20)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm=R3-0 * R7-4 (SSFR), Ra=R11-8 + R15-12, Rs=R11-8 – R15-12</td>
<td>110</td>
</tr>
<tr>
<td>Fm=F3-0 * F7-4, Fa=F11-8 + F15-12, Fs=F11-8 – F15-12</td>
<td>111</td>
</tr>
</tbody>
</table>

Compute Field (Fixed-Point)

| 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 10 | Rs | Rm | Ra | Rxm | Rym | Rxa | Rya |

Compute Field (Floating-Point)

| 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 11 | Fs | Fm | Fa | Fxm | Fym | Fxa | Fya |
INDEX

Numerics
16-bit floating-point data, 6-91, 6-92
23-bit compute field, 6-1
4-bit accurate seed, 6-46
8-bit accurate seed, 6-44

A
ABS (absolute value) computation, 6-19
ABS Fx computation, 6-35
ABS (Fx – Fy) computation, 6-31
ABS (Fx + Fy) computation, 6-30
absolute address, 1-16, 3-25, 3-28
access between DM or PM & a universal
register, 2-9, 4-2, 4-5
access between DM or PM & the register
file, 2-14
addition
   with borrow computation, 6-15
   with carry computation, 6-9, 6-14
addition computation, 6-7, 6-28
addition/division ((Rx + Ry)/2)
   computation, 6-11
Ai values and MR registers, 6-62
ALU
   operations, 6-3
   saturation mode bit, 6-7, 6-8, 6-9, 6-14,
   6-15, 6-16, 6-17, 6-18, 6-19, 6-41
ALU Carry (AC) bit, 1-20
ALU Overflow (AV) bit, 1-20
ALU saturation mode bit, 6-10
AND (logical) computation, 6-21
AN flag, 6-12, 6-13, 6-33
arithmetically shifting the fixed-point
operand, 6-69
ASHIFT computation, 6-68
assembler, 3-25
ASTAT registers, 1-13
AV status bit, 6-88
AZ flag, 6-12, 6-13, 6-33

B
BCLR (bit clear) computation, 6-71
BCNT register, 1-14
BDCST1 bit, 3-17
bit
   clearing, 6-71
   setting, 6-72
   testing, 6-74
   toggling, 6-73
Bit Clr instruction, 6-71
bit manipulation, on a system register, 5-2
BITREV (bit-reversed) instruction, 5-6
Bit Set instruction, 6-72
bit test flag (BTF), 5-2
Bit Tgl instruction, 6-73
Bit Tst instruction, 6-74
BMAX register, 1-14
BSET (bit set) computation, 6-72
BTGL computation, 6-73
BTST (bit test) computation, 6-74
bus exchange, 1-12
bus master (Bm) condition, 1-21
Index

C

cache flush, 5-10
CBUFEN (circular buffering enable) bit, 2-28, 5-6
circular buffering enable (CBUFEN) bit, 2-28, 5-6
Cjump/Rframe (Type 24) instruction, 1-10, 5-13
clear interrupt (CI) modifier, 3-3, 3-8
CLIP computation, 6-27
CLIP Fx by Fy computation, 6-51
CNST1|2 registers, 1-15
compare fixed-point fields, 6-12
COMP (compare) computation, 6-12
COMP(Fx, Fy) computation, 6-33
compiler-generated instruction, 5-13
complementary registers (Ureg–Cureg), 1-30
complement (Fn = –Fx) computation, 6-34
complement (Rn = –Rx) computation, 6-18
COMPU computation, 6-13

computation (continued)
COMP(Fx, Fy), 6-33
complement (Fn = –Fx), 6-34
complement (Rn = –Rx), 6-18
COMPU, 6-13
COPYSIGN, 6-48
decrement (Rn = Rx – 1), 6-17
division (Fx + Fy)/2, 6-32
EXP, 6-87
EXP Rx (EX), 6-88
FDEP, 6-75, 6-77
FDEP Rx by Ry (SE), 6-79
FEXT, 6-83
FEXT Rx by Ry (SE), 6-85
FIX Fx, 6-41
FLOAT, 6-43
FPACK, 6-91
FUNPACK, 6-92
increment, 6-16
LEFTO, 6-90
LEFTZ, 6-89
LOGBFx, 6-40
LSHIFT, 6-66
MANT Fx, 6-39
MAX, 6-26
MAX(Fx, Fy), 6-50
MIN, 6-25
MIN(Fx, Fy), 6-49
multiplication, 6-56
multiplication/addition (Rn = MRF + Rx
* Ry mod2), 6-57
multiplication (Fn = Fx * Fy), 6-64
multiplication/subtraction (Rn = MRF –
Rx * Ry mod2), 6-58
NOT, 6-24
OR (logical), 6-22
parallel multiplier & ALU, 6-98
parallel multiplier with add & subtract, 6-101
PASS, 6-20
Index

computation (continued)
PASS Fx, 6-36
RECPIS Fx, 6-44
RND, 6-60
RND Fx, 6-37
Rn OR ASHIFT Rx by Ry, 6-69
Rn OR FDEP Rx by Ry (SE), 6-81
Rn OR LSHIFT Rx by Ry, 6-67
ROT (rotate), 6-70
RSQRTS, 6-46
SAT, 6-59
SCALB Fx by Ry, 6-38
subtraction, 6-8
subtraction (Fn = Fx – Fy), 6-29
subtraction with borrow, 6-10
transfer (MR = RN/Rn = MR), 6-62
TRUNC Fx, 6-41
XOR (logical), 6-23
zero (MRF = 0), 6-61
compute and move/modify instruction summary, 1-3
calculate/dreg«⋯»DMlPM, immediate modify (Type 4), 1-4, 2-14
calculate/dreg«⋯»DM/dreg«⋯»PM (Type 1), 1-3, 2-3
calculate field, single-function operation, 6-2
calculate/modify (Type 7) instruction, 1-5, 2-28
calculate & move instructions, 2-1
calculate operations, 6-1
calculate (Type 2) instruction, 1-3, 2-7
calculate/ureg«⋯»DMlPM, register modify (Type 3), 1-4-2-9
calculate/ureg«⋯»ureg (Type 5), 1-5, 2-19
conditional
    call, 3-3, 3-8
codes, 1-20

conditional (continued)
    conditions, 1-20
evaluation, 1-22
execution codes summary, 1-20
instructions, 1-20
instructions, 2-7
jump, 3-3, 3-8, 3-14
program loop (DO), 3-26
condition & termination codes (IF & DO UNTIL), 1-20
COPYSIGN calculation, 6-48
CROSSCORE software, 1-2
CURLCNTR register, 1-11
current loop counter (CURLCNTR), 3-24

D
DADDR register, 1-11
DAGs, 1-12
    registers, 1-17, 5-6
data register file location, 6-93
data register file (Ureg & Dreg), 1-11
data registers, for input operands, 6-98
declare address (read-only, 1-11
decrement (Rn = Rx – 1) calculation, 6-17
delayed branch (DB) modifier, 3-3, 3-8, 3-20
development tools, 1-2
direct addressing, 1-16, 4-2
direct jump|call (Type 8) instruction, 1-6, 3-3
division (Fx + Fy)/2 calculation, 6-32
DMA (DMAC) control registers, 1-14
DMA registers, 1-14
DMASTAT register, 1-14
DO LOOP instruction address, 3-25, 3-28
do until counter expired (Type 12) instruction, 1-7, 3-24
do until (Type 13) instruction, 1-7, 3-26
Index

E
ELAST register, 1-14
end-of-loop address, 3-25, 3-28
EPBn registers, 1-14
equals (EQ) condition, 1-20
EXP computation, 6-87
exponent, extracting, 6-87
EXP Rx (EX) computation, 6-88
external port FIFO buffers, 1-14

F
FADDR register, 1-11
false always (FOREVER) do/until condition, 1-21
FDEP computation, 6-75, 6-77
FDEP Rx by Ry (SE) computation, 6-79
fetch address, incrementing, 5-11
fetch address (read-only), 1-11
FEXT computation, 6-83
FEXT Rx by Ry (SE) computation, 6-85
field alignment, 6-75, 6-79, 6-83
extracting from a register, 6-83
fixed-point ALU operations, 6-4
fixed-point multiplier operations, 6-53
fixed-point operand
converting to a floating-point result, 6-43
passing through ALU, 6-20
FIX Fx computation, 6-41
fix operation, 6-41
flag input (FLAGx_IN) conditions, 1-21
FLAGS register, 1-13
FLOAT computation, 6-43
floating-point
ALU operations, 6-5
multiplier operations, 6-53
operand comparison, 6-33
floating-point operands
adding, 6-28
comparing, 6-33
extracting mantissa, 6-39
multiplying, 6-64
rounding, 6-37
scaling the exponent of, 6-38
subtracting, 6-29
Fn registers, 1-11
FPACK computation, 6-91
FUNPACK computation, 6-92

G
greater or equals (GE) condition, 1-20
greater than (GT) condition, 1-20
Group I (compute & move) instructions, 2-1
Group III (immediate move) instructions, 4-1
Group II (program flow control) instructions, 3-1
Group IV (miscellaneous) instructions, 5-1

I
idle (Type 22) instruction, 1-10, 5-12
IMASKP register, 1-13, 3-19
IMASK register, 1-13
immediate data ···DM|PM (Type 16), 1-8
immediate data ···ureg (Type 17), 1-8
immediate data write to a universal register, 4-12
immediate move instructions, 4-1
immediate move instruction summary, 1-7
immediate shift, 2-23, 6-64
Index

immediate shift/dreg«···»DM|PM
(Type 6), 1-5, 2-23
increment (Rn = Rx + 1) computation,
6-16
indirect addressing, 1-17, 4-5, 4-9
using DAG registers, 1-17
indirect jump|call / compute (Type 9)
instruction, 1-6, 3-8
indirect jump or compute/dreg«···»DM
(Type 10), 3-14
indirect jump or compute/dreg«···»DM
(Type 10) instruction, 1-6
input operands, valid sources of, 6-101
instruction
  cache clearing, 5-9
  opcodes, 1-33
  opcode summary, 1-23
set notation, 1-17
set summary, 1-1
(Type 1) compute/dreg«···»DM/dreg«···»PM,
  1-3, 2-3
(Type 2) compute, 1-3, 2-7
(Type 3) compute/ureg«···»DM|PM,
  register modify, 1-3, 2-9
(Type 4) compute/dreg«···»DM|PM,
  immediate modify, 1-4, 2-14
(Type 5) compute/ureg«···»ureg, 1-5,
  2-19
(Type 6) immediate shift/dreg«···»DM|PM,
  1-5, 2-23
(Type 7) compute/modify, 1-5, 2-28
(Type 8) direct jump|call, 1-6, 3-3
(Type 9) indirect jump|call / compute,
  1-6, 3-8
(Type 10) indirect jump or compute/dreg«···»DM,
  1-6, 3-14

instruction (continued)
(Type 11) return from subroutine|interrupt/compute,
  1-7, 3-19
(Type 12) do until counter expired, 1-7,
  3-24
(Type 13) do until, 1-7, 3-26
(Type 14) Ureg«···»DM|PM (direct
  addressing), 1-8, 4-2
(Type 15) Ureg«···»DM|PM (indirect
  addressing), 1-8, 4-5
(Type 16) immediate data «···»DM|PM, 1-8
(Type 16) immediate data «···»DM|PM,
  4-9
(Type 17) immediate data «···»ureg, 1-8
(Type 17) immediate data «···»ureg, 4-12
(Type 18) system register bit
  manipulation, 1-9, 5-2
(Type 19) I register modify/bit-reverse,
  1-9, 5-6
(Type 20) push|pop stacks/flush cache,
  1-9, 5-9
(Type 21) Nop, 1-10, 5-11
(Type 22) idle, 1-10, 5-12
(Type 24) Cjump/Rframe, 1-10, 5-13
instruction opcode acronym summary,
1-23
instructions
  Group I (compute & move), 2-1
  Group III (immediate move), 4-1
  Group II (program flow control), 3-1
  group IV (miscellaneous), 5-1
interrupt latch register (IRPTL), 3-4, 3-9
interrupt mask, 1-13
interrupt mask pointer (IMASKP), 3-4, 3-9
I/O and multiplier registers, 1-10
IOP registers
  DMA, 1-14
link ports, 1-15
IOP registers (continued)
  SPORTs, 1-16
  system control, 1-14
I register modify/bit-reverse (Type 19) instruction, 1-9, 5-6
I register update, 1-17
IRPTL register, 1-13, 3-19
ISR programming issues, 3-9, 3-19

L
LADDR register, 1-11
LAR (link assignment) register, 1-15
LBUFn registers, 1-15
LCNTR register, 1-11, 3-24, 3-25
LCOM register, 1-15
LCTLn registers, 1-15
leading 0s, extracting, 6-89
leading 1s, extracting, 6-90
LEFTO computation, 6-90
LEFTZ computation, 6-89
len6 field, 6-79, 6-81
less or equals (LE) condition, 1-20
less than (LT) condition, 1-20
link port buffers, 1-15
link ports, 1-15
LIRPTL register, 1-13
LOGB Fx computation, 6-40
logically ORed shifted result, 6-67, 6-69
logically shifting the fixed-point operand, 6-67
loop
  address, 5-9
  stack, 3-3, 3-8, 3-26, 5-10
  termination, 1-20, 3-26
loop abort (LA) modifier, 3-3, 3-8
loop counter, 1-11
  expired, 3-24
  setup, 3-24
  stack, 3-24
  stacks, 5-9
  loop counter expired (LCE) condition, 1-21
  loop reentry (LR), 3-20
  low power state, 5-12
  LPATHn registers, 1-15
  LPCNT register, 1-15
  LR modifier, 3-4, 3-20
  LSHIFT computation, 6-66
  LSRQ (link service request) register, 1-15

M
magnitude comparison of fixed-point contents, 6-13
MANT Fx computation, 6-39
mantissa, extracting, 6-39
map 1 & 2 registers, 1-29, 1-31, 1-32
MAX computation, 6-26
MAX(Fx, Fy) computation, 6-50
memory addressing summary, 1-16
MIN computation, 6-25
MIN(Fx, Fy) computation, 6-49
miscellaneous instructions, 1-9
miscellaneous operation instructions, 5-1
miscellaneous operations summary, 1-9
MMASK register, 1-13
Mod1 options and opcodes, 6-54
Mod2 options and opcodes, 6-55
MODE1|2 registers, 1-13
MODE1 system register, 3-17
mode mask, 1-13
modifiers, Mod1 and Mod2, 6-54
modify/update an I register with a DAG, 2-28, 5-6
MR registers, 1-16, 2-1, 3-1
MR register transfers, 6-1
MSGRn registers, 1-14
multifunction
  instruction registers, 6-93
  instructions, 6-1, 6-93
multifunction, multiplier & ALU opcodes, 6-99
multifunction, multiplier & dual add & subtract opcodes, 6-102
multifunction, parallel add & subtract opcodes, 6-95
multiplication/addition (Rn = MRF + Rx * Ry mod2) computation, 6-57
multiplication computation, 6-56
multiplication (Fn = Fx * Fy) computation, 6-64
multiplication/subtraction (Rn = MRF – Rx * Ry mod2) computation, 6-58
multiplier operations, 6-52
registers, 1-16
multiplier overflow (MV) bit, 1-20
multiplier signed (MS) bit, 1-20
multiply-accumulate, 6-98

P
parallel
accesses to data and program memory, 2-3
add & subtract, 6-95
multiplier & ALU computation, 6-98
multiplier with add & subtract computation, 6-101
PASS computation, 6-20
PASS Fx computation, 6-36
PC register, 1-11
PC-relative address, 1-16, 3-3, 3-8
PC-relative jump|call, 3-3, 3-8
PC stack, 3-3, 3-8, 3-26, 5-9, 5-10
pointer, 1-11
top of, 1-11
PCSTKP register, 1-11
PCSTK register, 1-11
post-modify with immediate value, 1-17
post-modify with M register, 1-17
pre-modify, no update, 1-17
pre-modify with M register, 1-17
program control instructions, 3-1
program counter (read-only), 1-11
program flow control
instruction summary, 1-5
program sequencer, 1-11
push|pop stacks/flush cache (Type 20), 5-9
push|pop stacks/flush cache (Type 20) instruction, 1-9
PX register, 1-12

R
RECIPS Fx computation, 6-44
reference notation summary, 1-17
registers, for multifunction computations, 6-93
register-to-memory data moves, 4-1
register types summary, 1-10

N
Nop (Type 21) instruction, 1-10, 5-11, 5-12
NOT computation, 6-24
Not Equal (NE) instruction, 1-20
null operation, 5-11

O
opcode acronyms, 1-23
opcodes, 1-33
operand constraints, 6-93
OR (logical) computation, 6-22
overflow (See ALU, multiplier, or shifter)
RELADDR bit, 3-25, 3-28
relative address, 3-25, 3-28
return from an interrupt service routine (RTI), 3-19
return from a subroutine (RTS) instruction, 3-4, 3-9, 3-19, 3-20
return from subroutine|interrupt/compute (Type 11), 3-19
return from subroutine|interrupt/compute (Type 11) instruction, 1-7
RND computation, 6-60
RND Fx computation, 6-37
Rn OR ASHIFT Rx by Ry computation, 6-69
Rn OR FDEP Rx by Ry (SE) computation, 6-81
Rn OR LSHIFT Rx by Ry computation, 6-67
Rn registers, 1-11
ROT (rotate) computation, 6-70
rounding mode bit, 6-11
RSQRTS computation, 6-46
RTI instruction, 3-19
RTS instruction, 3-9, 3-19, 3-20

S
SAT computation, 6-59
saturation mode, 6-7, 6-8, 6-9, 6-10, 6-14, 6-15, 6-16, 6-17, 6-18, 6-19, 6-40, 6-41
SCALB Fx by Ry computation, 6-38
SFn registers, 1-11
shifter
    immediate operation, 6-64
    opcodes, 6-65
shifter operations, 6-64
two-input, 6-64
shifter overflow (SV) bit, 1-20
shifter zero (SZ) bit, 1-20
short float data format, 6-91, 6-92
sign-extended field, 6-79, 6-81
SIMD mode conditional execution, 1-23
single-function operations, 6-1, 6-2
SISD mode conditional execution, 1-22
SISD/SIMD conditional testing summary, 1-22
Sn registers, 1-11
SPORTs registers, 1-16
status stack, 3-19, 5-9
STKY registers, 1-13
subtraction, with borrow computation, 6-10
subtraction computation, 6-8
subtraction (Fx = Fx – Fy) computation, 6-29
subtract with borrow, 6-10
swap between universal registers, 2-19
SYSCON register, 1-14
SYSTAT register, 1-14
system control registers, 1-14
system operations, 5-1
system register bit manipulation (Type 18), 5-2
system register bit manipulation (Type 18) instructions, 1-9
system registers (Sreg & Ureg), 1-13

T
TCOUNT register, 1-12
technical support, -xiv
termination codes
    (See conditional codes and loop termination)
termination condition, 3-24, 3-26, 3-28
test flag true (TF) condition, 1-20
timer, 1-12
timer counter, 1-12
tools, development, 1-2
TPERIOD register, 1-12
transfer between data or program memory & universal register, 4-2
transfer between universal registers, 2-19
transfer (MR = RN/Rn = MR)
computation, 6-62
true always (TRUE) if condition, 1-21
TRUNC bit, 6-41
TRUNC Fx computation, 6-41
trunc operation, 6-41

U
underflow, gradual, 6-91, 6-92
underflow (See multiplier)
universal and system registers, 1-29
universal register codes (SIMD), 1-32
universal register codes (SISD/SIMD), 1-31
universal registers, 1-28, 1-31, 1-32, 4-2, 4-5
universal registers (Ureg), 1-10
update an I register with an M register, 2-28
Ureg«⋯»DM|PM (direct addressing)
(Type 14), 1-8
Ureg«⋯»DM|PM (direct addressing)
(Type 15), 1-8
Ureg«⋯»DM|PM (indirect addressing)
(Type 15) instruction, 4-5
user status register, 1-13
USTATn registers, 1-13

V
VIRPT register, 1-14

W
WAIT register, 1-14
write 32-bit immediate data to a Ureg register, 4-12
write 32-bit immediate data to DM or PM, 4-9

X
XOR (logical) computation, 6-23

Z
zero (MRF = 0) computation, 6-61
Index