Copyright Information

© 2009 Analog Devices, Inc., ALL RIGHTS RESERVED. This document may not be reproduced in any form without prior, express written consent from Analog Devices, Inc.

Printed in the USA.

Disclaimer

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

Trademark and Service Mark Notice

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

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

PREFACE

Purpose of This Manual ............................................................. xxxiii
Intended Audience ................................................................. xxxiii
Manual Contents ..................................................................... xxxiv
What’s New in This Manual ..................................................... xxxvi
Technical or Customer Support ............................................... xxxvi
  Registration for MyAnalog.com ........................................... xxxvii
  EngineerZone .................................................................... xxxvii
  Social Networking Web Sites ............................................. xxxviii
Supported Processors .............................................................. xxxviii
Product Information ............................................................... xxxviii
  Analog Devices Web Site ................................................... xxxix
  VisualDSP++ Online Documentation .................................. xxxix
  Technical Library CD ........................................................ xl
Notation Conventions ............................................................ xli

INTRODUCTION

SHARC Design Advantages ..................................................... 1-1
Architectural Overview ........................................................ 1-3
Contents

Processor Core ................................................................. 1-3
Dual Processing Elements .................................................. 1-5
Program Sequence Control .............................................. 1-6
JTAG Port ........................................................................ 1-9
Core Buses ....................................................................... 1-9
I/O Buses ........................................................................ 1-9
Development Tools .......................................................... 1-10
Differences From Previous SHARC Processors .................. 1-11

REGISTER FILES

Features .............................................................................. 2-1
Functional Description ...................................................... 2-1
   Core Register Classification ........................................... 2-2
   Register Types Overview .............................................. 2-2
   Data Registers ............................................................. 2-5
   Complementary Data Registers ...................................... 2-5
   Data and Complementary Data Register Access Priorities ... 2-6
   Data and Complementary Data Register Transfers .......... 2-7
   Data and Complementary Data Register Swaps .............. 2-7
   System Register Bit Manipulation .................................. 2-8
   Combined Data Bus Exchange Register ......................... 2-9
      PX to DREG Transfers .............................................. 2-10
      Immediate 40-bit Data Register Load ......................... 2-11
      PX to Memory Transfers ......................................... 2-11
      PX to Memory LW Transfers ..................................... 2-12
Uncomplimentary UREG to Memory LW Transfers ........... 2-13
Operating Modes ................................................................. 2-13
Alternate (Secondary) Data Registers ............................... 2-13
Alternate (Secondary) Data Registers SIMD Mode .......... 2-14
UREG/SREG SIMD Mode Transfers ................................... 2-16
Interrupt Mode Mask ......................................................... 2-17

PROCESSING ELEMENTS
Features .................................................................................. 3-1
Functional Description .......................................................... 3-2
Single Cycle Processing ....................................................... 3-2
Processing Data from Memory Loads ................................. 3-2
Data Format for Computation Units ....................................... 3-3
Arithmetic Logic Unit (ALU) .................................................. 3-3
  Functional Description ....................................................... 3-4
  ALU Instruction Types ...................................................... 3-5
    Compare Accumulation Instruction ................................. 3-5
    Fixed-to-Float Conversion Instructions ......................... 3-6
    Fixed-to-Float Conversion Instructions with Scaling ...... 3-6
    Reciprocal/Square Root Instructions ............................ 3-6
    Divide Instruction ......................................................... 3-6
  ALU Instruction Summary ................................................ 3-6
Multiply Accumulator ........................................................... 3-9
  Functional Description ....................................................... 3-10
  Asymmetric Multiplier Inputs .......................................... 3-10
Contents

Multiplier Result Register ........................................................ 3-11
Multiply Register Instruction Types ........................................ 3-12
  Clear MRx Instruction ....................................................... 3-12
  Round MRx Instruction .................................................... 3-12
  Saturate MRx Instruction .................................................. 3-13
Multiplier Instruction Summary ............................................... 3-13
Barrel Shifter ............................................................................ 3-16
Functional Description ........................................................... 3-17
Shifter Instruction Types .......................................................... 3-17
  Shift Compute Category .................................................... 3-17
  Shift Immediate Category ................................................... 3-17
  Bit Manipulation Instructions ............................................ 3-18
  Bit Field Manipulation Instructions ................................... 3-19
  Bit Stream Manipulation Instructions (ADSP-214xx) .......... 3-22
Floating-Point Packing Instructions ........................................... 3-25
Shifter Instruction Summary .................................................... 3-26
Multifunction Computation ..................................................... 3-27
  Software Pipelining for Multifunction Instructions ............ 3-28
  Multifunction and Data Move .............................................. 3-28
  Multifunction Input Operand Constraints .......................... 3-29
  Multifunction Input Modifier Constraints ............................ 3-30
  Multifunction Instruction Types ........................................ 3-30
Operating Modes .................................................................... 3-32
  ALU Saturation ................................................................. 3-32
Contents

Short Word Sign Extension ............................................... 3-33
Floating-Point Boundary Rounding Mode ......................... 3-33
Rounding Mode .............................................................. 3-34
Multiplier Result Register Swap ....................................... 3-35
SIMD Mode .................................................................. 3-36
Conditional Computations in SIMD Mode ....................... 3-37
Interrupt Mode Mask ...................................................... 3-38
Arithmetic Interrupts ...................................................... 3-38
Computational Status ...................................................... 3-38
Computation Status Update Priority ................................. 3-39
SIMD Computation and Status Flags ............................... 3-40
SIMD Computation Interrupts ......................................... 3-40
ALU Interrupts ............................................................. 3-40
Interrupt Acknowledge .................................................. 3-41
Multiplier Interrupts ...................................................... 3-42
Shifter Status Flags ....................................................... 3-43

PROGRAM SEQUENCER

Features ........................................................................... 4-1
Functional Description ................................................... 4-4
Instruction Pipeline ....................................................... 4-5
VISA Instruction Alignment Buffer (IAB) ......................... 4-6
Linear Program Flow ..................................................... 4-8
Direct Addressing .......................................................... 4-9
Variation In Program Flow .............................................. 4-10
## Contents

- Functional Description .......................................................... 4-10
- Hardware Stacks ........................................................................ 4-10
- PC Stack Access .......................................................................... 4-12
- PC Stack Status ........................................................................... 4-12
- PC Stack Manipulation ............................................................... 4-13
- PC Stack Access Priorities ........................................................ 4-13
- Status Stack Access .................................................................... 4-13
- Status Stack Status ..................................................................... 4-15
- Instruction Driven Branches ..................................................... 4-15
- Direct Versus Indirect Branches ............................................... 4-17
- Restrictions for VISA Operation ................................................ 4-18
- Delayed Branches (DB) ............................................................... 4-19
- Branch Listings ........................................................................... 4-19
- Operating Mode ........................................................................... 4-25
- Interrupt Branch Mode ............................................................... 4-26
- Interrupt Processing Stages ....................................................... 4-27
- Interrupt Categories .................................................................... 4-28
- Interrupt Processing .................................................................... 4-32
- Latching Interrupts ...................................................................... 4-34
- Interrupt Acknowledge ............................................................... 4-35
- Interrupt Self-Nesting ................................................................. 4-35
- Release From IDLE ....................................................................... 4-37
- Causes of Delayed Interrupt Processing ..................................... 4-38
- Interrupt Mask Mode .................................................................... 4-39
Interrupt Nesting Mode ................................................................. 4-41
Loop Sequencer .............................................................................. 4-44
Restrictions .................................................................................. 4-45
Functional Description ................................................................ 4-45
  Entering Loop Execution .............................................................. 4-45
  Terminating Loop Execution ...................................................... 4-46
  Loop Stack ................................................................................... 4-48
  Loop Address Stack Access ......................................................... 4-48
  Loop Address Stack Status ........................................................... 4-48
  Loop Address Stack Manipulation .............................................. 4-49
  Loop Counter Stack Access ......................................................... 4-49
  Loop Counter Stack Status ........................................................... 4-49
  Loop Counter Stack Manipulation .............................................. 4-50
Counter Based Loops ..................................................................... 4-51
  Reading LCNTR in Counter Based Loops ...................................... 4-51
  IF NOT LCE Condition in Counter Based Loops ...................... 4-52
Arithmetic Loops ........................................................................... 4-53
Indefinite Loops ............................................................................. 4-54
VISA-Related Restrictions on Hardware Loops ......................... 4-54
Restrictions on Ending Loops ......................................................... 4-55
Short Counter Based Loops .............................................................. 4-57
Short Arithmetic Based Loops ......................................................... 4-58
  Restrictions on Short Loops ......................................................... 4-59
  Short Loops Listings .................................................................... 4-60
Contents

Nested Loops ................................................................. 4-67
  Example For Six Nested Loops ................................. 4-69
  Restrictions on Ending Nested Loops ...................... 4-70
Loop Abort ..................................................................... 4-71
  Instruction Driven Loop Abort ................................. 4-71
  Interrupt Driven Loop Abort ................................. 4-73
Loop Abort Restrictions ............................................. 4-74
Loop Resource Manipulation ................................. 4-75
  Popping and Pushing Loop and PC Stack Inside an
  Active Loop ............................................................ 4-76
  Stack Manipulation Restrictions on ADSP-2136x
  Processors ................................................................. 4-78
Cache Control ............................................................... 4-79
  Functional Description .............................................. 4-79
  Conflict Cache for Internal Instruction Fetch ............ 4-79
  Instruction Data Bus Conflicts ............................. 4-80
  Cache Miss .............................................................. 4-80
  Instruction Cache for External Instruction Fetch ....... 4-82
  Block Conflicts .......................................................... 4-83
  Caching Instructions ................................................. 4-83
  Cache Invalidate Instruction .................................... 4-86
Operating Modes ...................................................... 4-87
  Cache Restrictions ..................................................... 4-87
  Cache Disable ............................................................ 4-87
  Cache External Memory Disable (ADSP-214xx) ........ 4-88
Cache Freeze ................................................................. 4-88
Cache Efficiency ........................................................... 4-88
I/O Flags ........................................................................ 4-90
Conditional Instruction Execution .................................. 4-91
  IF Conditions with Complements ................................... 4-92
  DO/UNTIL Terminations Without Complements .............. 4-94
  Operating Modes ....................................................... 4-94
    Conditional Instruction Execution in SIMD Mode .......... 4-94
      Bit Test Flag in SIMD Mode .................................... 4-96
    Conditional Compute .............................................. 4-96
    Conditional Data Move ............................................ 4-97
    Listings for Conditional Register-to-Register Moves ..... 4-97
      Listing 2 – UREG/CUREG to UREG/CUREG
        Register Moves .................................................. 4-98
      Listing 3 – CUREG/UREG to UREG/CUREG
        Registers Moves ............................................... 4-99
      Listing 4 – UREG to UREG/CUREG Register Moves .... 4-100
      Listing 5 – UREG/CUREG to UREG Register Moves .... 4-101
    Listings for Conditional register-to-Memory Moves ..... 4-102
    Conditional Branches .............................................. 4-105
      IF Conditional Branch Instructions ......................... 4-105
      IF Then ELSE Conditional Indirect Branch
        Instructions ....................................................... 4-106
      IF Conditional Branch Limitations in VISA ............... 4-107
    Instruction Pipeline Hazards ..................................... 4-108
Contents

Structural Hazard Stalls ............................................................. 4-108
  Simultaneous Access Over the DMD and PMD Buses .......... 4-109
  DMA Block Conflict with PM or DM Access .................. 4-109
  Core Memory-Mapped Registers ..................................... 4-109
Data Hazard Stalls ................................................................... 4-109
  Multiplier Operand Load Stalls ........................................ 4-110
  DAG Register Load Stalls ............................................... 4-110
  Branch Stalls .................................................................... 4-112
  Conditional Branch Stalls .............................................. 4-113
Control Hazard Stalls ............................................................. 4-115
  Loop Stalls ...................................................................... 4-117
Compiler Related Stalls ........................................................... 4-117
  CJUMP Instruction ........................................................ 4-117
  RFRAME Instruction ..................................................... 4-118
Sequencer Interrupts ................................................................. 4-119
  External Interrupts ........................................................ 4-119
  Software Interrupts ...................................................... 4-120
  Hardware Stack Interrupts ............................................. 4-121
Summary .................................................................................. 4-122

TIMER
  Features .................................................................................. 5-1
  Functional Description ........................................................ 5-1
  Timer Interrupts ................................................................... 5-4
## DATA ADDRESS GENERATORS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Features</td>
<td>6-1</td>
</tr>
<tr>
<td>Functional Description</td>
<td>6-2</td>
</tr>
<tr>
<td>DAG Address Output</td>
<td>6-4</td>
</tr>
<tr>
<td>Address Versus Word Size</td>
<td>6-4</td>
</tr>
<tr>
<td>DAG Register-to-Bus Alignment</td>
<td>6-5</td>
</tr>
<tr>
<td>32-Bit Alignment</td>
<td>6-5</td>
</tr>
<tr>
<td>40-Bit Alignment</td>
<td>6-5</td>
</tr>
<tr>
<td>64-Bit Alignment</td>
<td>6-6</td>
</tr>
<tr>
<td>DAG1 Versus DAG2</td>
<td>6-6</td>
</tr>
<tr>
<td>DAG Instruction Types</td>
<td>6-7</td>
</tr>
<tr>
<td>Long Word Memory Access Restrictions</td>
<td>6-7</td>
</tr>
<tr>
<td>Forced Long Word (LW) Memory Access Instructions</td>
<td>6-8</td>
</tr>
<tr>
<td>Pre-Modify Instruction</td>
<td>6-10</td>
</tr>
<tr>
<td>Post-Modify Instruction</td>
<td>6-11</td>
</tr>
<tr>
<td>Modify Instruction</td>
<td>6-11</td>
</tr>
<tr>
<td>Enhanced Modify Instruction (ADSP-214xx)</td>
<td>6-12</td>
</tr>
<tr>
<td>Immediate Modify Instruction</td>
<td>6-12</td>
</tr>
<tr>
<td>Bit-Reverse Instruction</td>
<td>6-13</td>
</tr>
<tr>
<td>Enhanced Bit-Reverse Instruction (ADSP-214xx)</td>
<td>6-13</td>
</tr>
<tr>
<td>Dual Data Move Instructions</td>
<td>6-13</td>
</tr>
<tr>
<td>Conditional DAG Transfers</td>
<td>6-14</td>
</tr>
<tr>
<td>DAG Breakpoint Units</td>
<td>6-14</td>
</tr>
<tr>
<td>DAG Instruction Restrictions</td>
<td>6-15</td>
</tr>
</tbody>
</table>
Instruction Summary ................................................................. 6-15
Operating Modes .......................................................................... 6-18
   Normal Word (40-Bit) Accesses ........................................ 6-18
   Circular Buffering Mode ...................................................... 6-19
      Circular Buffer Programming Model ............................... 6-21
      Wraparound Addressing ............................................... 6-23
   Broadcast Load Mode .......................................................... 6-24
   Bit-Reverse Mode ................................................................. 6-25
   SIMD Mode ........................................................................ 6-26
      DAG Transfers in SIMD Mode ........................................ 6-27
      Conditional DAG Transfers in SIMD Mode ..................... 6-28
   Alternate (Secondary) DAG Registers ................................. 6-28
   Interrupt Mode Mask ......................................................... 6-30
   Access Modes Summary ...................................................... 6-30
      SISD Mode .................................................................... 6-31
      SIMD Mode Normal Word ............................................. 6-31
      SIMD Mode Short Word ................................................. 6-31
   DAG Interrupts ..................................................................... 6-31
   DAG Status ......................................................................... 6-32

MEMORY
Features ..................................................................................... 7-1
Von Neuman Versus Harvard Architectures ............................... 7-2
   Super Harvard Architecture .............................................. 7-2
   Functional Description ...................................................... 7-4
Contents

Address Decoding of Memory Space ........................................ 7-4
I/O Processor Space ................................................................ 7-5
  IOP Peripheral Registers ................................................. 7-5
  IOP Core Registers .......................................................... 7-7
Writes to IOP Peripheral Registers ...................................... 7-8
  Back to Back Writes to IOP Peripheral Registers .............. 7-8
  Alternate Writes to IOP Peripheral Registers ................. 7-8
Reads from IOP Peripheral Registers .................................. 7-8
IOP Register Core Access .................................................. 7-8
Out of Order Execution ..................................................... 7-9
IOP Register Access Arbitration ........................................ 7-10
Internal Memory Space ..................................................... 7-11
  Internal Memory Interface .............................................. 7-11
  On-Chip Buses ............................................................... 7-11
  Internal Memory Block Architecture .............................. 7-12
  Normal Word Space 48/40-Bit Word Rotations .............. 7-13
Rules for Wrapping Memory Layout ..................................... 7-14
  Mixing Words in Normal Word Space ......................... 7-14
  Mixing 32-Bit Words and 48-Bit Words ...................... 7-16
  32-Bit Word Allocation ............................................... 7-17
Example: Calculating a Starting Address for 32-Bit
  Addresses ................................................................. 7-17
  48-Bit Word Allocation ............................................... 7-18
Memory Address Aliasing .................................................. 7-18
Memory Block Arbitration ................................................ 7-20
Contents

VISA Instruction Arbitration ........................................ 7-22
Using Single Ported Memory Blocks Efficiently ............... 7-22
Shadow Write FIFO ...................................................... 7-23
External Memory Space .............................................. 7-24
Interrupts ....................................................................... 7-24
Internal Interrupt Vector Table ..................................... 7-24
Illegal I/O Processor Register Access ......................... 7-25
Unaligned Forced Long Word Access ......................... 7-25
Internal Memory Access Listings .................................... 7-27
Short Word Addressing of Single-Data in SISD Mode ...... 7-28
Short Word Addressing of Dual-Data in SISD Mode ...... 7-31
Short Word Addressing of Single-Data in SIMD Mode ...... 7-33
Short Word Addressing of Dual-Data in SIMD Mode ...... 7-35
32-Bit Normal Word Addressing of Single-Data in
SISD Mode .................................................................. 7-37
32-Bit Normal Word Addressing of Dual-Data in
SISD Mode .................................................................. 7-39
32-Bit Normal Word Addressing of Single-Data in
SIMD Mode .................................................................. 7-41
32-Bit Normal Word Addressing of Dual-Data in
SIMD Mode .................................................................. 7-43
Extended-Precision Normal Word Addressing of
Single-Data ................................................................. 7-45
Extended-Precision Normal Word Addressing of
Dual-Data ................................................................. 7-47
Long Word Addressing of Single-Data ......................... 7-49
Contents

Long Word Addressing of Dual-Data .............................................. 7-51
Broadcast Load Access .................................................................. 7-53
Mixed-Word Width Addressing of Long Word with
  Short Word .............................................................................. 7-62
Mixed-Word Width Addressing of Long Word with
  Extended Word ....................................................................... 7-64

JTAG TEST EMULATION PORT

Features ........................................................................................ 8-1
Functional Description .................................................................. 8-1
  TAP Controller ....................................................................... 8-2
  JTAG Test Access Port ............................................................. 8-3
Instruction Registers ................................................................. 8-5
  Emulation Instruction Registers (Private) ................................. 8-5
Operating Modes ................................................................. 8-6
  Boundary Scan Mode ............................................................. 8-6
  Boundary Scan Register Instructions ........................................ 8-7
Emulation Space Mode ............................................................ 8-7
Emulation Control .................................................................. 8-8
  Emulation Status Register (EMUSTAT) ............................... 8-9
  Emulation Control Register (EMUCTL) ............................. 8-9
Address Breakpoints .............................................................. 8-13
  Address Breakpoint Registers ............................................. 8-14
Conditional Breakpoints ........................................................ 8-15
  Event Count Register .......................................................... 8-16
Contents

Emulation Cycle Counting .................................................. 8-17
Enhanced Emulation Mode ................................................. 8-17
Statistical Profiling ........................................................... 8-17
User Space Mode ............................................................. 8-18
User Breakpoint Control ................................................... 8-18
User Breakpoint Status ...................................................... 8-18
User Breakpoint System Exception Handling ..................... 8-19
Programming Model User Breakpoints .............................. 8-19
Programming Examples .................................................... 8-20
Emulation vs. User Space Mode ............................................. 8-21
Single Step ....................................................................... 8-22
Instruction Pipeline Fetch Inputs .................................. 8-22
JTAG Interrupts ........................................................................ 8-23
Interrupt Types ..................................................................... 8-23
Entering Into Emulation Space ........................................... 8-24
Breakpoints .......................................................................... 8-24
Software Breakpoints ........................................................ 8-24
Automatic Breakpoints ..................................................... 8-25
Hardware Breakpoints ....................................................... 8-25
General Restrictions on Software Breakpoints ................. 8-26
JTAG Register Effect Latency ............................................. 8-26
References ........................................................................ 8-27

INSTRUCTION SET TYPES

Instruction Groups ............................................................. 9-2
Contents

Instruction Set Notation Summary ................................................ 9-2

Group I – Conditional Compute and Move or Modify Instructions  9-4
  Type 1a ISA/VISA (compute + mem dual data move)
  Type 1b VISA (mem dual data move) ........................................ 9-7
  Type 2a ISA/VISA (cond + compute)
  Type 2b VISA (compute)
  Type 2c VISA (short compute) ................................................. 9-10
  Type 3a ISA/VISA (cond + comp + mem data move)
  Type 3b VISA (cond + mem data move)
  Type 3c VISA (mem data move) ............................................... 9-12
  Type 4a ISA/VISA (cond + comp + mem data move
  with 6-bit immediate modifier)
  Type 4b VISA (cond + mem data move
  with 6-bit immediate modifier) .............................................. 9-17
  Type 5a ISA/VISA (cond + comp + reg data move)
  Type 5b VISA (cond + reg data move) .................................... 9-22
  Type 6 ISA/VISA (cond + shift imm + mem data move) ............ 9-25
  Type 7a ISA/VISA (cond + comp + index modify)
  Type 7b ISA/VISA (cond + index modify) ................................... 9-28

Group II – Conditional Program Flow Control Instructions .......... 9-30
  Type 8a ISA/VISA (cond + branch) ............................................ 9-32
  Type 9a ISA/VISA (cond + Branch + comp/else comp) .............. 9-35
  Type 10 ISA (cond + branch + else comp + mem data move) ... 9-40
  Type 11a ISA/VISA (cond + branch return + comp/else comp)
  Type 11c VISA (cond + branch return) ..................................... 9-44
  Type 12a ISA/VISA (do until loop counter expired) ............... 9-48
  Type 13a ISA/VISA (do until termination) ................................ 9-49
Contents

Group III – Immediate Data Move
Instructions ................................................................. 9-51
  Type 14a ISA/VISA (mem data move) ......................... 9-53
  Type 15a ISA/VISA (<data32> move)
  Type 15b VISA (<data7> move) ................................. 9-56
  Type 16a ISA/VISA (<data32> move)
  Type 16b VISA (<data16> move) ......................... 9-60
  Type 17a ISA/VISA (<data32> move)
  Type 17b VISA (<data16> move) ......................... 9-62
Group IV – Miscellaneous Instructions ........................... 9-64
  Type 18a ISA/VISA (register bit manipulation) ............ 9-66
  Type 19a ISA/VISA (index modify/bitrev) .................... 9-69
  Type 20a ISA/VISA (push/pop stack) ......................... 9-70
  Type 21a ISA/VISA (nop)
  Type 21c VISA (nop) .............................................. 9-71
  Type 22a ISA/VISA (idle/emuidle) ............................. 9-72
  Type 25a ISA/VISA (cjump/rframe)
  Type 25c VISA (RFRAME) ........................................ 9-73

COMPUTATION INSTRUCTIONS

Compute Field ............................................................. 10-1
ALU Operations .......................................................... 10-2
  Rn = Rx + Ry .......................................................... 10-2
  Rn = Rx – Ry .......................................................... 10-3
  Rn = Rx + Ry + CI .................................................. 10-4
  Rn = Rx – Ry + CI – 1 ............................................. 10-5
  Rn = (Rx + Ry)/2 .................................................. 10-6
Contents

COMP(Rx, Ry) ..................................................................... 10-7
COMPU(Rx, Ry) .................................................................. 10-8
Rn = Rx + CI ........................................................................ 10-9
Rn = Rx + CI – 1 ................................................................. 10-10
Rn = Rx + 1 ........................................................................ 10-11
Rn = Rx – 1 ........................................................................ 10-12
Rn = –Rx ............................................................................ 10-13
Rn = ABS Rx ...................................................................... 10-14
Rn = PASS Rx ..................................................................... 10-15
Rn = Rx AND Ry ............................................................... 10-16
Rn = Rx OR Ry ................................................................... 10-17
Rn = Rx XOR Ry ................................................................. 10-18
Rn = NOT Rx .................................................................... 10-19
Rn = MIN(Rx, Ry) .............................................................. 10-20
Rn = MAX(Rx, Ry) ............................................................. 10-21
Rn = CLIP Rx BY Ry ........................................................... 10-22
Fn = Fx + Fy ....................................................................... 10-23
Fn = Fx – Fy ....................................................................... 10-24
Fn = ABS (Fx + Fy) ............................................................ 10-25
Fn = ABS (Fx – Fy) ............................................................ 10-26
Fn = (Fx + Fy)/2 ................................................................. 10-27
COMP(Fx, Fy) .................................................................... 10-28
Fn = –Fx ............................................................................ 10-29
Fn = ABS Fx ..................................................................... 10-30
Contents

Fn = PASS Fx ................................................................. 10-31
Fn = RND Fx ................................................................. 10-32
Fn = SCALB Fx BY Ry .................................................. 10-33
Rn = MANT Fx ............................................................. 10-34
Rn = LOGB Fx .............................................................. 10-35
Rn = FIX Fx
Rn = TRUNC Fx
Rn = FIX Fx BY Ry
Rn = TRUNC Fx BY Ry ................................................ 10-36
Fn = FLOAT Rx BY Ry
Fn = FLOAT Rx ........................................................... 10-38
Fn = RECIPS Fx ........................................................... 10-39
Fn = RSQRTS Fx .......................................................... 10-41
Fn = Fx COPYSIGN Fy .................................................. 10-43
Fn = MIN(Fx, Fy) ......................................................... 10-44
Fn = MAX(Fx, Fy) ......................................................... 10-45
Fn = CLIP Fx BY Fy ....................................................... 10-46
Multiplier Operations ...................................................... 10-47
Mod1, Mod2, and Mod3 Modifiers .................................. 10-47
Rn = Rx * Ry (mod1)
MRF = Rx * Ry (mod1)
MRB = Rx * Ry (mod1) .................................................. 10-48
Rn = MRF + Rx * Ry (mod1)
Rn = MRB + Rx * Ry (mod1)
MRF = MRF + Rx * Ry (mod1)
MRB = MRB + Rx * Ry (mod1) ........................................ 10-49
Rn = MRF – Rx * Ry (mod1)
Rn = MRB – Rx * Ry (mod1)
MRF = MRF – Rx * Ry (mod1)
MRB = MRB – Rx * Ry (mod1) ............................................. 10-50

Rn = SAT MRF (mod2)
Rn = SAT MRB (mod2)
MRF = SAT MRF (mod2)
MRB = SAT MRB (mod2) ..................................................... 10-51

Rn = RND MRF (mod3)
Rn = RND MRB (mod3)
MRF = RND MRF (mod3)
MRB = RND MRB (mod3) .................................................... 10-52

MRF = 0
MRB = 0 ................................................................................ 10-53

MRxF/B = Rn
Rn = MRxF/B ........................................................................ 10-54

Fn = Fx * Fy ........................................................................... 10-55

Shifter Operations ..................................................................... 10-56

Rn = LSHIFT Rx BY Ry
Rn = LSHIFT Rx BY <data8> ................................................. 10-56

Rn = Rn OR LSHIFT Rx BY Ry
Rn = Rn OR LSHIFT Rx BY <data8> ..................................... 10-57

Rn = ASHIFT Rx BY Ry
Rn = ASHIFT Rx BY <data8> ................................................ 10-58

Rn = Rn OR ASHIFT Rx BY Ry
Rn = Rn OR ASHIFT Rx BY <data8> ................................. 10-59

Rn = ROT Rx BY Ry
Rn = ROT Rx BY <data8> ....................................................... 10-60
Contents

Rn = BCLR Rx BY Ry ............................................................... 10-61
Rn = BCLR Rx BY <data8> ...................................................... 10-61
Rn = BSET Rx BY Ry ............................................................. 10-62
Rn = BSET Rx BY <data8> ...................................................... 10-62
Rn = BTGL Rx BY Ry ............................................................. 10-63
Rn = BTGL Rx BY <data8> ...................................................... 10-63
BTST Rx BY Ry ................................................................. 10-64
BTST Rx BY <data8> ............................................................. 10-64
Rn = FDEP Rx BY Ry ............................................................. 10-65
Rn = FDEP Rx BY <bit6>:<len6> ............................................. 10-65
Rn = Rn OR FDEP Rx BY Ry .................................................. 10-67
Rn = Rn OR FDEP Rx BY <bit6>:<len6> .................................. 10-67
Rn = FDEP Rx BY Ry (SE) ..................................................... 10-69
Rn = FDEP Rx BY <bit6>:<len6> (SE) ....................................... 10-69
Rn = Rn OR FDEP Rx BY Ry (SE) ......................................... 10-71
Rn = Rn OR FDEP Rx BY <bit6>:<len6> (SE) ......................... 10-71
Rn = FEXT Rx BY Ry ............................................................. 10-73
Rn = FEXT Rx BY <bit6>:<len6> ............................................. 10-73
Rn = FEXT Rx BY Ry (SE) ..................................................... 10-75
Rn = FEXT Rx BY <bit6>:<len6> (SE) ....................................... 10-75
Rn = EXP Rx ................................................................. 10-77
Rn = EXP Rx (EX) ............................................................. 10-78
Rn = LEFTZ Rx ................................................................. 10-79
Rn = LEFTO Rx ................................................................. 10-80
Rn = FPACK Fx ................................................................. 10-81
F0 = FUNPACK Rx ............................................................. 10-82
BITDEP Rx by Ry|<bitlen> .................................................... 10-83
Rn = BITEXT Rx|<bitlen>[(NU)] ........................................ 10-85
Rn = BFFWRP ................................................................. 10-87
BFFWRP = Rn|<value7> ..................................................... 10-88
Multifunction Computations ........................................... 10-89
  Fixed-Point ALU (dual Add and Subtract) ......................... 10-89
  Floating-Point ALU (dual Add and Subtract) ...................... 10-89
  Fixed-Point Multiplier and ALU ..................................... 10-89
  Floating-Point Multiplier and ALU ................................ 10-90
  Fixed-Point Multiplier and ALU (dual Add and Subtract) .... 10-90
  Floating Point Multiplier and ALU (dual Add and Subtract) 10-90

INSTRUCTION OPCODES

Instruction Opcode Types ............................................. 11-1
  Group I – Conditional Compute and Move or Modify Instructions 11-4
    Type 1a ISA/VISA (compute + mem dual data move) ........... 11-4
    Type 1b VISA (mem dual data move) .............................. 11-5
    Type 2a ISA/VISA (cond + compute) ............................... 11-6
    Type 2b VISA (compute) ............................................ 11-7
    Type 2c VISA (short compute) ..................................... 11-8
    Type 3a ISA/VISA (cond + comp + mem data move) ............. 11-9
    Type 3b VISA (cond + mem data move) ............................ 11-10
    Type 3c VISA (mem data move) ................................... 11-11
    Type 4a ISA/VISA (cond + comp + mem data move with 6-bit immediate modifier) ............................... 11-12
Type 4b VISA (cond + mem data move with 6-bit immediate modifier) ......................................................... 11-13
Type 5a ISA/VISA (cond + comp + reg data move) .............. 11-14
  Ureg = Ureg transfer ........................................................... 11-14
  Dreg <-> CDreg swap .......................................................... 11-14
Type 5b VISA (cond + reg data move) ................................. 11-16
  Ureg = Ureg move ........................................................... 11-16
  Dreg <-> CDreg swap .......................................................... 11-16
Type 6 ISA/VISA (cond + shift imm + mem data move) ................................................................ 11-17
  with mem data move .......................................................... 11-17
  without mem data move ...................................................... 11-17
Type 7a ISA/VISA (cond + comp + index modify) ................ 11-19
Type 7b VISA (cond + index modify) .................................. 11-20
Group II – Conditional Program Flow Control Instructions ..... 11-21
Type 8a ISA/VISA (cond + branch) ..................................... 11-21
  direct branch ................................................................. 11-21
  PC-relative branch .......................................................... 11-21
Type 9a ISA/VISA (cond + branch + comp else/comp) ............. 11-23
  with indirect branch ....................................................... 11-23
  with PC-relative branch .................................................. 11-23
Type 9b VISA (cond + branch) .......................................... 11-25
  with indirect branch ....................................................... 11-25
  with PC-relative branch .................................................. 11-25
Type 10 ISA (cond + branch + else comp + mem data move) ................................................................. 11-27
  with indirect jump .......................................................... 11-27
  with PC-relative jump ..................................................... 11-27
Type 11a ISA/VISA (cond + branch return + comp/else comp) ........................................................................... 11-29
  branch return from subroutine ........................................ 11-29
  branch return from interrupt .......................................... 11-29
Type 11c VISA (cond + branch return) ................................................... 11-30
  branch return from subroutine ........................................ 11-30
  branch return from interrupt .......................................... 11-30
Type 12a ISA/VISA (do until loop counter expired) .......................... 11-31
  with immediate loop counter load ................................... 11-31
  with Ureg load .................................................................. 11-31
Group III – Immediate Data Move Instructions ................................. 11-32
  Type 13a ISA/VISA (do until termination) ......................... 11-32
  Type 14a ISA/VISA (mem data move) ................................ 11-33
  Type 15a ISA/VISA (<data32> move) .................................. 11-34
  Type 15b VISA (<data7> move) ......................................... 11-35
  Type 16a ISA/VISA (<data32> move) .................................. 11-36
  Type 16b VISA (<data16> move) ......................................... 11-37
  Type 17a ISA/VISA (<data32> move) .................................. 11-38
  Type 17b VISA (<data16> move) ......................................... 11-39
Group IV – Miscellaneous Instructions ....................................... 11-40
  Type 18a ISA/VISA (register bit manipulation) .................... 11-40
Contents

Type 19a ISA/VISA (index modify/bitrev) ......................... 11-41
  with modify ................................................................. 11-41
  with bit-reverse .......................................................... 11-41
Type 20a ISA/VISA (push/pop stack) .............................. 11-42
Type 21a ISA/VISA (nop) ................................................. 11-43
Type 21c VISA (nop) ....................................................... 11-44
Type 22a ISA/VISA (idle/emuidle) ................................. 11-45
Type 22c VISA (idle/emuidle) ......................................... 11-46
Type 25a ISA/VISA (cjump/rframe) ............................... 11-47
  cjump/rframe ............................................................... 11-47
  with PC-relative branch ............................................. 11-47
  RFRAME ................................................................... 11-48
Type 25c VISA (RFRAME) .............................................. 11-48

Computation Type Opcodes ............................................. 11-49
  ALU Opcodes ............................................................... 11-51
  Multiplier Opcodes ....................................................... 11-53
    Mod1 Modifiers ......................................................... 11-55
    Mod2 Modifiers ......................................................... 11-56
    Mod3 Modifiers ......................................................... 11-56
    MRxF/B = Rn
    Rn = MRxF/B ........................................................... 11-56
  Shifter Opcodes .......................................................... 11-58
  Multifunction Opcodes ............................................... 11-61
  Dual ALU (Parallel Add and Subtract) .......................... 11-61
    Compute Field (Fixed-Point) ...................................... 11-61
Contents

Compute Field (Floating-Point) ........................................... 11-61
Multiplier and Dual ALU (Parallel Add and Subtract) ........... 11-62
Compute Field (Fixed-Point) ............................................. 11-62
Compute Field (Floating-Point) ........................................ 11-62
Multiplier and ALU ......................................................... 11-64
Compute Field (Fixed-Point) ........................................... 11-64
Compute Field (Floating-Point) ........................................ 11-64
Register Opcodes ............................................................. 11-66
Non Universal Registers .................................................... 11-66
Universal Register Opcodes ............................................... 11-67
Condition and Termination Opcodes ................................... 11-69

REGISTERS

Notes on Reading Register Drawings .................................. A-2
Mode Control 1 Register (MODE1) ....................................... A-3
Mode Control 2 Register (MODE2) ....................................... A-7
Program Sequencer Registers .......................................... A-8
  Fetch Address Register (FADDR) ..................................... A-9
  Decode Address Register (DADDR) .................................. A-9
  Program Counter Register (PC) ...................................... A-10
  Program Counter Stack Register (PCSTK) ......................... A-10
  Program Counter Stack Pointer Register (PCSTKP) ............ A-11
Loop Registers ............................................................... A-11
  Loop Address Stack Register (LADDR) ............................. A-11
  Loop Counter Register (LCNTR) ..................................... A-12
<table>
<thead>
<tr>
<th>Content</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current Loop Counter Register (CURLCNTR)</td>
<td>A-12</td>
</tr>
<tr>
<td>Timer Registers</td>
<td>A-12</td>
</tr>
<tr>
<td>Timer Period Register (TPERIOD)</td>
<td>A-12</td>
</tr>
<tr>
<td>Timer Count Register (TCOUNT)</td>
<td>A-12</td>
</tr>
<tr>
<td>Flag I/O Register (FLAGS)</td>
<td>A-13</td>
</tr>
<tr>
<td>Processing Element Registers</td>
<td>A-14</td>
</tr>
<tr>
<td>PEx Data Registers (Rx)</td>
<td>A-14</td>
</tr>
<tr>
<td>PEy Data Registers (Sx)</td>
<td>A-14</td>
</tr>
<tr>
<td>Alternate Data Registers (Rx’, Sx’)</td>
<td>A-15</td>
</tr>
<tr>
<td>PEx Multiplier Results Registers (MRFx, MRBx)</td>
<td>A-15</td>
</tr>
<tr>
<td>PEy Multiplier Results Registers (MSFx, MSBx)</td>
<td>A-15</td>
</tr>
<tr>
<td>Processing Status Registers</td>
<td>A-15</td>
</tr>
<tr>
<td>Arithmetic Status Registers (ASTATx and ASTATy)</td>
<td>A-16</td>
</tr>
<tr>
<td>Sticky Status Registers (STKYx and STKYy)</td>
<td>A-20</td>
</tr>
<tr>
<td>Data Address Generator Registers</td>
<td>A-24</td>
</tr>
<tr>
<td>Index Registers (Ix)</td>
<td>A-24</td>
</tr>
<tr>
<td>Modify Registers (Mx)</td>
<td>A-24</td>
</tr>
<tr>
<td>Length and Base Registers (Lx, Bx)</td>
<td>A-25</td>
</tr>
<tr>
<td>Alternate DAG Registers (Ix’, Mx’, Lx’, Bx’)</td>
<td>A-25</td>
</tr>
<tr>
<td>Miscellaneous Registers</td>
<td>A-25</td>
</tr>
<tr>
<td>Bus Exchange Register (PX)</td>
<td>A-25</td>
</tr>
<tr>
<td>User-Defined Status Registers (USTATx)</td>
<td>A-26</td>
</tr>
<tr>
<td>Emulation Counter Registers (EMUCLKx)</td>
<td>A-26</td>
</tr>
<tr>
<td>Universal Register Effect Latency</td>
<td>A-26</td>
</tr>
</tbody>
</table>
Contents

Interrupt Registers ........................................................................... A-31
  Interrupt Latch Register (IRPTL) ........................................ A-31
  Interrupt Mask Register (IMASK) ........................................ A-31
  Interrupt Mask Pointer Register (IMASKP) ......................... A-31
  Interrupt Register (LIRPTL) ............................................. A-36
  Mode Mask Register (MMASK) ......................................... A-39
Memory-Mapped Registers .......................................................... A-41
  System Control Register (SYSCTL) .............................. A-41
  Revision ID Register (REVPID) ........................................ A-43
  Breakpoint Control Register (BRKCTL) ............................ A-43
  Enhanced Emulation Status Register (EEMUSTAT) ........... A-47
Register Listing ............................................................................. A-50

CORE INTERRUPT CONTROL
  Interrupt Acknowledge ......................................................... B-1
  Interrupt Priority ............................................................... B-2
  Interrupt Vector Tables ..................................................... B-2

NUMERIC FORMATS
  IEEE Single-Precision Floating-Point Data Format .............. C-1
  Extended-Precision Floating-Point Format ......................... C-3
  Short Word Floating-Point Format .................................. C-4
  Packing for Floating-Point Data ....................................... C-4
  Fixed-Point Formats ......................................................... C-6
Contents

GLOSSARY

INDEX
Thank you for purchasing and developing systems using SHARC® processors from Analog Devices, Inc.

**Purpose of This Manual**

The *SHARC Processor Programming Reference* provides architectural and programming information about the SHARC processors. The architectural descriptions cover functional blocks and buses, including features and processes that they support. The manual also provides information on the I/O capabilities and peripherals supported on these processors. For timing, electrical, and package specifications, see the processor-specific data sheet. The programming information covers the instruction set and compute operations.

**Intended Audience**

The primary audience for this manual is a programmer who is familiar with Analog Devices processors. This manual assumes that the audience has a working knowledge of the appropriate processor architecture and instruction set. Programmers who are unfamiliar with Analog Devices processors can use this manual, but should supplement it with other texts, such as the appropriate hardware reference manuals and data sheets, that describe their target architecture.
This manual provides detailed information about the SHARC processor family in the following chapters. Please note that there are differences in this section from previous manual revisions.

- Chapter 1, “Introduction”
  Provides an architectural overview of the SHARC processors.

- Chapter 2, “Register Files”
  Describes the core register files including the data exchange register (PX).

- Chapter 3, “Processing Elements”
  Describes the arithmetic/logic units (ALUs), multiplier/accumulator units, and shifter. The chapter also discusses data formats, data types, and register files.

- Chapter 4, “Program Sequencer”
  Describes the operation of the program sequencer, which controls program flow by providing the address of the next instruction to be executed. The chapter also discusses loops, subroutines, jumps, interrupts, exceptions, and the IDLE instruction.

- Chapter 5, “Timer”
  Describes the operation of the processor’s core timer.

- Chapter 6, “Data Address Generators”
  Describes the Data Address Generators (DAGs), addressing modes, how to modify DAG and pointer registers, memory address alignment, and DAG instructions.

- Chapter 7, “Memory”
  Describes aspects of processor memory including internal memory, address and data bus structure, and memory accesses.
• Chapter 8, “JTAG Test Emulation Port”
  Discusses the JTAG standard and how to use the SHARC processors in a test environment. Includes boundary-scan architecture, instruction and boundary registers, and breakpoint control registers.

• Chapter 9, “Instruction Set Types”
  Provides reference information for the machine language opcode for the processor.

• Chapter 10, “Computation Instructions”
  Describes each compute operation in detail. Compute operations execute in the multiplier, the ALU, and the shifter

• Chapter 11, “Instruction Opcodes”
  Describes each compute operation opcode field.

• Appendix A, “Registers”
  Provides register and bit descriptions for all of the registers that are used to control the operation of the SHARC processor core.

• Appendix B, “Core Interrupt Control”
  Provides interrupt vector tables.

• Appendix C, “Numeric Formats”
  Provides descriptions of the supported data formats.
What’s New in This Manual

This is Revision 2.1 of *SHARC Processor Programming Reference*. Since the last revision of this manual the following has changed or been corrected.

- The revision 2.0 chapter “Data Bus Exchange” has been re titled to “Register Files”.
- All of the chapters have been reorganized. In general they contain the follow heading structure: Features, Functional Description, Operating Modes and Interrupts.
- The Instructions chapters from revision 2.0—Chapter 9 Instruction Set, Chapter 10 Instruction Set (VISA), Chapter 11, Computations Reference, and Appendix A, Instruction Set Quick Reference have been reorganized. See “Manual Contents” above for a description of this manuals structure.
- Errors and omissions, including removing legacy information which did not apply to the products described in this manual, have been corrected—most of which were not reported through the errata system.

Technical or Customer Support

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

- E-mail tools questions to processor.tools.support@analog.com
Preface

- E-mail processor questions to
  processor.support@analog.com (World wide support)
  processor.europe@analog.com (Europe support)
  processor.china@analog.com (China support)

- Phone questions to 1-800-ANALOGD

- Contact your Analog Devices, Inc. local sales office or authorized distributor

- Send questions by mail to:
  Analog Devices, Inc.
  One Technology Way
  P.O. Box 9106
  Norwood, MA 02062-9106
  USA

Registration for MyAnalog.com

MyAnalog.com is a free feature of the Analog Devices Web site that allows customization of a Web page to display only the latest information about products you are interested in. Click Register to use this site. Registration takes about five minutes and serves as a means to select the information you want to receive.

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

EngineerZone

EngineerZone is a technical support forum from Analog Devices. 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.

Social Networking Web Sites

You can now follow Analog Devices SHARC development on Twitter and LinkedIn. To access:

- Twitter: http://twitter.com/ADISHARC
- LinkedIn: Network with the LinkedIn group, Analog Devices SHARC: http://www.linkedin.com

Supported Processors

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

ADSP-2106x, ADSP-2116x, ADSP-2126x, ADSP-2136x, ADSP-2137x, ADSP-2146x, ADSP-2147x and ADSP-2148x.

Product Information

Product information can be obtained from the Analog Devices Web site, VisualDSP++ online Help system, and a technical library CD.
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.com is a free feature of the Analog Devices Web site that allows customization of a Web page to display only the latest information on products you are interested in. You can also choose to receive weekly e-mail notifications containing updates to the Web pages that meet your interests, including documentation errata against all manuals. MyAnalog.com provides access to books, application notes, data sheets, code examples, and more.

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

VisualDSP++ Online Documentation

Online documentation comprises the VisualDSP++ Help system, software tools manuals, hardware tools manuals, processor manuals, Dinkum Abridged C++ library, and FLEXnet License Tools software documentation. You can search easily across the entire VisualDSP++ documentation set for any topic of interest.

For easy printing, supplementary Portable Documentation Format (.pdf) files for all manuals are provided on the VisualDSP++ installation CD.

Each documentation file type is described as follows.
Product Information

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.chm</td>
<td>Help system files and manuals in Microsoft help format</td>
</tr>
<tr>
<td>.htm</td>
<td>Dinkum Abridged C++ library and FLEXnet License Tools software documentation</td>
</tr>
<tr>
<td>.html</td>
<td>Viewing and printing the .html files requires a browser, such as Internet</td>
</tr>
<tr>
<td></td>
<td>Explorer 6.0 (or higher).</td>
</tr>
<tr>
<td>.pdf</td>
<td>VisualDSP++ and processor manuals in PDF format.</td>
</tr>
<tr>
<td></td>
<td>Viewing and printing the .pdf files requires a PDF reader, such as Adobe</td>
</tr>
<tr>
<td></td>
<td>Acrobat Reader (4.0 or higher).</td>
</tr>
</tbody>
</table>

Technical Library CD

The technical library CD contains seminar materials, product highlights, a selection guide, and documentation files of processor manuals, VisualDSP++ software manuals, and hardware tools manuals for the following processor families: Blackfin®, SHARC, TigerSHARC®, ADSP-218x, and ADSP-219x.

To order the technical library CD, go to http://www.analog.comprocessors/manuals, navigate to the manuals page for your processor, click the request CD check mark, and fill out the order form.

Data sheets, which can be downloaded from the Analog Devices Web site, change rapidly, and therefore are not included on the technical library CD. Technical manuals change periodically. Check the Web site for the latest manual revisions and associated documentation errata.
## Notation Conventions

Text conventions used in this manual are identified and described as follows. Note that additional conventions, which apply only to specific chapters, may appear throughout this document.

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Close command (File menu)</td>
<td>Titles in reference sections indicate the location of an item within the VisualDSP++ environment's menu system (for example, the Close command appears on the File menu).</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 ellipse; 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>
<tr>
<td>SWRST Software Reset register</td>
<td>Register names appear in UPPERCASE and a special typeface. The descriptive names of registers are in mixed case and regular typeface.</td>
</tr>
<tr>
<td>TMR0E, RESET</td>
<td>Pin names appear in UPPERCASE and a special typeface. Active low signals appear with an OVERBAR.</td>
</tr>
<tr>
<td>DRx, I[3:0] SMS[3:0]</td>
<td>Register, bit, and pin names in the text may refer to groups of registers or pins: A lowercase x in a register name (DRx) indicates a set of registers (for example, DR2, DR1, and DR0). \n</td>
</tr>
<tr>
<td>0xabcd, b#1111</td>
<td>A 0x prefix indicates hexadecimal; a b# prefix indicates binary.</td>
</tr>
</tbody>
</table>
# Notation Conventions

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Note" /></td>
<td><strong>Note</strong>: For correct operation, ...&lt;br&gt;A Note: provides supplementary information on a related topic. In the online version of this book, the word <strong>Note</strong> appears instead of this symbol.</td>
</tr>
<tr>
<td><img src="image" alt="Caution" /></td>
<td><strong>Caution</strong>: Incorrect device operation may result if ...&lt;br&gt;<strong>Caution</strong>: Device damage may result if ...&lt;br&gt;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 <strong>Caution</strong> appears instead of this symbol.</td>
</tr>
<tr>
<td><img src="image" alt="Warning" /></td>
<td><strong>Warning</strong>: Injury to device users may result if ...&lt;br&gt;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 <strong>Warning</strong> appears instead of this symbol.</td>
</tr>
</tbody>
</table>
1 INTRODUCTION

The SHARC processors are high performance 32-/40-bit processors used for medical imaging, communications, military, audio, test equipment, 3D graphics, speech recognition, motor control, imaging, automotive, and other applications. By adding on-chip SRAM, integrated I/O peripherals, and an additional processing element for single-instruction, multiple-data (SIMD) support, this processor builds on the ADSP-21000 family processor core to form a complete system-on-a-chip.

The SHARC processors are comprised of several distinct groups, the ADSP-21362/3/4/5/6 processors, the ADSP-21367/8/9 and ADSP-21371/5 processors, and the ADSP-214xx processors. The groups are differentiated by on-chip memories, peripheral choices, packaging, and operating speeds. However, the core processor operates in the same way in all groups so this manual applies to all groups. Where differences exist (such as external memory interfacing) they will be noted.

SHARC Design Advantages

A digital signal processor’s data format determines its ability to handle signals of differing precision, dynamic range, and signal-to-noise ratios. Because floating-point math reduces the need for scaling and probability of overflow, using a floating-point processor can ease algorithm and software development. The extent to which this is true depends on the floating-point processor’s architecture. Consistency with IEEE workstation simulations and the elimination of scaling are clearly two ease-of-use advantages. High level language programmability, large address spaces, and wide dynamic range allow system development time to
be spent on algorithms and signal processing concerns, rather than assembly language coding, code paging, and error handling. The processors are highly integrated, 32-/40-bit floating-point processors that provide many of these design advantages.

The SHARC processor architecture balances a high performance processor core with four high performance memory blocks and two input/output (I/O) buses. In the core, every instruction can execute in a single cycle. The buses and instruction cache provide rapid, unimpeded data flow to the core to maintain the execution rate.

The processors address the five central requirements for signal processing:

1. **Fast, flexible arithmetic.** The ADSP-21000 family processors execute all instructions in a single cycle. They provide fast cycle times and a complete set of arithmetic operations. The processors are IEEE floating-point compatible and allow either interrupt on arithmetic exception or latched status exception handling.

2. **Unconstrained data flow.** The processors have a Super Harvard Architecture combined with a ten-port data register file. For more information, see “Register Files” on page 2-1. In every cycle, the processor can write or read two operands to or from the register file, supply two operands to the ALU, supply two operands to the multiplier, and receive three results from the ALU and multiplier. The processor’s 48-bit orthogonal instruction word supports parallel data transfers and arithmetic operations in the same instruction.

3. **40-Bit extended precision.** The processor handles 32/40-bit IEEE floating-point format, 32-bit integer and fractional formats (twos-complement and unsigned). The processors carry extended precision throughout their computation units, limiting intermediate data truncation errors. For fixed point operations up to 80 bits of precision are maintained during multiply-accumulate operations.
4. **Dual address generators.** The processor has two data address generators (DAGs) that provide immediate or indirect (pre- and post-modify) addressing. Modulus, bit-reverse, and broadcast operations are supported with no constraints on data buffer placement.

5. **Efficient program sequencing.** In addition to zero-overhead loops, the processor supports single-cycle setup and exit for loops. Loops are both nestable (six levels in hardware) and interruptable. The processors support both delayed and non-delayed branches.

**Architectural Overview**

The SHARC processors form a complete system-on-a-chip, integrating a large, high speed SRAM and I/O peripherals supported by I/O buses. The following sections summarize the features of each functional block.

**Processor Core**

The processor core consists of two processing elements (each with three computation units and data register file), a program sequencer, two DAGs, a timer, and an instruction cache. All processing occurs in the processor core. The following list and Figure 1-1 describes some of the features of the SHARC core processor.
Figure 1-1. SHARC SIMD Core Block Diagram
Dual Processing Elements

The processor core contains two processing elements: PEx and PEy. Each element contains a data register file and three independent computation units: an arithmetic logic unit (ALU), a multiplier with an 80-bit fixed-point accumulator, and a shifter. For meeting a wide variety of processing needs, the computation units process data in three formats: 32-bit fixed-point, 32-bit floating-point, and 40-bit floating-point. The floating-point operations are single-precision IEEE-compatible. The 32-bit floating-point format is the standard IEEE format, whereas the 40-bit extended-precision format has eight additional least significant bits (LSBs) of mantissa for greater accuracy.

The ALU performs a set of arithmetic and logic operations on both fixed-point and floating-point formats. The multiplier performs floating-point or fixed-point multiplication and fixed-point multiply/accumulate or multiply/cumulative-subtract operations. The shifter performs logical and arithmetic shifts, bit manipulation, bit-wise field deposit and extraction, and exponent derivation operations on 32-bit operands. These computation units complete all operations in a single cycle; there is no computation pipeline. The output of any unit may serve as the input of any unit on the next cycle. All units are connected in parallel, rather than serially. In a multifunction computation, the ALU and multiplier perform independent, simultaneous operations.

Each processing element has a general-purpose data register file that transfers data between the computation units and the data buses and stores intermediate results. A register file has two sets (primary and secondary) of 16 general-purpose registers each for fast context switching. All of the registers are 40 bits wide. The register file, combined with the core processor’s Super Harvard Architecture, allows unconstrained data flow between computation units and internal memory.

Processing element (PEx). PEx processes all computational instructions whether the processors are in single-instruction, single-data (SISD) or single-instruction, multiple-data (SIMD) mode. This element corresponds to
Architectural Overview

the computational units and register file in previous ADSP-2106x family processors.

Complimentary processing element (PEy). PEy processes each computational instruction in lock-step with PEx, but only processes these instructions when the processors are in SIMD mode. Because many operations are influenced by this mode, more information on SIMD is available in multiple locations:

- For information on PEy operations, see “Processing Elements” on page 3-1.
- For information on data accesses in SIMD mode, and data addressing in SIMD mode, see “Internal Memory Access Listings” on page 7-27.
- For information on SIMD programming, see Chapter 9, Instruction Set Types, and Chapter 10, Computation Instructions.

Program Sequence Control

Internal controls for program execution come from four functional blocks: program sequencer, data address generators, core timer, and instruction cache. Two dedicated address generators and a program sequencer supply addresses for memory accesses. Together the sequencer and data address generators allow computational operations to execute with maximum efficiency since the computation units can be devoted exclusively to processing data. With its instruction cache, the SHARC processors can simultaneously fetch an instruction from the cache and access two data operands from memory. The DAGs also provide built-in support for zero-overhead circular buffering.

Program sequencer. The program sequencer supplies instruction addresses to program memory. It controls loop iterations and evaluates conditional instructions. With an internal loop counter and loop stack, the processors execute looped code with zero overhead. No explicit jump instructions are
required to loop or to decrement and test the counter. To achieve a high execution rate while maintaining a simple programming model, the processor employs a five stage pipeline to process instructions — fetch1, fetch2, decode, address and execute. For more information, see “Instruction Pipeline” on page 4-5.

Data address generators. The DAGs provide memory addresses when data is transferred between memory and registers. Dual data address generators enable the processor to output simultaneous addresses for two operand reads or writes. DAG1 supplies 32-bit addresses for accesses using the DM bus. DAG2 supplies 32-bit addresses for memory accesses over the PM bus.

Each DAG keeps track of up to eight address pointers, eight address modifiers, and for circular buffering eight base-address registers and eight buffer-length registers. A pointer used for indirect addressing can be modified by a value in a specified register, either before (pre-modify) or after (post-modify) the access. A length value may be associated with each pointer to perform automatic modulo addressing for circular data buffers. The circular buffers can be located at arbitrary boundaries in memory. Each DAG register has a secondary register that can be activated for fast context switching.

Circular buffers allow efficient implementation of delay lines and other data structures required in digital signal processing. They are also commonly used in digital filters and Fourier transforms. The DAGs automatically handle address pointer wraparound, reducing overhead, increasing performance, and simplifying implementation.

Interrupts. The processors have three external hardware interrupts and a special interrupt for reset. The processor has internally-generated interrupts for the timer, circular buffer overflow, stack overflows, arithmetic exceptions, and user-defined software interrupts and different levels for emulation support.
Architectural Overview

For the external hardware and the internal timer interrupt, the processor automatically stacks the arithmetic status ($\text{ASTAT}_x$) register and mode ($\text{MODE}_1$) registers in parallel with the interrupt servicing, allowing 15 nesting levels of very fast service for these interrupts. Moreover, up to 19 programmable interrupts allow programs to change the interrupt priorities among the different peripheral DMA channels.

**Context switch.** Many of the processor’s registers have secondary registers that can be activated during interrupt servicing for a fast context switch. The data registers in the register file, the DAG registers, and the multiplier result register all have secondary registers. The primary registers are active at reset, while the secondary registers are activated by control bits in a mode control register.

**Timer.** The core’s programmable interval timer provides periodic interrupt generation. When enabled, the timer decrements a 32-bit count register every cycle. When this count register reaches zero, the processors generate an interrupt and asserts their timer expired output. The count register is automatically reloaded from a 32-bit period register and the countdown resumes immediately.

**Instruction cache.** The program sequencer includes a 32-word instruction cache that effectively provides three-bus operation for fetching an instruction and two data values. The cache is selective; only instructions whose fetches conflict with data accesses using the PM bus are cached. This caching allows full speed execution of core, looped operations such as digital filter multiply-accumulates, and FFT butterfly processing. For more information on the cache, refer to “Operating Modes” on page 4-87.

**Data bus exchange.** The data bus exchange ($\text{PX}$) register permits data to be passed between the 64-bit PM data bus and the 64-bit DM data bus, or between the 40-bit register file and the PM/DM data bus. These registers contain hardware to handle the data width difference. For more information, see “Register Files” on page 2-1.
Introduction

JTAG Port

The JTAG port supports the IEEE standard 1149.1 Joint Test Action Group (JTAG) standard for system test. This standard defines a method for serially scanning the I/O status of each component in a system. Emulators use the JTAG port to monitor and control the processor during emulation. Emulators using this port provide full speed emulation with access to inspect and modify memory, registers, and processor stacks. JTAG-based emulation is non-intrusive and does not effect target system loading or timing.

Core Buses

The processor core has two buses—PM data and DM data. The PM bus is used to fetch instructions from memory, but may also be used to fetch data. The DM bus can only be used to fetch data from memory. In conjunction with the cache, this Super Harvard Architecture allows the core to fetch an instruction and two pieces of data in the same cycle that a data word is moved between memory and a peripheral. This architecture allows dual data fetches, when the instruction is supplied by the cache.

I/O Buses

The I/O buses are used solely by the IOP to facilitate DMA transfers. These buses give the I/O processor access to internal memory for DMA without delaying the processor core (in the absence of memory block conflicts). One of the I/O buses is used for all peripherals (SPORT, SPI, IDP, UART, TWI etc.) while the second I/O bus is only used for the external port. The address bus is 19 bits wide, and both I/O data buses are 32 bits wide.

Bus capacities. The PM and DM address buses are both 32 bits wide, while the PM and DM data buses are both 64 bits wide.
These two buses provide a path for the contents of any register in the processor to be transferred to any other register or to any data memory location in a single cycle. When fetching data over the PM or DM bus, the address comes from one of two sources: an absolute value specified in the instruction (direct addressing) or the output of a data address generator (indirect addressing). These two buses share the same port of the memory. Each of the four memory blocks can be accessed by any of the two dedicated core and I/O buses assuming the accesses are conflict free.

**Data transfers.** Nearly every register in the processor core is classified as a universal register (Ureg). Instructions allow the transfer of data between any two universal registers or between a universal register and memory. This support includes transfers between control registers, status registers, and data registers in the register file. The bus connect (PX) registers permit data to be passed between the 64-bit PM data bus and the 64-bit DM data bus, or between the 40-bit register file and the PM/DM data bus. These registers contain hardware to handle the data width difference. For more information, see “Processing Element Registers” on page A-14.

**Development Tools**

The SHARC processors are supported by VisualDSP++, an easy to use Integrated Development and Debugging Environment (IDDE). VisualDSP++ allows you to manage projects from start to finish from within a single, integrated interface. Because the project development and debug environments are integrated, you can move easily between editing, building, and debugging activities.
Differences From Previous SHARC Processors

This section identifies differences between the current generation processors and previous SHARC processors: ADSP-2126x/2116x and ADSP-2106x. Like the ADSP-2116x family, the current generation is based on the original ADSP-2106x SHARC family. The current products preserve much of the ADSP-2106x architecture and is code compatible to the ADSP-2116x, while extending performance and functionality. For background information on SHARC and the ADSP-2106x Family processors, see the ADSP-2106x SHARC User’s Manual.

Table 1-1 shows the high level differences between the SHARC families.

Table 1-1. Differences Between SHARC Core Generations

<table>
<thead>
<tr>
<th>Feature</th>
<th>ADSP-2106x</th>
<th>ADSP-2116x/ADSP-2126x</th>
<th>ADSP-2136x/ADSP-2137x</th>
<th>ADSP-214xx</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMD Mode</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ISA/VISA</td>
<td>Yes/No</td>
<td>Yes/No</td>
<td>Yes/No</td>
<td>Yes/Yes</td>
</tr>
<tr>
<td>Broadcast Mode</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DAG1 (Addr/Data-bits)</td>
<td>32/40</td>
<td>32/64</td>
<td>32/64</td>
<td>32/64</td>
</tr>
<tr>
<td>DAG2 (Addr/Data-bits)</td>
<td>24/48</td>
<td>32/64</td>
<td>32/64</td>
<td>32/64</td>
</tr>
<tr>
<td>PX Register (PX1/PX2)</td>
<td>48-bit 16/32</td>
<td>64-bit 32/32</td>
<td>64-bit 32/32</td>
<td>64-bit 32/32</td>
</tr>
<tr>
<td>GPIO Flags</td>
<td>4</td>
<td>11</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>Programmable Interrupt Priorities</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Instruction Pipeline</td>
<td>3 Stages</td>
<td>3 Stages</td>
<td>5 Stages</td>
<td>5 Stages</td>
</tr>
<tr>
<td>Interrupt Mode Mask</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
## Differences From Previous SHARC Processors

Table 1-1. Differences Between SHARC Core Generations (Cont’d)

<table>
<thead>
<tr>
<th>Feature</th>
<th>ADSP-2106x</th>
<th>ADSP-2116x/ADSP-2126x</th>
<th>ADSP-2136x/ADSP-2137x</th>
<th>ADSP-214xx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Ports Per Block</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Internal Memory Ports</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Internal ROM</td>
<td>No</td>
<td>2116x: No</td>
<td>2126x: Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Data Sizes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>64-bit (LW)</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>48-bit (NW)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>40-bit (NW)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>32-bit (NW)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>16-bit (SW)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Conflict Cache</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>(Internal Memory)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction Cache</td>
<td>No</td>
<td>2116x: Conflict only</td>
<td>2136x: Conflict only</td>
<td>Yes</td>
</tr>
<tr>
<td>(External Memory)</td>
<td></td>
<td>2126x: No</td>
<td>2137x: Yes</td>
<td></td>
</tr>
<tr>
<td>I/O Buses</td>
<td>18/48</td>
<td>2116x: 18/64</td>
<td>2136-6: 1x19/32</td>
<td>2x19/32</td>
</tr>
<tr>
<td>(Addr/Data-bits)</td>
<td></td>
<td>2126x: 19/32</td>
<td>2136-7-9: 2x19/32</td>
<td></td>
</tr>
<tr>
<td>Emulation Background telemetry channel</td>
<td>No</td>
<td>2116x: No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Emulation User Break-point</td>
<td>No</td>
<td>2116x: No</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
2 REGISTER FILES

The SHARC core is controlled by non memory-mapped registers which are used for computation, data move or bit manipulation techniques and temporary data storage.

Features

- The non memory-mapped registers are called universal registers and can be used by almost all instructions
- Data registers are used for computation units
- Complementary data registers are used for the complementary computation units
- System registers are used for bit manipulation

Functional Description

The following sections provide a functional description of the register files.
Core Register Classification

The core architecture has three register categories:

- Data registers (PEx unit) and complementary data register (PEy unit)
- System registers (bit manipulation units)
- Universal registers (almost all core registers)

Most registers are universal registers; the data and system registers are subgroups of universal registers. This chapter describes access handling for these registers. For register coding details, see Chapter 9, Instruction Set Types.

Register Types Overview

Table 2-1 and Table 2-2 list the SHARC core registers. The registers in Table 2-1 are in the core processor.

Table 2-1. Universal Registers (Ureg)

<table>
<thead>
<tr>
<th>Register Type</th>
<th>Register(s)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dreg</td>
<td>R0 – R15</td>
<td>Processing element X register file locations, fixed-point</td>
</tr>
<tr>
<td></td>
<td>F0 – F15</td>
<td>Processing element X register file locations, floating-point</td>
</tr>
<tr>
<td>cdreg</td>
<td>S0 – S15</td>
<td>Processing element Y register file locations, fixed-point</td>
</tr>
<tr>
<td></td>
<td>SF0 – SF15</td>
<td>Processing element Y register file locations, floating-point</td>
</tr>
</tbody>
</table>
### Register Files

#### Program Sequencer
- **PC**: Program counter (read-only)
- **PCSTK**: Top of PC stack
- **PCSTKP**: PC stack pointer
- **FADDR**: Fetch address (read-only)
- **DADDR**: Decode address (read-only)
- **LADDR**: Loop termination address, code; top of loop address stack
- **CURLCNTR**: Current loop counter; top of loop count stack
- **LCNTR**: Loop count for next nested counter-controlled loop

#### Data Address Generators
- **I0 – I7**: DAG1 index registers
- **M0 – M7**: DAG1 modify registers
- **L0 – L7**: DAG1 length registers
- **B0 – B7**: DAG1 base registers
- **I8 – I15**: DAG2 index registers
- **M8 – M15**: DAG2 modify registers
- **L8 – L15**: DAG2 length registers
- **B8 – B15**: DAG2 base registers

#### Bus Exchange
- **PX**: 64-bit combination of PX1 and PX2
- **PX1**: PMD-DMD bus exchange 1 (32 bits)
- **PX2**: PMD-DMD bus exchange 2 (32 bits)

#### Timer
- **TPERIOD**: Timer period
- **TCOUNT**: Timer counter

<table>
<thead>
<tr>
<th>Register Type</th>
<th>Register(s)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>Data Address Generators</td>
<td>I0 – I7</td>
<td>DAG1 index registers</td>
</tr>
<tr>
<td></td>
<td>M0 – M7</td>
<td>DAG1 modify registers</td>
</tr>
<tr>
<td></td>
<td>L0 – L7</td>
<td>DAG1 length registers</td>
</tr>
<tr>
<td></td>
<td>B0 – B7</td>
<td>DAG1 base registers</td>
</tr>
<tr>
<td></td>
<td>I8 – I15</td>
<td>DAG2 index registers</td>
</tr>
<tr>
<td></td>
<td>M8 – M15</td>
<td>DAG2 modify registers</td>
</tr>
<tr>
<td></td>
<td>L8 – L15</td>
<td>DAG2 length registers</td>
</tr>
<tr>
<td></td>
<td>B8 – B15</td>
<td>DAG2 base registers</td>
</tr>
<tr>
<td></td>
<td>PX</td>
<td>64-bit combination of PX1 and PX2</td>
</tr>
<tr>
<td></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>TPERIOD</td>
<td>Timer period</td>
</tr>
<tr>
<td></td>
<td>TCOUNT</td>
<td>Timer counter</td>
</tr>
</tbody>
</table>
Table 2-1. Universal Registers (Ureg) (Cont’d)

<table>
<thead>
<tr>
<th>Register Type</th>
<th>Register(s)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>sreg</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>STKYx</td>
<td>Element x sticky arithmetic status flags, stack status flags, and so on.</td>
</tr>
<tr>
<td></td>
<td>USTAT1</td>
<td>User status register 1</td>
</tr>
<tr>
<td></td>
<td>USTAT3</td>
<td>User status register 3</td>
</tr>
<tr>
<td>csreg</td>
<td>ASTATy</td>
<td>Element y arithmetic status flags, bit test flag, etc.</td>
</tr>
<tr>
<td></td>
<td>STKYy</td>
<td>Element y sticky arithmetic status flags, stack status flags, and so on.</td>
</tr>
<tr>
<td></td>
<td>USTAT2</td>
<td>User status register 2</td>
</tr>
<tr>
<td></td>
<td>USTAT4</td>
<td>User status register 4</td>
</tr>
</tbody>
</table>

Table 2-2. Multiplier Registers

<table>
<thead>
<tr>
<th>Register Type</th>
<th>Register(s)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier Registers (no ureg 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>
Data Registers

Each of the processor’s processing elements has a data register file, which is a set of data registers that transfers data between the data buses and the computational units. These registers also provide local storage for operands and results.

The two register files consist of 16 primary registers and 16 alternate (secondary) registers. The data registers are 40 bits wide. Within these registers, 32-bit data is left-justified. If an operation specifies a 32-bit data transfer to these 40-bit registers, the eight LSBs are ignored on register reads, and the LSBs are cleared to zeros on writes.

Program memory data accesses and data memory accesses to and from the register file(s) occur on the PM data (PMD) bus and DM data (DMD) bus, respectively. One PMD bus access for each processing element and/or one DMD bus access for each processing element can occur in one cycle. Transfers between the register files and the DMD or PMD buses can move up to 64 bits of valid data on each bus.

Note that 16 data registers are sufficient to store the intermediate result of a FFT radix-4 butterfly stage.

Complementary Data Registers

The computational units (ALU, multiplier, and shifter) in PEx and PEy processing elements are identical. The data bus connections for the dual computational units permit asymmetric data moves to, from, and between the two processing elements. Identical instructions execute on the PEx and PEy units; the difference is the data. The data registers for PEy operations are identified (implicitly) from the PEx registers in the instruction. This implicit relationship between PEx and PEy data registers corresponds to the complementary register pairs in Table 2-3.
Data moves to the complementary data registers also occur in SISD mode. For PEy computations SIMD mode is required.

### Table 2-3. SIMD Mode Complementary RF Pairs

<table>
<thead>
<tr>
<th>PEx</th>
<th>PEy</th>
<th>PEx</th>
<th>PEy</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>S0</td>
<td>R8</td>
<td>S8</td>
</tr>
<tr>
<td>R1</td>
<td>S1</td>
<td>R9</td>
<td>S9</td>
</tr>
<tr>
<td>R2</td>
<td>S2</td>
<td>R10</td>
<td>S10</td>
</tr>
<tr>
<td>R3</td>
<td>S3</td>
<td>R11</td>
<td>S11</td>
</tr>
<tr>
<td>R4</td>
<td>S4</td>
<td>R12</td>
<td>S12</td>
</tr>
<tr>
<td>R5</td>
<td>S5</td>
<td>R13</td>
<td>S13</td>
</tr>
<tr>
<td>R6</td>
<td>S6</td>
<td>R14</td>
<td>S14</td>
</tr>
<tr>
<td>R7</td>
<td>S7</td>
<td>R15</td>
<td>S15</td>
</tr>
</tbody>
</table>

1. For Fixed-point operations, the prefixes are Rx (PEx) or Sx (PEy). For floating-point operations, the prefixes are Fx (PEx) or SFx (PEy).

### Data and Complementary Data Register Access Priorities

If writes to the same location take place in the same cycle, only the write with higher precedence actually occurs. The processor determines precedence for the write operation from the source of the data; from highest to lowest, the precedence is:

1. DAG1 or universal register (UREG)
2. DAG2
3. PEx ALU
4. PEy ALU
5. PEx Multiplier
6. PEy Multiplier
7. PEx Shifter
8. PEy Shifter

Example:
\[ r0 = r1 + r2, \quad r0 = dm(i0,m0), \quad r0 = pm(i8,m8); \quad /* r0 is loaded from i0 */ \]
\[ r0 = r1 + r2, \quad r0 = pm(i8,m8); \quad /* r0 is loaded from i8 */ \]

**Data and Complementary Data Register Transfers**

These 10-port, 16-register register files, combined with the enhanced Harvard architecture, allow unconstrained data flow between computation units and internal memory.

To support SIMD operation, the elements support a variety of dual data move features. The dual processing elements execute the same instruction, but operate on different data.

**Data and Complementary Data Register Swaps**

Registers swaps use the special swap operator, <->. A register-to-register swap occurs when registers in different processing elements exchange values; for example \( R0 <-> S1 \). Only single, 40-bit register-to-register swaps are supported. Double register operations are not supported as shown in the example below.

\[ R7 <-> S7; \]
\[ R2 <-> S0; \]

Regardless of SIMD/SISD mode, the processor supports bidirectional register-to-register swaps. The swap occurs between one register in each processing element’s data register file.
Functional Description

Note that the processor supports unidirectional and bidirectional register-to-register transfers with the Conditional Compute and Move instruction. For more information, see Chapter 4, Program Sequencer.

System Register Bit Manipulation

The system registers (SREG) support fast bit manipulation. The next example uses the shifter for bit manipulations:

R1 = MODE1;
R1 = BSET R1 by 21; /* sets PEYEN bit */
R1 = BSET R1 by 24; /* sets CBUFEN bit */
MODE1 = R1;

However the following example is more efficient.

BIT SET MODE1 PEYEN|CBUFEN; /* change both modes */
Nop; /* effect latency */

To set or test individual bits in a control register using the shifter:

R1 = dm(SYSCTL);
R1 = BSET R1 by 11; /* sets IMDW2 bit 11 */
R1 = BSET R1 by 12; /* sets IMDW3 bit 12 */
dm(SYSCTL) = R1;
BTST R1 by 11; /* clears SZ bit */
IF SZ jump func;
BTST R1 by 12; /* clears SZ bit */
IF SZ jump func;

The core has four user status registers (USTAT4-1) also classified as system registers but for general-purpose use. These registers allow flexible manipulation/testing of single or multiple individual bits in a register without affecting neighbor bits as shown in the following example.
USTAT1= dm(SYSCTL);
BIT SET USTAT1 IMDW2|IMDW3; /* sets bits 12-11 */
dm(SYSCTL)=USTAT1;
USTAT1= dm(SYSCTL);
BIT TST USTAT1 IMDW2|IMDW3; /* test bits 12-11 */
IF TF r15=r15+1; /* BTF = 1 PEx OR PEy */

Combined Data Bus Exchange Register

The two 64-bit data DMD and PMD buses allow programs to transfer the contents of any register in the processor to any other register or to any internal memory location in a single cycle. As shown in Figure 2-1, the bus exchange (PX) register permits data to flow between the PMD and DMD buses.

The PX register can work as one combined 64-bit register or as two 32-bit registers (PX1 and PX2).

The USTAT4-1 and PX2-1 registers allow load and store operations from memory. However, direct computations using universal registers is not supported and therefore a data move to the data register is required.

PX = DM(0x98000)(LW); /* read from DMD bus */
PM(0x4C000) = PX; /* write to PMD bus */

Figure 2-1. Bus Exchange (PX, PX1, and PX2) Registers
The alignment of PX1 and PX2 within PX appears in Figure 2-2. The combined PX register is an universal register (UREG) that is accessible for register-to-register or memory-to-register transfers.

**PX to DREG Transfers**

The PX register to data register transfers are either 40-bit transfers for the combined PX or 32-bit transfers for PX1 or PX2. Figure 2-2 shows the bit alignment and gives an example of instructions for register-to-register transfers. shows that during a transfer between PX1 or PX2 and a data register (Dreg), the bus transfers the upper 32 bits of the register file and zero-fills the eight least significant bits (LSBs). During a transfer between the combined PX register and a register file, the bus transfers the upper 40 bits of PX and zero-fills the lower 24 bits.

All transfers between the PX register (or any other internal register or memory) and any I/O processor register are 32-bit transfers (least significant 32 bits of PX). All transfers between the PX register and DREG/CDREG (R0–R15 or S0–S15) are 40-bit transfers. The most significant 40 bits are transferred as shown in Figure 2-2.

\[
\begin{align*}
R3 &= PX; & R3 &= PX1; \text{ or } R3 &= PX2; \\
\text{Register File Transfer} & & \text{Register File Transfer} \\
\begin{array}{c}
R3 \\
\hline
39 & 40 \text{ bits} & 0 \\
\end{array} & & \begin{array}{c}
R3 \\
\hline
39 & 32 \text{ bits} & 0x0 \\
\end{array} \\
\begin{array}{c}
PX \\
\hline
63 & 40 \text{ bits} & 24 & 23 & 0 \\
\end{array} & & \begin{array}{c}
32 \text{ bits} \\
\hline
31 & 8 & 7 & 0 \\
\end{array} \\
\begin{array}{c}
\text{Combined PX} \\
\hline
\text{PX2} & \text{PX1} \\
0 & 0x0 \\
\end{array} & & \begin{array}{c}
\text{PX1 or PX2} \\
\hline
\text{PX1 or PX2} \\
0 & 0x0 \\
\end{array}
\end{align*}
\]

Figure 2-2. PX to DREG Transfers
Immediate 40-bit Data Register Load

Extended precision data can’t be load immediately by using the following code.

R0 = 0x123456789A; /* asm error data field max 32-bits*/

The next example is an alternative which requires a combined PX1/PX2 register alignment for immediate load in SISD mode:

Bit CLR MODE1 PEYEN;
NOP;
PX2 = 0x55555555;     /* load data 39-8*/
PX1 = 0x9A000000;    /* load data 7-0*/
R1 = PX;             /* R1 load with 40-bit*/

PX to Memory Transfers

The PX register-to-internal memory transfers over the DMD or PMD bus are either 48-bit transfers for the combined PX or 32-bit transfers (on bits 31-0 of the bus) for PX1 or PX2. Figure 2-3 shows these transfers.

Figure 2-3. PX, PX1, PX2 Register-to-Memory Transfers on DM or PM Data Bus
Figure 2-3 also shows that during a transfer between \( PX_1 \) or \( PX_2 \) and internal memory, the bus transfers the lower 32 bits of the register. During a transfer between the combined \( PX \) register and internal memory, the bus transfers the upper 48 bits of \( PX \) and zero-fills the lower 16 bits.

**PX to Memory LW Transfers**

Figure 2-4 shows the transfer size between \( PX \) and internal memory over the PMD or DMD bus when using the long word (LW) option.

The LW notation in Figure 2-4 shows an important feature of \( PX \) register-to-internal memory transfers over the PM or DM data bus for the combined \( PX \) register. The \( PX \) register transfers to memory are 48-bit (three column) transfers on bits 63-16 of the PM or DM data bus, unless a long word transfer is used, or the transfer is forced to be 64-bit (four column) with the LW (long word) mnemonic.

The LW mnemonic affects data accesses that use the NW (normal word) addresses irrespective of the settings of the \( PEYEN \) (processor element Y enable) and \( IMDWx \) (internal memory data width) bits.

\[
PX = PM (0xB8000)(LW);
\]

**Figure 2-4. PX Register-to-Memory Transfers on PM Data Bus (LW)**
Uncomplimentary UREG to Memory LW Transfers

If a register without a complimentary register (such as the PC or LCNTR registers), or if immediate data is a source for a transfer to a long word memory location, the 32 bit source data is replicated within the long word. This is shown in the example below where the long word location 0x4F800 is written with the 64-bit data abbaabba_abbaabba. This is the case for all registers without peers.

\[
\begin{align*}
\text{I0} &= 0x4F800; \\
\text{MO} &= 0x1; \\
\text{L0} &= 0x0; \\
\text{DM(I0,MO)} &= 0xabbaabba;
\end{align*}
\]

Long word accesses using the USTATx registers is shown below.

\[
\begin{align*}
\text{USTAT1} &= \text{DM (LW address)}; & \text{/* Loads only USTAT1 in SISD mode */} \\
\text{DM (LW address)} &= \text{USTAT1}; & \text{/* Stores both USTAT1 and USTAT2 */}
\end{align*}
\]

Operating Modes

The following sections detail the operation of the register files.

Alternate (Secondary) Data Registers

Each data register file has an alternate data register set. To facilitate fast context switching, the processor includes alternate register sets for data, results, and data address generator registers. Bits in the MODE1 register control when alternate registers become accessible. While inaccessible, the contents of alternate registers are not affected by processor operations.
Operating Modes

Note that there is a one cycle latency from the time when writes are made to the MODE1 register until an alternate register set can be accessed.

The alternate register sets for data and results are shown in Figure 2-5. For more information on alternate data address generator registers, see “Alternate (Secondary) DAG Registers” on page 6-28. Bits in the MODE1 register can activate independent alternate data register sets: the lower half (R0–R7) and the upper half (R8–R15). To share data between contexts, a program places the data to be shared in one half of either the current processing element’s register file or the opposite processing element’s register file and activates the alternate register set of the other half. For information on how to activate alternate data registers, see the description of the MODE1 register below. The register files consist of a primary set of 16 x 40-bit registers and an alternate set of 16 x 40-bit registers.

Alternate (Secondary) Data Registers SIMD Mode

Context switching between the two sets of data registers (SIMD mode) occurs in parallel between the two processing elements. Figure 2-5 shows the lower half (S0–S7) and the upper half (S8–S15) of the data register file.
Figure 2-5. Alternate (Secondary) Data Register File
Operating Modes

UREG/SREG SIMD Mode Transfers

Table 2-4 shows the user status and PX registers and their complementary registers.

Table 2-4. Complementary Register Pairs

<table>
<thead>
<tr>
<th>USTAT1</th>
<th>USTAT2</th>
</tr>
</thead>
<tbody>
<tr>
<td>USTAT3</td>
<td>USTAT4</td>
</tr>
<tr>
<td>PX1</td>
<td>PX2</td>
</tr>
</tbody>
</table>

There is no implicit move when the combined PX register is used in SIMD mode. For example, in SIMD mode, the following moves occur:

\[
\begin{align*}
    \text{PX1} &= R0; \quad /* R0 32-bit explicit move to PX1, \\
                & \quad \text{and S0 32-bit implicit move to PX2 */} \\
    \text{PX} &= R0; \quad /* R0 40-bit explicit move to PX, \\
                & \quad \text{but no implicit move for S0 */} \\
\end{align*}
\]

However, the following exceptions should be noted:

- Transfers between USTATx and PX registers as in the following example and Figure 2-6. Note that all user status registers behave in this manner.

\[
\begin{align*}
    \text{PX} &= \text{USTAT1}; \quad /* \text{loads PX1 with USTAT1 and PX2 with USTAT2 */} \\
    \text{USTAT1} &= \text{PX}; \quad /* \text{loads only PX1 to USTAT1 */} \\
\end{align*}
\]
• Transfers between DAG and other system registers and the $PX$ register as shown in the following example:

```c
I0 = PX;    /* Moves PX1 to I0 */
PX = I0;    /* Loads both PX1 and PX2 with I0 */
LCNTR = PX; /* Loads LCNTR with PX1 */
PX = PC;    /* Loads both PX1 and PX2 with PC */
```

**Interrupt Mode Mask**

On the SHARC processors, programs can mask automated individual operating modes bits of the $MODE1$ register by entering into an ISR. This reduces latency cycles.

For the data registers the alternate registers ($SRRFH/L$) are optional masks in use. For more information, see Chapter 4, Program Sequencer.
Operating Modes
3 PROCESSING ELEMENTS

The PEx and PEy processing elements perform numeric processing for processor algorithms. Each element contains a data register file and three computation units—an arithmetic/logic unit (ALU), a multiplier, and a barrel shifter. Computational instructions for these elements include both fixed-point and floating-point operations, and each computational instruction executes in a single cycle.

Features

The processing elements have the following features.

- **Data Formats.** The units support 32-bit fixed and floating point single precision IEEE 32-bit and extended precision IEEE 40-bit.
- **Arithmetic/logic unit.** The ALU performs arithmetic and logic operations on fixed-point and floating-point data.
- **Multiplier.** The multiplier performs floating-point and fixed-point multiplication and executes fixed-point multiply/add and multiply/subtract operations.
- **Barrel Shifter.** The barrel shifter performs bit shifts, bit, bit field, and bit stream manipulation on 32-bit operands. The shifter can also derive exponents.
Functional Description

- **Multifunction.** The ALU and Multiplier support simultaneous operations for fixed- and floating-point data formats. The fixed-point multiplier can return results as 32 or 80 bits.

- **One Cycle Arithmetic Pipeline.** All computation instructions execute in one cycle.

Functional Description

The computational units in a processing element handle different types of operations.

Data flow paths through the computation units are arranged in parallel, as shown in Figure 3-1. The output of any computation unit may serve as the input of any computation unit on the next instruction cycle. Data moving in and out of the computation units goes through a 10-port register file, consisting of 16 primary and 16 alternate registers. Two ports on the register file connect to the PM and DM data buses, allowing data transfers between the computation units and memory (and anything else) connected to these buses.

Single Cycle Processing

Based on the 5-stage pipeline in the SHARC processor core, the operands are fetched during the second half of the address phase of pipeline before the results are written back in the first half of the execution phase of pipeline. Therefore, the ALU, multiplier and shifter can read and write the same register file location in an instruction cycle. For more information, see Chapter 4, Program Sequencer.

Processing Data from Memory Loads

Almost all processing operations require data streams from the internal memory. However since memory data load takes 2 cycles to terminate
(data stored in the data register) data forwarding is used to improve throughput. The data path already forwarded to the data register is directly feed into the computation unit to be processed in the next stage. The data register is updated afterwards.

**Data Format for Computation Units**

The processor’s assembly language provides access to the data register files in both processing elements. The syntax allows programs to move data to and from these registers, specify a computation’s data format and provide naming conventions for the registers, all at the same time. For information on the data register names, see Chapter 2, Register Files.

Note the register name(s) within the multiplier instruction specify input data type(s)—Fx for floating-point and Rx for fixed-point.

The computation input format is not an operating mode, it is based on the instruction prefix.

**Arithmetic Logic Unit (ALU)**

The ALU performs arithmetic operations on fixed-point or floating-point data and logical operations on fixed-point data. ALU fixed-point instructions operate on 32-bit fixed-point operands and output 32-bit fixed-point results, and ALU floating-point instructions operate on 32-bit or 40-bit floating-point operands and output 32-bit or 40-bit floating-point results. ALU instructions include:

- Floating-point addition, subtraction, add/subtract, average
- Fixed-point addition, subtraction, add/subtract, average
- Floating-point manipulation – binary log, scale, mantissa
- Fixed-point add with carry, subtract with borrow, increment, decrement
Functional Description

ALU instructions take one or two inputs: X input and Y input. These inputs (known as operands) can be any data registers in the register file. Most ALU operations return one result. However, in add/subtract operations, the ALU operation returns two results and in compare operations the ALU returns no result (only flags are updated). ALU results can be returned to any location in the register file.

Figure 3-1. Computational Block

- Logical AND, OR, XOR, NOT
- Functions – ABS, PASS, MIN, MAX, CLIP, COMPARE
- Format conversion
- Floating-point iterative reciprocal and reciprocal square root functions
If the ALU operation is fixed-point, the inputs are treated as 32-bit fixed-point operands. The ALU transfers the upper 32 bits from the source location in the register file. For fixed-point operations, the result(s) are 32-bit fixed-point values. Some floating-point operations (\texttt{LOGB, MANT} and \texttt{FIX}) can also yield fixed-point results.

The processor transfers fixed-point results to the upper 32 bits of the data register and clears the lower eight bits of the register. The format of fixed-point operands and results depends on the operation. In most arithmetic operations, there is no need to distinguish between integer and fractional formats. Fixed-point inputs to operations such as scaling a floating-point value are treated as integers. For purposes of determining status such as overflow, fixed-point arithmetic operands and results are treated as two's-complement numbers.

**ALU Instruction Types**

The following sections provide details about the instruction types supported by the ALU.

**Compare Accumulation Instruction**

Bits 31–24 in the \texttt{ASTATx/y} registers store the flag results of up to eight ALU compare operations. These bits form a right-shift register. When the processor executes an ALU compare operation, it shifts the eight bits toward the LSB (bit 24 is lost). Then it writes the MSB, bit 31, with the result of the compare operation. If the X operand is greater than the Y operand in the compare instruction, the processor sets bit 31. Otherwise, it clears bit 31.

Applications can use the accumulated compare flags to implement two- and three-dimensional clipping operations.
Functional Description

Fixed-to-Float Conversion Instructions

The ALU supports conversion between floating and fixed point as shown in the following example.

\[
\begin{align*}
Fn &= \text{FLOAT} \ Rx; \quad /* \text{floating-point} */ \\
Rn &= \text{FIX} \ Fx; \quad /* \text{fixed-point} */
\end{align*}
\]

Fixed-to-Float Conversion Instructions with Scaling

The ALU supports conversion between floating- and fixed-point by using a scaling factor as shown in the following example.

\[
\begin{align*}
Fn &= \text{FLOAT} \ Rx \text{ by 31}; \quad /* \text{floating-point [-1.0 to 1.0]} */ \\
Rn &= \text{FIX} \ Fx \text{ by 31} \quad /* \text{fixed-point 1.31 format} */
\end{align*}
\]

Reciprocal/Square Root Instructions

The reciprocal/square root floating-point instruction types do not execute in a single cycle. Iterative algorithms are used to compute both reciprocals and square roots. The \texttt{RECIPS} and \texttt{RSQRTS} operations are used to start these iterative algorithms as shown below.

\[
\begin{align*}
Fn &= \text{RECIPS} \ Fx; \quad /* \text{creates seed for reciprocal} */ \\
Fn &= \text{RSQRTS} \ Fx; \quad /* \text{creates seed for reciprocal square root} */
\end{align*}
\]

Divide Instruction

The SHARC processor does not support a floating-point divide instruction. The \texttt{RECIPS} instruction is used to simplify the divide implementation instruction by using an iterative convergence algorithm. For more information, see Chapter 9, Instruction Set Types.

ALU Instruction Summary

Table 3-1 and Table 3-2 list the ALU instructions and show how they relate to the \texttt{ASTATx/ASTATy} and \texttt{STKYx/STKYy} flags. For more information on assembly language syntax, see Chapter 9, Instruction Set Types, and
Chapter 10, Computation Instructions. In these tables, note the meaning of the following symbols:

- \( R_n, R_x, R_y \) indicate any register file location; treated as fixed-point
- \( F_n, F_x, F_y \) indicate any register file location; treated as floating-point
- * indicates that the flag may be set or cleared, depending on the results of instruction
- ** indicates that the flag may be set (but not cleared), depending on the results of the instruction
- – indicates no effect
- In SIMD mode all instructions in this table use the complement data registers

### Table 3-1. Fixed-Point ALU Instruction Summary

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ASTATx, ASTATy Status Flags</th>
<th>STKYx, STKYy Status Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-Point:</td>
<td>AZ</td>
<td>AV</td>
</tr>
<tr>
<td>( R_n = R_x + R_y )</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>( R_n = R_x - R_y )</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>( R_n = R_x + R_y + CI )</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>( R_n = R_x - R_y + CI - 1 )</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>( R_n = (R_x + R_y)/2 )</td>
<td>*</td>
<td>0</td>
</tr>
<tr>
<td>COMP(Rx, Ry)</td>
<td>*</td>
<td>0</td>
</tr>
<tr>
<td>COMPU(Rx, Ry)</td>
<td>*</td>
<td>0</td>
</tr>
<tr>
<td>( R_n = R_x + CI )</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>( R_n = R_x + CI - 1 )</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>( R_n = R_x + 1 )</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>( R_n = R_x - 1 )</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>
**Functional Description**

Table 3-1. Fixed-Point ALU Instruction Summary (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ASTATx, ASTATy Status Flags</th>
<th>STKYx, STKYy Status Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-Point:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rn = –Rx</td>
<td>*  *  *  * 0 0 0</td>
<td>–  –  –  * * –</td>
</tr>
<tr>
<td>Rn = ABS Rx</td>
<td>*  * 0 0 * 0 0</td>
<td>–  –  –  * * –</td>
</tr>
<tr>
<td>Rn = PASS Rx</td>
<td>*  0 * 0 0 0 0</td>
<td>–  –  –  –</td>
</tr>
<tr>
<td>Rn = Rx AND Ry</td>
<td>* 0 * 0 0 0 0</td>
<td>–  –  –  –</td>
</tr>
<tr>
<td>Rn = Rx OR Ry</td>
<td>* 0 * 0 0 0 0</td>
<td>–  –  –  –</td>
</tr>
<tr>
<td>Rn = Rx XOR Ry</td>
<td>* 0 * 0 0 0 0</td>
<td>–  –  –  –</td>
</tr>
<tr>
<td>Rn = NOT Rx</td>
<td>* 0 * 0 0 0 0</td>
<td>–  –  –  –</td>
</tr>
<tr>
<td>Rn = MIN(Rx, Ry)</td>
<td>* 0 * 0 0 0 0</td>
<td>–  –  –  –</td>
</tr>
<tr>
<td>Rn = MAX(Rx, Ry)</td>
<td>* 0 * 0 0 0 0</td>
<td>–  –  –  –</td>
</tr>
<tr>
<td>Rn = CLIP Rx BY Ry</td>
<td>* 0 * 0 0 0 0</td>
<td>–  –  –  –</td>
</tr>
</tbody>
</table>

Table 3-2. Floating-Point ALU Instruction Summary

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ASTATx, ASTATy Status Flags</th>
<th>STKYx, STKYy Status Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating-Point:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fn = Fx + Fy</td>
<td>*  *  * 0 0 * 1</td>
<td>–  **  **  –  **</td>
</tr>
<tr>
<td>Fn = Fx – Fy</td>
<td>*  *  * 0 0 * 1</td>
<td>–  **  **  –  **</td>
</tr>
<tr>
<td>Fn = ABS (Fx + Fy)</td>
<td>* 0 0 0 0 * 1</td>
<td>–  **  **  –  **</td>
</tr>
<tr>
<td>Fn = ABS (Fx – Fy)</td>
<td>* 0 0 0 0 * 1</td>
<td>–  **  **  –  **</td>
</tr>
<tr>
<td>Fn = (Fx + Fy)/2</td>
<td>* 0 * 0 0 * 1</td>
<td>–  **  –  –  **</td>
</tr>
<tr>
<td>COMP(Fx, Fy)</td>
<td>* 0 * 0 0 * 1</td>
<td>–  –  –  **  **</td>
</tr>
<tr>
<td>Fn = –Fx</td>
<td>*  *  * 0 0 * 1</td>
<td>–  –  **  –  **</td>
</tr>
<tr>
<td>Fn = ABS Fx</td>
<td>*  * 0 0 * 1</td>
<td>–  –  **  –  **</td>
</tr>
<tr>
<td>Fn = PASS Fx</td>
<td>*  0 * 0 0 * 1</td>
<td>–  –  –  –  **</td>
</tr>
</tbody>
</table>
Multiply Accumulator

The multiplier performs fixed-point or floating-point multiplication and fixed-point multiply/accumulate operations. Fixed-point multiply/accumulates are available with cumulative addition or cumulative subtraction. Multiplier floating-point instructions operate on 32-bit or 40-bit floating-point operands and output 32-bit or 40-bit floating-point results. Multiplier fixed-point instructions operate on 32-bit fixed-point data and produce 80-bit results. Inputs are treated as fractional or integer, unsigned or two’s-complement. Multiplier instructions include:

- Floating-point multiplication
Functional Description

- Fixed-point multiplication
- Fixed-point multiply/accumulate with addition, rounding optional
- Fixed-point multiply/accumulate with subtraction, rounding optional
- Rounding multiplier result register
- Saturating multiplier result register
- Clearing multiplier result register

Functional Description

The multiplier takes two inputs, X and Y. These inputs (also known as operands) can be any data registers in the register file. The multiplier can accumulate fixed-point results in the local multiplier result (MRF) registers or write results back to the register file. The results in MRF can also be rounded or saturated in separate operations. Floating-point multiplies yield floating-point results, which the multiplier writes directly to the register file.

For fixed-point multiplies, the multiplier reads the inputs from the upper 32 bits of the data registers. Fixed-point operands may be either both in integer format, or both in fractional format. The format of the result matches the format of the inputs. Each fixed-point operand may be either an unsigned number or a two’s-complement number. If both inputs are fractional and signed, the multiplier automatically shifts the result left one bit to remove the redundant sign bit.

Asymmetric Multiplier Inputs

In cases of dual operand forwarding from a compute instruction in the previous cycle, wherein both the X and Y inputs are required for multiplication, there is a one cycle stall. However, this is not a very common case in DSP processing, and therefore high architectural efficiency is still
achieved using an asymmetrical multiplier. For more information, see Chapter 4, Program Sequencer.

Multiplier Result Register

Fixed-point operations place 80-bit results in the multiplier’s foreground register (MRF) or background register (MRB), depending on which is active. For more information on selecting the result register, see “Alternate (Secondary) Data Registers” on page 2-13.

The location of a result in the MRF register’s 80-bit field depends on whether the result is in fractional or integer format, as shown in Figure 3-2. If the result is sent directly to a data register, the 32-bit result with the same format as the input data is transferred, using bits 63–32 for a fractional result or bits 31–0 for an integer result. The eight LSBs of the 40-bit register file location are zero-filled.

Fractional results can be rounded-to-nearest before being sent to the register file. If rounding is not specified, discarding bits 31–0 effectively truncates a fractional result (rounds to zero). For more information on rounding, see “Rounding Mode” on page 3-34.

The MRF register is comprised of the MR2F, MR1F, and MR0F registers, which individually can be read from or written to the register file. Each of these
**Functional Description**

registers has the same format. When data is read from \( MR2F \), it is sign-extended to 32 bits. The processor zero-fills the eight LSBs of the 40-bit register file location when data is read from/written to the \( MR2F \), \( MR1F \), or \( MRF \) register file. When the processor writes data into \( MR2F \), \( MR1F \), or \( MRF \) from the 32 MSBs of a register file location, the eight LSBs are ignored. Data written to \( MR1F \) register is sign-extended to \( MR2F \), repeating the MSB of \( MR1F \) in the 16 bits of the \( MR2F \) register. Data written to the \( MRF \) register is not sign-extended.

Note that the multiply result register (\( MRF, MRB \)) is not an orthogonal register in the instruction set. Only specific instructions decode it as an operand or as a result register (no universal register). For more information, see Chapter 9, Instruction Set Types.

**Multiply Register Instruction Types**

In addition to multiply, fixed-point operations include accumulate, round, and saturate fixed-point data. The three MRx register instructions are described in the following sections.

**Clear MRx Instruction**

The clear operation (\( MRF = 0 \)) resets the specified \( MRF \) register to zero. Often, it is best to perform this operation at the start of a multiply/accumulate operation to remove the results of the previous operation.

**Round MRx Instruction**

The \( RND \) operation (\( MRF = RND \ MRF \)) applies only to fractional results, integer results are not effected. This operation performs a round to nearest of the 80-bit \( MRF \) value at bit 32, for example, the \( MR1F - MRF \) boundary. Rounding a fixed-point result occurs as part of a multiply or multiply/accumulate operation or as an explicit operation on the \( MRF \) register. The rounded result in \( MR1F \) can be sent to the register file or back to the same \( MRF \) register. To round a fractional result to zero (truncation) instead of to
nearest, a program transfers the unrounded result from \( MR1F \), discarding the lower 32 bits in \( MROF \).

**Saturate MRx Instruction**

The \( \text{SAT} \) operation \( (MRF = \text{SAT} MRF) \) sets \( MRF \) to a maximum value if the \( MRF \) value has overflowed. Overflow occurs when the \( MRF \) value is greater than the maximum value for the data format—unsigned or two’s-complement and integer or fractional—as specified in the saturate instruction. The six possible maximum values appear in Table 3-3. The result from \( MRF \) saturation can be sent to the register file or back to the same \( MRF \) register.

Table 3-3. Fixed-Point Format Maximum Values (Saturation)

<table>
<thead>
<tr>
<th>Maximum Number</th>
<th>(Hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( MR2F )</td>
</tr>
<tr>
<td>Two’s-complement fractional (positive)</td>
<td>0000</td>
</tr>
<tr>
<td>Two’s-complement fractional (negative)</td>
<td>FFFF</td>
</tr>
<tr>
<td>Two’s-complement integer (positive)</td>
<td>0000</td>
</tr>
<tr>
<td>Two’s-complement integer (negative)</td>
<td>FFFF</td>
</tr>
<tr>
<td>Unsigned fractional number</td>
<td>0000</td>
</tr>
<tr>
<td>Unsigned integer number</td>
<td>0000</td>
</tr>
</tbody>
</table>

**Multiplier Instruction Summary**

Table 3-4 and Table 3-5 list the multiplier instructions and describe how they relate to the \( \text{ASTAT}x/\text{ASTAT}y \) and \( \text{STKY}x/\text{STKY}y \) flags. For more information on assembly language syntax, see Chapter 9, Instruction Set Types, and Chapter 10, Computation Instructions. In these tables, note the meaning of the following symbols:

- \( Rn, Rx, Ry \) indicate any register file location; treated as fixed-point
Functional Description

- \( F_n, F_x, F_y \) indicate any register file location; treated as floating-point
- * indicates that the flag may be set or cleared, depending on results of instruction
- ** indicates that the flag may be set (but not cleared), depending on results of instruction
- – indicates no effect
- The Input Mods column indicates the types of optional modifiers that can be applied to the instruction inputs. For a list of modifiers, see Table 3-6.
- In SIMD mode all instruction uses the complement data/multiply result registers.

Table 3-4. Fixed-Point Multiplier Instruction Summary

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Input Mods</th>
<th>ASTATx, ASTATy Flags</th>
<th>STKYx, STKYy Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MU MN MV MI</td>
<td>MUS MOS MVS MIS</td>
<td></td>
</tr>
<tr>
<td>( R_n = R_x \times R_y )</td>
<td>1</td>
<td>* * * 0</td>
<td>– ** – –</td>
</tr>
<tr>
<td>( MRF = R_x \times R_y )</td>
<td>1</td>
<td>* * * 0</td>
<td>– ** – –</td>
</tr>
<tr>
<td>( MRB = R_x \times R_y )</td>
<td>1</td>
<td>* * * 0</td>
<td>– ** – –</td>
</tr>
<tr>
<td>( R_n = MRF + R_x \times R_y )</td>
<td>1</td>
<td>* * * 0</td>
<td>– ** – –</td>
</tr>
<tr>
<td>( R_n = MRB + R_x \times R_y )</td>
<td>1</td>
<td>* * * 0</td>
<td>– ** – –</td>
</tr>
<tr>
<td>( MRF = MRF + R_x \times R_y )</td>
<td>1</td>
<td>* * * 0</td>
<td>– ** – –</td>
</tr>
<tr>
<td>( MRB = MRB + R_x \times R_y )</td>
<td>1</td>
<td>* * * 0</td>
<td>– ** – –</td>
</tr>
<tr>
<td>( R_n = MRF – R_x \times R_y )</td>
<td>1</td>
<td>* * * 0</td>
<td>– ** – –</td>
</tr>
<tr>
<td>( R_n = MRB – R_x \times R_y )</td>
<td>1</td>
<td>* * * 0</td>
<td>– ** – –</td>
</tr>
<tr>
<td>( MRF = MRF – R_x \times R_y )</td>
<td>1</td>
<td>* * * 0</td>
<td>– ** – –</td>
</tr>
<tr>
<td>( MRB = MRB – R_x \times R_y )</td>
<td>1</td>
<td>* * * 0</td>
<td>– ** – –</td>
</tr>
<tr>
<td>( R_n = SAT MRF )</td>
<td>2</td>
<td>* * * 0</td>
<td>– ** – –</td>
</tr>
<tr>
<td>( R_n = SAT MRB )</td>
<td>2</td>
<td>* * * 0</td>
<td>– ** – –</td>
</tr>
</tbody>
</table>
Table 3-4. Fixed-Point Multiplier Instruction Summary (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Input Mods</th>
<th>ASTATx, ASTATy Flags</th>
<th>STKYx, STKYy Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-Point</td>
<td></td>
<td>MU</td>
<td>MN</td>
</tr>
<tr>
<td>MRF = SAT MRF</td>
<td>2</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>MRB = SAT MRB</td>
<td>2</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Rn = RND MRF</td>
<td>3</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Rn = RND MRB</td>
<td>3</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>MRF = RND MRF</td>
<td>3</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>MRB = RND MRB</td>
<td>3</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>MRF = 0</td>
<td>–</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MRB = 0</td>
<td>–</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MRxF = Rn</td>
<td>–</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MRxB = Rn</td>
<td>–</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Rn = MRxF</td>
<td>–</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Rn = MRxB</td>
<td>–</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3-5. Floating-Point Multiplier Instruction Summary

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ASTATx, ASTATy Flags</th>
<th>STKYx, STKYy Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating-Point</td>
<td>MU</td>
<td>MN</td>
</tr>
<tr>
<td>Fx = Fx × Fy</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>
Functional Description

Table 3-6. Input Modifiers for Fixed-Point Multiplier Instruction

<table>
<thead>
<tr>
<th>Input Mods from Table 3-4</th>
<th>Input Mods—Options For Fixed-Point Multiplier Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Note the meaning of the following symbols in this table:</td>
</tr>
<tr>
<td></td>
<td>Signed input — S</td>
</tr>
<tr>
<td></td>
<td>Unsigned input — U</td>
</tr>
<tr>
<td></td>
<td>Integer input — I</td>
</tr>
<tr>
<td></td>
<td>Fractional input — F</td>
</tr>
<tr>
<td></td>
<td>Fractional inputs, Rounded output — FR</td>
</tr>
<tr>
<td></td>
<td>Note that (SF) is the default format for one-input operations, and (SSF) is the default format for two-input operations.</td>
</tr>
<tr>
<td>1</td>
<td>(SSF), (SSI), (SSFR), (SUF), (SUFR), (USF), (USI), (USFR), (UUF), (UUI), or (UUFR)</td>
</tr>
<tr>
<td>2</td>
<td>(SI), (SU), (UF), or (UI) saturation only</td>
</tr>
<tr>
<td>3</td>
<td>(SF) or (UF) rounding only</td>
</tr>
</tbody>
</table>

Barrel Shifter

The barrel shifter is a combination of logic with X inputs and Y outputs and control logic that specifies how to shift data between input and output within one cycle.

The shifter performs bit-wise operations on 32-bit fixed-point operands. Shifter operations include the following.

- Bit wise operations such as shifts and rotates from off-scale left to off-scale right
- Bit wise manipulation operations, including bit set, clear, toggle, and test
- Bit field manipulation operations, including extract and deposit
- Bit stream manipulation operations using a bit FIFO
- Bit field conversion operations including exponent extract, number of leading 1s or 0s
• Pack and unpack conversion between 16-bit and 32-bit floating-point

Functional Description

The shifter takes one to three inputs: X, Y, and Z. The inputs (known as operands) can be any register in the register file. Within a shifter instruction, the inputs serve as follows.

• The X input provides data that is operated on.
• The Y input specifies shift magnitudes, bit field lengths, or bit positions.
• The Z input provides data that is operated on and updated.

Shifter Instruction Types

There are two shifter instruction categories: shift compute or shift immediate instructions. Both instruction types operate identically. Only the Y input is either in an instruction or in a data register.

Shift Compute Category

The shift compute instruction uses a data register for the Y input. The data register operates based on the instruction’s 12-bit field for the bit position start (bit6) and the bit field length (len6). Other instructions may use only the 8-bit field.

Shift Immediate Category

The shift immediate instruction uses immediate data for the Y input. This input comes from the instruction’s 12-bit field for the bit position start (bit6) and the bit field length (len6). Other instructions may use only the 8-bit field.
Functional Description

Bit Manipulation Instructions

In the following example, $R_x$ is the X input, $R_y$ is the Y input, and $R_n$ is the Z input. The shifter returns one output ($R_n$) to the register file.

$$R_n = R_n \text{ OR LSHIFT } R_x \text{ BY } R_y;$$

As shown in Figure 3-3, the shifter fetches input operands from the upper 32 bits of a register file location (bits 39–8) or from an immediate value in the instruction.

The X input and Z input are always 32-bit fixed-point values. The Y input is a 32-bit fixed-point value or an 8-bit field ($\text{SHF8}$), positioned in the register file. These inputs appear in Figure 3-3.

Some shifter operations produce 8 or 6-bit results. As shown in Figure 3-3, the shifter places these results in the $\text{SHF8}$ field or the $\text{bit6}$ field and sign-extends the results to 32 bits. The shifter always returns a 32-bit result.

![Figure 3-3. Register File Fields for Shifter Instructions](image-url)
Bit Field Manipulation Instructions

The shifter supports bit field deposit and bit field extract instructions for manipulating groups of bits within an input. The Y input for bit field instructions specifies two 6-bit values, bit6 and len6, which are positioned in the Ry register as shown in Figure 3-4. The shifter interprets bit6 and len6 as positive integers. The bit6 value is the starting bit position for the deposit or extract, and the len6 value is the bit field length, which specifies how many bits are deposited or extracted.

![Figure 3-4. Register File Fields for FDEP, FEXT Instructions](image)

Field deposit (FDEP) instructions take a group of bits from the input register (starting at the LSB of the 32-bit integer field) and deposit the bits as directed anywhere within the result register. The bit6 value specifies the starting bit position for the deposit. Figure 3-5 shows how the inputs, bit6 and len6, work in the following field deposit instruction.

\[ Rn = \text{FDEP} \ Rx \ By \ Ry \]
Figure 3-6 shows bit placement for the following field deposit instruction.

\[ R_0 = \text{FDEP} \ R_1 \text{ By } R_2; \]

Field extract (FEXT) instructions extract a group of bits as directed from anywhere within the input register and place them in the result register, aligned with the LSB of the 32-bit integer field. The bit6 value specifies the starting bit position for the extract.
Figure 3-6. Bit Field Deposit Example

Figure 3-7 shows bit placement for the following field extract instruction.

R3 = FEXT R4 By R5;
The bit stream manipulation operations, in conjunction with the bit FIFO write pointer (BFFWRP) instruction, implement a bit FIFO used for modifying the bits in a contiguous bit stream. The shifter supports bit stream manipulation as shown in the following list.

- **BITDEP** deposits bits into a bit stream
- **BITEXT** extracts bits from a bit stream

The bit FIFO consists of a 64-bit register internal to the shifter and an associated write pointer register which keeps track of the number of valid bits in the FIFO. When the bit FIFO is empty, the write pointer is 0, when the FIFO is full, the write pointer is 64. The bit FIFO register and write pointer can be accessed only through the **BITDEP** and **BITEXT** instructions. For more information, see “Shifter Operations” on page 10-56.
The bit FIFO contains a status flag (shifter FIFO, SF) which reflects the current value of the write pointer – SF is set when the write pointer is greater than or equal to 32, it is cleared otherwise. Another status flag SV, indicates the exception condition such as overflow or underflow.

The SF flag has two related conditions – SF and NOT SF, which are for exclusive use in instructions involving the bit FIFO.

The shifter FIFO bit (SF in ASTATx/y registers) reflects the status flag. Note this bit is a read-only bit unlike other flags in the ASTATx/y registers. The value is pushed into the stack during a PUSH operation but a POP operation does not restore this ASTAT bit.

Listing 3-1 and Listing 3-2 demonstrate the BITDEP instruction where 32-bit words are appended to the bit FIFO whenever the total number of bits falls below 32. A variable number of bits are read.

Listing 3-1. Example of Header Extraction

I13 = buffer_base;
M13 = 1;
BFFWRP = 0x0; /* initialize Bit Fifo */
R10 = pm(I13,M13);
If NOT SF BITDEP R10 by 32, R10 = PM(I13,M13); /* appends R10 to BFF */
R6 = BITEXT (6); /* extracts 6 bits from head of BFF and left-shifts BFF by that amount */
DM(Var_1) = R6;
If NOT SF BITDEP R10 by 32, R10 = PM(I13,M13);
R6 = BITEXT(3); /* extracts 3 bits */
DM(Var_2) = R6;

The bit extracts are in variable quantities, but the deposit is always in 32-bits whenever the total number of bits in the bit FIFO increases beyond 32.
Functional Description

Listing 3-2. Header Creation

I13 = buffer_base;
M13 = 1;
BFFWRP=0x0;
R10 = dm(_var1); /* get the variable */
BITDEP R10 by 6; /* append it to BFF */
If SF R10 = BITEXT(32), pm(I,M) = R10; /* if the balance > 32,
transfer a word*/
R10 = dm(Var_1);
BITDEP R10 by 3;
If NOT SF R10 = BITEXT(32), pm(I,M) = R10;

Interrupts Using Bit FIFO Instructions

If the program vectors to an ISR during bit FIFO operations, and the ISR uses the bit FIFO for different other purposes, then the state of the bit FIFO has to be preserved if the program needs to restart the previous bit FIFO operations after returning from the ISR. This is shown in Listing 3-3.

Listing 3-3. Storing and Restoring Bit FIFO State

/* Storing Bit FIFO State */
R0 = BFFWRP;
BFFWRP = 64;
R1 = BITEXT 32:
R2 = BITEXT 32;

/* Restoring the Bit FIFO State */
BFFWRP = 0;
BITDEP R2 BY 32;
BITDEP R1 BY 32;
BFFWRP = R0;

In the same fashion the bit FIFO can be used to extract and create different headers in a kind of time-division multiplex fashion by storing and
restoring the bit FIFO between two different sequences of bit FIFO operations.

If a bit FIFO related instruction is interrupted and the ISR uses the bit FIFO, the state of the bit FIFO must be preserved and restored by the ISR.

**Floating-Point Packing Instructions**

The processor supports a 16-bit floating-point storage format and provides instructions that convert the data for 40-bit computations. The 16-bit floating-point format uses an 11-bit mantissa with a 4-bit exponent plus a sign bit. The 16-bit data goes into bits 23 through 8 of a data register. Two shifter instructions, `FPACK` and `FUNPACK`, perform the packing and unpacking conversions between 32-bit floating-point words and 16-bit floating-point words. The `FPACK` instruction converts a 32-bit IEEE floating-point number in a data register into a 16-bit floating-point number. `FUNPACK` converts a 16-bit floating-point number in a data register to a 32-bit IEEE floating-point number. Each instruction executes in a single cycle.

When 16-bit data is written to bits 23 through 8 of a data register, the processor automatically extends the data into a 32-bit integer (bits 39 through 8).

The 16-bit floating-point format supports gradual underflow. This method sacrifices precision for dynamic range. When packing a number that would have underflowed, the exponent clears to zero and the mantissa (including a “hidden” 1) right-shifts the appropriate amount. The packed result is a denormal, which can be unpacked into a normal IEEE floating-point number.

The shifter instructions may help to perform data compression, converting 32-bit into 16-bit floating point, storing the data into short word space, and, if required, fetching and converting them back for further processing.
Shifter Instruction Summary

Table 3-7 lists the shifter instructions and Table 3-8 lists the shifter bit FIFO instructions (ADSP-214xx processors only) and shows how they relate to ASTATx/ASTATy flags. For more information on assembly language syntax, see Chapter 9, Instruction Set Types, and Chapter 10, Computation Instructions. In these tables, note the meaning of the following symbols:

- The \( R_n, R_x, R_y \) operands indicate any register file location; bit fields used depend on instruction.
- The \( F_n, F_x \) operands indicate any register file location; floating-point word.
- The * symbol indicates that the flag may be set or cleared, depending on data.
- In SIMD mode all instruction uses the complement data registers, immediate data are valid for both units.

Table 3-7. Shifter Instruction Summary

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ASTATx, ASTATy Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( SZ )</td>
</tr>
<tr>
<td>( R_n = LSHIFT R_x BY R_y )</td>
<td>+</td>
</tr>
<tr>
<td>( R_n = R_n OR LSHIFT R_x BY R_y )</td>
<td>+</td>
</tr>
<tr>
<td>( R_n = ASHIFT R_x BY R_y )</td>
<td>+</td>
</tr>
<tr>
<td>( R_n = R_n OR ASHIFT R_x BY R_y )</td>
<td>+</td>
</tr>
<tr>
<td>( R_n = ROT R_x BY R_y )</td>
<td>+</td>
</tr>
<tr>
<td>( R_n = BCLR R_x BY R_y )</td>
<td>+</td>
</tr>
<tr>
<td>( R_n = BSET R_x BY R_y )</td>
<td>+</td>
</tr>
<tr>
<td>( R_n = BTGL R_x BY R_y )</td>
<td>+</td>
</tr>
<tr>
<td>( R_n = FDEP R_x BY R_y )</td>
<td>+</td>
</tr>
</tbody>
</table>
Multifunction Computations

The processor supports multiple parallel (multifunction) computations by using the parallel data paths within its computational units. These instructions complete in a single cycle, and they combine parallel operation of the multiplier and the ALU or they perform dual ALU functions. The multiple operations work as if they were in corresponding single function computations. Multifunction computations also handle flags in the same
Functional Description

way as the single function computations, except that in the dual
add/subtract computation, the ALU flags from the two operations are
ORed together.

To work with the available data paths, the computational units constrain
which data registers hold the four input operands for multifunction com-
putations. These constraints limit which registers may hold the X input
and Y input for the ALU and multiplier.

Software Pipelining for Multifunction Instructions

As previously mentioned, multifunction instructions are parallel opera-
tions of both the ALU and multiplier units where each unit has new data
available after one cycle. However, for floating-point MAC operations, the
processor needs to emulate the MAC instruction with a multifunction
instruction. Results from the multiplier unit are available in the next cycle
for the ALU unit. Coding these instructions requires software pipelining
to ensure correct data as shown below.

F8=0; /* clear MAC result */
F12=F3*F7; /* first MUL */
lcntr=N-1, do (pc,1) until lce;
F12=F3*F7, F8=F8+F12; /* first ALU, loop body */
F8=F8+F12; /* last ALU */

Since a single floating-point MAC operation takes at least 2 cycles (for a
typical DSP application compute multiple data) the same example exer-
cised with a hardware loop body results in a throughput of 1 cycle per
word assuming a high word count.

Multifunction and Data Move

Another type of multifunction operation available on the processor com-
bines transfers between the results and data registers and transfers between
memory and data registers. These parallel operations complete in a single
cycle. For example, the processor can perform the following MAC and
Processing Elements

parallel read of data memory. However if data dependency exists, software pipeline coding is required as shown in Listing 3-4.

Listing 3-4. MAC and Parallel Read With Software Pipeline Coding

\begin{verbatim}
MRF=0, R5 = DM(I1,M2), R6 = PM(I9,M9); /* first data */
Lcntr=N-1, do (pc,1) until lce;
MRF = MRF-R5*R6, R5 = DM(I1,M2), R6 = PM(I9,M9); /* loop body */
MRF = MRF-R5*R6; /* last MAC*/
\end{verbatim}

Another example is illustrated for an IIR biquad stage in Listing 3-5:

Listing 3-5. IIR Biquad Stage

\begin{verbatim}
B1=B0;
F12=F12-F12, F2 = DM(I0,M1), F4 = PM(I8,M8); /* first data */
Lcntr=N, do (pc,4) until lce; /* loop body */
F12=F2*F4, F8=F8+F12, F3 = DM(I0,M1), F4 = PM(I8,M8);
F12=F3*F4, F8=F8+F12, DM(I1,M1)=F3, F4 = PM(I8,M8);
F12=F2*F4, F8=F8+F12, F2 = DM(I0,M1), F4 = PM(I8,M8);
F12=F3*F4, F8=F8+F12, DM(I1,M1)=F8, F4 = PM(I8,M8);
RTS(db), F8=F8+F12, /* last MAC */
Nop;
Nop;
\end{verbatim}

Multifunction Input 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 3-8. 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.

The multiport data register file can normally be read from and written to without restriction. However, in multifunction instructions, the ALU and
**Functional Description**

Multiplier input are restricted to particular sets of registers while the outputs are unrestricted.

![Diagram](image.png)

**Figure 3-8. Permitted Input Registers for Multifunction Computations**

**Multifunction Input Modifier Constraints**

The multifunction fixed-point computation does support the instruction input modifier signed signed fractional (SSF) and signed signed fractional rounding (SSFR) only.

**Multifunction Instruction Types**

Table 3-10, through Table 3-13 list the multifunction computations. For more information on assembly language syntax, see Chapter 9, Instruction Set Types, and Chapter 10, Computation Instructions. Table 3-9 provides the description of the following symbols. Note that the MRB register is not supported in multifunction instructions.
Table 3-9. Multifunction Computation Symbol Descriptions

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3–0</td>
<td>Fixed-point multiplier x input data file registers</td>
</tr>
<tr>
<td>F3–0</td>
<td>Floating-point multiplier x input data file registers</td>
</tr>
<tr>
<td>R7–4</td>
<td>Fixed-point multiplier y input data file registers</td>
</tr>
<tr>
<td>F7–4</td>
<td>Floating-point multiplier y input data file registers</td>
</tr>
<tr>
<td>R11–8</td>
<td>Fixed-point ALU x input data file registers</td>
</tr>
<tr>
<td>F11–8</td>
<td>Floating-point ALU x input data file registers</td>
</tr>
<tr>
<td>R15–12</td>
<td>Fixed-point ALU y input data file registers</td>
</tr>
<tr>
<td>F15–12</td>
<td>Floating-point ALU y input data file registers</td>
</tr>
<tr>
<td>SSFR</td>
<td>the X input is signed, the Y input is signed, use fractional inputs, and rounded-to-nearest output</td>
</tr>
<tr>
<td>SSF</td>
<td>the X input is signed, Y input is signed, use fractional input</td>
</tr>
</tbody>
</table>

Table 3-10. Dual Add and Subtract

Ra = Rx + Ry, Rs = Rx – Ry
Fa = Fx + Fy, Fs = Fx – Fy

Table 3-11. Fixed-Point Multiply and Add, Subtract, or Average

(Any combination of left and right column)

<table>
<thead>
<tr>
<th>Ra = R11–8 + R15–12</th>
<th>Ra = R11–8 – R15–12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm = R3–0 * R7–4 (SSFR),</td>
<td>Rm = MRF + R3-0 * R7–4 (SSF),</td>
</tr>
<tr>
<td>MRF = MRF + R3-0 * R7–4 (SSF),</td>
<td>MRF = MRF – R3-0 * R7–4 (SSF),</td>
</tr>
<tr>
<td>Rm = MRF + R3-0 * R7–4 (SSF),</td>
<td>Rm = MRF – R3-0 * R7–4 (SSF),</td>
</tr>
</tbody>
</table>
Functional Description

Table 3-12. Floating-Point Multiply and ALU Operation

<table>
<thead>
<tr>
<th>Formula</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F_m = F_{3-0} \cdot F_{7-4} )</td>
<td>( F_a = F_{11-8} + F_{15-12} )</td>
</tr>
<tr>
<td>( F_m = F_{3-0} \cdot F_{7-4} )</td>
<td>( F_a = F_{11-8} - F_{15-12} )</td>
</tr>
<tr>
<td>( F_m = F_{3-0} \cdot F_{7-4} )</td>
<td>( F_a = \text{FLOAT R}<em>{11-8} ) by ( R</em>{15-12} )</td>
</tr>
<tr>
<td>( F_m = F_{3-0} \cdot F_{7-4} )</td>
<td>( R_a = \text{FIX F}<em>{11-8} ) by ( R</em>{15-12} )</td>
</tr>
<tr>
<td>( F_m = F_{3-0} \cdot F_{7-4} )</td>
<td>( F_a = (F_{11-8} + F_{15-12})/2 )</td>
</tr>
<tr>
<td>( F_m = F_{3-0} \cdot F_{7-4} )</td>
<td>( F_a = \text{ABS F}_{11-8} )</td>
</tr>
<tr>
<td>( F_m = F_{3-0} \cdot F_{7-4} )</td>
<td>( F_a = \text{MAX (F}<em>{11-8}, F</em>{15-12}) )</td>
</tr>
<tr>
<td>( F_m = F_{3-0} \cdot F_{7-4} )</td>
<td>( F_a = \text{MIN (F}<em>{11-8}, F</em>{15-12}) )</td>
</tr>
</tbody>
</table>

Note that both instructions in Table 3-13 are typically used for fixed- or floating-point FFT butterfly calculations.

Table 3-13. Multiply With Dual Add and Subtract

<table>
<thead>
<tr>
<th>Formula</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_m = R_{3-0} \cdot R_{7-4} ) (SSFR)</td>
<td>( R_a = R_{11-8} + R_{15-12} ), ( R_s = R_{11-8} - R_{15-12} )</td>
</tr>
<tr>
<td>( F_m = F_{3-0} \cdot F_{7-4} )</td>
<td>( F_a = F_{11-8} + F_{15-12} ), ( F_s = F_{11-8} - F_{15-12} )</td>
</tr>
</tbody>
</table>

Operating Modes

The MODE1 register controls the operating mode of the processing elements. Table A-1 on page A-4 lists the bits in the MODE1 register. The bits are described in the following sections.

ALU Saturation

When the ALUSAT bit in the MODE1 register is set (= 1), the ALU is in saturation mode. In this mode, positive fixed-point overflows return the maximum positive fixed-point number (0x7FFF FFFF), and negative overflows return the maximum negative number (0x8000 0000).

When the ALUSAT bit is cleared (= 0), fixed-point results that overflow are not saturated, the upper 32 bits of the result are returned unaltered.
Short Word Sign Extension

In short word space, the upper 16-bit word is not accessed. If the SSE bit in MODE1 is set (1), the processor sign-extends the upper 16 bits. If the SSE bit is cleared (0), the processor zeros the upper 16 bits.

Floating-Point Boundary Rounding Mode

In the default mode, (RND32 bit = 1), the processor supports a 40-bit extended-precision floating-point mode, which has eight additional LSBs of the mantissa and is compliant with the 754/854 standards. However, results in this format are more precise than the IEEE single-precision standard specifies. Extended-precision floating-point data uses a 31-bit mantissa with a 8-bit exponent plus sign bit.

For rounding mode the multiplier and ALU support a single-precision floating-point format, which is specified in the IEEE 754/854 standard.

IEEE single-precision floating-point data uses a 23-bit mantissa with an 8-bit exponent plus sign bit. In this case, the computation unit sets the eight LSBs of floating-point inputs to zeros before performing the operation. The mantissa of a result rounds to 23 bits (not including the hidden bit), and the 8 LSBs of the 40-bit result clear to zeros to form a 32-bit number, which is equivalent to the IEEE standard result.

Information: In fixed-point to floating-point conversion, the rounding boundary is always 40 bits, even if the RND32 bit is set.

For more information on this standard, see Appendix C, Numeric Formats. This format is IEEE 754/854 compatible for single-precision floating-point operations in all respects except for the following:

- The processor does not provide inexact flags. An inexact flag is an exception flag whose bit position is inexact. The inexact exception occurs if the rounded result of an operation is not identical to the exact (infinitely precise) result. Thus, an inexact exception always occurs when an overflow or an underflow occurs.
Functional Description

- NAN (Not-A-Number) inputs generate an invalid exception and return a quiet NAN (all 1s).

- Denormal operands, using denormalized (or tiny) numbers, flush to zero when input to a computational unit and do not generate an underflow exception. A denormal operand is one of the floating-point operands with an absolute value too small to represent with full precision in the significant. The denormal exception occurs if one or more of the operands is a denormal number. This exception is never regarded as an error.

- The processor supports round-to-nearest and round-toward-zero modes, but does not support round to $+\infty$ and round-to-infinity.

Rounding Mode

The $\text{TRUNC}$ bit in the $\text{MODE1}$ register determines the rounding mode for all ALU operations, all floating-point multiplies, and fixed-point multiplies of fractional data. The processor supports two rounding modes—round-toward-zero and round-toward-nearest. The rounding modes comply with the IEEE 754 standard and have the following definitions.

- Round-toward-zero ($\text{TRUNC}$ bit = 1). If the result before rounding is not exactly representable in the destination format, the rounded result is the number that is nearer to zero. This is equivalent to truncation.

- Round-toward-nearest ($\text{TRUNC}$ bit = 0). If the result before rounding is not exactly representable in the destination format, the rounded result is the number that is nearer to the result before rounding. If the result before rounding is exactly halfway between two numbers in the destination format (differing by an LSB), the rounded result is the number that has an LSB equal to zero.
Statistically, rounding up occurs as often as rounding down, so there is no large sample bias. Because the maximum floating-point value is one LSB less than the value that represents infinity, a result that is halfway between the maximum floating-point value and infinity rounds to infinity in this mode.

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

**Multiplier Result Register Swap**

Each multiplier has a primary or foreground (MRF) register and alternate or background (MRB) results register. The (SRCU) bit in the MODE1 register selects which result register receives the result from the multiplier operation, swapping which register is the current MRF or MRB. This swapping facilitates context switching.

Unlike other registers that have alternates, both the MRF and MRB registers are coded into instructions, without regard to the state of the MODE1 register as shown in the following example.

\[
\begin{align*}
\text{MRB} &= \text{MRB} - R3 \times R2 \text{ (SSFR)}; \\
\text{MRF} &= \text{MRF} + R4 \times R12 \text{ (UUI)};
\end{align*}
\]

With this arrangement, programs can use the result registers as primary and alternate accumulators, or programs can use these registers as two parallel accumulators. This feature facilitates complex math. The MODE1 register controls the access to alternate registers. In SIMD mode, swapping also occurs with the PEY unit based registers (MSF and MSB).
Functional Description

SIMD Mode

The SHARC core contains two sets of computational units and associated register files. As shown in Figure 1-1 on page 1-4, these two processing elements (PEx and PEy) support SIMD operation.

The mode1 register controls the operating mode of the processing elements. The PEYEN bit (bit 21) in the mode1 register enables or disables the PEy processing element. When PEYEN is cleared (0), the processor operates in SISD mode, using only PEx. When the PEYEN bit is set (1), the processor operates in SIMD mode, using both the PEx and PEy processing elements. There is a one cycle delay after PEYEN is set or cleared, before the mode change takes effect.

For shift immediate instructions the Y input is driven by immediate data from the instructions (and has no complement data as a register does). If using SIMD mode, the immediate data are valid for both PEx and PEy units as shown in Listing 3-6.

Listing 3-6. Compute Instructions in SIMD Mode

```
bit set mode1 peyen;       /* enable SIMD */
nop;                       /* effect latency */
R0 = R1 + R2;           /* explicit ALU instruction */
S0 = S1 + S2;           /* implicit ALU instruction */
F0 = F1 *  F2;           /* explicit MUL instruction */
SF0 = SF1 * SF2;          /* implicit MUL instruction */

MRB = MRB - R3 * R2 (SSFR): /* explicit MUL instruction */
MSB = MSB - S3 * S2 (SSFR): /* implicit MUL instruction */

R5 = LSHIFT R6 by <data8>; /* explicit shift imm instruction */
S5 = LSHIFT S6 by <data8>; /* implicit shift imm instruction */
```
To support SIMD, the processor performs these parallel operations:

- Dispatches a single instruction to both processing element’s computational units.
- Loads two sets of data from memory, one for each processing element.
- Executes the same instruction simultaneously in both processing elements.
- Stores data results from the dual executions to memory.

Using the information here and in Chapter 9, Instruction Set Types, and Chapter 10, Computation Instructions, it is possible, using SIMD mode’s parallelism, to double performance over similar algorithms running in SISD (ADSP-2106x processor compatible) mode.

The two processing elements are symmetrical; each contains these functional blocks:

- ALU
- Multiplier primary and alternate result registers
- Shifter
- Data register file and alternate register file

**Conditional Computations in SIMD Mode**

Conditional computations allows the computation units to make computations conditional in SIMD mode. For more information, see “Conditional Instruction Execution” on page 4-91.
Arithmetic Interrupts

Interrupt Mode Mask

On the SHARC processors, programs can mask automated individual operating mode bits in the MODE1 register by entering into an ISR. This reduces latency cycles.

For the processing units, the short word sign extension (SSE) the truncation (TRUNC) the ALU saturation (ALUSAT) the floating-point boundary rounding (RND32) and the multiply register swap (SRCU) bits can be masked. For more information, see Chapter 4, Program Sequencer.

Arithmetic Interrupts

The following sections describe how the processor core handles arithmetic interrupts. Note that the shifter does not generate interrupts for exception handling.

Computational Status

The multiplier and ALU each provide exception information when executing floating-point or fixed-point operations (see Table 3-14). Each unit updates overflow, underflow, and invalid operation flags in the processing element’s arithmetic status (ASTATx and ASTATy) registers and sticky status (STKYx and STKYy) registers. An underflow, overflow, or invalid operation from any unit also generates a maskable interrupt. There are three ways to use floating-point or fixed-point exceptions from computations in program sequencing.

- Enable interrupts and use an interrupt service routine (ISR) to handle the exception condition immediately. This method is appropriate if it is important to correct all exceptions as they occur.
• Use conditional instructions to test the exception flags in the ASTATx or ASTATy registers after the instruction executes. This method permits monitoring each instruction’s outcome.

• Use the bit test (BTST) instruction to examine exception flags in the STKY register after a series of operations. If any flags are set, some of the results are incorrect. Use this method when exception handling is not critical.

### Computation Status Update Priority

Flag updates occur at the end of the cycle in which the status is generated and is available on the next cycle. If a program writes the arithmetic status register or sticky status register explicitly in the same cycle that the unit is performing an operation, the explicit write to the status register supersedes any flag update from the unit operation as shown in the following example.

\[
R0 = R1 + R2, \text{ASTATx} = R6; \quad /* R6 overrides ALU status */ \\
F0 = F1 \times F2, \text{STKYx} = F6; \quad /* F6 overrides MUL status */
\]

Interrupt processing starts two cycles after an arithmetic exception occurs because of the one cycle delay between an arithmetic exception and the STKYx, STKYy register update.

### Table 3-14. Exceptions versus Processing Units

<table>
<thead>
<tr>
<th>Flag bits (ASTAT/STYK)</th>
<th>ALU</th>
<th>Multiplier</th>
<th>Shifter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-point Overflow Interrupt</td>
<td>AV</td>
<td>MV</td>
<td>—</td>
</tr>
<tr>
<td>Float-point Underflow Interrupt</td>
<td>AZ</td>
<td>MU</td>
<td>—</td>
</tr>
<tr>
<td>Float-point Overflow Interrupt</td>
<td>AV</td>
<td>MV</td>
<td>—</td>
</tr>
<tr>
<td>Float-point Invalid Interrupt</td>
<td>AI</td>
<td>MI</td>
<td>—</td>
</tr>
</tbody>
</table>
Arithmetic Interrupts

SIMD Computation and Status Flags

When the processors are in SIMD mode, computations on both processing elements generate status flags, producing a logical ORing of the exception status test on each processing element.

Table 3-15. Computation Status Register Pairs

<table>
<thead>
<tr>
<th>ASTATx</th>
<th>ASTATy</th>
<th>STKYx</th>
<th>STKYy</th>
</tr>
</thead>
</table>

SIMD Computation Interrupts

If one of the four fixed-point or floating-point exceptions is enabled, an exception condition on one or both processing elements generates an exception interrupt. Interrupt service routines (ISRs) must determine which of the processing elements encountered the exception. Returning from a floating-point interrupt does not automatically clear the STKY state. Program code must clear the STKY bits in both processing element’s sticky status (STKYx and STKYy) registers as part of the exception service routine. For more information, see “Interrupt Branch Mode” on page 4-26.

ALU Interrupts

Table 3-16 provides an overview of the ALU interrupts.

Table 3-16. ALU Interrupt Overview

<table>
<thead>
<tr>
<th>Interrupt Source</th>
<th>Interrupt Condition</th>
<th>Interrupt Priorities</th>
<th>Interrupt Acknowledge</th>
<th>IVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>ALU fixed-point overflow</td>
<td>33–36</td>
<td>Clear STKYx/y + RTI instruction</td>
<td>FIXI, FLTOI, FLTUI, FLTIII</td>
</tr>
<tr>
<td></td>
<td>ALU floating-point overflow</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ALU floating-point underflow</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ALU invalid floating-point</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Processing Elements

ALU operations update seven status flags in the processing element’s arithmetic status (ASTATx and ASTATy) registers. The following bits in ASTATx or ASTATy registers flag the ALU status (a 1 indicates the condition) of the most recent ALU operation.

- ALU result zero or floating-point underflow, bit 0 (AZ)
- ALU overflow, bit 1 (AV)
- ALU result negative, bit 2 (AN)
- ALU fixed-point carry, bit 3 (AC)
- ALU X input sign for ABS, MANT operations, bit 4 (AS)
- ALU floating-point invalid operation, bit 5 (AI)
- Last ALU operation was a floating-point operation, bit 10 (AF)
- Compare accumulation register results of last eight compare operations, bits 31–24 (CACC)

ALU operations also update four sticky status flags in the processing element’s sticky status (STKYx and STKYy) registers. The following bits in STKYx or STKYy flag the ALU status (a 1 indicates the condition). Once set, a sticky flag remains high until explicitly cleared.

- ALU floating-point underflow, bit 0 (AUS)
- ALU floating-point overflow, bit 1 (AVS)
- ALU fixed-point overflow, bit 2 (AOS)
- ALU floating-point invalid operation, bit 5 (AIS)

Interrupt Acknowledge

After an exception has been detected the ISR routine needs to clear the flag bit as shown in Listing 3-7.
### Arithmetic Interrupts

Listing 3-7. Clearing the FLAG Bit

ISR_ALU_Exception:

```assembly
bit tst STKYx AVS; /* check condition */
```

IF TF jump ALU_Float_Overflow:

```assembly
bit tst STKYx AOS; /* check condition */
IF TF jump ALU_Fixed_Overflow;
```

ALU_Fixed_Overflow:

```assembly
bit clr STKYx AOS; /* clear sticky bit */
rti;
```

ALU_Float_Overflow:

```assembly
bit clr STKYx AVS; /* clear sticky bit */
rti;
```

### Multiplier Interrupts

Table 3-17 provides an overview of the multiplier interrupts.

**Table 3-17. Multiplier Interrupt Overview**

<table>
<thead>
<tr>
<th>Interrupt Source</th>
<th>Interrupt Condition</th>
<th>Interrupt Priorities</th>
<th>Interrupt Acknowledge</th>
<th>IVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier</td>
<td>MUL fixed-point overflow</td>
<td>33–36</td>
<td>Clear STKYx/y + RTI instruction</td>
<td>FIXI</td>
</tr>
<tr>
<td></td>
<td>MUL floating-point overflow</td>
<td></td>
<td></td>
<td>FLTOI</td>
</tr>
<tr>
<td></td>
<td>MUL floating-point underflow</td>
<td></td>
<td></td>
<td>FLTUI</td>
</tr>
<tr>
<td></td>
<td>MUL invalid floating-point</td>
<td></td>
<td></td>
<td>FLTH</td>
</tr>
</tbody>
</table>

Multiplier operations update four status flags in the processing element’s arithmetic status registers (ASTATx and ASTATy). A 1 indicates the condition of the most recent multiplier operation and are as follows.

- Multiplier result negative, bit 6 (MN)
- Multiplier overflow, bit 7 (MV)
Processing Elements

- Multiplier underflow, bit 8 (MU)
- Multiplier floating-point invalid operation, bit 9 (MI)

Multiplier operations also update four “sticky” status flags in the processing element’s sticky status (STKYx and STKYy) registers. Once set (a 1 indicates the condition), a sticky flag remains set until explicitly cleared. The bits in the STKYx or STKYy registers are as follows.

- Multiplier fixed-point overflow, bit 6 (MOS)
- Multiplier floating-point overflow, bit 7 (MVS)
- Multiplier underflow, bit 8 (MUS)
- Multiplier floating-point invalid operation, bit 9 (MIS)

Shifter Status Flags

Shifter operations update four status flags in the processing element’s arithmetic status registers (ASATx and ASATy) where a 1 indicates the condition. The bits that indicate shifter status for the most recent ALU operation are as follows.

- Shifter overflow of bits to left of MSB, bit 11 (SV)
- Shifter result zero, bit 12 (SZ)
- Shifter input sign for exponent extract only, bit 13 (SS)
- Shifter bit FIFO status (SF)

Note that the shifter does not generate an exception handle.
Arithmetic Interrupts
4 PROGRAM SEQUENCER

The program sequencer is responsible for the control flow of programs and data within the processor. It is closely connected to the system interface, DAGs and cache. It controls non sequential program flows such as jumps, calls and loop instructions.

The program sequencer controls program flow (see Figure 4-1) by constantly providing the address of the next instruction to be fetched for execution. Program flow in the processors is mostly linear, with the processor executing instructions sequentially. This linear flow varies occasionally when the program branches due to nonsequential program structures, such as those described below. Nonsequential structures direct the processor to execute an instruction that is not at the next sequential address following the current instruction.

Features

The sequencer controls the following operations.

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

- **Subroutines.** The processor temporarily breaks sequential flow to execute instructions from another part of program memory.

- **Jumps.** Program flow is permanently transferred to another part of program memory.
Features

- **Interrupts.** Subroutines in which a runtime event (not an instruction) triggers the execution of the routine.
- **Idle.** An instruction that causes the processor to cease operations and hold its current state until an interrupt occurs. Then, the processor services the interrupt and continues normal execution.

Figure 4-1. Program Flow
Program Sequencer

- **ISA or VISA** instruction fetches. The fetch address is interpreted as an ISA (NW address, traditional) or VISA instruction (SW address) this allows fast switching between both instruction types.

- **Direct Addressing.** Provides data address specified as absolute value in instruction.

The sequencer manages execution of these program structures by selecting the address of the next instruction to execute. As part of its process, the sequencer handles the following tasks:

- Increments the fetch address
- Maintains stacks
- Evaluates conditions
- Decrements the loop counter
- Calculates new addresses
- Maintains an instruction cache
- Interrupt control

To accomplish these tasks, the sequencer uses the blocks shown in Figure 4-2. The sequencer’s address multiplexer selects the value of the next fetch address from several possible sources. The fetched address enters the instruction pipeline, made up of the fetch1, fetch2, decode, address, and execute registers. These contain the 24-bit addresses of the instructions currently being fetched, decoded, and executed. The program counter, coupled with the program counter stack, which stores return addresses and top-of-loop addresses. All addresses generated by the sequencer are 24-bit program memory instruction addresses.
Functional Description

The sequencer uses the blocks shown in Figure 4-2 to execute instructions. The sequencer’s address multiplexer selects the value of the next fetch address from several possible sources. These registers contain the 24-bit addresses of the instructions currently being fetched, decoded, and executed.

Figure 4-2. Sequencer Control Diagram

Functional Description
Program Sequencer

Instruction Pipeline

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

To achieve a high execution rate while maintaining a simple programming mode, the processor employs a five stage interlocked pipeline, shown in Table 4-1, to process instructions and simplify programming models. All possible hazards are controlled by hardware.

The legacy Instruction Set Architecture (ISA) instructions are addressed using normal word (NW) address space, whereas Variable Instruction Set Architecture (VISA) instructions are addressed using short word (SW) address space. Switching between traditional ISA and VISA instruction spaces happens not via any bit settings in any registers. Instead, the transition occurs automatically when branches (JUMP/CALL or interrupts) take the execution from ISA address space to VISA address space or vice versa.

Note that the processor always emerges from reset in ISA mode, so the interrupt vector table must always reside in ISA address space.

The processor controls the fetch address, decode address, and program counter (FADDR, DADDR, and PC) registers which store the Fetch1, decode, and execution phase addresses of the pipeline.
**Table 4-1. Instruction Pipeline Processing Stages**

<table>
<thead>
<tr>
<th>Stage</th>
<th>ISA</th>
<th>Visa Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch1</td>
<td>In this stage, the appropriate instruction address is chosen from various sources and driven out to memory. The instruction address is matched with the cache to generate a condition for cache miss/hit. The next NW address is auto incremented by one.</td>
<td>Next SW address is auto incremented by three for every 48-bit fetch.</td>
</tr>
<tr>
<td>Fetch2</td>
<td>This stage is the data phase of the instruction fetch memory access wherein the data address generator (DAG) performs some amount of pre-decode. Based on a cache condition, the instruction is read from cache/driven from the memory instruction data bus.</td>
<td>Stores 3 x 16-bit instruction data into the IAB buffer and presents 1 instruction/cycle to the decoder.</td>
</tr>
<tr>
<td>Decode</td>
<td>The instruction is decoded and various conditions that control instruction execution are generated. The main active units in this stage are the DAGs, which generate the addresses for various types of functions like data accesses (load/store) and indirect branches. DAG premodify (M+I) operation is performed. For a cache miss, instruction data read from memory are loaded into the cache.</td>
<td>Decode VISA instruction; store its length information in short words.</td>
</tr>
<tr>
<td>Address</td>
<td>The addresses generated by the DAGs in the previous stage are driven to the memory through memory interface logic. The addresses for the branch operation are made available to the fetch unit. For instruction branches (Call/Jump) the address is forward to the Fetch1 stage. For a do until instruction the next address is fetched.</td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td>The operations specified in the instruction are executed and the results written back to memory or the universal registers. For interrupt branch the IVT address is forward to the Fetch1 stage. ISA instructions always increment PC value by 1 each cycle.</td>
<td>Executing VISA instructions the PC value is incremented by 1, 2 or 3 depending on length information from the Instruction decode.</td>
</tr>
</tbody>
</table>

**VISA Instruction Alignment Buffer (IAB)**

The IAB, shown in Figure 4-3, is a 5 short-word (5 x 16-bit words) capacity FIFO that is part of the program sequencer. The IAB is responsible for buffering 48 bits of code at a time from memory per cycle and presenting one instruction per core clock cycle (CCLK) to the execution unit. When
the instruction is shorter than 48 bits, the IAB keeps the unused bits for the next cycle. When the IAB determines that it has no room to accommodate 48 more bits from memory, it stalls the fetch engine. Consequently, the average fetch bandwidth for executing VISA instructions is less than 48 bits per cycle.

A decode of the instruction indicates the length of the instruction in unit of short words. At the end of the current decode cycle, the short words that are part of the current instruction are discarded and the remaining bits are shifted left to align at the MSB of IAB. The three fetched short words in the following cycle are concatenated to the existing bits of IAB. The next instruction, therefore, is always available in MSB aligned fashion.
Functional Description

Linear Program Flow

In the sequential program flow, when one instruction is being executed, the next four instructions that follow are being processed in the Address, Decode, Fetch2 and Fetch1 stages of the instruction pipeline. Sequential program flow usually has a throughput of one instruction per cycle.

Table 4-2 illustrates how the instructions starting at address n are processed by the pipeline. While the instruction at address n is being executed, the instruction n+1 is being processed in the address phase, n+2 in the Decode phase, n+3 in the Fetch2 phase and n+4 in the Fetch1 phase.

Table 4-2. ISA/VISA Linear Flow 48-bit Instructions Only

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td></td>
<td></td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td></td>
<td></td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td></td>
</tr>
<tr>
<td>Decode</td>
<td></td>
<td></td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
</tr>
<tr>
<td>Fetch2</td>
<td></td>
<td></td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
</tr>
<tr>
<td>Fetch1</td>
<td></td>
<td></td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
</tr>
</tbody>
</table>

In VISA mode, the situation is different since the instruction fetch rate is always 48 bits but the consumption rate can vary. In Table 4-3, the instruction fetch (48-bit) stalls because the IAB FIFO is filling up. After decoding the next instructions, the IAB indicates space for new instructions which tells the sequencer to continue fetching by increasing the program counter.

The sequencer continues to fetch 48 bits from memory until cycle 3 because it knows the instruction width of n only when it is decoded. In cycle 4 (Table 4-3), the decoder tells the sequencer that n+1 is now 16 bits wide. By now, the sequencer has fetched 9 short words (n to n+8). The IAB can buffer up to 5 short words and since the sequencer has already
fetched 2 short words \((n, n+1)\), the sequencer now stalls the fetch and holds the fetched short words in intermediate buffers and the IAB. As instructions are executed, the IAB frees up and the fetch starts again.

**Table 4-3. VISA Linear Flow 16-bit Instructions Only**

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
<td>n+7</td>
<td>n+8</td>
<td>n+9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
<td>n+7</td>
<td>n+8</td>
<td>n+9</td>
<td>n+10</td>
<td>n+11</td>
<td>n+12</td>
<td></td>
</tr>
<tr>
<td>Decode</td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
<td>n+7</td>
<td>n+8</td>
<td>n+9</td>
<td>n+10</td>
<td>n+11</td>
<td>n+12</td>
<td></td>
</tr>
<tr>
<td>Fetch2</td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
<td>n+7</td>
<td>n+8</td>
<td>n+9</td>
<td>n+10</td>
<td>n+11</td>
<td>n+12</td>
<td></td>
</tr>
<tr>
<td>Fetch1</td>
<td>n</td>
<td>n+3</td>
<td>n+6</td>
<td>n+9</td>
<td>n+12</td>
<td>n+15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Instr Fetch n: 16-bit instr \(n\) to (n+2)*  
*Instr Fetch n+3: 16-bit instr \((n+3)\) to (n+5)*

**Direct Addressing**

Similar to the DAGs, the sequencer also provides the data address for direct addressing types as shown in the following example.

```c
R0 = DM(0x90500); /* sequencer generated data address */
PM(0x90600) = R7; /* sequencer generated data address */
```

as compared to the DAG

```c
R0 = DM(10.M0); /* DAG1 generated data address */
PM(18.M8) = R7; /* DAG2 generated data address */
```

*For more information, see Chapter 6, Data Address Generators.*
Variation In Program Flow

While sequential execution takes one core clock cycle per instruction, nonsequential program flow can potentially reduce the instruction throughput. Non-sequential program operations include:

- Jumps
- Subroutine calls and returns
- Interrupts and returns
- Loops

**Functional Description**

In order to manage these variations, the processor uses several mechanisms, primarily hardware stacks, which are described in the following sections.

**Hardware Stacks**

If the programmed flow varies (non-sequential and interrupted), the processor requires hardware or software mechanisms (stacks, Table 4-4) to support changes of the regular program flow. The SHARC core supports three hardware stack types which are implemented outside of the memory space and are used and accessed for any non-sequential process. The stack types are:

- Program count stack – Used to store the return address (call, IVT branch, do until).
- Status stack – Used to store some context of status registers.
- “Loop Stack” on page 4-48 for address and count – Used for hardware looping (un-nested and nested). This stack is described in “Loop Sequencer” section later in this chapter.
The SHARC processor does not have a general-purpose hardware stack. However, the DAG architecture allows a software stack implementation by using post (push) and pre-modify (pop) DAG instruction types.

The stacks are fully controlled by hardware. Manipulation of these stacks by using explicit PUSH/POP instructions and explicit writes to PCSTK, LADDR and CURLCNTR registers may affect the correct functioning of the loop.

Table 4-4. Core Stack Overview

<table>
<thead>
<tr>
<th>Attribute</th>
<th>PC Stack</th>
<th>Loop Address Stack</th>
<th>Loop Count Stack</th>
<th>Status Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Size</td>
<td>30 x 26 bits</td>
<td>6 x 32 bits</td>
<td>6 x 32 bits</td>
<td>15 x 3 x 32 bits</td>
</tr>
<tr>
<td>Top Entry</td>
<td>Return Address</td>
<td>Loop End Address</td>
<td>Loop iteration count</td>
<td>MODE1 ASTATx/ASTATy</td>
</tr>
<tr>
<td>Empty Flag</td>
<td>PCEM</td>
<td>LSEM</td>
<td></td>
<td>SSEM</td>
</tr>
<tr>
<td>Full Flag</td>
<td>PCFL</td>
<td>LSOV</td>
<td></td>
<td>SSOV</td>
</tr>
<tr>
<td>Stack Pointer</td>
<td>PCSTKP</td>
<td>No</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>Exception IRQ</td>
<td>SOVFI</td>
<td>SOVFI</td>
<td></td>
<td>SOVFI</td>
</tr>
</tbody>
</table>

Automated Access

<table>
<thead>
<tr>
<th>Push Condition</th>
<th>CALL, IVT branch DO UNTIL</th>
<th>DO UNTIL</th>
<th>IVT Branch (Timer, TR02~0 only)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pop Condition</td>
<td>RTS, RTI</td>
<td>CURLCNTR = 1 or COND = true</td>
<td>RTI (Timer, TR02~0 only)</td>
</tr>
</tbody>
</table>

Manual Access

<table>
<thead>
<tr>
<th>Register Access</th>
<th>PCSTK</th>
<th>LADDR</th>
<th>CURLCNTR</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit Push</td>
<td>Push PCSTK</td>
<td>Push Loop</td>
<td>Push STS</td>
<td></td>
</tr>
<tr>
<td>Explicit Pop</td>
<td>Pop PCSTK</td>
<td>Pop Loop</td>
<td>Pop STS</td>
<td></td>
</tr>
</tbody>
</table>
PC Stack Access

The sequencer includes a program counter (PC) stack, which appears in Figure 4-2 on page 4-4. At the start of a subroutine or loop, the sequencer pushes return addresses for subroutines (CALL instructions with RTI/RTS) and top-of-loop addresses for loops (DO/UNTIL instructions) onto the PC stack. The sequencer pops the PC stack during a return from interrupt (RTI), return from subroutine (RTS), and a loop termination.

The program counter (PC) register is the last stage in the instruction pipeline. It contains the 24-bit address of the instruction the processor executes on the next cycle. This register, combined with the PC stack (PCSTK) register, stores return addresses and top-of-loop addresses.

For the ADSP-2137x processors and later, the PC register size has been enlarged to 26-bits. This allows read/write to the former hidden bits allowing full software control of the stack registers.

PC Stack Status

The PC stack is 30 locations deep. The stack is full when all entries are occupied, is empty when no entries are occupied, and is overflowed if a push occurs when the stack is full. The following bits in the STKYx register indicate the PC stack full and empty states.

- **PC stack full.** Bit 21 (PCFL) indicates that the PC stack is full (if 1) or not full (if 0)—not a sticky bit, cleared by a pop.

- **PC stack empty.** Bit 22 (PCEM) indicates that the PC stack is empty (if 1) or not empty (if 0)—not sticky, cleared by a push.

To prevent a PC stack overflow, the PC stack full condition generates the (maskable) stack overflow interrupt (SOVF1). This interrupt occurs when the PC stack has 29 of 30 locations filled (the almost full state). The PC stack full interrupt occurs at this point because the PC stack full interrupt service routine needs that last location for its return address.
**Program Sequencer**

**PC Stack Manipulation**

The **PCSTK** register contains the top entry on the PC stack. This register is readable and writable by the core. Reading from and writing to **PCSTK** does not move the PC stack pointer. Only a stack push or pop performed with explicit instructions moves the stack pointer. The **PCSTK** register contains the value 0x3FFFFF when the PC stack is empty. A write to **PCSTK** has no effect when the PC stack is empty. “Program Counter Stack Register (PCSTK)” on page A-10 lists the bits in the **PCSTK** register.

The address of the top of the PC stack is available in the **PC stack pointer (PCSTKP)** register. The value of **PCSTKP** is zero when the PC stack is empty, is 1 through 30 when the stack contains data, and is 31 when the stack overflows. A write to **PCSTKP** takes effect after one cycle of delay. If the PC stack is overflowed, a write to **PCSTKP** has no effect. For example a write to **PCSTKP** = 3 deletes all entries except the three oldest.

**PC Stack Access Priorities**

Since the architecture allows manipulation of the stack, simultaneous stack accesses may occur (writes to the **PCSTK** register during a branch). In such a case the **PCSTK** access has higher priority over the push operation from the sequencer.

**Status Stack Access**

The sequencer’s status stack eases the return from branches by eliminating some service overhead like register saves and restores as shown in the following example.

```c
CALL fft1024;       /* Where fft1024 is an address label */
fft1024:push sts;   /* save MODE1/ASTATx/y registers */
instruction;
instruction;
pop sts;            /* re-store MODE1/ASTATx/y registers */
rts;
```

---

SHARC Processor Programming Reference 4-13
For some interrupts, (IRQ2-0 and timer expired), the sequencer automatically pushes the ASTATx, ASTATy, and MODE1 registers onto the status stack. When the sequencer pushes an entry onto the status stack, the processor uses the MMASK register to clear the corresponding bits in the MODE1 register. All other bit settings remain the same. See the example in “Interrupt Mask Mode” on page 4-39.

The sequencer automatically pops the ASTATx, ASTATy, and MODE1 registers from the status stack during the return from interrupt instruction (RTI). In one other case, JUMP (CI), the sequencer pops the stack. For more information, see “Interrupt Self-Nesting” on page 4-35. Only the TRQ2-0 and timer expired interrupts cause the sequencer to push an entry onto the status stack. All other interrupts require either explicit saves and restores of effected registers or an explicit push or pop of the stack (PUSH/POP STS).

Pushing the ASTATx, ASTATy, and MODE1 registers preserves the status and control bit settings. This allows a service routine to alter these bits with the knowledge that the original settings are automatically restored upon return from the interrupt.

The top of the status stack contains the current values of ASTATx, ASTATy, and MODE1. Explicit PUSH or POP instructions (not reading and writing these registers) are used move the status stack pointer.

As shown in the following example, do not use (DB) modifier in instructions exiting from TRQx or timer ISRs (RTI; and JUMP (CI)).

```
JUMP ISR_IRQ2;     /* Where ISR_IRQ2 is an address label */
ISR_IRQ2:          /* save MODE1/ASTATx/y registers */
    instruction;
    instruction;
    rti;

    /* re-store MODE1/ASTATx/y registers */
```
**Program Sequencer**

**Status Stack Status**

The status stack is fifteen locations deep. The stack is full when all entries are occupied, is empty when no entries are occupied, and is overflowed if a push occurs when the stack is already full. Bits in the $STKY_x$ register indicate the status stack full and empty states as describe below.

- **Status stack overflow.** Bit 23 ($SSOV$) indicates that the status stack is overflowed (if 1) or not overflowed (if 0)—a sticky bit.

- **Status stack empty.** Bit 24 ($SSEM$) indicates that the status stack is empty (if 1) or not empty (if 0)—not sticky, cleared by a push.

Both $ASTAT_x$ and $ASTAT_y$ register values are pushed/popped regardless of SISD/SIMD mode.

**Instruction Driven Branches**

One type of non-sequential program flow that the sequencer supports is branching. A branch occurs when a `JUMP` or `CALL` instruction moves execution to a location other than the next sequential address. For descriptions on how to use `JUMP` and `CALL` instructions, see Chapter 9, Instruction Set Types, and Chapter 10, Computation Instructions. Briefly, these instructions operate as follows.

- In processors with 5-stage pipelines, the instruction driven branch (`CALL, JUMP, DO UNTIL`) occurs in the address phase on the sequencer while the interrupt (IVT) branch occurs in the Execute phase. This is different from 3-stage pipelines were all branches occur in the Execute stage of the pipeline.

- A `JUMP` or a `CALL` instruction transfers program flow to another memory location. The difference between a `JUMP` and a `CALL` is that a `CALL` automatically pushes the return address (the next sequential address after the `CALL` instruction) onto the PC stack. This push
Variation In Program Flow

makes the address available for the CALL instruction’s matching return instruction, (RTS) in the subroutine, allowing an easy return from the subroutine.

- A RTS instruction causes the sequencer to fetch the instruction at the return address, which is stored at the top of the PC stack. The two types of return instructions are return from subroutine (RTS) and return from interrupt (RTI). While the RTS instruction only pops the return address off the PC stack, the RTI pops the return address and:

1. Clears the interrupt’s bit in the interrupt latch register (IRPTL) and the interrupt mask pointer register (IMASKP). This allows another interrupt to be latched in the IRPTL register and the interrupt mask pointer (IMASKP) register.

2. Pops the status stack if the ASTATx/y and MODE1 status registers have been pushed for the interrupts for the IRQ2-0 signals or for the core timer.

The following are parameters that can be specified for branching instructions.

- JUMP and CALL instructions can be conditional. The program sequencer can evaluate the status conditions to decide whether or not to execute a branch. If no condition is specified, the branch is always taken. For more information on these conditions, see “Interrupt Branch Mode” on page 4-26.

- JUMP and CALL instructions can be immediate or delayed. Because of the instruction pipeline, an immediate branch incurs three lost (overhead) cycles. As shown in Table 4-5 and Table 4-6, the processor aborts the three instructions after the branch, which are in the Fetch1, Fetch2, and Decode stages, while instructions are fetched from the branched address. A delayed branch reduces the overhead to one cycle by allowing the two instructions following
the branch to propagate through the instruction pipeline and execute. For more information, see “Delayed Branches (DB)” on page 4-19.

• JUMP instructions that appear within a loop or within an interrupt service routine have additional options. For information on the loop abort (JA) option, see “Functional Description” on page 4-45. For information on the loop reentry (LR) option, see “Restrictions on Ending Loops” on page 4-55. For information on the clear interrupt (CI) option, see “Interrupt Self-Nesting” on page 4-35.

Direct Versus Indirect Branches

Branches can be direct or indirect. With direct branches the sequencer generates the address while for indirect branches, the PM data address generator (DAG2) produces the address.

Direct branches are JUMP or CALL instructions that use an absolute—not changing at run time—address (such as a program label) or use a PC-relative address. Some instruction examples that cause a direct branch are:

```
CALL fft1024;    /* Where fft1024 is an address label */
JUMP (pc,10);    /* Where (pc,10) is 10-relative addresses after this instruction */
```

Indirect branches are JUMP or CALL instructions that use a dynamic address that comes from the DAG2. For more information on the data address generator, see “Data Address Generators” on page 6-1. Two instruction examples that cause an indirect branch are:

```
JUMP (M8, I12);   /* where (M8, I12) are DAG2 registers */
CALL (M9, I13);   /* where (M9, I13) are DAG2 registers */
```
Variation In Program Flow

Restrictions for VISA Operation

The following should be noted for VISA operation:

- The program counter (PC) now points to short word address space. The PC increments by one, two or three in each cycle depending on the actual size of an instruction (16-bit, 32-bit, or 48-bit).

- Any source files that use hard-coded numbers (as opposed to labels) for branch offsets in the relative offset field will not assemble correctly. What used to be N 48-bit instructions could be a different number of VISA instructions.

The use of absolute addressing in programs is discouraged and these programs should be re-written. For example, the following code sequence that uses absolute addressing will work in traditional ISA operations, but has unexpected behavior if it is not re-written for VISA operation:

```c
I9 = my_jump_table;
M9 = 2;
JUMP (M9, I9);

my_jump_table:
JUMP function0;
JUMP function1;
JUMP function2;
...
```

The value of 2 in the modify register represents a jump of two 48-bit instructions for ISA SHARC processors. In VISA however, this represents two 16-bit locations.

While the instructions themselves may take up more than two 16-bit units, the jump could go to an invalid memory location (not to the start of a valid VISA instruction). Regardless, good programming rules require that such “absolute addressing” be discouraged.
Delayed Branches (DB)

The instruction pipeline influences how the sequencer handles delayed branches (Table 4-5 through Table 4-8). For immediate branches in which JUMP and CALL instructions are not specified as delayed branches (DB), three instruction cycles are lost (NOP) as the instruction pipeline empties and refills with instructions from the new branch.

Branch Listings

As shown in Table 4-5 and Table 4-6, the processor aborts the three instructions after the branch, which are in the Fetch1, Fetch2 and Decode stages. For a CALL instruction, the address of the instruction after the CALL is the return address. During the three lost (no-operation) cycles, the first instruction at the branch address passes through the Fetch2, Decode and address phases of the instruction pipeline.

In the tables that follow, shading indicates aborted instructions, which are followed by NOP instructions.

Table 4-5. Pipelined Execution Cycles for Immediate Branch (Jump or Call)

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>n–2</td>
<td>n–1</td>
<td>n</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>j</td>
</tr>
<tr>
<td>Address</td>
<td>n–1</td>
<td>n</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>j</td>
<td>j+1</td>
</tr>
<tr>
<td>Decode</td>
<td>n</td>
<td>n+1→nop</td>
<td>n+2→nop</td>
<td>n+3→nop</td>
<td>j</td>
<td>j+1</td>
<td>j+2</td>
</tr>
<tr>
<td>Fetch2</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>j</td>
<td>j+1</td>
<td>j+2</td>
<td>j+3</td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+2</td>
<td>n+3</td>
<td>j</td>
<td>j+1</td>
<td>j+2</td>
<td>j+3</td>
<td>j+4</td>
</tr>
</tbody>
</table>

n is the branching instruction and j is the instruction branch address
1. Cycle2: n+1 instruction suppressed
2. Cycle3: n+2 instruction suppressed and for call, n+1 address pushed on, to PC stack
3. Cycle4: n+3 instruction suppressed
### Table 4-6. Pipelined Execution Cycles for Immediate Branch (RTI)

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>n–2</td>
<td>n–1</td>
<td>n</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>r</td>
</tr>
<tr>
<td>Address</td>
<td>n–1</td>
<td>n</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>r</td>
<td>r+1</td>
</tr>
<tr>
<td>Decode</td>
<td>n</td>
<td>n+1 → nop</td>
<td>n+2 → nop</td>
<td>n+3 → nop</td>
<td>r</td>
<td>r+1</td>
<td>r+2</td>
</tr>
<tr>
<td>Fetch2</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>r</td>
<td>r+1</td>
<td>r+2</td>
<td>r+3</td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+2</td>
<td>n+3</td>
<td>r</td>
<td>r+1</td>
<td>r+2</td>
<td>r+3</td>
<td>r+4</td>
</tr>
</tbody>
</table>

n is the branching instruction and r is the instruction at the return address
1. Cycle2: n+1 instruction suppressed
2. Cycle3: n+2 instruction suppressed and r address popped from PC stack
3. Cycle4: n+3 instruction suppressed

### Table 4-7. Pipelined Execution Cycles for Delayed Branch (JUMP or Call)

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>n–2</td>
<td>n–1</td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>nop</td>
<td>j</td>
</tr>
<tr>
<td>Address</td>
<td>n–1</td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>nop</td>
<td>j</td>
<td>j+1</td>
</tr>
<tr>
<td>Decode</td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3 → nop</td>
<td>j</td>
<td>j+1</td>
<td>j+2</td>
</tr>
<tr>
<td>Fetch2</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>j</td>
<td>j+1</td>
<td>j+2</td>
<td>j+3</td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+2</td>
<td>n+3</td>
<td>j</td>
<td>j+1</td>
<td>j+2</td>
<td>j+3</td>
<td>j+4</td>
</tr>
</tbody>
</table>

n is the branching instruction and j is the instruction branch address
1. Cycle3: For call n+3 address pushed on the PC stack
2. Cycle4: n+3 instruction suppressed
In **JUMP** and **CALL** instructions that use the delayed branch (DB) modifier, one instruction cycle is lost in the instruction pipeline. This is because the processor executes the two instructions after the branch and the third is aborted while the instruction pipeline fills with instructions from the new location. This is shown in the sample code below.

```
jump (pc, 3) (db):
instruction 1;
instruction 2;
```

As shown in Table 4-7 and Table 4-8, the processor executes the two instructions after the branch and the third is aborted, while the instruction at the branch address is being processed at the Fetch2, Decode and Address stages of the instruction pipeline. In the case of a **CALL** instruction, the return address is the third address after the branch instruction. While delayed branches use the instruction pipeline more efficiently than immediate branches, delayed branch code can be harder to implement because of the instructions between the branch instruction and the actual branch. This is described in more detail in “Restrictions when Using Delayed Branches” on page 4-22.

### Table 4-8. Pipelined Execution Cycles for Delayed Branch (RTS(db))

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>n−2</td>
<td>n−1</td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>nop</td>
<td>r</td>
</tr>
<tr>
<td>Address</td>
<td>n−1</td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>nop</td>
<td>r</td>
<td>r+1</td>
</tr>
<tr>
<td>Decode</td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>nop</td>
<td>r</td>
<td>r+1</td>
</tr>
<tr>
<td>Fetch2</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>r</td>
<td>r+1</td>
<td>r+2</td>
<td>r+3</td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+2</td>
<td>n+3</td>
<td>r</td>
<td>r+1</td>
<td>r+2</td>
<td>r+3</td>
<td>r+4</td>
</tr>
</tbody>
</table>

n is the branching instruction and r is the instruction at the return address
1. Cycle3: r address popped from PC stack
2. Cycle4: n+3 instruction suppressed
Variation In Program Flow

Atomic Execution of Delayed Branches

Delayed branches and the instruction pipeline also influence interrupt processing. Because the delayed branch instruction and the two instructions that follow it are atomic, the processor does not immediately process an interrupt that occurs between a delayed branch instruction and either of the two instructions that follow. Any interrupt that occurs during these instructions is latched and is not processed until the branch is complete.

This may be useful when two instructions must execute atomically (without interruption), such as when working with semaphores. In the following example, instruction 2 immediately follows instruction 1 in all situations:

```
jump (pc, 3) (db):
instruction 1;
instruction 2;
```

Note that during a delayed branch, a program can read the PC stack register or PC stack pointer register. This read shows the return address on the PC stack has already been pushed or popped, even though the branch has not yet occurred.

IDLE Instruction in Delayed Branch

An interrupt is needed to come out of the IDLE instruction. If a program places an IDLE instruction inside the delayed branch the processor remains in the idled state because interrupts are latched but not serviced until the program exits a delayed branch.

Restrictions when Using Delayed Branches

Besides being more challenging to code, delayed branches impose some limitations that stem from the instruction pipeline architecture. Because the delayed branch instruction and the two instructions that follow it must execute sequentially, the instructions in the two locations that follow a delayed branch instruction cannot be any of those described below.
Development software for the processor should always flag the operations described below as code errors in the two locations after a delayed branch instruction.

Two Subsequent Delayed Branch Instructions

Normally it is not valid to use two conditional instructions using the (DB) option following each other. But the execution is allowed when these instructions are mutually exclusive:

If gt jump (PC, 7) (db);
If le jump (PC, 11) (db);

Other Jumps or Branches

These instructions cannot be used when they follow a delayed branch instruction. This is shown in the following code that uses the JUMP instruction.

jump foo(db);
jump my(db);
r0 = r0+r1;
r1 = r1+r2;

In this case, the delayed branch instruction \( r1 = r1+r2 \), is not executed. Further, the control jumps to \( my \) instead of \( foo \), where the delayed branch instruction is the execution of \( foo \).

The exception is for the JUMP instruction, which applies for the mutually exclusive conditions EQ (equal), and NE (not equal). If the first EQ condition evaluates true, then the NE conditional jump has no meaning and is the same as a NOP instruction as shown below.

if eq jump labell (db);
if ne jump labell (db);
nop;
nop;
Variation In Program Flow

Explicit Pushes or Pops of the PC Stack

In this case a push of the PC stack in a delayed branch is followed by a pop. If a value is pushed in the delayed branch of a call, it is first popped in the called subroutine. This is followed by an RTS instruction.

call foo (db);
push PCSTK;
nop: /* second push due to PCSTK */
foo: /* first push because of call */

This example shows that when a program pushes the PCSTK during a delayed slot, the PC stack pointer is pushed onto the PCSTK.

The following instructions are executed prior to executing the RTS.

pop PCSTK;
RTS (db);
nop;
nop;

If pushing the PC stack, a stack pop must be performed first, followed by an RTS instruction. If a value is popped inside a delayed branch, whatever subroutine return address is pushed is popped back, which is not allowed.

Manipulation of these stacks by using PUSH/POP instructions and explicit writes to these stacks may affect the correct loop function.

Writes to the PCSTK or PCSTKP Registers

The following two situations may arise when programs attempt to write to the PC stack inside a delayed branch.

1. If programs write into the PCSTK inside a jump, one of the following situations can occur.
   a. The PC stack cannot hold a value that has already been pushed onto the PC stack.
Program Sequencer

When the PC stack contains a value and a program writes that same value onto the stack (via PCSTK), the original value is overwritten by the new value of the PCSTK register.

b. The PC stack is empty.

Programs cannot write to the PC stack when they are inside a jump. In this case the PC stack remains empty.

2. Write to the PCSTK inside a call.

If a program writes to the PC stack inside of a call, the value that is pushed onto the PC stack because of that call is overwritten by the value written onto the PC stack. Therefore, when a program performs an RTS, the program returns to the address pushed onto the PC stack and not to the address pushed while branching to the subroutine as shown below.

```
[0x90100]    call foo3 (db);
[0x90101]    PCSTK = 0x90200;
[0x90102]    nop;
[0x90103]    nop;
```

The value 0x90103 is pushed onto the PC stack, while the value 0x90200 is written to the PCSTK register. Accordingly, the value 0x90103 is overwritten by the value 0x90200 in the PC stack because values that are pushed onto the stack have lower priority over values written to PCSTK register. Therefore, when the program executes an RTS, the return address is 0x90200 and not 0x90103.

Operating Mode

This section provides information on the operating mode that controls variations in program flow.
Variation In Program Flow

Interrupt Branch Mode

Interrupts are a special case of subroutines triggered by an event at runtime and are also another type of nonsequential program flow that the sequencer supports. Interrupts may stem from a variety of conditions, both internal and external to the processor. In response to an interrupt, the sequencer processes a subroutine call to a predefined address, called the interrupt vector. The processor assigns a unique vector to each type of interrupt and assigns a priority to each interrupt based on the Interrupt Vector Table (IVT) addressing scheme. For more information, see Appendix B, Core Interrupt Control.

The interrupt controller is enabled by setting the global \texttt{IRPTEN} bit in the \texttt{MODE1} register. The processor supports three prioritized, individually-maskable external interrupts, each of which can be programmed to be either level- or edge-triggered. External interrupts occur when an external device asserts one of the processor’s interrupt inputs (\texttt{IRQ2-0}). The processor also supports internally generated interrupts. An internal interrupt can occur due to arithmetic exceptions, stack overflows, DMA completion and/or peripheral data buffer status, or circular data buffer overflows. Several factors control the processor’s response to an interrupt. When an interrupt occurs, the interrupt is synchronized and latched in the interrupt latch register (\texttt{IRPTL}). The processor responds to an interrupt request if:

- The processor is executing instructions or is in an idle state
- The interrupt is not masked
- Interrupts are globally enabled
- A higher priority request is not pending

When the processor responds to an interrupt, the sequencer branches the program execution with a call to the corresponding interrupt vector address. Within the processor’s program memory, the interrupt vectors are grouped in an area called the interrupt vector table (IVT). The interrupt vectors in this table are spaced at 4-instruction intervals. Longer service
routines can be accommodated by branching to another region of memory. Program execution returns to normal sequencing when the return from interrupt (RTI) instruction is executed. Each interrupt vector has associated latch and mask bits.

The following example uses delayed branches to reduce latency.

ISR_IRQ2:   rti;
            rti;
            rti;
            rti;

ISR_IRQ1:   instruction;   /* IVT branch address */
            jump ISR (db);
            instruction;
            instruction;

ISR_IRQ0:   rti;
            rti;
            rti;
            rti;

**Interrupt Processing Stages**

The processor also has extensive programmable interrupt support. These interrupts are described in the processor specific hardware references.

To process an interrupt, the program sequencer:

1. Outputs the appropriate interrupt vector address.
2. Pushes the current PC value (the return address) onto the PC stack.
3. Automatically pushes the current value of the ASTATx/y and MODE1 registers onto the status stack (only if the interrupt is from IRQ2-0 or the timer).
4. Resets the appropriate bit in the interrupt latch register (IRPTL and LIRPTL registers).
Variation In Program Flow

5. Alters the interrupt mask pointer bits (IMASKP register) to reflect the current interrupt nesting state, depending on the nesting mode. The NESTM bit in the MODE1 register determines whether all the interrupts or only the lower priority interrupts are masked during the service routine.

At the end of the interrupt service routine, the sequencer processes the RTI instruction and performs the following sequence.

1. Returns to the address stored at the top of the PC stack.
2. Pops this value off the PC stack.
3. Automatically pops the status stack (only if the ASTATx,y and MODE1 status registers were pushed for the IRQ2–0, or timer interrupt).
4. Clears the appropriate bit in the interrupt mask pointer register (IMASKP).

Interrupt Categories

The three categories of interrupts are listed below and shown in Figure 4-4.

- Non maskable interrupts (RESET/emulator/boot peripheral)
- Maskable interrupts (core/IO)
- Software interrupts (core)

Except for reset and emulator, all interrupt service routines should end with a RTI instruction. After reset, the PC stack is empty, so there is no return address. The last instruction of the reset service routine should be a JUMP to the start of the main program.
Program Sequencer

Figure 4-4. Interrupt Process Flow

The sequencer supports interrupt masking—latching an interrupt, but not responding to it. Except for the \texttt{RESET} and \texttt{EMU} interrupts, all interrupts are maskable. If a masked interrupt is latched, the processor responds to the latched interrupt if it is later unmasked. Interrupts can be masked globally or selectively. Bits in the \texttt{MODE1}, \texttt{IMASK}, and \texttt{LIRPTL} registers control interrupt masking.

All interrupts are masked at reset except for the non-maskable reset and emulator and boot source. For booting, the processor automatically unmask and uses the interrupt after reset based on the boot configuration pins (\texttt{BOOT_CFGx}).
Variation In Program Flow

Sequencer Interrupt Response

The processor responds to interrupts in three stages:

1. Synchronization (1 cycle)
2. Latching and recognition (1 cycle)
3. Branching to the interrupt vector table (4 instruction cycles)

If the branch is taken from internal memory, the 4 instruction cycles correspond to 4 core clock cycles. If the branch is taken from external memory (ADSP-2137x and ADSP-214xx products) the 4 instruction cycles depend on instruction packing and timing related parameters for the external port (SRAM, SDRAM, DDR2).

Table 4-9, Table 4-10, and Table 4-11 show the pipelined execution cycles for interrupt processing.

Table 4-9. Pipelined Execution Cycles for Interrupt Based During Single Cycle Instruction

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>n–2</td>
<td>n–1</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>v</td>
</tr>
<tr>
<td>Address</td>
<td>n–1</td>
<td>n→nop</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>v</td>
<td>v+1</td>
</tr>
<tr>
<td>Decode</td>
<td>n</td>
<td>n+1→nop</td>
<td>n+2→nop</td>
<td>n+3→nop</td>
<td>v</td>
<td>v+1</td>
<td>v+2</td>
</tr>
<tr>
<td>Fetch2</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>v</td>
<td>v+1</td>
<td>v+2</td>
<td>v+3</td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+2</td>
<td>n+3</td>
<td>v</td>
<td>v+1</td>
<td>v+2</td>
<td>v+3</td>
<td>v+4</td>
</tr>
</tbody>
</table>

2. Cycle2: Interrupt is latched and recognized, but not processed.
3. Cycle3: n is pushed onto PC stack, fetch of vector address starts.
For most interrupts, both internal and external, only one instruction is executed after the interrupt occurs (and four instructions are aborted), before the processor fetches and decodes the first instruction of the service routine. There is also a five cycle latency associated with the IRQ2-0 interrupts.

Table 4-10. Pipelined Execution Cycles for Interrupt During Delayed Branch Instruction

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>n−1</td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>v</td>
</tr>
<tr>
<td>Address</td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>nop</td>
<td>j→nop</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>v</td>
<td>v+1</td>
</tr>
<tr>
<td>Decode</td>
<td>n+1</td>
<td>n+2</td>
<td>nop</td>
<td>j</td>
<td>j+1→nop</td>
<td>nop</td>
<td>j+2→nop</td>
<td>j+3→nop</td>
<td>v</td>
<td>v+1</td>
</tr>
<tr>
<td>Fetch2</td>
<td>n+2</td>
<td>n+3</td>
<td>j</td>
<td>j+1</td>
<td>j+2</td>
<td>j+3</td>
<td>v</td>
<td>v+1</td>
<td>v+2</td>
<td>v+3</td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+3</td>
<td>j</td>
<td>j+1</td>
<td>j+2</td>
<td>j+3</td>
<td>v</td>
<td>v+1</td>
<td>v+2</td>
<td>v+3</td>
<td>v+4</td>
</tr>
</tbody>
</table>

n is the delayed branch instruction, j is the jump address, and v is the interrupt vector.

2. Cycle2: Interrupt is latched and recognized, but not processed.
3. Cycle3: n+3 beyond delay slot, interrupt processing delayed.
5. Cycle5: Interrupt processed.
Variation In Program Flow

Table 4-11. Pipelined Execution Cycles for Interrupt During Instruction With Conflicting PM Data Access (Instruction not Cached)

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>n−2</td>
<td>n−1</td>
<td>n</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>v</td>
</tr>
<tr>
<td>Address</td>
<td>n−1</td>
<td>n</td>
<td>nop</td>
<td>n+1→n</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>v</td>
</tr>
<tr>
<td>Decode</td>
<td>n</td>
<td>n+1→n</td>
<td>nop</td>
<td>n+2→n</td>
<td>nop</td>
<td>n+3→n</td>
<td>nop</td>
<td>n+4→n</td>
<td>v</td>
</tr>
<tr>
<td>Fetch2</td>
<td>n+1</td>
<td>n+2</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>v</td>
<td>v+1</td>
<td>v+2</td>
<td>v+3</td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+2</td>
<td>–</td>
<td>n+3</td>
<td>n+4</td>
<td>v</td>
<td>v+1</td>
<td>v+2</td>
<td>v+3</td>
<td>v+4</td>
</tr>
</tbody>
</table>

n is the conflicting instruction, v is the interrupt vector instruction.
2. Cycle2: Interrupt is latched and recognized, but not processed.
3. Cycle3: PM data access stall cycle, n+3 cached interrupt not processed.
5. Cycle5: n+1 pushed onto PC stack, fetch of vector address starts.

If nesting is enabled and a higher priority interrupt occurs immediately after a lower priority interrupt, the service routine of the higher priority interrupt is delayed until the first instruction of the lower priority interrupt’s service routine is executed. For more information, see “Interrupt Nesting Mode” on page 4-41.

Interrupt Processing

The next several sections discuss the ways in which the SHARC core processes interrupts.
Core Interrupt Sources

According to the IVT table the core supports different groups of interrupts such as:

- Reset – hardware/software
- emulator – debugger, breakpoints, BTC
- core timer – high, low priority
- illegal memory access – forced long word, illegal IOP space
- stack exceptions – PC, Loop, Status
- IRQ2-0 – hardware inputs
- DAGs – Circular buffer wrap around
- Arithmetic exceptions – fixed-point, floating-point
- Software interrupts – programmed exceptions

Note that the interrupt priorities of the core are fixed and cannot be changed.

The interrupt latch bits in the IRPTL register correspond to interrupt mask bits in the IMASK register. (In the LIRPTL register both mask and latch bits are present). In both registers, the interrupt bits are arranged in order of priority. The interrupt priority is from 0 (highest) up to 41 (lowest). Interrupt priority determines which interrupt must be serviced first, when more than one interrupt occurs in the same cycle. Priority also determines which interrupts are nested when the processor has interrupt nesting enabled. For more information, see “Interrupt Nesting Mode” on page 4-41 and Appendix B, Core Interrupt Control.
Variation In Program Flow

Programmable Interrupt Priorities for Peripherals

Peripheral interrupts can be routed to a set of programmable interrupts (18–0). This increases the flexibility across different I/O DMA channels and priorities. For more details see the processor specific hardware reference manual.

Delays in Interrupt Service Routines for Peripherals

Between servicing and returning, the sequencer clears the latch bit of the in-progress ISR every cycle until the \texttt{RTI} (return from interrupt) instruction is executed. When using an ISR, writes into an IOP control register or a buffer to clear the interrupt causes some latency. During this delay, the interrupt may be generated a second time. For more information, see the processor specific hardware reference manual.

Latching Interrupts

When the processor recognizes an interrupt, the processor’s interrupt latch (\texttt{IRPTL} and \texttt{LIRPTL}) registers set a bit (latch) to record that the interrupt occurred. The bits set in these registers indicate interrupts that are currently being latched and are pending for execution. Because these registers are readable and writable, any interrupt except reset (\texttt{RSTI}) and emulator (\texttt{EMUI}) can be set or cleared in software.

Throughout the execution of the interrupt’s service routine, the processor clears the latch bit during every cycle. This prevents the same interrupt from being latched while its service routine is executing. After the \texttt{RTI} instruction, the sequencer stops clearing the latch bit.

If necessary, an interrupt can be reused while it is being serviced. (This is a matter of disabling this automatic clearing of the latch bit.) For more information, see “Interrupt Self-Nesting” on page 4-35.
Interrupt Acknowledge

Every software routine that services core/peripheral interrupts must clear the signalling interrupt request in the respective interrupt channel. The individual channels provide customized mechanisms for clearing interrupt requests. Receive interrupts, for example, are cleared when received data is read from the respective buffer. Transmit requests typically clear when software (or DMA) writes new data into the transmit buffer. These implicit acknowledge mechanisms avoid the need for cycle-consuming software handshakes in streaming interfaces. Sources such as error requests require explicit acknowledge instructions, which are typically performed by clear operations.

For detailed information on core interrupts, see the element specific chapter (for example DAGs). For peripheral interrupts, refer to the processor specific hardware reference manual.

Interrupt Self-Nesting

When an interrupt occurs, the sequencer sets the corresponding bit in the IRPTL register. During execution of the service routine, the sequencer keeps this bit cleared which prevents the same interrupt from being latched while its service routine is executing. If necessary, programs may reuse an interrupt while it is being serviced. Using a jump clear interrupt instruction, (JUMP (CI)) in the interrupt service routine clears the interrupt, allowing its reuse while the service routine is executing.

The JUMP (CI) instruction reduces an interrupt service routine to a normal subroutine, clearing the appropriate bit in the interrupt latch and interrupt mask pointer registers and popping the status stack. After the JUMP (CI) instruction, the processor stops automatically clearing the interrupt’s latch bit, allowing the interrupt to latch again (Figure 4-5).
When returning from a subroutine that was entered with a \texttt{JUMP (CI)} instruction, a program must use a return loop reentry instruction, \texttt{RTS (LR)}, instead of an \texttt{RTI} instruction. For more information, see “Restrictions on Ending Loops” on page 4-55. The following example shows an interrupt service routine that is reduced to a subroutine with the (CI) modifier.

\begin{verbatim}
INSTR1;       /* Interrupt entry from main program*/
JUMP(PC,4) (DB,CI); /* Clear interrupt status*/
INSTR3;
INSTR4;
\end{verbatim}
INSTR5;
INSTR6;
RTS (LR);          /*Use LR modifier with return from subroutine*/

The JUMP (PC,4)(DB,CI) instruction only continues linear execution flow by jumping to the location PC + 4 (INSTR6). The two intervening instructions (INSTR3, INSTR4) are executed and INSTR5 is aborted because of the delayed branch (DB). This JUMP instruction is only an example—a JUMP (CI) can perform a JUMP to any location.

This implementation is useful if two subsequent interrupt events are closer to each other than the execution time of the ISR itself. If self-nesting is not used, the second interrupt event is lost. If used, the ISR itself should be coded atomically, otherwise the second event forces the sequencer to immediately jump to the IVT location.

**Release From IDLE**

The sequencer supports placing the processor in a low power halted state called idle. The processor is in this state until an interrupt occurs. The execution of the ISR releases the processor from the idle state. When executing an IDLE instruction (Figure 4-2 on page 4-4, Table 4-12), the sequencer fetches one more instruction at the current fetch address and then suspends operation. The processor’s internal clock and core timer (if enabled) continue to run while in the idle state. When an interrupt occurs, the processor responds normally after a five cycle latency to fetch the first instruction of the interrupt service routine.

The processor’s I/O processor is not affected by the IDLE instruction. DMA transfers to or from internal memory continue uninterrupted.

The debugger allows you to single step over the IDLE instruction in single step mode. This feature is enabled by the emulator interrupt which is also a valid interrupt to release the processor from the IDLE instruction.
Variation In Program Flow

Certain processor operations that span more than one cycle or which occur at a certain state of the instruction pipeline that involves a change of program flow can delay interrupt processing. If an interrupt occurs during one of these operations, the processor synchronizes and latches the interrupt, but delays its processing. The operations that have delayed interrupt processing are:

- The first of the two cycles used to perform a program memory data access and an instruction fetch (a bus conflict) when the instruction is not cached.
- Any cycle in which the core access of internal memory is delayed due to a conflict with the DMA, or the access to the memory-mapped registers is delayed due to wait states.

Causes of Delayed Interrupt Processing

Table 4-12. Pipelined Execution Cycles for IDLE Instruction

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>n–4</td>
<td>n–3</td>
<td>n–2</td>
<td>n–1</td>
<td>idle</td>
<td>n→</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>v</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>n–3</td>
<td>n–2</td>
<td>n–1</td>
<td>idle</td>
<td>n→</td>
<td>nop</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>v</td>
<td>v+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decode</td>
<td>n–2</td>
<td>n–1</td>
<td>idle</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>v</td>
<td>v+1</td>
<td>v+2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fetch2</td>
<td>n–1</td>
<td>idle</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>v</td>
<td>v+1</td>
<td>v+2</td>
<td>v+3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fetch1</td>
<td>n(idle)</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>v</td>
<td>v+1</td>
<td>v+2</td>
<td>v+3</td>
<td>v+4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cycle 1: IDLE instruction is fetched at n
Cycle 8: interrupt is latched and recognized
Cycle 9: interrupt branch v and (n+1) pushed onto PC stack
Program Sequencer

- A branch (JUMP or CALL) instruction and the following two cycles, whether they are instructions (in a delayed branch) or a NOP (in a non-delayed branch).

- In addition to the above, the cycle in which a branch is in the Address stage of the pipeline along with the last instruction of a counter based loop in the Fetch1 stage.

- The first four of the five cycles used to fetch and execute the first instruction of an interrupt service routine.

- In the case of arithmetic loops, the cycle in which the loop aborts and the following three cycles.

- In the case of counter based loops:
  - The cycle in which the counter-expired condition tests true and the following three cycles in the case of loops having less than four instructions in the body.
  - The cycle in which the DO UNTIL LCE instruction executes and the following cycle for a loop that is composed of one, two or four instructions.

Interrupt Mask Mode

Because the SHARC core supports many different operating modes (SIMD, bit reversal, circular buffer, rounding) it is essential to provide a mechanism whereby the core can change the operating mode without performing an explicit instruction in the ISR such as:

```
BIT SET MODE1 PEYEN|CBUFEN|ALUSAT;
NOP;
```

because this requires instructions and causes longer responses times. To accomplish this, a copy of the MODE1 register is used to mask specific operating modes across interrupts.
Variation In Program Flow

Bits that are set in the MMASK register are used to clear bits in the MODE1 register when the processor’s status stack is pushed. This effectively disables different modes when servicing an interrupt, or when executing a PUSH STS instruction. The processor’s status stack is pushed in two cases:

1. When executing a PUSH STS instruction explicitly in code.
2. When an IRQ2–0 or timer expired interrupt occurs.

For example:

Before the PUSH STS instruction, the MODE1 register enabled the following bit configurations:

- Bit-reversing for register I8
- Secondary registers for DAG2 (high)
- Interrupt nesting
- ALU saturation
- SIMD
- Circular buffering

The system needs to disable ALU saturation, SIMD, and bit-reversing for I8 after pushing the status stack then pushing the MMASK register (these bit locations should = 1).

The value in the MODE1 register after PUSH STS instruction is:

- Secondary registers for DAG2 (high)
- Interrupt nesting enabled
- Circular buffering enabled

The other settings that were previously set in the MODE1 register remain the same. The only bits that are affected are those that are set both in the
Program Sequencer

**MMASK** and in **MODE1** registers. These bits are cleared after the status stack is pushed.

- If the program does not make any changes to the **MMASK** register, the default setting automatically disables SIMD when servicing any of the hardware interrupts mentioned above, or during any push of the status stack.

**Interrupt Nesting Mode**

The sequencer supports interrupt nesting—responding to another interrupt while a previous interrupt is being serviced. Bits in the **MODE1**, **IMASKP**, and **LIRPTL** registers control interrupt nesting as described below.

The **NESTM** bit in the **MODE1** register directs the processor to enable (if 1) or disable (if 0) interrupt nesting.

When interrupt nesting is enabled, a higher priority interrupt can interrupt a lower priority interrupt’s service routine (Figure 4-6). Lower priority interrupts are latched as they occur, but the processor processes them according to their priority after the nested routines finish.

The **IMASKP** bits in the **IMASKP** register and the **MSKP** bits in the **LIRPTL** register list the interrupts in priority order and provide a temporary interrupt mask for each nesting level.

When interrupt nesting is disabled, a higher priority interrupt cannot interrupt a lower priority interrupt’s service routine. Interrupts are latched as they occur and the processor processes them in the order of their priority, after the active routine finishes.
**Variation In Program Flow**

**No Interrupt Nesting (NESTM bit = 0)**

![Diagram showing no interrupt nesting](image)

**Interrupt Nesting (NESTM bit = 1)**

![Diagram showing interrupt nesting](image)

Figure 4-6. Interrupt Nesting
Program Sequencer

Programs should change the interrupt nesting enable (NESTM) bit only while outside of an interrupt service routine or during the reset service routine.

If nesting is enabled and a higher priority interrupt occurs immediately after a lower priority interrupt, the service routine of the higher priority interrupt is delayed. This delay allows the first instruction of the lower priority interrupt routine to be executed, before it is interrupted (Figure 4-6).

When servicing nested interrupts, the processor uses the interrupt mask pointer (IMASKP) to create a temporary interrupt mask for each level of interrupt nesting but the IMASK value is not effected. The processor changes IMASKP each time a higher priority interrupt interrupts a lower priority service routine.

The bits in IMASKP correspond to the interrupts in their order of priority. When an interrupt occurs, the processor sets its bit in IMASKP. If nesting is enabled, the processor uses IMASKP to generate a new temporary interrupt mask, masking all interrupts of equal or lower priority to the highest priority bit set in IMASKP and keeping higher priority interrupts the same as in IMASK. When a return from an interrupt service routine (RTI) is executed, the processor clears the highest priority bit set in IMASKP and generates a new temporary interrupt mask.

The processor masks all interrupts of equal or lower priority to the highest priority bit set in IMASKP. The bit set in IMASKP that has the highest priority always corresponds to the priority of the interrupt being serviced.

The MSKP bits in the LIRPTL register and the entire set of IMASKP registers are for interrupt controller use only. Modifying these bits interferes with the proper operation of the interrupt controller. Furthermore, explicit bit manipulation of any of the bits in the LIRPTL register, while IRPTEN (bit 12 in the MODE1 register) is set, causes an interrupt to be serviced twice.
Loop Sequencer

The main role of the sequencer is to generate the address for the next instruction fetch. In normal program flow, the next fetch address is the previous fetch address plus one (plus three in VISA). When the program deviates from this standard course, (for example with calls, returns, jumps, loops) the program sequencer uses a special logic. In cases of program loops, the sequencer logic:

- Updates the PC stack with the top of loop address.
- Updates the loop stack with the address of the last instruction of the loop.
- Initializes the $\text{LCNTR/CURLCNTR}$ registers and update the loop counter stack, if the loop is counter based (do until 1ce).
- Generates the loop-back (go to the beginning of loop) and loop abort (come out of loop, fetch next instruction from “last instruction of loop plus one” address) signals, according to defined termination condition.
- Generates the abort signals to suppress some of the extra fetched instructions (in case of special loops, some unwanted instructions may get fetched).
- Provides correct instructions (via loop buffer) to the instruction bus (in case of one and two instruction loops).
- Handles interrupts without distorting the intended loop-sequencing (until or unless interrupt service routine deliberately manipulates the status of loop-sequencer resources).
- Handles the branches from within the loop to outside the loop or to some other instruction, within the loop. Updates the loop resources if a branch is paired with an abort option.
Program Sequencer

- Handles the different types of returns from a subroutine and to manage loop-sequencer resources accordingly.
- Provides access to non-loop related instructions (like write, read, push, pop).

Restrictions

There are some restrictions that apply to loop instructions. These restrictions can be classified as general (for example applicable to counter, arithmetic and short loops), or specific (for example arithmetic only, or short loops only).

Functional Description

A loop occurs when a `DO/UNTIL` instruction causes the processor to repeat a sequence of instructions until a condition tests true or indefinite by using `FOREVER` as termination condition. Unlike other processors, the SHARC processors automatically evaluate the loop termination condition and modify the program counter (PC) register appropriately. This allows zero overhead looping.

A `DO UNTIL` instruction may be broadly classified as counter based and arithmetic or indefinite.

Entering Loop Execution

Even though `DO/UNTIL` loops are executed in the Execute stage of the instruction pipeline, the next instruction to be fetched is determined when the `DO/UNTIL` instruction is in the Address stage. This helps to reduce overhead when executing short loops as shown in the following example.

DO/UNTIL Termination:  => pushes loop count onto loop count stack
instruction 1:  => pushes top loop address onto PC stack
instruction 2:  ...

...


**Loop Sequencer**

...  

Instruction n;  => pushes end loop address onto loop address stack

When executing a **DO/UNTIL** instruction, the program sequencer pushes the address of the loop’s last instruction and its termination condition onto the loop address stack. The sequencer also pushes the top-of-loop address, (the address of the instruction following the **DO/UNTIL** instruction), onto the PC stack.

Because of the pipeline, the processor tests the termination condition (and, if the loop is counter-based, decrements the counter) before the end-of-loop is executed so that the next fetch either exits the loop or returns to the top, based on the test condition. If the termination condition is not satisfied, the processor re-fetches the instruction from the top-of-loop address stored on the top of PC stack.

**Terminating Loop Execution**

If the termination condition is true, the sequencer fetches the next instruction after the end of the loop and pops the loop stack and PC stack.

The sequencer’s instruction pipeline architecture influences loop termination. Because instructions are pipelined, the sequencer must test the termination condition and, if the loop is counter based, decrement the counter before the end of the loop. Based on the test’s outcome, the next fetch either exits the loop or returns to the top-of-loop.

The termination condition test occurs when the processor executes the instruction that is four locations before the last instruction in the loop (at location $e - 4$, where $e$ is the end-of-loop address). If the condition tests false, the sequencer repeats the loop and fetches the instruction from the top-of-loop address, which is stored on the top of the PC stack. If the condition tests true, the sequencer terminates the loop and fetches the next instruction after the end of the loop, popping the loop and PC stacks.
Table 4-13 and Table 4-14 show the instruction pipeline states for loop iteration and termination.

### Table 4-13. Pipelined Execution Cycles for Loop Back (Iteration)

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>e–4</td>
<td>e–3</td>
<td>e–2</td>
<td>e–1</td>
<td>e</td>
<td>b</td>
</tr>
<tr>
<td>Address</td>
<td>e–3</td>
<td>e–2</td>
<td>e–1</td>
<td>e</td>
<td>b</td>
<td>b+1</td>
</tr>
<tr>
<td>Decode</td>
<td>e–2</td>
<td>e–1</td>
<td>e</td>
<td>b</td>
<td>b+1</td>
<td>b+2</td>
</tr>
<tr>
<td>Fetch2</td>
<td>e–1</td>
<td>e</td>
<td>b</td>
<td>b+1</td>
<td>b+2</td>
<td>b+3</td>
</tr>
<tr>
<td>Fetch1</td>
<td>e</td>
<td>b</td>
<td>b+1</td>
<td>b+2</td>
<td>b+3</td>
<td>b+4</td>
</tr>
</tbody>
</table>

e is the loop end instruction and b is the loop start instruction
1. Cycle1: Termination condition tests false
2. Cycle2: Top-of-loop address from PC stack

### Table 4-14. Pipelined Execution Cycles for Loop Termination

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>e–4</td>
<td>e–3</td>
<td>e–2</td>
<td>e–1</td>
<td>e</td>
<td>e+1</td>
</tr>
<tr>
<td>Address</td>
<td>e–3</td>
<td>e–2</td>
<td>e–1</td>
<td>e</td>
<td>e+1</td>
<td>e+2</td>
</tr>
<tr>
<td>Decode</td>
<td>e–2</td>
<td>e–1</td>
<td>e</td>
<td>e+1</td>
<td>e+2</td>
<td>e+3</td>
</tr>
<tr>
<td>Fetch2</td>
<td>e–1</td>
<td>e</td>
<td>e+1</td>
<td>e+2</td>
<td>e+3</td>
<td>e+4</td>
</tr>
<tr>
<td>Fetch1</td>
<td>e</td>
<td>e+1</td>
<td>e+2</td>
<td>e+3</td>
<td>e+4</td>
<td>e+5</td>
</tr>
</tbody>
</table>

e is the loop end instruction
1. Cycle1: Termination condition tests true
2. Cycle2: Loop aborts, PC and loop stacks popped
Loop Sequencer

Loop Stack

The loop controller supports a stack that controls saving various loop address and loop counts automatically. This is required for nesting operations including loop abort calls or jumps.

The loop controller uses the loop and program stack for its operation. Manipulation of these stacks by using PUSH/POP instructions and explicit writes to these stacks may affect the correct functioning of the loop.

Loop Address Stack Access

The sequencer’s loop support, shown in Figure 4-2 on page 4-4, includes a loop address stack. The sequencer pushes the termination address, termination code and the loop type information onto the loop address stack when executing a DO/UNTIL instruction. Because the sequencer tests the termination condition four instructions before the end of the loop, the loop stack pops before the end of the loop’s final iteration. If a program reads the LADDR register in these last four instructions, the value is already the termination address for the next loop stack entry.

Loop Address Stack Status

The loop address stack is six levels deep by 32 bits wide. A stack overflow occurs if a seventh entry (one more than full) is pushed onto the loop stack. The stack is empty when no entries are occupied. Because the sequencer keeps the loop stack and loop counter stack synchronized, the same overflow and empty status flags apply to both stacks. These flags are in the sticky status register (STKYx). For more information on STKYx, see Table A-7 on page A-22. For more information on how these flags work with the loop stacks, see “Loop Counter Stack Access” on page 4-49. Note that a loop stack overflow causes a maskable interrupt.
Program Sequencer

Loop Address Stack Manipulation

The LADDR register contains the top entry on the loop address stack. This register is readable and writable over the DM data bus. Reading from and writing to LADDR does not move the loop address stack pointer. Only a stack push or pop performed with explicit instructions moves the stack pointer. The LADDR register contains the value 0xFFFF FFFF when the loop address stack is empty. A write to LADDR has no effect when the loop address stack is empty. “Loop Address Stack Register (LADDR)” on page A-11 lists the bits in the LADDR register.

The PUSH LOOP instruction pushes the stack by changing the pointer only. It does not alter the contents of the loop address stack. Therefore, the PUSH LOOP instruction should be usually followed by a write to the LADDR register. The stack entry pops off the stack four instructions before the end of its loop’s last iteration or on a POP LOOP instruction.

Loop Counter Stack Access

The sequencer’s loop support, shown in Figure 4-2 on page 4-4, also includes a loop counter stack. The loop counter stack is six locations deep by 32 bits wide. The stack is full when all entries are occupied, is empty when no entries are occupied, and is overflowed if a push occurs when the stack is already full. Bits in the STKYx register indicate the loop counter stack full and empty states.

A value of zero in LCNTR causes a loop to execute $2^{32}$ times.

Loop Counter Stack Status

The loop counter stack is six locations deep by 32 bits wide. The stack is full when all entries are occupied, is empty when no entries are occupied, and is overflowed if a push occurs when the stack is already full. Bits in the STKYx register indicate the loop counter stack full and empty states. The stack is full when all entries are occupied, is empty when no entries are occupied, and is overflowed if a push occurs when the stack is already full.
Loop Sequencer

The following bits in the $STKYx$ register indicate the loop counter stack full and empty states.

- **Loop stacks overflowed.** Bit 25 ($LSOV$) indicates that the loop counter stack and loop stack are overflowed (if set to 1) or not overflowed (if set to 0)—$LSOV$ is a sticky bit.

- **Loop stacks empty.** Bit 26 ($LSEM$) indicates that the loop counter stack and loop stack are empty (if set to 1) or not empty (if set to 0)—not sticky, cleared by a $PUSH$.

Table A-7 on page A-22 lists the bits in the $STKYx$ register.

Loop Counter Stack Manipulation

The top entry in the loop counter stack always contains the current loop count. This entry is the $CURLCNTR$ register which is readable and writable by the core. Reading $CURLCNTR$ when the loop counter stack is empty returns the value 0xFFFF FFFF. A write to $CURLCNTR$ has no effect when the loop counter stack is empty.

Writing to the $CURLCNTR$ register does not cause a stack push. If a program writes a new value to $CURLCNTR$, the count value of the loop currently executing is affected. When a $DO$/$UNTIL$ LCE loop is not executing, writing to $CURLCNTR$ has no effect. Because the processor must use $CURLCNTR$ to perform counter based loops, there are some restrictions as to when a program can write to $CURLCNTR$. See “Restrictions on Ending Loops” on page 4-55 for more information.
Counter Based Loops

Counter based loops are comprised of instructions that are set to run a specified number of iterations. These iterations are controlled by the loop counter register (LCNTR). The LCNTR register is a non memory-mapped universal register that is initialized to the count value and the loop counter expired (LCE) instruction is used to check the termination condition. Expiration of LCE signals that the loop has completed the number of iterations as per the count value in LCNTR. Loops that terminate with conditions other than LCE have some additional restrictions. For more information, see "Restrictions on Ending Loops" on page 4-55 and "Restrictions on Short Loops" on page 4-59. For more information on condition types in DO/UNTIL instructions, see “Interrupt Branch Mode” on page 4-26.

Note that the processor’s SIMD mode influences the execution of loops.

The DO/UNTIL instruction uses the sequencer’s loop and condition features, as shown in Figure 4-2 on page 4-4. These features provide efficient software loops without the overhead of additional instructions to branch, test a condition, or decrement a counter. The following code example shows a DO/UNTIL loop that contains three instructions and iterates N times.

```
LCNTR = N, DO the_end UNTIL LCE; /* => push loop count stack, iterates N times */
R0 = DM(I0,M0), R2 = PM(I8,M8);    /* => push return address on PCSTK */
F15 = FLOAT R0;
F1 = F0 - F15;
the_end: F4 = F2 + F3; /* => push Loop address stack */
```

Reading LCNTR in Counter Based Loops

Unlike previous SHARC processors with a 3-stage pipeline, the LCNTR register in 5-stage processors no longer changes value unless explicitly loaded as shown in the following example.
Loop Sequencer

R12 = 0x8;
LCNTR = R12, do (PC,7) until lce;
nop;
nop;
nop;
nop;

dm(I0,MO) = LCNTR;
dm(I0,MO) = LCNTR;
/* 3-stage products: LCNTR is 8 in first 7 iterations, in the last iteration it is 1.
For 5-stage products: LCNTR is always 8. */

IF NOT LCE Condition in Counter Based Loops

During the normal execution of the counter based loop, CURLCNTR is decremented in every iteration of the loop, when the end-of-loop instruction is fetched. Therefore, the NOT LCE condition changes accordingly. Since there are two cycles of latency for the NOT LCE condition to change after CURLCNTR value has changed, an instruction with a branch on the NOT LCE condition also has two cycles of latency. For all other instructions, the latency is one cycle. The following is an example.

LCNTR = <COUNT>, DO End UNTIL LCE:

... Instr(e-4); /* In last iteration CURLCNTR = 1 */
IF NOT LCE CALL (sub1); /* In all iterations branch is taken */
IF NOT LCE CALL (sub2); /* In all iterations branch is taken. However, a non-branch instruction aborts only in the last iteration */
IF NOT LCE <any type>; /* Branch aborts only in the last iteration */
End: Instr(e)

Note that the latency is counted in terms of machine cycles and not in terms of instruction cycles. Therefore, if the pipeline is stalled for some reason (for example for a DMA) the behavior is different from that shown in the example.
Arithmetic Loops

Arithmetic loops are loops where the termination condition in the DO/UNTIL loop is anything other than LCE. In this type of loop, where the body has more than one instruction, the termination condition is checked when the second instruction of the loop body is fetched. In loops that contain a single instruction, the termination condition is checked in every cycle after the DO/UNTIL instruction is executed. An example of arithmetic loop is given below.

R7 = 14;
R6 = 10;
R5 = 6;

DO label UNTIL EQ;
R6 = R6 - 1;
R7 = R7 - 1; /* if fetched EQ condition is tested */
R5 = R5 - 1;
nop;
nop;
Label: nop; /* after loop termination R5 = 0; R6 = 4; R7 = 8; */

If the termination condition tests false, then the next instruction is fetched. If the termination condition tests true, then the instruction following the end-of-loop instruction is fetched in the next cycle and the two instructions currently in the Fetch1 and Fetch2 stages of the instruction pipeline are flushed.

Table 4-15 shows the execution cycles for an arithmetic loop with six instructions.
**Loop Sequencer**

Table 4-15. Pipelined Execution Cycles for Six Instruction Non-Counter Based Loop

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>b</td>
<td>b+1</td>
<td>b+2</td>
<td>b+3</td>
<td>b+4</td>
<td>b+5</td>
<td>nop</td>
<td>nop</td>
<td>b+6</td>
</tr>
<tr>
<td>Address</td>
<td>b+1</td>
<td>b+2</td>
<td>b+3</td>
<td>b+4</td>
<td>b+5</td>
<td>nop</td>
<td>nop</td>
<td>b+6</td>
<td>b+7</td>
</tr>
<tr>
<td>Decode</td>
<td>b+2</td>
<td>b+3</td>
<td>b+4</td>
<td>b+5</td>
<td>b→nop</td>
<td>b+1→nop</td>
<td>b+6</td>
<td>b+7</td>
<td>b+8</td>
</tr>
<tr>
<td>Fetch2</td>
<td>b+3</td>
<td>b+4</td>
<td>b+5</td>
<td>b</td>
<td>b+1</td>
<td>b+6</td>
<td>b+7</td>
<td>b+8</td>
<td>b+9</td>
</tr>
<tr>
<td>Fetch1</td>
<td>b+4</td>
<td>b+5</td>
<td>b</td>
<td>b+1</td>
<td>b+6</td>
<td>b+7</td>
<td>b+8</td>
<td>b+9</td>
<td>b+10</td>
</tr>
</tbody>
</table>

b is the first instruction of the body of the loop and b+6 is the instruction after the loop
1. Cycle2: Loop back, next fetch instruction is b.
2. Cycle4: Termination condition tests true, loop-back aborts, PC and loop stacks popped.

**Indefinite Loops**

A `DO FOREVER` instruction executes a loop indefinitely, until an interrupt or reset intervenes as shown below.

```
DO label UNTIL FOREVER; /* pushed LCNTR onto Loop count stack */
R6 = DM(IO,M0);        /* pushed to PC stack */
R6 = R6 - 1;
IF EQ CALL SUB;
nop;
label: nop;            /* pushed to loop address stack */
```

**VISA-Related Restrictions on Hardware Loops**

The last four instructions of a hardware loop are required to be encoded as traditional 48-bit instructions. Analog Devices VisualDSP++ code-generation tools automatically do this. The contents of this section are provided for information purposes only.

In other words, even if there exists a more efficient VISA equivalent for the same instruction, the traditional opcode still needs to be used for
instructions in the last four instructions of a loop. This is required for two reasons:

- To handle interrupts when the sequencer is fetching and executing the last few instructions.
- To reliably detect the fetch of the last instruction.

The assembler automatically identifies the last four instructions of a hardware loop and treats them appropriately.

In cases of short loops (loops with a body shorter than four instructions), the above rule extends to state that all the instructions in the loop are left uncompressed as shown in the following example.

```
[130000] LCNTR = N, DO the_end UNTIL LCE;
[130001] R0 = R0 + 1; /* short compute */
[130002] R0 = R0 + 1; /* short compute */
[130003] R0 = R0 + 1; /* compute */
[130006] R0 = R0 + 1; /* compute */
[130009] R0 = R0 + 1; /* compute */
[13000C] the_end: R0 = R0 + 1; /* compute */
```

**Restrictions on Ending Loops**

The sequencer’s loop features (which optimize performance in many ways) limit the types of instructions that may appear at or near the end of the loop. These restrictions include:

- Branch (`JUMP` or `CALL`) instructions may not be used as any of the last three instructions of a loop. This *no end-of-loop* branches rule also applies to single instruction, two instruction, and three instruction loops.
Loop Sequencer

- There is a one cycle latency between a multiplier status change and arithmetic loop abort (LA). This extra cycle is a machine cycle, not an instruction cycle. Therefore, if there is a pipeline stall (due to external memory access for example), then the latency is not applicable.

- For counter based loops, an instruction that writes to the current loop counter (CURLCNTR) from memory cannot be used as the fifth-to-last instruction of a counter-based loop (at e–4, where e is the end-of-loop address).

- An IF NOT LCE conditional instruction cannot be used as the instruction that follows a write to CURLCNTR.

- The loop controller uses the loop, and program control stack for its operation. Manipulation of these stacks by using PUSH/POP instructions and explicit writes to these stacks may affect the correct functioning of the loop.

- The IDLE and EMUIDLE instructions should not be used in:
  - Counter based loops of one, two or three instructions
  - The fourth instruction of a counter based loop with four instructions
  - The fifth from last (e–4) instruction of a loop with more than four instructions
  - The last three instructions of any arithmetic loop

Note that any modification of the loop resources, such as the PC stack, loop stack and the CURLCNTR register within the loop may adversely affect the proper functioning of the looping operation and should be avoided. This is applicable even when the program execution branches to an interrupt service routine or a subroutine from within a loop.
Short Counter Based Loops

Short loops are loops that have one, two or three instructions in the body of the loop. Since the body of the loop is less than the depth of the instruction pipeline, short loops tend to have more overhead or lost cycles. Some of the overhead is eliminated by handling these short loops in a special way. The following describes how to minimize or eliminate overhead in short loops.

1. Determine the next fetch address at the start of the loop.

   When the DO/UNTIL instruction is in the address phase of the instruction pipeline, the next fetch address is determined based on the following rule.

   Assuming **DO/UNTIL** is the *nth* instruction:

   a. Fetch n+1 in the next cycle in the case of one and three instruction loops.

   b. Fetch n+2 in the next cycle in the case of a two-instruction loop.

   c. Fetch the next instruction in all other cases.

2. Special handling

   When a DO/UNTIL instruction (n) is in the Address stage of the instruction pipeline, the three instructions following it (n+1, n+2, n+3) are also in the pipeline. In the case of a one-instruction loop, the instructions at the Fetch2 and Fetch1 stages (n+2 and n+3) are not part of the loop body. For two-instruction loops, the instruction at the Fetch1 stage (n+3) is not part of the loop body. The unwanted instructions are eliminated by the following.
Loop Sequencer

a. In the case of one-instruction loop, the instruction (n+1) is held in the Decode stage for two additional cycles to allow the instruction pipeline to complete the first fetch from memory.

b. In the case of two-instruction loop, the processor makes use of a loop buffer. Whenever a DO/UNTIL instruction is detected, the loop buffer is updated with the instruction following it. The instruction from the loop buffer (n+1) is substituted for the instruction (n+3), when it moves to the Decode stage of the instruction pipeline.

Short Arithmetic Based Loops

Short arithmetic based loops terminate differently from short counter based loops. These differences stem from the architecture of the pipeline and the conditional logic as described below.

- In a three instruction loop, the termination condition is checked during the cycle where the second instruction is in the Fetch1 stage of the pipeline (when the top of the loop is executed). If the condition becomes true, the sequencer completes one full pass (after the current pass) of the loop before exiting.

- In a two instruction loop, the termination condition is checked during the cycle where the last (second from top-of-loop) instruction is in the Fetch1 stage of the pipeline. If the condition becomes true when the first instruction is being executed, it tests true during the second instruction as well and one more full pass completes before exiting the loop. If the condition becomes true during the second instruction, two more full passes complete before exiting the loop.
In a one instruction loop, the sequencer tests the termination condition every cycle. After the cycle when the condition becomes true, the sequencer completes three more iterations of the loop before exiting.

The pipeline is never flushed in cases of arithmetic loops for 3-stage processors. Two instructions are always flushed for 5-stage processors to provide backward compatibility.

Restrictions on Short Loops

The sequencer’s instruction pipeline features (which can optimize performance in many ways) restrict how short loops iterate and terminate. Short loops (one, two, or three instruction loops) terminate in a special way because they are shorter than the instruction pipeline. Counter based loops (DO/UNTIL LCE) of one, two, or three instructions are not long enough for the sequencer to check the termination condition four instructions before the end of the loop. In these short loops, the sequencer has already looped back when the termination condition is tested. The sequencer provides special handling to prevent overhead (NOP) cycles if the loop is iterated a minimum number of times. This is described below.

- A loop that contains one instruction must iterate at least four times (only initial stall).
- A loop that contains two instructions must iterate at least two times (only initial stall).
- A loop that contains three instructions must iterate at least two times.

Short loops that iterate less than minimum number of times, incur up to three cycles of overhead, because there can be up to three aborted instructions after the last iteration to clear the instruction pipeline.
**Loop Sequencer**

**Short Loops Listings**

Table 4-16 summarizes all the cases of the loops and the way the termination condition is checked.

Table 4-16. Loop Termination Condition Checks

<table>
<thead>
<tr>
<th>Loop Body</th>
<th>Iteration</th>
<th>Condition Check</th>
<th>Stall Cycles</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 2, 3</td>
<td>CURLCNTR==1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4 and more</td>
<td>CURLCNTR==4</td>
<td>None</td>
<td>Special case</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>CURLCNTR==1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2 and more</td>
<td>CURLCNTR==2</td>
<td>None</td>
<td>Special case</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>CURLCNTR==1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2 and more</td>
<td>CURLCNTR==2</td>
<td>None</td>
<td>Special case</td>
</tr>
<tr>
<td>4 and more</td>
<td>Any</td>
<td>CURLCNTR==1</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

1 The termination condition is always checked when the last instruction of the loop is fetched, (when the instruction that is four instructions before the end-of-loop is executed).

The following sections provide more detail for these types of loops.

**Loop Body – One Instruction**

Table 4-17 through Table 4-21 show the instruction pipeline execution for counter based single instruction loops. Table 4-22 through Table 4-24 show the pipeline execution for counter based two instruction loops. Table 4-25 and Table 4-26 show the pipeline execution for counter based three instruction loops.
**Program Sequencer**

Table 4-17. Pipelined Execution Cycles for Single Instruction Counter
Based Loop With Five Iterations

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>n</td>
<td>n+1</td>
<td>nop</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
</tr>
<tr>
<td>Address</td>
<td>n</td>
<td>n+1</td>
<td>nop</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
</tr>
<tr>
<td>Decode</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
</tr>
<tr>
<td>Fetch2</td>
<td>n+2</td>
<td>n+3</td>
<td>n+3</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td></td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+3</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
</tr>
</tbody>
</table>

n is the loop start instruction and n+2 is the instruction after the loop.
1. Cycle1: Next fetch address determined as n+1. n+1 locked in decode stage.
2. Cycle2: Loop count (LCNTR) equals 5, Decode stalls.
3. Cycle3: n+1 stays in decode, n+1 put into fetch stage.
4. Cycle4: Last instruction fetched, counter expired tests true, n+1 stays in decode.
5. Cycle5: Loop back aborts, PC and Loop stacks popped, the instruction after the loop (n+2) is put in fetch2.

Table 4-18. Pipelined Execution Cycles for Single Instruction Counter
Based Loop With Four Iterations

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>n</td>
<td>n+1</td>
<td>nop</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>n</td>
<td>n+1</td>
<td>nop</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
</tr>
<tr>
<td>Decode</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td></td>
</tr>
<tr>
<td>Fetch2</td>
<td>n+2</td>
<td>n+3</td>
<td>n+3</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+3</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
</tr>
</tbody>
</table>

n is the loop start instruction and n+2 is the instruction after the loop.
1. Cycle1: Next fetch address determined as n+1. n+1 locked in decode stage.
2. Cycle2: Loop count (LCNTR) equals 4, Decode stalls.
3. Cycle3: LCNTR equals 4, n+1 stays in decode, last instruction fetched, counter expired tests true.
4. Cycle4: n+1 stays in decode, loop back aborts, PC and Loop stacks popped, the next instruction after the loop (n+2) is put into fetch.
5. Cycle5: Decode stage updates from fetch2.
**Loop Sequencer**

Table 4-19. Pipelined Execution Cycles for Single Instruction Counter Based Loop With Three Iterations

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>n</td>
<td>n+1</td>
<td>nop</td>
<td>n+1</td>
<td>nop</td>
<td>n+1</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
</tr>
<tr>
<td>Address</td>
<td>n</td>
<td>n+1</td>
<td>nop</td>
<td>n+1</td>
<td>nop</td>
<td>n+1</td>
<td>nop</td>
<td>nop</td>
<td>n+2</td>
</tr>
<tr>
<td>Decode</td>
<td>n+1</td>
<td>n+1</td>
<td>nop</td>
<td>n+1</td>
<td>nop</td>
<td>n+1</td>
<td>nop</td>
<td>nop</td>
<td>n+2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>n+3</td>
<td>n+3</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+3</td>
<td>n+3</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
</tr>
</tbody>
</table>

n is the loop start instruction and n+2 is the instruction after the loop.
1. Cycle1: Next fetch address determined as n+1. n+1 locked in decode stage.
2. Cycle2: Loop count (LCNTR) equals 3, decode stalls.
3. Cycle3: n+1 stays in decode, n+1 put in fetch1 stage.
4. Cycle4: n+1 stays in decode, n+1 put in fetch1 stage.
5. Cycle5: Last instruction fetched, counter expired tests true.

Table 4-20. Pipelined Execution Cycles for Single Instruction Counter Based Loop With Two Iterations

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>n</td>
<td>n+1</td>
<td>nop</td>
<td>n+1</td>
<td>nop</td>
<td>n+1</td>
<td>nop</td>
<td>nop</td>
<td>n+2</td>
</tr>
<tr>
<td>Address</td>
<td>n</td>
<td>n+1</td>
<td>nop</td>
<td>n+1</td>
<td>nop</td>
<td>n+1</td>
<td>nop</td>
<td>nop</td>
<td>n+2</td>
</tr>
<tr>
<td>Decode</td>
<td>n+1</td>
<td>n+1</td>
<td>nop</td>
<td>n+1</td>
<td>nop</td>
<td>n+1</td>
<td>nop</td>
<td>nop</td>
<td>n+2</td>
</tr>
<tr>
<td></td>
<td>n+2</td>
<td>n+3</td>
<td>n+3</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+3</td>
<td>n+3</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
</tr>
</tbody>
</table>

n is the loop start instruction and n+2 is the instruction after the loop.
1. Cycle1: Next fetch address determined as n+1. n+1 locked in decode stage 2.
2. Cycle2: Loop count (LCNTR) equals 2, decode stalls.
3. Cycle3: n+1 stays in decode, n+1 put in fetch1 stage.
4. Cycle4: Last instruction fetched, counter expired tests true.
5. Cycle5: Loop-back aborts, PC and loop stacks popped.
Table 4-21. Pipelined Execution Cycles for Single Instruction Counter Based Loop With One Iteration

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>n</td>
<td>n+1</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>n+2</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>n</td>
<td>n+1</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>n+2</td>
<td>n+3</td>
</tr>
<tr>
<td>Decode</td>
<td>n+1</td>
<td>n+1→nop</td>
<td>n+1→nop</td>
<td>n+1→nop</td>
<td>n+1→nop</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
</tr>
<tr>
<td>Fetch2</td>
<td>n+2</td>
<td>n+3</td>
<td>n+3</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+3</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
</tr>
</tbody>
</table>

n is the loop start instruction and n+2 is the instruction after the loop.
1. Cycle1: Next fetch address determined as n+1.
2. Cycle2: Loop count (LCNTR) equals 1, decode stalls.
3. Cycle3: Last instruction fetched, counter expired tests true.
4. Cycle5: Loop-back aborts, PC and loop stacks popped, n+2 put in fetch1 stage.
Table 4-22. Pipelined Execution Cycles for Two Instruction Counter Based Loop With Three Iterations

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>n</td>
<td>n+1</td>
<td>nop</td>
<td>n+2</td>
<td>n+1</td>
<td>n+2</td>
<td>n+1</td>
<td>n+2</td>
<td>n+2</td>
</tr>
<tr>
<td>Address</td>
<td>n</td>
<td>n+1</td>
<td>nop</td>
<td>n+2</td>
<td>n+1</td>
<td>n+2</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
</tr>
<tr>
<td>Decode</td>
<td>n+1</td>
<td>n+2→nop</td>
<td>n+2</td>
<td>n+1+1</td>
<td>n+2</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
</tr>
<tr>
<td>Fetch2</td>
<td>n+2</td>
<td>n+3</td>
<td>n+3</td>
<td>n+2</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+3</td>
<td>n+2</td>
<td>n+2</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
</tr>
</tbody>
</table>

Note: n is the loop start instruction and n+3 is the instruction after the loop.
1. Cycle1: Next fetch address determined as n+2.
2. Cycle2: Loop count (LCNTR) equals 3, decode stalls.
3. Cycle3: Next fetch address determined as n+1, n+3 and n+2 held in Fetch2 and Fetch1 respectively.
4. Cycle4: n+1 supplied from loop buffer into decode, PC stack supplies top of loop address.
5. Cycle5: Last instruction fetched, counter expired tests true.
Table 4-23. Pipelined Execution Cycles for Two Instruction Counter Based Loop With Two Iterations

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>n</td>
<td>n+1</td>
<td>nop</td>
<td>n+2</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>n</td>
<td>n+1</td>
<td>nop</td>
<td>n+2</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
</tr>
<tr>
<td>Decode</td>
<td>n+1</td>
<td>n+2→nop</td>
<td>n+2</td>
<td>n+1→nop</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
</tr>
<tr>
<td>Fetch2</td>
<td>n+2</td>
<td>n+3</td>
<td>n+3</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+3</td>
<td>n+2</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
<td>n+7</td>
</tr>
</tbody>
</table>

*n is the loop start instruction and n+3 is the instruction after the loop.*

1. Cycle1: Next fetch address determined as n+2.
2. Cycle2: Loop count (LCNTR) equals 2, decode stalls.
3. Cycle3: n+3, and n+2 held in fetch2 and fetch1 respectively counter expired tests true.
4. Cycle4: n+1 supplied from loop buffer into decode, loop-back aborts, PC and loop stacks popped.

Table 4-24. Pipelined Execution Cycles for Two Instruction Counter Based Loop With One Iteration

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>n</td>
<td>n+1</td>
<td>nop</td>
<td>n+2</td>
<td>nop</td>
<td>nop</td>
<td>n+3</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>n</td>
<td>n+1</td>
<td>nop</td>
<td>n+2</td>
<td>nop</td>
<td>n+3</td>
<td>n+4</td>
<td></td>
</tr>
<tr>
<td>Decode</td>
<td>n+1</td>
<td>n+2→nop</td>
<td>n+2</td>
<td>n+3→nop</td>
<td>n+2→nop</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
</tr>
<tr>
<td>Fetch2</td>
<td>n+2</td>
<td>n+3</td>
<td>n+3</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+3</td>
<td>n+2</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
<td>n+7</td>
</tr>
</tbody>
</table>

*n is the loop start instruction and n+3 is the instruction after the loop.*

1. Cycle1: Next fetch address determined as n+2.
2. Cycle2: Loop count (LCNTR) equals 1, decode stalls.
3. Cycle3: Last instruction fetched, counter expired tests true.
Loop Sequencer

# Loop Body - Three Instructions

Table 4-25. Pipelined Execution Cycles for Three Instruction Counter Based Loop With Two Iterations

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
</tr>
<tr>
<td>Address</td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
</tr>
<tr>
<td>Decode</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
</tr>
<tr>
<td>Fetch2</td>
<td>n+2</td>
<td>n+3</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+3</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
<td>n+7</td>
</tr>
</tbody>
</table>

n is the loop start instruction and n+4 is the instruction after the loop.

1. Cycle1: Next fetch address determined as n+1.
2. Cycle2: Loop count (LCNTR) equals 2, fetch address determined by the given rule.
3. Cycle3: Last instruction fetched, counter expired tests true.

Table 4-26. Pipelined Execution Cycles for Three Instruction Counter Based Loop With One Iteration

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>n+4</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>n+4</td>
<td>n+5</td>
</tr>
<tr>
<td>Decode</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
</tr>
<tr>
<td>Fetch2</td>
<td>n+2</td>
<td>n+3</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
<td>n+7</td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+3</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
<td>n+7</td>
<td>n+8</td>
</tr>
</tbody>
</table>

n is the loop start instruction and n+4 is the instruction after the loop.

1. Cycle1: Next fetch address determined as n+1.
2. Cycle2: Loop count (LCNTR) equals 1, fetch address determined by the given rule.
3. Cycle4: Last instruction fetched, counter expired tests true.
Nested Loops

Signal processing algorithms like FFTs and matrix multiplications require nested loops. Nested loop constructs are built using multiple `DO/UNTIL` instructions. If using counter based instructions the following occurs:

Within the loop sequencer, two separate loop counters operate:

- loop counter (`LCNTR`) register has top level entry to loop counter stack
- current loop counter (`CURLCNTR`) iterates in the current loop

The `CURLCNTR` register tracks iterations for a loop being executed, and the `LCNTR` register holds the count value before the loop is executed. The two counters let the processor maintain the count for an outer loop, while a program is setting up the count for an inner loop.

Table 4-27. Pipelined Execution Cycles for Four Instruction Counter Based Loop With One Iteration

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>n</td>
<td>n+1</td>
<td>'nop'</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>n</td>
<td>n+1</td>
<td>'nop'</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
</tr>
<tr>
<td>Decode</td>
<td>n+1</td>
<td>n+2→'nop'</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
<td>n+7</td>
</tr>
<tr>
<td>Fetch2</td>
<td>n+2</td>
<td>n+3</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
<td>n+7</td>
<td>n+8</td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+3</td>
<td>n+4</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
<td>n+7</td>
<td>n+8</td>
<td>n+9</td>
</tr>
</tbody>
</table>

n is the loop start instruction and n+5 is the instruction after the loop
1. Cycle2: Loop count (LCNTR) equals 1, decode stalls
2. Cycle3: Last instruction fetched, Counter expired tests true
3. Cycle4: Loop-back aborts, PC and loop stacks popped
Loop Sequencer

The loop logic decrements the value of CURLCNTR for each loop iteration. Because the sequencer tests the termination condition four instruction cycles before the end of the loop, the loop counter also is decremented before the end of the loop. If a program reads CURLCNTR during these last four loop instructions, the value is already the count for the next iteration.

The loop counter stack is popped four instructions before the end of the last loop iteration. When the loop counter stack is popped, the new top entry of the stack becomes the CURLCNTR value—the count in effect for the executing loop. Two examples of nested loops are shown in Listing 4-1 and Listing 4-2.

Listing 4-1. Nested Counter-Based Loop

```
LCNTR = S, DO the_end UNTIL LCE: /*outer Loop*/
Instruction;
Instruction;
LCNTR = N, DO the_end1 UNTIL LCE: /*inner Loop */
    instruction;
the_end1:instruction; /*inner loop end address */
the_end: instruction; /*outer loop end address*/
```

Listing 4-2. Nested Mixed-Based Loop

```
DO the_end UNTIL EQ; /*outer Loop*/
Instruction;
Instruction;
LCNTR = N, DO the_end1 UNTIL LCE: /*inner Loop */
    instruction;
the_end1:instruction; /*inner loop end address */
Instruction;
the_end: instruction; /*outer loop end address*/
```
Example For Six Nested Loops

A \texttt{DO/UNTIL} instruction pushes the value of \texttt{LCNTR} onto the loop counter stack, making that value the new \texttt{CURLCNTR} value. The following procedure and Figure 4-7 demonstrate this process for a set of nested loops. The previous \texttt{CURLCNTR} value is preserved one location down in the stack.

1. The processor is not executing a loop, and the loop counter stack is empty (\texttt{LSEM} bit =1). The program sequencer loads \texttt{LCNTR} with \texttt{AAAAAAAA}.

2. The processor is executing a single loop. The program sequencer loads \texttt{LCNTR} with the value \texttt{BBBBBBBB} (\texttt{LSEM} bit =0).

3. The processor is executing two nested loops. The program sequencer loads \texttt{LCNTR} with the value \texttt{CCCCCCCC}.

4. The processor is executing three nested loops. The program sequencer loads \texttt{LCNTR} with the value \texttt{DDDDDDDD}.

5. The processor is executing four nested loops. The program sequencer loads \texttt{LCNTR} with the value \texttt{EEEEEEEE}.

6. The processor is executing five nested loops. The program sequencer loads \texttt{LCNTR} with the value \texttt{FFFFFFFF}.

7. The processor is executing six nested loops. The loop counter stack (\texttt{LCNTR}) is full (\texttt{LSOV} bit =1).

A read of \texttt{LCNTR} when the loop counter stack is full results in invalid data. When the loop counter stack is full, the processor discards any data written to \texttt{LCNTR}. 
Loop Sequencer

Restrictions on Ending Nested Loops

The sequencer’s loop features (which optimize performance in many ways) limit the types of instructions that may appear at or near the end of the loop. These restrictions include:

- Nested loops cannot use the same end-of-loop instruction address. The sequencer resolves whether to loop back or not, based on the termination condition. If multiple nested loops end on the same instruction, the sequencer exits all the loops when the termination condition for the current loop tests true. There may be other sequencing errors.

Figure 4-7. Pushing the Loop Counter Stack for Nested Loops
Program Sequencer

- Nested loops with an arithmetic loop as the outer loop must place the end address of the outer loop at least two addresses after the end address of the inner loop.

- Nested loops with an arithmetic based loop as the outer loop that use the loop abort instruction, JUMP (LA), to abort the inner loop, may not use JUMP (LA) to the last instruction of the outer loop.

Loop Abort

The following sections describe different scenarios of how a hardware loop is aborted or interrupted. As previously discussed, instruction and interrupt driven branch mechanisms execute differently, causing different effects for aborting loops.

The hardware for counter-based loops uses the current counter register, CURLCNTR, such that it is decremented when the last instruction of the loop is in the Fetch1 stage of the pipeline. This is done so that branching to the beginning of the loop for the next iteration can occur without wasting any cycles. In the case of a CALL or interrupt, this poses a problem since some instructions are replaced with NOPs before branching to a subroutine or an ISR, and these instructions are fetched again when the control returns. If one of the instructions happens to be the end-of-loop instruction, the CURLCNTR may be decremented twice. To avoid this, after the control returns, the hardware freezes that counter for the number of fetches equal to the number of instructions replaced with NOPs.

Instruction Driven Loop Abort

A special case of loop termination is the loop abort instruction, JUMP (LA). This instruction causes an automatic loop abort when it occurs inside a loop. When the loop aborts, the sequencer pops the PC and loop address stacks once. If the aborted loop was nested, the single pop of the stacks leaves the correct values in place for the outer loop. However, because only one pop is performed, the loop abort cannot be used to jump more than one level of loop nesting as shown in Listing 4-3.
Listing 4-3. Loop Abort Instruction, JUMP (LA)

LCNTR = N, DO the_end UNTIL LCE; /*Loop iteration*/
instruction;
instruction;
instruction;
instruction;
IF EQ JUMP LABEL(LA); /* jump outside of loop */
instruction;
the_end: instruction; /*Last instruction in loop*/

For a branch (call), three instructions in the various stages of the pipeline (Decode through Fetch1) are replaced with NOP instructions. Accordingly, the hardware loop logic freezes the CURLCNTR for three fetch cycles on return from a subroutine. The hardware determines this based on the sequencer executing a RTS instruction. The immediate CALL may be one of the last three instructions of a loop except for one instruction loops, or two instruction one iteration loops as shown in Listing 4-4.

Listing 4-4. Loop Re-entry RTS (LR)

LCNTR = N, DO the_end UNTIL LCE; /*Loop iteration*/
instruction;
instruction;
instruction;
instruction;
CALL SUB; /* call outside of loop */
instruction;
the_end: instruction; /*Last instruction in loop*/

SUB: instruction;
instruction;
instruction;
RTS (LR); /* ensures proper re-entry in loop */
Table 4-28 shows a pipeline where a CALL is in the last but one instruction of a loop. E = end-of-loop instruction, B = top-of-loop instruction.

Table 4-28. CALL in a Loop

<table>
<thead>
<tr>
<th>Execute</th>
<th>Address</th>
<th>Decode</th>
<th>Fetch2</th>
<th>Fetch1</th>
<th>E–2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>CALL</td>
<td>CALL</td>
<td>CALL</td>
<td>CALL</td>
<td>E–2</td>
</tr>
<tr>
<td>SUB</td>
<td>E</td>
<td>E</td>
<td>B</td>
<td>B</td>
<td>E–2</td>
</tr>
<tr>
<td>RTS (LR)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Interrupt Driven Loop Abort

For servicing the interrupt, four instructions in the various stages of the pipeline (Address through Fetch1) are replaced with NOP instructions. Accordingly, the hardware loop logic freezes the CURLCNTR for four fetch cycles on return from an ISR. The hardware determines this based on the sequencer executing a RTI instruction.

Table 4-29 shows a pipeline where an interrupt is being serviced in a loop. E = end-of-loop instruction, B = top-of-loop instruction. E–1 is the return address.
Loop Sequencer

Note that there is one situation where an ISR returns into the loop body using the RTS instruction, when JUMP (CI) is used to convert an ISR to a normal subroutine. Therefore RTS cannot be used to determine that the sequencer branched off to a subroutine or ISR. For this reason, the hardware sets an additional (hidden) bit in PCSTK register, before branching off to an ISR so that on return, either with a RTI or JUMP (CI) + RTS - CURL-CNTR instruction can be frozen for four fetch cycles.

Table 4-29. Pipeline Interrupt in a Loop

<table>
<thead>
<tr>
<th></th>
<th>CCNTR decrement (\downarrow)</th>
<th>Execute</th>
<th>Address</th>
<th>Decode</th>
<th>Fetch2</th>
<th>Fetch1</th>
<th>4 instrs replaced with NOP</th>
<th>ISR returns here</th>
<th>CCNTR frozen for all 4 fetch cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td></td>
<td>E–2</td>
<td>E–1</td>
<td>E–1</td>
<td>E–1</td>
<td>E–1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fetch2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fetch1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(\uparrow) CCNTR decrement</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Loop Abort Restrictions

The last three instructions of a loop may contain an immediate CALL (without a DB modifier) which is paired with a loop re-entry return, (RTS) with the loop reentry modifier (LR). The RTS(LR) instruction ensures that the loop counter is not decremented twice. The immediate CALL may be one of the last three instructions of a loop except for one instruction loops, or two instruction one iteration loops as shown in Listing 4-5.
Listing 4-5. Loop Re-entry RTS (LR)

```
LCNTR = N, DO the_end UNTIL LCE; /*Loop iteration*/
instruction;
instruction;
instruction;
instruction;
CALL SUB; /* call outside of loop */
instruction;
the_end: instruction; /*Last instruction in loop*/

SUB: instruction;
instruction;
instruction;
RTS (LR); /* ensures proper re-entry in loop */
```

Loop Resource Manipulation

In RTOS based systems a fundamental requirement for context switching enforces a save all core registers on the software stack, including the core stack registers.

The SHARC processor prohibits any modification of loop resources, such as the PCSTK, LADDR, and CURLCNTR registers within the loop (including subroutines and ISRs starting from a loop) as doing this may adversely affect the proper function of the looping operation for reasons described below.

Short loops—those with 1, 2, or 3 instructions in the loop body with a small iteration count—are handled differently in hardware from other loops. The exact characterization of the loop, short or otherwise, is determined when the loop startup instruction (DO ... UNTIL termination) is executed and retained during execution of the loop. This start information is not stored in a state register that is popped and pushed along with LADDR/CURLCNTR and PCSTK registers. During normal nesting of the loops
within a short loop, hardware recreates this information based on the stack values. In summary, popping and pushing \texttt{LADDR/CURLCNTR} and \texttt{PCSTK} with new values generally interferes with proper loop function.

However, popping and pushing the loop and PC stack to temporarily vacate the stacks can still be performed such that this information is recreated automatically by following the procedure described in the next section.

**Popping and Pushing Loop and PC Stack Inside an Active Loop**

Use the following sequence to pop and push \texttt{LADDR/CURLCNTR} and \texttt{PCSTK} inside an active loop to temporarily vacate the stacks. A code example is shown in Listing 4-6.

1. Pop \texttt{LOOP} and \texttt{PCSTK} after storing the value of the \texttt{CURLCNTR}, \texttt{LADDR}, and \texttt{PC} registers.

2. Use the empty entry/entries of stacks.

3. Recreate the loops by performing the following steps in the prescribed sequence.
   a. Push \texttt{LOOP} stack.
   b. Load the value of \texttt{CURLCNTR}.
   c. Load the \texttt{LADDR}.
   d. Push the \texttt{PCSTK}.
   e. Load the \texttt{PC} with the stored value.

Sequence a–b–c is critical and therefore must be followed strictly. Any number of unrelated instructions may be executed between the a–b–c sequence.
Listing 4-6. Sequence for Pop and Push of Two-deep Nested Loops

/ *----------Pop and Store--------------*/
R1 = LADDR;
R2 = CURLCNTR;
R3 = PCSTK;
POP LOOP;
POP PCSTK;
NOP;
R4 = LADDR;
R5 = CURLCNTR;
R6 = PCSTK;
POP LOOP;
POP PCSTK;
NOP;
<Store the registers to memory here>
<Miscellaneous instruction/s related/unrelated to hardware loops>
<Load the registers from memory here>
/ *----------Push and Load--------------*/
PUSH LOOP;
CURLCNTR = R5;
LADDR = R4;
PUSH PCSTK;
PCSTK = R6;
PUSH LOOP;
CURLCNTR = R2;
LADDR = R1;
PUSH PCSTK;
PCSTK = R3;

In Listing 4-6, LADDR is restored after CURLCNTR. This ensures that when
LADDR is restored, the correct value of loop count is available. At the time
of LADDR restoration, the hardware recreates the information about the
exact characterization of the loop.
**Loop Sequencer**

**Stack Manipulation Restrictions on ADSP-2136x Processors**

The loop and PC stack registers on the ADSP-2136x processors store some hidden bits in addition to the address. These hidden bits are not readable or writable under software control. The processor sets these hidden bits to indicate the nature of the operation that loaded the `PCSTK` (in the case of a branch or loop). These bits are automatically set to 0 when a write to the `PCSTK` is performed. Because of this, the hidden bits are not restored properly when the `PCSTK` is saved and later restored, even though the address is restored properly.

Therefore, the following functionality is affected when an application saves and restores the PC or loop stack registers.

- A single-instruction arithmetic loop may not work properly after `LADDR/CURLCNTR` restoration.
- An arithmetic loop that contains a branch-related instruction (`CALL/JUMP`) immediately preceding the last instruction of the loop may not work after `PCSTK` restoration.
- After `LADDR/CURLCNTR` restoration, arithmetic loops having `CALL` for the first instruction may not work if the `CALL` is not paired with a `RTS (LR)`.
- Use of the `JUMP (CI) + RTS (LR)` instruction for returning from an ISR to a counter-based loop may not work if the ISR involves saving and restoring the `PCSTK`.

Therefore, in application code that requires that the `LADDR/CURLCNTR` and `PCSTK` be saved and restored, in addition to following the sequence...
Program Sequencer

described in “Popping and Pushing Loop and PC Stack Inside an Active Loop” on page 4-76, observe the following additional precautions.

- Single-instruction arithmetic loops are prohibited.
- The instruction immediately preceding the last instruction of an arithmetic loop may not contain any branches (CALL/JUMP).
- If the application code contains a CALL for its first instruction of an arithmetic loop, it should be paired with the RTS (LR) instruction.
- Re-entry (return) into a counter-based loop after interrupt servicing should be through a RTI instruction. This applies to when the interrupt is cleared inside the ISR.

Cache Control

In this section cache control, which is used for internal and external instruction fetch, is described.

Functional Description

Cache performance (hits) improves if code is executed periodically/repetitively (for example as loops). For linear program flow the cache entries are only filled (misses) and based on the code size cache entries overridden.

Conflict Cache for Internal Instruction Fetch

A sequencer bus conflict occurs when an instruction fetch and a data access are made on the same bus. A block conflict occurs when multiple accesses are made to the same block in internal memory. The following sections describe these memory conflicts in detail. For additional information, see “Memory and Internal Buses Block Diagram (ADSP-21362/3/4/5/6 Only)” on page 7-6.
Cache Control

Instruction Data Bus Conflicts

A bus is comprised of two parts, the address bus and the data bus. Because the bus can be accessed simultaneously by different sources (illustrated in Figure 4-2 on page 4-4), there is a potential risk of bus conflicts.

A bus conflict occurs when the PM data bus, normally used to fetch an instruction in each cycle, is used to fetch an instruction and to access data in the same cycle. Because of the five stage instruction pipeline, if an instruction at the Address stage uses the PM bus to access data it creates a conflict with the instruction fetch at the Fetch1 stage, assuming sequential executions.

Cache Miss

In the instruction \( \text{PM}(Ip, Mq) = \text{UREG} \), the data access over the PMD bus conflicts with the fetch of instruction \( n+2 \) (shown in Table 4-30). In this case the data access completes first. This is true of any program memory data access type instruction. This stall occurs only when the instruction to be fetched is not cached.

Table 4-30. PM Access Conflict

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>( \text{pm}(Ip, Mq) = \text{ureg} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>( \text{pm}(Ip, Mq) = \text{ureg} )</td>
<td>( n )</td>
<td></td>
</tr>
<tr>
<td>Decode</td>
<td>( n )</td>
<td></td>
<td>( n+1 )</td>
</tr>
<tr>
<td>Fetch2</td>
<td>( n+1 )</td>
<td>( n+2 )</td>
<td>( n+3 )</td>
</tr>
<tr>
<td>Fetch1</td>
<td>( n+2 )</td>
<td>( n+2 )</td>
<td>( n+3 )</td>
</tr>
</tbody>
</table>

1. Cycle1: \( n+2 \) Instruction fetch postponed
2. Cycle2: Stall Cycle

Note that the cache stores the fetched instruction \( n+2 \), not the instruction requiring the program memory data access.
When the processor first encounters a bus conflict, it must stall for one cycle while the data is transferred, and then fetch the instruction in the following cycle. To prevent the same delay from happening again, the processor automatically writes the fetched instruction to the cache. The sequencer checks the instruction cache on every data access using the PM bus. If the instruction needed is in the cache, a cache hit occurs. The instruction fetch from the cache happens in parallel with the program memory data access, without incurring a delay.

If the instruction needed is not in the cache, a cache miss occurs, and the instruction fetch (from memory) takes place in the cycle following the program memory data access, incurring one cycle of overhead. The fetched instruction is loaded into the cache (if the cache is enabled and not frozen), so that it is available the next time the same instruction (that requires program memory data) is executed.

Figure 4-8 shows a block diagram of the 2-way set associative instruction cache. The cache holds 32 instruction-address pairs. These pairs (or cache entries) are arranged into 16 (15–0) cache sets according to the four least significant bits (3–0) of their address. The two entries in each set (entry 0 and entry 1) have a valid bit, indicating whether the entry contains a valid instruction. The least recently used (LRU) bit for each set indicates which entry was not placed in the cache last (0 = entry 0 and 1 = entry 1).

The cache places instructions in entries according to the four LSBs of the instruction’s address. When the sequencer checks for an instruction to fetch from the cache, it uses the four address LSBs as an index to a cache set. Within that set, the sequencer checks the addresses of the two entries as it looks for the needed instruction. If the cache contains the instruction, the sequencer uses the entry and updates the LRU bit (if necessary) to indicate the entry did not contain the needed instruction.
When the cache does not contain a needed instruction, it loads a new instruction and address and places them in the least recently used entry of the appropriate cache set. The cache then toggles the LRU bit, if necessary.

### Instruction Cache for External Instruction Fetch

As previously discussed, the cache only generates misses during conflicts on the internal PMD instruction data bus (conflict cache in previous generation SHARCs).

However, in the newer SHARC processors (from the introduction of the ADSP-2137x processors) the cache operation is enhanced to operate as a true instruction cache. For every external instruction fetch (regardless of conflict with the DMD or PMD bus) the cache checks for a hit condition. This ensures better performance since all instructions are loaded into the cache.

---

**Figure 4-8. Instruction Cache Architecture**

<table>
<thead>
<tr>
<th>SET 0</th>
<th>ENTRY 0</th>
<th>LRU BIT</th>
<th>VALID BIT</th>
<th>INSTRUCTIONS</th>
<th>ADDRESSES BITS (23-4)</th>
<th>ADDRESSES BITS (3-0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ENTRY 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SET 1</th>
<th>ENTRY 0</th>
<th>LRU BIT</th>
<th>VALID BIT</th>
<th>INSTRUCTIONS</th>
<th>ADDRESSES BITS (23-4)</th>
<th>ADDRESSES BITS (3-0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ENTRY 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SET 2</th>
<th>ENTRY 0</th>
<th>LRU BIT</th>
<th>VALID BIT</th>
<th>INSTRUCTIONS</th>
<th>ADDRESSES BITS (23-4)</th>
<th>ADDRESSES BITS (3-0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ENTRY 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0010</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SET 13</th>
<th>ENTRY 0</th>
<th>LRU BIT</th>
<th>VALID BIT</th>
<th>INSTRUCTIONS</th>
<th>ADDRESSES BITS (23-4)</th>
<th>ADDRESSES BITS (3-0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ENTRY 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1101</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SET 14</th>
<th>ENTRY 0</th>
<th>LRU BIT</th>
<th>VALID BIT</th>
<th>INSTRUCTIONS</th>
<th>ADDRESSES BITS (23-4)</th>
<th>ADDRESSES BITS (3-0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ENTRY 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1110</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SET 15</th>
<th>ENTRY 0</th>
<th>LRU BIT</th>
<th>VALID BIT</th>
<th>INSTRUCTIONS</th>
<th>ADDRESSES BITS (23-4)</th>
<th>ADDRESSES BITS (3-0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ENTRY 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1111</td>
</tr>
</tbody>
</table>
Program Sequencer

cache for a miss and executed internally for the next hit. For more information, see the processor specific hardware reference manual.

Block Conflicts

A block conflict occurs when multiple data accesses are made to the same block in memory from which the instructions are executed. For more information, see Chapter 7, Memory.

Block conflicts are not cached.

Caching Instructions

The caching of instructions happens in the Fetch and Decode stages of the instruction pipeline.

- **Fetch1 Stage** – The core launches the instruction fetch address in the Fetch1 stage. In this stage, the PM address is matched with the existing addresses in the cache. If the address is found in the cache, then a cache hit occurs, else a cache miss occurs. In case of a cache miss, the PM address is loaded into the cache in this stage.

  For execution from internal memory, the PM address matching happens only when the instruction fetch conflicts with a PM data access (PMD).

  For execution from external memory, the address is matched for all instructions that are fetched.

- **Fetch2 Stage** – In case of a cache miss, the instruction data is driven by the memory PMD in this stage. In the case of a cache hit, the instruction PMD is read out from the cache in this stage.

- **Decode Stage** – In case of a cache miss, the instruction read from the 48-bit PMD memory in the Fetch2 stage is loaded into the cache in this stage.
Cache Control

Table 4-31, Table 4-32 and Table 4-33 illustrate the pipeline versus cache operation.

Table 4-31. Cache Miss – Internal Memory Execution

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td></td>
<td></td>
<td></td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>n</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
</tr>
<tr>
<td>Decode</td>
<td>n</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
</tr>
<tr>
<td>Fetch2</td>
<td>n+1</td>
<td>n+2</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
</tr>
</tbody>
</table>

If the cache hit immediately follows a cache miss of the same address (Table 4-33), then the instruction would not have been loaded into the cache by then. In this case, the instruction is driven directly from the input instruction load bus of the cache instead of the cache itself.

Table 4-32. Cache Hit – Internal Memory Execution

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td></td>
<td></td>
<td></td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>n</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
</tr>
<tr>
<td>Decode</td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
</tr>
<tr>
<td>Fetch2</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
</tr>
</tbody>
</table>

\[\text{Add Match}\] \[\text{Instr Read from Cache}\] (n+3) \[\text{Instr Load}\] (n+3)

4-84 SHARC Processor Programming Reference
Table 4-33. Cache Miss Followed by Cache Hit to Same Address

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>n(PMDA)</td>
<td>n+1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>n(PMDA)</td>
<td>n+1</td>
<td>n+2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decode</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3*</td>
<td></td>
</tr>
<tr>
<td>Fetch2</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+3*</td>
<td></td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+3</td>
<td>n+3*</td>
<td></td>
</tr>
</tbody>
</table>

* Same address as previous instruction
** Here the instruction has not yet been loaded into the cache, so the instruction is read from the instruction load bus of the cache instead of the cache itself.

Table 4-34 and Table 4-35 illustrate the pipeline versus cache operation in external memory.

Table 4-34. Cache Miss – External Memory Execution

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>n–2</td>
<td>n–2</td>
<td>n–1</td>
<td>n–1</td>
<td>n–1</td>
<td>n</td>
</tr>
<tr>
<td>Address</td>
<td>n–1</td>
<td>n–1</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n+1</td>
</tr>
<tr>
<td>Decode</td>
<td>n</td>
<td>n</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
</tr>
<tr>
<td>Fetch2</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+2</td>
<td>n+2</td>
<td>n+3</td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+2</td>
<td>n+2</td>
<td>n+3</td>
<td>n+3</td>
<td>n+3</td>
<td>n+4</td>
</tr>
</tbody>
</table>

* Add Match (n+2)
| Add Load (n+2)
| Add Match (n+3)
| Add Load (n+3)
| Add Instr (n+2)
| Add Load (n+4)
Cache Control

Table 4-35. Cache Hit – External Memory Execution

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td></td>
<td></td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
</tr>
<tr>
<td>Address</td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td></td>
</tr>
<tr>
<td>Decode</td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
</tr>
<tr>
<td>Fetch2</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
</tr>
</tbody>
</table>

Cache Invalidate Instruction

The **FLUSH CACHE** instruction allows programs to explicitly invalidate the cache content by clearing all valid bits. The execution of the **FLUSH CACHE** instruction is independent of the cache enable bit in the **MODE2** register.

The **FLUSH CACHE** instruction has a latency of one cycle. Using an instruction that contains a PM data access immediately following a **FLUSH CACHE** instruction is prohibited.

This instruction is required in systems using software overlay programming techniques. With these overlays, software functions are loaded via DMA during runtime into the internal RAM. Since the cache entries are still valid from any previous function, it is essential to flush all the valid cache entries to prevent system crashes. Note that the **FLUSH CACHE** instruction has a 1 cycle instruction latency while executing from internal memory and a 2 cycle instruction latency while executing from external memory.
**Program Sequencer**

**Operating Modes**

The following sections describe the cache operating modes.

After power-up and or reset, the cache content is not predictable in that it may contain valid/invalid instructions, be unfrozen and enabled. However, all LRU and valid bits are cleared. So after a processor power-up or reset, the cache performs only cache miss-cache entry until the same entry causes later hits.

**Cache Restrictions**

The following restrictions on cache use should be noted.

- If the cache freeze bit of the \texttt{MODE2} register is set by instruction \( n \), then this feature is effective from the \( n+2 \) instruction onwards. This results from the effect latency of the \texttt{MODE2} register.

- When a program changes the cache mode, an instruction containing a program memory data access must not be placed directly after a cache enable or cache disable instruction. This is because the processor must wait at least one cycle before executing the PM data access. A program should have a \texttt{NOP} (no operation) or other non-conflicting instruction inserted after the cache enable or cache disable instruction.

**Cache Disable**

The cache disable bit (bit 4, \texttt{CADIS}) directs the sequencer to disable the cache (if 1) or enable the cache (if 0).

Note that the cache content stays valid when the cache is disabled as an already cached instruction does not generate a cache hit. The same instruction does generate a hit and can be taken from the cache after the cache is enabled.
Cache Control

Cache External Memory Disable (ADSP-214xx)

The cache disable external memory bit (bit 6, EXT CADIS) directs the sequencer to disable the cache for external memory (if 1) or enable the cache (if 0).

If this bit is set, only external instruction fetches are not cached, the internal cache operates independent from this bit setting.

Cache Freeze

The cache freeze bit (bit 19, CAFRZ) directs the sequencer to freeze the contents of the cache (if 1) or let new entries displace the entries in the cache (if 0).

Freezing the cache prevents any changes to its contents—a cache miss does not result in a new instruction being stored in the cache. Disabling the cache stops its operation completely—all instruction fetches conflicting with program memory data accesses are delayed. These functions are selected by the CADIS (cache enable/disable) and CAFRZ (cache freeze) bits in the MODE2 register.

Cache Efficiency

Cache operation is usually efficient and requires no intervention. However, certain ordering in the sequence of instructions can work against the cache’s architecture, reducing its efficiency. When the order of PM data accesses and instruction fetches continuously displaces cache entries and loads new entries, the cache does not operate efficiently. Rearranging the order of these instructions remedies this inefficiency. Optionally, a dummy PM read can be inserted to trigger the cache.

When a cache miss occurs, the needed instruction is loaded into the cache so that if the same instruction is needed again, it will be available (that is, a cache hit will occur). However, if another instruction whose address is mapped to the same set displaces this instruction and loads a new
instruction, a cache miss occurs. The LRU bits help to reduce the occurrence of a cache miss since at least two other instructions, mapped to the same set, are needed before an instruction is displaced. If three instructions mapped to the same set are all needed repeatedly, cache efficiency (that is, the cache hit rate) can go to zero. To keep this from happening, move one or more instructions to a new address that is mapped to a different cache set.

An example of inefficient cache code appears in Table 4-36. The PM bus data access at address 0x101 in the loop, OUTER, causes a bus conflict and also causes the cache to load the instruction being fetched at 0x104 (into set 4). Each time the program calls the subroutine, INNER, the program memory data accesses at 0x201 and 0x211 displace the instruction at 0x104 by loading the instructions at 0x204 and 0x214 (also into set 4).

If the program rarely calls the INNER subroutine during the OUTER loop execution, the repeated cache loads do not greatly influence performance. If the program frequently calls the subroutine while in the loop, cache inefficiency has a noticeable effect on performance. To improve cache efficiency on this code (if for instance, execution of the OUTER instruction of the loop is time critical), rearrange the order of some instructions. Moving the subroutine call up one location (starting at 0x201) also works. By using that order, the two cached instructions end up in cache set 5, instead of set 4.

Table 4-36. Cache Inefficient Code

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0100</td>
<td>Lcntr = 1024, do Outer until LCE;</td>
</tr>
<tr>
<td>0x0101</td>
<td>r0 = dm(i0,m0), pm(i8,m8) = f3;</td>
</tr>
<tr>
<td>0x0102</td>
<td>f2 = float r1;</td>
</tr>
<tr>
<td>0x0103</td>
<td>f3 = f2 * f2;</td>
</tr>
<tr>
<td>0x0104</td>
<td>if eq call (Inner);</td>
</tr>
<tr>
<td>0x0105</td>
<td>r1 = r0-r15;</td>
</tr>
</tbody>
</table>
I/O Flags

Table 4-36. Cache Inefficient Code (Cont’d)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0106</td>
<td>Outer: $f3 = f3 + f4$;</td>
</tr>
<tr>
<td>0x0107</td>
<td>pm(i8,m8) = f3;</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x0200</td>
<td>Inner: r1 = R13;</td>
</tr>
<tr>
<td>0x0201</td>
<td>r14 = pm(i9,m9);</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x0211</td>
<td>pm(i9,m9) = r12;</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x021F</td>
<td>rts;</td>
</tr>
</tbody>
</table>

I/O Flags

There are 16 general-purpose I/O flags in SHARC processors. Each `FLAG` pin (3–0) has four dedicated signals. All flag pins can be multiplexed with parallel/external port pins. The `FLAG4-15` pins are also accessible to the signal routing unit (SRU). A flag pin can be routed to a DAI/DPI pin and therefore operate in parallel to the parallel/external port. Refer to the product specific hardware reference manual for more information.

Programs cannot change the output selects of the `FLAGS` register and provide a new value in the same instruction. Instead, programs must use two write instructions—the first to change the output select of a particular `FLAG` pin, and the second to provide the new value as shown below.

```c
bit set flags FLG20;  /* set flag2 as output */
bit clr flags FLG2;   /* set flag2 output low */
```
Program Sequencer

The FLAGS register is used to control all FLAG15-0 pins. Based on FLAG register effect latency and internal timings there must be at least 4 wait states in order to toggle the same flag correctly as shown in the following example. For more information refer to the specific product data sheet.

```c
bit tgl flags FLG2;
nop; nop; nop; nop;   /* wait 4 cycles */
bit tgl flags FLG2;
nop; nop; nop; nop;   /* wait 4 cycles */
bit tgl flags FLG2;
```

Conditional Instruction Execution

Conditional instructions provide many options for program execution which are discussed in this section. There are three types of conditional instructions:

- Conditional compute (ALU/Multiplier/Shifter)
- Conditional data move (reg-to-reg, reg-to-memory)
- Conditional branch (direct branch, indirect branch)

If the condition is evaluated as true, the operation is performed, if it is false, it gets aborted as shown in the example below.

```c
R10 = R12-R13;
If LT R0=R1+R2;  /* if ALU less than zero, do computation */
```

If an if-then-else construct is used, the else evaluates the inverse of the if condition:

```c
R10 = R12-R13;
If LT CALL SUB, ELSE R0=R1+R2;    /* do computation if condition is false */
```

The processor records status for the PEx element in the ASTATx and STKYx registers and the PEy element in the ASTATy and STKYy registers.
IF Conditions with Complements

Each condition that the processor evaluates has an assembler mnemonic. The condition mnemonics for conditional instructions appear in Table 4-37. For most conditions, the sequencer can test both true and false (complement) states. For example, the sequencer can evaluate ALU equal-to-zero (EQ) and its complement ALU not-equal-to-zero (NE).

Note that since the IF condition is optional and if it is not placed in the instruction the condition is always true.

Table 4-37. IF Condition 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>
</tbody>
</table>
# Program Sequencer

## Program Sequencer

**Table 4-37. IF Condition 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>Shifter</td>
<td>Shifter overflow</td>
<td>SV = 1</td>
<td>SV</td>
</tr>
<tr>
<td>Shifter overflow</td>
<td>SV = 1</td>
<td>NOT SV</td>
<td></td>
</tr>
<tr>
<td>Shifter zero</td>
<td>SZ = 1</td>
<td>NOT SZ</td>
<td></td>
</tr>
<tr>
<td>Shifter not zero</td>
<td>SZ = 0</td>
<td>NOT SZ</td>
<td></td>
</tr>
<tr>
<td>Shifter bit FIFO overflow</td>
<td>SF = 1</td>
<td>SF</td>
<td></td>
</tr>
<tr>
<td>Shifter bit FIFO not overflow</td>
<td>SF = 0</td>
<td>NOT SF</td>
<td></td>
</tr>
<tr>
<td>System Register</td>
<td>Bit test flag true</td>
<td>BTF = 1</td>
<td>TF</td>
</tr>
<tr>
<td>Flag 0 asserted</td>
<td>Flag0 = 1</td>
<td>NOT FLAG0_IN</td>
<td></td>
</tr>
<tr>
<td>Flag 0 not asserted</td>
<td>Flag0 = 0</td>
<td>NOT FLAG0_IN</td>
<td></td>
</tr>
<tr>
<td>Flag 1 asserted</td>
<td>Flag1 = 1</td>
<td>NOT FLAG1_IN</td>
<td></td>
</tr>
<tr>
<td>Flag 1 not asserted</td>
<td>Flag1 = 0</td>
<td>NOT FLAG1_IN</td>
<td></td>
</tr>
<tr>
<td>Flag 2 asserted</td>
<td>Flag2 = 1</td>
<td>NOT FLAG2_IN</td>
<td></td>
</tr>
<tr>
<td>Flag 2 not asserted</td>
<td>Flag2 = 0</td>
<td>NOT FLAG2_IN</td>
<td></td>
</tr>
<tr>
<td>Flag 3 asserted</td>
<td>Flag3 = 1</td>
<td>NOT FLAG3_IN</td>
<td></td>
</tr>
<tr>
<td>Flag 3 not asserted</td>
<td>Flag3 = 0</td>
<td>NOT FLAG3_IN</td>
<td></td>
</tr>
<tr>
<td>Loop Sequencer</td>
<td>Loop counter not expired</td>
<td>CURLCNTR ≠ 1</td>
<td>NOT LCE6</td>
</tr>
<tr>
<td>External Port Bus</td>
<td>Bus master true</td>
<td>The CSEL bits 18–17 in the MODE1 register must =0, otherwise the condition is always evaluated as false</td>
<td>BM</td>
</tr>
<tr>
<td>(ADSP-21368, ADSP-2146x)</td>
<td>Bus master false</td>
<td>Not BM</td>
<td></td>
</tr>
</tbody>
</table>

1 ALU greater than (GT) is true if: \( \overline{AF} \) and \( (AN \oplus (AV \land ALUSAT)) \) or \( (AF \land AN) \) or \( AZ = 0 \)
2 ALU less than (LT) is true if: \( \overline{AF} \) and \( (AN \oplus (AV \land ALUSAT)) \) or \( (AF \land AN \land AZ) \) = 1
3 ALU greater equal (GE) is true if: \( \overline{AF} \) and \( (AN \oplus (AV \land ALUSAT)) \) or \( (AF \land AN \land AZ) \) = 0
4 ALU lesser or equal (LT) is true if: \( \overline{AF} \) and \( (AN \oplus (AV \land ALUSAT)) \) or \( (AF \land AN) \) or \( AZ = 1 \)
5 For ADSP-214xx processors and later.
6 Does not have a complement.
Conditional Instruction Execution

DO/UNTIL Terminations Without Complements

Programs should use \texttt{FOREVER} and \texttt{LCE} to specify loop (DO/UNTIL) termination. A \texttt{DO FOREVER} instruction executes a loop indefinitely, until an interrupt or reset intervenes. There are some restrictions on how programs may use conditions in DO/UNTIL loops. For more information, see “Restrictions on Ending Loops” on page 4-55 and “Restrictions on Short Loops” on page 4-59.

Table 4-38. 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>Loop Sequencer</td>
<td>Loop counter expired</td>
<td>CURLCNTR = 1</td>
<td>LCE</td>
</tr>
<tr>
<td></td>
<td>Always false (Do)</td>
<td>Always</td>
<td>FOREVER</td>
</tr>
</tbody>
</table>

Operating Modes

The following sections describe the operating modes for conditional instruction execution.

Conditional Instruction Execution in SIMD Mode

Because the two processing elements can generate different outcomes, the sequencer must evaluate conditions from both elements (in SIMD mode) for conditional (IF) instructions and loop (DO/UNTIL) terminations. The processor records status for the PEx element in the \texttt{ASTATx} and \texttt{STKYx} registers and the PEy element in the \texttt{ASTATy} and \texttt{STKYy} registers.

Even though the processor has dual processing elements PEx and PEy, the sequencer does not have dual sets of stacks.

The sequencer has one PC stack, one loop address stack, and one loop counter stack. The status bits for stacks are in the \texttt{STKYx} register and are not duplicated in the \texttt{STKYy} register.
The processor handles conditional execution differently in SISD versus SIMD mode. There are three ways that conditionals differ in SIMD mode. These are described below and in Table 4-39.

- In conditional computation and data move (IF ... compute/move) instructions, each processing element executes the computation/move based on evaluating the condition in that processing element. See Chapter 9, Instruction Set Types for coding information.

- In conditional branch (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 indirect branch (if ... pc, reladdr/Md, Ic) instructions with an ELSE clause, each processing element executes the ELSE computation/data move based on evaluating the inverse of the condition (NOT IF) in that processing element.

Table 4-39. Conditional SIMD Execution Summary

<table>
<thead>
<tr>
<th>Conditional Operation</th>
<th>Conditional Outcome Depends On …</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Operations</td>
<td>Executes in each PE independently depending on condition test in each PE</td>
</tr>
<tr>
<td>Register-to-register Move</td>
<td></td>
</tr>
<tr>
<td>UREG/CUREG to UREG/CUREG (from complementary pair to complementary pair)</td>
<td>Executes move in each PE (and/or memory) independently depending on condition test in each PE</td>
</tr>
<tr>
<td>UREG to UREG/CUREG (from uncomplementary register to complementary pair)</td>
<td>Executes move in each PE (and/or memory) independently depending on condition test in each PE;</td>
</tr>
</tbody>
</table>
Conditional Instruction Execution

Table 4-39. Conditional SIMD Execution Summary (Cont’d)

<table>
<thead>
<tr>
<th>Conditional Operation</th>
<th>Conditional Outcome Depends On …</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register-to-memory Move</td>
<td>DAG post-modify</td>
</tr>
<tr>
<td></td>
<td>Executes memory move depending on ORing condition test on both PE’s</td>
</tr>
<tr>
<td></td>
<td>DAG pre-modify</td>
</tr>
<tr>
<td></td>
<td>Pre-modify operations always occur independent of the conditions</td>
</tr>
<tr>
<td>Branches and Loops</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Executes in sequencer depending on ANDing condition test on both PEs</td>
</tr>
</tbody>
</table>

1 Complementary pairs are registers with SIMD complements, include PEx/y data registers and USTAT1/2, USTAT3/4, ASTATx/y, STKYx/y, and PX1/2 Uregs.
2 Uncomplementary registers are Uregs that do not have SIMD complements.

Bit Test Flag in SIMD Mode

In SIMD mode, two independent bit tests can occur from individual registers as shown in the following example.

```c
bit set model PEYEN;
nop;
r2=0x80000000;
ustat1=r2;
bit TST ustat1 BIT_31; /* test bit 31 in ustat1/ustat2 */
if TF call SUB;       /* branch if both cond are true */
if TF r10=r10+1;      /* compute on any cond */
```

Conditional Compute

While in SIMD mode, a conditional compute operation can execute on both processing elements, either element, or neither element, depending on the outcome of the status flag test. Flag testing is independently performed on each processing element.
Conditional Data Move

The execution of a conditional (IF) data move (register-to-register and register-to/from-memory) instruction depends on three factors:

- The explicit data move depends on the evaluation of the conditional test in the PEx processing element.
- The implicit data move depends on the evaluation of the conditional test in the PEy processing element.
- Both moves depend on the types of registers used in the move.

Listings for Conditional Register-to-Register Moves

In this section the various register files move types are listed and illustrated with examples.

Listing 1 - DREG/C DREG to DREG/C DREG Register Moves/Swaps

When register-to-register swaps are unconditional, they operate the same in SISD mode and SIMD mode. If a condition is added to the instruction in SISD mode, the condition tests only in the PEx element and controls the entire operation. If a condition is added in SIMD mode, the condition tests in both the PEx and PEy elements separately and the halves of the operation are controlled as detailed in Table 4-40.
Conditional Instruction Execution

Table 4-40. DREG/CDREG Register Moves Summary (SISD Versus SIMD)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Instruction</th>
<th>Explicit Transfer Executed According to PEx</th>
<th>Implicit Transfer Executed According to PEy</th>
</tr>
</thead>
<tbody>
<tr>
<td>SISD1</td>
<td>IF condition Rx = Ry;</td>
<td>Rx loaded from Ry</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>IF condition Rx = Sy;</td>
<td>Rx loaded from Sy</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>IF condition Sx = Ry;</td>
<td>Sx loaded from Ry</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>IF condition Sx = Sy;</td>
<td>Sx loaded from Sy</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>IF condition Rx &lt;-&gt; Sy;</td>
<td>Rx loaded from Sy</td>
<td>Sy loaded from Rx</td>
</tr>
<tr>
<td>SIMD2</td>
<td>IF condition Rx = Ry;</td>
<td>Rx loaded from Ry</td>
<td>Sx loaded from Sy</td>
</tr>
<tr>
<td></td>
<td>IF condition Rx = Sy;</td>
<td>Rx loaded from Sy</td>
<td>Sx loaded from Ry</td>
</tr>
<tr>
<td></td>
<td>IF condition Sx = Ry;</td>
<td>Sx loaded from Ry</td>
<td>Rx loaded from Sy</td>
</tr>
<tr>
<td></td>
<td>IF condition Sx = Sy;</td>
<td>Sx loaded from Sy</td>
<td>Rx loaded from Ry</td>
</tr>
<tr>
<td></td>
<td>IF condition Rx &lt;-&gt; Sy;</td>
<td>Rx loaded from Sy</td>
<td>Sy loaded from Rx</td>
</tr>
</tbody>
</table>

1 In SISD mode, the conditional applies only to the entire operation and is only tested against PEx’s flags. When the condition tests true, the entire operation occurs.

2 In SIMD mode, the conditional applies separately to the explicit and implicit transfers. Where the condition tests true (PEx for the explicit and PEy for the implicit), the operation occurs in that processing element.

Listing 2 - UREG/CUREG to UREG/CUREG Register Moves

For the following instructions, the processors are operating in SIMD mode and registers in the PEx data register file are used as the explicit registers. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements is shown in Table 4-41.

IF EQ R9 = R2;
IF EQ PX1 = R2;
IF EQ USTAT1 = R2;
For the following instructions, the processors are operating in SIMD mode and registers in the PEy data register file are used as explicit registers. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements is shown in Table 4-42.

```
IF EQ R9 = $2;
IF EQ PX1 = $2;
IF EQ USTAT1 = $2;
```

Table 4-41. Register-to-Register Moves – Complementary Pairs

<table>
<thead>
<tr>
<th>Condition in PEx</th>
<th>Condition in PEy</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZx</td>
<td>AZy</td>
<td>Explicit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>No data move occurs</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>No data move to registers r9, px1, and ustat1 occurs</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>r2 transfers to registers r9, px1, and ustat1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>r2 transfers to registers r9, px1, and ustat1</td>
</tr>
</tbody>
</table>

Table 4-42. Register-to-Register Moves – Complementary Pairs

<table>
<thead>
<tr>
<th>Condition in PEx</th>
<th>Condition in PEy</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZx</td>
<td>AZy</td>
<td>Explicit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>No data move occurs</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>No data move to registers r9, px1, and ustat1 occurs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r2 transfers to registers s9, px2 and ustat2</td>
</tr>
</tbody>
</table>

Table 4-42. Register-to-Register Moves – Complementary Pairs

<table>
<thead>
<tr>
<th>Condition in PEx</th>
<th>Condition in PEy</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZx</td>
<td>AZy</td>
<td>Explicit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>No data move occurs</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>No data move to registers r9, px1, and ustat1 occurs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r2 transfers to registers s9, px2 and ustat2</td>
</tr>
</tbody>
</table>
Conditional Instruction Execution

Table 4-42. Register-to-Register Moves – Complementary Pairs (Cont’d)

<table>
<thead>
<tr>
<th>Condition in PEx</th>
<th>Condition in PEy</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZx</td>
<td>AZy</td>
<td>Explicit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>s2 transfers to registers r9, px1, and ustat1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>s2 transfers to registers r9, px1, and ustat1</td>
</tr>
</tbody>
</table>

Listing 4 - UREG to UREG/CUREG Register Moves

In this case, data moves from an uncomplementary register (ureg without a SIMD complement) to a complementary register pair. The processor executes the explicit move depending on the evaluation of the conditional test in the PEx processing element. The processor executes the implicit move depending on the evaluation of the conditional test in the PEy processing element. In each processing element where the move occurs, the content of the source register is duplicated in the destination register.

Note that while PX1 and PX2 are complementary registers, the combined PX register has no complementary register. For more information, see Chapter 2, Register Files.

For the following instruction the processors are operating in SIMD mode. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements is shown in Table 4-43.

IF EQ R1 = PX:
Program Sequencer

Listing 5 - UREG/CUREG to UREG Register Moves

In this case data moves from a complementary register pair to an uncomplementary register. The processor executes the explicit move to the uncomplemented universal register, depending on the condition test in the PEx processing element only. The processor does not perform an implicit move.

For all of the following instructions, the processors are operating in SIMD mode. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements for all of the example code samples are shown in Table 4-44.

IF EQ R1 = PX;

Uncomplementary register to DAG move:
if EQ m1 = PX;

DAG to uncomplementary register move:
if EQ PX = m1;

For more information, see Chapter 2, Register Files.

Note that the PX1 and PX2 registers have compliments, but PX as a register is uncomplementary.

Table 4-43. Uncomplimentary-to-Complementary Register Move

<table>
<thead>
<tr>
<th>Condition in PEx</th>
<th>Condition in PEy</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZx</td>
<td>AZy</td>
<td>Explicit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>r1 remains unchanged</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>r1 remains unchanged</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>r1 gets px value</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>r1 gets px value</td>
</tr>
</tbody>
</table>

SHARC Processor Programming Reference 4-101
Conditional Instruction Execution

DAG to DAG move:
if EQ m1 = i15;

Complimentary register to DAG move:
if EQ i6 = r9;

In all the cases described above, the behavior is the same. If the condition in PEx is true, then only the transfer occurs.

Table 4-44. Complementary-to-Uncomplimentary Register Move

<table>
<thead>
<tr>
<th>Condition in PEx</th>
<th>Condition in PEy</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZx</td>
<td>AZy</td>
<td>Explicit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>px remains unchanged</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>px remains unchanged</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>r1 40-bit explicit move to px</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>r1 40-bit explicit move to px</td>
</tr>
</tbody>
</table>

Listings for Conditional Register-to-Memory Moves

Conditional post-modify DAG operations update the DAG register based on ORing of the condition tests on both processing elements. Actual data movement involved in a conditional DAG operation is based on independent evaluation of condition tests in PEx and PEy. Only the post-modify update is based on the ORing of these conditional tests.

Conditional pre-modify DAG operations behave differently. The DAGs always pre-modify an index, independent of the outcome of the condition tests on each processing element.
### Program Sequencer

#### Listing 1 - DREG to Memory

For this instruction, the processors are operating in SIMD mode, a register in the PEx data register file is the explicit register, and \( I0 \) is pointing to an even address in internal memory (ADSP-214xx products external memory is also allowed). Indirect addressing is shown in the instructions in the example. However, the same results occur using direct addressing. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements is shown in Table 4-45.

\[
\text{IF EQ DM}(I0,M0) = R2;
\]

Table 4-45. Register-to-Memory Moves—Complementary Pairs (PEx Explicit Register)

<table>
<thead>
<tr>
<th>Condition in PEx</th>
<th>Condition in PEy</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZx</td>
<td>AZy</td>
<td>Explicit</td>
</tr>
<tr>
<td>0 0</td>
<td>No data move occurs</td>
<td>No data move occurs</td>
</tr>
<tr>
<td>0 1</td>
<td>No data move occurs from r2 to location I0</td>
<td>s2 transfers to location ((I0+n)^1)</td>
</tr>
<tr>
<td>1 0</td>
<td>r2 transfers to location I0</td>
<td>No data move occurs from s2 to location ((I0+n)^1)</td>
</tr>
<tr>
<td>1 1</td>
<td>r2 transfers to location I0</td>
<td>s2 transfers to location ((I0+n)^1)</td>
</tr>
</tbody>
</table>

1. In NW space \( n = 1 \), in SW space \( n = 2 \)

#### Listing 2 - CDREG to Memory

For the following instruction, the processors are operating in SIMD mode, a register in the PEy data register file is the explicit register and \( I0 \) is pointing to an even address in internal memory. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements is shown in Table 4-46.

\[
\text{IF EQ DM}(I0,M0) = S2;
\]
Conditional Instruction Execution

Table 4-46. Register-to-Memory Moves – Complementary Pairs (PEy Explicit Register)

<table>
<thead>
<tr>
<th>Condition in PEx</th>
<th>Condition in PEy</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZx</td>
<td>AZy</td>
<td>Implicit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>No data move occurs</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>No data move occurs from r2 to location 10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>s2 transfers to location 10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>s2 transfers to location 10 + n</td>
</tr>
</tbody>
</table>

1 In NW space n = 1, in SW space n = 2

Listing 3 - DREG/C DREG to IOP Memory Space

For the following instructions the processors are operating in SIMD mode and the explicit register is either a PEx register or PEy register. I0 points to IOP memory space. This example shows indirect addressing. However, the same results occur using direct addressing.

IF EQ DM(I0,M0) = R2;  
IF EQ DM(I0,M0) = S2;

Listing 4 - UREG to IOP Memory Space

In the case of memory-to-DAG register moves, the transfer does not occur when both PEx and PEy are false. Otherwise, if either PEx or PEy is true, transfers to the DAG register occur. For example:

if EQ m13 = dm(i0,m1);

Conditional data moves from a complementary register pair to an uncomplementary register with an access to IOP memory space results in unexpected behavior and should not be used.
Conditional Branches

The processor executes a conditional branch (\texttt{JUMP} or \texttt{CALL} with \texttt{RTI/RTS}) or loop (\texttt{DO/UNTIL}) based on the result of ANDing the condition tests on both PEx and PEy. A conditional branch or loop in SIMD mode occurs only when the condition is true in PEx and PEy.

Using complementary conditions (for example \texttt{EQ} and \texttt{NE}), programs can produce an ORing of the condition tests for branches and loops in SIMD mode. A conditional branch or loop that uses this technique must consist of a series of conditional compute operations. These conditional computes generate \texttt{NOP}s on the processing element where a branch or loop does not execute. For more information on programming in SIMD mode, see \textit{Chapter 9, Instruction Set Types}, and \textit{Chapter 10, Computation Instructions}.

\textbf{IF Conditional Branch Instructions}

The \texttt{IF} conditional direct branch instruction is available in Type 8 instruction. The \texttt{IF} conditional indirect branch instruction is available in the Type 9, 10, and 11 instructions. The instructions are shown in \textit{Table 4-47} and \textit{Table 4-48}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Conditional Test} & \textbf{Execution for Instruction Types 8–11} \\
\hline
0 (false) & IF not exe \\
\hline
1 (true) & IF exe \\
\hline
\end{tabular}
\caption{IF Conditional Branch Execution (SISD mode)}
\end{table}
Conditional Instruction Execution

Table 4-48. If Conditional Branch Instruction (SIMD Mode)

<table>
<thead>
<tr>
<th>Conditional Test</th>
<th>Execution for Instruction Types 8–11</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEy</td>
<td></td>
</tr>
<tr>
<td>0 (false)</td>
<td>IF not exe</td>
</tr>
<tr>
<td>1 (true)</td>
<td>IF exe</td>
</tr>
<tr>
<td>0 (false)</td>
<td>IF not exe</td>
</tr>
<tr>
<td>1 (true)</td>
<td>IF exe</td>
</tr>
</tbody>
</table>

IF Then ELSE Conditional Indirect Branch Instructions

The conditional IF then ELSE construct for indirect branch instructions is available in the Type 9, 10, and 11 instructions. The instructions are shown in Table 4-49 and Table 4-50.

Table 4-49. IF then ELSE Conditional Branch Execution (SISD mode)

<table>
<thead>
<tr>
<th>Conditional Test</th>
<th>Execution for Instruction Types 9–11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (false)</td>
<td>IF not exe</td>
</tr>
<tr>
<td>1 (true)</td>
<td>ELSE exe</td>
</tr>
<tr>
<td>1 (true)</td>
<td>ELSE not exe</td>
</tr>
</tbody>
</table>

Table 4-50. IF then ELSE Conditional Branch Instruction (SIMD Mode)

<table>
<thead>
<tr>
<th>Conditional Test</th>
<th>Execution for Instruction Types 9–11</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEy</td>
<td></td>
</tr>
<tr>
<td>0 (false)</td>
<td>IF not exe</td>
</tr>
<tr>
<td>1 (true)</td>
<td>ELSE PEx exe – PEY exe</td>
</tr>
<tr>
<td>0 (false)</td>
<td>IF not exe</td>
</tr>
<tr>
<td>1 (true)</td>
<td>ELSE PEx exe – PEY not exe</td>
</tr>
<tr>
<td>1 (true)</td>
<td>ELSE PEx not exe – PEY exe</td>
</tr>
<tr>
<td>1 (true)</td>
<td>ELSE PEx not exe – PEY not exe</td>
</tr>
</tbody>
</table>
For more information and examples, see the following instruction reference pages.

- “Type 8a ISA/VISA (cond + branch)” on page 9-32
- “Type 9a ISA/VISA (cond + Branch + comp/else comp)” on page 9-35
- “Type 10 ISA (cond + branch + else comp + mem data move)” on page 9-40
- “Type 11a ISA/VISA (cond + branch return + comp/else comp)
  Type 11c VISA (cond + branch return)” on page 9-44

**IF Conditional Branch Limitations in VISA**

Type 10 instructions are the most infrequently used instructions in the Instruction Set Architecture:

/* Template: */
IF COND JUMP (Md, Ic), ELSE compute, DM(Ia, Mb) = dreg ;

To make maximum use of available opcode combinations, the ADSP-214xx processor’s use the Type 10 instruction opcode to encode a simpler and more commonly used compute instructions such as:

Rm = Rn + Rm ;

Code generated by the VisualDSP++ C compiler does not use the Type 10 instruction.

If assembly code containing Type 10 instructions are run through the VisualDSP++ code generation tools, the assembler issues an error message stating that a Type 10 instruction is not supported while in VISA short word space.
Instruction Pipeline Hazards

Instruction Pipeline Hazards

The processors use instruction pipeline stalls to ensure correct and efficient program execution. Since the instruction pipeline is fully interlocked, programmers need to be aware of the different control and data hazards. Stalls are used in the following situations.

- "Structural Hazard Stalls" on page 4-108 are incurred when different instructions at various stages of the instruction pipeline attempt to use the same processor resources simultaneously.

- "Data Hazard Stalls" on page 4-109 are incurred when an instruction attempts to read a value from a register or from a condition flag, that has been updated by an earlier instruction, before the value becomes available.

- Stalls are incurred to achieve high performance, when the processor executes a certain sequence of instructions.

- Stalls are incurred to retain effect latency compatible with earlier SHARC processors when the processor executes a certain sequence of instructions.

The following sections describe the various kinds of stalls in detail.

Structural Hazard Stalls

In general, structural stalls occur when different instructions at various stages of the instruction pipeline attempt to use the same resource at the same time during the same cycle. The following sections describe variations of structural stalls and provide examples.
Simultaneous Access Over the DMD and PMD Buses

Data access over the DM bus to a particular block of memory and a data access over the PM bus to the same block. These two operations conflict over the single read or write port of the given block. In this example, the data access instruction over the DM bus completes first.

DMA Block Conflict with PM or DM Access

A direct memory access (DMA) by a peripheral such as the external port to a particular block of memory and a data/instruction access by the sequencer over the DM or PM bus to the same block of memory. The DMA transfer completes first to ensure that no data overflow or underflow takes place in the processor’s peripherals.

Core Memory-Mapped Registers

The SYSCTL and BRKCTL are two memory-mapped registers, which, unlike many other memory-mapped registers in the processor core, serve as control registers. The effect latency for these registers is one cycle following a write to these registers.

Data Hazard Stalls

In general, data and control hazard stalls occur when a register or a condition flag is being updated by an instruction and a subsequent instruction attempts to read the value before the update has actually taken place.

When this occurs, the instruction that is to update the value and the following instruction, (if not dependent on the new value), are allowed to execute. If the following instruction needs the updated value, then that instruction and the instructions that follow it in the earlier stages of the instruction pipeline are stalled.

The conditions under which data/control hazard stalls occur are described in the following sections.
Instruction Pipeline Hazards

Multiplier Operand Load Stalls

When both of the operands of the multiplier (fixed or floating point) are produced as a result of either a multiplier or an ALU operation in the immediate preceding instruction, the pipeline is stalled for one cycle as shown in the following example.

\[
F_0 = F_0 + F_4, F_1 = F_0 - F_4; \\
F_0 = F_0 * F_1; \\
/* stalls a cycle since both the operands are produced by ALU in the immediately preceding instruction */
\]

DAG Register Load Stalls

Stalls occur when a register in a DAG is loaded and either of the two following instructions (shown in the code examples below) attempts to generate an address based on that register. This is because address generation requires that the value of the related DAG register is read in the Decode stage, while any other register load completes in the Execution stage of the pipeline. Note that registers can be loaded either by explicit or implicit references (such as in a long word load).

\[
M_0 = 1; \\
DM(I2, M_0) = R1; /* stalls for 2 cycles */
\]

In the example shown in Table 4-51, \(M_0\) is written back at the end of the execution stage, while the DM access instruction reads \(M_0\) in the Decode stage to generate the address. The first instruction is allowed to execute normally, while the remaining instructions are delayed by two cycles.

Also in Table 4-51, the data memory instruction is stalled if the preceding instruction is a load of the \(I2, B2,\) or \(L2\) registers, regardless of whether circular buffering is enabled or not.
In the code example below and Table 4-52, an unrelated instruction is introduced after a write instruction to the DAG. In this case the processor stalls for one cycle only.

\[
\begin{align*}
M0 &= 1; \\
R0 &= 0x8 \quad /* \text{any unrelated instruction} */ \\
Dm(I2, M0) &= R1 \quad /* \text{Stalls for one cycle} */
\end{align*}
\]

**Table 4-51. Indirect Access One Cycle After DAG Register Load**

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td></td>
<td></td>
<td>M0 = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>M0 = 1</td>
<td></td>
<td>DM (I2, M0) = R1;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decode</td>
<td>M0 = 1</td>
<td></td>
<td>DM (I2, M0) = R1;</td>
<td></td>
<td>n</td>
</tr>
<tr>
<td>Fetch2</td>
<td>DM (I2, M0) = R1;</td>
<td>n</td>
<td></td>
<td>n+1</td>
<td>n+2</td>
</tr>
<tr>
<td>Fetch1</td>
<td>n</td>
<td></td>
<td></td>
<td>n+1</td>
<td>n+2</td>
</tr>
</tbody>
</table>

1. Cycle2: Stall cycle
2. Cycle3: Stall cycle

Table 4-52. Indirect Access Two Cycles After DAG Register Load

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td></td>
<td>M0 = 1</td>
<td>R0 = 0x8;</td>
<td></td>
<td>DM (I2, M0) = R1;</td>
</tr>
<tr>
<td>Address</td>
<td>M0 = 1</td>
<td>R0 = 0x8;</td>
<td></td>
<td>DM (I2, M0) = R1;</td>
<td>n</td>
</tr>
<tr>
<td>Decode</td>
<td>R0 = 0x8;</td>
<td></td>
<td>DM (I2, M0) = R1;</td>
<td>n</td>
<td>n+1</td>
</tr>
<tr>
<td>Fetch2</td>
<td>DM (I2, M0) = R1;</td>
<td>n</td>
<td></td>
<td>n+1</td>
<td>n+2</td>
</tr>
<tr>
<td>Fetch1</td>
<td>n</td>
<td></td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
</tr>
</tbody>
</table>

1. Cycle2: Stall cycle
Instruction Pipeline Hazards

Branch Stalls

A data stall can also occur when a register in a DAG is loaded and either of the following two instructions shown in the code examples below attempts to generate an indirect target address based on that DAG register for a branch such as a JUMP or CALL. This happens because the address generation requires the values of the related DAG register to be read in the Decode stage, while the load of any register completes in the Execute stage of the pipeline. The JUMP or CALL itself has three cycles of overhead as described in “Instruction Driven Branches” on page 4-15.

M8 = 1;
JUMP (M8,19);   /* stalls for two cycles */

In the example shown in Table 4-53, M8 is written back at the end of the Execute stage of the pipeline, while the following JUMP (or CALL) instruction has to read M8 in the Decode stage to generate the target address. The first instruction is allowed to complete normally, while all following instructions are stalled for two cycles.

In the following code example, an unrelated instruction is inserted between the write instruction to the DAG register and the jump instruction requiring address generation. In this instance, the pipeline stalls for only one cycle.

M8 = 1;
R0 = 0x8;        /* any unrelated instruction */
JUMP (M0,19);   /* stalls for one cycle */
Conditional Branch Stalls

There are three cases related to conditional branches, where the pipeline is stalled for one or more cycles.

1. A control hazard stall occurs when a conditional branch follows a compute or a bit manipulation instruction as shown in the code example and Table 4-54. This occurs because the branch instruction needs the condition flags information in the Address stage of the pipeline, while the compute and bit manipulation instructions update condition flags at the end of Execute phase. (An RTS has three additional overhead cycles. See “Instruction Driven Branches” on page 4-15.)

   \[\text{R0} = \text{R0-1;}\]
   \[\text{If ne RTS:} \quad /* \text{stalls pipe for a cycle} */\]
2. If the compute involves the multiplier unit and the condition is based on a multiplier flag (as shown in the code sample below), and the conditional branch is in Decode stage of the pipeline, the pipeline is stalled for an additional cycle.

```c
R0 = R0*R1(ssi);
IF MV CALL (_MultOverFlow); /* stalls for two cycles in decode */
```

3. The pipeline stalls for two cycles when a branch instruction, conditional on `NOT LCE` (loop counter not expired), is in the Decode stage and is immediately followed by any instruction involving a change in an `LCE` (loop counter expired) condition, due to the execution of a `DO/UNTIL`, `POP/PUSH`, `JUMP(LA)` or load of the `CURLCNTR` register. A one cycle stall occurs when the instruction is an operation other than a branch.

### Table 4-54. Conditional Branch Stall

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td></td>
<td>R0 = R0 – 1</td>
<td>if ne RTS</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td></td>
<td>R0 = R0 – 1</td>
<td>if ne RTS</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>r</td>
<td>r+1</td>
</tr>
<tr>
<td>Decode</td>
<td>if ne RTS</td>
<td>n \rightarrow nop</td>
<td>n+1 \rightarrow nop</td>
<td>n+2 \rightarrow nop</td>
<td>r</td>
<td>r+1</td>
<td>r+2</td>
<td></td>
</tr>
<tr>
<td>Fetch2</td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>r</td>
<td>r+1</td>
<td>r+2</td>
<td>r+3</td>
<td></td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+1</td>
<td>n+2</td>
<td>r</td>
<td>r+1</td>
<td>r+2</td>
<td>r+3</td>
<td>r+4</td>
<td></td>
</tr>
</tbody>
</table>

r is the instruction branch address
1. Cycle2: Stall cycle
2. Cycle4: r popped from PC stack
Note that if the CURLCNTR register changes due to the normal loop-back operation within a counter based loop, the pipeline is not stalled for any branch instruction conditional on the NOT LCE condition.

Control Hazard Stalls

A control hazard stall occurs when the sequence of three instructions shown below is executed. The first may be a compute instruction, which directly modifies the ASTATx, ASTATy or FLAGS registers, either through an explicit write to the register or through bit manipulation instruction. The second instruction contains a conditional post-modify address generation. The third instruction is either an address generation operation using the same index register or a read of that index register.

The example code and Table 4-55 below shows that when this sequence of instructions is executed, and the third instruction is in the Decode stage of the pipeline, the pipeline is stalled for two cycles.

R2 = R3 - R4;            /* ALU instruction, setting a condition flag */
IF EQ DM(I1,M0) = R15    /* conditional post-modify addressing */
DM(I1,M2) = R14;         /* address generation using the same I register stalls for two cycles */

When the conditional post-modify instruction is either preceded or followed by instructions other than those involving address generation using the same I register, the last instruction stalls the pipeline for one cycle. When the conditional post-modify instruction is either preceded or followed by two or more such unrelated instructions, the pipeline is not stalled.

Note that a conditional instruction based on an ALU generated flag has a dependency on an ALU operation only. This also holds true in the case of multiplier flags and multiplier operations or a BTF flag and a BIT TST instruction. This is valid for any such kind of dependency.
Instruction Pipeline Hazards

Table 4-55. Indirect Branch Two Cycles After DAG Register Load

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>n</td>
<td>n+1</td>
<td></td>
<td></td>
<td></td>
<td>n+2</td>
</tr>
<tr>
<td>Address</td>
<td>n</td>
<td>n+1</td>
<td></td>
<td></td>
<td>n+2</td>
<td>n+3</td>
</tr>
<tr>
<td>Decode</td>
<td>n+1</td>
<td></td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td></td>
</tr>
<tr>
<td>Fetch2</td>
<td>n+2</td>
<td></td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td></td>
</tr>
<tr>
<td>Fetch1</td>
<td>n+3</td>
<td></td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
<td></td>
</tr>
</tbody>
</table>

1. Cycle 2: Stall cycle
2. Cycle 3: Stall cycle

Also note that when this kind of instruction sequence has other reasons to stall the pipeline, all the stalls arising out of different kinds of dependencies may not merge and some stalls appear as redundant stall cycles.

The pipeline is stalled when the processor executes certain sequence of instructions to maximize the frequency of operation. The case arises when a compute operation involving any fixed-point operand register follows a floating-point multiply operation, and the instruction involving the fixed-point register is in the Decode stage of the pipeline, the pipeline stalls for one cycle as shown in the following example. Note that the actual register used for the operation is not relevant.

\[
\begin{align*}
F_0 &= F_0 \cdot F_4; \\
F_5 &= \text{FLOAT } R_1; & \quad \text{/* stalls the pipe when in decode */} \\
F_0 &= F_0 \cdot F_4; \\
R_5 &= \text{LSHIFT } R_{10} \text{ by 2}; & \quad \text{/* stalls the pipe when in decode */} \\
F_0 &= F_0 \cdot F_4; \\
R_5 &= R_5 - 1; & \quad \text{/* stalls the pipe when in decode */}
\end{align*}
\]
Loop Stalls

1. A \texttt{JUMP(LA)} stalls the instruction pipeline for one cycle when it is in the Address stage of the instruction pipeline.

2. When the length of the counter based loop is one, two or four instructions, the pipeline is stalled by one cycle after the \texttt{DO/UNTIL} instruction.

3. A one cycle stall is incurred when a \texttt{RTS} (return from subroutine) or \texttt{RTI} (return from interrupt) instruction causes the sequencer to return to the last instruction of a loop instruction, and the \texttt{RTI/RTS} is in the Address stage of the instruction pipeline. This is to avoid the coincidence of two implicit operations of the \texttt{PCSTK}—one due to the \texttt{RTI/RTS} instruction and the other due to the possible termination of the loop. The pipeline stalls so that the pop operation from the \texttt{RTI/RTS} is executed first.

Compiler Related Stalls

The following sections discuss stalls introduced by the compiler.

\textbf{CJUMP Instruction}

The following code examples show a two cycle data hazard stall that occurs when DAG1 attempts to generate addresses based on the I6 register or when either or both of the I6 or I7 registers are used as a source of some data transfer operation immediately after a \texttt{CJUMP} instruction. This occurs because the \texttt{CJUMP} instruction modifies the I6 register.

Example 1

\begin{verbatim}
CJUMP(_SUBI)(DB);       /* executes R2 = I6, I6 = I7, 
                        jump(_sub1) (db) */
DM(I6,M0) = R2;         /*stalls for two cycles */
\end{verbatim}
Instruction Pipeline Hazards

Example 2

CJUMP(_SUB1)(DB); /* executes R2 = I6, I6 = I6, 
jump(_sub1) (db) */
R2 = I7;        /* stalls for two cycles */

If there is an unrelated instruction before the second instruction, the pipeline stalls for one cycle only. Note that an address generation operation using register I7 immediately after a CJUMP instruction does not stall the pipeline.

The CJUMP instruction is intended to be used by the compiler only. Normally the compiler uses the following sequence of instructions when calling a subroutine, which does not stall the pipeline.

CJUMP (_SUB1) (DB); /* executes R2 = I6, I6 = I7 */
jump(_sub1)(db)
DM(I7,M0) = R2; /* stores previous I6 */
DM(I7,M0) = PC; /* stores return_address-1 */

RFRAME Instruction

A data hazard stall occurs when DAG1 attempts to generate addresses based on the I6 or I7 registers or when any or both of the I6 or I7 registers are used as a source of some data transfer operation immediately after a RFRAME instruction. This occurs because RFRAME modifies the I6 and I7 registers. In this situation, the pipeline is stalled for two cycles.

RFRAME: /* executes I7 = I6, I6 = dm(0,I6); */
DM(I6,M0) = R2 /* stalls for two cycles */

In a program where there is an unrelated instruction before the DM instruction, then the pipeline stalls for one cycle only.

The RFRAME instruction is only used by the compiler.
Sequencer Interrupts

This section describes the interrupts that are triggered by the sequencer itself.

External Interrupts

For external interrupts (IRQ2–0, DAI, DPI) the processor supports two types of interrupt sensitivity—edge-sensitive and level-sensitive. The interrupt overview is shown in Table 4-56.

The DAI/DPI modules also incorporate interrupt controllers for external events. For more information refer to the processor specific hardware reference manual “Masking Interrupts”.

Table 4-56. External Interrupt Overview

<table>
<thead>
<tr>
<th>Interrupt Source</th>
<th>Interrupt Condition</th>
<th>Interrupt Priorities</th>
<th>Interrupt Acknowledge</th>
<th>IVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRQ2–0</td>
<td>–level triggered</td>
<td>8–10</td>
<td>RTI instruction</td>
<td>IRQ2–0I</td>
</tr>
<tr>
<td></td>
<td>–falling edge triggered</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The processor detects a level-sensitive interrupt if the signal input is low (active) when sampled on the rising edge of PCLK/2. A level-sensitive interrupt must go high (inactive) before the processor returns from the interrupt service routine. If a level-sensitive interrupt is still active when the processor samples it after returning from its service routine, the processor treats the signal as a new request. The processor repeats the same interrupt routine without returning to the main program, assuming no higher priority interrupts are active.

The processor detects an edge-sensitive interrupt if the input signal is high (inactive) on one cycle and low (active) on the next cycle when sampled on the rising edge of PCLK/2. An edge-sensitive interrupt signal can stay
Sequencer Interrupts

active indefinitely without triggering additional interrupts. To request another interrupt, the signal must go high, then low again.

Edge-sensitive interrupts require less external hardware compared to level-sensitive requests, because negating the request is unnecessary. An advantage of level-sensitive interrupts is that multiple interrupting devices may share a single level-sensitive request line on a wired OR basis, allowing easy system expansion.

The MODE2 register controls external interrupt sensitivity as described below.

- **Interrupt 0 Sensitivity.** Bit 0 (IRQ0E) directs the processor to detect IRQ0 as edge-sensitive (if 1) or level-sensitive (if 0).

- **Interrupt 1 Sensitivity.** Bit 1 (IRQ1E) directs the processor to detect IRQ1 as edge-sensitive (if 1) or level-sensitive (if 0).

- **Interrupt 2 Sensitivity.** Bit 2 (IRQ2E) directs the processor to detect IRQ2 as edge-sensitive (if 1) or level-sensitive (if 0).

The processor accepts external interrupts that are asynchronous to the processor’s clocks, allowing external interrupt signals to change at any time.

External interrupts must meet the minimum pulse width requirement. For information on interrupt signal timing requirements, see the appropriate SHARC processor data sheet.

Software Interrupts

Software interrupts (or programmed exceptions) are instructions which explicitly generate an exception. The interrupt overview is shown in Table 4-57.
Program Sequencer

The `IRPTL` register provides four software interrupts. When a program sets the latch bit for one of these interrupts (`SFT0I`, `SFT1I`, `SFT2I`, or `SFT3I`), the sequencer services the interrupt, and the processor branches to the corresponding interrupt routine. Software interrupts have the same behavior as all other maskable interrupts. For more information, see Appendix B, Core Interrupt Control.

If programs force an interrupt by writing to a bit in the `IRPTL` register, the processor recognizes the interrupt in the following cycle, and four cycles of branching to the interrupt vector follow the recognition cycle.

**Hardware Stack Interrupts**

The hardware stack (status stack, loop stack and PC stack) conditions trigger a maskable interrupt shown in Table 4-58. The overflow and full flags provide diagnostic aid only. Programs should not use these flags for runtime recovery from overflow. The empty flags can ease stack saves to memory. Programs can monitor the empty flag when saving a stack to memory to determine when the processor has transferred all the values.

Table 4-57. Software Interrupt Overview

<table>
<thead>
<tr>
<th>Interrupt Source</th>
<th>Interrupt Condition</th>
<th>Interrupt Priorities</th>
<th>Interrupt Acknowledge</th>
<th>IVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>Bit set IRPTL instruction</td>
<td>38–41</td>
<td>RTI instruction</td>
<td>SFT0–3I</td>
</tr>
</tbody>
</table>

The `IRPTL` register provides four software interrupts. When a program sets the latch bit for one of these interrupts (`SFT0I`, `SFT1I`, `SFT2I`, or `SFT3I`), the sequencer services the interrupt, and the processor branches to the corresponding interrupt routine. Software interrupts have the same behavior as all other maskable interrupts. For more information, see Appendix B, Core Interrupt Control.

If programs force an interrupt by writing to a bit in the `IRPTL` register, the processor recognizes the interrupt in the following cycle, and four cycles of branching to the interrupt vector follow the recognition cycle.

**Hardware Stack Interrupts**

The hardware stack (status stack, loop stack and PC stack) conditions trigger a maskable interrupt shown in Table 4-58. The overflow and full flags provide diagnostic aid only. Programs should not use these flags for runtime recovery from overflow. The empty flags can ease stack saves to memory. Programs can monitor the empty flag when saving a stack to memory to determine when the processor has transferred all the values.

Table 4-58. Hardware Stack Interrupt Overview

<table>
<thead>
<tr>
<th>Interrupt Source</th>
<th>Interrupt Condition</th>
<th>Interrupt Priorities</th>
<th>Interrupt Acknowledge</th>
<th>IVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>HW Stack</td>
<td>–PC stack overflow</td>
<td>3</td>
<td>RTI instruction</td>
<td>SOVF1</td>
</tr>
</tbody>
</table>
Summary

To manage events, the sequencer’s interrupt controller handles interrupt processing, determines whether an interrupt is masked, and generates the appropriate interrupt vector address. With selective caching, the instruction cache lets the processor access data in program memory and fetch an instruction (from the cache) in the same cycle. The DAG2 data address generator outputs program memory data addresses.

Figure 4-2 on page 4-4 identifies all the functional blocks and their relationship to one another in detail.

The sequencer evaluates conditional instructions and loop termination conditions by using information from the status registers. The loop address stack and loop counter stack support nested loops. The status stack stores status registers for implementing nested interrupt routines.

“Program Sequencer Registers” on page A-8 lists the registers within and related to the program sequencer. All registers in the program sequencer are universal registers (Uregs), so they are accessible to other universal registers and to data memory. All of the sequencer’s registers and the top of stacks are readable and writable, except for the Fetch1, decode, and PC registers. Pushing or popping the PC stack is done with a write to the PC stack pointer, which is readable and writable. Pushing or popping the loop address stack requires explicit instructions.

A set of system control registers configures or provides input to the sequencer. A bit manipulation instruction permits setting, clearing, toggling, or testing specific bits in the system registers. For information on this instruction (bit) and the instruction set, see Chapter 9, Instruction Set Types, and Chapter 10, Computation Instructions. Writes to some of these registers do not take effect on the next cycle. For example, after a write to the MODE1 register enables ALU saturation mode, the change takes effect two cycles after the write. Also, some of these registers do not update on the cycle immediately following a write. An extra cycle is required before a register read returns the new value.
5  TIMER

The core includes a programmable interval timer, which appears in Figure 5-1. Bits in the MODE2, TCOUNT, and TPERIOD registers control timer operations. Table A-2 on page A-7 lists the bits in the MODE2 register.

Features

• Simple programming model of 3 registers for interval timer
• Provides high or low priority interrupt
• If counter expired timer expired pin is asserted
• If core is in emulation space timer halts

Functional Description

The bits that control the timer are given as follows:

• **Timer enable.** MODE2 Bit 5 (TIMEN). This bit directs the processor to enable (if 1) or disable (if 0) the timer.

• **Timer count.** (TCOUNT) This register contains the decrementing timer count value, counting down the cycles between timer interrupts.

• **Timer period.** (TPERIOD) This register contains the timer period, indicating the number of cycles between timer interrupts. The TCOUNT register contains the timer counter.
To start and stop the timer, programs use the \texttt{MODE2} register’s \texttt{TIMEN} bit. With the timer disabled (\texttt{TIMEN} = 0), the program loads \texttt{TCOUNT} with an initial count value and loads \texttt{TPERIOD} with the number of cycles for the desired interval. Then, the program enables the timer (\texttt{TIMEN}=1) to begin the count.

On the core clock cycle after \texttt{TCOUNT} reaches zero, the timer automatically reloads \texttt{TCOUNT} from the \texttt{TPERIOD} register. The \texttt{TPERIOD} value specifies the frequency of timer interrupts. The number of cycles between interrupts is \texttt{TPERIOD} + 1. The maximum value of \texttt{TPERIOD} is $2^{32} - 1$.

The timer decrements the \texttt{TCOUNT} register during each clock cycle. When the \texttt{TCOUNT} value reaches zero, the timer generates an interrupt and asserts the \texttt{TMREXP} output pin high for several cycles (when the timer is enabled), as shown in Figure 5-1. For more information about \texttt{TMREXP} pin muxing, refer to system design chapter in the processor specific hardware reference.

Programs can read and write the \texttt{TPERIOD} and \texttt{TCOUNT} registers by using universal register transfers. Reading the registers does not effect the timer. Note that an explicit write to \texttt{TCOUNT} takes priority over the sequencer’s loading \texttt{TCOUNT} from \texttt{TPERIOD} and the timer’s decrementing of \texttt{TCOUNT}. Also note that \texttt{TCOUNT} and \texttt{TPERIOD} are not initialized at reset. Programs should initialize these registers before enabling the timer.
To start and stop the timer, the TIMEN bit in MODE2 register has to be set or cleared respectively. The latency of this bit is two core clock cycles at the start of the counter and one core clock cycle at the stop of the counter shown in Figure 5-2.
Timer Interrupts

The timer expired event (TCOUNT decrements to zero) generates two interrupts, TMZHI and TMZLI. For information on latching and masking these interrupts to select timer expired priority, see “Latching Interrupts” on page 4-34.

The Timer interrupt overview is shown in Table 5-1.

Table 5-1. DAG Interrupt Overview

<table>
<thead>
<tr>
<th>Interrupt Source</th>
<th>Interrupt Condition</th>
<th>Interrupt Priorities</th>
<th>Interrupt Acknowledge</th>
<th>IVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Timer</td>
<td>–Timer high expired</td>
<td>4, 32</td>
<td>RTI instruction</td>
<td>TMZHI</td>
</tr>
<tr>
<td></td>
<td>–Timer low expired</td>
<td></td>
<td></td>
<td>TMZLI</td>
</tr>
</tbody>
</table>

Figure 5-2. Timer Enable and Disable
One event can cause multiple interrupts. The timer decrementing to zero causes two timer expired interrupts to be latched, TMZHI (high priority) and TMZLI (low priority). This feature allows selection of the priority for the timer interrupt. Programs should unmask the timer interrupt with the desired priority and leave the other one masked. If both interrupts are unmasked, the processor services the higher priority interrupt first and then services the lower priority interrupt.
Timer Interrupts
6 DATA ADDRESS GENERATORS

The processor’s data address generators (DAGs) generate addresses for data moves to and from data memory (DM) and program memory (PM). By generating addresses, the DAGs let programs refer to addresses indirectly, using a DAG register instead of an absolute address. The DAG’s architecture, which appears in Figure 6-1, supports several functions that minimize overhead in data access routines.

Features

- **Supply address and post-modify.** Provides an address during a data move and auto-increments the stored address for the next move.

- **Supply pre-modified address.** Provides a modified address during a data move without incrementing the stored address.

- **Modify address.** Increments the stored address without performing a data move.

- **Bit-reverse address.** Provides a bit-reversed address during a data move without reversing the stored address, as well as an instruction to explicitly bit-reverse the supplied address.

- **Broadcast data loads.** Performs dual data moves to complementary registers in each processing element to support single-instruction multiple-data (SIMD) mode.
Functional Description

- **Circular Buffering.** Supports addressing a data buffer at any address with predefined boundaries, wrapping around to cycle through this buffer repeatedly in a circular pattern.

Functional Description

As shown in Figure 6-1, each DAG has four types of registers. These registers hold the values that the DAG uses for generating addresses. The four types of registers are:

- **Index registers** (I0–I7 for DAG1 and I8–I15 for DAG2). An index register holds an address and acts as a pointer to memory. For example, the DAG interprets \texttt{DM(I0,0)} and \texttt{PM(I8,0)} syntax in an instruction as addresses.

- **Modify registers** (M0–M7 for DAG1 and M8–M15 for DAG2). A modify register provides the increment or step size by which an index register is pre- or post-modified during a register move. For example, the \texttt{DM(I0,M1)} instruction directs the DAG to output the address in register I0 then modify the contents of I0 using the M1 register.

- **Length and base registers** (L0–L7 and B0–B7 for DAG1 and L8–L15 and B8–B15 for DAG2). Length and base registers set the range of addresses and the starting address for a circular buffer. For more information on circular buffers, see “Circular Buffer Programming Model” on page 6-21.
Data Address Generators

Figure 6-1. Data Address Generator (DAG) Block Diagram
**Functional Description**

**DAG Address Output**

The following sections describe how the DAGs output addresses.

**Address Versus Word Size**

The processor’s internal memory accommodates the following word sizes:

- 64-bit long word data (LW)
- 40-bit extended-precision normal word data (NW, 48-bit)
- 32-bit normal word data (NW, 32-bit)
- 16-bit short word data (SW, 16-bit)

Only the address space determines which memory word size is accessed. An important item to note is that the DAG automatically adjusts the output address per the word size of the address location (short word, normal word, or long word). This address adjustment allows internal memory to use the address directly as shown in the following example.

```c
I15=LW_addr;
pm(i15,0)=r0; /* 64-bit transfer */

I7=NW_addr;
dm(i7,0)=r8; /* 32-bit transfer */

I7=SW_addr;
dm(i7,0)=r14; /* 16-bit transfer */
```
Data Address Generators

DAG Register-to-Bus Alignment

There are three word alignment types for DAG registers and PM or DM data buses:

- Normal word (32-bit)
- extended-precision normal word (40-bit)
- long word (64-bit)

32-Bit Alignment

The DAGs align normal word (32-bit) addressed transfers to the low order bits of the buses. These transfers between memory and 32-bit DAG1 or DAG2 registers use the 64-bit DM and PM data buses. Figure 6-2 illustrates these transfers.

40-Bit Alignment

The DAGs align register-to-register transfers to bits 39–8 of the buses. These transfers between a 40-bit data register and 32-bit DAG1 or DAG2 registers use the 64-bit DM and PM data buses. Figure 6-3 illustrates these transfers.
Functional Description

64-Bit Alignment

Long word (64-bit) addressed transfers between memory and 32-bit DAG1 or DAG2 registers target double DAG registers and use the 64-bit DM and PM data buses. Figure 6-4 illustrates how the bus works in these transfers.

Figure 6-4. Long Word DAG Register-to-Data Register Transfers

DAG1 Versus DAG2

DAG registers are part of the universal register ($\text{Ureg}$) set. Programs may load the DAG registers from memory, from another universal register, or with an immediate value. Programs may store the DAG registers' contents to memory or to another universal register.
Both DAGs are identical in their operation modes and can access the entire memory-mapped space. However, the following differences should be noted.

- Only DAG1 is capable of supporting compiler specific instructions like `RFRAME` and `CJUMP`.
- Only DAG2 is capable of supporting flow control instruction for indirect branches. Additionally DAG2 access can cause cache miss/hits for internal memory execution.

## DAG Instruction Types

The processor’s DAGs perform several types of operations to generate data addresses. As shown in Figure 6-1 on page 6-3, the DAG registers and the `MODE1` and `MODE2` registers contribute to DAG operations. The `STKYx` registers may be affected by the DAG operations and are used to check the status of a DAG operation.

An important item to note from Figure 6-1 is that the DAG automatically adjusts the output address per the word size of the address location (short word, normal word, or long word). This address adjustment lets internal memory use the address directly.

*SISD/SIMD mode, access word size, and data location (internal) all influence data access operations.*

## Long Word Memory Access Restrictions

If the long word transfer specifies an even numbered DAG register (10 or 12), then the even numbered register value transfers on the lower half of the 64-bit bus, and the even numbered register + 1 value transfers on the upper half (bits 63–32) of the bus as shown below.
**DAG Instruction Types**

\[ I_8 = DM(I_2,M_2); \quad \text{/* I2 loads to I8/9 pair */} \]
\[ PM(I_{14},M_{14}) = M_5; \quad \text{/* stores M5/4 pair to I14*/} \]

If the long word transfer specifies an odd numbered DAG register (I1 or B3), the odd numbered register value transfers on the lower half of the 64-bit bus, and the odd numbered register – 1 value (I0 or B2 in this example) transfers on the upper half (bits 63–32) of the bus.

In both the even and odd numbered cases, the explicitly specified DAG register sources or sinks bits 31–0 of the long word addressed memory.

**Table 6-1. Neighbor DAG Register for Long Word Accesses (x = B, I, L, M)**

<table>
<thead>
<tr>
<th>DAG Neighbor Registers</th>
<th>x0 and x1</th>
<th>x8 and x9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x2 and x3</td>
<td>x10 and x11</td>
</tr>
<tr>
<td></td>
<td>x4 and x5</td>
<td>x12 and x13</td>
</tr>
<tr>
<td></td>
<td>x6 and x7</td>
<td>x14 and x15</td>
</tr>
</tbody>
</table>

**Forced Long Word (LW) Memory Access Instructions**

When data is accessed using long word addressing, the data is always long word aligned on 64-bit boundaries in internal memory space. When data is accessed using normal word addressing and the LW mnemonic, the program should maintain this alignment by using an even normal word address (least significant bit of address = 0). This register selection aligns the normal word address with a 64-bit boundary (long word address). For more information, see “Unaligned Forced Long Word Access” on page 7-25.

The forced long word (LW) mnemonic only effects normal word address accesses and overrides all other factors (PEYEN, IMDWx).
Data Address Generators

All long word accesses load or store two consecutive 32-bit data values. The register file source or destination of a long word access is a set of two neighboring data registers (Table 6-1) in a processing element. In a forced long word access (using the LW mnemonic), the even (normal word address) location moves to or from the explicit register in the neighbor-pair, and the odd (normal word address) location moves to or from the implicit register in the neighbor-pair. In Listing 6-1 the following long word moves could occur.

Listing 6-1. Long Word Move Options

DM(0x98000) = R0 (LW);
/* The data in R0 moves to location DM(0x98000), and the data in R1 moves to location DM(0x98001) */
R15 = DM(0x98003)(LW);
/* The data at location DM(0x98003) moves to R15, and the data at location DM(0x98002) moves to R14 */

The forced long word (LW) mnemonic can be used for context switch between tasks in system applications. It only effects normal word address accesses and overrides all other factors (PEYEN, IMDWx bit settings) as shown in Listing 6-2.

Listing 6-2. Push the DAG Registers onto SW Stack

pm(i15,m15)=i0(lw);
/*until*/

pm(i15,m15)=i6(lw);
dm(i7,m7)=i8(lw);
/*until*/
dm(i7,m7)=i14(lw);
DAG Instruction Types

Listing 6-3. Pop the DAG Registers from SW Stack:

\[ i_0 = \text{pm}(i_{15}, m_{15})(lw); \]

/*until*/
\[ i_6 = \text{pm}(i_{15}, m_{15})(lw); \]
\[ i_8 = \text{dm}(i_{7}, m_{7})(lw); \]

/*until*/
\[ i_{14} = \text{dm}(i_{7}, m_{7})(lw); \]

Pre-Modify Instruction

As shown in Figure 6-5, the DAGs support two types of modified addressing, pre- and post-modify. Modified addressing is used to generate an address that is incremented by a value or a register.

![Pre-Modify and Post-Modify Operations](image)

Figure 6-5. Pre-Modify and Post-Modify Operations

In pre-modify addressing, the DAG adds an offset (modifier), which is either an M register or an immediate value, to an I register and outputs the resulting address. Pre-modify addressing does not change or update the I register.
Data Address Generators

The DAG pre-modify addressing type can be used to emulate the pop (restore of registers) from a SW stack.

Pre-modify addressing operations must not change the memory space of the address.

Post-Modify Instruction

The DAGs support post-modify addressing. Modified addressing is used to generate an address that is incremented by a value or a register. In post-modify addressing, the DAG outputs the I register value unchanged, then adds an M register or immediate value, updating the I register value.

The DAG post-modify addressing type can be used to emulate the push (save of registers) to a SW stack.

Listing 6-4. Post-Modify Addressing

```
BIT CLR MODE1 CBUFEN; /* clear circular buffer*/
nop;
I1 = buffer; /* Index Pointer */
M1 = 1; /* Modify */
instruction; /* stall, any non-DAG instruction */
instruction; /* stall, any non-DAG instruction */
R3 = dm(I1,M1); /* 1st access */
R3 = dm(I1,M1); /* 2nd access */
```

Modify Instruction

The DAGs support two operations that modify an address value in an index register without outputting an address. These two operations, address bit-reversal and address modify, are useful for bit-reverse addressing and maintaining pointers.

The MODIFY instruction modifies addresses in any DAG index register (I0-I15) without accessing memory.
**DAG Instruction Types**

The syntax for the `MODIFY` instruction is similar to post-modify addressing (index, then modifier). The `MODIFY` instruction accepts either a 32-bit immediate value or an M register as the modifier. The following example adds 4 to I1 and updates I1 with the new value.

```plaintext
MODIFY(I1,4);
```

If the I register’s corresponding B and L registers are set up for circular buffering, a `MODIFY` instruction performs the specified buffer wraparound (if needed).

The `MODIFY` instruction executes independent of the state of the CBUFEN bit. The `MODIFY` instruction always performs circular buffer modify of the index registers if the corresponding B and L registers are configured, independent of the state of the CBUFEN bit.

**Enhanced Modify Instruction (ADSP-214xx)**

An enhanced version of the `MODIFY` instruction is: `Ib = MODIFY(Ia,Mc);`. This instruction loads the modified index pointer into another index register. If the source and destination registers are the same, circular buffer behaves like the traditional modify (Ia, Mc) instruction.

If the source and destination registers are different `Ib = MODIFY(Ia,Mc);` the source register (Ia) is not updated, the destination register (Ib) does only change with absolute offsets.

**Immediate Modify Instruction**

Instructions can also use a number (immediate value), instead of an M register, as the modifier. The size of an immediate value that can modify an I register depends on the instruction type. For all single data access operations, modify immediate values can be up to 32 bits wide. Instructions that combine DAG addressing with computations limit the size of the modify immediate value. In these instructions (multifunction computa-
Data Address Generators

The modify immediate values can be up to 6 bits wide. The following example instruction accepts up to 32-bit modifiers:

\[
R1 = DM(0x40000000, I1); \quad /* \text{DM address} = I1 + 0x40000000 */
\]

The following example instruction accepts up to 6-bit modifiers:

\[
PM(I8, 0x0B) = ASTATx; \quad /* \text{PM address} = I8, I8 = I8 + 0x0B */
\]

**Bit-Reverse Instruction**

The BITREV instruction modifies and bit-reverses addresses in any DAG index register (I0–I15) without accessing memory. This instruction is independent of the bit-reverse mode. The BITREV instruction adds a 32-bit immediate value to a DAG index register, bit-reverses the result, and writes the result back to the same index register. The following example adds 4 to I1, bit-reverses the result, and updates I1 with the new value:

\[
\text{BITREV}(I1, 4);
\]

The processor does support bit-reverse mode. For more information, see “Operating Modes” on page 6-18.

**Enhanced Bit-Reverse Instruction (ADSP-214xx)**

An enhanced version of the BITREV instruction, that loads the bit reversed index pointer into another index register is shown below:

\[
I6 = \text{BITREV}(I1, 0);
\]

**Dual Data Move Instructions**

The number of transfers that occur in a clock cycle influences the data access operation. As described in “Internal Memory Space” on page 7-11, the processor supports single cycle, dual-data accesses to and from internal
DAG Instruction Types

memory for register-to-memory and memory-to-register transfers. Dual-data accesses occur over the PM and DM bus and act independent of SIMD/SISD mode setting. Though only available for transfers between memory and data registers, dual-data transfers are extremely useful because they double the data throughput over single-data transfers.

Note that the explicit use of complementary registers (CDREG) is not supported for dual data access.

On the ADSP-21367, ADSP-21368, and ADSP-21369 processors, it is illegal to use the DAGs in Type 1 instructions with the DM and PM buses both accessing external memory space.

```
R8 = DM(I4,M3), PM(I12,M13) = R0; /* Dual access */
R0 = DM(I5,M5);/* Single access */
```

For examples of data flow paths for single and dual-data transfers, see Chapter 2, Register Files.

The processor can use its complementary registers explicitly in SIMD mode. They support single data access as shown in the example below.

```
S8 = DM(I4,M3);
PM(I12,M13) = S12;
COMP, S8 = DM(I5,M5);
COMP, DM(I5,M5) = S14;
```

Conditional DAG Transfers

Conditions with DAG transfers allows programs to make memory accesses conditional. For more information see Chapter 4, Program Sequencer.

DAG Breakpoint Units

Both DAGs are connected to the breakpoint units used for hardware breakpoints. They are used if user breakpoints are enabled. For more information, Chapter 8, JTAG Test Emulation Port.
**Data Address Generators**

**DAG Instruction Restrictions**

Modify (M) registers can work with any index (I) register in the same DAG (DAG1 or DAG2).

The DAGs does allow transfers between the two DAG registers as in the following example.

```
DM(M2, I1) = I12;
L7 = PM(M12, I12);
```

However, transfers to the same DAG registers are not allowed and the assembler returns an error message.

```
DM(M2, I1) = I0;  /* generates asm error */
```

**Instruction Summary**

Table 6-3 through Table 6-8 list the DAG instructions. For more information on assembly language syntax, see Chapter 9, Instruction Set Types, and Chapter 10, Computation Instructions. In these tables, note the meaning of the following symbols:

- **I15–8** indicates a DAG2 index register: I15, I14, I13, I12, I11, I10, I9, or I8, and **I7–0** indicates a DAG1 index register I7, I6, I5, I4, I3, I2, I1, or I0.

- **M15–8** indicates a DAG2 modify register: M15, M14, M13, M12, M11, M10, M9, or M8, and **M7–0** indicates a DAG1 modify register M7, M6, M5, M4, M3, M2, M1, or M0.

- **Ureg** indicates any universal register.

- **Dreg** indicates any data register.
<Data32> indicates any immediate 32-bit value, and <Data6> indicates any immediate 6-bit value.

**Table 6-2. Pre-Modify Addressing, I + M Register, no Updating I Register**

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>DM(M7–0,17–0) = Ureg (LW); /* DAG1 */</td>
</tr>
<tr>
<td>PM(M15–8,I15–8) = Ureg (LW); /* DAG2 */</td>
</tr>
<tr>
<td>Ureg=DM(M7–0,17–0) (LW); /* DAG1 */</td>
</tr>
<tr>
<td>Ureg=PM(M15–8,I15–8) (LW); /* DAG2 */</td>
</tr>
<tr>
<td>DM(M7–0,17–0) = &lt;Data32&gt;; /* DAG1 */</td>
</tr>
<tr>
<td>PM(M15–8,I15–8) = &lt;Data32&gt;; /* DAG2 */</td>
</tr>
</tbody>
</table>

**Table 6-3. Post-Modify Addressing, Modified by M Register and Updating I Register**

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>DM(I7–0,M7–0) = Ureg (LW); /* DAG1 */</td>
</tr>
<tr>
<td>PM(I15–8,M15–8) = Ureg (LW); /* DAG2 */</td>
</tr>
<tr>
<td>Ureg=DM(I7–0,M7–0) (LW); /* DAG1 */</td>
</tr>
<tr>
<td>Ureg=PM(I15–8,M15–8) (LW); /* DAG2 */</td>
</tr>
<tr>
<td>DM(I7–0,M7–0) = Data32; /* DAG1 */</td>
</tr>
<tr>
<td>PM(I15–8,M15–8) = Data32; /* DAG2 */</td>
</tr>
</tbody>
</table>
Table 6-4. Post-Modify Addressing, Modified by 6-Bit Data and Updating I Register

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DM(I7–0,&lt;Data6&gt;) = Dreg; /* DAG1 */</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PM(I15–8,&lt;Data6&gt;) = Dreg; /* DAG2 */</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg = DM(I7–0,&lt;Data6&gt;); /* DAG1 */</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreg = PM(I15–8,&lt;Data6&gt;); /* DAG2 */</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6-5. Update (Modify) I Register, Modified by 32-Bit Data

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Modify(I7–0,&lt;Data32&gt;); /* DAG1 */</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modify(I15–8,&lt;Data32&gt;); /* DAG2 */</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6-6. Update (Modify) I Register, Modified by M Register

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>I7–0=Modify(I7–0,M7–0); /* DAG1 */ (ADSP-214xx processors only)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I15–8=Modify(I15–8,M15–8); /* DAG2 */ (ADSP-214xx processors only)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6-7. Bit-Reverse and Update I Register, Modified By 32-Bit Data

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitrev(I7–0,&lt;Data32&gt;); /* DAG1 */</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitrev(I15–8,&lt;Data32&gt;); /* DAG2 */</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6-8. Bit-Reverse and Update I Register, Modified By Immediate Data

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>I7–0 = Bitrev(I7–0,&lt;Data32&gt;); /* DAG1 */ (ADSP-214xx processors only)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I15–8 = Bitrev(I15–8,&lt;Data32&gt;); /* DAG2 */ (ADSP-214xx processors only)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Operating Modes

This section describes all modes related to the DAG which are enabled by a control bit in the MODE1, MODE2 and SYSCTL registers.

Normal Word (40-Bit) Accesses

A program makes an extended-precision normal word (40-bit) access to internal memory using an access to a normal word address when that memory block’s IMDWx bit is set (=1) for 40-bit words. The address ranges for internal memory accesses appear in the product specific data sheet. For more information on configuring memory for extended-precision normal word accesses, see “Extended-Precision Normal Word Addressing of Single-Data” on page 7-45.

The processor transfers the 40-bit data to internal memory as a 48-bit value, zero-filling the least significant 8 bits on stores and truncating these 8 bits on loads. The register file source or destination of such an access is a single 40-bit data register as shown in Listing 6-5.

Table 6-9. Dual Data DAG Transfers for Post Modify

<table>
<thead>
<tr>
<th>Description</th>
<th>DAG Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>DREG = DM(I7-0,M7-0), DREG = PM(I15-8,M15-8); /* both DAG read */</td>
<td></td>
</tr>
<tr>
<td>DM(I7-0,M7-0) = DREG, PM(I15-8,M15-8) = DREG; /* both DAG write */</td>
<td></td>
</tr>
<tr>
<td>DREG = DM(I7-0,M7-0), PM(I15-8,M15-8) = DREG; /* DAG1 read, DAG2 write */</td>
<td></td>
</tr>
<tr>
<td>DM(I7-0,M7-0) = DREG, DREG = PM(I15-8,M15-8); /* DAG2 read, DAG1 write */</td>
<td></td>
</tr>
<tr>
<td>DREG = DM(I1,M7-0); /* DAG1 broadcast read */</td>
<td></td>
</tr>
<tr>
<td>DREG = PM(I9,M15-8); /* DAG2 broadcast read */</td>
<td></td>
</tr>
<tr>
<td>DREG = DM(I1,M7-0), DREG = PM(I9,M15-8); /* both DAG broadcast read */</td>
<td></td>
</tr>
</tbody>
</table>
**Data Address Generators**

Listing 6-5. Normal Word (40-Bit) Accesses

```
bit clr MODE1 CBUFEN;
nop;
I9=0x90500;    /* start of 40-bit block 0 */
M9=1;
I5=0xB8000;   /* start of 32-bit block 1 */
M5=1;
USTAT1 = dm(SYSCTL);
bit set USTAT1 IMDW0; /* Blk0 access 40-bit precision */
dm(SYSCTL) = USTAT1;
NOP;            /* effect latency */
DM(I5,M5)=R0, PM(I9,M9)=R4; /* DAG1 32-bit, DAG2 40-bit */
```

Note that the sequencer uses 48-bit memory accesses for instruction fetches. Programs can make 48-bit accesses with PX register moves, which default to 48 bits. For more information, see Chapter 2, Register Files.

Listing 6-6. Input Sections Definition for 32/40-bit Data Access in LDF File

```
/* block 0 */
seg_pmco       /* TYPE(PM RAM) START(0x00090200) END(0x000904FF)
                WIDTH(48) */
seg_pmda_40    /* TYPE(PM RAM) START(0x00090500) END(0x00090FFF)
                WIDTH(48) */
/* block 1 */
seg_dmda_32    /* TYPE(DM RAM) START(0x000B8000) END(0x000B87FF)
                WIDTH(32)*/
```

**Circular Buffering Mode**

The **CBUFEN** bit in the **MODE1** register enables circular buffering—a mode where the DAG supplies addresses that range within a constrained buffer.
length (set with an \texttt{L} register). Circular buffers start at a base address (set with a \texttt{B} register), and increment addresses on each access by a modify value (set with an \texttt{M} register).

The circular buffer enable bit (\texttt{CBUFEN}) in the \texttt{MODE1} register is cleared (= 0) at processor reset.

\begin{itemize}
\item On previous SHARC processors (ADSP-2116x), circular buffering is always enabled. For code compatibility, programs ported to the ADSP-2136x processors should include the instruction:
\begin{verbatim}
Bit Set Mode1 CBUFEN;
\end{verbatim}
\end{itemize}

For more information on setting up and using circular buffers, see “Circular Buffering Mode” on page 6-19. When using circular buffers, the DAGs can generate an interrupt on buffer overflow (wraparound). For more information, see “DAG Status” on page 6-32.

The DAGs support addressing circular buffers. This is defined as addressing a range of addresses which contain data that the DAG steps through repeatedly, \textit{wrapping around} to repeat stepping through the range of addresses in a circular pattern. To address a circular buffer, the DAG steps the index pointer (\texttt{I} register) through the buffer, post-modifying and updating the index on each access with a positive or negative modify value (\texttt{M} register or immediate value). If the index pointer falls outside the buffer, the DAG subtracts or adds the buffer length to the index value, wrapping the index pointer back within the start and end boundaries of the buffer. The DAG’s support for circular buffer addressing appears in Figure 6-1 on page 6-3, and an example of circular buffer addressing appears in Figure 6-6 and Figure 6-7.

The starting address that the DAG wraps around is called the buffer’s base address (\texttt{B} register). There are no restrictions on the value of the base address for a circular buffer.

\begin{itemize}
\item Circular buffering starting at any address may only use post-modify addressing.
\end{itemize}
It is important to note that the DAGs do not detect memory map overflow or underflow. If the address post-modify produces \( I + M > 0xFFFF \) or \( I - M < 0 \), circular buffering may not function correctly. Also, the length of a circular buffer should not let the buffer straddle the top of the memory map. For more information on the processor’s memory map, see “Internal Memory Space” on page 7-11 and the product specific data sheet.

### Circular Buffer Programming Model

As shown in Figure 6-6, programs use the following steps to set up a circular buffer:

1. **Enable circular buffering** (\texttt{BIT SET MODE1 CBUFEN};). This operation is only needed once in a program.

2. **Load the buffer’s base address into the \( B \) register.** This operation automatically loads the corresponding \( I \) register. If an offset is required the I register can be changed accordingly.

3. **Load the buffer’s length into the corresponding \( L \) register.** For example, \( L0 \) corresponds to \( B0 \).

4. **Load the modify value (step size) into an \( M \) register in the corresponding DAG.** For example, \( M0 \) through \( M7 \) correspond to \( B0 \). Alternatively, the program can use an immediate value for the modifier.
**Operating Modes**

Figure 6-6. Circular Data Buffers With Positive Modifier

Figure 6-7 shows a circular buffer with the same syntax as in Figure 6-6, but with a negative modifier ($M_1 = -4$).

Figure 6-7. Circular Data Buffers With Negative Modifier
Data Address Generators

After circular buffering is set up, the DAGs use the modulus logic in Figure 6-1 on page 6-3 to process circular buffer addressing.

Using circular buffering with odd length in SIMD mode allows the implicit move to exceed the circular buffer limits.

Wraparound Addressing

When circular buffering is enabled, on the first post-modify access to the buffer, the DAG outputs the $I$ register value on the address bus then modifies the address by adding the modify value. If the updated index value is within limits of the buffer, the DAG writes the value to the $I$ register. If the updated value is outside the buffer limits, the DAG subtracts (for positive $M$) or adds (for negative $M$) the $I$ register value before writing the updated index value to the $I$ register. In equation form, these post-modify and wraparound operations work as follows.

- If $M$ is positive:
  - $I_{\text{new}} = I_{\text{old}} + M$ if $I_{\text{old}} + M < \text{Buffer base} + \text{length}$ (end of buffer)
  - $I_{\text{new}} = I_{\text{old}} + M - L$ if $I_{\text{old}} + M \geq \text{buffer base} + \text{length}$

- If $M$ is negative:
  - $I_{\text{new}} = I_{\text{old}} + M$ if $I_{\text{old}} + M \geq \text{buffer base}$ (start of buffer)
  - $I_{\text{new}} = I_{\text{old}} + M + L$ if $I_{\text{old}} + M < \text{buffer base}$ (start of buffer)

The DAGs use all four types of DAG registers for addressing circular buffers. These registers operate as follows for circular buffering.

- The index ($I$) register contains the value that the DAG outputs on the address bus.

- The modify ($M$) register contains the post-modify value (positive or negative) that the DAG adds to the $I$ register at the end of each memory access. The $M$ register can be any $M$ register in the same
Operating Modes

DAG as the \( I \) register and does not have to have the same number. The modify value can also be an immediate value instead of an \( M \) register. The size of the modify value, whether from an \( M \) register or immediate, must be less than the length (\( L \) register) of the circular buffer.

- The length (\( L \)) register sets the size of the circular buffer and the address range that the DAG circulates the \( I \) register through. The \( L \) register must be positive and cannot have a value greater than \( 2^{31} - 1 \). If an \( L \) register’s value is zero, its circular buffer operation is disabled.

- The DAG compares the base (\( B \)) register, or the \( B \) register plus the \( L \) register, to the modified \( I \) value after each access. When the \( B \) register is loaded, the corresponding \( I \) register is simultaneously loaded with the same value. When \( I \) is loaded, \( B \) is not changed. Programs can read the \( B \) and \( I \) registers independently.

Clearing the \texttt{CBUFEN} bit disables circular buffering for all data load and store operations. The DAGs perform normal post-modify load and store accesses, ignoring the \( B \) and \( L \) register values. Note that a write to a \( B \) register modifies the corresponding \( I \) register, independent of the state of the \texttt{CBUFEN} bit.

Broadcast Load Mode

The processor’s \texttt{BDCST1} and \texttt{BDCST9} bits in the \texttt{MODE1} register control broadcast register loading. When broadcast loading is enabled, the processor writes to complementary registers or complementary register pairs in each processing element on writes that are indexed with DAG1 register 11 (if \texttt{BDCST1 = 1}) or DAG2 register 19 (if \texttt{BDCST9 = 1}). Broadcast load accesses are similar to SIMD mode accesses in that the processor transfers both an explicit (named) location and an implicit (unnamed, complementary) location. However, broadcast loading only influences writes to registers and writes identical data to these registers.
Broadcast mode is independent of SIMD mode. Broadcast load mode is a hybrid between SISD and SIMD modes that transfers dual-data under special conditions.

Broadcast Load Mode performs memory reads only. Broadcast mode only operates with data registers (DREG) or complement data registers (CDREG). Enabling either DAG register to perform a broadcast load has no effect on register stores or loads to universal registers (Ureg). For example:

\[
\begin{align*}
R0 &= \text{DM}(i1, m1); & /* I1 load to R0 and S0 */ \\
S10 &= \text{PM}(i9, m9); & /* I9 load to S10 and R10 */ \\
\end{align*}
\]

Table 6-10 shows examples of Broadcast load instructions.

Table 6-10. Table 5-2. Instruction Summary Broadcast Load

<table>
<thead>
<tr>
<th>Explicit, PEx Operation</th>
<th>Implicit, PEy operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rx = dm(i1, ma);</td>
<td>$sx = dm(i1, ma);</td>
</tr>
<tr>
<td>Rx = pm(i9, mb);</td>
<td>$sx = pm(i9, mb);</td>
</tr>
<tr>
<td>Rx = dm(i1, ma), Ry = pm(i9, mb);</td>
<td>$sx = dm(i1, ma), $y = pm(i9, mb);</td>
</tr>
</tbody>
</table>

The PEYEN bit (SISD/SIMD mode select) does not influence broadcast operations. Broadcast loading is particularly useful in SIMD applications where the algorithm needs identical data loaded into each processing element. For more information on SIMD mode (in particular, a list of complementary data registers), see “Complementary Data Registers” on page 2-5.

Bit-Reverse Mode

The bit reserve mode is useful for FFT calculations, if using a DIT (decimation in time) FFT, all inputs must be scrambled before running the FFT, thus the output samples are directly interpretable. For DIF (decimation in frequency) FFT the process is reversed. This mode automates bit reversal, no specific instruction is required.
Operating Modes

The BR0 and BR8 bits in the MODE1 register enable the bit-reverse addressing mode where addresses are output in reverse bit order. When BR0 is set (= 1), DAG1 bit-reverses 32-bit addresses output from I0. When BR8 is set (= 1), DAG2 bit-reverses 32-bit addresses output from I8. The DAGs bit-reverse only the address output from I0 or I8; the contents of these registers are not reversed. Bit-reverse addressing mode effects post-modify operations.

Listing 6-7 demonstrates how bit-reverse mode effects address output.

Listing 6-7. Bit Reverse Addressing

```
BIT SET MODE1 BR0; /* Enables bit-rev. addressing for DAG1 */
I0 = 0x83000 /* Loads I0 with the bit reverse of the buffer's base address DM(0xC1000) */

MO = 0x4000000; /* Loads MO with value for post-modify, which is the bit reverse value of the modifier value MO = 32 */

R1 = DM(I0,MO); /* Loads R1 with contents of DM address DM(0xC1000), which is the bit-reverse of 0x83000, then post-modifies I0 for the next access with (0x83000 + 0x4000000) = 0x4083000, which is the bit-reverse of DM(0xC1020) */
```

SIMD Mode

When the PEYEN bit in the MODE1 register is set (=1), the processors are in single-instruction, multiple-data (SIMD) mode. In SIMD mode, many data access operations differ from the processor’s default single-instruction, single-data (SISD) mode. These differences relate to doubling the amount of data transferred for each data access.
For example, processing two channels in parallel requires a more complex data layout since all inputs and outputs for the two channels have to be interleaved—that is all even array elements represent one channel while all odd elements represent the other.

### DAG Transfers in SIMD Mode

Accesses in SIMD mode transfer both an explicit (named) location and an implicit (unnamed, complementary) location (Table 6-11). The explicit transfer is a data transfer between the explicit register and the explicit address, and the implicit transfer is between the implicit register and the implicit address.

<table>
<thead>
<tr>
<th>DAG Instruction</th>
<th>Post-Modify</th>
<th>Pre-Modify (M+I, no I update)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Explicit Access</td>
<td>Implicit Access</td>
</tr>
<tr>
<td>SISD</td>
<td>DM(Ia, Mb)</td>
<td>PM(Ic, Md)</td>
</tr>
<tr>
<td>SIMD NW 32-bit</td>
<td>DM(Ia+1, Mb)</td>
<td>PM(Ic+1, Md)</td>
</tr>
<tr>
<td>SIMD SW 16-bit</td>
<td>DM(Ia+2, Mb)</td>
<td>PM(Ic+2, Md)</td>
</tr>
<tr>
<td>Broadcast</td>
<td>DM(Ia, Mb)</td>
<td>PM(Ic, Md)</td>
</tr>
</tbody>
</table>

In SIMD mode, both aligned (explicit even address) and un-aligned (explicit odd address) transfers are supported.

- RO=DM(I1,M1): /* I1 points to NW space */
- SO=DM(I1+1,M1): /* implicit instruction */
- R10=PM(I10,M11): /* I10 points to SW space */
- S10=PM(I10+2,M11): /* implicit instruction */

DAGs support SIMD mode in Normal word (32-bit) and short word (16-bit) only.
Operating Modes

The DAG registers support the bidirectional register-to-register transfers that are described in "SIMD Mode" on page 3-36. When the DAG register is a source of the transfer, the destination can be a register file data register. This transfer results in the contents of the single source register being duplicated in complementary data registers in each processing element as shown below.

```
BIT SET MODE1 PEYEN;       /* SIMD */
NOP;                       /* effect latency */
R5 = I8;                     /* Loads R5 and S5 with I8 */
```

When the processors are in SIMD mode, if the DAG register is a destination of a transfer from a register file data register source, the processor executes the explicit move only on the condition in PEx becoming true, whereas the implicit move is not performed. This is also true when both the source and the destination is a DAG register.

```
BIT SET MODE1 PEYEN;        /* SIMD */
NOP;                        /* effect latency */
I8 = R5;                      /* Loads I8 with R5 */
```

Conditional DAG Transfers in SIMD Mode

Conditions in SIMD allows programs to make memory accesses conditional. For more information see Chapter 4, Program Sequencer.

```
IF EQ S8 = DM(I4,M3);         /* S8 load with I4, 
                             R8 load with I4+1*/
IF NOT AV PM(I12,M13) = S12;    /* I12 load with S12, 
                             I12+1 load with R12*/
```

Alternate (Secondary) DAG Registers

To facilitate fast context switching, the processor includes alternate register sets for all DAG registers. Bits in the MODE1 register control when alternate registers become accessible. While inaccessible, the contents of alternate registers are not affected by processor operations. Note that there
is a one cycle latency between writing to MODE1 and being able to access an alternate register set. The alternate register sets for the DAGs are described in this section. For more information on alternate data and results registers, see “Alternate (Secondary) Data Registers” on page 2-13.

Bits in the MODE1 register can activate alternate register sets within the DAGs: the lower half of DAG1 (I, M, L, B0–3), the upper half of DAG1 (I, M, L, B4–7), the lower half of DAG2 (I, M, L, B8–11), and the upper half of DAG2 (I, M, L, B12–15). Figure 6-8 shows the primary and alternate register sets of the DAGs.

Figure 6-8. DAG Primary and Alternate Registers
Operating Modes

To share data between contexts, a program places the data to be shared in one half of either the current data address generator’s registers or the other DAG’s registers and activates the alternate register set of the other half. The following examples demonstrate how the code handles the one cycle latency from the instruction that sets the bit in **MODE1** to when the alternate registers may be accessed. Note that programs can use a NOP instruction or any other instruction not related to the DAG to take care of this latency.

Example 1

```
BIT SET MODE1 SRD1L;  /* Activate alternate dag1 lo reg */
NOP;                   /* Wait for access to alternates */
R0 = DM(i0,m1);
```

Example 2

```
BIT SET MODE1 SRD1L;  /* activate alternate dag1 lo registers */
R13 = R12 + R11;      /* Any unrelated instruction */
R0 = DM(I0,M1);
```

Interrupt Mode Mask

On the SHARC processors, programs can mask automated individual operating mode bits in the **MODE1** register by entering into an ISR. This reduces latency cycles.

For the DAGs, the alternate registers (**SRD1L/H** and **SRD2L/H**), circular buffer (**CBUFEN**), bit-reverse (**BR0/8**) and broadcast (**BDCST1/9**) are optional masks in use. For more information, see Chapter 4, Program Sequencer.

Access Modes Summary

The following sections summarize the access modes supported by the DAGs.
**SISD Mode**

Programs can use odd or even modify values (1, 2, 3, …) to step through a buffer in single- or dual-data, SISD or broadcast load mode regardless of the data word size (long word, extended-precision normal word, normal word, or short word).

**SIMD Mode Normal Word**

Programs should use a multiple of 2 modify values (2, 4, 6, …) to step through a buffer of normal word data in single- or dual-data SIMD mode.

**SIMD Mode Short Word**

Programs should use a multiple of 4 modify values (4, 8, 12, …) to step through a buffer of short word data in single- or dual-data.

Note that programs must step through a buffer twice, once for addressing even short word addresses and once for addressing odd short word addresses.

**DAG Interrupts**

The DAG interrupt overview is shown in Table 6-12.

Table 6-12. DAG Interrupt Overview

<table>
<thead>
<tr>
<th>Interrupt Source</th>
<th>Interrupt Condition</th>
<th>Interrupt Priorities</th>
<th>Interrupt Acknowledge</th>
<th>IVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAG1, DAG2</td>
<td>–Index 7 overflow</td>
<td>30–31</td>
<td>RTI instruction</td>
<td>CB7I, CB15I</td>
</tr>
<tr>
<td></td>
<td>–Index 15 overflow</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

There is one set of registers (17 and 115) in each DAG that can generate an interrupt on circular buffer overflow (address wraparound). For more information, see “DAG Status” on page 6-32.
DAG Interrupts

When a program needs to use I7 or I15 without circular buffering and the processor has the circular buffer overflow interrupts unmasked, the program should disable the generation of these interrupts by setting the B7/B15 and L7/L15 registers to values that prevent the interrupts from occurring. If, for example, I7 were accessing the address range 0x1000 – 0x2000, the program could set B7 = 0x0000 and L7 = 0xFFFF. Because the processor generates the circular buffer interrupt based on the wraparound equations on page 6-23, setting the L register to zero does not necessarily achieve the desired results. If the program is using either of the circular buffer overflow interrupts, it should avoid using the corresponding I register(s) (I7 or I15) where interrupt branching is not needed.

There are two special situations to be aware of when using circular buffers:

1. In the case of circular buffer overflow interrupts, if CBUFEN = 1 and register L7 = 0 (or L15 = 0), then the CB7I (or CB15I) interrupt occurs at every change of I7 (or I15), after the index register (I7 or I15) crosses the base register (B7 or B15) value. This behavior is independent of the context of both primary and alternate DAG registers.

2. When a LW access, SIMD access, or normal word access with the LW option crosses the end of the circular buffer, the processor completes the access before responding to the end of buffer condition.

Enable interrupts and use an interrupt service routine (ISR) to handle the overflow condition immediately. This method is appropriate if it is important to handle all overflows as they occur; for example in a “ping-pong” or swap I/O buffer pointers routine.

DAG Status

The DAGs can provide buffer overflow information when executing circular buffer addressing for the I7 or I15 registers. When a buffer overflow occurs (a circular buffering operation increments the I register past the end of the buffer or decrements below the start of the buffer), the appro-
Data Address Generators

appropriate DAG updates a buffer overflow flag in a sticky status (STKYx) register. Use the \texttt{BIT TST} instruction to examine overflow flags in the \texttt{STKY} register after a series of operations. If an overflow flag is set, the buffer has overflowed or wrapped around at least once. This method is useful when overflow handling is not time sensitive.
DAG Interrupts
7 MEMORY

The SHARC processors contain up to 5M bits of internal RAM and up to 4M bits of internal ROM. This memory is organized into four independent single ported memory blocks. This organization allows greater system flexibility in regards to code, data and stack or heap allocation. For information about the maximum number of data or instruction words that can fit into internal memory, see the processor specific data sheet.

Features

The following are the memory interface features.

- Four independent internal memory blocks comprised of RAM and ROM.
- Each block can be configured for different combinations of code and data storage.
- Each block consists of four columns and each column is 16 bits wide.
- Each block maps to separate regions in memory address space and can be accessed as 16-bit, 32-bit, 48-bit, or 64-bit words.
- Each block also has its own two-deep self clearing shadow write buffers with automatic hit detection and data forwarding logic for read access.
**Von Neuman Versus Harvard Architectures**

- Memory aliasing allows inter access of same space from different word sizes
- Block 0 has 256 addresses reserved for internal interrupt vector table (IVT), controller jump after interrupt latch to a specific IVT address.

While each memory block can store combinations of code and data, accesses are most efficient when one block stores data using the DM bus, for transfers, the second block stores instructions and data using the PM bus and a third and fourth block stores data using the I/O bus. Using the DM and PM buses in this way assures single-cycle execution with two data transfers. In this case, the instruction must be available in the cache.

**Von Neuman Versus Harvard Architectures**

Most microprocessors use a single address and a single-data bus for memory accesses. This type of memory architecture is referred to as the Von Neumann architecture. Because processors require greater data throughput than the Von Neumann architecture provides, many processors use memory architectures that have separate data and address buses for program and data storage. These two sets of buses let the processor retrieve a data word and an instruction simultaneously. This type of memory architecture is called Harvard architecture.

**Super Harvard Architecture**

SHARC processors go a step further by using a Super Harvard architecture. This four bus architecture has two address buses and two data buses, but provides a single, unified address space for program and data storage. While the data memory (DM) bus only carries data, the program memory (PM) bus handles instructions and data, allowing dual-data accesses.
The following code examples and Table 7-1 illustrate the differences between Harvard and Super Harvard capabilities.

**Standard Harvard Architecture**

```
Compute, r0=dm(i0,m0): /* instruction performs 2 accesses */
/* cycle4: IF (PM) at n+3 (Fetch1) and DF (DM) at n (Address)*/
```

**Super Harvard Architecture**

```
Compute, r0=dm(i0,m0), r1=pm(i8,m8): /* instruction performs 3 accesses */
/* cycle4: IF (PM) at n+3 (Fetch1) and DF (DM AND PM) at n (Address)*/
```

Table 7-1 illustrates multiple accesses in the instruction pipeline.

**Table 7-1. Pipelined Execution Cycles**

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td></td>
<td></td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td></td>
<td></td>
<td>n</td>
<td></td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
</tr>
<tr>
<td>Decode</td>
<td></td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
<td></td>
</tr>
<tr>
<td>Fetch2</td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
<td>n+7</td>
<td></td>
</tr>
<tr>
<td>Fetch1</td>
<td>n</td>
<td>n+1</td>
<td>n+2</td>
<td>n+3</td>
<td>n+4</td>
<td>n+5</td>
<td>n+6</td>
<td>n+7</td>
<td>n+8</td>
</tr>
</tbody>
</table>

When instructions and data passing over the PM bus cause a conflict, the conflict cache resolves them using hardware that act as a third bus feeding the sequencer’s pipeline with instructions.

Processor core and I/O processor accesses to internal memory are completely independent and transparent to one another. Each block of memory can be accessed by the processor core and I/O processor in every cycle provided the access is to different block of the memory.
Functional Description

Address Decoding of Memory Space

The SHARC processor’s memory maps appears in the processor specific data sheet and shows three memory spaces: internal memory space, external memory space, and I/O processor space. These spaces have the following definitions:

- **I/O processor Space.** The I/O processor’s memory-mapped registers control the system configuration of the processor and I/O operations. For information about the I/O processor, see the product specific hardware reference. These registers occupy consecutive 32-bit locations in this region. For information on IOP memory space, please refer to the processor specific hardware reference and data sheet.

- **Internal memory space.** Internal memory space refers to the processor’s on-chip RAM, on-chip ROM, memory-mapped registers and reserved memory space.

- **External memory space.** External memory space refers to the external memories (SRAM, SDRAM, DDR2, FLASH or FIFO). For information on external memory space please refer to the processor specific hardware reference and data sheet.

- **Shared memory bank space.** The ADSP-21368 processor supports shared memory space which allows sharing of external memory space among multiple processors using hardware arbitration. For more information refer to the processor specific hardware reference and the data sheet.
Figure 7-1 shows how the memory map addresses the different memory regions.

**I/O Processor Space**

The IOP register space is the address space where the core or peripheral’s control, status or address memory-mapped registers are located. This region (0x0000 0000 to 0x0003 FFFF) is divided into 2 clock domains:

- IOP core registers (core clock domain, CCLK).
- IOP peripheral registers (peripheral clock domain, PCLK = CCLK/2).

**IOP Peripheral Registers**

All writes to IOP peripheral register space pass through a bridge (CCLK to PCLK) as shown in Figure 7-2 and Figure 7-3. The bridge contains a write buffer to hold the write address and data. After the core has written to the bridge, it is the bridge’s responsibility to complete a write access (which
Functional Description

allows pipelined accesses. The write access takes one core clock cycle (CCLK). Since the CCLK to PCLK ratio is 1:2, the core IOP register access can occur during rising or falling edge of PCLK. The rising edge takes four (best case) and falling edge takes five (worst case) CCLK cycles to terminate the write. The newly written value to the IOP register can be read back on the next instruction.

Figure 7-2. Memory and Internal Buses Block Diagram (ADSP-21362/3/4/5/6 Only)
IOP Core Registers

 Writes take effect without any stalls, whereas a read needs two core clock cycles. The bridge (CCLK to PCLK) decodes the address from the core and generates the read/write strobes for the respective registers. The core itself handles the data.

---

SHARC Processor Programming Reference 7-7
Functional Description

Writes to IOP Peripheral Registers

Writes to IOP peripheral registers can occur on the positive or negative \textit{PCLK} edge.

IOP peripheral registers have a write latency of minimum of 4 and a maximum of 5 \textit{CCLK} cycles to terminate.

Back to Back Writes to IOP Peripheral Registers

If the core requests continuously the bridge, it stalls for one core cycle for each write starting with the second. Therefore, each write takes two cycles except for the first, which takes just one.

Alternate Writes to IOP Peripheral Registers

When the core requests a write once in every cycle of \textit{PCLK} clock, (every alternate \textit{CCLK} cycle) then writes occur without stalls.

Reads from IOP Peripheral Registers

Single reads take 7 or 8 core cycles, depending on whether the request starts in the positive or negative half of the \textit{PCLK} cycle. Reads are not pipelined and so back to back reads behave in the same way as isolated reads. However irrespective of whether the first read begins in positive or negative \textit{PCLK}, the rest of the reads align themselves to the negative edge of \textit{PCLK}.

IOP Register Core Access

Table 7-2 illustrates the different access times for the core to any IOP register.

Accesses to IOP registers (from the processor core) should not use Type 1 (dual access) or LW or forced LW instructions.
Memory

Table 7-2. I/O Processor Access Conditions

<table>
<thead>
<tr>
<th>Access Type</th>
<th>Core domain (core cycles)</th>
<th>Peripheral domain (core cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOP register write/read</td>
<td>1/2</td>
<td>1/8</td>
</tr>
<tr>
<td>IOP register back-to-back write/read</td>
<td>1/2</td>
<td>2/8</td>
</tr>
<tr>
<td>Conditional IOP register write/read</td>
<td>1/2</td>
<td>3/10</td>
</tr>
<tr>
<td>Aborted IOP register write/read</td>
<td>2/3</td>
<td>4/4</td>
</tr>
</tbody>
</table>

Note that an atomic write and read from the same IOP peripheral register takes 11 (best case) or 13 (worst case) CCLK cycles. The following additional information about access to peripheral data buffers should be noted.

- Attempting to write to a full (or read from empty) peripheral data buffer causes the core to hang indefinitely, unless the BHD (buffer hang disable) bit for that peripheral is set.
- In case of a full transmit buffer, the held-off I/O processor register read or write access incurs one extra core-clock cycle.
- Interrupted IOP register reads and writes, if preceded by another write creates one additional core stall cycle.

Out of Order Execution

In the next examples different effect latencies are shown. Because the SPI control write (N+1) requires 4–5 CCLK cycles to have an effect but the next access to a system register (SREG) (N+2) does not pass the bridge (non memory-mapped) and therefore pipelining may affect the next instruction executed before the previous one. The following example would cause pipeline execution problems.

N:r0=SPIEN;
N+1:dm(SPICTL)=r0;
N+2:bit CLR FLAGS FLG0;
To prevent out of order instruction execution the above code can be modified to:

\[
\begin{align*}
N & : r0 = SPIEN; \\
N+1 & : dm(SPICTL) = r0; \\
N+2 & : nop; nop; nop; nop; \\
N+7 & : bit CLR FLAGS FLG0;
\end{align*}
\]

or:

\[
\begin{align*}
N & : r0 = SPIEN; \\
N+1 & : dm(SPICTL) = r0; \\
N+2 & : r10 = dm(SPICTL); /* dummy read forces previous write \\
N+3 & : bit CLR FLAGS FLG0; \\
N+7 & : bit CLR FLAGS FLG0;
\end{align*}
\]

**IOP Register Access Arbitration**

All of the peripherals supporting DMA have two ports—one for core accesses and one for DMA accesses. While these registers act as memory-mapped locations, they are separate from the processor’s internal memory and have different bus accesses. One bus can access one I/O processor register at a time. (A typical situation occurs if the core reads or writes to the same register set used by the active chained DMA channel).

When there is contention among the buses for access to the same I/O processor register, the peripheral performs the following arbitration:

1. DMD bus accesses (highest priority)
2. PMD bus accesses
3. IOD0 or IOD1 bus accesses (lowest priority)

Internal memory block access arbitration is different—the highest priority favors IOD0 followed by IOD1, DMD and finally the PMD bus.
Internal Memory Space

The SHARC processor's internal memory block space is divided into four blocks—block 0 through block 3. RAM and ROM memory space and addressing varies by processor model and is available in the product specific data sheet.

Internal Memory Interface

The internal memory interface is responsible for all address and strobe generation for internal memory accesses. It also performs the necessary 48-bit address rotation, pin multiplexing and other interface tasks for instruction fetch or 40-bit data access. All data writes to the internal memory blocks pass a shadow write FIFO logic. Apart from performing memory accesses, the interface also performs bus-switching for the various buses. The crossbar switches between all buses; DMD, PMD, IOD0 and IOD1 to the single ported memory blocks.

On-Chip Buses

The processor has up to four sets of internal buses connected to its single-ported memory, the program memory (PM), data memory (DM), and I/O processor (IOP) buses. The IOP bus is designed to run only at half the core clock frequency. The three buses share the single port on each of the four memory blocks. Memory accesses from the processor's core (computational units, data address generators, or program sequencer) use the PM or DM buses, while the I/O processor uses the IOP bus for memory accesses. The I/O processor can access external memory devices. For more information about the external memory and I/O capabilities of the processor, see the product specific hardware reference. Figure 7-2 on page 7-6 and Figure 7-3 on page 7-7 show the bus structures of the ADSP-21362/3/4/5/6 processors and the ADSP-21367/8/9 and later products respectively.
Internal Memory Block Architecture

Because the processor’s internal memory is organized as four 16-bit wide by 64K high columns, memory is addressable in widths that are multiples of columns up to 64 bits:

- 1 column = 16-bit words
- 2 columns = 32-bit words
- 3 columns = 48- or 40-bit words
- 4 columns = 64-bit words

Each block is physically comprised of four 16-bit columns. Wrapping, as shown in Figure 7-10 on page 7-30, is a method where memory can efficiently store different combinations of 16-bit, 32-bit, 48-bit or 64-bit wide words.

The width of the data word fetched from memory is dependent upon the address range used. The same physical location in memory can be accessed using four different addresses.

These columns of memory are addressable as a variety of word sizes:

- 64-bit long word (LW) data (four columns)
- 48-bit instruction words or 40-bit extended-precision normal word (NW) data (3 columns)
- 32-bit normal word data (2 columns)
- 16-bit short word (SW) data (1 column)

Extended-precision normal word (40-bit) data is only accessible if the \texttt{IMDWx} bit is set in the \texttt{SYSCTL} register. It is left-justified within a three column location, using bits 47–8 of the location.
After power-up the content of the SRAM memory is not predictable.

**Normal Word Space 48/40-Bit Word Rotations**

When the processor core addresses memory, the word width of the access determines which columns within the memory are accessed. For instruction word (48 bits) or extended-precision normal word data (40 bits), the word width is 48 bits, and the processor accesses the memory’s 16-bit columns in groups of three. Because these sets of three column accesses are packed into a 4 column matrix, there are four rotations of the columns for storing 40- or 48-bit data. The three column word rotations within the four column matrix appear in Figure 7-4.

![Figure 7-4. 48-Bit Word Rotations](image)

Extended precision floating-point (40-bit) data and instruction fetches (48-bit) need a different type of manipulation of their addresses to derive the corresponding row addresses. Since each row contains 4 columns while 48-bit words span across 3 columns, the address is multiplied by ¾ (add address to its left-shifted version, right-shift the result by two bit-positions) to derive the first row address. The next address is the incremented version of the first one. Note that this assumes that the beginning addresses of 48-bit/32-bit/64-bit addresses align.
Functional Description

For long word (64 bits), normal word (32 bits), and short word (16 bits) memory accesses, the processor selects from fixed columns in memory. No rotations of words within columns occur for these data types.

Word rotation across subsequent row addresses is only required in the NW space for 48-bit instruction fetch or extended precision floating point mode.

Figure 7-5 shows the memory ranges for each data size in the processor’s internal memory.

Rules for Wrapping Memory Layout

The following sections describe memory wrapping, a method where programs can efficiently store different combinations of 16-bit, 32-bit, 48-bit or 64-bit wide words.

Mixing Words in Normal Word Space

The processor’s memory organization lets programs freely place memory words of all sizes (see “Internal Memory Block Architecture” on page 7-12) with few restrictions (see “Mixing 32-Bit Words and 48-Bit Words” on page 7-16). This memory organization also lets programs mix (place in adjacent addresses) words of all sizes. This section discusses how to mix odd (three column) and even (four column) data words in the processor’s memory.

Transition boundaries between 48-bit (three column) data and any other data size can occur only at any 64-bit address boundary within either internal memory block. Depending on the ending address of the 48-bit words, there are zero, one, or two empty locations at the transition between the 48-bit (three column) words and the 64-bit (four column) words. These empty locations result from the column rotation for storing 48-bit words. The three possible transition arrangements appear in Figure 7-5, Figure 7-6, and Figure 7-7.
Figure 7-5. Mixed Instructions and Data with No Unused Locations

Transitioning from 48-bit to 32-bit data with zero empty locations:
(48-bit word top address)

Addresses

Column 3 Column 2 Column 1 Column 0

32-bit word 3 32-bit word 2
32-bit word 1 32-bit word 0
48-bit word top-1
48-bit word top
48-bit word top-2
48-bit word top-3

Figure 7-6. Mixed Instructions and Data With One Unused Location

Transitioning from 48-bit to 32-bit data with one empty locations:
(48-bit word top address)

Addresses

Column 3 Column 2 Column 1 Column 0

32-bit word 3 32-bit word 2
32-bit word 1 32-bit word 0
Empty 48-bit word top
48-bit word top-1
48-bit word top-2
48-bit word top-3

SHARC Processor Programming Reference 7-15
Mixing 32-Bit Words and 48-Bit Words

There are some restrictions that stem from the memory column rotations for three column data (48 or 40-bit words) and they relate to the way that three column data can mix with two column data (32-bit words) in memory. These restrictions apply to mixing 48 and 32-bit words, because the processor uses a normal word address to access both of these types of data even though 48-bit data maps onto three columns of memory and 32-bit data maps onto two columns of memory.

When a system has a range of three column (48-bit) words followed by a range of two column (32-bit) words, there is often a gap of empty 16-bit locations between the two address ranges. The size of the address gap varies with the ending address of the range of 48-bit words. Because the addresses within the gap alias to both 48 and 32-bit words, a 48-bit write into the gap corrupts 32-bit locations, and a 32-bit write into the gap corrupts 48-bit locations. The locations within the gap are only accessible with short word (16-bit) accesses.
32-Bit Word Allocation

Calculating the starting address for two column data that minimizes the gap after three column data is useful for programs that are mixing three and two column data. Given the last address of the three column (48-bit) data, the starting address of the 32-bit range that most efficiently uses memory can be determined by the equation:

\[ m = B + \left(\frac{3}{2} (n - B)\right) + 1 \]

where:

- \( n \) is the first unused address after the end of 48-bit words
- \( B \) is the base normal word 48-bit address of the internal memory block
- \( m \) is the first 32-bit normal word address to use after the end of 48-bit words. For the ADSP-21367 memory layout:
  - block 0 = 0x80000 \# n \# 0x93FFF
  - block 1 = 0xA0000 \# n \# 0xB3FFF
  - block 2 = 0xC0000 \# n \# 0xC1554
  - block 3 = 0xE0000 \# n \# 0xE1554

Note that the linker verifies the wrapping rules of different output sections and returns an overlap error message during project build if the rules are violated.

Example: Calculating a Starting Address for 32-Bit Addresses

Given a block of words in the range 0x90000 to 0x92694 (block 0), the next valid address is 0x92695. The number of 48-bit words (n) is:

\[ n = 0x92695 - 0x80000 = 0x12695. \]
When 0x12695 is converted to decimal representation, the result is 75413.

The base (B) normal word address of the internal memory block is 0x80000. The first 32-bit normal word address to use after the end of the 48-bit words is given by:

\[ m = 0x80000 + \frac{3}{2} (75413) + 1 \]

\[ m = 0x80000 + 0x1B9E0 \]

\[ m = 0x80000 + 0x1B9E0 = 0x9B9E0 \]

The first valid starting 32-bit address is 0x9B9E0.

### 48-Bit Word Allocation

Another useful calculation for programs that are mixing two and three column data is to calculate the amount of three column data that minimizes the gap before starting four column data. Given the starting address of the two column (32-bit) data, the number of 48-bit words that most efficiently uses memory can be determined by the equation:

\[ n = B + \frac{2}{3} (m - B) - 1 \]

- \( m \) is the first 32-bit normal word address after the end of 32-bit words (1 \( m \) values falls in the valid normal word address space)
- \( B \) is the base normal word 48-bit address of the internal memory block
- \( n \) is the address of the first 48-bit word to use after the end of 32-bit words

### Memory Address Aliasing

For example, the long word address 0x4C000 corresponds to the same locations as normal word address 0x98000 and 0x98001. This also corresponds to the same locations as short word addresses 0x0013 0000,
0x0013 0001, 0x0013 0002 and 0x0013 0003. There are gaps in the memory map when using normal word addressing for 48-bit or 40-bit accesses. These gaps of missing addresses stem from the arrangement of this 3-column data in the memory.

As shown in Listing 7-1, accessing a short word memory address gets one 16-bit word. Consecutive 16-bit short-words are accessed from columns #1, #2, #3, #4, #1 and so on. Accessing a normal word memory address transfers 32 bits (from columns 1 and 2 or 3 and 4). Consecutive 32-bit words are accessed from columns 1 and 2, 3 and 4, 1 and 2 etc. Accessing a long word address transfers 64 bits (from all four columns). For example, the same 16 bits of Block-0 are overwritten in each of the following four write instructions (some, but not all of the short word accesses overwrite more than 16 bits).

Listing 7-1. Overwriting Bits

DM(0x0004C000) = R0;   /* long word transfer (64 bits/four columns) */
DM(0x00098000) = R0;   /* normal word transfer (32 bits/two columns) */
DM(0x00130000) = R0;   /* short word transfer (16 bits/1-column) */
USTAT1 = dm(SYSCTL);
bit set USTAT1 IMDW0;   /* set Blk0 access as ext. precision */
dm(SYSCTL) = USTAT1;
NOP; /* effect latency */
DM(0x00090000) = R0;    /* normal word transfer (40 bits/three columns) */

This mechanism is called address aliasing in that the same physical memory can be accessed using multiple addresses. This concept is essential to understand the memory operation.
Examples of memory address aliasing are:

- Boot instructions via DMA (32-bit NW) into memory block, fetch the instructions in 48-bit NW.
- Boot instructions via DMA (32-bit NW) into memory block, fetch the instructions in 16-bit SW.
- Shifter reads 32-bit NW floating-point data and stores 16-bit SW floating-point data.

Normal word address space is also used by the program sequencer to fetch 48-bit instructions. Note that a 48-bit fetch spans three columns that can lead to a different address range between instruction fetches and data fetches (Figure 7-1 on page 7-5).

Normal word address space can also optionally be used to fetch 40-bit data (from three columns) if the \texttt{IMDWx} (internal memory data width) bit in the \texttt{SYSCTL} register is set. There are four bits in the \texttt{SYSCTL} register, \texttt{IMDW0–3} that determine whether access to each block is 32 or 40 bits. For more information, see “SIMD Mode” on page 6-26.

Memory Block Arbitration

A memory access conflict can occur when the processor attempts two accesses to the same internal memory block in the same cycle. When this conflict, known as a block conflict occurs, the memory interface logic resolves it according the following rules. The instruction that causes this conflict may take two or three core clock cycles to complete execution.

1. Between DM and PM accesses, conflict is always resolved in favor of DM, with the PM access occurring in the second cycle.

2. Between IO0 and IO1 accesses, conflict is always resolved in favor to IO0, with the IO1 access occurring in the second cycle (for the ADSP-21367/8/9 and later SHARC processors.)
3. Between the core (DM/PM) and I/O (IO0/IO1) accesses, the conflict is resolved in favor of I/O. Note that since the I/O buses run at half the core clock frequency ($PCLK$), I/O accesses are requested at a maximum rate of once in two core clock cycles. This provides a fair sharing of memory access to the core and I/O buses.

During a single-cycle, dual-data access, the processor core uses the independent PM and DM buses to simultaneously access data from two memory blocks. Though dual-data accesses provide greater data throughput, it is important to note some limitations on how programs may use them. The limitations on single cycle, dual-data accesses are:

- The two pieces of data must come from different memory blocks.
- If the core accesses two words from the same memory block over the same bus in a single instruction, an extra cycle is needed.
- The data access execution may not conflict with an instruction fetch operation. The PM data bus tries to fetch an instruction in every cycle. If a data fetch is also attempted over the PM bus, an extra cycle may be required depending on the cache.
- If the cache contains the conflicting instruction, the data access completes in a single cycle and the sequencer uses the cached instruction. If the conflicting instruction is not in the cache, an extra cycle is needed to complete the data access and cache the conflicting instruction. For more information, see “Instruction Cache for External Instruction Fetch” on page 4-82.

For more information on how the buses access memory blocks, see “On-Chip Buses” on page 7-11.

Note that on previous SIMD SHARC processors (ADSP-2116x and ADSP-2126x) block conflicts between core and DMA do not occur because the memory blocks are dual-ported.
Functional Description

VISA Instruction Arbitration

With standard arbitration processes, 48-bits of data are fetched at a time. In VISA operation, this data may either be 1, 2, or 3 instructions. This is an advantage of VISA operation—during the execution of a typical VISA application there are fewer accesses to internal memory from the core, causing less conflict on the internal buses with other peripheral DMAs or dedicated hardware accelerators using the same bus.

Using Single Ported Memory Blocks Efficiently

Since the newer SHARC processor’s are designed with four single-ported memory blocks, software needs to be designed so that data is continuously being processed and there are no memory block conflicts.

Typically data is pushed into memory using the DMA infrastructure. The core loads the data from memory, performs a computation, and stores the data back into memory. Then the DMA drives this data off-chip.

To ensure continuous data streams, mechanisms like ping-pong buffers, together with chained DMA transfers, can be implemented as shown in Figure 7-8. Designs should ensure that while the DMA moves data to the primary memory block, the core processes the secondary block’s data. Then, after the DMA interrupt is generated, the memory block processing between core and DMA is flipped which prevents memory block conflicts between the core and DMA.

For complete information on using DMA, see the product specific hardware reference, “I/O Processor” chapter.
Because the processor’s internal memory operates at high speeds, writes to the memory block do not go directly into the memory array, but rather to a two-deep FIFO called the shadow write FIFO. The four shadow FIFOs are located inside the internal memory interface block (Figure 7-2 and Figure 7-3) which is responsible for access control to the individual blocks.

This FIFO uses a non-read cycle (either a write cycle, or a cycle in which there is no access of internal memory) to load data from the FIFO into internal memory. When an internal memory write cycle occurs, the FIFO loads any data from a previous write into memory and accepts new data.

When writing into a memory block, the writes pass through the shadow write buffer. Note the shadow FIFO is self-clearing, the last two writes are moved at any point into the block array.
Interrupts

Data can be read from internal memory in either of the following ways.

1. From the shadow write FIFO (caused by immediately read of the same data after a write)

2. From the memory block

The operation of the shadow write FIFO is completely transparent to the user. The logic takes automatic control of SIMD, 32-bit NW to 40-bit NW, LW or unaligned access types.

External Memory Space

External memory space is product specific and only applies to products that have an external port. For more information refer to the product specific hardware reference manual and the product specific data sheet.

Interrupts

Table 7-3 provides an overview of interrupts associated with the SHARC memory.

Table 7-3. Memory Interrupts

<table>
<thead>
<tr>
<th>Source</th>
<th>Condition</th>
<th>Priorities (0–41)</th>
<th>Interrupt Acknowledge</th>
<th>IVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>- Illegal IOP access</td>
<td>2</td>
<td>RTI instruction</td>
<td>IICDI</td>
</tr>
<tr>
<td></td>
<td>- Unaligned 64-bit forced long word access</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Internal Interrupt Vector Table

The default location of the SHARC’s processor’s interrupt vector table (IVT) depends basically on the processor’s booting mode. When any external boot source is selected (FLASH, SPI, Link Port), the vector table
Memory

starts at the first internal RAM normal word address. If the boot mode is
selected to reserved boot mode on ROM based versions, the vector table
starts in ROM normal word address.

The internal interrupt vector table (IIVT) bit in the SYSCTL register over-
rides the default placement of the vector table. If IIVT is set (=1), the
interrupt vector table starts at internal RAM regardless of the booting
mode. If IIVT is cleared (=0), the IIVT starts in the internal ROM.

For information about processor booting, see the processor specific hard-
ware manual.

Illegal I/O Processor Register Access

The processor monitors I/O processor register access when the illegal I/O
processor register access (IIRAE) bit in the MODE2 register is set (=1). If
access to the IOP registers is detected, an illegal input condition detected
(IICDI) interrupt occurs. The interrupt is latched in the IRPTL register (see
"Interrupt Latch Register (IRPTL)" on page A-31) when a core access to
an IOP register occurs.

The I/O processor’s DMA controller cannot generate the IICDI
interrupt. For more information, see “Mode Control 2 Register
(MODE2)” on page A-7.

Unaligned Forced Long Word Access

The processor monitors for unaligned 64-bit memory accesses (access
from two successive rows) if the unaligned 64-bit memory accesses
(U64MAE) bit in the MODE2 register (bit 21) is set (=1). An unaligned access
is an odd numbered address normal word access that is forced to 64 bits
with the LW mnemonic. When detected, this condition is an input that can
cause an illegal input condition detected (IICDI) interrupt if the interrupt
is enabled in the IMASK register. For more information, see “Mode Control
2 Register (MODE2)” on page A-7.
The following code example shows the access for even and odd addresses. When accessing an odd address, the sticky bit is set to indicate the unaligned access.

```c
bit set mode2 U64MAE;       /* set bit for aligned or unaligned 64-bit access */

r0 = 0x11111111;
r1 = 0x22222222;
pm(0x98200) = r0(lw);       /* even address in 32-bit, access is aligned */
pm(0x98201) = r0(lw);       /* odd address in 32-bit, sticky bit is set */
```

Figure 7-9. Unaligned Long Word Accesses
Internal Memory Access Listings

The processor’s DM and PM buses support many combinations of register-to-memory data access options. The following factors influence the data access type:

- Size of words—short word, normal word, extended-precision normal word, or long word
- Number of words—single or dual-data move
- Processor mode—SISD, SIMD, or broadcast load

The following list shows the processor’s possible memory transfer modes and provides a cross-reference to examples of each memory access option that stems from the processor’s data access options.

These modes include the transfer options that stem from the following data access options:

- The mode of the processor: SISD, SIMD, or Broadcast Load
- The size of access words: long, extended-precision normal word, normal word, or short word
- The number of transferred words

To take advantage of the processor’s data accesses to three and four column locations, programs must adjust the interleaving of data into memory locations to accommodate the memory access mode. The following guidelines provide overviews of how programs should interleave data in memory locations. For more information and examples, see "Instruction
Programs can use odd or even modify values (1, 2, 3, …) to step through a buffer in single- or dual-data, SISD or broadcast load mode regardless of the data word size (long word, extended-precision normal word, normal word, or short word).

Programs should use a multiple of 4 modify values (4, 8, 12, …) to step through a buffer of short word data in single- or dual-data, SIMD mode. Programs must step through a buffer twice, once for addressing even short word addresses and once for addressing odd short word addresses.

Programs should use a multiple of 2 modify values (2, 4, 6, …) to step through a buffer of normal word data in single- or dual-data SIMD mode.

Programs can use odd or even modify values (1, 2, 3, …) to step through a buffer of long word or extended-precision normal word data in single- or dual-data SIMD modes.

Where a cross (†) appears in the PEx registers in any of the following figures, it indicates that the processor zero-fills or sign-extends the most significant 16 bits of the data register while loading the short word value into a 40-bit data register. Zero-filling or sign-extending depends on the state of the SSE bit in the MODE1 system register. For short word transfers, the least significant 8 bits of the data register are always zero.

Short Word Addressing of Single-Data in SISD Mode

Figure 7-10 shows the SISD single-data, short word addressed access mode. For short word addressing, the processor treats the data buses as four 16-bit short word lanes. The 16-bit value for the short word access is transferred using the least significant short word lane of the PM or DM.
data bus. The processor drives the other short word lanes of the data buses with zeros.

In SISD mode, the instruction accesses the PEx registers to transfer data from memory. This instruction accesses \texttt{WORD X0}, whose short word address has “00” for its least significant two bits of address. Other locations within this row have addresses with least significant two bits of “01”, “10”, or “11” and select \texttt{WORD X1}, \texttt{WORD X2}, or \texttt{WORD X3} from memory respectively. The syntax targets register \texttt{RX} in PEx.
Figure 7-10. Short Word Addressing of Single-Data in SISD Mode
Short Word Addressing of Dual-Data in SISD Mode

Figure 7-11 shows the SISD, dual-data, short word addressed access mode. For short word addressing, the processor treats the data buses as four 16-bit short word lanes. The 16-bit values for short word accesses are transferred using the least significant short word lanes of the PM and DM data buses. The processor drives the other short word lanes of the data buses with zeros.

In SISD mode, the instruction explicitly accesses $\text{PEx}$ registers. This instruction accesses $\text{WORD X0}$ in any block and $\text{WORD Y0}$ in any other block. Each of these words has a short word address with “00” for its least significant two bits of address. Other accesses within these four column locations have addresses with their least significant two bits as “01”, “10”, or “11” and select $\text{WORD X1/Y1}$, $\text{WORD X2/Y2}$, or $\text{WORD X3/Y3}$ from memory respectively. The syntax explicitly accesses registers $\text{RX}$ and $\text{RA}$ in $\text{PEx}$. 
Figure 7-11. Short Word Addressing of Dual-Data in SISD Mode
Short Word Addressing of Single-Data in SIMD Mode

Figure 7-12 shows the SIMD, single-data, short word addressed access mode. For short word addressing, the processor treats the data buses as four 16-bit short word lanes. The explicitly addressed (named in the instruction) 16-bit value is transferred using the least significant short word lane of the PM or DM data bus. The implicitly addressed (not named in the instruction, but inferred from the address in SIMD mode) short word value is transferred using the 47–32 bit short word lane of the PM or DM data bus. The processor drives the other short word lanes of the PM or DM data buses with zeros (31–16 bit lane and 63–48 bit lane).

The instruction explicitly accesses the register $RX$ and implicitly accesses that register’s complementary register, $SX$. This instruction uses a $PEx$ register with an $RX$ mnemonic. If the syntax named the $PEy$ register $SX$ as the explicit target, the processor uses that register’s complement $RX$ as the implicit target. For more information on complementary registers, see “SIMD Mode” on page 3-36.

Figure 7-12 shows the data path for one transfer. The processor accesses short words sequentially in memory. For more information on arranging data in memory to take advantage of this access pattern, see Figure 7-28 on page 7-60.
**Figure 7-12. Short Word Addressing of Single-Data in SIMD Mode**

---

### Internal Memory Access Listings

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>PM DATA BUS</th>
<th>DM DATA BUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-48</td>
<td>47-32</td>
<td>31-16</td>
</tr>
</tbody>
</table>

- **PM DATA BUS**: 63-48, 47-32, 31-16, 15-0
- **DM DATA BUS**: 0x0000, 0x0000, 0x0000, 0x0000

**Word Access**

<table>
<thead>
<tr>
<th>WORD Y11</th>
<th>WORD Y10</th>
<th>WORD Y9</th>
<th>WORD Y8</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD Y7</td>
<td>WORD Y6</td>
<td>WORD Y5</td>
<td>WORD Y4</td>
</tr>
<tr>
<td>WORD Y3</td>
<td>WORD Y2</td>
<td>WORD Y1</td>
<td></td>
</tr>
</tbody>
</table>

**NO ACCESS**

**Short Word Access**

- **RX**: DM(SHORT WORD X0 ADDRESS)
- **RA**: 39-24, 23-8, 7-0
- **SA**: 39-24, 23-8, 7-0
- **SX**: 39-24, 23-8, 7-0

**THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:**

- **RX = DM(SHORT WORD X0 ADDRESS);**
- ** Other instructions with similar data flows for SIMD, short word, single-data transfers are:**
  - **UREG = PM(SHORT WORD ADDRESS);**
  - **UREG = DM(SHORT WORD ADDRESS);**
  - **PM(SHORT WORD ADDRESS) = UREG;**
  - **DM(SHORT WORD ADDRESS) = UREG;**

---

SHARC Processor Programming Reference
Short Word Addressing of Dual-Data in SIMD Mode

Figure 7-13 shows the SIMD, dual-data, short word addressed access. For short word addressing, the processor treats the data buses as four 16-bit short word lanes. The explicitly addressed 16-bit values are transferred using the least significant short word lanes of the PM and DM data bus. The implicitly addressed short word values are transferred using the 47-32 bit short word lanes of the PM and DM data buses. The processor drives the other short word lanes of the PM and DM data buses with zeros.

The instruction explicitly accesses registers RX and RA, and implicitly accesses the complementary registers, SX and SA. This instruction uses PEX registers with the RX and RA mnemonics.

The second word from any other block is shown as x2 on the data bus and in the Sx register. It is shown as Y2 and Y0 respectively in the left side of the block. The Sx and SA registers are transparent and look similar to RX and RA. All bits should be shown as in RX and RA. For more information on arranging data in memory to take advantage of short word addressing of dual-data in SIMD mode, see Figure 7-29 on page 7-61.
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SIMD, SHORT WORD, DUAL-DATA TRANSFERS ARE:

$\text{DREG} = \text{PM}(\text{SHORT WORD ADDRESS})$, $\text{DREG} = \text{DM}(\text{SHORT WORD ADDRESS})$.

Figure 7-13. Short Word Addressing of Dual-Data in SIMD Mode
32-Bit Normal Word Addressing of Single-Data in SISD Mode

Figure 7-14 shows the SISD, single-data, 32-bit normal word addressed access mode. For normal word addressing, the processor treats the data buses as two 32-bit normal word lanes. The 32-bit value for the normal word access completes a transfer using the least significant normal word lane of the PM or DM data bus. The processor drives the other normal word lanes of the data buses with zeros.

In SISD mode, the instruction accesses a PEx register. This instruction accesses WORD X0 whose normal word address has “0” for its least significant address bit. The other access within this four column location has an address with a least significant bit of “1” and selects WORD X1 from memory. The syntax targets register RX in PEx.

For normal word accesses, the processor zero-fills the least significant 8 bits of the data register on loads and truncates these bits on stores to memory.
Figure 7-14. Normal Word Addressing of Single-Data in SISD Mode
32-Bit Normal Word Addressing of Dual-Data in SISD Mode

Figure 7-15 shows the SISD dual-data, 32-bit normal word addressed access mode. For normal word addressing, the processor treats the data buses as two 32-bit normal word lanes. The 32-bit values for normal word accesses transfer using the least significant normal word lanes of the PM and DM data buses. The processor drives the other normal word lanes of the data buses with zeros.

In Figure 7-15, the access targets the PEx registers in a SISD mode operation. This instruction accesses WORD X0 in any other block and WORD Y0 in any block. Each of these words has a normal word address with 0 for its least significant address bit. Other accesses within these four column locations have addresses with the least significant bit of 1 and select WORD X1/Y1 from memory. The syntax targets registers RX and RA in PEx.
Figure 7-15. Normal Word Addressing of Dual-Data in SISD Mode
32-Bit Normal Word Addressing of Single-Data in SIMD Mode

Figure 7-16 shows the SIMD, single-data, normal word addressed access mode. For normal word addressing, the processor treats the data buses as two 32-bit normal word lanes. The explicitly addressed (named in the instruction) 32-bit value completes a transfer using the least significant normal word lane of the PM or DM data bus. The implicitly addressed (not named in the instruction, but inferred from the address in SIMD mode) normal word value completes a transfer using the most significant normal word lane of the PM or DM data bus.

In Figure 7-16, the explicit access targets the named register RX, and the implicit access targets that register’s complementary register, SX. This instruction uses a PeX register with an RX mnemonic. If the syntax named the PeY register SX as the explicit target, the processor would use that register’s complement, RX, as the implicit target. For more information on complementary registers, see “SIMD Mode” on page 3-36.

Figure 7-16 shows the data path for one transfer. The processor accesses normal words sequentially in memory. For more information on arranging data in memory to take advantage of this access pattern, see Figure 7-29 on page 7-61.
Figure 7-16. Normal Word Addressing of Single-Data in SIMD Mode
Memory

32-Bit Normal Word Addressing of Dual-Data in SIMD Mode

Figure 7-17 shows the SIMD, dual-data, 32-bit normal word addressed access mode. For normal word addressing, the processor treats the data buses as two 32-bit normal word lanes. The explicitly addressed (named in the instruction) 32-bit values are transferred using the least significant normal word lane of the PM or DM data bus. The implicitly addressed (not named in the instruction, but inferred from the address in SIMD mode) normal word values are transferred using the most significant normal word lanes of the PM and DM data bus.

In Figure 7-17, the explicit access targets the named registers $RX$ and $RA$, and the implicit access targets those register’s complementary registers $SX$ and $SA$. This instruction uses the $PEx$ registers with the $RX$ and $RA$ mnemonics.

Figure 7-15 shows the data path for one transfer. The processor accesses normal words sequentially in memory. For more information on arranging data in memory to take advantage of this access pattern, see Figure 7-29 on page 7-61.
Figure 7-17. Normal Word Addressing of Dual-Data in SIMD Mode
Extended-Precision Normal Word Addressing of Single-Data

Figure 7-18 on page 7-46 displays a possible single-data, 40-bit extended-precision normal word addressed access. For extended-precision normal word addressing, the processor treats each data bus as a 40-bit extended-precision normal word lane. The 40-bit value for the extended-precision normal word access is transferred using the most significant 40 bits of the PM or DM data bus. The processor drives the lower 24 bits of the data buses with zeros.

In Figure 7-18, the access targets a PEx register in a SISD or SIMD mode operation; extended-precision normal word single-data access operate the same in SISD or SIMD mode. This instruction accesses WORD X0 with syntax that targets register RX in PEx. The example targets a PEy register when using the syntax SX.

Extended precision can’t be supported in SIMD mode since the both PM and DM data busses are limited to 64-bits but would require 80-bits.
Figure 7-18. Extended-Precision Normal Word Addressing of Single-Data
Memory

Extended-Precision Normal Word Addressing of Dual-Data

Figure 7-19 shows the SISD, dual-data, 40-bit extended-precision normal word addressed access mode. For extended-precision normal word addressing, the processor treats each data bus as a 40-bit extended-precision normal word lane. The 40-bit values for the extended-precision normal word accesses are transferred using the most significant 40 bits of the PM and DM data bus. The processor drives the lower 24 bits of the data buses with zeros.

In Figure 7-19, the access targets the PE\textsubscript{X} registers in a SISD mode operation. This instruction accesses WORD X\textsubscript{0} in block 1 and WORD Y\textsubscript{0} in block 0 with syntax that targets registers RX and RY in PE\textsubscript{X}. The example targets a PE\textsubscript{Y} register when using the syntax SX or SY.
Figure 7-19. Extended-Precision Normal Word Addressing of Dual-Data in SISD Mode
Long Word Addressing of Single-Data

Figure 7-20 displays one possible single-data, long word addressed access. For long word addressing, the processor treats each data bus as a 64-bit long word lane. The 64-bit value for the long word access completes a transfer using the full width of the PM or DM data bus.

In Figure 7-20, the access targets a $P_E^x$ register in a SISD or SIMD mode operation. Long word single-data access operate the same in SISD or SIMD mode. This instruction accesses `WORD X0` with syntax that explicitly targets register $RX$ and implicitly targets its neighbor register, $RY$, in $P_E^x$. The processor zero-fills the least significant 8 bits of both the registers. The example targets $P_E^y$ registers when using the syntax $SX$. For more information on how neighbor registers work, see “Complementary Data Registers” on page 2-5.
Figure 7-20. Long Word Addressing of Single-Data
Long Word Addressing of Dual-Data

Figure 7-21 shows the SISD, dual-data, long word addressed access mode. For long word addressing, the processor treats each data bus as a 64-bit long word lane. The 64-bit values for the long word accesses completes a transfer using the full width of the PM or DM data bus.

In Figure 7-21, the access targets $P_{Ex}$ registers in SISD mode operation. This instruction accesses $\text{WORD X0}$ and $\text{WORD Y0}$ with syntax that explicitly targets registers $RX$ and $RA$ and implicitly targets their neighbor registers $RY$ and $RB$ in $P_{Ex}$. The processor zero-fills the least significant 8 bits of all the registers. For more information on how neighbor registers work, see Table 6-1 on page 6-8.

Programs must be careful not to explicitly target neighbor registers in this instruction. While the syntax lets programs target these registers, one of the explicit accesses targets the implicit target of the other access. The processor resolves this conflict by performing only the access with higher priority. For more information on the priority order of data register file accesses, see “Register Files” in Chapter 2, Register Files.

$\text{SIMD mode operation is only supported in NW and SW space.}$
Figure 7-21. Long Word Addressing of Dual-Data in SISD Mode
Broadcast Load Access

Figure 7-22 through Figure 7-29 provide examples of broadcast load accesses for single and dual-data transfers. These read examples show that the broadcast load’s to register access from memory is a hybrid of the corresponding non-broadcast SISD and SIMD mode accesses. The exceptions to this relation are broadcast load dual-data, extended-precision normal word and long word accesses. These broadcast accesses differ from their corresponding non-broadcast mode accesses.
Figure 7-22. Short Word Addressing of Single-Data in Broadcast Load
Figure 7-23. Short Word Addressing of Dual-Data in Broadcast Load
Figure 7-24. Normal Word Addressing of Single-Data in Broadcast Load
### Figure 7-25. Normal Word Addressing of Dual-Data in Broadcast Load

**NORMAL WORD ACCESS**

<table>
<thead>
<tr>
<th>Any Block</th>
<th>Memory</th>
<th>Any Other Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>WORD Y5</td>
<td>WORD Y4</td>
<td></td>
</tr>
<tr>
<td>WORD Y3</td>
<td>WORD Y2</td>
<td></td>
</tr>
<tr>
<td>WORD Y1</td>
<td>WORD Y0</td>
<td></td>
</tr>
<tr>
<td>normal word access</td>
<td></td>
<td>normal word access</td>
</tr>
</tbody>
</table>

**PM DATA BUS**

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
</tr>
<tr>
<td>0x0000</td>
</tr>
</tbody>
</table>

**DM DATA BUS**

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
</tr>
<tr>
<td>0x0000</td>
</tr>
</tbody>
</table>

**Word Flow Example:**

```
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(NORMAL WORD X0 ADDRESS), RA = PM(NORMAL WORD Y0 ADDRESS);

OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR BROADCAST, NORMAL WORD, DUAL-DATA TRANSFERS ARE:
DREG = PM(NORMAL WORD ADDRESS), DREG = DM(NORMAL WORD ADDRESS);```

---

**SHARC Processor Programming Reference**

7-57
Figure 7-26. Extended-Precision Normal Word Addressing of Single-Data in Broadcast Load
Figure 7-27. Extended-Precision Normal Word Addressing of Dual-Data in Broadcast Load
Figure 7-28. Long Word Addressing of Single-Data in Broadcast Load
Figure 7-29. Long Word Addressing of Dual-Data in Broadcast Load
Mixed-Word Width Addressing of Long Word with Short Word

The mixed mode requires a dual data access in all cases. Modes like SISD, SIMD and Broadcast in conjunction with the address types LW, NW-40, NW-32 and SW will result in many different mixed word width access types to use in parallel between the two memory blocks.

Figure 7-30 shows an example of a mixed-word width, dual-data, SISD mode access. This example shows how the processor transfers a long word access on the DM bus and transfers a short word access on the PM bus.

In case of conflicting dual access to the data register file, the processor only performs the access with higher priority. For more information on how the processor prioritizes accesses, see “Register Files” in Chapter 2, Register Files.
Figure 7-30. Mixed-Word Width Addressing of Dual-Data in SISD Mode
 Internal Memory Access Listings

Mixed-Word Width Addressing of Long Word with Extended Word

Figure 7-31 shows an example of a mixed-word width, dual-data, SISD mode access. This example shows how the processor transfers a long word access on the DM bus and transfers an extended-precision normal word access on the PM bus.
Figure 7-31. Mixed-Word Width Addressing of Dual-Data in SIMD Mode
Internal Memory Access Listings
8  JTAG TEST EMULATION PORT

The Analog Devices Tools JTAG emulator is a development tool for debugging programs running in real time on target system hardware.

Because the Analog Devices Tools product line of JTAG emulator controls the target system’s processor through the processor’s IEEE 1149.1 JTAG Test Access Port (TAP), non-intrusive in-circuit emulation is assured. Furthermore, boundary scan test can be performed for specific layout/board tests.

Features

The JTAG port has the following features.

- Support Boundary scan (PCB interconnect test)
- Support standard emulation (conditional breakpoints)
- Support enhanced emulation (statistical profiling, background telemetry channel)
- Support for user breakpoint (core access to breakpoint)

Functional Description

The following sections provide descriptions about the JTAG functionality.
TAP Controller

The TAP controller is a synchronous, 16-state, finite-state machine controlled by the TCK and TMS pins. Transitions to the various states in the diagram occur on the rising edge of TCK and are defined by the state of the TMS pin, here denoted by either a logic 1 or logic 0 state. For full details of the operation, see the JTAG standard. Figure 8-1 shows the state diagram for the TAP controller.

Figure 8-1. TAP Controller State Diagram
**JTAG Test Access Port**

A device operating in IEEE 1149.1 BST (boundary scan test) mode uses four required pins TCK, TMS, TDI, TDO and one optional pin TRST. Table 8-1 summarizes the function of each of these pins.

Table 8-1. JTAG Test Access Port (TAP) Pins

<table>
<thead>
<tr>
<th>Pin</th>
<th>I/O</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCK</td>
<td>I</td>
<td>Test Clock: pin used to clock the TAP state machine (Asynchronous with CLKIN)</td>
</tr>
<tr>
<td>TMS</td>
<td>I</td>
<td>Test Mode Select: pin used to control the TAP state machine sequence</td>
</tr>
<tr>
<td>TDI</td>
<td>I</td>
<td>Test Data In: serial shift data input pin</td>
</tr>
<tr>
<td>TDO</td>
<td>O</td>
<td>Test Data Out: serial shift data output pin</td>
</tr>
<tr>
<td>TRST</td>
<td>I</td>
<td>Test Logic Reset: resets the TAP state machine (STD optional)</td>
</tr>
<tr>
<td>EMU</td>
<td>O</td>
<td>Emulation Status pin (no STD, Analog Devices Inc., specific)</td>
</tr>
</tbody>
</table>

An ADI specific pin (EMU) is used in the JTAG emulators from Analog Devices. This pin is not defined in the IEEE-1149.1 specification. Refer to the IEEE 1149.1 JTAG specification for detailed information on the JTAG interface.

Target systems must have a 14-pin connector in order to accept the Analog Devices Tools product line of JTAG emulator in-circuit probe, a 14-pin plug. For more information refer to Engineer-to-Engineer note EE-68.
Figure 8-2. Serial Scan Path
Instruction Registers

Information in this section describes the control (JTAG) registers. The instruction register is used to determine the action to be performed and the data register to be accessed. There are two types of instructions, one for boundary scan mode and the other for emulation mode. This register selects the performed test and/or the access of the test data register. The instruction register is 5 bits long with no parity bit.

Emulation Instruction Registers (Private)

The emulator can access the internal emulation register by shifting in the JTAG instruction code for the particular emulation register.

The new JTAG instruction set, shown in Table 8-2, lists the binary code for each instruction. Bit 0 is nearest \texttt{TDO} and bit 4 is nearest \texttt{TDI}. No data registers are placed into test modes by any of the public instructions. The instructions affect the processor as defined in the 1149.1 specification.

Table 8-2. JTAG Instruction Register

<table>
<thead>
<tr>
<th>Mode</th>
<th>Instruction</th>
<th>Comment</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boundary Scan</td>
<td>BYPASS</td>
<td>Supported</td>
<td>Public</td>
</tr>
<tr>
<td></td>
<td>EXTEST</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SAMPLE</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEST</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IDCODE</td>
<td>Supported in ADSP-2137x and ADSP-214xx processors</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RUNBIST</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td></td>
<td>USRCODE</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>Emulation</td>
<td></td>
<td>ADI Only</td>
<td>Private</td>
</tr>
</tbody>
</table>

SHARC Processor Programming Reference 8-5
No special values need to be written into any register prior to the selection of any instruction. Other registers, reserved for use by Analog Devices, exist. However, this group of registers should not be accessed as they can cause damage to the part.

Operating Modes

The following sections detail the operation of the JTAG port.

Boundary Scan Mode

A boundary scan allows a system designer to test interconnections on a printed circuit board with minimal test-specific hardware. The scan is made possible by the ability to control and monitor each input and output pin on each chip through a set of serially scannable latches. Each input and output is connected to a latch, and the latches are connected as a long shift register so that data can be read from or written to them through a serial test access port (TAP).

The SHARC processors contain a test access port compatible with the industry-standard IEEE 1149.1 (JTAG) specification. Only the IEEE 1149.1 features specific to the processors are described here. For more information, see the IEEE 1149.1 specification and the other documents listed in “References” on page 8-27.

The boundary scan allows a variety of functions to be performed on each input and output signal of the SHARC processors. Each input has a latch that monitors the value of the incoming signal and can also drive data into the chip in place of the incoming value. Similarly, each output has a latch that monitors the outgoing signal and can also drive the output in place of the outgoing value. For bidirectional pins, the combination of input and output functions is available.
Boundary Scan Register Instructions

The boundary-scan register is selected by the \texttt{EXTEST}, \texttt{INTEST}, \texttt{SAMPLE} and \texttt{IDCODE} instructions. These instructions allow the pins of the processor to be controlled and sampled for board-level testing. For the most recent BSDL files, please visit the Analog Devices Web site.

Note that the optional public instructions \texttt{RUNBIST}, and \texttt{USERCODE} are not supported by the SHARC processors.

Also note that the optional public instructions \texttt{IDCODE} is supported in the ADSP-2137x and ADSP-214xx SHARC processors.

Every latch associated with a pin is part of a single serial shift register path. Each latch is a master/slave type latch with the controlling clock provided externally. This clock (\texttt{TCK}) is asynchronous to the core input clock (\texttt{CLKIN}).

To protect the internal logic when the boundary outputs are overdriven or signals are received on the boundary inputs, make sure that nothing else drives data on the processor’s output pins.

Boundary Scan Description Language (BSDL) is a subset of VHDL that is used to describe how JTAG (IEEE 1149.1) is implemented in a particular device. For a device to be JTAG compliant, it must have an associated BSDL file. For the SHARC processors, BSDL files are available on the Analog Devices Inc., web site.

Emulation Space Mode

The processor emulation features halt the processor at a predefined point to examine the state of the processor, execute arbitrary code, restore the original state, and continue execution. If the processor hits a valid breakpoint it triggers an emulator interrupt which puts the processor into emulation space (core halt). In this state, the processor waits until the emulator continues to scan new instructions into the processor over the TAP.
Operating Modes

If the emulator scans an RTI instruction into the processor, it is released back into user space (core run).

DMA can be used as an optional halt for a breakpoint hit.

The emulator uses the TAP to access the internal space of the processor, allowing the developer to:

- Load code
- Set SW/HW breakpoints
- Set user breakpoints
- Observe variables
- Observe memory
- Examine registers
- Perform cycle counting
- Run background telemetry channels
- Additional features

The processor must be halted to send data and commands, but once an operation is completed by the emulator, the system is set running at full speed with no impact on system timing. The emulator does not impact target loading or timing. The emulator’s in-circuit probe connects to a variety of host computers (USB or PCI) with plug-in boards.

Emulation Control

The processor is free running. In order to observe the state of the core, the emulator must first halt instruction execution and enter emulation mode. In this mode, the emulation software sets up a halt condition by selecting the EMUCTL register and enabling bits 1–0 and 5.
The emulator then returns to run-test-idle. At this point, the processor is not halted. In the next scan, the emulator selects the EMUIR register, and shifts in the NOP instruction. At the very beginning of the scan, the TMS signal rises, and at this point, before the scan has ended, the processor halts. When the emulator finishes the scan by returning to run-test-idle, the processor executes a NOP instruction. Not that the EMUCTL register is only accessible via the TAP.

**Emulation Status Register (EMUSTAT)**

The EMUSTAT register, described in Table 8-3, is 8-bits wide and is accessed by the emulator through the TAP. This register is updated by the SHARC processor when the TAP is in the CAPTURE state. The emulator reads EMUSTAT to determine the state of the SHARC processor. None of the bits in this register can be written by the emulator.

Table 8-3. EMUSTAT Register Bit Descriptions

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>EMUSPACE</td>
<td>Indicates that the next instruction is to be fetched from the emulator</td>
</tr>
<tr>
<td>1</td>
<td>EMUREADY</td>
<td>Indicates that core has finished executing the previous emulator instructions</td>
</tr>
<tr>
<td>2</td>
<td>INIDLE</td>
<td>Indicates that core was in IDLE prior to the latest emulator interrupt</td>
</tr>
<tr>
<td>3</td>
<td>PB_HUNG</td>
<td>Core access to buffer hung</td>
</tr>
<tr>
<td>7–4</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>

**Emulation Control Register (EMUCTL)**

The 40-bit EMUCTL serial shift register shown in Figure 8-3 and described in Table 8-4, is located in the system unit and controls all processor emulation function. It is accessed by the emulator through the TAP.
Operating Modes

Figure 8-3. EMUCTL Register
### Table 8-4. EMUCTL Bit Descriptions

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>EMUENA</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Emulator Function Enable. Enables processor emulation functions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = Emulator interface disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Emulator interface enabled</td>
</tr>
<tr>
<td>1</td>
<td>EIRQENA</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Emulator Interrupt Enable. Enables the emulation logic to recognize external breakpoints (interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>from HW emulator) to move part into emulation space</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = ignore external breakpoints</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = enable external breakpoints</td>
</tr>
<tr>
<td>2</td>
<td>BKSTOP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Halt on Internal Breakpoint. Enables the processor to generate an external emulator interrupt when</td>
</tr>
<tr>
<td></td>
<td></td>
<td>any breakpoint event occurs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = ignore internal breakpoints</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = respond to internal breakpoints</td>
</tr>
<tr>
<td>3</td>
<td>SS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = disable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = enable</td>
</tr>
<tr>
<td>4</td>
<td>SYSRST</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Software Reset. Resets the processor in the same manner as the software reset bit in the SYSCTL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>register. The SYSRST bit must be cleared by the emulator.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = normal operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = reset</td>
</tr>
<tr>
<td>5</td>
<td>ENBRK-OUT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Enable the Emulation Status Pin. Enables the EMU pin operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Whenever core enters emulation space it is notified by assertion of the EMU pin to the emulator.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = EMU pin at high impedance state</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = EMU pin enabled</td>
</tr>
<tr>
<td>6</td>
<td>IOSTOP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Stop IOP DMAs in EMU Space. Disables all DMA requests when the processors are in emulation space.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data that is currently in the EP, LINK, or SPORT DMA buffers is held there unless the internal DMA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>request was already granted. IOSTOP causes incoming data to be held off and outgoing data to cease.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Because SPORT receive data cannot be held off, it is lost and the overrun bit is set. The direct</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write buffer (internal memory write) and the EP pad buffer are allowed to flush any remaining data to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>internal memory.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = I/O continues</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = I/O stops</td>
</tr>
</tbody>
</table>

---

**Note:** The text above provides a detailed explanation of the EMUCTL bit descriptions for the SHARC Processor. Each bit and its corresponding function is described in detail, including how to enable or disable specific features of the processor through these bits. The table format helps to organize the information clearly, allowing for easy reference.
### Table 8-4. EMUCTL Bit Descriptions (Cont’d)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>NEGPA1</td>
<td>Negate program memory data address breakpoint. Enable breakpoint events if the address is greater than the end register value OR less than the start register value. This function is useful to detect index range violations in user code. (0 = disable breakpoint, 1 = enable breakpoint)</td>
</tr>
<tr>
<td>9</td>
<td>NEGDA1</td>
<td>Negate data memory address breakpoint #1 See NEGPA1 bit description.</td>
</tr>
<tr>
<td>10</td>
<td>NEGDA2</td>
<td>Negate data memory address breakpoint #2. See NEGPA1 bit description.</td>
</tr>
<tr>
<td>11</td>
<td>NEGIA1</td>
<td>Negate instruction address breakpoint #1. See NEGPA1 bit description.</td>
</tr>
<tr>
<td>12</td>
<td>NEGIA2</td>
<td>Negate instruction address breakpoint #2. See NEGPA1 bit description.</td>
</tr>
<tr>
<td>13</td>
<td>NEGIA3</td>
<td>Negate instruction address breakpoint #3. See NEGPA1 bit description.</td>
</tr>
<tr>
<td>14</td>
<td>NEGIA4</td>
<td>Negate instruction address breakpoint #4. See NEGPA1 bit description.</td>
</tr>
<tr>
<td>15</td>
<td>NEGIO1</td>
<td>Negate I/O address breakpoint. See NEGPA1 bit description.</td>
</tr>
<tr>
<td>16</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>ENBPA</td>
<td>Enable program memory data address breakpoints. Enable each breakpoint group. Note that when the ANDBKP bit is set, breakpoint types not involved in the generation of the effective breakpoint must be disabled. 0 = disable breakpoints 1 = enable breakpoints</td>
</tr>
<tr>
<td>18</td>
<td>ENBDA</td>
<td>Enable data memory address breakpoints. See ENBPA bit description.</td>
</tr>
<tr>
<td>19</td>
<td>ENBIA</td>
<td>Enable instruction address breakpoints. See ENBPA bit description.</td>
</tr>
<tr>
<td>20–21</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>PA1MODE</td>
<td>PA1 breakpoint triggering mode. Trigger on the following conditions: 00 = Breakpoint is disabled 01 = WRITE accesses only 10 = READ accesses only 11 = any access</td>
</tr>
</tbody>
</table>
Address Breakpoints

The SHARC processors contain sets of emulation breakpoint registers. Each set consists of a start and an end register which describe an address.
Operating Modes

range, with the start register setting the lower end of the address range. Each breakpoint set monitors a particular address bus. When a valid address is in the address range, then a breakpoint signal is generated. The address range includes start and end addresses.

Some of the breakpoints monitor the instruction address, some monitor the data memory address, the program memory data address, and the I/O address bus.

Address Breakpoint Registers

The address breakpoint registers shown in Table 8-5 are used by the emulator and the user breakpoint control to specify address ranges to verify if specific conditions become true. The reset values are not defined.

Table 8-5. Core Domain IOP Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSA1S</td>
<td>Instruction Address Start # 1</td>
<td>24 bits</td>
</tr>
<tr>
<td>PSA1E</td>
<td>Instruction Address Start # 1</td>
<td>24 bits</td>
</tr>
<tr>
<td>PSA2S</td>
<td>Instruction Address Start # 2</td>
<td>24 bits</td>
</tr>
<tr>
<td>PSA2E</td>
<td>Instruction Address End # 2</td>
<td>24 bits</td>
</tr>
<tr>
<td>PSA3S</td>
<td>Instruction Address Start # 3</td>
<td>24 bits</td>
</tr>
<tr>
<td>PSA3E</td>
<td>Instruction Address End # 3</td>
<td>24 bits</td>
</tr>
<tr>
<td>PSA4S</td>
<td>Instruction Address Start # 4</td>
<td>24 bits</td>
</tr>
<tr>
<td>PSA4E</td>
<td>Instruction Address End # 4</td>
<td>24 bits</td>
</tr>
<tr>
<td>IOAS</td>
<td>I/O Address Start</td>
<td>32 bits</td>
</tr>
<tr>
<td>IOAE</td>
<td>I/O Address End</td>
<td>32 bits</td>
</tr>
<tr>
<td>DMA1S</td>
<td>Data Address Start # 1</td>
<td>32 bits</td>
</tr>
<tr>
<td>DMA1E</td>
<td>Data Address End # 1</td>
<td>32 bits</td>
</tr>
<tr>
<td>DMA2S</td>
<td>Data Address Start # 2</td>
<td>32 bits</td>
</tr>
<tr>
<td>DMA2E</td>
<td>Data Address End # 2</td>
<td>32 bits</td>
</tr>
</tbody>
</table>
Conditional Breakpoints

The breakpoint sets are grouped into four types:

- 4x instruction (IA)
- 2x DM data (DA)
- 1x PM data (PA)
- 1x I/O data (I/O)

The individual breakpoint signals in each group are logically ORed together to create a composite breakpoint signal per group.

Each breakpoint group has an enable bit in the EMUCTL/BRKCTL register. When set, these bits add the specified breakpoint group into the generation of the effective breakpoint signal. If cleared, the specified breakpoint group is not used in the generation of the effective breakpoint signal. This allows the user to trigger the effective breakpoint from a subset of the breakpoint groups.

These composite signals can be optionally ANDed or ORed together to create the effective breakpoint event signal used to generate an emulator interrupt. The ANDBKP bit in the BRKCTL register selects the function used.

The ANDBKP bit has no impact within the same group of breakpoints (DA group, IA group). It has significance when the program uses different groups of breakpoints (IA, DM, PM, IO) and the resultant breakpoint is logically ANDed of all those breakpoints which are enabled.

Table 8-5. Core Domain IOP Registers (Cont’d)

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMDAS</td>
<td>Program Data Address Start</td>
<td>32 bits</td>
</tr>
<tr>
<td>PMDAE</td>
<td>Program Data Address End</td>
<td>32 bits</td>
</tr>
</tbody>
</table>
Operating Modes

To provide further flexibility, each individual breakpoint can be programmed to trigger if the address is in range AND one of these conditions is met: READ access, WRITE access, or ANY access. The control bits for this feature are also located in the BRKCTL register.

Note the following restrictions on breakpoints.

1. At least two breakpoints must be enabled prior to enabling the ANDBKP bit.

2. Enabling of breakpoints and ANDBKP bit should not be done in the same instruction.

For index range violations in user code, the address ranges of the emulation breakpoint registers are negated (twos complement) by setting the appropriate negation bits in the BRKCTL register.

Each breakpoint can be disabled by setting the start address larger than the end address.

The instruction address breakpoints monitor the address of the instruction being executed, not the address of the instruction being fetched.

If the current execution is aborted, the breakpoint signal does not occur even if the address is in range. Data address breakpoints (DA and PA only) are also ignored during aborted instructions.

The breakpoint sets can be found in “Programming Model User Breakpoints” on page 8-19.

Event Count Register

The EMUN register is a 32-bit memory-mapped I/O register and can be accessed in user space. Core can write to it in user space. This register is used to detect the Nth breakpoint. This EMUN register allows the breakpoint to occur at Nth count. If the register is loaded with N, the processor...
is interrupted only after the detection of N breakpoint conditions. At every breakpoint occurrence the processor decrements the EMUN register and it generates an interrupt when content of EMUN is zero and a breakpoint event occurs.

Note that programs must load this register with a value greater or equal to zero for proper breakpoint generation under the condition that bit 25 (UMODE bit) in the BRKCTL register is set.

**Emulation Cycle Counting**

The emulation clock counter consists of a 32-bit count register, EMUCLK and a 32 bit scaling register, EMUCLK2. The EMUCLK register counts clock cycles while the user has control of the chip and stops counting when the emulator gains control. This allows a user to gauge the amount of time spent executing a particular section of code. The EMUCLK2 register is used to extend the time EMUCLK can count by incrementing itself each time the EMUCLK value rolls over to Zero. Both EMUCLK and EMUCLK2 are emulation registers, which can only be written in emulation space. Reads of EMUCLK and EMUCLK2 can be performed in user space. This allows simple benchmarking of code.

**Enhanced Emulation Mode**

This section describes the enhanced emulation features, which are used for the Background Telemetry Channel (BTC) and statistical profiling. In enhanced emulation space, there is a continuous data stream to the target system over the TAP. Notice that single step mode is not allowed using the enhanced emulation features.

**Statistical Profiling**

Statistical profiling allows the emulation software to sample the processors PC value while the processor is running. By sampling at random intervals, a profile can be created which can aid the developer in tuning
Operating Modes

performance critical code sections. As a second use, statistical profiling can also aid in finding dead code as well as being used to make code partition decisions. Fundamentally, statistical profiling is supported by one additional JTAG shift register called EMUPC and a register which latches the sampled PC. The EMPUC register is a 24-bit serial shift register which samples the program counter whenever the JTAG TAP controller is in RUNTEST state. So, whenever TAP controller is in RUNTEST state the EMUPC is overridden every CCLK (core clock) cycle. The EMUPC register is not a memory-mapped register and is accessed over the TAP. This instruction is used for statistical profiling.

User Space Mode

The following sections describe user space mode operation.

User Breakpoint Control

By default, the emulator has control over the breakpoint unit. However, if there is a need for faster system debug without the delay incurred when the core halts and enters emulations space, then the core can gain control by setting the UMODE bit in the BRKCTL register.

Conversely, if the UMODE (bit 25) is cleared, only the emulator has breakpoint control over the TAP.

If the UMODE bit in the BRKCTL register is set, all address breakpoint registers can be written in user space.

For more information, see “Breakpoint Control Register (BRKCTL)” on page A-43.

User Breakpoint Status

The EEMUSTAT register acts as the breakpoint status register for the SHARC processors. This register is a memory-mapped IOP register. The processor core can access this register if the UMODE bit (bit 25) is set.
The enhanced emulation status register, EEMUSTAT, indicates which breakpoint hit occurred, all the breakpoint status bits are cleared when the program exits the ISR with an RTI instruction. Such interrupts may contain error handling if the processor accesses any of the addresses in the address range defined in the breakpoint registers.

*Status update of the EEMUSTAT register does not work in single step mode for user break points.*

For more information, see “Enhanced Emulation Status Register (EEMUSTAT)” on page A-47.

**User Breakpoint System Exception Handling**

Through the proper configuration of the BRKCTL and EEMUSTAT registers, and by using different logical combined address breakpoint regions in conjunction with event count registers for core or DMA operations, programs can take advantage of system specific exception handling based on specified conditions which trigger the low priority emulator interrupt (BKPI).

**Programming Model User Breakpoints**

To set up the user controlled breakpoint functionality use the following steps.

1. Unmask the BKPI interrupt (low priority interrupt).
2. Set the UMODE bit in the BRKCTL register.
3. Set the breakpoint count in EMUN register to the required value.
4. Initialize the breakpoint address registers with required address ranges.
Operating Modes

5. Enable the breakpoint conditions as required in the BRKCTL register.

6. Enable the logical ANDing of breakpoints if required in the BRKCTL register.

Programming Examples

Listing 8-1 is an example that shows how to trigger an exception for a valid address.

Listing 8-1. Trigger an Exception for a Valid Address

```
bit set IMASK BKPI;   /* unmask BKPI */
bit set MODE1 IRPTEN; /* enable global int */
r5 = ADDR_S;          /* valid start addr for the break */
r6 = ADDR_E;          /* valid end addr for the break */
r3 = UMODE | DA1MODE; /* set the user mode and dm access functionality for r/w access */
dm(BRKCTL) = r3;
dm(DMAIS) = r5;       /* start addr for break */
dm(DMA1E) = r6;       /* end addr for break */
r5 = 0x15;
dm(EMUN) = r5;        /* set event count */
USTAT1 = dm(BRKCTL);
BIT SET USTAT1 ENBDA; /* enable the dm access break points */
dm(BRKCTL) = USTAT1;
ISR_BKPI:
  r4 = dm(EEMUSTAT);  /* read status bits */
  rti;               /* status register cleared */
```

Listing 8-2 is an example that shows how to trigger an exception for an invalid address range.
Listing 8-2. Trigger an Exception for an Invalid Address Range

bit set IMASK BKPI;     /* unmask BKPI */
bit set MODE1 IRPTEN;   /* enable global int */
r4 = ADDR_S;            /* valid start address for the break */
r5 = ADDR_E;            /* valid end address for the break */

USTAT1 = UMODE | DA2MODE | NEGDA2; /* set the user mode and negate dm access functionality for r/w access */
dm(BRKCTL) = USTAT1;

dm(DMA2S) = r4;
dm(DMA2E) = r5;

r5 = 0x0;                /* no event count */
dm(EMUN) = r5;

USTAT1 = dm(BRKCTL);
BIT SET USTAT1 ENBDA;   /* enable the dm access break points */
dm(BRKCTL) = USTAT1;

ISR_BKPI:
r4 = dm(EEMUSTAT);      /* read status bits */
rtn;                   /* status register cleared */

Emulation vs. User Space Mode

Note that the major difference between user space and emulation space operation is that in emulation space, the processor holds while the instruction is scanned in, and that the instruction is taken from an emulation instruction register, rather than from the PMD bus. The PC also stops incrementing. All other aspects of instruction execution are the same in both modes.
Operating Modes

Note that the control for breakpoints is also available in emulation space. The emulation control register has equivalent control bits similar to the BRKCTL register to control breakpoints. The control of breakpoints can be flipped back and forth between emulation space or and the core by flipping the (UMODE) bit 25 in the BRKCTL register.

Note that the EMUCTL and BRKCTL register bit settings are almost identical. The EMUCTL register is accessed by the debugger over the TAP while the BRKCTL register access is user code specific.

Single Step

When the single step bit in the emulation control register is set, single step mode is enabled. In single step mode, the processor executes a single instruction, and then automatically generates an internal emulator interrupt to return to emulation space. While in emulation space the emulator can execute a RTI instruction to do a single step again. Each user instruction execution in single step mode clears the instruction pipeline when the part reenters user space.

Instruction Pipeline Fetch Inputs

The instruction pipeline is fed by four inputs:

1. Instruction fetch from memory, this is the user mode (also known as user space) and described in the sequencer chapter
2. Instruction fetch from boot channel, during boot operation (256 instruction words) the pipeline is fed with the IDLE instruction until the peripheral’s interrupt is generated
3. Instruction fetch from an emulator register, by using tools (debugger) in single step mode (also known as emulation space) the instruction pipeline is deactivated. In this mode, each instruction is
fetched from an emulation register over the JTAG interface (rather from memory) and executed in isolation. The process is repetitive for all the next instructions in single step mode.

4. Instruction fetched from cache during an cache hit. If a hit occurs, the instruction is loaded from cache and not from memory.

**JTAG Interrupts**

Table 8-6 provides an overview of the interrupts associated with the JTAG port.

Table 8-6. JTAG Interrupt Overview

<table>
<thead>
<tr>
<th>Source</th>
<th>Condition</th>
<th>Priorities (0–41)</th>
<th>Interrupt Acknowledge</th>
<th>IVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>JTAG</td>
<td>- TMS pin - EMUIDLE instruction - Hardware breakpoint (emu space, user space) - BTC channel (Input FIFO full, output FIFO empty)</td>
<td>0, 6, 37</td>
<td>RTI instruction</td>
<td>EMUI BKPI EMULI</td>
</tr>
</tbody>
</table>

**Interrupt Types**

Four different types of interrupts/breakpoints are generated.

1. External Emulator generates EMUI interrupt via TMS (highest priority)

2. Breakpoint generates an internal EMUI interrupt (highest priority)

3. User space breakpoint generates an internal BKPI interrupt (lower priority)

4. BTC generates an internal EMULI interrupt (lowest priority)
Breakpoints

Entering Into Emulation Space

When the core receives emulator interrupt, the following sequence occurs:

1. The PC stack is pushed and the PC vectors to location 0
2. The core is idle, waiting for an emulator instruction
3. The core timer and emulation counter stop counting
4. The cache is disabled
5. DMA operation is may be optionally stalled
6. The core notifies emulation space via the EMU pin

Breakpoints

This section explains the different types of breakpoint and conditions to hit breakpoints.

Software Breakpoints

Software breakpoints are implemented by the processor as a special type of instruction. The instruction, EMUIDLE is not a public instruction, and is only decoded by the processor when specific bits are set in emulation control. If the processor encounters the EMUIDLE instruction and the specific bits are not set in emulation control, then the processor executes a NOP instruction. The EMUIDLE instruction triggers a high emulator interrupt. When EMUIDLE is executed, the emulation clock counter halts immediately.
**Automatic Breakpoints**

The IDDE (tools environment) places the labels `_main` and `(__lib_prog_term)` automatically at software breakpoints (EMUIDLE). If you place the `_main` label at the beginning of user code it will simplify start execution code after reset (initialization like DDR2/SDRAM or runtime environment) until the breakpoint `_main` is hit before the programs enters user code.

For more information, refer to the tools documentation.

**Hardware Breakpoints**

Hardware breakpoints allow much greater flexibility than software breakpoints provided by the EMUIDLE instruction. As such, they require much more design thought and resources within the processor. At the simplest level, hardware breakpoints are helpful when debugging ROM code where the emulation software can not replace instructions with the EMUIDLE instruction. As hardware breakpoint units capabilities are increased, so are the benefits to the developer. At a minimum, an effective hardware breakpoint unit will have the capability to trigger a break on load, store, and fetch activities.

Additionally, address ranges, both inclusive (bounded) and exclusive (unbounded) should be included.
Breakpoints

General Restrictions on Software Breakpoints

Based on the 5 stage instruction pipeline, the following restrictions apply when setting software breakpoints.

- If a breakpoint interrupt comes at a point when a program is coming out of an interrupt service routine of a prior breakpoint, then in some cases the breakpoint status does not reflect that the second breakpoint interrupt has occurred.
- If an instruction address breakpoint is placed just after a short loop, a spurious breakpoint is generated.
- Delay slots of delayed branch instructions.
- Within the last instruction of zero overhead loops.
- Counter based loops of length one two and three
- Fourth instruction of a counter based loop of length four
- Last but fourth (e-4) instruction of a loop of length more than four
- Last three instructions of any arithmetic loop

JTAG Register Effect Latency

The I/O processor breakpoint address registers have a one-cycle effect latency (changes take effect on the second cycle after the change). Instruction address and program memory breakpoint negates have an effect latency of 4 core clock cycles.
References

- To order a copy, contact the IEEE society.
- Parker, Kenneth. The Boundary Scan Handbook.
- Hewlett-Packard Co. HP Boundary-Scan Tutorial and BSDL Reference Guide.
9 INSTRUCTION SET TYPES

In the SHARC processor family two different instruction types are supported:

- Instruction Set Architecture (ISA) is the traditional instruction set and is supported by all the SHARC processors.
- Variable Instruction Set Architecture (VISA) is supported by the newer ADSP-214xx processors.

The instruction types linked into normal word space are valid ISA instructions (48-bit). When linked into short word space they become valid VISA instructions (48/32/16-bits).

Many ISA instruction types have conditions and compute/data move options. However as programmer there may be situations where options in an instruction are not required. Moreover, many instructions have spare bits which are unused. For ISA instructions the opcode always consumes 48 bits, which results in wasted memory space. For VISA instruction types, all possible options have been extracted to generate new sub instructions resulting in 32-bit or 16-bit instructions.

This chapter provides information on the instructions associated with the SHARC core. Each instruction group has an overview table of its instruction types. For information on computation instructions (ALU, multiplier, shifter, multifunction) see Chapter 10, Computation Instructions. The opcodes relating to the instruction types are shown in Chapter 11, Instruction Opcodes.
Instruction Groups

The instruction groups are:

- “Group I – Conditional Compute and Move or Modify Instructions” on page 9-4
- “Group II – Conditional Program Flow Control Instructions” on page 9-30
- “Group III – Immediate Data Move Instructions” on page 9-51
- “Group IV – Miscellaneous Instructions” on page 9-64

Instruction Set Notation Summary

The conventions for instruction syntax descriptions appear in Table 9-1. Other parts of the instruction syntax and opcode information also appear in this section.

Table 9-1. 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>option1</td>
</tr>
<tr>
<td></td>
<td>option2</td>
</tr>
<tr>
<td>compute</td>
<td>ALU, multiplier, shifter or multifunction operation (see “Computation Instructions” on page 10-1)</td>
</tr>
</tbody>
</table>
### Instruction Set Types

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

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>shiftimm</td>
<td>Shifter immediate operation (see &quot;Computation Instructions&quot; on page 10-1)</td>
</tr>
<tr>
<td>cond</td>
<td>Status condition (see condition codes in Table 4-37 on page 4-92)</td>
</tr>
<tr>
<td>termination</td>
<td>Loop termination condition (see condition codes in Table 4-37 on page 4-92)</td>
</tr>
<tr>
<td>ureg</td>
<td>Universal register</td>
</tr>
<tr>
<td>cureg</td>
<td>Complementary universal register (see Table 2-3 on page 2-6)</td>
</tr>
<tr>
<td>sreg</td>
<td>System register</td>
</tr>
<tr>
<td>csreg</td>
<td>Complementary system register (see Table 2-3 on page 2-6)</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): S15–S0 or SF15–SF0 (see Table 2-3 on page 2-6)</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>&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>+k</td>
<td>the implicit incremental address depending on SISD, SIMD or Broadcast mode</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>
The list of UREGs (universal registers) can be found in Table 2-1 on page 2-2.

**Group I - Conditional Compute and Move or Modify Instructions**

The group I instructions contain a condition a computation and a data move operation.

The COND field selects whether the operation specified in the COMPUTE field and data move is executed. If the COND is true, the compute and data move are executed. If no condition is specified, COND is true condition, and the compute and data move are executed.

The COMPUTE field 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 Chapter 10, Computation Instructions.

- “Type 1a ISA/VISA (compute + mem dual data move) Type 1b VISA (mem dual data move)” on page 9-7
- “Type 2a ISA/VISA (cond + compute) Type 2b VISA (compute) Type 2c VISA (short compute)” on page 9-10
- “Type 3a ISA/VISA (cond + comp + mem data move) Type 3b VISA (cond + mem data move) Type 3c VISA (mem data move)” on page 9-12
- “Type 4a ISA/VISA (cond + comp + mem data move with 6-bit immediate modifier) Type 4b VISA (cond + mem data move with 6-bit immediate modifier)” on page 9-17
- “Type 5a ISA/VISA (cond + comp + reg data move) Type 5b VISA (cond + reg data move)” on page 9-22
Instruction Set Types

- “Type 6 ISA/VISA (cond + shift imm + mem data move)” on page 9-25
- “Type 7a ISA/VISA (cond + comp + index modify) Type 7b ISA/VISA (cond + index modify)” on page 9-28

The following table provides an overview of the Group I instructions. The letter after the instruction type denotes the instruction size as follows: a = 48-bit, b = 32-bit, c = 16-bit. Note that items in italics are optional.

<table>
<thead>
<tr>
<th>Type</th>
<th>Addr</th>
<th>Option 1</th>
<th>Option 2</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>ISA</td>
<td></td>
<td></td>
<td>compute,</td>
</tr>
<tr>
<td></td>
<td>VISA</td>
<td></td>
<td></td>
<td>DM(Ia,Mb) = DREG, PM(Ic,Md) = DREG;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DREG = DM(Ia,Mb), DREG = PM(Ic,Md);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DREG = DM(Ia,Mb), PM(Ic,Md) = DREG;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DM(Ia,Mb) = DREG, DREG = PM(Ic,Md);</td>
</tr>
<tr>
<td>1b</td>
<td>VISA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2a</td>
<td>ISA</td>
<td></td>
<td></td>
<td>IF condition compute;</td>
</tr>
<tr>
<td></td>
<td>VISA</td>
<td></td>
<td></td>
<td>DM(Ia,Mb) = UREG(LW);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DM(Mb,Ia);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PM(Ic,Md);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PM(Md,Ic);</td>
</tr>
<tr>
<td>2b</td>
<td>VISA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3a</td>
<td>ISA</td>
<td></td>
<td></td>
<td>IF condition compute;</td>
</tr>
<tr>
<td></td>
<td>VISA</td>
<td></td>
<td></td>
<td>DM(Ia,Mb) = UREG(LW);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DM(Mb,Ia);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PM(Ic,Md);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PM(Md,Ic);</td>
</tr>
<tr>
<td>3b</td>
<td>VISA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4a</td>
<td>ISA</td>
<td></td>
<td></td>
<td>IF condition compute;</td>
</tr>
<tr>
<td></td>
<td>VISA</td>
<td></td>
<td></td>
<td>DM(Ia, &lt;data6&gt;) = DREG;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DM(&lt;data6&gt;,Ia);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PM(Ic, &lt;data6&gt;);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PM(&lt;data6&gt;,Ic);</td>
</tr>
<tr>
<td>4b</td>
<td>VISA</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Group I - Conditional Compute and Move or Modify Instructions

<table>
<thead>
<tr>
<th>Type</th>
<th>Addr</th>
<th>Option 1</th>
<th>Option 2</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5a</td>
<td>ISA VISA</td>
<td>IF condition</td>
<td>compute.</td>
<td>UREG = UREG; DREG &lt;-&gt; CREG;</td>
</tr>
<tr>
<td>5b</td>
<td>VISA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6a</td>
<td>ISA VISA</td>
<td>IF condition</td>
<td>shiftimm</td>
<td>DM(Ia,Mb) = DREG; PM(Ic,Md); DREG = DM(Ia,Mb); PM(Ic,Md);</td>
</tr>
<tr>
<td>7a</td>
<td>ISA VISA</td>
<td>IF condition</td>
<td></td>
<td>MODIFY(Ia,Mb); MODIFY(Ic,Md); Ia = MODIFY(Ia,Mb); /* for ADSP-214xx */</td>
</tr>
<tr>
<td>7b</td>
<td>VISA</td>
<td></td>
<td></td>
<td>ic = MODIFY(Ic,Md);</td>
</tr>
</tbody>
</table>
Instruction Set Types

Type 1a ISA/VISA (compute + mem dual data move)
Type 1b VISA (mem dual data move)

Type 1a Syntax

Comput + parallel memory (data and program) transfer.

\[
\begin{align*}
\text{compute,} & \quad DM(ia, Mb) = dreg \\
dreg &= DM(ia, Mb) \\
& \quad PM(ic, Md) = dreg \\
dreg &= PM(ic, Md) \\
\end{align*}
\]

Type 1b Syntax

Parallel data memory and program memory transfers with register file, without the Type 1 compute operation.

\[
\begin{align*}
\text{DM}(ia, Mb) &= dreg \\
dreg &= DM(ia, Mb) \\
\text{PM}(ic, Md) &= dreg \\
dreg &= PM(ic, Md) \\
\end{align*}
\]

SISD Mode

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 Chapter 6, Data Address Generators.

SIMD Mode

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.
Group I - Conditional Compute and Move or Modify Instructions

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.

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 Chapter 6, Data Address Generators.

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 2-3 on page 2-6.

Broadcast Mode

If the broadcast read bits—BCST1 (for I1) or BCST9 (for I9)—are set, the Y element uses the specified I register without adding one.

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

<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>compute</td>
<td>compute</td>
</tr>
<tr>
<td>, DM(Ia, Mb) = dreg</td>
<td>, PM(Ic, Md) = dreg</td>
</tr>
<tr>
<td>, dreg = DM(Ia, Mb)</td>
<td>, dreg = PM(Ic, Md)</td>
</tr>
</tbody>
</table>

If broadcast mode memory read k=0.
If SIMD mode NW access k=1, SW access k=2.

Examples

R7=BSET R6 BY R0, DM(I0,M3)=R5, PM(I11,M15)=R4;
R8=DM(I14,M1), PM(I12 M12)=R0;

When the processors are in SISD mode, 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 processors are in SIMD mode, the first instruction in this example performs the same computation and performs two writes in parallel on both PEx and PEy. The $R7$ register on PEx and $S7$ on PEy both store the results of the Bset computations. Also, simultaneous dual memory writes occur with DM and PM, writing in values from $R5$, $S5$ (DM) and $R4$, $S4$ (PM) respectively. In the second instruction, values are simultaneously read from data memory to registers $R8$ and $S8$ and written to program memory from registers $R0$ and $S0$.

When the processors are in broadcast mode (the $BDCST1$ bit is set in the $MODE1$ system register), the $R0$ (PEx) data register in this example is loaded with the value from data memory utilizing the $I1$ register from DAG1, and $S0$ (PEy) is loaded with the same value.
Group I - Conditional Compute and Move or Modify Instructions

Type 2a ISA/VISA (cond + compute)
Type 2b VISA (compute)
Type 2c VISA (short compute)

Type 2a Syntax

Compute operation, condition

IF COND compute ;

Type 2b Syntax

Compute operation, without the Type 2 condition

compute ;

Type 2c Syntax

Short (16-bit) compute operation, without the Type 2 condition

short compute ;

The short compute is related for some instructions where one operand (Rn/Fn) is the source AND destination and short compute instructions are listed below

Rn = Rn + Rx   Fn = Fn + Fx
Rn = Rn - Rx   Fn = Fn - Fx
Rn = PASS Rx   Fn = FLOAT Rx
COMP (Rn, Rx)  COMP (Fn, Fx)
Rn = NOT Rx    Rn = Rn AND Rx
Rn = Rx + 1    Rn = Rn OR Rx
Rn = Rx - 1    Rn = Rn XOR Rx
Rn = Rn * Rx (SSI)   Fn = Fn * Fx

9-10 SHARC Processor Programming Reference
SISD Mode

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

SIMD Mode

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 ;
```

Examples

```plaintext
IF MV R6=SAT MRF (UI);
```

When the processors are in SISD mode, 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 processors are in SIMD mode, the condition is evaluated on each processing element, PEx and PEy, independently. The computation executes on both PEs, 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.
Group I - Conditional Compute and Move or Modify Instructions

Type 3a ISA/VISA (cond + comp + mem data move)
Type 3b VISA (cond + mem data move)
Type 3c VISA (mem data move)

Type 3a Syntax

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

IF COND compute

, $\text{DM}(Ia, Mb)$ = ureg (LW);
, $\text{PM}(Ic, Md)$

, $\text{DM}(Mb, Ia)$ = ureg (LW);
, $\text{PM}(Md, Ic)$

, ureg = $\text{DM}(Ia, Mb)$ (LW);
$\text{PM}(Ic, Md)$ (LW);

, ureg = $\text{DM}(Mb, Ia)$ (LW);
$\text{PM}(Md, Ic)$ (LW);

Type 3b Syntax

Transfer operation between data or program memory and universal register, optional condition, without the Type 3 optional compute operation

IF COND

$\text{DM}(Ia, Mb)$ = ureg (LW);
$\text{PM}(Ic, Md)$

$\text{DM}(Mb, Ia)$ = ureg (LW);

9-12 SHARC Processor Programming Reference
Instruction Set Types

Type 3c Syntax

Transfer operation between data memory and data register, *without* the Type 3 optional condition, *without* the Type 3 optional compute operation

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

SISD Mode

In SISD mode, the Type 3a and 3b 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 programs 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 Chapter 6, Data Address Generators.
Group I - Conditional Compute and Move or Modify Instructions

SIMD Mode

In SIMD mode, the Type 3a and 3b 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/two (for normal/short word access) to the specified I register (before pre-modify or post-modify) to address data or program memory. If the I value post-modified, the I register is updated with the modified value from the specified M register. The optional \( \text{(LW)} \) in this syntax lets programs specify long word addressing, overriding default addressing from the memory map.

For the universal register, the X element uses the specified \( \text{Ureg} \) register, and the Y element uses the corresponding complementary register \( \text{(Cureg)} \). For a list of complementary registers, see Table 2-3 on page 2-6. Note that the \( \text{Ureg} \) may not be from the same DAG (DAG1 or DAG2) as \( \text{Ia/Mb} \) or \( \text{Ic/Md} \).

The compute operation 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.

Broadcast Mode

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 implicit address addition.

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

---

9-14 SHARC Processor Programming Reference
**Instruction Set Types**

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

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

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

\[
\text{IF PEy COND compute} \quad , \text{DM}(Ia+k, 0) = \text{cureg} (LW); \\
, \text{PM}(Ic+k, 0) \\
, \text{DM}(Mb+k, Ia) = \text{cureg} (LW); \\
, \text{PM}(Md+k, Ic) \\
, \text{cureg} = \text{DM}(Ia+k, 0) (LW); \\
, \text{PM}(Ic+k, 0) (LW); \\
, \text{cureg} = \text{DM}(Mb+k, Ia) (LW); \\
, \text{PM}(Md+k, Ic) (LW); \\
\]

If broadcast mode memory read k=0.
If SIMD mode NW access k=1, SW access k=2.

**Examples**

\[
R6=R3-R11, \text{DM}(I0,M1)=\text{ASTAT}x; \\
\text{IF NOT SV F8=CLIP F2 BY F14, F7=PM(I12,M12)];}
\]

When the processors are in SISD mode, 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 processors are in SIMD mode, 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 \( ASTATx \) and \( ASTATy \). The condition is evaluated on each processing element, PEx and PEy, independently. The computation executes on both PEs, 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 \( SF7 \).

IF NOT SV F8=CLIP F2 BY F14, F7=PM(I9,M12);

When the processors are in broadcast mode (the BDCST9 bit is set in the 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 DAG2 is stored in \( F7 \). The \( SF7 \) register is loaded with the same value from program memory as \( F7 \).
Instruction Set Types

**Type 4a ISA/VISA (cond + comp + mem data move with 6-bit immediate modifier)**

**Type 4b VISA (cond + mem data move with 6-bit immediate modifier)**

**Type 4a Syntax**

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

\[
\text{IF COND compute} \quad , \quad \text{DM}(\text{Ia}, <\text{data6}>), \quad \text{PM}(\text{Ic}, <\text{data6}>) = \text{dreg} ;
\]

\[
, \quad \text{DM}(<\text{data6}>, \text{Ia}), \quad \text{PM}(<\text{data6}>, \text{Ic}) = \text{dreg} ;
\]

\[
, \quad \text{dreg} = \quad \text{DM}(\text{Ia}, <\text{data6}>) ; \quad \text{PM}(\text{Ic}, <\text{data6}>) ;
\]

\[
, \quad \text{dreg} = \quad \text{DM}(<\text{data6}>, \text{Ia}) ; \quad \text{PM}(<\text{data6}>, \text{Ic}) ;
\]
Group I - Conditional Compute and Move or Modify Instructions

Type 4b Syntax

Index-relative transfer between data or program memory and register file, optional condition, without the Type 4 optional compute operation

\[
\begin{align*}
\text{IF COND} & \quad \text{DM}(I_a, \text{<data6>}) = \text{dreg (LW)} ; \\
& \quad \text{PM}(I_c, \text{<data6>}) \\
& \quad \text{DM}(\text{<data6>}, I_a) = \text{dreg (LW)} ; \\
& \quad \text{PM}(\text{<data6>}, I_c) \\
\text{dreg} & = \quad \text{DM}(I_a, \text{<data6>}) \text{ (LW)} ; \\
& \quad \text{PM}(I_c, \text{<data6>}) \text{ (LW)} \\
\text{dreg} & = \quad \text{DM}(\text{<data6>}, I_a) \text{ (LW)} ; \\
& \quad \text{PM}(\text{<data6>}, I_c) \text{ (LW)} ;
\end{align*}
\]

SISD Mode

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 Chapter 6, Data Address Generators.

SIMD Mode

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/two (for normal/short word access) to the specified I register (before pre-modify or post-modify) to address data or program memory. 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 programs 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 2-3 on page 2-6.

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.

Broadcast Mode

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.

The following pseudo code compares the Type 4 instruction’s explicit and implicit operations in SIMD mode.
Group I - Conditional Compute and Move or Modify Instructions

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

\[
\text{IF PEx COND compute }, \text{ DM}(Ia, \text{<data6>}), \text{ DM}(<data6>, Ia), \text{ PM}(Ic, \text{<data6>}), \text{ PM}(<data6>, Ic) = \text{dreg} ;
\]

\[
\text{dreg} = \text{DM}(Ia, \text{<data6>}) ; \text{ PM}(Ic, \text{<data6>}) ;
\]

\[
\text{dreg} = \text{DM}(<data6>, Ia) ; \text{ PM}(<data6>, Ic) ;
\]

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

\[
\text{IF PEy COND compute }, \text{ DM}(Ia+k, 0), \text{ DM}(<data6>+k, Ia), \text{ PM}(Ic+k, 0), \text{ PM}(<data6>+k, Ic) = \text{cdreg} ;
\]

\[
\text{cdreg} = \text{DM}(Ia+k, 0) ; \text{ PM}(Ic+k, 0) ;
\]

\[
\text{cdreg} = \text{DM}(<data6>+k, Ia) ; \text{ PM}(<data6>+k, Ic) ;
\]

If broadcast mode memory read \(k=0\).
If SIMD mode NW access \(k=1\), SW access \(k=2\).

Examples

\[
\text{IF FLAG0\_IN F1=F5*F12, F11=PM(I10,6);}
\]
\[
R12=R3 \text{ AND R1, DM(6,11)=R6;}
\]

When the processors are in SISD mode, 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 processors are in SIMD mode, the condition is evaluated on each processing element, PEx and PEy, independently. The computation and program memory read execute on both PEs, 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.

\[
\text{If } \text{FLAG0}_\text{IN} \ F1=F5+F12, \ F11=\text{PM}(I9,3);
\]

When the processors are in broadcast mode (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.
Group I - Conditional Compute and Move or Modify Instructions

Type 5a ISA/VISA (cond + comp + reg data move)
Type 5b VISA (cond + reg data move)

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

Type 5a Syntax

```
IF COND compute,     ureg1 = ureg2      ;
               dreg <-> cdreg
```

Type 5b Syntax

Transfer between two universal registers or swap between a data register in each processing element, optional condition, without the Type 5 optional compute operation

```
IF COND     ureg1 = ureg2      ;
               dreg <-> cdreg
```

SISD Mode

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.

SIMD Mode

In SIMD mode, the Type 5 instruction provides the same transfer (=) from one register to another as is available in SISD mode, but provides
Instruction Set Types

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 ;
  dreg <-> cdreg

SIMD Implicit Operation (PEy Operation Implied by the Instruction Syntax)
IF PEy COND compute,
  cureg1 = cureg2 ;
  /* no implicit operation */
```

Examples

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

When the processors are in SISD mode, 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 \( R_0 \) and \( S_1 \).

When the processors are in SIMD mode, the condition is evaluated on each processing element, PEx and PEy, independently. The computation executes on both PEs, 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 6 ISA/VISA (cond + shift imm + mem data move)

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

**Syntax**

SISD Mode

<table>
<thead>
<tr>
<th>IF COND shiftimm</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DM(Ia, Mb)</td>
<td>= dreg ;</td>
</tr>
<tr>
<td></td>
<td>PM(Ic, Md)</td>
<td></td>
</tr>
<tr>
<td>, dreg =</td>
<td>DM(Ia, Mb) ;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PM(Ic, Md) ;</td>
<td></td>
</tr>
</tbody>
</table>

SIMD Mode

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 10-56. For more information on register restrictions, see Chapter 6, Data Address Generators.

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.

SIMD Mode

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.
Group I - Conditional Compute and Move or Modify Instructions

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/two (for normal/short word access) to the specified I register to address data or program memory.

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.

Broadcast Mode

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.

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

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

\[
\text{IF PEx COND shiftimm, DM(Ia, Mb) = dreg ; PM(Ic, Md)}
\]

\[
, \text{dreg} = \begin{cases} 
\text{DM(Ia, Mb) ; PM(Ic, Md) ;}
\end{cases}
\]

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

\[
\text{IF PEy COND shiftimm, DM(Ia+k, 0) = cdreg ; PM(Ic+k, 0)}
\]

\[
, \text{cdreg} = \begin{cases} 
\text{DM(Ia+k, 0) ; PM(Ic+k, 0) ;}
\end{cases}
\]

If broadcast mode memory read k=0.
If SIMD mode NW access k=1, SW access k=2.
Instruction Set Types

Examples

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

When the processors are in SISD mode, 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 processors are in SIMD mode, the condition is evaluated on each processing element, PEx and PEy, independently. The computation and data memory write executes on both PEs, 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 processors are in broadcast mode (the BDCST1 bit is 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 11 register from DAG1. The SF3 register is also loaded with the same value from data memory as F3.
Group I - Conditional Compute and Move or Modify Instructions

Type 7a ISA/VISA (cond + comp + index modify)
Type 7b ISA/VISA (cond + index modify)

Index register modify, optional condition, optional compute operation. See also “Type 19a ISA/VISA (index modify/bitrev)” on page 9-69.

Type 7a Syntax

1 Applies to ADSP-214xx models only.

Type 7b Syntax

Index register modify, optional condition, *without* the Type 7 optional compute operation

1 Applies to ADSP-214xx models only.

SISD Mode

In SISD mode, the Type 7 instruction provides an update of the specified Ia/Ic register by the specified Mb/Md register. If the destination register is not specified, Ia/Ic is used as destination register. Unless destination I register is specified or implied to be the same as the source I register, the source I register is left unchanged. M register is always left unchanged. 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 Chapter 6, Data Address Generators.
Instruction Set Types

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 wraparound, independent of the state of the CBUFEN bit.

SIMD Mode

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 index register modify operation, in SIMD mode, occurs based on the logical ORing of the outcome of the conditions tested on both PEs. In the second instruction, the index register modify also occurs based on the logical ORing of the outcomes of the conditions tested on both PEs. Because both threads of a SIMD sequence may be dependent on a single DAG index value, either thread needs to be able to cause a modify of the index.

Examples

IF NOT FLAG2_IN R4=R6*R12(SUF), MODIFY(I10,M8);
IF FLAG2_IN R4=R6*R12(SUF), I9 = MODIFY(I10,M8);
IF NOT LCE MODIFY(I3,M1);
IF NOT LCE I0 = MODIFY(I3,M1);
MODIFY(I10,M9);
I15 = MODIFY(I11,M12);
I0 = MODIFY(I2,M2);
I3 = MODIFY(I3,M5); /* Semantically same as MODIFY(I3,M5) */;

SHARC Processor Programming Reference 9-29
Group II - Conditional Program Flow Control Instructions

The group II instructions contain data move operation and compute/else compute operation.

The \texttt{COND} field selects whether the operation specified in the \texttt{COMPUTE} field and branch are executed. If the \texttt{COND} is true, the compute and branch are executed. If no condition is specified, \texttt{COND} is true condition, and the compute and branch are executed.

The \texttt{ELSE} field selects whether the condition is not true, in this case the computation is performed. The \texttt{ELSE} condition always requires an condition.

The \texttt{COMPUTE} field 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 Chapter 10, Computation Instructions.

- “Type 8a ISA/VISA (cond + branch)” on page 9-32
- “Type 9a ISA/VISA (cond + Branch + comp/else comp)” on page 9-35
- “Type 10 ISA (cond + branch + else comp + mem data move)” on page 9-40
- “Type 11a ISA/VISA (cond + branch return + comp/else comp) Type 11c VISA (cond + branch return)” on page 9-44
- “Type 12a ISA/VISA (do until loop counter expired)” on page 9-48
- “Type 13a ISA/VISA (do until termination)” on page 9-49
The following table provides an overview of the Group II instructions. The letter after the instruction type denotes the instruction size as follows: a = 48-bit, b = 32-bit, c = 16-bit. Note that items in *italics* are optional.

<table>
<thead>
<tr>
<th>Type</th>
<th>Addr</th>
<th>Option1</th>
<th>Operation</th>
<th>Option2</th>
</tr>
</thead>
<tbody>
<tr>
<td>8a</td>
<td>ISA</td>
<td>IF condition</td>
<td>CALL &lt;addr24&gt; [(PC,&lt;reladdr24&gt;)] JUMP &lt;addr24&gt; [(PC,&lt;reladdr24&gt;)] (DB)(LA)(CI)(DB,LA)(DB,CI)</td>
<td></td>
</tr>
<tr>
<td>9a</td>
<td>ISA</td>
<td>IF condition</td>
<td>CALL (Md, Ic) [(PC,&lt;reladdr6&gt;)] JUMP (Md, IC) [(PC, &lt;reladdr6&gt;)] (DB)(LA)(CI)(DB,LA)(DB,CI)</td>
<td>ELSE compute; compute;</td>
</tr>
<tr>
<td>9b</td>
<td>VISA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10a</td>
<td>ISA</td>
<td>IF condition</td>
<td>JUMP (Md, Ic). [(PC,&lt;reladdr6&gt;)]</td>
<td>ELSE compute, DM(Ia,Mb) = DREG; DREG = DM(Ia,Mb);</td>
</tr>
<tr>
<td>11a</td>
<td>ISA</td>
<td>IF condition</td>
<td>RTS (DB)(LR)(DB;LR),RTI (DB).</td>
<td>ELSE compute;</td>
</tr>
<tr>
<td>11c</td>
<td>VISA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12a</td>
<td>ISA</td>
<td></td>
<td>LCNTR = &lt;data16&gt;, DO &lt;addr24&gt; UNTIL LCE; LCNTR = &lt;data16&gt;, DO (PC,&lt;reladdr24&gt;) UNTIL LCE; LCNTR = UREG, DO &lt;addr24&gt; UNTIL LCE; LCNTR = UREG, DO(PC,&lt;reladdr24&gt;) UNTIL LCE;</td>
<td></td>
</tr>
<tr>
<td>13a</td>
<td>ISA</td>
<td></td>
<td>DO &lt;addr24&gt; UNTIL termination; DO (PC,&lt;reladdr24&gt;) UNTIL termination;</td>
<td></td>
</tr>
</tbody>
</table>
Group II - Conditional Program Flow Control Instructions

Type 8a ISA/VISA (cond + branch)

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

Syntax

**SISD Mode**

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 programs reuse an interrupt while it is being serviced

Normally, the processors ignore and do 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 SHARC processor system. The jump (CI) instruction should be located within the interrupt service routine. For more information on interrupts, see Chapter 4, Program Sequencer.

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 processor then allows the interrupt to occur again.

When returning from a reduced subroutine, programs 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 11a ISA/VISA (cond + branch return + comp/else comp) Type 11c VISA (cond + branch return)” on page 9-44.

SIMD Mode

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 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)

| IF (PEx AND PEy COND) JUMP | <addr24> | (DB) |
| (PC, <reladdr24>) | (LA) |
| (CI) | (DB, LA) |
| (DB, CI) |

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

### SIMD Implicit Operation (PEy Operation IMPLIED by the Instruction Syntax)

/* No implicit PEy operation */

**Examples**

```plaintext
IF AV JUMP(PC, OxA4) (LA);
CALL init (DB); /* init is a program label */
JUMP (PC, 2) (DB, CI); /* clear current int. for reuse */
```

When the processors are in SISD mode, 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 processors are in SIMD mode, the first instruction performs a jump to the PC-relative address depending on the logical ANDing of the outcomes of the conditions tested in both PEs. 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 `init` occurs. A PC-relative jump takes place in the third instruction.
### Instruction Set Types

**Type 9a ISA/VISA (cond + Branch + comp/else comp)**

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

**Type 9a Syntax**

IF COND JUMP

<table>
<thead>
<tr>
<th>(Md, Ic)</th>
<th>(DB)</th>
<th>, compute</th>
</tr>
</thead>
<tbody>
<tr>
<td>(PC, &lt;reladdr6&gt;)</td>
<td>(LA)</td>
<td>, ELSE compute</td>
</tr>
<tr>
<td>(CI)</td>
<td>(DB, LA)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(DB, CI)</td>
<td></td>
</tr>
</tbody>
</table>

IF COND CALL

<table>
<thead>
<tr>
<th>(Md, Ic)</th>
<th>(DB)</th>
<th>, compute</th>
</tr>
</thead>
<tbody>
<tr>
<td>(PC, &lt;reladdr6&gt;)</td>
<td>(LA)</td>
<td>, ELSE compute</td>
</tr>
</tbody>
</table>

**Type 9b Syntax**

Indirect (or PC-relative) jump/call, optional condition, *without* the Type 9 optional compute operation
Group II - Conditional Program Flow Control Instructions

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 programs reuse an interrupt while it is being serviced

Normally, the 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 system. The jump (CI) instruction should be located within the interrupt service routine. For more information on interrupts, see Chapter 4, Program Sequencer.

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 processor then allows the interrupt to occur again.

When returning from a reduced subroutine, programs 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 11a ISA/VISA (cond + branch return + comp/else comp) Type 11c VISA (cond + branch return)” on page 9-44.

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.

SIMD Mode

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.
Group II - Conditional Program Flow Control Instructions

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 2-3 on page 2-6.

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

<table>
<thead>
<tr>
<th>SIMD Explicit Operation (PEx Operation Stated in the Instruction Syntax)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF (PEx AND PEy COND) JUMP</td>
</tr>
<tr>
<td>(Md, Ic)</td>
</tr>
<tr>
<td>(PC, &lt;reladdr6&gt;)</td>
</tr>
<tr>
<td>(DB)</td>
</tr>
<tr>
<td>(LA)</td>
</tr>
<tr>
<td>(CI)</td>
</tr>
<tr>
<td>(DB, LA)</td>
</tr>
<tr>
<td>(DB, CI)</td>
</tr>
<tr>
<td>, (if PEx COND) compute</td>
</tr>
<tr>
<td>, ELSE (if NOT PEx) compute</td>
</tr>
<tr>
<td>;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SIMD Implicit Operation (PEy Operation Implied by the Instruction Syntax)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF (PEx AND PEy COND) CALL</td>
</tr>
<tr>
<td>(Md, Ic)</td>
</tr>
<tr>
<td>(PC, &lt;reladdr6&gt;)</td>
</tr>
<tr>
<td>(DB)</td>
</tr>
<tr>
<td>, (if PEx COND) compute</td>
</tr>
<tr>
<td>, ELSE (if NOT PEx) compute</td>
</tr>
<tr>
<td>;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SIMD Explicit Operation (PEx Operation Stated in the Instruction Syntax)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF (PEx AND PEy COND) CALL</td>
</tr>
<tr>
<td>(Md, Ic)</td>
</tr>
<tr>
<td>(PC, &lt;reladdr6&gt;)</td>
</tr>
<tr>
<td>(DB)</td>
</tr>
<tr>
<td>(LA)</td>
</tr>
<tr>
<td>(CI)</td>
</tr>
<tr>
<td>(DB, LA)</td>
</tr>
<tr>
<td>(DB, CI)</td>
</tr>
<tr>
<td>, (if PEx COND) compute</td>
</tr>
<tr>
<td>, ELSE (if NOT PEx) compute</td>
</tr>
<tr>
<td>;</td>
</tr>
</tbody>
</table>
Instruction Set Types

Examples

\texttt{JUMP(M8,112), \texttt{R6=R6-1};}
\texttt{IF EQ CALL(PC,17)(DB), ELSE R6=R6-1;}

When the processors are in SISD mode, 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 processors are in SIMD mode, the indirect jump in the first instruction occurs in parallel with both processing elements executing computations. In PEx, \texttt{R6} stores the result, and \texttt{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 \texttt{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, \texttt{R6} stores the result of the computation in PEx, and \texttt{S6} stores the result of the computation in PEy.
Group II - Conditional Program Flow Control Instructions

Type 10 ISA (cond + branch + else comp + mem data move)

Indirect (or PC-relative) jump or optional compute operation with transfer between data memory and register file. This instruction is not supported for VISA instructions.

Syntax

<table>
<thead>
<tr>
<th>IF COND Jump (Md, Ic) , Else compute, DM(Ia, Mb) = dreg ; compute, dreg = DM(Ia, Mb) ;</th>
</tr>
</thead>
<tbody>
<tr>
<td>(PC, &lt;reladdr6&gt;)</td>
</tr>
</tbody>
</table>

SISD Mode

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 conditon 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, twos-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, programs 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.
SIMD Mode

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 2-3 on page 2-6. 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 (Ia) to address memory. The I register value is post-modified by the specified M register (Mb) 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.

Broadcast Mode

If the broadcast read bits—BDCST1 (for I1) or BDCST9 (for I9)—are set, the Y element uses the specified I register without adding one.
Group II - Conditional Program Flow Control Instructions

Examples

IF TF JUMP(M8, I8), ELSE R6=DM(I6, M1);

IF NE JUMP(PC, 0x20), ELSE F12=FLOAT R10 BY R3, R6=DM(I5, M0);

When the processors are in SISD mode, 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 processors are in SIMD mode, 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 ANDing 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...
based on the negative evaluation of the condition code seen by that processing element.
IF TF JUMP(M8,I8), ELSE R6=DM(I1,M1):

When the processors are in broadcast mode (the BDCST1 bit is 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.
Group II - Conditional Program Flow Control Instructions

Type 11a ISA/VISA (cond + branch return + comp/else comp)
Type 11c VISA (cond + branch return)

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

Type 11a Syntax

\[
\text{IF COND RTS} \begin{cases} 
(DB) \\
(LR) \\
(DB, LR) 
\end{cases}, \text{ compute} \\
, \text{ ELSE compute}
\]

\[
\text{IF COND RTI} \begin{cases} 
(DB) 
\end{cases}, \text{ compute} \\
, \text{ ELSE compute}
\]

Type 11c Syntax

Indirect (or PC-relative) jump with transfer between data memory and register file; \textit{without} Type 11 optional compute operation

\[
\text{IF COND RTS} \begin{cases} 
(DB) \\
(LR) \\
(DB, LR) 
\end{cases}
\]

\[
\text{IF COND RTI} \begin{cases} 
(DB) 
\end{cases}
\]

SISD Mode

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
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 or the timer 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, programs 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 processor 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.

Programs 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 8a ISA/VISA (cond + branch)” on page 9-32 or “Type 9a ISA/VISA (cond + Branch + comp/else comp)” on page 9-35.
SIMD Mode

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 2-3 on page 2-6.

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

<table>
<thead>
<tr>
<th>SIMD Explicit Operation (PEx Operation Stated in the Instruction Syntax)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF (PEx AND PEy COND) RTS</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>IF (PEx AND PEy COND) RTI</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SIMD Implicit Operation (PEy Operation Implied by the Instruction Syntax)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF (PEx AND PEy COND) RTS</td>
</tr>
<tr>
<td>IF (PEx AND PEy COND) RTI</td>
</tr>
</tbody>
</table>
Examples

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

When the processors are in SISD mode, 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 processors are in SIMD mode, 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 ANDing 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.
**Group II - Conditional Program Flow Control Instructions**

**Type 12a ISA/VISA (do until loop counter expired)**

Load loop counter, do loop until loop counter expired

**Syntax**

<table>
<thead>
<tr>
<th>LCNTR</th>
<th>, DO</th>
<th>UNTIL LCE;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;data16&gt;</td>
<td>&lt;addr24&gt;</td>
<td>(PC, &lt;reladdr24&gt;)</td>
</tr>
</tbody>
</table>

**SISD and SIMD Modes**

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 two’s-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{verbatim}
LCNTR=100, DO fmax UNTIL LCE; /* fmax is a program label */
LCNTR=R12, DO (PC,16) UNTIL LCE;
\end{verbatim}

The processor (in SISD or SIMD) executes the action at the indicated address for the duration of the loop.
Instruction Set Types

Type 13a ISA/VISA (do until termination)

Do until termination

Syntax

SISD Mode

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 twos-complement address. The loop executes until the termination condition tests true.

SIMD Mode

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

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

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

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

```
/* No implicit PEy operation */
```
Examples

DO end UNTIL FLAG1_IN;  /* end is a program label */
DO (PC,7) UNTIL AC;

When the processors are in SISD mode, 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 instruction, the start address is given in the form of a PC-relative address. The loop executes until the instruction’s condition tests true.

When the processors are in SIMD mode, 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.
**Group III - Immediate Data Move Instructions**

The group III instructions contain data move operation with immediate data or indirect addressing.

- “Type 14a ISA/VISA (mem data move)” on page 9-53
- “Type 15a ISA/VISA (<data32> move) Type 15b VISA (<data7> move)” on page 9-56
- “Type 16a ISA/VISA (<data32> move) Type 16b VISA (<data16> move)” on page 9-60
- “Type 17a ISA/VISA (<data32> move) Type 17b VISA (<data16> move)” on page 9-62

The following table provides an overview of the Group III instructions. The letter after the instruction type denotes the instruction size as follows: a = 48-bit, b = 32-bit, c = 16-bit.

<table>
<thead>
<tr>
<th>Type</th>
<th>Addr</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>14a</td>
<td>ISA</td>
<td>DM(&lt;addr32&gt;) = UREG(LW); PM(&lt;addr32&gt;)</td>
</tr>
<tr>
<td></td>
<td>VISA</td>
<td>UREG = DM(&lt;addr32&gt;)(LW); PM(&lt;addr32&gt;)</td>
</tr>
<tr>
<td>15a</td>
<td>ISA</td>
<td>DM(&lt;data32&gt;,Ia) = UREG(LW); PM(&lt;data32&gt;,Ic)</td>
</tr>
<tr>
<td></td>
<td>VISA</td>
<td>UREG = DM(&lt;data32&gt;,Ia)(LW); PM(&lt;data32&gt;,Ic)</td>
</tr>
<tr>
<td>15b</td>
<td>VISA</td>
<td>DM(&lt;data7&gt;,Ia) = UREG(LW); PM(&lt;data7&gt;,Ic)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UREG = DM(&lt;data7&gt;,Ia)(LW); PM(&lt;data7&gt;,Ic)</td>
</tr>
<tr>
<td>16a</td>
<td>ISA</td>
<td>DM(Ia,Mb) = &lt;data32&gt;; PM(Ic,Md)</td>
</tr>
<tr>
<td></td>
<td>VISA</td>
<td></td>
</tr>
</tbody>
</table>

SHARC Processor Programming Reference 9-51
### Group III - Immediate Data Move Instructions

<table>
<thead>
<tr>
<th>Type</th>
<th>Addr</th>
<th>Operation</th>
</tr>
</thead>
</table>
| 16b  | VISA | \( \text{DM}(Ia,Mb) = \text{<data16>}; \)  
\( \text{PM}(Ic,Md) \) |
| 17a  | ISA  | \( \text{UREG} = \text{<data32>}; \) |
| 17b  | VISA | \( \text{UREG} = \text{<data16>}; \) |
Instruction Set Types

Type 14a ISA/VISA (mem data move)

Type 14a Syntax

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

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

SISD Mode

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 programs specify long word addressing, overriding default addressing from the memory map.

SIMD Mode

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 \(k\) to the specified 32-bit address to address memory.
For the universal register, the X element uses the specified $U_{reg}$, and the Y element uses the complementary register ($C_{ureg}$) that corresponds to the $U_{reg}$ register specified in the instruction. For a list of complementary registers, see Table 2-3 on page 2-6. Note that only the $C_{ureg}$ subset registers which have complementary registers are effected by SIMD mode.

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

**Examples**

```plaintext
DM(temp)=MODE1;   /* temp is a program label */
LCNTR=PM(0x90500);
```

When the processors are in SISD mode, 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 LCNTR 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 \texttt{MODE1 (SREG)} and \texttt{LCNTR (UREG)} registers have no complements, so they do not operate differently in SIMD mode.
Group III - Immediate Data Move Instructions

Type 15a ISA/VISA (<data32> move)  
Type 15b VISA (<data7> move)

Type 15a Syntax
Transfer between data or program memory and universal register, indirect addressing, immediate modifier

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

Type 15b Syntax
Transfer (7-bit data) between data or program memory and universal register, indirect addressing, immediate modifier

\[
\begin{align*}
\text{DM(<data7>, Ia)} & \quad = \quad \text{ureg} \\
\text{PM(<data7>, Ic)} & \\
\text{ureg} = & \quad \text{DM(<data7>, Ia)} \\
& \quad \text{PM(<data7>, Ic)} \\
\end{align*}
\]

SISD Mode
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 Chapter 6, Data Address Generators. The optional (LW) in this syntax lets programs specify long word addressing, overriding default addressing from the memory map.

SIMD Mode

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 k 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 Ia/Mb or Ic/Md. 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 2-3 on page 2-6. Note that only the Cureg subset registers which have complimentary registers are effected by SIMD mode. For more information on register restrictions, see Chapter 6, Data Address Generators.

The following code compares the Type 15 instruction’s explicit and implicit operations in SIMD mode.
Type 15a ISA/VISA (<data32> move) Type 15b VISA (<data7> move)

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

<table>
<thead>
<tr>
<th>DM(&lt;data32&gt;, Ia)</th>
<th>=  ureg (LW);</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM(&lt;data32&gt;, Ic)</td>
<td></td>
</tr>
</tbody>
</table>

ureg =

<table>
<thead>
<tr>
<th>DM(&lt;data32&gt;, Ia)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM(&lt;data32&gt;, Ic)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>DM(&lt;data32&gt;+k, Ia)</th>
<th>=  cureg (LW);</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM(&lt;data32&gt;+k, Ic)</td>
<td></td>
</tr>
</tbody>
</table>

cureg =

<table>
<thead>
<tr>
<th>DM(&lt;data32&gt;+k, Ia)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM(&lt;data32&gt;+k, Ic)</td>
</tr>
</tbody>
</table>

Examples

DM(24, I5)=TCOUNT;
USTAT1=PM(offs, I13); /* "offs" is a user-defined constant */

When the processors are in SISD mode, 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, $USTAT_1$ and its complimentary register $USTAT_2$—is performed in parallel on PEx 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 $k$ to address memory on PEy. The $I13$ register is not updated after the memory access occurs in SIMD mode.
Group III - Immediate Data Move Instructions

Type 16a ISA/VISA (<data32> move)
Type 16b VISA (<data16> move)

Type 16a Syntax

Immediate data write to data or program memory

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

Type 16b Syntax

Immediate 16-bit data write to data or program memory

\[
\begin{align*}
DM(Ia, Mb) & = <data16> ; \\
PM(Ic, Md) & = <data16> ;
\end{align*}
\]

SISD Mode

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.

SIMD Mode

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 k to the I register to address memory. The I register is post-modified and updated by the specified M register.
Instruction Set Types

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

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

| DM(Ia, Mb) | = <data32> ; |
| PM(Ic, Md) |

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

| DM(Ia+k, 0) | = <data32> ; |
| PM(Ic+k, 0) |

If broadcast mode k=0.
If SIMD mode NW access k=1, SW access k=2.

Examples

```
DM(14, M0)=19304;
PM(114, M11)=count; /* count is user-defined constant */
```

When the processors are in SISD mode, 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 I4 and I14 registers are post-modified and updated by M0 and M11 respectively.

When the processors are in SIMD mode, 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 I4 and I14 registers are post-modified and updated by M0 and M11 respectively.
Group III - Immediate Data Move Instructions

Type 17a ISA/VISA (<data32> move)
Type 17b VISA (<data16> move)

Type 17a Syntax
Immediate 32-bit data write to universal register

\[
\text{ureg} = \langle\text{data32}\rangle ;
\]

Type 17b Syntax
Immediate 16-bit data write to universal register

\[
\text{ureg} = \langle\text{data16}\rangle ;
\]

SISD Mode
In SISD mode, the Type 17 instruction writes 16-bit/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.

SIMD Mode
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 Creg. Note that only the Creg subset registers which have complimentary registers are effected by SIMD mode. For a list of complimentary registers, see Table 2-3 on page 2-6.
The following code compares the Type 17 instruction’s explicit and implicit operations in SIMD mode.

SIMD Explicit Operation (PEx Operation Stated in the Instruction Syntax)
ureg = <data32> ;

SIMD Implicit Operation (PEy Operation Implied by the Instruction Syntax)
cureg = <data32> ;

SIMD Explicit Operation (PEx Operation Stated in the Instruction Syntax)
ureg = <data16> ;

SIMD Implicit Operation (PEy Operation Implied by the Instruction Syntax)
cureg = <data16> ;

Examples

ASTATx=0x0;
M15=mod1;  /* mod1 is user-defined constant */

When the processors are in SISD mode, the two instructions load immediate values into the specified registers.

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 complimentary 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. So, the second instruction operates the same in SIMD and SISD mode.
Group IV – Miscellaneous Instructions

The group IV instructions contain miscellaneous operations.

- “Type 18a ISA/VISA (register bit manipulation)” on page 9-66
- “Type 19a ISA/VISA (index modify/bitrev)” on page 9-69
- “Type 20a ISA/VISA (push/pop stack)” on page 9-70
- “Type 21a ISA/VISA (nop) Type 21c VISA (nop)” on page 9-71
- “Type 22a ISA/VISA (idle/emuidle)” on page 9-72
- “Type 25a ISA/VISA (cjump/rframe) Type 25c VISA (RFRAME)” on page 9-73

The following table provides an overview of the Group II instructions. The letter after the instruction type denotes the instruction size as follows: a = 48-bit, b = 32-bit, c = 16-bit.

<table>
<thead>
<tr>
<th>Type</th>
<th>Addr</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>18a</td>
<td>ISA</td>
<td>BIT SET SREG &lt;data32&gt;;</td>
</tr>
<tr>
<td></td>
<td>VISA</td>
<td>CLR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TGL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TST</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XOR</td>
</tr>
<tr>
<td>19a</td>
<td>ISA</td>
<td>BITREV (Ia, &lt;data32&gt;);</td>
</tr>
<tr>
<td></td>
<td>VISA</td>
<td>(Ic, &lt;data32&gt;);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MODIFY (Ia, &lt;data32&gt;);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Ic, &lt;data32&gt;);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ja = MODIFY (Ia, &lt;data32&gt;); // for ADSP-214xx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ic = MODIFY (Ic, &lt;data32&gt;); // for ADSP-214xx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ja = BITREV (Ia, &lt;data32&gt;); // for ADSP-214xx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ic = BITREV (Ic, &lt;data32&gt;); // for ADSP-214xx</td>
</tr>
<tr>
<td>20a</td>
<td>ISA</td>
<td>PUSH LOOP, PUSH STS, PUSH PCSTK,</td>
</tr>
<tr>
<td></td>
<td>VISA</td>
<td>POP LOOP, POP STS, POP PCSTK,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FLUSH CACHE;</td>
</tr>
</tbody>
</table>
## Instruction Set Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Addr</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>21a</td>
<td>ISA</td>
<td>VISA NOP;</td>
</tr>
<tr>
<td>21c</td>
<td>VISA</td>
<td></td>
</tr>
<tr>
<td>22a</td>
<td>ISA</td>
<td>VISA IDLE;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EMUIDLE;</td>
</tr>
<tr>
<td>22c</td>
<td>VISA</td>
<td></td>
</tr>
<tr>
<td>23–24</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>25a</td>
<td>ISA</td>
<td>VISA CJUMP &lt;addr24&gt; (db);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CJUMP (PC, &lt;reladdr24&gt;) (db);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RFRAME;</td>
</tr>
<tr>
<td>25c</td>
<td>VISA</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RFRAME;</td>
</tr>
</tbody>
</table>
Group IV - Miscellaneous Instructions

Type 18a ISA/VISA (register bit manipulation)

System register bit manipulation

Syntax

<table>
<thead>
<tr>
<th>BIT</th>
<th>SET</th>
<th>CLR</th>
<th>TGL</th>
<th>TST</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sreg &lt;data32&gt; ;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SISD Mode

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 Chapter 10, Computation Instructions. For more information on system registers, see Appendix A, Registers.
**SIMD Mode**

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 2-3 on page 2-6.

The following 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><strong>BIT</strong></td>
<td><strong>sreg</strong>&lt;sup&gt; &lt;/sup&gt;&lt;data32&gt; ;</td>
</tr>
<tr>
<td>SET</td>
<td></td>
</tr>
<tr>
<td>CLR</td>
<td>csreg &lt;data32&gt; ;</td>
</tr>
<tr>
<td>TGL</td>
<td></td>
</tr>
<tr>
<td>TST</td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td></td>
</tr>
</tbody>
</table>

**Examples**

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

When the processors are in SISD mode, 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 \textit{Cureg} subset registers which have complimentary registers are affected in SIMD mode. The \textit{ASTATx} (system) register is included in the \textit{Cureg} subset, so the bit test operations are performed independently on each processing element in parallel using these complimentary registers. The \textit{BTF} is set on both PEs (\textit{ASTATx} and \textit{ASTATy}), either one PE (\textit{ASTATx} or \textit{ASTATy}), or neither PE dependent on the outcome of the bit test operation.
Instruction Set Types

**Type 19a ISA/VISA (index modify/bitrev)**

Immediate I register modify or bit-reverse

**Syntax**

```
Ia = MODIFY (Ia, <data32>)
Ic = BITREV (Ic, <data32>)
```

**SISD and SIMD Modes**

In SISD and SIMD modes, the Type 19 instruction modifies and adds the specified source Ia/Ic register with an immediate 32-bit data value and stores the result to the specified destination Ia/Ic register (ADSP-214xx processors only). If no destination register is specified then the source I register is updated. If the address is to be bit-reversed (as specified by mnemonic), the modified value is bit-reversed before being written back to the destination I register. No address is output in either case. For more information on register restrictions, see Chapter 6, Data Address Generators.

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 wraparound, independent of the CBUFEN bit.

**Examples**

```
MODIFY (I4, 304);
/* operation is the same as I4=MODIFY(I4,304) */
BITREV (I7, space);
/* “space” is a user-defined constant. */
    operation is the same as I7=BITREV(I7,space) */
I3 = MODIFY (12,0x123);
I9 = MODIFY (19,0x1);
I2 = BITREV (11,122);
I15 =BITREV(I12,0x10);
```
Group IV - Miscellaneous Instructions

Type 20a ISA/VISA (push/pop stack)

Push or Pop of loop and/or status stacks

Syntax

SISD and SIMD Modes

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, and has an effect latency of one instruction when executing from internal memory, and two instructions when executing from external memory.

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.
Instruction Set Types

Type 21a ISA/VISA (nop)
Type 21c VISA (nop)

Type 21a Syntax
No Operation (NOP)

NOP ;

Type 21c Syntax
No operation (NOP)

NOP

SISD and SIMD Modes
In SISD and SIMD modes, the Type 21 instruction provides a null operation; it increments only the fetch address.
Group IV - Miscellaneous Instructions

Type 22a ISA/VISA (idle/emuidle)

Low power/emulation halt instruction

Type 22a Syntax

<table>
<thead>
<tr>
<th>IDLE ;</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMUIDLE ;</td>
</tr>
</tbody>
</table>

SISD and SIMD Modes

In SISD and SIMD modes, the Type 22 idle instruction 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. The emuidle instruction halts the core caused by a software breakpoint hit and places the core in emulation space. An RTI instruction releases the core back to user space.
**Instruction Set Types**

**Type 25a ISA/VISA (cjump/rframe)**

**Type 25c VISA (RFRAME)**

**Type 25a Syntax**

Cjump/Rframe (Compiler-generated instruction)

\[
\text{CJUMP function (DB)}; \\
\text{(PC, <reladdr24>)} \\
\text{RFRAME;} \\
\]

**Type 25c Syntax**

Rframe (Compiler-generated instruction); *without* Type 25 Cjump option

\[
\text{RFRAME;} \\
\]

**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 assembly programs. The cjump instruction should always use the DB modifier.
The different forms of this instruction perform the operations listed in Table 9-2 where \( \text{raddr} \) indicates a relative 24-bit address.

Table 9-2. 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); )</td>
</tr>
</tbody>
</table>
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-21xxx 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 9, “Instruction Set Types”.

In single-function operations (other than type 6), the compute field of a single-function operation is made up of the bit fields described in this chapter. For type 6 instructions (immediate shift) see “Type 6 ISA/VISA (cond + shift imm + mem data move)” on page 9-25.
ALU Operations

This section describes the ALU operations. For information on syntax and opcodes, see “Computation Instructions” in Chapter 10, Computation Instructions and “Instruction Opcodes” in Chapter 11, Instruction Opcodes.

\[ R_n = R_x + R_y \]

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
**Computation Instructions**

**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 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>
**Compute Field**

\[ R_n = R_x + R_y + C_i \]

**Function**

Adds with carry (\( AC \) from \( ASTAT \)) the fixed-point fields in registers \( R_x \) and \( R_y \). 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
### Computation Instructions

**Rn = Rx - Ry + CI - 1**

**Function**

Subtracts with borrow (AC – 1 from ASTAT) 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 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

\[ R_n = \frac{R_x + R_y}{2} \]

**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
**Computation Instructions**

**COMP(Rx, Ry)**

**Function**

Compares the signed 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 $ASTAT$ 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>Set if the signed operands in registers Rx and Ry are equal, otherwise cleared</td>
</tr>
<tr>
<td>AU</td>
<td>Cleared</td>
</tr>
<tr>
<td>AN</td>
<td>Set if the signed operand in the Rx register is smaller than the operand in the Ry register, 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

**COMPU(Rx, Ry)**

**Function**

Compares the unsigned 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>Set if the unsigned operands in registers Rx and Ry are equal, otherwise cleared</td>
</tr>
<tr>
<td>AU</td>
<td>Cleared</td>
</tr>
<tr>
<td>AN</td>
<td>Set if the unsigned operand in the Rx register is smaller than the operand in the Ry register, 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>
Computation Instructions

**Rn = Rx + CI**

**Function**

Adds the fixed-point field in register Rx with the carry flag from the ASTAT register (AC). 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).

**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**

\[ R_n = R_x + C1 - 1 \]

**Function**

Adds the fixed-point field in register Rx with the borrow from the ASTAT register (AC – 1). 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).

**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
Computation Instructions

**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**

- **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**

\[ R_n = R_x - 1 \]

**Function**

Decrements the fixed-point operand 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 \texttt{MODE1} set), underflow causes the minimum negative number (0x8000 0000) 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**: Cleared
- **AI**: Cleared
**Computation Instructions**

\[ R_n = -R_x \]

**Function**

Negates the fixed-point operand in \( R_x \) by two’s-complement. 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. Negation of the minimum negative number (0x8000 0000) causes an overflow. In saturation mode (the ALU saturation mode bit in \text{MODE1} \) set), overflow causes the maximum positive number (0x7FFFFFFF) 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
**Compute Field**

**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 (0x7FFFF 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
Computation Instructions

**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
**Compute Field**

\[ R_n = Rx \text{ AND } Ry \]

**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**

- **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
**Computation Instructions**

**Rn = 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**

<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>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

\( R_n = R_x \text{ XOR } R_y \)

Function

Logically XORs the fixed-point operands in \( R_x \) and \( R_y \). The result is placed in the fixed-point field in \( R_n \). The floating-point extension field in \( R_n \) 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
Computation Instructions

**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**

- **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 = 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**

| 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 |
Computation Instructions

\[ R_n = \text{MAX}(R_x, R_y) \]

Function

Returns the larger of the two fixed-point operands in \( R_x \) and \( R_y \). 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.

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**

\[ R_n = \text{CLIP} \, R_x \, \text{BY} \, R_y \]

**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 \(|R_y|\) if Rx is positive, and \(-|R_y|\) 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
Computation Instructions

\[ Fn = Fx + Fy \]

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
Compute Field

\( Fn = Fx - Fy \)

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

- **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 like-signed infinities, otherwise cleared
**Computation Instructions**

**Fn = ABS (Fx + Fy)**

**Function**

Adds the floating-point operands in registers Fx and Fy, and places the absolute value of the normalized result 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**: 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
**Compute Field**

\[ F_n = \text{ABS} (F_x - F_y) \]

**Function**

Subtracts the floating-point operand in Fy from the floating-point operand in Fx and places the absolute value of the normalized result 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**: 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 like-signed infinities, otherwise cleared
Fn = (Fx + Fy)/2

Function

Adds the floating-point operands in registers Fx and Fy and divides the result by 2, by decrementing the exponent of the sum before rounding. 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 results return ±zero. A denormal input is 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 |
**Compute Field**

**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
Computation Instructions

\textbf{Fn} = –Fx

\textbf{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 ±zero. A NAN input returns an all 1s result.

\textbf{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
Compute Field

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>AZ</th>
<th>Set if the result operand is +zero, otherwise cleared</th>
</tr>
</thead>
<tbody>
<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>
**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
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 ±infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). A denormal input is 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  | 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    |
**Computation Instructions**

**Fn = SCALB Fx BY Ry**

**Function**

Scales the exponent of the floating-point operand in Fx by adding to it the fixed-point two’s-complement integer in Ry. The scaled floating-point result is placed in register Fn. Overflow returns ±infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). Denormal returns ±zero. Denormal inputs are flushed to ±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
**Compute Field**

**Rn = MANTFx**

**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
- **AU**: Cleared
- **AN**: Cleared
- **AV**: Cleared
- **AC**: Cleared
- **AS**: Set if the input is negative, otherwise cleared
- **AI**: Set if the input operands is a NAN or an infinity, otherwise cleared
**Computation Instructions**

**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
**Compute Field**

\[
\begin{align*}
Rn &= \text{FIX } Fx \\
Rn &= \text{TRUNC } Fx \\
Rn &= \text{FIX } Fx \text{ BY } Ry \\
Rn &= \text{TRUNC } Fx \text{ BY } Ry
\end{align*}
\]

**Function**

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

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

The trunc operation always truncates toward 0. The 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.
Computation Instructions

All positive underflows return zero. Negative underflows that are rounded-to-nearest return zero, and negative underflows that are rounded by truncation return −1 (0xFFF 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
**Function**

Converts the fixed-point operand in Rx to a floating-point result. 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 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**

<table>
<thead>
<tr>
<th>AZ</th>
<th>Set if the result is a denormal (unbiased exponent &lt; –126) or zero, otherwise cleared</th>
</tr>
</thead>
<tbody>
<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 result overflows (unbiased exponent &gt;127)</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>
**Computation Instructions**

 Fn = 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 two’s-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 ±zero returns ±infinity and sets the overflow flag. If the unbiased exponent of Fx is greater than +125, the result is ±zero. A NAN input returns an all 1s result.

The following code performs floating-point division using an iterative convergence algorithm. The result is accurate to one LSB in whichever format mode, 32-bit or 40-bit, is set. The following inputs are required: F0=numerator, F12=denominator, F11=2.0. The quotient is returned in F0. (The two indented instructions can be removed if only a ±1 LSB accurate single-precision result is necessary.) Note that, in the algorithm example’s comments, references to R0, R1, R2, and R3 do not refer to data registers. Rather, they refer to variables in the algorithm.

```
F0=RECIPS F12, F7=F0;  /* Get 8-bit seed R0=1/D */
F12=F0*F12;          /* D’ = D*R0 */
F7=F0*F7, F0=F11-F12; /* F0=R1=2-D’, F7=N*R0 */
F12=F0*F12;          /* F12=D’*D’*R1 */
F7=F0*F7, F0=F11-F12; /* F7=N*R0*R1, F0=R2=2-D’ */
F12=F0*F12;          /* F12=D’*D’*R2 */
F7=F0*F7, F0=F11-F12; /* F7=N*R0*R1*R2, F0=R3=2-D’ */
F0=F0*F7;            /* F7=N*R0*R1*R2*R3 */
```

To make this code segment a subroutine, add an RTS(DB) clause to the third-to-last instruction.

---

### Compute Field

#### Status Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZ</td>
<td>Set if the floating-point result is ±zero (unbiased exponent of $Fx$ is greater than +125), otherwise cleared</td>
</tr>
<tr>
<td>AU</td>
<td>Cleared</td>
</tr>
<tr>
<td>AN</td>
<td>Set if the input operand is negative, otherwise cleared</td>
</tr>
<tr>
<td>AV</td>
<td>Set if the input operand is ±zero, 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 the input operand is a NaN, otherwise cleared</td>
</tr>
</tbody>
</table>
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 index.

The unbiased exponent of the seed is calculated as the two’s-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 $F_n = -\text{INT}[e/2] - 1$.

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. The result is accurate to one LSB in whichever format mode, 32-bit or 40-bit, is set.

To calculate the square root, simply multiply the result by the original input. The following inputs are required: $F_0 =$ input, $F_8 = 3.0$, $F_1 = 0.5$. The result is returned in $F_4$. (The four indented instructions can be removed if only a ±1 LSB accurate single-precision result is necessary.)

```
F4=RSQRTS F0; /* Fetch 4-bit seed */
F12=F4*F4;   /* F12=X0^2 */
F12=F12*F0;  /* F12=C*X0^2 */
F4=F1*F4, F12=F8-F12; /* F4=.5*X0, F12=3-C*X0^2 */
F4=F4*F12;   /* F4=X1=.5*X0(3-C*X0^2) */
F12=F4*F4;   /* F12=X1^2 */
```

Compute Field

\[ F_{12} = F_{12} \times F_0; \quad \text{/* } F_{12} = C \times X_1^2 \ */ \]
\[ F_4 = F_1 \times F_4, \quad F_{12} = F_8 - F_{12}; \quad \text{/* } F_4 = 0.5 \times X_1, \quad F_{12} = 3 - C \times X_1^2 \ */ \]
\[ F_4 = F_4 \times F_{12}; \quad \text{/* } F_4 = X_2 = 0.5 \times X_1(3 - C \times X_1^2) \ */ \]
\[ F_{12} = F_4 \times F_4; \quad \text{/* } F_{12} = X_2^2 \ */ \]
\[ F_{12} = F_{12} \times F_0; \quad \text{/* } F_{12} = C \times X_2^2 \ */ \]
\[ F_4 = F_1 \times F_4, \quad F_{12} = F_8 - F_{12}; \quad \text{/* } F_4 = 0.5 \times X_2, \quad F_{12} = 3 - C \times X_2^2 \ */ \]
\[ F_4 = F_4 \times F_{12}; \quad \text{/* } F_4 = X_3 = 0.5 \times X_2(3 - C \times X_2^2) \ */ \]

Note that this code segment can be made into a subroutine by adding an RTS(DB) clause to the third-to-last instruction.

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
**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
**Compute Field**

**Fn = MIN(Fx, Fy)**

**Function**

Returns the smaller of the floating-point operands in register \( Fx \) and \( Fy \). A NAN input returns an all 1s result. The MIN of +zero and –zero returns –zero. 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
**Function**

Returns the larger of the floating-point operands in registers Fx and Fy. A NAN input returns an all 1s result. The MAX of +zero and –zero returns +zero. 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
**Compute Field**

\[ \text{Fn} = \text{CLIP Fx BY Fy} \]

**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 \( | \text{Fx} | \) if Fx is positive, and \( -| \text{Fx} | \) 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. Note that data moves between the MR registers and the data registers are considered multiplier operations and are also covered in this chapter. Group I instructions include the following.

The tables in this chapter use the following symbols to indicate location of operands and other features:

- $y = y$-input (1 = signed, 0 = unsigned)
- $x = x$-input (1 = signed, 0 = unsigned)
- $f = format$ (1 = fractional, 0 = integer)
- $r = rounding$ (1 = yes, 0 = no)

Mod1, Mod2, and Mod3 Modifiers

Mod1, Mod2, and Mod3 are optional modifiers. They are 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 is rounded-to-nearest (R).

“Multiplier Instruction Summary” on page 3-13 provides information on multiplier instructions. Table 3-6 on page 3-16 lists the options for the mod1 – mod3 options and the corresponding opcode values.
**Compute Field**

\[ Rn = Rx \times Ry \, (\mod 1) \]

\[ MRF = Rx \times Ry \, (\mod 1) \]

\[ MRB = Rx \times Ry \, (\mod 1) \]

**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 M 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.
Computation Instructions

\[
\begin{align*}
Rn &= MRF + Rx \times Ry \pmod{1} \\
Rn &= MRB + Rx \times Ry \pmod{1} \\
MRF &= MRF + Rx \times Ry \pmod{1} \\
MRB &= MRB + Rx \times Ry \pmod{1}
\end{align*}
\]

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

\[ R_n = MRF - Rx \times Ry \pmod{1} \]
\[ R_n = MRB - Rx \times Ry \pmod{1} \]
\[ MRF = MRF - Rx \times Ry \pmod{1} \]
\[ MRB = MRB - Rx \times Ry \pmod{1} \]

Function

Multiplies the fixed-point fields in registers Rx and Ry, 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 Rn or in 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
Computation Instructions

Rn = SAT MRF (mod2)
Rn = SAT MRB (mod2)
MRF = SAT MRF (mod2)
MRB = SAT MRB (mod2)

Function

If the value of the specified MR register is greater than the maximum value for the specified data format, the multiplier sets the result to the maximum value. Otherwise, the MR value is unaffected. 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

<table>
<thead>
<tr>
<th>MN</th>
<th>Set if the result is negative, otherwise cleared</th>
</tr>
</thead>
<tbody>
<tr>
<td>MV</td>
<td>Cleared</td>
</tr>
<tr>
<td>MU</td>
<td>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</td>
</tr>
<tr>
<td>MI</td>
<td>Cleared</td>
</tr>
</tbody>
</table>
Compute Field

\[ Rn = \text{RND MRF (mod3)} \]
\[ Rn = \text{RND MRB (mod3)} \]
\[ \text{MRF} = \text{RND MRF (mod3)} \]
\[ \text{MRB} = \text{RND MRB (mod3)} \]

Function

Rounds the specified MR value to nearest at bit 32 (the MR1-MR0 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 MRF or MRB is specified, the entire 80-bit result is placed in MRF or MRB.

Status Flags

<table>
<thead>
<tr>
<th>MN</th>
<th>Set if the result is negative, otherwise cleared</th>
</tr>
</thead>
<tbody>
<tr>
<td>MV</td>
<td>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</td>
</tr>
<tr>
<td>MU</td>
<td>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</td>
</tr>
<tr>
<td>MI</td>
<td>Cleared</td>
</tr>
</tbody>
</table>
Computation Instructions

**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

MRxF/B = Rn
Rn = MRxF/B

Function

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

Syntax Variations

MROF = Rn   Rn = MROF
MR1F = Rn   Rn = MR1F
MR2F = Rn   Rn = MR2F
MR0B = Rn   Rn = MR0B
MR1B = Rn   Rn = MR1B
MR2B = Rn   Rn = MR2B

Status Flags

MN  Not Affected
MV  Not Affected
MU  Not Affected
MI  Not Affected
**Computation Instructions**

\[ F_n = F_x \times F_y \]

**Function**

Multiplies the floating-point operands in registers \( F_x \) and \( F_y \) and places the result in the register \( F_n \).

**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 \pm\infty or \pm\text{zero}, otherwise cleared
Shifter Operations

Shifter operations are described in this section. The succeeding pages provide detailed descriptions of each operation. Some of the instructions accept the following modifiers.

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

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

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

<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 to the left by more than 0, otherwise cleared</td>
</tr>
<tr>
<td>SS</td>
<td>Cleared</td>
</tr>
</tbody>
</table>


**Computation Instructions**

\[
R_n = R_n \text{ OR LSHIFT } R_x \text{ BY } R_y \\
R_n = R_n \text{ OR LSHIFT } R_x \text{ BY } <\text{data8}>
\]

**Function**

Logically shifts the fixed-point operand in register \(R_x\) by the 32-bit value in register \(R_y\) or by the 8-bit immediate value in the instruction. The shifted result is logically ORed with the fixed-point field of register \(R_n\) and then written back to register \(R_n\). The floating-point extension field of \(R_n\) 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 left by more than 0, otherwise cleared
- **SS**: Cleared
**Compute Field**

\[ R_n = \text{ASHIFT}_R \times \text{BY} \ R_y \]
\[ R_n = \text{ASHIFT}_R \times \text{BY} \ <\text{data8}> \]

**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**

- **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
Computation Instructions

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 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 left by more than 0, otherwise cleared
- **SS**: Cleared


**Compute Field**

\[ R_n = \text{ROT} R_x \text{ BY } R_y \]
\[ R_n = \text{ROT} R_x \text{ BY } \text{<data8> } \]

**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**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SZ</td>
<td>Set if the rotated result is zero, otherwise cleared</td>
</tr>
<tr>
<td>SV</td>
<td>Cleared</td>
</tr>
<tr>
<td>SS</td>
<td>Cleared</td>
</tr>
</tbody>
</table>
Computation Instructions

**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. The Bit Clr instruction should not be confused with the BCLR shifter operation. For more information on Bit Clr, see “Type 18a ISA/VISA (register bit manipulation)” on page 9-66.
Compute Field

\[ R_n = \text{BSET } R_x \text{ BY } R_y \]
\[ R_n = \text{BSET } R_x \text{ BY } <\text{data8}> \]

Function

Sets a bit in the fixed-point operand in register \( R_x \). The result is placed in the fixed-point field of register \( R_n \). The floating-point extension field of \( R_n \) is set to all 0s. The position of the bit is the 32-bit value in register \( R_y \) 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

- **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 Set instruction should not be confused with the Bset shifter operation. For more information on Bit Set, see “Type 18a ISA/VISA (register bit manipulation)” on page 9-66.
**Computation Instructions**

\[ R_n = BTGL Rx \ BY \ Rx \]
\[ R_n = BTGL Rx \ BY \ <data8> \]

**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

ℹ️ 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 18a ISA/VISA (register bit manipulation)” on page 9-66.
Comput 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 18a ISA/VISA (register bit manipulation)” on page 9-66.
**Computation Instructions**

\[ \text{Rn} = \text{FDEP Rx BY Ry} \]
\[ \text{Rn} = \text{FDEP Rx BY <bit6>:<len6}> \]

**Function**

Deposits a field from register Rx to register Rn. (See Figure 10-1.) 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 10-1. Field Alignment](image)

---

SHARC Processor Programming Reference 10-65
**Compute Field**

**Example**

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

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

39 31 23 15 7 0
|00000abc|defghijk|lmn00000|00000000|00000000|          Rn
 \-------------/
     bit position 13 (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 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|--       |
\--------/|
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
|abcdeopq|rstuvwxyz|zabtuvwxy|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
**Computation Instructions**

\[ R_n = \text{FDEP} \ Rx \ \text{BY} \ Ry \ (SE) \]
\[ R_n = \text{FDEP} \ Rx \ \text{BY} \ \text{<bit6>:<len6> (SE)} \]

**Function**

Deposits and sign-extends a field from register Rx to register Rn. (See Figure 10-2.) 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.

![Figure 10-2. Field Alignment](image)

**SHARC Processor Programming Reference** 10-69
**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)
```

**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
Computation Instructions

\[ R_n = R_n \text{ OR } \text{FDEP } R_x \text{ BY } R_y \text{ (SE)} \]
\[ R_n = R_n \text{ OR } \text{FDEP } R_x \text{ BY } \text{<bit6>:<len6>} \text{ (SE)} \]

**Function**

Deposits and sign-extends a field from register \( R_x \) to register \( R_n \). The sign-extended field value is logically ORed bitwise with the value of register \( R_n \) and the new value is written back to register \( R_n \). The input field is right-aligned within the fixed-point field of \( R_x \). Its length is determined by the \( \text{len6} \) field in register \( R_y \) or by the immediate \( \text{len6} \) field in the instruction. The field is deposited in the fixed-point field of \( R_n \), starting from a bit position determined by the \( \text{bit6} \) field in register \( R_y \).

The bit position can also be determined by the immediate \( \text{bit6} \) 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.

**Example**

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

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

39   31   23   15   7   0
|abcdefg|hijklmnop|qrstuvwxy|yzabcdef|ghijklmn|      Rn old
```
**Compute Field**

39 31 23 15 7 0
|vwxyzabc|defghijk|lmntuvwxyz|yzabcdef|ghijklmn| 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
**Computation Instructions**

**Rn = FEXT Rx BY Ry**
**Rn = FEXT Rx BY <bit6>:<len6>**

**Function**

Extracts a field from register Rx to register Rn. (See Figure 10-3.) 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 10-3. Field Alignment](image)

SHARC Processor Programming Reference 10-73
**Compute Field**

**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
|00000000|00000000|00abcdef|ghijklmn|00000000|          Rn
```

**Status Flags**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SZ</td>
<td>Set if the output operand is 0, otherwise cleared</td>
</tr>
<tr>
<td>SV</td>
<td>Set if any bits are extracted from the left of the 32-bit fixed-point, input field (that is, if len6 + bit6 &gt; 32), otherwise cleared</td>
</tr>
<tr>
<td>SS</td>
<td>Cleared</td>
</tr>
</tbody>
</table>
**Computation Instructions**

\[
\text{Rn} = \text{FEXT Rx BY Ry (SE)} \\
\text{Rn} = \text{FEXT Rx BY <bit6>:<len6> (SE)}
\]

**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
|aaaaaaaaaa|aaaaaaaaa|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
Computation Instructions

**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:

\[ \# \text{ leading sign bits in Rx} - 1 \]

**Status Flags**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SZ</td>
<td>Set if the extracted exponent is 0, otherwise cleared</td>
</tr>
<tr>
<td>SV</td>
<td>Cleared</td>
</tr>
<tr>
<td>SS</td>
<td>Set if the fixed-point operand in Rx is negative (bit 31 is a 1), otherwise cleared</td>
</tr>
</tbody>
</table>
Compute Field

\[ R_n = \text{EXP Rx (EX)} \]

Function

Extracts the exponent of the fixed-point operand in Rx, assuming that the operand is the result of an ALU operation. The exponent is placed in the shf8 field in register Rn. 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 two’s-complement of:

\[ \# \text{leading sign bits in Rx} - 1 \]

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 Rx is equal to 1, otherwise cleared |
**Computation Instructions**

**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**

- **SZ**  Set if the MSB of Rx is 1, otherwise cleared
- **SV**  Set if the result is 32, otherwise cleared
- **SS**  Cleared
**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
Computation Instructions

\[ Rn = \text{FPACK} \, Fx \]

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:

\[
\begin{align*}
135 < \exp^1 & \quad \text{Largest magnitude representation} \\
120 < \exp \leq 135 & \quad \text{Exponent is MSB of source exponent concatenated with the three LSBs of source exponent; the packed fraction is the rounded upper 11 bits of the source fraction} \\
109 < \exp \leq 120 & \quad \text{Exponent=0; packed fraction is the upper bits (source exponent – 110) of the source fraction prefixed by zeros and the “hidden” 1; the packed fraction is rounded} \\
\exp < 110 & \quad \text{Packed word is all zeros}
\end{align*}
\]

\[ ^1 \exp = \text{source exponent sign bit remains the same in all cases} \]

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**

\[ \text{Fn} = \text{FUNPACK Rx} \]

**Function**

Converts the 16-bit floating-point value in Rx to an IEEE 32-bit floating-point value stored in Fx.

**Result**

\[
\begin{align*}
0 < \exp^1 & \leq 15 & \text{Exponent is the three LSBs of the source exponent prefixed by the MSB of the source exponent and four copies of the complement of the MSB; the unpacked fraction is the source fraction with 12 zeros appended} \\
\exp = 0 & & \text{Exponent is } (120 - N) \text{ where } N \text{ is the number of leading zeros in the source fraction; the unpacked fraction is the remainder of the source fraction with zeros appended to pad it and the “hidden” 1 stripped away} \\
1 & & \text{exp = source exponent sign bit remains the same in all cases}
\end{align*}
\]

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
**BITDEP Rx by Ry| <bitlen>**

**Function**

Deposits the bitlen number of bits (specified by Ry or bitlen) in the bit FIFO from Rx. The bits read from Rx are right justified. Write pointer incremented by the number of bit appended. To understand the BITDEP instruction, it is easiest to observe how the data register and bit FIFO behave during instruction execution. If the data register, Rx (40 Bits), contains:

```
 39  32
|--------|
31  23  15  7  0
|--------|----abcd|efghijkl|--|
 \--------/ bitlen bits
```

And, the bit FIFO (64 Bits), before instruction execution contains:

```
 63  55  47  39  32
|qwertyui|opasdfgh|lmn-----|-------|
   ^- BFWRP – Write Pointer
31  23  15  7  0
|--------|--------|--------|--------|
```

Then, after instruction execution, the bit FIFO (64 Bits) contains:

```
 63  55  47  39  32
|qwertyui|opasdfgh|lmnabcde|fghijkl-|
   ^- BFWRP – Write Pointer
31  23  15  7  0
|--------|--------|--------|--------|
```
This operation on the bit FIFO is equivalent to:

1. \( \text{BFF} = \text{BFF OR FDEP Rx BY} <64-(\text{BFWRP}+\text{bitlen})> : <\text{bitlen}> \)
2. \( \text{BFWRP} = \text{BFWRP} + <\text{bitlen}> \)

Note: Do not use the pseudo code above as instruction syntax.

The first operation is similar to the FDEP instruction, but the right and left shifters are modified to be 64-bit shifters.

The second operation provides write pointer update and flag update, which differs from the FDEP instruction.

**Status Flags**

- **SF**
  - SF is set or reset according to the value of write pointer. A value of more than 32 in the lower 6 bits of Ry or immediate field (bitlen) is prohibited, and use of such value sets SV. Attempts to append more bits than the bit FIFO has room for results in an undefined bit FIFO and write pointer. SV is set in that case, otherwise SV is cleared. SZ and SS are cleared.

- **SZ**
  - Cleared

- **SV**
  - Set if any bits are deposited to the left of the 32-bit fixed-point output field (that is, if Ry or bitlen > 32), otherwise cleared

- **SS**
  - Cleared
**Rn = BITEXT Rx| <bitlen>[(NU)]**

**Function**

Extracts bitlen number of bits (specified by Rx or bitlen) from the bit FIFO and places the data in Rn. The bits in Rn are right justified. Decrements write pointer by same number as read bits. Remaining content of the bit FIFO is left-shifted so that it is MSB aligned. The optional modifier NU (no update or query only), returns the requested number of bits as usual but does not modify the bit FIFO or Write pointer. To understand the BITEXT instruction, it is easiest to observe how the data register and bit FIFO behave during instruction execution. If the bit FIFO (64 bits) contains:

```
63 55 47 39
|abcdefg|ijklmn-|--------|
\--------/  ^ - Write Pointer

bitlen bits
31 23 15 7 0
|--------|--------|--------|
```

After instruction execution, the Rn register (40 bits) contains:

```
39 32
|00000000|
31 23 15 7 0
|00000000|abcd|efghijkl|00000000|
```

And the bit FIFO (64 Bits) contains:

```
63 55 47 39 32
|mn-----|--------|--------|
\--------/  ^- BFWRP - Write Pointer
31 23 15 7 0
|--------|--------|--------|
```
Compute Field

This operation on the Bit FIFO is equivalent to:

1. $Rn = \text{FEXT BFF}[63:32] \text{ BY } \langle(32-\text{bitlen})\rangle:\langle\text{bitlen}\rangle$
2. $\text{BFF} = \text{BFF} \ll \text{bitlen}$
3. $\text{BFWRP} = \text{BFWRP} - \text{bitlen}$

Note: Do not use the pseudo code above as instruction syntax.

The first operation is the same as an FEXT instruction operation.

The second operation (bit FIFO 64-bit register with a left shift) and third operation (write pointer update and flag update) are unique to the bit FIFO operation.

Status Flags

A value of more than 32 in the lower 6 bits of $Rx$ or the bitlen immediate field is prohibited and use of such a value sets SV. Attempts to get more bits than those in the bit FIFO results in undefined pointer and bit FIFO. SV is set in that case. SF is set if write pointer is greater than or equal to 32. SZ is set if output is zero, otherwise cleared. SS is cleared. Usage of the NU modifier affects SV, SZ, and SS as said above.

- **SZ**: Set if output is zero, otherwise cleared.
- **SF**: SF will reflect the un-updated Write pointer status.
- **SV**: Set if an attempt is made to extract more bits than those in bit FIFO, otherwise cleared.
- **SS**: Cleared.
**Rn = BFFWRP**

**Function**

Transfers write pointer value to Rn.

**Examples**

For bit FIFO examples, see the BITDEP instruction “BITDEP Rx by Ry<bitlen>” on page 10-83.

**Status Flags**

Flags SV, SZ, SS are cleared.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>SZ</td>
<td>Cleared</td>
</tr>
<tr>
<td>SV</td>
<td>Cleared</td>
</tr>
<tr>
<td>SS</td>
<td>Cleared</td>
</tr>
<tr>
<td>SF</td>
<td>0</td>
</tr>
</tbody>
</table>
Compute Field

**BFFWRP = Rn|<value7>**

Function

Updates write pointer from Rn or the immediate 7 bit value specified. Only 7 least significant bits of Rn are written.

The maximum permissible value to be written into BFFWRP is 64.

Examples

For bit FIFO examples, see the BITDEP instruction “BITDEP Rx by Ry|<bitlen>” on page 10-83.

Status Flags

SF is set if updated BFFWRP is greater than or equal to 32, cleared otherwise. SV is set if the written value is greater than 64 else SV is cleared. Flags SZ, SS are cleared.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>SZ</td>
<td>Cleared</td>
</tr>
<tr>
<td>SF</td>
<td>Set if updated BFFWRP ≥ 32, otherwise cleared</td>
</tr>
<tr>
<td>SV</td>
<td>Set if written value is &gt; 64, otherwise cleared</td>
</tr>
<tr>
<td>SS</td>
<td>Cleared</td>
</tr>
</tbody>
</table>
Multifunction Computations

Multifunction instructions are parallelized single ALU and Multiplier instructions. For functional description and status flags and for parallel Multiplier and ALU instructions input operand constraints see “ALU Operations” on page 10-2 and “Multiplier Operations” on page 10-47. This section lists all possible instruction syntax options.

**Fixed-Point ALU (dual Add and Subtract)**

\[
Ra = Rx + Ry, \quad Rs = Rx - Ry
\]

**Floating-Point ALU (dual Add and Subtract)**

\[
Fa = Fx + Fy, \quad Fs = Fx - Fy
\]

**Fixed-Point Multiplier and ALU**

\[
\begin{align*}
Rm &= R3\cdot0 \times R7\cdot4 \text{ (SSFR)}, \quad Ra = R11\cdot8 + R15\cdot12 \\
Rm &= R3\cdot0 \times R7\cdot4 \text{ (SSFR)}, \quad Ra = R11\cdot8 - R15\cdot12 \\
Rm &= R3\cdot0 \times R7\cdot4 \text{ (SSFR)}, \quad Ra = (R11\cdot8 + R15\cdot12)/2 \\
MRF &= MRF + R3\cdot0 \times R7\cdot4 \text{ (SSF)}, \quad Ra = R11\cdot8 + R15\cdot12 \\
MRF &= MRF + R3\cdot0 \times R7\cdot4 \text{ (SSF)}, \quad Ra = R11\cdot8 - R15\cdot12 \\
MRF &= MRF + R3\cdot0 \times R7\cdot4 \text{ (SSF)}, \quad Ra = (R11\cdot8 + R15\cdot12)/2 \\
Rm &= MRF + R3\cdot0 \times R7\cdot4 \text{ (SSFR)}, \quad Ra = R11\cdot8 + R15\cdot12 \\
Rm &= MRF + R3\cdot0 \times R7\cdot4 \text{ (SSFR)}, \quad Ra = R11\cdot8 - R15\cdot12 \\
Rm &= MRF + R3\cdot0 \times R7\cdot4 \text{ (SSFR)}, \quad Ra = (R11\cdot8 + R15\cdot12)/2 \\
MRF &= MRF - R3\cdot0 \times R7\cdot4 \text{ (SSF)}, \quad Ra = R11\cdot8 + R15\cdot12 \\
MRF &= MRF - R3\cdot0 \times R7\cdot4 \text{ (SSF)}, \quad Ra = R11\cdot8 - R15\cdot12 \\
MRF &= MRF - R3\cdot0 \times R7\cdot4 \text{ (SSF)}, \quad Ra = (R11\cdot8 + R15\cdot12)/2 \\
Rm &= MRF - R3\cdot0 \times R7\cdot4 \text{ (SSFR)}, \quad Ra = R11\cdot8 + R15\cdot12 \\
Rm &= MRF - R3\cdot0 \times R7\cdot4 \text{ (SSFR)}, \quad Ra = R11\cdot8 - R15\cdot12 \\
Rm &= MRF - R3\cdot0 \times R7\cdot4 \text{ (SSFR)}, \quad Ra = (R11\cdot8 + R15\cdot12)/2
\end{align*}
\]

See also Table 3-11 on page 3-31.
**Compute Field**

**Floating-Point Multiplier and ALU**

Fm = F3-0 * F7-4, Fa = F11-8 + F15-12  
Fm = F3-0 * F7-4, Fa = F11-8 - F15-12  
Fm = F3-0 * F7-4, Fa = FLOAT R11-8 by R15-12  
Fm = F3-0 * F7-4, Fa = FIX F11-8 by R15-12  
Fm = F3-0 * F7-4, Fa = (F11-8 + F15-12)/2  
Fm = F3-0 * F7-4, Fa = ABS F11-8  
Fm = F3-0 * F7-4, Fa = MAX (F11-8, F15-12)  
Fm = F3-0 * F7-4, Fa = MIN (F11-8, F15-12)

See also Table 3-12 on page 3-32.

**Fixed-Point Multiplier and ALU (dual Add and Subtract)**

Rm = R3-0 * R7-4 (SSFR), Ra = R11-8 + R15-12, Rs = R11-8 - R15-12

**Floating Point Multiplier and ALU (dual Add and Subtract)**

Fm = F3-0 * F7-4, Fa = F11-8 + F15-12, Fs = F11-8 - F15-12
11 INSTRUCTION OPCODES

This chapter is divided into two sections. The first section, “Instruction Opcode Types” on page 11-1, lists the various instruction type opcodes and their ISA or VISA operation. The instruction types linked into normal word space are valid ISA opcodes and if linked into short word space they become valid VISA opcodes (valid for the ADSP-214xx processors).

The second section, “Computation Type Opcodes” on page 11-49 describes all of the available compute type operations. The logical function of the instruction is described in Chapter 9, Instruction Set Types.

Instruction Opcode Types

Table 11-1 shows acronyms for instruction type opcodes

Table 11-1. Opcode Acronyms ((ISA/VISA)

<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>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>
</tbody>
</table>
### Instruction Opcode Types

#### Table 11-1. Opcode Acronyms ((ISA/VISA) (Cont’d))

<table>
<thead>
<tr>
<th>Bit/Field</th>
<th>Description</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDREG</td>
<td>Complementary data Register file locations 0–15</td>
<td></td>
</tr>
<tr>
<td>COMPUTE</td>
<td>Compute operation field (see &quot;Compute Field&quot; on page 10-1)</td>
<td></td>
</tr>
<tr>
<td>COND</td>
<td>IF condition codes</td>
<td>0–31 (see Table 11-16 on page 11-69)</td>
</tr>
<tr>
<td>CI</td>
<td>Clear interrupt code</td>
<td>0 Do not clear current interrupt 1 Clear current interrupt</td>
</tr>
<tr>
<td>CU</td>
<td>Computation unit select codes</td>
<td>00 ALU 01 Multiplier 10 Shifter</td>
</tr>
<tr>
<td>DATA</td>
<td>Immediate data field</td>
<td></td>
</tr>
<tr>
<td>DMD</td>
<td>DAG1 access direction</td>
<td>0 Read 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>Data Register file locations</td>
<td>0–15</td>
</tr>
<tr>
<td>E</td>
<td>ELSE clause code</td>
<td>0 No ELSE clause 1 ELSE clause</td>
</tr>
<tr>
<td>FC</td>
<td>Flush cache code</td>
<td>0 No cache flush 1 Cache flush</td>
</tr>
<tr>
<td>G</td>
<td>DAG select</td>
<td>0 DAG1 1 DAG2</td>
</tr>
<tr>
<td>J</td>
<td>Jump type</td>
<td>0 Non-delayed 1 Delayed</td>
</tr>
<tr>
<td>L</td>
<td>Long word memory address</td>
<td>0 Access size based on memory map 1 Long word (64-bit) access size</td>
</tr>
<tr>
<td>LPO</td>
<td>Loop stack pop code</td>
<td>0 No stack pop 1 Stack pop</td>
</tr>
<tr>
<td>LPU</td>
<td>Loop stack push code</td>
<td>0 No stack push 1 Stack push</td>
</tr>
</tbody>
</table>
### Instruction Opcodes

Table 11-1. Opcode Acronyms ((ISA/VISA) (Cont’d))

<table>
<thead>
<tr>
<th>Bit/Field</th>
<th>Description</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR</td>
<td>Loop reentry code</td>
<td>0: No loop reentry; 1: Loop reentry</td>
</tr>
<tr>
<td>M</td>
<td>Multiply result register</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See “MRxF/B = Rn Rn = MRxF/B” on page 11-56</td>
<td></td>
</tr>
<tr>
<td>PMD</td>
<td>DAG2 access direction</td>
<td>0: Read; 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; 1: Stack pop</td>
</tr>
<tr>
<td>PPU</td>
<td>PC stack push code</td>
<td>0: No stack push; 1: Stack push</td>
</tr>
<tr>
<td>RELADDR</td>
<td>PC-relative address field</td>
<td></td>
</tr>
<tr>
<td>SHIFTIM</td>
<td>Shift Immediate field (valid for Type 6a Instruction)</td>
<td></td>
</tr>
<tr>
<td>SPO</td>
<td>Status stack pop code</td>
<td>0: No stack pop; 1: Stack pop</td>
</tr>
<tr>
<td>SPU</td>
<td>Status stack push code</td>
<td>0: No stack push; 1: Stack push</td>
</tr>
<tr>
<td>SREG</td>
<td>System register code</td>
<td>0–15 (see “Register Opcodes” on page 11-66)</td>
</tr>
<tr>
<td>TERM</td>
<td>Termination condition codes</td>
<td>0–31 (see Table 11-16 on page 11-69)</td>
</tr>
<tr>
<td>U</td>
<td>Update, index (I) register</td>
<td>0: Pre-modify, no update; 1: Post-modify with update</td>
</tr>
<tr>
<td>UREG</td>
<td>Universal register code</td>
<td>0–256 (see “Register Opcodes” on page 11-66)</td>
</tr>
</tbody>
</table>

The letter after the instruction in the next sections denotes the instruction size as follows: a = 48-bit, b = 32-bit, c = 16-bit.
Group I - Conditional Compute and Move or Modify Instructions

Type 1a ISA/VISA (compute + mem dual data move)

<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 the DAGs</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>

| 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 |
|-------------------------------|-------------------------------|
| 001 DMD DMI DMM PMD DM DREG PMI PMM PM DREG |
### Instruction Opcodes

#### Type 1b VISA (mem dual data move)

<table>
<thead>
<tr>
<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>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</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></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| PMI | PMM | PM DREG | PM DREG | 0111111 |

**Bits Description**

- **DMD, PMD**: Select the access types (read or write)
- **DM DREG, PM DREG**: Specify register file location
- **DMI, PMI**: Specify I registers for the DAGs
- **DMM, PMM**: Specify M registers used to update the I registers
**Type 2a ISA/VISA (cond + compute)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COND</td>
<td>Specifies the test condition; if omitted, COND is TRUE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COND</td>
<td>COND</td>
</tr>
</tbody>
</table>
**Instruction Opcodes**

**Type 2b VISA (compute)**

<table>
<thead>
<tr>
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00001</td>
<td>1</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 2c VISA (short compute)**

<table>
<thead>
<tr>
<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>1100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OP</td>
<td></td>
<td>Rn</td>
<td></td>
<td></td>
<td>Rx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Even though the Type 2c instruction appears syntactically similar to the Type 2b instruction, the encoding is different. A limited set of compute operations (OP field) are supported.

<table>
<thead>
<tr>
<th>OP</th>
<th>Operation</th>
<th>OP</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Rn = Rn + Rx</td>
<td>1000</td>
<td>Fn = Fn + Fx</td>
</tr>
<tr>
<td>0001</td>
<td>Rn = Rn – Rx</td>
<td>1001</td>
<td>Fn = Fn – Fx</td>
</tr>
<tr>
<td>0010</td>
<td>Rn = PASS Rx</td>
<td>1010</td>
<td>Fn = FLOAT Rx</td>
</tr>
<tr>
<td>0011</td>
<td>COMP (Rn, Rx)</td>
<td>1011</td>
<td>COMP (Fn, Fx)</td>
</tr>
<tr>
<td>0100</td>
<td>Rn = NOT Rx</td>
<td>1100</td>
<td>Rn = Rn AND Rx</td>
</tr>
<tr>
<td>0101</td>
<td>Rn = Rx + 1</td>
<td>1101</td>
<td>Rn = Rn OR Rx</td>
</tr>
<tr>
<td>0110</td>
<td>Rn = Rx – 1</td>
<td>1110</td>
<td>Rn = Rn XOR Rx</td>
</tr>
<tr>
<td>0111</td>
<td>Rn = Rn * Rx (SSI)</td>
<td>1111</td>
<td>Fn = Fn * Fx</td>
</tr>
</tbody>
</table>
### Type 3a ISA/VISA (cond + comp + mem data move)

<table>
<thead>
<tr>
<th>Bit Positions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>47-30</td>
<td>Instruction Opcodes</td>
</tr>
<tr>
<td>29-21</td>
<td>COMPUTE</td>
</tr>
<tr>
<td>20-12</td>
<td>COND</td>
</tr>
<tr>
<td>11-4</td>
<td>D</td>
</tr>
<tr>
<td>3</td>
<td>G</td>
</tr>
<tr>
<td>2</td>
<td>L</td>
</tr>
<tr>
<td>1</td>
<td>U</td>
</tr>
<tr>
<td>0</td>
<td>UREG</td>
</tr>
<tr>
<td>22-21</td>
<td>I</td>
</tr>
<tr>
<td>19-18</td>
<td>M</td>
</tr>
<tr>
<td>16-15</td>
<td>COND</td>
</tr>
<tr>
<td>14</td>
<td>G</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
</tr>
<tr>
<td>12</td>
<td>L</td>
</tr>
<tr>
<td>11</td>
<td>U</td>
</tr>
<tr>
<td>10</td>
<td>UREG</td>
</tr>
<tr>
<td>9</td>
<td>I</td>
</tr>
<tr>
<td>8</td>
<td>M</td>
</tr>
<tr>
<td>7</td>
<td>COND</td>
</tr>
<tr>
<td>6</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td>L</td>
</tr>
<tr>
<td>4</td>
<td>U</td>
</tr>
<tr>
<td>3</td>
<td>UREG</td>
</tr>
<tr>
<td>2</td>
<td>I</td>
</tr>
<tr>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>0</td>
<td>COND</td>
</tr>
</tbody>
</table>

**Bits Description**

- **COND**: Specifies the test condition; if omitted, COND is TRUE
- **D**: Selects the access Type (read or write)
- **G**: Selects the DAG
- **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 3b VISA (cond + mem data move)

<table>
<thead>
<tr>
<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>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
</tr>
</thead>
<tbody>
<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>010</td>
</tr>
<tr>
<td>U</td>
<td>I</td>
<td>M</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>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<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>G</td>
<td>D</td>
<td>L</td>
<td>UREG</td>
<td>0111111</td>
<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>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 the DAG</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 universal register</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>U</td>
<td>Selects either update (post-modify) or no update (pre-modify)</td>
</tr>
</tbody>
</table>
Type 3c VISA (mem data move)

<table>
<thead>
<tr>
<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>1001</td>
<td>I</td>
<td>M</td>
<td>D</td>
<td>1</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>

Bits Description

- **D**: DAG1 selects the access Type (read or write)
- **DREG**: Specifies the data register
- **I**: Specifies the I register which is post-modified by the M register
- **M**: Specifies the M register
**Type 4a ISA/VISA (cond + comp + mem data move with 6-bit immediate modifier)**

<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 the DAG</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>
Instruction Opcodes

Type 4b VISA (cond +mem data move with 6-bit immediate modifier)

<table>
<thead>
<tr>
<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>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>011</td>
<td>0</td>
<td>I</td>
<td>G</td>
<td>D</td>
<td>U</td>
<td>COND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| DATA | DREG | 0111111 |

Bits Description

- **COND**: Specifies the test condition; if omitted, COND is TRUE
- **D**: Selects the access Type (read or write)
- **G**: Selects the DAG
- **DREG**: Specifies the register file location
- **I**: Specifies the I register
- **DATA**: Specifies a 6-bit, twos-complement modify value
- **U**: Selects either pre-modify without update or post-modify with update
Type 5a ISA/VISA (cond + comp + reg data move)

Ureg = Ureg transfer

| 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| 1  | 0  | SRC UREG | COND | SU | DEST UREG |

Dreg <-> C Dreg swap

| 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 26 | 25 | 24 | 23 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 011| 1  | 1  | CDREG | COND | | DREG |

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

<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>CDREG</td>
<td>Identifies the PEy data registers for swap (must appear to right of swap operator)</td>
</tr>
<tr>
<td>DREG</td>
<td>Identifies the PEx data register for swap (must appear to left of swap operator)</td>
</tr>
</tbody>
</table>
### Instruction Opcodes

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 5b VISA (cond + reg data move)**

**Ureg = Ureg move**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>ConD</td>
</tr>
<tr>
<td>30</td>
<td>SRC UREG</td>
</tr>
<tr>
<td>29</td>
<td>SU</td>
</tr>
<tr>
<td>28</td>
<td>DEST UREG</td>
</tr>
<tr>
<td>27</td>
<td>UREG</td>
</tr>
<tr>
<td>26</td>
<td>COND</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>DREG</td>
</tr>
<tr>
<td>15</td>
<td>01111111</td>
</tr>
</tbody>
</table>

**Dreg <-> CDreg swap**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>ConD</td>
</tr>
<tr>
<td>30</td>
<td>CDREG</td>
</tr>
<tr>
<td>29</td>
<td>SU</td>
</tr>
<tr>
<td>28</td>
<td>DEST UREG</td>
</tr>
<tr>
<td>27</td>
<td>UREG</td>
</tr>
<tr>
<td>26</td>
<td>COND</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>DREG</td>
</tr>
<tr>
<td>15</td>
<td>01111111</td>
</tr>
</tbody>
</table>

**Bits Description**

<table>
<thead>
<tr>
<th>Bit</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>CDREG</td>
<td>Identifies the PEy data registers for swap (must appear to right of swap operator)</td>
</tr>
<tr>
<td>DREG</td>
<td>Identifies the PEx data register for swap (must appear to left of swap operator)</td>
</tr>
</tbody>
</table>
### Instruction Opcodes

#### Type 6 ISA/VISA (cond + shift imm + mem data move)

**with mem data move**

<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>100</td>
<td>0</td>
<td>I</td>
<td>M</td>
<td>COND</td>
<td>G</td>
<td>D</td>
<td>DATAEX</td>
<td>DM</td>
<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>
</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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | SHIFTIM | DATA | RN | RX |

**without mem data move**

<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>00010</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>
<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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | SHIFTIM | DATA | RN | RX |

**Bits** | **Description**
--- | ---
COND | Specifies the test condition; if omitted, COND is TRUE
SHIFTIM | Specifies the shift immediate operation. For more information, see “Shifter Operations” on page 10-56
DATA | Specifies an 8-bit immediate shift value.
<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATAEX</td>
<td>For two 6-bit immediate Y input 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 (bit6), and the six MSBs are the length value (len6)</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 the DAG</td>
</tr>
<tr>
<td>DM DREG</td>
<td>Specifies the register file location for DAG1</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>
<tr>
<td>Rn</td>
<td>Shift immediate result data register location</td>
</tr>
<tr>
<td>Rx</td>
<td>Shift immediate X-input data register location</td>
</tr>
</tbody>
</table>
## Instruction Opcodes

### Type 7a ISA/VISA (cond + comp + index modify)

<table>
<thead>
<tr>
<th>Bit Positions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>47–45</td>
<td>G COND Is M Id</td>
</tr>
<tr>
<td>22–20</td>
<td>Is</td>
</tr>
<tr>
<td>19–11</td>
<td>M</td>
</tr>
<tr>
<td>10–0</td>
<td>Id⊕Is</td>
</tr>
</tbody>
</table>

**Bits**

- **COND**: Specifies the test condition; if omitted, COND is TRUE
- **G**: Selects the DAG
- **Is**: Specifies the source I register
- **Id⊕Is**: Specifies destination I register indirectly. Destination I register is derived by performing bitwise exclusive OR between Is and these bits.
- **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
**Type 7b VISA (cond + index 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>G</td>
<td>Selects the DAG</td>
</tr>
<tr>
<td>Is</td>
<td>Specifies the source I register</td>
</tr>
<tr>
<td>M</td>
<td>Specifies the M register</td>
</tr>
<tr>
<td>Id⊕Is</td>
<td>Specifies destination I register indirectly. Destination I register is derived by performing bitwise exclusive OR between Is and these bits.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<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>
<th>22</th>
<th>21</th>
<th>19</th>
<th>18</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td></td>
<td></td>
<td>00100</td>
<td></td>
<td>G</td>
<td>COND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<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>Is</td>
<td>M</td>
<td>Id⊕Is</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>011111</td>
</tr>
</tbody>
</table>

11-20 SHARC Processor Programming Reference
**Instruction Opcodes**

## Group II - Conditional Program Flow Control Instructions

### Type 8a ISA/VISA (cond + branch)

**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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>J</td>
<td>CI</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>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>

**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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>J</td>
<td>CI</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>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>Bits</td>
<td>Description</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>
</tr>
<tr>
<td>COND</td>
<td>Specifies the test condition; if omitted, COND is TRUE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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</td>
<td>Selects the branch type, jump or call; for calls, A and CI are ignored</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>J</td>
<td>Determines whether the branch is delayed or non-delayed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>Specifies a 24-bit program memory address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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</td>
<td>Activates loop abort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>CI</td>
<td>Activates clear interrupt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>Holds a 24-bit, twos-complement value that is added to the current PC value to generate the branch address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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 9a ISA/VISA (cond + branch + comp else/comp)

#### with indirect branch

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>47-43</td>
<td>COND</td>
</tr>
<tr>
<td>42</td>
<td>B</td>
</tr>
<tr>
<td>41</td>
<td>A</td>
</tr>
<tr>
<td>40-35</td>
<td>PMI/PMM</td>
</tr>
<tr>
<td>34-30</td>
<td>J/E/CI</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>22-19</td>
<td>COMPUTE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>47-43</td>
<td>COND</td>
</tr>
<tr>
<td>42</td>
<td>B</td>
</tr>
<tr>
<td>41</td>
<td>A</td>
</tr>
<tr>
<td>40-31</td>
<td>RELADDR</td>
</tr>
<tr>
<td>30-26</td>
<td>J/E/CI</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>22-19</td>
<td>COMPUTE</td>
</tr>
</tbody>
</table>

### Bits Description

- **COND**: Specifies the test condition; if omitted, COND is true
- **E**: Specifies whether or not an ELSE clause is used
- **B**: Selects the branch type, jump or call; for calls, A and CI are ignored
- **J**: Selects delayed or non-delayed branch
- **A**: Activates loop abort
<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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, twos-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 9b VISA (cond + branch)

with indirect branch

<table>
<thead>
<tr>
<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>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>01000</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>
</tr>
</tbody>
</table>

| 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| PMI | PMM | J | CI | 0111111 |

with PC-relative branch

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 000 | 01001 | B | A | COND |

| 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| RELADDR | J | CI | 0111111 |

<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>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>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RELADDR</td>
<td>Holds a 6-bit, two's-complement value that is added to the current PC value</td>
</tr>
<tr>
<td></td>
<td>to generate the branch address</td>
</tr>
<tr>
<td>PMI</td>
<td>Specifies the I register for indirect branches; the I register is pre-</td>
</tr>
<tr>
<td></td>
<td>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>
### Instruction Opcodes

#### Type 10 ISA (cond + branch + else comp + mem data move)

**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 |  1  |  1  | D  | DMI | DMM | COND | PMI | PMM | DREG |

**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 |  1  |  1  | D  | DMI | DMM | COND | RELADDR | DREG |

**Bits Description**

<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 DAG1 access direction (read or write)</td>
</tr>
<tr>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>------------------------------------------------------------------------------</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, two's-complement value that is added to the current PC value to generate the branch address</td>
</tr>
</tbody>
</table>
## Instruction Opcodes

### Type 11a ISA/VISA (cond + branch return + comp/else comp)

#### branch 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>

#### branch 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

<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>J</td>
<td>Determines whether the return is delayed or non-delayed</td>
</tr>
<tr>
<td>E</td>
<td>Specifies whether an ELSE clause is used</td>
</tr>
<tr>
<td>COMPUTE</td>
<td>Defines the compute operation to be performed; if omitted, this is a NOP</td>
</tr>
<tr>
<td>LR</td>
<td>Specifies whether or not the loop reentry modifier is specified</td>
</tr>
</tbody>
</table>
Group II - Conditional Program Flow Control Instructions

Type 11c VISA (cond + branch return)

branch return from subroutine

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>000</td>
<td>01010</td>
</tr>
</tbody>
</table>

branch return from interrupt

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>000</td>
<td>01011</td>
</tr>
</tbody>
</table>

<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>J</td>
<td>Determines whether the return is delayed or non-delayed</td>
</tr>
<tr>
<td>LR</td>
<td>Specifies whether or not the loop reentry modifier is specified</td>
</tr>
</tbody>
</table>
# Instruction Opcodes

## Type 12a ISA/VISA (do until loop counter expired)

### 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 |

### with Ureg 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 | 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>
**Group III - Immediate Data Move Instructions**

**Type 13a ISA/VISA (do until termination)**

<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 accepts an absolute address as well and converts the absolute address to the equivalent relative address for coding</td>
</tr>
<tr>
<td>TERM</td>
<td>Specifies the termination condition</td>
</tr>
</tbody>
</table>
**Instruction Opcodes**

### Type 14a ISA/VISA (mem data move)

<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>100</td>
<td>G</td>
<td>D</td>
<td>L</td>
<td>UREG</td>
<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>
</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>
</table>

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 DAG</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>
### Type 15a ISA/VISA (<data32> move)

| 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 101 | G | I | D | L | UREG | DATA (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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| DATA (lower 24 bits) |

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>Selects the DAG</td>
</tr>
<tr>
<td>I</td>
<td>Selects the I register</td>
</tr>
<tr>
<td>D</td>
<td>Selects the access Type (read or write)</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>
**Instruction OpCodes**

### Type 15b VISA (<data7> move)

<table>
<thead>
<tr>
<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>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>I</td>
<td>D</td>
<td>L</td>
<td>G</td>
<td>010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| UREG | DATA |

**Bits** | **Description**
--- | ---
I | Selects the I register
D | Selects the access Type (read or write)
G | Selects the DAG
L | Forces a long word (LW) access when address is in normal word address range
UREG | Specifies the number of a universal register
DATA | Specifies the immediate modify value for the I register
### Group III - Immediate Data Move Instructions

#### Type 16a ISA/VISA (<data32> move)

| 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 100 | 1  | I  | M  | G  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     | DATA (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  | DATA (lower 24 bits) |

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Selects the I register</td>
</tr>
<tr>
<td>M</td>
<td>Selects the M register</td>
</tr>
<tr>
<td>G</td>
<td>Selects the memory (data or program)</td>
</tr>
<tr>
<td>DATA</td>
<td>Specifies the 32-bit immediate data</td>
</tr>
</tbody>
</table>
Instruction Opcodes

Type 16b VISA (<data16> move)

<table>
<thead>
<tr>
<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>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>I</td>
<td>M</td>
<td>G</td>
<td>001</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>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>
</tr>
</tbody>
</table>

Bits Description

- **I**: Selects the I register
- **M**: Selects the M register
- **G**: Selects the DAG
- **DATA**: Specifies the 16-bit immediate data
Group III - Immediate Data Move Instructions

Type 17a ISA/VISA (<data32> move)

| 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 | 01111 | 0 | UREG | DATA | (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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| DATA | (lower 24 bits) |

<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>
Instruction Opcodes

**Type 17b VISA (<data16> move)**

<table>
<thead>
<tr>
<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>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>01111</td>
<td>1</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>
</tbody>
</table>

<table>
<thead>
<tr>
<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>
</tr>
</tbody>
</table>

<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>
### Type 18a ISA/VISA (register bit manipulation)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>10100</td>
</tr>
<tr>
<td>BOP</td>
<td></td>
</tr>
<tr>
<td>SREG</td>
<td></td>
</tr>
<tr>
<td>DATA (upper 8 bits)</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>22</td>
</tr>
<tr>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>19</td>
<td>18</td>
</tr>
<tr>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>DATA (lower 24 bits)</td>
<td></td>
</tr>
</tbody>
</table>

**Bits**

- **BOP**: Selects one of the five bit operations
- **SREG**: Specifies the system register
- **DATA**: Specifies the data value
## Type 19a ISA/VISA (index modify/bitrev)

### with modify

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>If this bit is cleared (=0) the modify instruction is selected, if set (=1) bitrev instruction</td>
</tr>
<tr>
<td>G</td>
<td>Selects the DAG</td>
</tr>
<tr>
<td>Is</td>
<td>Selects the source I register</td>
</tr>
<tr>
<td>Id⊕Is</td>
<td>Specifies destination I register indirectly. Destination I register is derived by performing bitwise exclusive OR between Is and these bits.</td>
</tr>
<tr>
<td>DATA</td>
<td>Specifies the immediate modifier</td>
</tr>
</tbody>
</table>

### with bit-reverse

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>If this bit is cleared (=0) the modify instruction is selected, if set (=1) bitrev instruction</td>
</tr>
<tr>
<td>G</td>
<td>Selects the DAG</td>
</tr>
<tr>
<td>Is</td>
<td>Selects the source I register</td>
</tr>
<tr>
<td>Id⊕Is</td>
<td>Specifies destination I register indirectly. Destination I register is derived by performing bitwise exclusive OR between Is and these bits.</td>
</tr>
<tr>
<td>DATA</td>
<td>Specifies the immediate modifier</td>
</tr>
</tbody>
</table>
Group IV - Miscellaneous Instructions

Type 20a ISA/VISA (push/pop stack)

<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>000</td>
<td>10111</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>S</td>
<td>P</td>
<td>U</td>
<td>S</td>
<td>P</td>
<td>U</td>
<td>P</td>
<td>P</td>
<td>O</td>
<td>F</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<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>

<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>
### Type 21a ISA/VISA (nop)

| 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 | 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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
### Group IV - Miscellaneous Instructions

#### Type 21c VISA (nop)

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

11-44 SHARC Processor Programming Reference
## Instruction Opcodes

### Type 22a ISA/VISA (idle/emuidle)

<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>00000</td>
<td>I</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>

| 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>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Idle (must be set)</td>
</tr>
<tr>
<td>E</td>
<td>Emulator idle (must be set)</td>
</tr>
</tbody>
</table>

#### idle
- I: 1
- E: 0

#### emuidle
- I: 1
- E: 1
Group IV - Miscellaneous Instructions

Type 22c VISA (idle/emuidle)

<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>
</tr>
</thead>
<tbody>
<tr>
<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>000</td>
<td>00000</td>
<td>I</td>
<td>E</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>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Idle (must be set)</td>
</tr>
<tr>
<td>E</td>
<td>Emulator idle (must be set)</td>
</tr>
</tbody>
</table>

idle

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
</tr>
</tbody>
</table>

emuidle

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
</tr>
</tbody>
</table>
### Type 25a ISA/VISA (cjump/rframe)

**cjump/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>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>

<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>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>

**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>

<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>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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDR</td>
<td>Specifies a 24-bit program memory address for &quot;function&quot;</td>
</tr>
<tr>
<td>RELADDR</td>
<td>Specifies a 24-bit, two's-complement value added to the current PC value to generate the branch address</td>
</tr>
</tbody>
</table>
Group IV - Miscellaneous Instructions

**RFRAME**

| 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0001 | 1001 | 0000 | 0000 | 0000 | 0000 |       |

| 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |       |

**Type 25c VISA (RFRAME)**

| 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0001 | 1001 | 0000 | 0001 |
Computation Type Opcodes

The 23-bit compute field is a mini instruction within the ADSP-21xxx 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.

In single-function operations (other than type 6), the compute field of a single-function operation is made up of the following bit fields. For type 6 instructions (immediate shift) see “Type 6 ISA/VISA (cond + shift imm + mem data move)” on page 9-25.

<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 11-3 on page 11-51, Table 11-4 on page 11-52, Table 11-5 on page 11-53, Table 11-11 on page 11-59.

Table 11-2. Compute Instruction Selection Table

<table>
<thead>
<tr>
<th>Bits 22–20</th>
<th>Bits 19–12</th>
<th>Computation Type</th>
<th>Data Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Computation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>0xxxxxxx</td>
<td>ALU</td>
<td>Fixed</td>
</tr>
<tr>
<td>000</td>
<td>1xxxxxxx</td>
<td>ALU</td>
<td>Float</td>
</tr>
<tr>
<td>001</td>
<td>xxxxxxxx</td>
<td>Multiply</td>
<td>Fixed</td>
</tr>
<tr>
<td>001</td>
<td>00110000</td>
<td>Multiply</td>
<td>Float</td>
</tr>
<tr>
<td>010</td>
<td>xxxxxxxx</td>
<td>Shifter</td>
<td>Fixed</td>
</tr>
<tr>
<td>0xx</td>
<td>xxxx</td>
<td>Shift Immediate</td>
<td>Fixed</td>
</tr>
<tr>
<td>Multiple Computation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>0111</td>
<td>Dual ALU (+/-)</td>
<td>Fixed</td>
</tr>
<tr>
<td>000</td>
<td>1111</td>
<td>Dual ALU (+/-)</td>
<td>Float</td>
</tr>
<tr>
<td>10x</td>
<td>xxxx</td>
<td>MUL/ALU</td>
<td>Fixed</td>
</tr>
<tr>
<td>101</td>
<td>1xx</td>
<td>MUL/ALU</td>
<td>Float</td>
</tr>
<tr>
<td>110</td>
<td></td>
<td>MUL/dual ALU (+/-)</td>
<td>Fixed</td>
</tr>
<tr>
<td>111</td>
<td></td>
<td>MUL/dual ALU (+/-)</td>
<td>Float</td>
</tr>
<tr>
<td>Data Move</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>000</td>
<td>MRx data move</td>
<td>Fixed</td>
</tr>
</tbody>
</table>
### ALU Opcodes

Table 11-3 and Table 11-4 summarize the syntax and opcodes for the fixed-point and floating-point ALU operations, respectively.

**Table 11-3. Fixed-Point ALU Operations**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn = Rx + Ry</td>
<td>0000 0001</td>
</tr>
<tr>
<td>Rn = Rx – Ry</td>
<td>0000 0010</td>
</tr>
<tr>
<td>Rn = Rx + Ry + CI</td>
<td>0000 0101</td>
</tr>
<tr>
<td>Rn = Rx – Ry + CI – 1</td>
<td>0000 0110</td>
</tr>
<tr>
<td>Rn = (Rx + Ry)/2</td>
<td>0000 1001</td>
</tr>
<tr>
<td>COMP(Rx, Ry)</td>
<td>0000 1010</td>
</tr>
<tr>
<td>COMPU(Rx, Ry)</td>
<td>0000 1011</td>
</tr>
<tr>
<td>Rn = Rx + CI</td>
<td>0010 0011</td>
</tr>
<tr>
<td>Rn = Rx + CI – 1</td>
<td>0010 0110</td>
</tr>
<tr>
<td>Rn = Rx + 1</td>
<td>0010 1001</td>
</tr>
<tr>
<td>Rn = Rx – 1</td>
<td>0010 1010</td>
</tr>
<tr>
<td>Rn = – Rx</td>
<td>0010 0010</td>
</tr>
<tr>
<td>Rn = ABS Rx</td>
<td>0011 0000</td>
</tr>
<tr>
<td>Rn = PASS Rx</td>
<td>0010 0001</td>
</tr>
<tr>
<td>Rn = Rx AND Ry</td>
<td>0100 0000</td>
</tr>
<tr>
<td>Rn = Rx OR Ry</td>
<td>0100 0001</td>
</tr>
<tr>
<td>Rn = Rx XOR Ry</td>
<td>0100 0010</td>
</tr>
<tr>
<td>Rn = NOT Rx</td>
<td>0100 0011</td>
</tr>
<tr>
<td>Rn = MIN(Rx, Ry)</td>
<td>0110 0001</td>
</tr>
<tr>
<td>Rn = MAX(Rx, Ry)</td>
<td>0110 0010</td>
</tr>
<tr>
<td>Rn = CLIP Rx BY Ry</td>
<td>0110 0011</td>
</tr>
</tbody>
</table>
## Computation Type Opcodes

### Table 11-4. Floating-Point ALU Operations

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Fn = Fx + Fy )</td>
<td>1000 0001</td>
</tr>
<tr>
<td>( Fn = Fx - Fy )</td>
<td>1000 0010</td>
</tr>
<tr>
<td>( Fn = \text{ABS} (Fx + Fy) )</td>
<td>1001 0001</td>
</tr>
<tr>
<td>( Fn = \text{ABS} (Fx - Fy) )</td>
<td>1001 0010</td>
</tr>
<tr>
<td>( Fn = (Fx + Fy)/2 )</td>
<td>1000 1001</td>
</tr>
<tr>
<td>COMP(Fx, Fy)</td>
<td>1000 1010</td>
</tr>
<tr>
<td>( Fn = -Fx )</td>
<td>1010 0010</td>
</tr>
<tr>
<td>( Fn = \text{ABS} Fx )</td>
<td>1011 0000</td>
</tr>
<tr>
<td>( Fn = \text{PASS} Fx )</td>
<td>1010 0001</td>
</tr>
<tr>
<td>( Fn = \text{RND} Fx )</td>
<td>1010 0101</td>
</tr>
<tr>
<td>( Fn = \text{SCALB} Fx \text{ BY Ry} )</td>
<td>1011 1101</td>
</tr>
<tr>
<td>( Rn = \text{MANT} Fx )</td>
<td>1010 1101</td>
</tr>
<tr>
<td>( Rn = \text{LOGB} Fx )</td>
<td>1100 0001</td>
</tr>
<tr>
<td>( Rn = \text{FIX} Fx \text{ BY Ry} )</td>
<td>1101 1001</td>
</tr>
<tr>
<td>( Rn = \text{FIX} Fx )</td>
<td>1100 1001</td>
</tr>
<tr>
<td>( Rn = \text{TRUNC} Fx \text{ BY Ry} )</td>
<td>1101 1101</td>
</tr>
<tr>
<td>( Rn = \text{TRUNC} Fx )</td>
<td>1100 1101</td>
</tr>
<tr>
<td>( Fn = \text{FLOAT} Rx \text{ BY Ry} )</td>
<td>1101 1010</td>
</tr>
<tr>
<td>( Fn = \text{FLOAT} Rx )</td>
<td>1100 1010</td>
</tr>
<tr>
<td>( Fn = \text{RECIPS} Fx )</td>
<td>1100 0100</td>
</tr>
<tr>
<td>( Fn = \text{RSQRTS} Fx )</td>
<td>1100 0101</td>
</tr>
<tr>
<td>( Fn = Fx \text{ COPYSIGN} Fy )</td>
<td>1110 0000</td>
</tr>
<tr>
<td>( Fn = \text{MIN}(Fx, Fy) )</td>
<td>1110 0001</td>
</tr>
</tbody>
</table>
Multiplier Opcodes

This section describes the multiplier operations. These tables use the following symbols to indicate location of operands and other features:

- $y =$ y-input (1 = signed, 0 = unsigned)
- $x =$ x-input (1 = signed, 0 = unsigned)
- $f =$ format (1 = fractional, 0 = integer)
- $r =$ rounding (1 = yes, 0 = no)

Table 11-5 and Table 11-6 summarize the syntax and opcodes for the fixed-point and floating-point multiplier operations.

Table 11-5. Multiplier Fixed-Point Operations

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_n = R_x \times R_y \mod 1$</td>
<td>01yx f00r</td>
</tr>
<tr>
<td>$MRF = R_x \times R_y \mod 1$</td>
<td>01yx f10r</td>
</tr>
<tr>
<td>$MRB = R_x \times R_y \mod 2$</td>
<td>01yx f11r</td>
</tr>
<tr>
<td>$R_n = MRF + R_x \times R_y \mod 1$</td>
<td>10yx f00r</td>
</tr>
<tr>
<td>$R_n = MRB + R_x \times R_y \mod 1$</td>
<td>10yx f01r</td>
</tr>
<tr>
<td>$MRF = MRF + R_x \times R_y \mod 1$</td>
<td>10yx f10r</td>
</tr>
<tr>
<td>$MRB = MRB + R_x \times R_y \mod 1$</td>
<td>10yx f11r</td>
</tr>
<tr>
<td>$R_n = MRF - R_x \times R_y \mod 1$</td>
<td>11yx f00r</td>
</tr>
</tbody>
</table>
Table 11-5. Multiplier Fixed-Point Operations (Cont’d)

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Rn = \text{MRB} - Rx \times Ry \ mod2 )</td>
<td>11yx f01x</td>
</tr>
<tr>
<td>( \text{MRF} = \text{MRF} - Rx \times Ry \ mod2 )</td>
<td>11yx f10x</td>
</tr>
<tr>
<td>( \text{MRB} = \text{MRB} - Rx \times Ry \ mod1 )</td>
<td>11yx f11x</td>
</tr>
<tr>
<td>( Rn = \text{SAT MRF} \ mod2 )</td>
<td>0000 f00x</td>
</tr>
<tr>
<td>( Rn = \text{SAT MRB} \ mod2 )</td>
<td>0000 f01x</td>
</tr>
<tr>
<td>( \text{MRF} = \text{SAT MRF} \ mod2 )</td>
<td>0000 f10x</td>
</tr>
<tr>
<td>( \text{MRB} = \text{SAT MRB} \ mod2 )</td>
<td>0000 f11x</td>
</tr>
<tr>
<td>( Rn = \text{RND MRF} \ mod3 )</td>
<td>0001 100x</td>
</tr>
<tr>
<td>( Rn = \text{RND MRB} \ mod3 )</td>
<td>0001 101x</td>
</tr>
<tr>
<td>( \text{MRF} = \text{RND MRF} \ mod3 )</td>
<td>0001 110x</td>
</tr>
<tr>
<td>( \text{MRB} = \text{RND MRB} \ mod3 )</td>
<td>0001 111x</td>
</tr>
<tr>
<td>( \text{MRF} = 0 )</td>
<td>0001 0100</td>
</tr>
<tr>
<td>( \text{MRB} = 0 )</td>
<td>0001 0110</td>
</tr>
<tr>
<td>( \text{MRxF/B} = Rn )</td>
<td>0000 0000</td>
</tr>
<tr>
<td>( Rn = \text{MRxF/B} )</td>
<td>0000 0000</td>
</tr>
</tbody>
</table>

Table 11-6. Multiplier Floating-Point Operations

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F_n = F_x \times F_y )</td>
<td>0011 0000</td>
</tr>
</tbody>
</table>
Mod1 Modifiers

The Mod1 modifiers in Table 11-7 are optional modifiers. 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 11-7. Mod1 Options and Opcodes

<table>
<thead>
<tr>
<th>Option</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SSI)</td>
<td>_.11 _0 _0</td>
</tr>
<tr>
<td>(SUI)</td>
<td>_.01 _0 _0</td>
</tr>
<tr>
<td>(USI)</td>
<td>_.10 _0 _0</td>
</tr>
<tr>
<td>(UUI)</td>
<td>_.00 _0 _0</td>
</tr>
<tr>
<td>(SSF)</td>
<td>_.11 _1 _0</td>
</tr>
<tr>
<td>(SUF)</td>
<td>_.01 _1 _0</td>
</tr>
<tr>
<td>(USF)</td>
<td>_.10 _1 _0</td>
</tr>
<tr>
<td>(UUF)</td>
<td>_.00 _1 _0</td>
</tr>
<tr>
<td>(SSFR)</td>
<td>_.11 _1 _1</td>
</tr>
<tr>
<td>(SUFR)</td>
<td>_.01 _1 _1</td>
</tr>
<tr>
<td>(USFR)</td>
<td>_.10 _1 _1</td>
</tr>
<tr>
<td>(UUF)</td>
<td>_.00 _1 _1</td>
</tr>
</tbody>
</table>
Computation Type Opcodes

Mod2 Modifiers

The Mod2 modifiers in Table 11-8 are optional modifiers, 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.

Table 11-8. Mod2 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) (RND only)</td>
<td>_ _ _ _ 1 _ _ 1</td>
</tr>
<tr>
<td>(UF) (RND only)</td>
<td>_ _ _ _ 1 _ _ 0</td>
</tr>
</tbody>
</table>

Mod3 Modifiers

Table 11-9. Mod3 Options and Opcodes

<table>
<thead>
<tr>
<th>Option</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SF) (RND only)</td>
<td>_ _ _ _ 1 _ _ 1</td>
</tr>
<tr>
<td>(UF) (RND only)</td>
<td>_ _ _ _ 1 _ _ 0</td>
</tr>
</tbody>
</table>

MRxF/B = Rn
Rn = MRxF/B

<table>
<thead>
<tr>
<th></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>100000</td>
<td>T</td>
<td>M</td>
<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>
</tr>
</tbody>
</table>
Table 11-10 indicates how M specifies the MR register, and Dreg specifies the data register. T determines the direction of the transfer (0 = register file, 1 = MR register).

Table 11-10. M Values and MR Registers

<table>
<thead>
<tr>
<th>M</th>
<th>MR Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>MR0F</td>
</tr>
<tr>
<td>0001</td>
<td>MR1F</td>
</tr>
<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>
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 y input from the register file when the compute operation is a two-input shifter operation.

Table 11-11 shows 8-bit opcodes which are valid for shifter immediate and Table 11-12 shows the shift operations. For shifter computations the entire 8-bit opcode is valid for bits 19–12, for shift immediate (type 6 instructions only) the upper 6 MSBs represent valid bits 21–16.
### Table 11-11. Shifter Operations Immediate

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn = LSHIFT Rx BY Ry</td>
<td>&lt;data8&gt;</td>
</tr>
<tr>
<td>Rn = Rn OR LSHIFT Rx BY Ry</td>
<td>&lt;data8&gt;</td>
</tr>
<tr>
<td>Rn = ASHIFT Rx BY Ry</td>
<td>&lt;data8&gt;</td>
</tr>
<tr>
<td>Rn = Rn OR ASHIFT Rx BY Ry</td>
<td>&lt;data8&gt;</td>
</tr>
<tr>
<td>Rn = ROT Rx BY Ry</td>
<td>&lt;data8&gt;</td>
</tr>
<tr>
<td>Rn = BCLR Rx BY Ry</td>
<td>&lt;data8&gt;</td>
</tr>
<tr>
<td>Rn = BSET Rx BY Ry</td>
<td>&lt;data8&gt;</td>
</tr>
<tr>
<td>Rn = BTGL Rx BY Ry</td>
<td>&lt;data8&gt;</td>
</tr>
<tr>
<td>BTST Rx BY Ry</td>
<td>&lt;data8&gt;</td>
</tr>
<tr>
<td>Rn = FDEP Rx BY Ry</td>
<td>&lt;bit6&gt;:&lt;len6&gt;</td>
</tr>
<tr>
<td>Rn = Rn OR FDEP Rx BY Ry</td>
<td>&lt;bit6&gt;:&lt;len6&gt;</td>
</tr>
<tr>
<td>Rn = FDEP Rx BY Ry</td>
<td>&lt;bit6&gt;:&lt;len6&gt; (SE)</td>
</tr>
<tr>
<td>Rn = Rn OR FDEP Rx BY Ry</td>
<td>&lt;bit6&gt;:&lt;len6&gt;(SE)</td>
</tr>
<tr>
<td>Rn = FEXT Rx BY Ry</td>
<td>&lt;bit6&gt;:&lt;len6&gt;</td>
</tr>
<tr>
<td>Rn = FEXT Rx BY Ry</td>
<td>&lt;bit6&gt;:&lt;len6&gt; (SE)</td>
</tr>
</tbody>
</table>
| BITDEP Rx BY Ry|<bitlen> 
1  | 0111 0100 |
| Rn = BITEXT Rx|<bitlen>[(NU)] 
1  | 0101 x000  
(x = 1 for NU) |
| BFFWRP = Rn|<value7> 
1  | 0111 1100 |

1 This instruction works on ADSP-214xx processors only.
## Computation Type Opcodes

Table 11-12. Shifter Operations

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn = EXP Rx</td>
<td>1000 0000</td>
</tr>
<tr>
<td>Rn = EXP Rx (EX)</td>
<td>1000 0100</td>
</tr>
<tr>
<td>Rn = LEFTZ Rx</td>
<td>1000 1000</td>
</tr>
<tr>
<td>Rn = LEFTO Rx</td>
<td>1000 1100</td>
</tr>
<tr>
<td>Rn = FPACK Fx</td>
<td>1001 0000</td>
</tr>
<tr>
<td>Fn = FUNPACK Rx</td>
<td>1001 0100</td>
</tr>
<tr>
<td>Rn = BFFWRP^1</td>
<td>0111 0000</td>
</tr>
</tbody>
</table>
# Instruction Opcodes

## Multifunction Opcodes

### Dual ALU (Parallel Add and Subtract)

#### Compute Field (Fixed-Point)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rx</td>
<td>Specifies fixed-point X input ALU register</td>
</tr>
<tr>
<td>Ry</td>
<td>Specifies fixed-point Y input ALU register</td>
</tr>
<tr>
<td>Rs</td>
<td>Specifies fixed-point ALU subtraction result</td>
</tr>
<tr>
<td>Ra</td>
<td>Specifies fixed-point ALU addition result</td>
</tr>
</tbody>
</table>

#### Compute Field (Floating-Point)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fs</td>
<td>Specifies floating-point X input ALU register</td>
</tr>
<tr>
<td>Fa</td>
<td>Specifies floating-point Y input ALU register</td>
</tr>
<tr>
<td>Fs</td>
<td>Specifies floating-point ALU subtraction result</td>
</tr>
<tr>
<td>Fa</td>
<td>Specifies floating-point ALU addition result</td>
</tr>
</tbody>
</table>
## Multifunction Opcodes

### Multiplier and Dual ALU (Parallel Add and Subtract)

#### Compute Field (Fixed-Point)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rxa</td>
<td>Specifies fixed-point X input ALU register (R11–8)</td>
</tr>
<tr>
<td>Rya</td>
<td>Specifies fixed-point Y input ALU register (R15–12)</td>
</tr>
<tr>
<td>Rs</td>
<td>Specifies fixed-point ALU subtraction result</td>
</tr>
<tr>
<td>Ra</td>
<td>Specifies fixed-point ALU addition result</td>
</tr>
<tr>
<td>Fxa</td>
<td>Specifies floating-point X input ALU register (F11–8)</td>
</tr>
<tr>
<td>Fya</td>
<td>Specifies floating-point Y input ALU register (F15–12)</td>
</tr>
<tr>
<td>Fs</td>
<td>Specifies floating-point ALU subtraction result</td>
</tr>
<tr>
<td>Fa</td>
<td>Specifies floating-point ALU addition result</td>
</tr>
</tbody>
</table>

#### Compute Field (Floating-Point)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rxa</td>
<td>Specifies fixed-point X input multiply register (R3–0)</td>
</tr>
<tr>
<td>Rya</td>
<td>Specifies fixed-point Y input multiply register (R7–4)</td>
</tr>
<tr>
<td>Rs</td>
<td>Specifies fixed-point multiply result register</td>
</tr>
</tbody>
</table>
**Instruction Opcodes**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fxa</td>
<td>Specifies floating-point X input multiply register (F3–0)</td>
</tr>
<tr>
<td>Fya</td>
<td>Specifies floating-point Y input multiply register (F7–4)</td>
</tr>
<tr>
<td>Fs</td>
<td>Specifies floating-point multiply result register</td>
</tr>
</tbody>
</table>
**Multifunction Opcodes**

**Multiplier and ALU**

**Compute Field (Fixed-Point)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Ra</td>
<td>Specifies fixed-point ALU result</td>
</tr>
<tr>
<td>21</td>
<td>Rm</td>
<td>Specifies fixed-point multiply result register</td>
</tr>
<tr>
<td>20</td>
<td>Rxm</td>
<td>Specifies fixed-point X input multiply register</td>
</tr>
<tr>
<td>19</td>
<td>Rym</td>
<td>Specifies fixed-point Y input multiply register</td>
</tr>
<tr>
<td>18</td>
<td>Rxa</td>
<td>Specifies fixed-point X input ALU register (R11–8)</td>
</tr>
<tr>
<td>17</td>
<td>Rya</td>
<td>Specifies fixed-point Y input ALU register (R15–12)</td>
</tr>
</tbody>
</table>

**Compute Field (Floating-Point)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Fa</td>
<td>Specifies floating-point ALU result</td>
</tr>
<tr>
<td>21</td>
<td>Fm</td>
<td>Specifies floating-point multiply result register</td>
</tr>
<tr>
<td>20</td>
<td>Fxa</td>
<td>Specifies floating-point X input multiply register (F11–8)</td>
</tr>
<tr>
<td>19</td>
<td>Fym</td>
<td>Specifies floating-point Y input multiply register (F15–12)</td>
</tr>
<tr>
<td>18</td>
<td>Fya</td>
<td>Specifies floating-point Y input ALU register (F15–12)</td>
</tr>
</tbody>
</table>

**Bits**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rxm</td>
</tr>
<tr>
<td>Rym</td>
</tr>
<tr>
<td>Rm</td>
</tr>
<tr>
<td>Fxm</td>
</tr>
<tr>
<td>Fym</td>
</tr>
<tr>
<td>Fm</td>
</tr>
</tbody>
</table>
Table 11-13 provides the syntax and opcode for each of the parallel multiplier and ALU instructions for both fixed-point and floating-point versions.

Table 11-13. Multifunction, Multiplier and ALU

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Opcode (Bits 21–16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_m = R_3–0 \times R_7–4$ (SSFR), $R_a = R_{11–8} + R_{15–12}$</td>
<td>000100</td>
</tr>
<tr>
<td>$R_m = R_3–0 \times R_7–4$ (SSFR), $R_a = R_{11–8} – R_{15–12}$</td>
<td>000101</td>
</tr>
<tr>
<td>$R_m = R_3–0 \times R_7–4$ (SSFR), $R_a = (R_{11–8} + R_{15–12})/2$</td>
<td>000110</td>
</tr>
<tr>
<td>$MRF = MRF + R_3–0 \times R_7–4$ (SSF), $R_a = R_{11–8} + R_{15–12}$</td>
<td>001000</td>
</tr>
<tr>
<td>$MRF = MRF + R_3–0 \times R_7–4$ (SSF), $R_a = R_{11–8} – R_{15–12}$</td>
<td>001001</td>
</tr>
<tr>
<td>$MRF = MRF + R_3–0 \times R_7–4$ (SSF), $R_a = (R_{11–8} + R_{15–12})/2$</td>
<td>001010</td>
</tr>
<tr>
<td>$R_m = MRF – R_3–0 \times R_7–4$ (SSFR), $R_a = R_{11–8} + R_{15–12}$</td>
<td>001100</td>
</tr>
<tr>
<td>$R_m = MRF – R_3–0 \times R_7–4$ (SSFR), $R_a = R_{11–8} – R_{15–12}$</td>
<td>001101</td>
</tr>
<tr>
<td>$R_m = MRF – R_3–0 \times R_7–4$ (SSFR), $R_a = (R_{11–8} + R_{15–12})/2$</td>
<td>001110</td>
</tr>
<tr>
<td>$MRF = MRF – R_3–0 \times R_7–4$ (SSF), $R_a = R_{11–8} + R_{15–12}$</td>
<td>010000</td>
</tr>
<tr>
<td>$MRF = MRF – R_3–0 \times R_7–4$ (SSF), $R_a = R_{11–8} – R_{15–12}$</td>
<td>010001</td>
</tr>
<tr>
<td>$MRF = MRF – R_3–0 \times R_7–4$ (SSF), $R_a = (R_{11–8} + R_{15–12})/2$</td>
<td>010010</td>
</tr>
<tr>
<td>$R_m = MRF – R_3–0 \times R_7–4$ (SSFR), $R_a = R_{11–8} + R_{15–12}$</td>
<td>010100</td>
</tr>
<tr>
<td>$R_m = MRF – R_3–0 \times R_7–4$ (SSFR), $R_a = R_{11–8} – R_{15–12}$</td>
<td>010101</td>
</tr>
<tr>
<td>$R_m = MRF – R_3–0 \times R_7–4$ (SSFR), $R_a = (R_{11–8} + R_{15–12})/2$</td>
<td>010110</td>
</tr>
<tr>
<td>$F_m = F_3–0 \times F_7–4$, $F_a = F_{11–8} + F_{15–12}$</td>
<td>011000</td>
</tr>
<tr>
<td>$F_m = F_3–0 \times F_7–4$, $F_a = F_{11–8} – F_{15–12}$</td>
<td>011001</td>
</tr>
<tr>
<td>$F_m = F_3–0 \times F_7–4$, $F_a = FLOAT R_{11–8}$ by $R_{15–12}$</td>
<td>011010</td>
</tr>
<tr>
<td>$F_m = F_3–0 \times F_7–4$, $F_a = FIX F_{11–8}$ by $R_{15–12}$</td>
<td>011011</td>
</tr>
<tr>
<td>$F_m = F_3–0 \times F_7–4$, $F_a = (F_{11–8} + F_{15–12})/2$</td>
<td>011100</td>
</tr>
<tr>
<td>$F_m = F_3–0 \times F_7–4$, $F_a = ABS F_{11–8}$</td>
<td>011101</td>
</tr>
</tbody>
</table>
Register Opcodes

The SHARC core classifies the following register types.

- universal register (UREG)
- data register (DREG) subgroup of UREG
- system register (SREG) subgroup of UREG
- non universal register

When operating in SIMD mode, most of the register types use complementary registers (CDREG, CSREG, UUREG). One exception is for the combined PX register (PX1 and PX2) which are classified as complementary universal registers (CUREG). This classification is required to understand the instruction coding for universal registers in the tables in the following sections.

Non Universal Registers

Note the multiplier result registers (MRF/MRB/MSF/MSB) are not included into the universal registers and therefore do not support full orthogonal instruction coding. For these registers only specific multiplier instructions are coded.

Table 11-13. Multifunction, Multiplier and ALU (Cont’d)

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Opcode (Bits 21–16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_m = F_3–0 \times F_7–4$, $F_a = \text{MAX} (F_{11–8}, F_{15–12})$</td>
<td>011110</td>
</tr>
<tr>
<td>$F_m = F_3–0 \times F_7–4$, $F_a = \text{MIN} (F_{11–8}, F_{15–12})$</td>
<td>011111</td>
</tr>
</tbody>
</table>
### Universal Register Opcodes

Table 11-14 shows how the Ureg register codes appear to PEx.

#### Table 11-14. Processing Element X Universal Register Codes (SISD/SIMD)

<table>
<thead>
<tr>
<th>Bits: 3210</th>
<th>DREG</th>
<th>UUREG</th>
<th>CDREG</th>
<th>UUREG</th>
<th>SREG</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000</td>
<td>R0</td>
<td>001</td>
<td>010</td>
<td>011</td>
</tr>
<tr>
<td>3210</td>
<td>0000</td>
<td>R0</td>
<td>10</td>
<td>M0</td>
<td>L0</td>
</tr>
<tr>
<td>3120</td>
<td>0001</td>
<td>R1</td>
<td>11</td>
<td>M1</td>
<td>L1</td>
</tr>
<tr>
<td>3010</td>
<td>0010</td>
<td>R2</td>
<td>12</td>
<td>M2</td>
<td>L2</td>
</tr>
<tr>
<td>2901</td>
<td>0011</td>
<td>R3</td>
<td>13</td>
<td>M3</td>
<td>L3</td>
</tr>
<tr>
<td>2801</td>
<td>0100</td>
<td>R4</td>
<td>14</td>
<td>M4</td>
<td>L4</td>
</tr>
<tr>
<td>2701</td>
<td>0101</td>
<td>R5</td>
<td>15</td>
<td>M5</td>
<td>L5</td>
</tr>
<tr>
<td>2601</td>
<td>0110</td>
<td>R6</td>
<td>16</td>
<td>M6</td>
<td>L6</td>
</tr>
<tr>
<td>2501</td>
<td>0111</td>
<td>R7</td>
<td>17</td>
<td>M7</td>
<td>L7</td>
</tr>
<tr>
<td>2401</td>
<td>1000</td>
<td>R8</td>
<td>18</td>
<td>M8</td>
<td>L8</td>
</tr>
<tr>
<td>2301</td>
<td>1001</td>
<td>R9</td>
<td>19</td>
<td>M9</td>
<td>L9</td>
</tr>
<tr>
<td>2201</td>
<td>1010</td>
<td>R10</td>
<td>110</td>
<td>M10</td>
<td>L10</td>
</tr>
<tr>
<td>2101</td>
<td>1011</td>
<td>R11</td>
<td>111</td>
<td>M11</td>
<td>L11</td>
</tr>
<tr>
<td>2011</td>
<td>1100</td>
<td>R12</td>
<td>112</td>
<td>M12</td>
<td>L12</td>
</tr>
<tr>
<td>1911</td>
<td>1101</td>
<td>R13</td>
<td>113</td>
<td>M13</td>
<td>L13</td>
</tr>
<tr>
<td>1811</td>
<td>1110</td>
<td>R14</td>
<td>114</td>
<td>M14</td>
<td>L14</td>
</tr>
<tr>
<td>1711</td>
<td>1111</td>
<td>R15</td>
<td>115</td>
<td>M15</td>
<td>L15</td>
</tr>
</tbody>
</table>

Table 11-15 shows how the Ureg register codes appear to PEy.
## Register Opcodes

Table 11-15. Processing Element Y Universal Register Codes (SIMD)

<table>
<thead>
<tr>
<th>Bits: 3210</th>
<th>Bits: 654000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>S0 I0 M0 L0 B0 R0</td>
<td>FADDR</td>
<td>USTAT2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>S1 I1 M1 L1 B1 R1</td>
<td>DADDR</td>
<td>USTAT1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>S2 I2 M2 L2 B2 R2</td>
<td>MODE1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>S3 I3 M3 L3 B3 R3</td>
<td>PC</td>
<td>MMASK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>S4 I4 M4 L4 B4 R4</td>
<td>PCSTK</td>
<td>MODE2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>S5 I5 M5 L5 B5 R5</td>
<td>PCSTKP</td>
<td>FLAGS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>S6 I6 M6 L6 B6 R6</td>
<td>LADDR</td>
<td>ASTATy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>S7 I7 M7 L7 B7 R7</td>
<td>CURL-CNTR</td>
<td>ASTATx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>S8 I8 M8 L8 B8 R8</td>
<td>LCNTR</td>
<td>STKYy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>S9 I9 M9 L9 B9 R9</td>
<td>EMUCLK</td>
<td>STKYx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>S10 I10 M10 L10 B10 R10</td>
<td>EMUCLK2</td>
<td>IRPTL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>S11 I11 M11 L11 B11 R11</td>
<td>PX</td>
<td>IMASK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>S12 I12 M12 L12 B12 R12</td>
<td>PX2</td>
<td>IMASKP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>S13 I13 M13 L13 B13 R13</td>
<td>PX1</td>
<td>LRPTL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>S14 I14 M14 L14 B14 R14</td>
<td>TPERIOD</td>
<td>USTAT4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>S15 I15 M15 L15 B15 R15</td>
<td>TCOUNT</td>
<td>USTAT3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Condition and Termination Opcodes

The SHARC instruction set supports IF conditions and DO UNTIL terminations, these are coded in the 5-bit COND or TERM field (0–31),

Table 11-16. IF Conditions and Termination Codes

<table>
<thead>
<tr>
<th>COND/TERM</th>
<th>Opcode</th>
<th>COND/TERM</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>00000</td>
<td>NE</td>
<td>10000</td>
</tr>
<tr>
<td>LT</td>
<td>00001</td>
<td>GE</td>
<td>10001</td>
</tr>
<tr>
<td>LE</td>
<td>00010</td>
<td>GT</td>
<td>10010</td>
</tr>
<tr>
<td>AC</td>
<td>00011</td>
<td>NOT AC</td>
<td>10011</td>
</tr>
<tr>
<td>AV</td>
<td>00100</td>
<td>NOT AV</td>
<td>10100</td>
</tr>
<tr>
<td>MV</td>
<td>00101</td>
<td>NOT MV</td>
<td>10101</td>
</tr>
<tr>
<td>MS</td>
<td>00110</td>
<td>NOT MS</td>
<td>10110</td>
</tr>
<tr>
<td>SV</td>
<td>00111</td>
<td>NOT SV</td>
<td>10111</td>
</tr>
<tr>
<td>SZ</td>
<td>01000</td>
<td>NOT SZ</td>
<td>11000</td>
</tr>
<tr>
<td>FLAG0</td>
<td>01001</td>
<td>NOT FLAG0</td>
<td>11001</td>
</tr>
<tr>
<td>FLAG1</td>
<td>01010</td>
<td>NOT FLAG1</td>
<td>11010</td>
</tr>
<tr>
<td>FLAG2</td>
<td>01011</td>
<td>NOT FLAG2</td>
<td>11011</td>
</tr>
<tr>
<td>FLAG3</td>
<td>01100</td>
<td>NOT FLAG3</td>
<td>11100</td>
</tr>
<tr>
<td>TF</td>
<td>01101</td>
<td>NOT TF</td>
<td>11101</td>
</tr>
<tr>
<td>BM/SF1</td>
<td>01110</td>
<td>NOT BM/SF1</td>
<td>11110</td>
</tr>
<tr>
<td>LCE/NOT LCE</td>
<td>01111</td>
<td>TRUE2/FOREVER</td>
<td>11111</td>
</tr>
</tbody>
</table>

1 For ADSP-21368 valid bus master condition, for ADSP-214xx valid bit shifter FIFO.
2 COND 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.
Condition and Termination Opcodes
A REGISTERS

The SHARC processors have two types of registers, non memory-mapped and memory-mapped. Non memory-mapped registers are not accessed by an address (like memory-mapped registers), instead they are accessed by an instruction.

Memory-mapped registers are sub-classified as IOP (I/O processor) core registers and IOP peripheral registers. For information IOP peripheral registers, refer to the product specific hardware reference manual.

- “Program Sequencer Registers” on page A-8
- “Processing Element Registers” on page A-14
- “Data Address Generator Registers” on page A-24
- “Miscellaneous Registers” on page A-25
- “Memory-Mapped Registers” on page A-41
- “Interrupt Registers” on page A-31
- “Register Listing” on page A-50

When writing processor programs, it is often necessary to set, clear, or test bits in the processor’s registers. While these bit operations can all be done by referring to the bit’s location within a register it is much easier to use symbols that correspond to the bit’s or register’s name. For convenience and consistency, Analog Devices provides a header file that contains these bit and registers definitions. An #include file is provided with the
Many registers have reserved bits. When writing to a register, programs may only clear (write zero to) the register’s reserved bits.

Notes on Reading Register Drawings

The register drawings in this appendix provide “at-a-glance” information about specific registers. They are designed to give experienced users basic information about a register and its bit settings. When using these registers, the following should be noted.

1. The figures provide the bit mnemonic and its definition. Where necessary, detailed descriptions can be found in the tables that follow the register drawings and in the chapters that describe the particular module.

2. The VisualDSP++ tools suite contains the complete listing of registers in a header file.

3. “Register Listing” on page A-50 provides a complete list of user accessible registers, their addresses, and their state at reset.

4. In most cases, control registers are read/write (RW) and status registers are read only (RO). Some registers provide error status bits which can be written to clear. Where individual bits within a register differ, they are noted in the register drawing.
Mode Control 1 Register (MODE1)

Figure A-1 and Table A-2 provide bit information for the MODE1 register.

Figure A-1. Mode Control 1 Register
<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>BR8</td>
<td>Bit-Reverse Addressing For Index I8 Enable. Enables (bit reversed if set, = 1) or disables (normal if cleared, = 0) bit-reversed addressing for accesses that are indexed with DAG2 register I8.</td>
</tr>
<tr>
<td>1</td>
<td>BR0</td>
<td>Bit-Reverse Addressing For Index I0 Enable. Enables (bit reversed if set, = 1) or disables (normal if cleared, = 0) bit-reversed addressing for accesses that are indexed with DAG1 register I0.</td>
</tr>
<tr>
<td>2</td>
<td>SRCU</td>
<td>MRx Result Registers Swap Enable. Enables the swapping of the MRF and MRB registers contents if set (= 1). This can be used as foreground and background registers. In SIMD Mode the swapping also performed between MSF and MSB registers. This works similar to the data register swapping instructions Rx&lt;-&gt;Sx.</td>
</tr>
<tr>
<td>3</td>
<td>SRD1H</td>
<td>Secondary Registers For DAG1 High Enable. Enables (use secondary if set, = 1) or disables (use primary if cleared, = 0) secondary DAG1 registers for the upper half (I, M, L, B7–4) of the address generator.</td>
</tr>
<tr>
<td>4</td>
<td>SRD1L</td>
<td>Secondary Registers For DAG1 Low Enable. Enables (use secondary if set, = 1) or disables (use primary if cleared, = 0) secondary DAG1 registers for the lower half (I, M, L, B3–0) of the address generator.</td>
</tr>
<tr>
<td>5</td>
<td>SRD2H</td>
<td>Secondary Registers For DAG2 High Enable. Enables (use secondary if set, = 1) or disables (use primary if cleared, = 0) secondary DAG2 registers for the upper half (I, M, L, B15–12) of the address generator.</td>
</tr>
<tr>
<td>6</td>
<td>SRD2L</td>
<td>Secondary Registers For DAG2 Low Enable. Enables (use secondary if set, = 1) or disables (use primary if cleared, = 0) secondary DAG2 registers for the lower half (I, M, L, B11–8) of the address generator.</td>
</tr>
<tr>
<td>7</td>
<td>SRRFH</td>
<td>Secondary Registers For Register File High Enable. Enables (use secondary if set, = 1) or disables (use primary if cleared, = 0) secondary data registers for the upper half (R15-R8/S15-S8) of the computational units.</td>
</tr>
<tr>
<td>9–8</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>SRRFL</td>
<td>Secondary Registers For Register File Low Enable. Enables (use secondary if set, = 1) or disables (use primary if cleared, = 0) secondary data registers for the lower half (R7-R0/S7-S0) of the computational units.</td>
</tr>
</tbody>
</table>
### Registers

**Table A-1. MODE1 Register Bit Descriptions (RW) (Cont’d)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>NESTM</td>
<td><strong>Nesting Multiple Interrupts Enable.</strong> Enables (nest if set, = 1) or disables (no nesting if cleared, = 0) interrupt nesting in the interrupt controller. When interrupt nesting is disabled, a higher priority interrupt can not interrupt a lower priority interrupt's service routine. Other interrupts are latched as they occur, but the processor processes them after the active routine finishes. When interrupt nesting is enabled, a higher priority interrupt can interrupt a lower priority interrupt's service routine. Lower interrupts are latched as they occur, but the processor processes them after the nested routines finish.</td>
</tr>
<tr>
<td>12</td>
<td>IRPTEN</td>
<td><strong>Global Interrupt Enable.</strong> Enables (if set, = 1) or disables (if cleared, = 0) all maskable interrupts.</td>
</tr>
<tr>
<td>13</td>
<td>ALUSAT</td>
<td><strong>ALU Saturation Select.</strong> Selects whether the computational units saturate results on positive or negative fixed-point overflows (if 1) or return unsaturated results (if 0).</td>
</tr>
<tr>
<td>14</td>
<td>SSE</td>
<td><strong>Fixed-point Sign Extension Select.</strong> Selects whether the core unit sign-extend short-word, 16-bit data (if 1) or zero-fill the upper 16 bits (if 0).</td>
</tr>
<tr>
<td>15</td>
<td>TRUNC</td>
<td><strong>Truncation Rounding Mode Select.</strong> Selects whether the computational units round results with round-to-zero (if 1) or round-to-nearest (if 0).</td>
</tr>
<tr>
<td>16</td>
<td>RND32</td>
<td><strong>Boundary Rounding For 32-Bit Floating-Point Data Select.</strong> Selects whether the computational units round floating-point data to 32 bits (if 1) or round to 40 bits (if 0).</td>
</tr>
</tbody>
</table>
| 18–17| CSEL       | **Bus Master Selection.** These bits indicate whether the processor processor has control of the external bus as follows:  
00 = processor is bus master  
01, 10, 11 = processor is not bus master.  
The bus master condition (BM) indicates whether the SHARC processor is the current bus master in EP shared systems (e.g. ADSP-21368 with shared SDRAM memory). To enable the use of this condition, bits 17 and 18 of MODE1 must both be zeros; otherwise the condition is always evaluated as false. |
| 20–19| Reserved   |                                                                                                                                             |
### Mode Control 1 Register (MODE1)

Table A-1. MODE1 Register Bit Descriptions (RW) (Cont’d)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>PEYEN</td>
<td><strong>Processor Element Y Enable.</strong> Enables computations in PEy—SIMD mode—(if 1) or disables PEy—SISD mode—(if 0). When set, processing element Y (computation units and register files) accepts instruction dispatches. When cleared, processing element Y goes into a low power mode. Note if SIMD Mode is disabled, programs can load data to the secondary registers—for example s0=dm(i0,m0); only computation does not work.</td>
</tr>
<tr>
<td>22</td>
<td>BDCST9</td>
<td><strong>Broadcast Register Loads Indexed With I9 Enable.</strong> Enables (broadcast I9 if set, = 1) or disables (no I9 broadcast if cleared, = 0) broadcast register loads for loads that use the data address generator I9 index. When the BDCST9 bit is set, data register loads from the PM data bus that use the I9 DAG2 Index register are “broadcast” to a register or register pair in each PE.</td>
</tr>
<tr>
<td>23</td>
<td>BDCST1</td>
<td><strong>Broadcast Register Loads Indexed With I1 Enable.</strong> Enables (broadcast I1 if set, = 1) or disables (no I1 broadcast if cleared, = 0) broadcast register loads for loads that use the data address generator I1 index. When the BDCST1 bit is set, data register loads from the DM data bus that use the I1 DAG1 Index register are “broadcast” to a register or register pair in each PE.</td>
</tr>
<tr>
<td>24</td>
<td>CBUFEN</td>
<td><strong>Circular Buffer Addressing Enable.</strong> Enables (circular if set, = 1) or disables (linear if cleared, = 0) circular buffer addressing for buffers with loaded I, M, B, and L DAG registers.</td>
</tr>
<tr>
<td>31–25</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>
Mode Control 2 Register (MODE2)

Figure A-2 and Table A-2 provide bit information for the MODE2 register.

Table A-2. MODE2 Register Bit Descriptions (RW)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>IRQ0E</td>
<td>Sensitivity Select. Selects sensitivity for the flag configured as IRQ0 as edge-sensitive (if set, = 1) or level-sensitive (if cleared, = 0).</td>
</tr>
<tr>
<td>1</td>
<td>IRQ1E</td>
<td>Sensitivity Select. Selects sensitivity for the flag configured as IRQ1 as edge-sensitive (if set, = 1) or level-sensitive (if cleared, = 0).</td>
</tr>
<tr>
<td>2</td>
<td>IRQ2E</td>
<td>Sensitivity Select. Selects sensitivity for the flag configured as IRQ2 as edge-sensitive (if set, = 1) or level-sensitive (if cleared, = 0).</td>
</tr>
<tr>
<td>3</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>
Program Sequencer Registers

The processor’s program sequencer registers direct the execution of instructions. These registers include support for the:

- Instruction pipeline
- Program and loop stacks

### Table A-2. MODE2 Register Bit Descriptions (RW) (Cont’d)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>CADIS</td>
<td>Cache Disable. This bit disables the instruction cache (if set, = 1) or enables the cache (if cleared, = 0). If this bit is set, then the caching of instructions from internal memory and external memory both are disabled (see bit 6).</td>
</tr>
<tr>
<td>5</td>
<td>TIMEN</td>
<td>Timer Enable. Enables the core timer (starts, if set, = 1) or disables the core timer (stops, if cleared, = 0).</td>
</tr>
<tr>
<td>6</td>
<td>EXTCADIS</td>
<td>External Cache Only Disable. Disables the caching of the instructions coming from external memory (if set, =1) or enables caching of the instructions coming from external memory (if cleared, = 0 and CADIS bit 4 = 0). This bit can only be used with the ADSP-214xx products.</td>
</tr>
<tr>
<td>18–7 Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>CAFRZ</td>
<td>Cache Freeze. Freezes the instruction cache (retain contents if set, = 1) or thaws the cache (allow new input if cleared, = 0).</td>
</tr>
<tr>
<td>20</td>
<td>IIRAE</td>
<td>Illegal I/O Processor Register Access Enable. Enables (if set, = 1) or disables (if cleared, = 0) detection of I/O processor register accesses. If IIRAE is set, the processor flags an illegal access by setting the IIRA bit in the STKYx register.</td>
</tr>
<tr>
<td>21</td>
<td>U64MAE</td>
<td>Unaligned 64-Bit Memory Access Enable. Enables (if set, = 1) or disables (if cleared, = 0) detection of unaligned long word accesses. If U64MAE is set, the processor flags an unaligned long word access by setting the U64MA bit in the STKYx register.</td>
</tr>
<tr>
<td>31–22 Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
• Timer

• Interrupt mask and latch (for more information, see “Core Interrupt Control” in Appendix B, Core Interrupt Control.

**Fetch Address Register (FADDR)**

The fetch address register (RO) reads the F1 stage in the F1–F2–D–A–E pipeline stages instruction pipeline and contains the 24-bit address of the instruction that the processor fetches from memory on the next cycle as shown below.

```
n: RO=FADDR;
n+1: instruction1;
n+2: instruction2;
n+3: instruction3;
n+4: instruction4; /* Fetch1 address in FADDR */
n+5: instruction5;
```

**Decode Address Register (DADDR)**

The decode address register (RO) reads the third stage in the F1-F2-D-A-E pipeline stages and contains the 24-bit address of the instruction that the processor decodes on the next cycle as shown below.

```
n: RO=DADDR;
n+1: instruction1;
n+2: instruction2; /* Decode address in DADDR */
n+3: instruction3;
n+4: instruction4;
n+5: instruction5;
```
Program Sequencer Registers

Program Counter Register (PC)

The program count register (RO) reads the last stage in the F1–F2–D–A–E pipeline and contains the 24-bit address of the instruction that the processor executes on the next cycle. The PC register works with the program counter stack, PCSTK register which stores return addresses and top-of-loop addresses. All PC relative branch instruction require access to the register.

\[
\text{n:RO=PC; } \quad /* \text{Execution address in PC */}
\text{n+1:instruction1;}
\text{n+2:instruction2;}
\text{n+3:instruction3;}
\text{n+4:instruction4;}
\text{n+5:instruction5;}
\]

Program Counter Stack Register (PCSTK)

This is a 26-bit register. The program counter stack register contains the address of the top of the PC stack.

Table A-3. PCSTK Register Bit Descriptions (RW)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>23–0</td>
<td>Return Address</td>
</tr>
<tr>
<td>24^1</td>
<td>Set to 1 when the entry is pushed by a CALL</td>
</tr>
<tr>
<td>25^1</td>
<td>Set to 1 when a CALL pushes the return address under the situation when the loop termination condition tests true in the cycle CALL is in the Address stage of the pipeline OR when the push is result of servicing an interrupt.</td>
</tr>
</tbody>
</table>

^1 This bit is available on the ADSP-2137x and later models (ADSP-214xx).
Program Counter Stack Pointer Register (PCSTKP)

The program counter stack pointer register contains the value of PCSTKP. This value is given as follows: 0 when the PC stack is empty, 1...30 when the stack contains data, and 31 when the stack overflows. This register is readable and writable. A write to PCSTKP takes effect after a one-cycle delay. If the PC stack is overflowed, a write to PCSTKP has no effect.

Loop Registers

The loop registers are used set up and track loops in programs. These registers are described below.

Loop Address Stack Register (LADDR)

The loop address stack described in Table A-4, is six levels deep by 32 bits wide. The 32-bit word of each level consists of a 24-bit loop termination address, a 5-bit termination code, and a 3-bit loop type code.

Table A-4. LADDR Register Bit Descriptions (RW)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>23–0</td>
<td>Loop Termination Address</td>
</tr>
<tr>
<td>28–24</td>
<td>Termination Code</td>
</tr>
<tr>
<td>31–29</td>
<td>Loop Type Code</td>
</tr>
<tr>
<td></td>
<td>000 arithmetic condition-based loop (not LCE)</td>
</tr>
<tr>
<td></td>
<td>001 arithmetic condition-based, of length 1</td>
</tr>
<tr>
<td></td>
<td>010 counter-based loop, length 1</td>
</tr>
<tr>
<td></td>
<td>100 counter-based loop, length 2</td>
</tr>
<tr>
<td></td>
<td>110 counter-based loop, length 3</td>
</tr>
<tr>
<td></td>
<td>111 counter-based loop, length &gt; 3</td>
</tr>
</tbody>
</table>
**Timer Registers**

**Loop Counter Register (LCNTR)**

The loop counter register provides access to the loop counter stack and holds the count value before the **DO UNTIL** termination loop is executed. For more information on how to use the **LCNTR** register, see “Loop Counter Stack Access” on page 4-49.

**Current Loop Counter Register (CURLCNTR)**

The current loop counter register provides access to the loop counter stack and tracks iterations for the **DO UNTIL LCE** loop being executed. For more information on how to use the **CURLCNTR** register, see “Loop Counter Stack Access” on page 4-49.

**Timer Registers**

The SHARC processors contain a timer used to generate interrupts from the core. These registers are described below.

**Timer Period Register (TPERIOD)**

The timer period register contains the timer period, indicating the number of cycles between timer interrupts. For more information on how to use the **TPERIOD** register, see Chapter 5, Timer.

**Timer Count Register (TCOUNT)**

The timer count register contains the decrementing timer count value, counting down the cycles between timer interrupts. For more information on how to use the **TCOUNT** register, see Chapter 5, Timer.
Flag I/O Register (FLAGS)

The **FLAGS** register indicates the state of the **FLAGx** pins. When a **FLAGx** pin is an output, the processor outputs a high in response to a program setting the bit in the **FLAGS** register. The I/O direction (input or output) selection of each bit is controlled by its **FLGxO** bit in the **FLAGS** register.

There are 16 flags in SHARC processors. The **FLAG0-3** pins have four dedicated pins. All flag pins can be multiplexed with parallel or external port pins. The **FLAG4-15** pins are also accessible to the signal routing unit (SRU). A Flag pin can be routed to a DAI/DPI pin and multiplexed in parallel to the parallel or external port. Refer to the product specific hardware reference for more information.

Programs cannot change the output selects of the **FLAGS** register and provide a new value in the same instruction. Instead, programs must use two write instructions—the first to change the output select of a particular **FLAG** pin, and the second to provide the new value.

For the **FLAGS** register bit definitions in **Table A-5**:

- For all **FLGx** bits, **FLAGx** values are as follows: 0 = low, 1 = high.
- For all **FLGxO** bits, **FLAGx** output selects are as follows: 0 = **FLAGx** Input, 1 = **FLAGx** Output.
- **FLG3–0** can be immediately used for conditional instruction.

**Table A-5. FLAGS Register Bit Descriptions (RW)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>30–0 (Even bits)</td>
<td>FLGx</td>
<td><strong>FLAGx Value.</strong> Indicates the state of the <strong>FLAGx</strong> pin—high (if set, = 1) or low (if cleared, = 0).</td>
</tr>
<tr>
<td>31–1 (Odd bits)</td>
<td>FLGxO</td>
<td><strong>FLAGx Output Select.</strong> Selects the I/O direction for the <strong>FLAGx</strong> pin, the flag is programmed as an output (if set, = 1) or input (if cleared, = 0).</td>
</tr>
</tbody>
</table>
**Processing Element Registers**

**Processing Element Registers**

Except for the PX register, the processor’s processing element registers store data for each element’s ALU, multiplier, and shifter. The inputs and outputs for processing element operations go through these registers. All processing element registers are read-write (RW).

**PEx Data Registers (Rx)**

Each of the processor’s processing elements has a data register file—a set of 40-bit data registers that transfer data between the data buses and the computation units. These registers also provide local storage for operands and results.

The R, F prefixes on register names do not effect the 32-bit or 40-bit data transfer; the naming convention determines how the ALU, multiplier, and shifter treat the data and determines which processing element’s data registers are being used. For more information on how to use these registers, see Chapter 2, Register Files.

**PEy Data Registers (Sx)**

Each of the processor’s processing elements has a data register file—a set of 40-bit data registers that transfer data between the data buses and the computation units. These registers also provide local storage for SIMD mode.

The S prefix on register names do not effect the 32-bit or 40-bit data transfer; the naming convention determines how the ALU, multiplier, and shifter treat the data and determines which processing element’s data registers are being used.
Alternate Data Registers (Rx', Sx')

The processor includes alternate register sets for all data registers to facilitate fast context switching. Bits in the MODE1 register control when alternate registers become accessible. While inaccessible, the contents of alternate registers are not affected by processor operations. Note that there is an one cycle latency between writing to MODE1 and being able to access an alternate register set.

For more information, see “Complementary Data Registers” on page 2-5.

PEx Multiplier Results Registers (MRFx, MRBx)

Each of the processor’s multiply result has a primary or foreground (MRF) register and alternate or background (MRB) result register. Fixed-point operations place 80-bit results in the MAC’s foreground MRF register or background MRB register, depending on which is active.

PEy Multiplier Results Registers (MSFx, MSBx)

Each of the processor’s multiply result unit has a primary or foreground (MSF) register and alternate or background (MSB) result register. Fixed-point operations place 80-bit results in the MAC’s foreground MSF register or background MSB register, depending on which is active. Note that the PEy multiply result registers can’t be used in an explicit instruction.

Processing Status Registers

The following registers return status information for the processing elements. This information includes computation results and errors.
Arithmetic Status Registers (ASTATx and ASTATy)

Each processing element has its own ASTAT register. The ASTATx register indicates status for PEx operations, the ASTATy register indicates status for PEy operations. Figure A-3 and Table A-6 provide bit information for the ASTAT registers.

![ASTAT Register Diagram]

If these registers are loaded manually, there is a one cycle effect latency before the new value in the ASTATx register can be used in a conditional instruction.
Table A-6. ASTATx and ASTATy Register Bit Descriptions (RW)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AZ</td>
<td><strong>ALU Zero/Floating-Point Underflow.</strong> Indicates if the last ALU operation's result was zero (if set, = 1) or non-zero (if cleared, = 0). The ALU updates AZ for all fixed-point and floating-point ALU operations. AZ can also indicate a floating-point underflow. During an ALU underflow (indicated by a set (= 1) AUS bit in the STKYx/y register), the processor sets AZ if the floating-point result is smaller than can be represented in the output format.</td>
</tr>
<tr>
<td>1</td>
<td>AV</td>
<td><strong>ALU Overflow.</strong> Indicates if the last ALU operation's result overflowed (if set, = 1) or did not overflow (if cleared, = 0). The ALU updates AV for all fixed-point and floating-point ALU operations. For fixed-point results, the processor sets AV and the AOS bit in the STKYx/y register when the XOR of the two most significant bits (MSBs) is a 1. For floating-point results, the processor sets AV and the AVS bit in the STKYx/y register when the rounded result overflows (unbiased exponent &gt; 127).</td>
</tr>
<tr>
<td>2</td>
<td>AN</td>
<td><strong>ALU Negative.</strong> Indicates if the last ALU operation's result was negative (if set, = 1) or positive (if cleared, = 0). The ALU updates AN for all fixed-point and floating-point ALU operations.</td>
</tr>
<tr>
<td>3</td>
<td>AC</td>
<td><strong>ALU Fixed-Point Carry.</strong> Indicates if the last ALU operation had a carry out of the MSB of the result (if set, = 1) or had no carry (if cleared, = 0). The ALU updates AC for all fixed-point operations. The processor clears AC during the fixed-point logic operations: PASS, MIN, MAX, COMP, ABS, and CLIP. The ALU reads the AC flag for the fixed-point accumulate operations: Addition with Carry and Fixed-point Subtraction with Carry.</td>
</tr>
<tr>
<td>4</td>
<td>AS</td>
<td><strong>ALU X-Input Sign (for ABS and MANT).</strong> Indicates if the last ALU ABS or MANT operation's input was negative (if set, = 1) or positive (if cleared, = 0). The ALU updates AS only for fixed- and floating-point ABS and MANT operations. The ALU clears AS for all operations other than ABS and MANT.</td>
</tr>
</tbody>
</table>
## Processing Status Registers

### Table A-6. ASTATx and ASTATy Register Bit Descriptions (RW) (Cont’d)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 5   | AI   | **ALU Floating-Point Invalid Operation.** Indicates if the last ALU operation’s input was invalid (if set, = 1) or valid (if cleared, = 0). The ALU updates AI for all fixed- and floating-point ALU operations. The processor sets AI and AIS in the STKYx/y register if the ALU operation:  
  - Receives a NAN input operand  
  - Adds opposite-signed infinities  
  - Subtracts like-signed infinities  
  - Overflows during a floating-point to fixed-point conversion when saturation mode is not set  
  - Operates on an infinity during a floating-point to fixed-point operation when the saturation mode is not set |
| 6   | MN   | **Multiplier Negative.** Indicates if the last multiplier operation’s result was negative (if set, = 1) or positive (if cleared, = 0). The multiplier updates MN for all fixed- and floating-point multiplier operations. |
| 7   | MV   | **Multiplier Overflow.** Indicates if the last multiplier operation’s result overflowed (if set, = 1) or did not overflow (if cleared, = 0). The multiplier updates MV for all fixed-point and floating-point multiplier operations. For floating-point results, the processor sets MV and MVS in the STKYx/y register if the rounded result overflows (unbiased exponent > 127). For fixed-point results, the processor sets MV and the MOS bit in the STKYx/y register if the result of the multiplier operation is:  
  - Twos-complement, fractional with the upper 17 bits of MR not all zeros or all ones  
  - Twos-complement, integer with the upper 49 bits of MR not all zeros or all ones  
  - Unsigned, fractional with the upper 16 bits of MR not all zeros  
  - Unsigned, integer with the upper 48 bits of MR not all zeros  
If the multiplier operation directs a fixed-point result to an MR register, the processor places the overflowed portion of the result in MR1 and MR2 for an integer result or places it in MR2 only for a fractional result. |
### Table A-6. ASTATx and ASTATy Register Bit Descriptions (RW) (Cont’d)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 8   | MU            | **Multiplier Floating-Point Underflow.** Indicates if the last multiplier operation's result underflowed (if set, = 1) or did not underflow (if cleared, = 0). The multiplier updates MU for all fixed- and floating-point multiplier operations. For floating-point results, the processor sets MU and the MUS bit in the STKYx/y register if the floating-point result underflows (unbiased exponent < -126). Denormal operands are treated as zeros, therefore they never cause underflows. For fixed-point results, the processor sets MU and the MUS bit in the STKYx/y register if the result of the multiplier operation is:  
  - Twos-complement, fractional: with upper 48 bits all zeros or all ones, lower 32 bits not all zeros  
  - Unsigned, fractional: with upper 48 bits all zeros, lower 32 bits not all zeros  
If the multiplier operation directs a fixed-point, fractional result to an MR register, the processor places the underflowed portion of the result in MR0. |
| 9   | MI            | **Multiplier Floating-Point Invalid Operation.** Indicates if the last multiplier operation's input was invalid (if set, = 1) or valid (if cleared, = 0). The multiplier updates MI for floating-point multiplier operations. The processor sets MI and the MIS bit in the STKYx/y register if the ALU operation:  
  - Receives a NAN input operand  
  - Receives an Infinity and zero as input operands |
| 10  | AF            | **ALU Floating-Point Operation.** Indicates if the last ALU operation was floating-point (if set, = 1) or fixed-point (if cleared, = 0). The ALU updates AF for all fixed-point and floating-point ALU operations. |
| 11  | SV            | **Shifter Overflow.** Indicates if the last shifter operation's result overflowed (if set, = 1) or did not overflow (if cleared, = 0). The shifter updates SV for all shifter operations. The processor sets SV if the shifter operation:  
  - Shifts the significant bits to the left of the 32-bit fixed-point field  
  - Tests, sets, or clears a bit outside of the 32-bit fixed-point field  
  - Extracts a field that is past or crosses the left edge of the 32-bit fixed-point field  
  - Performs a LEFTZ or LEFTO operation that returns a result of 32 |
| 12  | SZ            | **Shifter Zero.** Indicates if the last shifter operation's result was zero (if set, = 1) or non-zero (if cleared, = 0). The shifter updates SZ for all shifter operations. The processor also sets SZ if the shifter operation performs a bit test on a bit outside of the 32-bit fixed-point field. |
Sticky Status Registers (STKYx and STKYy)

Each processing element has its own STKY register. The STKYx register indicates status for PEx operations and some program sequencer stacks. The STKYy register only indicates status for PEy operations.

STKY bits do not clear themselves after the condition they flag is no longer true. They remain “sticky” until cleared by the program.

The processor sets a STKY bit in response to a condition. For example, the processor sets the AUS bit in the STKY register when an ALU underflow set AZ in the ASTAT register. The processor clears AZ if the next ALU operation does not cause an underflow. The AUS bit remains set until a program
clears the $\text{STKY}$ bit. Interrupt service routines (ISRs) must clear their interrupt’s corresponding $\text{STKY}$ bit so the processor can detect a reoccurrence of the condition. For example, an ISR for a floating-point underflow exception interrupt ($\text{FLTUI}$) clears the $\text{AUS}$ bit in the $\text{STKY}$ register near the beginning of the routine. 

Figure A-4, Figure A-5, and Table A-7 provide bit information for both the $\text{STKYx}$ and $\text{STKYy}$ registers.

Figure A-4. STKYx Register
Figure A-5. STKYy Register

Table A-7. STKYx and STKYy Register Bit Descriptions (RW)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AUS</td>
<td>ALU Floating-Point Underflow. A sticky indicator for the ALU AS bit. For more information, see &quot;AZ&quot; on page A-17.</td>
</tr>
<tr>
<td>1</td>
<td>AVS</td>
<td>ALU Floating-Point Overflow. A sticky indicator for the ALU AV bit. For more information, see &quot;AV&quot; on page A-17.</td>
</tr>
<tr>
<td>2</td>
<td>AOS</td>
<td>ALU Fixed-Point Overflow. A sticky indicator for the ALU AV bit. For more information, see &quot;AV&quot; on page A-17.</td>
</tr>
<tr>
<td>4–3</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>AIS</td>
<td>ALU Floating-Point Invalid Operation. A sticky indicator for the ALU AI bit. For more information, see &quot;AI&quot; on page A-18.</td>
</tr>
<tr>
<td>6</td>
<td>MOS</td>
<td>Multiplier Fixed-Point Overflow. A sticky indicator for the multiplier MV bit. For more information, see &quot;MV&quot; on page A-18.</td>
</tr>
<tr>
<td>7</td>
<td>MVS</td>
<td>Multiplier Floating-Point Overflow. A sticky indicator for the multiplier MV bit. For more information, see &quot;MV&quot; on page A-18.</td>
</tr>
</tbody>
</table>
### Table A-7. STKYx and STKYy Register Bit Descriptions (RW) (Cont’d)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>MUS</td>
<td>Multiplier Floating-Point Underflow. A sticky indicator for the multiplier MU bit. For more information, see “MU” on page A-19.</td>
</tr>
<tr>
<td>9</td>
<td>MIS</td>
<td>Multiplier Floating-Point Invalid Operation. A sticky indicator for the multiplier MI bit. For more information, see “MI” on page A-19.</td>
</tr>
<tr>
<td>16–10</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>17 (RW)</td>
<td>CB7S</td>
<td>DAG1 Circular Buffer 7 Overflow. Indicates if a circular buffer being addressed with DAG1 register I7 has overflowed (if set, = 1) or has not overflowed (if cleared, = 0). A circular buffer overflow occurs when DAG circular buffering operation increments the I register past the end of buffer.</td>
</tr>
<tr>
<td>18 (RW)</td>
<td>CB15S</td>
<td>DAG2 Circular Buffer 15 Overflow. Indicates if a circular buffer being addressed with DAG2 register I15 has overflowed (if set, = 1) or has not overflowed (if cleared, = 0). A circular buffer overflow occurs when DAG circular buffering operation increments the I register past the end of buffer.</td>
</tr>
<tr>
<td>19 (RW)</td>
<td>IIRA</td>
<td>Illegal IOP Register Access. Indicates if set (= 1) the core had accessed the IOP register space or not.</td>
</tr>
<tr>
<td>20 (RW)</td>
<td>U64MA</td>
<td>Unaligned 64-Bit Memory Access. Indicates if set (= 1) if a forced Normal word access (LW mnemonic) addressing an uneven memory address has occurred or has not occurred (if 0).</td>
</tr>
<tr>
<td>21 (RO)</td>
<td>PCFL</td>
<td>PC Stack Full. Indicates if the PC stack is full (if 1) or not full (if 0)—Not a sticky bit, cleared by a Pop.</td>
</tr>
<tr>
<td>22 (RO)</td>
<td>PCEM</td>
<td>PC Stack Empty. Indicates if the PC stack is empty (if 1) or not empty (if 0)—Not sticky, cleared by a Push.</td>
</tr>
<tr>
<td>23 (RO)</td>
<td>SSOV</td>
<td>Status Stack Overflow. Indicates if the status stack is overflowed (if 1) or not overflowed (if 0)—sticky bit.</td>
</tr>
<tr>
<td>24 (RO)</td>
<td>SSEM</td>
<td>Status Stack Empty. Indicates if the status stack is empty (if 1) or not empty (if 0)—not sticky, cleared by a Push. Set by default</td>
</tr>
<tr>
<td>25 (RO)</td>
<td>LSOV</td>
<td>Loop Stack Overflow. Indicates if the loop counter stack and loop stack are overflowed (if 1) or not overflowed (if 0)—sticky bit.</td>
</tr>
</tbody>
</table>
Data Address Generator Registers

The processor’s data address generator (DAG) registers (RW) hold data addresses, modify values, and circular buffer configurations. Using these registers, the DAGs can automatically increment addressing for ranges of data locations (a buffer). Each set of DAG registers has a set of background registers. These registers are selected using bits 6–3 in the MODE1 register. For more information, see “Alternate (Secondary) DAG Registers” on page 6-28.

Index Registers (Ix)

The DAGs store addresses in index registers (I0–I7 for DAG1 and I8–I15 for DAG2). An index register holds an address and acts as a pointer to a memory location.

Modify Registers (Mx)

The DAGs update stored addresses using modify registers (M0–M7 for DAG1 and M8–M15 for DAG2). A modify register provides the increment or step size by which an index register is pre- or post-modified during a register move.

Table A-7. STKYx and STKYy Register Bit Descriptions (RW) (Cont’d)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>26 (RO)</td>
<td>LSEM</td>
<td>Loop Stack Empty. Indicates if the loop counter stack and loop stack are empty (if 1) or not empty (if 0)—not sticky, cleared by a Push. Set by default</td>
</tr>
<tr>
<td>31–27</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>

LSEM (Loop Stack Empty)

Indicates if the loop counter stack and loop stack are empty (if 1) or not empty (if 0)—not sticky, cleared by a Push. Set by default
Length and Base Registers (Lx, Bx)

The DAGs control circular buffering operations with length and base registers (L0–L7 and B0–B7 for DAG1 and L8–L15 and B8–B15 for DAG2). Length and base registers set up the range of addresses and the starting address for a circular buffer.

Alternate DAG Registers (Ix',Mx',Lx',Bx')

The processor includes alternate register sets for all DAG registers to facilitate fast context switching. Bits in the MODE1 register ("Mode Control 1 Register (MODE1)" on page A-3) control when alternate registers become accessible. While inaccessible, the contents of alternate registers are not affected by processor operations. Note that there is a one cycle latency between writing to MODE1 and being able to access an alternate register set.

For more information, see "Alternate (Secondary) DAG Registers" on page 6-28.

Miscellaneous Registers

The following sections provide descriptions of the miscellaneous registers.

Bus Exchange Register (PX)

The PM bus exchange (PX) register (RW) permits data to flow between the PM and DM data buses. The PX register can work as one 64-bit register or as two 32-bit registers (PX1 and PX2). The PX1 register is the lower 32 bits of the PX register and PX2 is the upper 32 bits of PX.

The PX register lets programs transfer data between the data buses, but cannot be an input or output in a calculation.
Universal Register Effect Latency

For more information, see Chapter 3, Data Bus Exchange.

User-Defined Status Registers (USTATx)

The USTATx registers (RW) are user-defined, general-purpose status registers. Programs can use these 32-bit registers with bit-wise instructions (SET, CLEAR, TEST, and others). Often, programs use these registers for low overhead, general-purpose flags or for temporary 32-bit storage of data.

Emulation Counter Registers (EMUCLKx)

These registers are read-only from user-space and can be written only when the processor is in emulation space.

The emulation clock counter consists of a 32-bit count register (EMUCLK) and a 32-bit scaling register (EMUCLK2). The EMUCLK counts core clock cycles while the user has control of the processor and stops counting when the emulator gains control. These registers let you gauge the amount of time spent executing a particular section of code. The EMUCLK2 register extends the time EMUCLK can count by incrementing each time the EMUCLK value rolls over to zero. The combined emulation clock counter can count accurately for thousands of hours. Note that the counters increment during an idle instruction.

Universal Register Effect Latency

Writes to some of the universal registers (UREG) do not take effect immediately. For example, if a program writes to the MODE1 register in order to set ALU saturation mode, any ALU operation in the instruction immediately following is not effected. The saturation mode takes effect in the second instruction following the instruction performing the write to MODE1. This is referred to as an effect latency of one cycle. Also, some registers are not updated on the cycle immediately following a write. It takes an extra cycle
before a read of the register returns the updated value. This is referred to as a *read latency* of one cycle.

Note that the effect latency and read latency are counted in a number of processor cycles rather than instruction cycles. Therefore, there may be situations when the effect latency may not be observed, such as when the pipeline stalls or when an interrupt breaks the normal sequence of instructions. Here, the effect latency and the read latency are interpreted as the maximum number of instructions, which is unaffected by the new settings after a write to one register.

In the SHARC 5-stage pipeline products, effect latencies were intentionally added in direct core writes to various registers for backward compatibility to the 3-stage pipeline products (though these latencies are not necessitated by the architecture as such). In some cases it is done by adding stall(s) to the pipeline, whereas in other cases, the execution (actual write-back to concerned registers) is delayed.

*Table A-8* and *Table A-9* summarize the number of extra cycles (latency) for a write to take effect (effect latency) and for a new value to appear in the register (read latency). A 0 (zero) indicates that the write takes effect or appears in the register on the next cycle after the write instruction is executed, and a 1 indicates one extra cycle.

**Table A-8. UREG Read and Effect Latencies**

<table>
<thead>
<tr>
<th>Register</th>
<th>Contents</th>
<th>Bits</th>
<th>Read Latency</th>
<th>Effect Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADDR</td>
<td>Fetch address</td>
<td>24</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>DADDR</td>
<td>Decode address</td>
<td>24</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>PC</td>
<td>Execute address</td>
<td>24</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>PCSTK</td>
<td>Top of PC stack</td>
<td>24</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PCSTKP</td>
<td>PC stack pointer</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LADDR</td>
<td>Top of loop address stack</td>
<td>32</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
**Universal Register Effect Latency**

Table A-8. UREG Read and Effect Latencies (Cont’d)

<table>
<thead>
<tr>
<th>Register</th>
<th>Contents</th>
<th>Bits</th>
<th>Read Latency</th>
<th>Effect Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURLCNTR</td>
<td>Top of loop count stack (current loop count)</td>
<td>32</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LCNTR</td>
<td>Loop count for next DO UNTIL loop</td>
<td>32</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table A-9. SREG Read and Effect Latencies

<table>
<thead>
<tr>
<th>Register</th>
<th>Contents</th>
<th>Bits</th>
<th>Read Latency</th>
<th>Effect Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODE1</td>
<td>Mode control bits</td>
<td>32</td>
<td>0</td>
<td>1 for internal access 2 for external data access</td>
</tr>
<tr>
<td>MODE2</td>
<td>Mode control bits</td>
<td>32</td>
<td>0</td>
<td>1 for internal access 2 for external data access</td>
</tr>
<tr>
<td>IRPTL</td>
<td>Interrupt latch</td>
<td>32</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>IMASK</td>
<td>Interrupt mask</td>
<td>32</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>IMASKP</td>
<td>Interrupt mask pointer (for nesting)</td>
<td>32</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MMASK</td>
<td>Mode mask</td>
<td>32</td>
<td>0</td>
<td>1 for internal access 2 for external data access</td>
</tr>
<tr>
<td>FLAGS</td>
<td>Flag inputs</td>
<td>32</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>LIRPTL2</td>
<td>Interrupt latch/mask</td>
<td>32</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ASTATx</td>
<td>Arithmetic status flags</td>
<td>32</td>
<td>0</td>
<td>1 for internal access 2 for external data access</td>
</tr>
<tr>
<td>ASTATy</td>
<td>Arithmetic status flags</td>
<td>32</td>
<td>0</td>
<td>1 for internal access 2 for external access</td>
</tr>
<tr>
<td>STKYx</td>
<td>Sticky status flags</td>
<td>32</td>
<td>0</td>
<td>1 for internal access 2 for external data access</td>
</tr>
<tr>
<td>STKYy</td>
<td>Sticky status flags</td>
<td>32</td>
<td>0</td>
<td>1 for internal access 2 for external data access</td>
</tr>
<tr>
<td>USTAT1</td>
<td>User-defined status flags</td>
<td>32</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Table A-9. SREG Read and Effect Latencies  (Cont’d)

<table>
<thead>
<tr>
<th>Register</th>
<th>Contents</th>
<th>Bits</th>
<th>Read Latency</th>
<th>Effect Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>USTAT2</td>
<td>User-defined status flags</td>
<td>32</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>USTAT3</td>
<td>User-defined status flags</td>
<td>32</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>USTAT4</td>
<td>User-defined status flags</td>
<td>32</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

1 All bits except CAFRZ, U64MAE, IIARE have one cycle of effect latency.
2 Bits 29-20 are the various mask pointer bits. These bits have one cycle of read latency. Other bits do not have read latency.

The following examples provide more detail on latency.

- The contents of the MODE1 and MODE2 registers are used in the decode stage of the instruction pipeline. To maintain the same effect latency of one cycle, a stall cycle is always added after a write to the MODE1 or MODE2 registers. A stall is also introduced when the contents of the MODE1 and MODE2 registers are modified through a bit manipulation instruction. The MODE1 register value also changes when the PUSH STS or POP STS instructions are executed or when the sequencer branches to, or returns from an ISR (interrupt service routine) which involves a PUSH/POP of the stack. This results in a one cycle stall.

```c
MODE1 = 0x1;    /* enable bit reverse addressing for I8 */
PM(I8,MB) = R14;  /* stalls for a cycle, but unaffected by mode setting */
PM(I8,MB) = R14;  /* performs bit reversed mode of addressing */
```

- When the contents of the ASTAT registers are updated by any operation other than a compute operation, the following instruction stalls for a cycle, if it performs a conditional branch and the condition is anything other than NOT LCE. An example is when ASTAT is explicitly loaded or when the sequencer branches to, or returns from an ISR involving a PUSH/POP of the status stack.
Universal Register Effect Latency

- The effect latency in the case of a \texttt{FLAGS} register is felt when a conditional instruction dependent on the \texttt{FLAGS} register values is executed after modifications to the \texttt{FLAGS} register.

\begin{verbatim}
BIT SET FLAGS 0x1;     /* set FLAG0 */
IF FLAG0 IN R0 = R0+1; /* conditional compute - aborts */
IF FLAG0 IN R0 = R0+1; /* conditional compute - executes */
\end{verbatim}

A stall cycle is introduced after a write to the \texttt{FLAGS} register, only if a conditional branch dependent on the \texttt{FLAGS} register settings follows it as the second instruction.

\begin{verbatim}
BIT SET FLAGS 0x1;     /* set FLAG0 */
IF FLAG0 IN R0 = R0+1; /* unaffected by prior instruction-aborts */
IF FLAG0 IN RTS;       /* stalls a cycle and executes RTS */
\end{verbatim}

- A stall cycle results after a write to the \texttt{ASTATx} or \texttt{ASTATy} registers, only if a conditional branch follows it as the second instruction.

\begin{verbatim}
ASTATX = 0x1;           /* set AZ flag */
IF NE JUMP(SOMEWHERE);  /* unaffected by prior instruction-aborts */
IF NE RTS;              /* stalls a cycle and executes RTS */
\end{verbatim}

- The following registers that normally have an effect latency of 1 cycle will have an effect latency of 2 cycles if any of their bits impact an instruction containing an external data access: \texttt{MODE1}, \texttt{MODE2}, \texttt{MMASK}, \texttt{ASTATx}, \texttt{ASTATy}, \texttt{STKYx}, and \texttt{STKYy}. 
Interrupt Registers

This section provides information on the registers that are used to configure and control interrupts.

Interrupt Latch Register (IRPTL)

The IRPTL register indicates latch status for interrupts. Figure A-6 and Table A-10 provide bit definitions for the IRPTL register.

The programmable interrupt latch bits (P0I-P5I, P14I-P16I) are controlled through the priority interrupt control registers (PICR). The descriptions provided are their default source. For information on their optional use, see “Programmable Interrupt Control Registers (PICRx)” in the processor specific hardware reference.

Interrupt Mask Register (IMASK)

Each bit in the IMASK register corresponds to a bit with the same name in the IRPTL registers. The bits in the IMASK register unmask (enable if set, = 1), or mask (disable if cleared, = 0) the interrupts that are latched in the IRPTL register. Except for the RSTI and EMUI bits, all interrupts are maskable.

When the IMASK register masks an interrupt, the masking disables the processor’s response to the interrupt. The IRPTL register still latches an interrupt even when masked, and the processor responds to that latched interrupt if it is later unmasked. Figure A-6 and Table A-10 provide bit definitions for the IMASK register.

Interrupt Mask Pointer Register (IMASKP)

Each bit in the IMASKP register corresponds to a bit with the same name in the IRPTL registers. The IMASKP register field descriptions are shown in
Interrupt Registers

Figure A-6, and described in Table A-10. Shaded cells indicate user programmable interrupts.

This register supports an interrupt nesting scheme that lets higher priority events interrupt an ISR and keeps lower priority events from interrupting.

When interrupt nesting is enabled, the bits in the IMASKP register mask interrupts that have a lower priority than the interrupt that is currently being serviced. Other bits in this register unmask interrupts having higher priority than the interrupt that is currently being serviced. Interrupt nesting is enabled using NESTM in the MODE1 register. The IRPTL register latches a lower priority interrupt even when masked, and the processor responds to that latched interrupt if it is later unmasked.

When interrupt nesting is disabled (NESTM = 0 in the MODE1 register), the bits in the IMASKP register mask all interrupts while an interrupt is currently being serviced. The IRPTL register still latches these interrupts even when masked, and the processor responds to the highest priority latched interrupt after servicing the current interrupt.
Figure A-6. IRPTL, IMASK, and IMASKP Registers


<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (RO)</td>
<td>EMUI</td>
<td><strong>Emulator Interrupt.</strong> An EMUI occurs when the external emulator triggers an interrupt or the core hits a emulator breakpoint. Note this interrupt has highest priority, it is read-only and non-maskable.</td>
</tr>
<tr>
<td>1 (RO)</td>
<td>RSTI</td>
<td><strong>Reset Interrupt.</strong> An RSTI occurs as an external device asserts the RESET pin or after a software reset (SYSCTL register). Note this interrupt is read-only and non-maskable.</td>
</tr>
<tr>
<td>2</td>
<td>IICDI</td>
<td><strong>Illegal Input Condition Detected Interrupt.</strong> An IICDI occurs when a TRUE results from the logical OR’ing of the illegal I/O processor register access (IIRA) and unaligned 64-bit memory access bits in the STKYx register.</td>
</tr>
<tr>
<td>3</td>
<td>SOVFI</td>
<td><strong>Stack Overflow/Full Interrupt.</strong> A SOVFI occurs when a stack in the program sequencer overflows or is full.</td>
</tr>
<tr>
<td>4</td>
<td>TMZHI</td>
<td><strong>Core Timer Expired High Priority.</strong> A TMZHI occurs when the timer decrements to zero. Note that this event also triggers a TMZLI. Since the timer expired event (TCOUNT decrements to zero) generates two interrupts, TMZHI and TMZLI, programs should unmask the timer interrupt with the desired priority and leave the other one masked.</td>
</tr>
<tr>
<td>5</td>
<td>SPERRI¹</td>
<td><strong>Sport Error Interrupt.</strong> A SPERRI occurs on a FIFO underflow/overflow or a frame sync error.</td>
</tr>
<tr>
<td>6</td>
<td>BKPI</td>
<td><strong>Hardware Breakpoint Interrupt.</strong> When the processor is servicing another interrupt, indicates if the BKPI interrupt is unmasked (if set, = 1), or masked (if cleared, = 0).</td>
</tr>
<tr>
<td>7</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>IRQ2I</td>
<td><strong>Hardware Interrupt.</strong> An IRQ2I occurs when an external device asserts the FLAG2 pin configured as TRST. The IRQ2E bit (MODE2) defines if interrupt latched on edge or level.</td>
</tr>
<tr>
<td>9</td>
<td>IRQ1I</td>
<td><strong>Hardware Interrupt.</strong> An IRQ1I occurs when an external device asserts the FLAG2 pin configured as TRST. The IRQ1E bit (MODE2) defines if interrupt latched on edge or level.</td>
</tr>
</tbody>
</table>
Table A-10. IRPTL, IMASK, IMASKP Register Bit Descriptions (RW)  
(Cont’d)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>IRQ0I</td>
<td>Hardware Interrupt. An IRQ0I occurs when an external device asserts the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FLAG2 pin configured as IRQ0. The IRQ0E bit (MODE2) defines if interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>latched on edge or level.</td>
</tr>
<tr>
<td>11</td>
<td>P0I</td>
<td>Programmable Interrupt 0. A P0I interrupt occurs when the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>default/programmed peripheral sets (= 1) this bit.</td>
</tr>
<tr>
<td>12</td>
<td>P1I</td>
<td>Programmable Interrupt 1. See P0I</td>
</tr>
<tr>
<td>13</td>
<td>P2I</td>
<td>Programmable Interrupt 2. See P0I</td>
</tr>
<tr>
<td>14</td>
<td>P3I</td>
<td>Programmable Interrupt 3. See P0I</td>
</tr>
<tr>
<td>15</td>
<td>P4I</td>
<td>Programmable Interrupt 4. See P0I</td>
</tr>
<tr>
<td>16</td>
<td>P5I</td>
<td>Programmable Interrupt 5. See P0I</td>
</tr>
<tr>
<td>17</td>
<td>P14I</td>
<td>Programmable Interrupt 14. See P0I</td>
</tr>
<tr>
<td>18</td>
<td>P15I</td>
<td>Programmable Interrupt 15. See P0I</td>
</tr>
<tr>
<td>19</td>
<td>P16I</td>
<td>Programmable Interrupt 16. See P0I</td>
</tr>
<tr>
<td>20</td>
<td>CB7I</td>
<td>DAG1 Circular Buffer 7 Overflow Interrupt. A circular buffer overflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>occurs when the DAG circular buffering operation increments the I7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>register past the end of the buffer.</td>
</tr>
<tr>
<td>21</td>
<td>CB15I</td>
<td>DAG2 Circular Buffer 15 Overflow Interrupt. A circular buffer overflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>occurs when the DAG circular buffering operation increments the I15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>register past the end of the buffer.</td>
</tr>
<tr>
<td>22</td>
<td>TMZLI</td>
<td>Core Timer Expired (Low Priority) Interrupt. A TMZLI occurs when</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the timer decrements to zero. (Refer to TMZHI)</td>
</tr>
<tr>
<td>23</td>
<td>FIXI</td>
<td>Fixed-Point Overflow Interrupt. Refer to the status registers for the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>execution units (ASTATx/y, STKYx/y).</td>
</tr>
<tr>
<td>24</td>
<td>FLTOI</td>
<td>Floating-Point Overflow Interrupt. Refer to the status registers for the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>execution units (ASTATx/y, STKYx/y).</td>
</tr>
<tr>
<td>25</td>
<td>FLTUI</td>
<td>Floating-Point Underflow Interrupt. Refer to the status registers for the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>execution units (ASTATx/y, STKYx/y).</td>
</tr>
</tbody>
</table>
Interrupt Registers

The *LIRPTL* register indicates latch status, select masking, and displays mask pointers for interrupts. Figure A-7 and Table A-11 provide bit definitions for the *LIRPTL* register.

![](image)

**The MSKP bits in the LIRPTL register, and the entire IMASKP register are for interrupt controller use only. Modifying these bits interferes with the proper operation of the interrupt controller.**

The programmable interrupt latch bits (P6I–P13I, P17I, P18I) are controlled through the programmable interrupt controller registers (*PICRx*). The descriptions provided are their default source. For information on their optional use, see “Programmable Interrupt Priority Control Registers” in the product related hardware reference.

---

Table A-10. IRPTL, IMASK, IMASKP Register Bit Descriptions (RW) (Cont’d)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>FLTII</td>
<td>Floating-Point Invalid Operation Interrupt. Refer to the status registers for the execution units (ASTATx/y, STKYx/y).</td>
</tr>
<tr>
<td>27</td>
<td>EMULI</td>
<td>Emulator Low Priority Interrupt. An EMULI occurs during Background telemetry channels (BTC). This interrupt has a lower priority than EMUI, but higher priority than software interrupts.</td>
</tr>
<tr>
<td>28</td>
<td>SFT0I</td>
<td>User Software Interrupt 0. An SFT0I occurs when a program sets (= 1) this bit.</td>
</tr>
<tr>
<td>29</td>
<td>SFT1I</td>
<td>User Software Interrupt 1. See SFT01.</td>
</tr>
<tr>
<td>30</td>
<td>SFT2I</td>
<td>User Software Interrupt 2. See SFT01.</td>
</tr>
<tr>
<td>31</td>
<td>SFT3I</td>
<td>User Software Interrupt 3. See SFT01. Lowest priority.</td>
</tr>
</tbody>
</table>

1 The SPERRI interrupt (bit 5) is reserved for ADSP-21362/3/4/5/6 SHARC processors.
Figure A-7. LIRPTL Register
# Interrupt Registers

## Table A-11. LIRPTL Register Bit Descriptions (RW)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>P6I</td>
<td>Programmable Interrupt 6.</td>
</tr>
<tr>
<td>1</td>
<td>P7I</td>
<td>Programmable Interrupt 7</td>
</tr>
<tr>
<td>2</td>
<td>P8I</td>
<td>Programmable Interrupt 8</td>
</tr>
<tr>
<td>3</td>
<td>P9I</td>
<td>Programmable Interrupt 9</td>
</tr>
<tr>
<td>4</td>
<td>P10I</td>
<td>Programmable Interrupt 10</td>
</tr>
<tr>
<td>5</td>
<td>P11I</td>
<td>Programmable Interrupt 11</td>
</tr>
<tr>
<td>6</td>
<td>P12I</td>
<td>Programmable Interrupt 12</td>
</tr>
<tr>
<td>7</td>
<td>P13I</td>
<td>Programmable Interrupt 13</td>
</tr>
<tr>
<td>8</td>
<td>P17I</td>
<td>Programmable Interrupt 17</td>
</tr>
<tr>
<td>9</td>
<td>P18I</td>
<td>Programmable Interrupt 18</td>
</tr>
<tr>
<td>10</td>
<td>P6IMSK</td>
<td>Programmable Interrupt Mask 6. Unmasks the P6I interrupt (if set, = 1), or masks the P6I interrupt (if cleared, = 0).</td>
</tr>
<tr>
<td>11</td>
<td>P7IMSK</td>
<td>Programmable Interrupt Mask 7. See P6IMSK.</td>
</tr>
<tr>
<td>12</td>
<td>P8IMSK</td>
<td>Programmable Interrupt Mask 8. See P6IMSK.</td>
</tr>
<tr>
<td>13</td>
<td>P9IMSK</td>
<td>Programmable Interrupt Mask 9. See P6IMSK.</td>
</tr>
<tr>
<td>14</td>
<td>P10IMSK</td>
<td>Programmable Interrupt Mask 9. See P6IMSK.</td>
</tr>
<tr>
<td>15</td>
<td>P11IMSK</td>
<td>Programmable Interrupt Mask 11. See P6IMSK.</td>
</tr>
<tr>
<td>16</td>
<td>P12IMSK</td>
<td>Programmable Interrupt Mask 12. See P6IMSK.</td>
</tr>
<tr>
<td>17</td>
<td>P13IMSK</td>
<td>Programmable Interrupt Mask 12. See P6IMSK.</td>
</tr>
<tr>
<td>18</td>
<td>P17IMSK</td>
<td>Programmable Interrupt Mask 17. See P6IMSK.</td>
</tr>
<tr>
<td>19</td>
<td>P18IMSK</td>
<td>Programmable Interrupt Mask 18. See P6IMSK.</td>
</tr>
<tr>
<td>20</td>
<td>P6IMSKP</td>
<td>Programmable Interrupt Mask Pointer 9. When the processor is servicing another interrupt, indicates if the P6I interrupt is unmasked (if set, = 1), or the P6I interrupt is masked (if cleared, = 0).</td>
</tr>
</tbody>
</table>
Mode Mask Register (MMASK)

Each bit in the MMASK register corresponds to a bit in the MODE1 register. Bits that are set in the MMASK register are used to clear bits in the MODE1 register when the processor’s status stack is pushed. This effectively disables different modes upon servicing an interrupt, or when executing a PUSH STS instruction. Figure A-8 provides bit information for the MMASK register.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>P7IMSKP</td>
<td>Programmable Interrupt Mask Pointer 7. See P6IMSKP.</td>
</tr>
<tr>
<td>22</td>
<td>P8IMSKP</td>
<td>Programmable Interrupt Mask Pointer 8. See P6IMSKP.</td>
</tr>
<tr>
<td>23</td>
<td>P9IMSKP</td>
<td>Programmable Interrupt Mask Pointer 9. See P6IMSKP.</td>
</tr>
<tr>
<td>24</td>
<td>P10IMSKP</td>
<td>Programmable Interrupt Mask Pointer 10. See P6IMSKP.</td>
</tr>
<tr>
<td>25</td>
<td>P11IMSKP</td>
<td>Programmable Interrupt Mask Pointer 11. See P6IMSKP.</td>
</tr>
<tr>
<td>26</td>
<td>P12IMSKP</td>
<td>Programmable Interrupt Mask Pointer 12. See P6IMSKP.</td>
</tr>
<tr>
<td>27</td>
<td>P13IMSKP</td>
<td>Programmable Interrupt Mask Pointer 13. See P6IMSKP.</td>
</tr>
<tr>
<td>28</td>
<td>P17IMSKP</td>
<td>Programmable Interrupt Mask Pointer 17. See P6IMSKP.</td>
</tr>
<tr>
<td>29</td>
<td>P18IMSKP</td>
<td>Programmable Interrupt Mask Pointer 18. See P6IMSKP.</td>
</tr>
<tr>
<td>31–30</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>
Interrupt Registers

Figure A-8. MMASK Register Bits (RW)
Memory-Mapped Registers

This section describes all IOP core registers which are memory mapped in the core clock domain.

System Control Register (SYSCTL)

The SYSCTL register as it relates to the processor core configures memory use and interrupts. For SYSCTL use as it applies to pin multiplexing, see the product specific hardware reference. Bit descriptions for this register are shown in Figure A-9 and described in Table A-12.

The SYSCTL register has an effect latency of 1 cycle. If a program writes to the SYSCTL or BRKCTL register before it access external memory it must perform at least two non external access before the external access.

Figure A-9. SYSCTL Register
## Memory-Mapped Registers

Table A-12. SYSCTL Register Bit Descriptions (RW)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0   | SRST     | **Software Reset.** When set, this bit resets the processor and the processor responds to the non-maskable RSTI interrupt and clears (=0) SRST. Unlike the HW reset, the PLL and Power Management (PMCTL register) are not reset. The part does also boot after SW reset. After one core clock cycle, the registers are put in the default settings (effect latency). The RESETOUT pin is asserted for 2 PCLK cycles.  
0 = No software reset  
1 = Software reset                                                |
| 1   | Reserved |                                                                             |
| 2   | IIVT     | **Internal Interrupt Vector Table.** If bit set (=1), IVT starts at internal RAM address, if cleared (=0) at internal ROM address. The default IIVT bit setting is enabled (=1) with any valid boot mode (BOOT_CFGx pins). If the reserved boot mode is selected, IVT bit is cleared (= 0). |
| 6–3 | Reserved |                                                                             |
| 7   | DCPR     | **DMA Channel Priority Rotating.** This bit enables or disables priority rotation among DMA channels on the DMA peripheral bus (IOD or IOD0). Permits core writes.  
0 = Arbiter uses fixed priority  
1 = Arbiter uses rotating priority                                  |
| 8   | Reserved |                                                                             |
| 9   | IMDW0    | **Internal Memory Data Width 0.** Selects the data access size for internal memory block0 as 48- or 32-bit data. Permits core writes.  
0 = Data bus width is 32 bits  
1 = Data bus width is 48 bits                                         |
| 10  | IMDW1    | **Internal Memory Data Width 1.** Selects the data access size for internal memory block1 as 48- or 32-bit data. Permits core writes.  
0 = Data bus width is 32 bits  
1 = Data bus width is 48 bits                                         |
| 11  | IMDW2    | **Internal Memory Data Width 2.** Selects the data access size for internal memory block2 as 48- or 32-bit data. Permits core writes.  
0 = Data bus width is 32 bits  
1 = Data bus width is 48 bits                                         |
Revision ID Register (REVPI D)

The REVPID register described in Table A-13 is a top layer metal programmable 8-bit register. Because these bits are the processor ID and silicon revision, the reset value varies with the system setting and silicon revision. That is, the value in top-level metal layer changes. This register is useful for conditional code execution based on the processor’s ID and silicon revision numbers.

The REVPI D coding is available available on the SHARC product pages on the Analog Devices web site.

Breakpoint Control Register (BRKCTL)

The BRKCTL register is a 32-bit memory-mapped I/O register. To enabled user breakpoints UMODE bit (bit 25) is set. On occurrence of a valid breakpoint hit, a low prioritization interrupt (BKPI) vectors to the ISR. The
Memory-Mapped Registers

enhanced emulation status register \texttt{EEMUSTAT} indicates which breakpoint hit occurred, all the breakpoint status bits are cleared when the program exits the ISR with an RTI instruction. Such interrupts may contain error handling if the processor accesses any of the addresses in the address range defined in the breakpoint registers. The bit settings for these registers are shown in Figure A-10 and described in Table A-14.

![Figure A-10. BRKCTL Register](image)

A-44 SHARC Processor Programming Reference
### Table A-14. BRKCTL Register Bit Descriptions (RW)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 1–0   | PA1MODE      | PA1 Triggering Mode.  
00 = Breakpoint disabled  
01 = WRITE access  
10 = READ access  
11 = Any access |
| 3–2   | DA1MODE      | DA1 Triggering Mode.  
00 = Breakpoint disabled  
01 = WRITE access  
10 = READ access  
11 = Any access |
| 5–4   | DA2MODE      | DA2 Triggering Mode.  
00 = Breakpoint disabled  
01 = WRITE access  
10 = READ access  
11 = Any access |
| 7–6   | IO1MODE      | I/O DMA Triggering Mode.  
00 = Breakpoint is disabled  
01 = WRITE accesses only  
10 = READ accesses only  
11 = Any access |
| 9–8   | Reserved     | -                                                                           |
| 10    | NEGPA1       | Negate Program Memory Data Address Breakpoint.  
Enable breakpoint events if the address is greater than the end  
register value OR less than the start register value. This func-  
tion is useful to detect index range violations in user code.  
0 = Do not negate breakpoint  
1 = Negate breakpoint |
| 11    | NEGDA1       | Negate Data Memory Address Breakpoint #1.  
For more information, see NEGPA1 bit description. |
| 12    | NEGDA2       | Negate Data Memory Address Breakpoint #2.  
For more information, see NEGPA1 bit description. |
| 13    | NEGIA1       | Negate Instruction Address Breakpoint #1.  
0 = Do not negate breakpoint  
1 = Negate breakpoint |
## Memory-Mapped Registers

### Table A-14. BRKCTL Register Bit Descriptions (RW) (Cont’d)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>NEGIA2</td>
<td>Negate Instruction Address Breakpoint #2.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information, see NEGPA1 bit description.</td>
</tr>
<tr>
<td>15</td>
<td>NEGIA3</td>
<td>Negate Instruction Address Breakpoint #3.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information, see NEGPA1 bit description.</td>
</tr>
<tr>
<td>16</td>
<td>NEGIA4</td>
<td>Negate Instruction Address Breakpoint #4.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information, see NEGPA1 bit description.</td>
</tr>
<tr>
<td>17</td>
<td>NEGIO1</td>
<td>Negate I/O Address Breakpoint.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information, see NEGPA1 bit description.</td>
</tr>
<tr>
<td>18</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>ENBPA</td>
<td>Enable Program Memory Data Address Breakpoints.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The ENB bits enable each breakpoint group. Note that when the ANDBKP bit is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>set, breakpoint types not involved in the generation of the effective</td>
</tr>
<tr>
<td></td>
<td></td>
<td>breakpoint must be disabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = Disable breakpoints</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Enable breakpoints</td>
</tr>
<tr>
<td>20</td>
<td>ENBDA</td>
<td>Enable Data Memory Address Breakpoints.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information, see ENBPA bit description.</td>
</tr>
<tr>
<td>21</td>
<td>ENBIA</td>
<td>Enable Instruction Address Breakpoints.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information, see ENBPA bit description.</td>
</tr>
<tr>
<td>23–22</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>ANDBKP</td>
<td>AND Composite Breakpoints.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Enables logical AND of each breakpoint type to generate an effective</td>
</tr>
<tr>
<td></td>
<td></td>
<td>breakpoint from the composite breakpoint signals.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = OR Breakpoint types</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = AND Breakpoint types</td>
</tr>
<tr>
<td>25</td>
<td>UMODE</td>
<td>User Mode Breakpoint Functionality Enable. Address Breakpoint 3.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = Disable user controlled breakpoint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Enable user controlled breakpoint</td>
</tr>
<tr>
<td>26</td>
<td>ENBIO1</td>
<td>IOD1 (EP DMA Bus) Breakpoint Enable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = Disable IOD1 breakpoint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Enable IOD1 breakpoint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(reserved for ADSP-21362/3/4/5/6 processors)</td>
</tr>
</tbody>
</table>
**Enhanced Emulation Status Register (EEMUSTAT)**

The EEMUSTAT register reports the breakpoint status of the programs that run on the SHARC processors. This register is a memory-mapped IOP register that can be accessed by the core. The bit settings for these registers are shown in Figure A-11.

![Figure A-11. EEMUSTAT Register](image)

Table A-14. BRKCTL Register Bit Descriptions (RW) (Cont’d)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>ENBIO0</td>
<td>IOD0 (Peripheral DMA Bus) Breakpoint Enable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = Disable IOD0 breakpoint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Enable IOD0 breakpoint</td>
</tr>
<tr>
<td>31–28</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>

SHARC Processor Programming Reference
## Memory-Mapped Registers

Table A-15. EEMUSTAT Register Bit Descriptions (RO)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STATPA</td>
<td>Program Memory Data Breakpoint Hit. (^1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = No program memory breakpoint occurs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Program memory breakpoint occurs</td>
</tr>
<tr>
<td>1</td>
<td>STATDA0</td>
<td>Data Memory Breakpoint Hit. (^1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = No data memory #0 breakpoint occurs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Data memory #0 breakpoint occurs</td>
</tr>
<tr>
<td>2</td>
<td>STATDA1</td>
<td>Data Memory Breakpoint Hit. (^1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = No data memory #1 breakpoint occurs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Data memory #1 breakpoint occurs</td>
</tr>
<tr>
<td>3</td>
<td>STATIA0</td>
<td>Instruction Address Breakpoint Hit. (^1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = No instruction address #0 breakpoint occurs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Instruction address #0 breakpoint occurs</td>
</tr>
<tr>
<td>4</td>
<td>STATIA1</td>
<td>Instruction Address Breakpoint Hit. (^1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = No instruction address #1 breakpoint occurs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Instruction address #1 breakpoint occurs</td>
</tr>
<tr>
<td>5</td>
<td>STATIA2</td>
<td>Instruction Address Breakpoint Hit. (^1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = No instruction address #2 breakpoint occurs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Instruction address #2 breakpoint occurs</td>
</tr>
<tr>
<td>6</td>
<td>STATIA3</td>
<td>Instruction Address Breakpoint Hit. (^1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = No instruction address #3 breakpoint occurs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Instruction address #3 breakpoint occurs</td>
</tr>
<tr>
<td>7</td>
<td>STATIO0</td>
<td>DMA Peripheral Address Breakpoint Status. (^1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set bit if breakpoint hit detected on the IOD/IOD0 bus</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = No DMA peripheral address breakpoint occurs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = DMA peripheral address breakpoint occurs</td>
</tr>
<tr>
<td>8</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>EEMUOUTIRQEN</td>
<td>Enhanced Emulation EEMUOUT Interrupt Enable. (^2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = EEMUOUT interrupt disable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = EEMUOUT interrupt enable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note: Interrupts are of the low priority variety</td>
</tr>
<tr>
<td>10</td>
<td>EEMUOUTRDKY</td>
<td>Enhanced Emulation EEMUOUT Ready. (^3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = EEMUOUT FIFO contains valid data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = EEMUOUT FIFO is empty</td>
</tr>
</tbody>
</table>
Table A-15. EEMUSTAT Register Bit Descriptions (RO) (Cont’d)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>EEMUOUTFULL</td>
<td>Enhanced Emulation EEMUOUT FIFO Status.(^3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = EEMUOUT FIFO is not full</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = EEMUOUT FIFO full</td>
</tr>
<tr>
<td>12</td>
<td>EEMUINFULL</td>
<td>Enhanced Emulation EEMUIN Register Status.(^4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = EEMUIN register is empty</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = EEMUIN register full</td>
</tr>
<tr>
<td>13</td>
<td>EEMUENS</td>
<td>Enhanced Emulation Feature Enable.(^5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = Enhanced emulation feature enable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Enhanced emulation feature disable</td>
</tr>
<tr>
<td>14</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>EEMUINENS</td>
<td>EEMUIN Interrupt Enable.(^4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = EEMUIN interrupt disable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = EEMUIN interrupt enable</td>
</tr>
<tr>
<td>16</td>
<td>STATIO1</td>
<td>DMA External Port Address Breakpoint Status.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set bit if breakpoint hit detected on the IOD1 bus (between external port</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and internal memory)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = No external port DMA breakpoint occurs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = External port DMA breakpoint occurs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(reserved for ADSP-21362/3/4/5/6 processors)</td>
</tr>
<tr>
<td>31–17</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>

1  Internal hardware sets this bit.
2  This bit is set and reset by the core.
3  The FIFO controller sets and resets this bit.
4  Internal hardware sets and resets this bit.
Table A-15 lists all available core non memory-mapped registers and their reset values. Table A-17 on page A-52 lists all memory-mapped I/O registers, their reset values and their addresses.

Table A-16. Core Non Memory-Mapped Register Listing

<table>
<thead>
<tr>
<th>Register</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mode</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MODE1</td>
<td>Mode control 1</td>
<td>0x0</td>
<td></td>
</tr>
<tr>
<td>MODE2</td>
<td>Mode control 2</td>
<td>0x4200 0000</td>
<td></td>
</tr>
<tr>
<td><strong>Sequencer</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FADDR</td>
<td>Fetch1 address stage</td>
<td>Undefined</td>
<td></td>
</tr>
<tr>
<td>DADDR</td>
<td>Decode address stage</td>
<td>Undefined</td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td>Execute address stage</td>
<td>Undefined</td>
<td></td>
</tr>
<tr>
<td>PCSTK</td>
<td>PC stack</td>
<td>0xFF FFFF for ADSP-2137x and later: 0x3FF FFFF</td>
<td></td>
</tr>
<tr>
<td>PCSTKP</td>
<td>PC stack pointer</td>
<td>0x0</td>
<td></td>
</tr>
<tr>
<td><strong>Interrupt</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRPTL</td>
<td>Interrupt latch</td>
<td>0x0</td>
<td></td>
</tr>
<tr>
<td>IMASK</td>
<td>Interrupt mask</td>
<td>0x0000 0003</td>
<td></td>
</tr>
<tr>
<td>IMASKP</td>
<td>Interrupt mask pointer</td>
<td>0x0</td>
<td></td>
</tr>
<tr>
<td>LIRPTL</td>
<td>Interrupt latch/mask</td>
<td>0x0</td>
<td></td>
</tr>
<tr>
<td>MMASK</td>
<td>Interrupt mode mask</td>
<td>0x0020 0000</td>
<td></td>
</tr>
</tbody>
</table>
### Registers

**Loop**
- **LADDR** | Loop address | 0xFFFF FFFF
- **LCNTR** | Loop counter | Undefined
- **CURLCNTR** | Current counter | 0xFFFF FFFF

**Timer**
- **TPERIOD** | Timer period | 0x0
- **TCOUNT** | Timer count | 0x0

**GPIO**
- **FLAGS** | GPIO flags | undefined

**Processing Foreground**
- **R15–0** | PEx data file (Fixed/Float) | Undefined
- **S15–0** | PEy data file (Fixed/Float) | Undefined
- **MRF2–0** | PEx Multiply Result (Fixed) | Undefined
- **MSF2–0** | PEy Multiply Result (Fixed) | Undefined

**Processing Background**
- **R’15–0** | PEx data file (Fixed/Float) | Undefined
- **S’15–0** | PEy data file (Fixed/Float) | Undefined
- **MRB2–0** | PEx Multiply Result (Fixed) | Undefined
- **MSB2–0** | PEy Multiply Result (Fixed) | Undefined

**Processing Status**
- **ASTATx** | PEx current status | 0x0
- **ASTATy** | PEy current status | 0x0
- **STKXy** | PEx sticky status | 0x0540 0000
- **STKYy** | PEy sticky status | 0x0540 0000

---

**Table A-16. Core Non Memory-Mapped Register Listing (Cont’d)**

<table>
<thead>
<tr>
<th>Register Mnemonic</th>
<th>Description</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LADDR</td>
<td>Loop address</td>
<td>0xFFFF FFFF</td>
</tr>
<tr>
<td>LCNTR</td>
<td>Loop counter</td>
<td>Undefined</td>
</tr>
<tr>
<td>CURLCNTR</td>
<td>Current counter</td>
<td>0xFFFF FFFF</td>
</tr>
<tr>
<td>Timer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TPERIOD</td>
<td>Timer period</td>
<td>0x0</td>
</tr>
<tr>
<td>TCOUNT</td>
<td>Timer count</td>
<td>0x0</td>
</tr>
<tr>
<td>GPIO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLAGS</td>
<td>GPIO flags</td>
<td>undefined</td>
</tr>
<tr>
<td>Processing Foreground</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R15–0</td>
<td>PEx data file (Fixed/Float)</td>
<td>Undefined</td>
</tr>
<tr>
<td>S15–0</td>
<td>PEy data file (Fixed/Float)</td>
<td>Undefined</td>
</tr>
<tr>
<td>MRF2–0</td>
<td>PEx Multiply Result (Fixed)</td>
<td>Undefined</td>
</tr>
<tr>
<td>MSF2–0</td>
<td>PEy Multiply Result (Fixed)</td>
<td>Undefined</td>
</tr>
<tr>
<td>Processing Background</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R’15–0</td>
<td>PEx data file (Fixed/Float)</td>
<td>Undefined</td>
</tr>
<tr>
<td>S’15–0</td>
<td>PEy data file (Fixed/Float)</td>
<td>Undefined</td>
</tr>
<tr>
<td>MRB2–0</td>
<td>PEx Multiply Result (Fixed)</td>
<td>Undefined</td>
</tr>
<tr>
<td>MSB2–0</td>
<td>PEy Multiply Result (Fixed)</td>
<td>Undefined</td>
</tr>
<tr>
<td>Processing Status</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASTATx</td>
<td>PEx current status</td>
<td>0x0</td>
</tr>
<tr>
<td>ASTATy</td>
<td>PEy current status</td>
<td>0x0</td>
</tr>
<tr>
<td>STKXy</td>
<td>PEx sticky status</td>
<td>0x0540 0000</td>
</tr>
<tr>
<td>STKYy</td>
<td>PEy sticky status</td>
<td>0x0540 0000</td>
</tr>
</tbody>
</table>
Table A-16. Core Non Memory-Mapped Register Listing (Cont’d)

<table>
<thead>
<tr>
<th>Register Mnemonic</th>
<th>Description</th>
<th>Address</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAG Registers Foreground</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I15–0</td>
<td>Index</td>
<td>Undefined</td>
<td></td>
</tr>
<tr>
<td>M15–0</td>
<td>Modify</td>
<td>Undefined</td>
<td></td>
</tr>
<tr>
<td>L15–0</td>
<td>Length</td>
<td>Undefined</td>
<td></td>
</tr>
<tr>
<td>B15–0</td>
<td>Base</td>
<td>Undefined</td>
<td></td>
</tr>
<tr>
<td>DAG Registers Background</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I’15–0</td>
<td>Index</td>
<td>Undefined</td>
<td></td>
</tr>
<tr>
<td>M’15–0</td>
<td>Modify</td>
<td>Undefined</td>
<td></td>
</tr>
<tr>
<td>L’15–0</td>
<td>Length</td>
<td>Undefined</td>
<td></td>
</tr>
<tr>
<td>B’15–0</td>
<td>Base</td>
<td>Undefined</td>
<td></td>
</tr>
<tr>
<td>Miscellaneous Registers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PX</td>
<td>Bus exchange 64-bit</td>
<td>undefined</td>
<td></td>
</tr>
<tr>
<td>PX2–1</td>
<td>Bus exchange 32-bit</td>
<td>undefined</td>
<td></td>
</tr>
<tr>
<td>USTAT4–1</td>
<td>Universal Status</td>
<td>0x0</td>
<td></td>
</tr>
</tbody>
</table>

Emulation Count

<table>
<thead>
<tr>
<th>Register Mnemonic</th>
<th>Description</th>
<th>Address</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMUCLK</td>
<td>Emulation Count</td>
<td>undefined</td>
<td></td>
</tr>
<tr>
<td>EMUCLK2</td>
<td>Emulation Count 2</td>
<td>undefined</td>
<td></td>
</tr>
</tbody>
</table>

Table A-17. Core Memory-Mapped Registers

<table>
<thead>
<tr>
<th>Register Mnemonic</th>
<th>Description</th>
<th>Address</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEMUIN</td>
<td>Emulator Input FIFO</td>
<td>0x30020</td>
<td>Undefined</td>
</tr>
<tr>
<td>EEMUSTAT</td>
<td>Enhanced Emulation Status Register</td>
<td>0x30021</td>
<td>0x0</td>
</tr>
<tr>
<td>EEMUOUT</td>
<td>Emulator Output FIFO</td>
<td>0x30022</td>
<td>Undefined</td>
</tr>
<tr>
<td>SYSCTL</td>
<td>System Control Register</td>
<td>0x30024</td>
<td>0x0</td>
</tr>
</tbody>
</table>
Table A-17. Core Memory-Mapped Registers (Cont’d)

<table>
<thead>
<tr>
<th>Register Mnemonic</th>
<th>Description</th>
<th>Address</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRKCTL</td>
<td>Hardware Breakpoint Control Register</td>
<td>0x30025</td>
<td>0x0</td>
</tr>
<tr>
<td>REVPID</td>
<td>Revision ID Register</td>
<td>0x30026</td>
<td>Mask Dependant</td>
</tr>
<tr>
<td>PSA1S¹</td>
<td>Instruction Breakpoint Address Start #1</td>
<td>0x300A0</td>
<td>Undefined</td>
</tr>
<tr>
<td>PSA1E</td>
<td>Instruction Breakpoint Address End #1</td>
<td>0x300A1</td>
<td>Undefined</td>
</tr>
<tr>
<td>PSA2S</td>
<td>Instruction Breakpoint Address Start #2</td>
<td>0x300A2</td>
<td>Undefined</td>
</tr>
<tr>
<td>PSA2E</td>
<td>Instruction Breakpoint Address End #2</td>
<td>0x300A3</td>
<td>Undefined</td>
</tr>
<tr>
<td>PSA3S</td>
<td>Instruction Breakpoint Address Start #3</td>
<td>0x300A4</td>
<td>Undefined</td>
</tr>
<tr>
<td>PSA3E</td>
<td>Instruction Breakpoint Address End #3</td>
<td>0x300A5</td>
<td>Undefined</td>
</tr>
<tr>
<td>PSA4S</td>
<td>Instruction Breakpoint Address Start #4</td>
<td>0x300A6</td>
<td>Undefined</td>
</tr>
<tr>
<td>PSA4E</td>
<td>Instruction Breakpoint Address End #4</td>
<td>0x300A7</td>
<td>Undefined</td>
</tr>
<tr>
<td>EMUN</td>
<td>Number of Breakpoint Hits Before EMU Interrupt</td>
<td>0x300AE</td>
<td>Undefined</td>
</tr>
<tr>
<td>IOAS</td>
<td>I/O Breakpoint Address Start</td>
<td>0x300B0</td>
<td>Undefined</td>
</tr>
<tr>
<td>IOAE</td>
<td>I/O Breakpoint Address End</td>
<td>0x300B1</td>
<td>Undefined</td>
</tr>
<tr>
<td>DMA1S</td>
<td>Data Memory Breakpoint Address Start #1</td>
<td>0x300B2</td>
<td>Undefined</td>
</tr>
<tr>
<td>DMA1E</td>
<td>Data Memory Breakpoint Address End #1</td>
<td>0x300B3</td>
<td>Undefined</td>
</tr>
<tr>
<td>DMA2S</td>
<td>Data Memory Breakpoint Address Start #2</td>
<td>0x300B4</td>
<td>Undefined</td>
</tr>
<tr>
<td>DMA2E</td>
<td>Data Memory Breakpoint Address End #2</td>
<td>0x300B5</td>
<td>Undefined</td>
</tr>
<tr>
<td>PMDAS</td>
<td>Program Memory Breakpoint Address Start</td>
<td>0x300B8</td>
<td>Undefined</td>
</tr>
<tr>
<td>PMDAE</td>
<td>Program Memory Breakpoint Address End</td>
<td>0x300B9</td>
<td>Undefined</td>
</tr>
</tbody>
</table>

¹ All PSAx registers are cleared for the ADSP-2137x products only.
B  CORE INTERRUPT CONTROL

This appendix provides information about controlling core based interrupts. For information about the IRPTL, LIRPTL, and IMASK registers see Appendix A, Registers.

Interrupt Acknowledge

When an interrupt is triggered, the sequencer typically finishes the current instruction and jumps to the IVT (interrupt vector table). From IVT the address then typically vectors to the ISR routine. The sequencer jumps into this routine, performs program execution and then exits the routine by executing the RTI (return from interrupt) instruction. However this rule does not apply for all cases. There are two interrupt acknowledge mechanisms used in an ISR Routine for the core are shown below and in Table B-1:

- RTI instruction
- Clear status bit + RTI instruction

The Arithmetic exception unit (computation units) is designed such that in order to terminate correctly, the status register must be read to identify the source of the interrupt. Afterwards, programs must write into that status bit (clear mechanism) in order to terminate the interrupt properly.

⚠️ If the acknowledge mechanism rules are not followed correctly, unwanted and sporadic interrupts will occur.
Interrupt Priority

Table B-1. Interrupt Acknowledge Mechanisms

<table>
<thead>
<tr>
<th>Interrupt Source</th>
<th>Interrupt Acknowledge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic Exception (Fixed/Floating Point)</td>
<td>ISR requires clear and RTI</td>
</tr>
<tr>
<td>All other core interrupts</td>
<td>ISR requires RTI only</td>
</tr>
</tbody>
</table>

Interrupt Priority

The core related interrupts have a fixed priority and cannot be changed (as the programmable interrupts for peripherals can).

Interrupt Vector Tables

The 48-bit addresses in the vector table represent offsets from a base IVT address. For an interrupt vector table in internal RAM or ROM consult the processor’s datasheet for the absolute address.

The interrupt name column in Table B-2 lists a mnemonic name for each interrupt as they are defined by the header file that comes with the software development tools. The shaded interrupts are programmable. For more information on using these interrupts, see the product specific hardware reference.

Table B-2. SHARC Interrupt Vector Routing

<table>
<thead>
<tr>
<th>Interrupt Number</th>
<th>Register</th>
<th>Vector Address</th>
<th>Interrupt Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>IRPTL</td>
<td>0x00</td>
<td>EMUI</td>
<td>Emulator (HIGHEST PRIORITY)</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0x04</td>
<td>RSTI</td>
<td>HW/SW Reset</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0x08</td>
<td>IICDI</td>
<td>Illegal IOP access condition OR unaligned long word access detected</td>
</tr>
</tbody>
</table>
## Core Interrupt Control

Table B-2. SHARC Interrupt Vector Routing (Cont’d)

<table>
<thead>
<tr>
<th>Interrupt Number</th>
<th>Register</th>
<th>Vector Address</th>
<th>Interrupt Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>IRPTL</td>
<td>0x0C</td>
<td>SOVFI</td>
<td>Status loop or mode stack overflow; or PC stack full</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0x10</td>
<td>TMZHI</td>
<td>Core Timer (high priority option)</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>0x14</td>
<td>SPERRI</td>
<td>SPORT Error Interrupt¹</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>0x18</td>
<td>BKPI</td>
<td>User Hardware Breakpoint</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>0x1C</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>0x20</td>
<td>IRQ2I</td>
<td>TRQ2 asserted</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>0x24</td>
<td>IRQ1I</td>
<td>TRQ1 asserted</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>0x28</td>
<td>IRQ0I</td>
<td>TRQ0 asserted</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>0x2C</td>
<td>P0I</td>
<td>Programmable Interrupt 0</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>0x30</td>
<td>P1I</td>
<td>Programmable Interrupt 1</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>0x34</td>
<td>P2I</td>
<td>Programmable Interrupt 2</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>0x38</td>
<td>P3I</td>
<td>Programmable Interrupt 3</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>0x3C</td>
<td>P4I</td>
<td>Programmable Interrupt 4</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>0x40</td>
<td>P5I</td>
<td>Programmable Interrupt 5</td>
</tr>
<tr>
<td>17</td>
<td>LIRPTL</td>
<td>0x44</td>
<td>P6I</td>
<td>Programmable Interrupt 6</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>0x48</td>
<td>P7I</td>
<td>Programmable Interrupt 7</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>0x4C</td>
<td>P8I</td>
<td>Programmable Interrupt 8</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>0x50</td>
<td>P9I</td>
<td>Programmable Interrupt 9</td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>0x54</td>
<td>P10I</td>
<td>Programmable Interrupt 10</td>
</tr>
<tr>
<td>22</td>
<td></td>
<td>0x58</td>
<td>P11I</td>
<td>Programmable Interrupt 11</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td>0x5C</td>
<td>P12I</td>
<td>Programmable Interrupt 12</td>
</tr>
<tr>
<td>24</td>
<td></td>
<td>0x60</td>
<td>P13I</td>
<td>Programmable Interrupt 13</td>
</tr>
</tbody>
</table>
Table B-2. SHARC Interrupt Vector Routing (Cont’d)

<table>
<thead>
<tr>
<th>Interrupt Number</th>
<th>Register</th>
<th>Vector Address</th>
<th>Interrupt Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>IRPTL</td>
<td>0x64</td>
<td>P14I</td>
<td>Programmable Interrupt 14</td>
</tr>
<tr>
<td>26</td>
<td></td>
<td>0x68</td>
<td>P15I</td>
<td>Programmable Interrupt 15</td>
</tr>
<tr>
<td>27</td>
<td></td>
<td>0x6C</td>
<td>P16I</td>
<td>Programmable interrupt 16</td>
</tr>
<tr>
<td>28</td>
<td>LIRPTL</td>
<td>0x70</td>
<td>P17I</td>
<td>Programmable Interrupt 17</td>
</tr>
<tr>
<td>29</td>
<td></td>
<td>0x74</td>
<td>P18I</td>
<td>Programmable Interrupt 18</td>
</tr>
<tr>
<td>30</td>
<td>IRPTL</td>
<td>0x78</td>
<td>CB7I</td>
<td>Circular Buffer 7 Overflow</td>
</tr>
<tr>
<td>31</td>
<td></td>
<td>0x7C</td>
<td>CB15I</td>
<td>Circular Buffer 15 Overflow</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td>0x80</td>
<td>TMZLI</td>
<td>Core Timer (Low Priority Option)</td>
</tr>
<tr>
<td>33</td>
<td></td>
<td>0x84</td>
<td>FIXI</td>
<td>Fixed-point overflow</td>
</tr>
<tr>
<td>34</td>
<td></td>
<td>0x88</td>
<td>FLTOI</td>
<td>Floating-point overflow exception</td>
</tr>
<tr>
<td>35</td>
<td></td>
<td>0x8C</td>
<td>FLTUI</td>
<td>Floating-point underflow exception</td>
</tr>
<tr>
<td>36</td>
<td></td>
<td>0x90</td>
<td>FLTII</td>
<td>Floating-point invalid exception</td>
</tr>
<tr>
<td>37</td>
<td></td>
<td>0x94</td>
<td>EMULI</td>
<td>Emulator low priority interrupt</td>
</tr>
<tr>
<td>38</td>
<td></td>
<td>0x98</td>
<td>SFT0I</td>
<td>User software interrupt 0</td>
</tr>
<tr>
<td>39</td>
<td></td>
<td>0x9C</td>
<td>SFT1I</td>
<td>User software interrupt 1</td>
</tr>
<tr>
<td>40</td>
<td></td>
<td>0xA0</td>
<td>SFT2I</td>
<td>User software interrupt 2</td>
</tr>
<tr>
<td>41</td>
<td></td>
<td>0xA4</td>
<td>SFT3I</td>
<td>User software interrupt 3, LOWEST PRIORITY</td>
</tr>
</tbody>
</table>

1 The SPERRI interrupt (bit 5) is reserved for ADSP-21362/3/4/5/6 SHARC processors.
The processor supports the 32-bit single-precision floating-point data format defined in the IEEE Standard 754/854. In addition, the processor supports an extended-precision version of the same format with eight additional bits in the mantissa (40 bits total). The processor also supports 32-bit fixed-point formats—fractional and integer—which can be signed (two’s-complement) or unsigned.

**IEEE Single-Precision Floating-Point Data Format**

The IEEE Standard 754/854 specifies a 32-bit single-precision floating-point format, shown in Figure C-1. A number in this format consists of a sign bit(s), a 24-bit significand, and an 8-bit unsigned-magnitude exponent (e).

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

The hidden bit effectively increases the precision of the floating-point significand to 24 bits from the 23 bits actually stored in the data format. It also ensures that the significand of any number in the IEEE normalized number format is always greater than or equal to one and less than two.
The unsigned exponent, $e$, can range between $1 \leq e \leq 254$ for normal numbers in single-precision format. This exponent is biased by $+127$. To calculate the true unbiased exponent, subtract 127 from $e$.

Figure C-1. IEEE 32-Bit Single-Precision Floating-Point Format

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

- An exponent value of 255 (all ones) with a non-zero fraction is a not-a-number (NAN). NANs are usually used as flags for data flow control, for the values of uninitialized variables, and for the results of invalid operations such as $0 \times \infty$.

- Infinity is represented as an exponent of 255 and a zero fraction. Note that because the fraction is signed, both positive and negative infinity can be represented.

- Zero is represented by a zero exponent and a zero fraction. As with infinity, both positive zero and negative zero can be represented.

The IEEE single-precision floating-point data types supported by the processor and their interpretations are summarized in Table C-1.
**Numeric Formats**

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Exponent</th>
<th>Fraction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>255</td>
<td>Non-zero</td>
<td>Undefined</td>
</tr>
<tr>
<td>Infinity</td>
<td>255</td>
<td>0</td>
<td>((-1)^s) Infinity</td>
</tr>
<tr>
<td>Normal</td>
<td>(1 \leq e \leq 254)</td>
<td>Any</td>
<td>((-1)^s (1.f_{22,0}) 2^{e-127})</td>
</tr>
<tr>
<td>Zero</td>
<td>0</td>
<td>0</td>
<td>((-1)^s) Zero</td>
</tr>
</tbody>
</table>

**Extended-Precision Floating-Point Format**

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

---

SHARC Processor Programming Reference C-3
Short Word Floating-Point Format

The processor supports a 16-bit floating-point data type and provides conversion instructions for it. The short float data format has an 11-bit mantissa with a 4-bit exponent plus sign bit, as shown in Figure C-3. The 16-bit floating-point numbers reside in the lower 16 bits of the 32-bit floating-point field.

![Figure C-3. 16-Bit Floating-Point Format](image)

Packing for Floating-Point Data

Two shifter instructions, **FPACK** and **FUNPACK**, perform the packing and unpacking conversions between 32-bit floating-point words and 16-bit floating-point words. The **FPACK** instruction converts a 32-bit IEEE floating-point number to a 16-bit floating-point number. The **FUNPACK** instruction converts 16-bit floating-point numbers back to 32-bit IEEE floating-point. Each instruction executes in a single cycle. The results of the **FPACK** and **FUNPACK** operations appear in Table C-2 and Table C-3.
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.

### Table C-2. FPACK Operations

<table>
<thead>
<tr>
<th>Condition</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>135 &lt; exp</td>
<td>Largest magnitude representation.</td>
</tr>
<tr>
<td>120 &lt; exp ( \leq ) 135</td>
<td>Exponent is most significant bit (MSB) of source exponent concatenated with the three least significant bits (LSBs) of source exponent. The packed fraction is the rounded upper 11 bits of the source fraction.</td>
</tr>
<tr>
<td>109 &lt; exp ( \leq ) 120</td>
<td>Exponent = 0. Packed fraction is the upper bits (source exponent – 110) of the source fraction prefixed by zeros and the “hidden” one. The packed fraction is rounded.</td>
</tr>
<tr>
<td>exp &lt; 110</td>
<td>Packed word is all zeros.</td>
</tr>
</tbody>
</table>

**exp = source exponent**<br>**sign bit remains the same in all cases**

### Table C-3. FUNPACK Operations

<table>
<thead>
<tr>
<th>Condition</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 &lt; exp ( \leq ) 15</td>
<td>Exponent is the 3 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.</td>
</tr>
<tr>
<td>exp = 0</td>
<td>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” one stripped away.</td>
</tr>
</tbody>
</table>

**exp = source exponent**<br>**sign bit remains the same in all cases**
During the **FPACK** operation, an overflow sets the **SV** condition and non-overflow clears it. During the **FUNPACK** operation, the **SV** condition is cleared. The **S2** and **SS** conditions are cleared by both instructions.

**Fixed-Point Formats**

The processor supports two 32-bit fixed-point formats—fractional and integer. In both formats, numbers can be signed (two’s-complement) or unsigned. The four possible combinations are shown in Figure C-4. In the fractional format, there is an implied binary point to the left of the most significant magnitude bit. In integer format, the binary point is understood to be to the right of the LSB. Note that the sign bit is negatively weighted in a two’s-complement format.

If one operand is signed and the other unsigned, the result is signed. If both inputs are signed, the result is signed and automatically shifted left one bit. The LSB becomes zero and bit 62 moves into the sign bit position. Normally bit 63 and bit 62 are identical when both operands are signed. (The only exception is full-scale negative multiplied by itself.) Thus, the left-shift normally removes a redundant sign bit, increasing the precision of the most significant product. Also, if the data format is fractional, a single bit left-shift renormalizes the MSP to a fractional format. The signed formats with and without left-shifting are shown in Figure C-5.

ALU outputs have the same width and data format as the inputs. The multiplier, however, produces a 64-bit product from two 32-bit inputs. If both operands are unsigned integers, the result is a 64-bit unsigned integer. If both operands are unsigned fractions, the result is a 64-bit unsigned fraction. These formats are shown in Figure C-5.

The multiplier has an 80-bit accumulator to allow the accumulation of 64-bit products. For more information on the multiplier and accumulator, see “Multiply Accumulator” on page 3-9.
Figure C-4. 32-Bit Fixed-Point Formats
Figure C-5. 64-Bit Unsigned and Signed Fixed-Point Product
Alternate Registers.

See index registers on page G-6.

Arithmetic Logic Unit (ALU).

This part of a processing element performs arithmetic and logic operations on fixed-point and floating-point data.

Asynchronous Transfers.

Communications in which data can be transmitted intermittently rather than in a steady stream.

Base Address.

The starting address of a circular buffer to which the DAG wraps around. This address is stored in a DAG $B_x$ register.

Base Register.

A base ($B_x$) register is a data address generator (DAG) register that sets up the starting address for a circular buffer.

Bit-Reverse Addressing.

The data address generator (DAG) provides a bit-reversed address during a data move without reversing the stored address.
Glossary

Boot Modes.

The boot mode determines how the processor starts up (loads its initial code). The ADSP-2136x processors can boot from its SPI port or through its parallel port via an EPROM.

Broadcast Data Moves.

The data address generator (DAG) performs dual data moves to complementary registers in each processing element to support SIMD mode.

Bus Slave or Slave Mode.

The ADSP-21368 SHARC processor can be a bus slave to another processor. The current processor becomes a bus slave when the BR signal of the requester is asserted.

Cache Block.

The smallest unit of memory that is transferred to/from the next level of memory from/to a cache as a result of a cache miss.

Cache Hit.

A memory access that is satisfied by a valid, present entry in the cache.

Cache Miss.

A memory access that does not match any valid entry in the cache.

Circular Buffer Addressing.

The DAG uses the Ix, Mx and Lx register settings to constrain addressing to a range of addresses. This range contains data that the DAG steps through repeatedly, “wrapping around” to repeat stepping through the range of addresses in a circular pattern.
Companding (Compressing/Expanding).

This is the process of logarithmically encoding and decoding data to minimize the number of bits that must be sent.

Conditional Branches.

These are `JUMP` or `CALL/return` instructions whose execution is based on testing an `IF` condition.

Core.

The core consists of these functional blocks: Processing units, memory, DAGs, sequencer, interrupt controller, loop controller, core timer, and emulation interface.

Complementary Data Registers (CDreg).

These are registers in the PEy processing element. These registers are hold operands for multiplier, ALU, or shifter operations and are denoted as $Sx$ when used for fixed point operations or $SFx$ when used for floating-point operations.

Complementary Universal Registers (CUreg).

These are any core registers (data registers), any data address generator (DAG) registers, used in SIMD mode.

Data Address Generator (DAG).

The data address generators (DAGs) provide memory addresses when data is transferred between memory and registers.

Data Registers (Dreg).

These are registers in the PEx processing element. These registers are hold operands for multiplier, ALU, or shifter operations and are denoted as $Rx$ when used for fixed point operations or $Fx$ when used for floating-point operations.
Delayed Branches.

In `JUMP` and `CALL` instructions that use the delayed branch (DB) modifier, one instruction cycle is lost in the instruction pipeline. This is because the processor executes the two instructions after the branch and the third is aborted while the instruction pipeline fills with instructions from the new location.

Denormal Operands.

When the biased exponent is zero, smaller numbers can only be represented by making the integer bit (and perhaps other leading bits) of the significant zero. The numbers in this range are called denormalized (or tiny) numbers. The use of leading zeros with denormalized numbers allows smaller numbers to be represented.

Direct Branches.

These are `JUMP` or `CALL` instructions that use an absolute—not changing at runtime—address (such as a program label) or use a PC-relative address.

DMA (Direct Memory Accessing).

The processor’s I/O processor supports DMA of data between processor memory and external memory, or peripherals. Each DMA operation transfers an entire block of data.

DMA Chaining.

The processor supports chaining together multiple DMA sequences. In chained DMA, the I/O processor loads the next transfer control block (DMA parameters) into the DMA parameter registers when the current DMA finishes and auto-initializes the next DMA sequence.

DMA Parameter Registers.

These registers function similarly to data address generator registers, setting up a memory access process. These registers include internal index
registers, internal modify registers, count registers, chain pointer registers,
external index registers, external modify registers, and external count
registers.

**DMA TCB Chain Loading.**

This is the process that the I/O processor uses for loading the TCB of the
next DMA sequence into the parameter registers during chained DMA.

**Edge-Sensitive Interrupt.**

The processor detects this type of interrupt if the input signal is high
(inactive) on one cycle and low (active) on the next cycle when sampled on
the rising edge of clock.

**Endian Format, Little Versus Big.**

The processor uses big-endian format—moves data starting with most-sig-
nificant-bit and finishing with least-significant-bit—in almost all
instances. There are some exceptions (such as serial port operations) which
provide both little-endian and big-endian format support to ensure their
compatibility with different devices.

**Explicit Versus Implicit Operations.**

In SIMD mode, identical instructions execute on the PEx and PEy com-
putational units; the difference is the data. The data registers for PEy
operations are identified (implicitly) from the PEx registers in the instruc-
tion. This implicit relation between PEx and PEy data registers
corresponds to complementary register pairs.

**Field Deposit (Fdep) Instructions.**

These shifter instructions take a group of bits from the input register
(starting at the LSB of the 32-bit integer field) and deposit the bits as
directed anywhere within the result register.
Glossary

Field Extract (Fext) Instructions.
These shifter extract a group of bits as directed from anywhere within the input register and place them in the result register (aligned with the LSB of the 32-bit integer field).

FIFO (First In, First Out).
A hardware buffer or data structure from which items are taken out in the same order they were put in.

Flag Pins (Programmable).
These pins ($FLGx$) can be programmed as input or output pins using bit settings in the $FLAGS$ register. The status of the flag pins is also given in the $FLAGS$ register.

Flag Update.
The processor’s update to status flags occurs at the end of the cycle in which the status is generated and is available on the next cycle.

General-Purpose Input/Output Pins.
See programmable flag pins.

Harvard Architecture.
Processor’s use memory architectures that have separate buses for program and data storage. The two buses let the processor get a data word and an instruction simultaneously.

I/O Processor Register.
One of the control, status, or data buffer registers of the processor’s on-chip I/O processor.
IDLE.

An instruction that causes the processor to cease operations, holding its current state until an interrupt occurs. Then, the processor services the interrupt and continues normal execution.

Index Registers.

An index register is a data address generator (DAG) register that holds an address and acts as a pointer to memory.

Indirect Branches.

These are JUMP or CALL instructions that use a dynamic—changes at runtime—address that comes from the PM data address generator.

Inexact Flags.

An exception flag whose bit position is inexact.

Input Clock.

Device that generates a steady stream of timing signals to provide the frequency, duty cycle, and stability to allow accurate internal clock multiplication via the phase locked loop (PLL) module.

Interleaved Data.

SIMD mode requires a special memory layout since the implicit modifier is 1 or 2 based on NW or SW addresses. This then requires data to be in an interleaved organization in the memory layout.

Internal Memory Space.

Internal memory space refers to the processor’s on-chip SRAM and memory-mapped registers.
Interrupts.
Subroutines in which a runtime event (not an instruction) triggers the execution of the routine.

JTAG Port.
This port supports the IEEE standard 1149.1 Joint Test Action Group (JTAG) standard for system test. This standard defines a method for serially scanning the I/O status of each component in a system.

Jumps.
Program flow transfers permanently to another part of program memory.

Latency.
Latency of memory access is the time between when an address is posted on the address bus and the core receives data on the corresponding data bus.

Length Registers.
A length register is a data address generator (DAG) register that sets up the range of addresses a circular buffer.

Level-Sensitive Interrupts.
The processor detects this type of interrupt if the signal input is low (active) when sampled on the rising edge of clock.

Loops.
One sequence of instructions executes several times with zero overhead.

Memory Blocks and Banks.
The processor’s internal memory is divided into blocks that are each associated with different data address generators. The processor’s external
memory spaces is divided into banks, which may be addressed by either data address generator.

**Modified Addressing.**

The DAG generates an address that is incremented by a value or a register.

**Modify Address.**

The data address generator (DAG) increments the stored address without performing a data move.

**Modify Registers.**

A modify register is a data address generator (DAG) register that provides the increment or step size by which an index register is pre- or post-modified during a register move.

**Multifunction Computations.**

Using the many parallel data paths within its computational units, the processor supports parallel execution of multiple computational instructions. These instructions complete in a single cycle, and they combine parallel operation of the multiplier and the ALU or dual ALU functions. The multiple operations perform the same as if they were in corresponding single-function computations.

**Multiplier.**

This part of a processing element does floating-point and fixed-point multiplication and executes fixed-point multiply/add and multiply/subtract operations.

**Nonzero numbers.**

Nonzero, finite numbers are divided into two classes: normalized and denormalized.
Glossary

Neighbor Registers.

In long word addressed accesses, the processor moves data to or from two neighboring data registers. The least-significant-32 bits moves to or from the explicit (named) register in the neighbor register pair. In forced long word accesses (normal word address with LW mnemonic), the processor converts the normal word address to long word, placing the even normal word location in the explicit register and the odd normal word location in the other register in the neighbor pair.

Peripherals.

This refers to everything outside the processor core. The SHARC processors’ peripherals include internal memory, parallel port, I/O processor, JTAG port, and any external devices that connect to the processor. Detailed information about the peripherals is found in the product specific hardware reference.

Peripheral Clock.

The peripheral clock controls the processor’s peripherals and is defined as (Peripheral) Clock Period = 2 × tCCLK.

Phase Locked Loop (PLL).

An on-chip frequency synthesizer that produces a full speed master clock from a lower frequency input clock signal.

Post-Modify Addressing.

The data address generator (DAG) provides an address during a data move and auto-increments the stored address for the next move.

Precision.

The precision of a floating-point number depends on the number of bits after the binary point in the storage format for the number. The processor supports two high precision floating-point formats: 32-bit IEEE sin-
gle-precision floating-point (which uses 8 bits for the exponent and 24 bits for the mantissa) and a 40-bit extended precision version of the IEEE format.

Pre-Modify Addressing.

The data address generator (DAG) provides a modified address during a data move without incrementing the stored address.

Register File.

This is the set of registers that transfer data between the data buses and the computation units and DAGs. These registers also provide local storage for operands and results.

Register Swaps.

This special type of register-to-register move instruction uses the special swap operator, <->. A register-to-register swap occurs when registers in different processing elements exchange values.

ROM (Read-Only Memory).

A data storage device manufactured with fixed contents. This term is most often used to refer to non-volatile semiconductor memory.

Saturation (ALU Saturation Mode).

In this mode, all positive fixed-point overflows return the maximum positive fixed-point number (0x7FFFF FFFF), and all negative overflows return the maximum negative number (0x8000 0000).

SHARC.

This is an acronym for Super Harvard Architecture. This processor architecture balances a high performance processor core with high performance buses (PM, DM, I/O, I/O1, I/O2).
Shifter.

This part of a processing element completes logical shifts, arithmetic shifts, bit manipulation, field deposit, and field extraction operations on 32-bit operands. Also, the shifter can derive exponents.

SIMD (Single-Instruction, Multiple-Data).

A parallel computer architecture in which multiple data operands are processed simultaneously using one instruction.

Stack, hardware.

A data structure for storing items that are to be accessed in last in, first out (LIFO) order. When a data item is added to the stack, it is “pushed”; when a data item is removed from the stack, it is “popped.”

Subroutines.

The processor temporarily interrupts sequential flow to execute instructions from another part of program memory.

Stalls.

The time spent waiting for an operation to take place. It may refer to a variable length of time a program has to wait before it can be processed, or to a fixed duration of time, such as a machine cycle. When memory is too slow to respond to the CPU’s request for it, wait states are introduced until the memory can catch up.

Three-State Versus Tristate.

Analog Devices documentation uses the term “three-state” instead of “tristate” because Tristate™ is a trademarked term, which is owned by National Semiconductor.
Universal Registers (Ureg).
These are any processing element registers (data registers), any data address generator (DAG) registers, any program sequencer registers.

Von Neumann Architecture.
This is the architecture used by most (non-processor) microprocessors. This architecture uses a single address and data bus for memory access.

Wait States.
See Stalls
## INDEX

### Numerics

16-bit
- floating-point data, 10-81, 10-82
- floating-point format, 3-25, C-4
- memory block, 7-14
- memory organization, 7-12
- packing, floating point, C-4

32-bit
- fixed-point format, C-6
- single-precision floating-point format, C-2

40-bit
- addressable memory, 7-19
- extended-precision floating-point format, C-3
- floating-point operands, 3-9
- register-to-register transfers, 2-10

48-bit
- access, 7-1
- data transfers (PX register), 2-11
- instructions, 7-20

64-bit
- ALU product (multiplier), C-6
- data passing, 1-10
- PX register, 2-10
- signed fixed-point product, C-6
- unsigned fixed-point product, 3-32
- unsigned integer, C-6

ABS (absolute value) computation, 10-14, 10-25, 10-26, 10-30

absolute address, G-4

AC (ALU fixed-point carry) bit, 3-41, A-17

access between DM or PM and a universal register, 9-13, 9-53, 9-56

access between DM or PM and the register file, 9-18

accessing memory, 7-19

ADD instruction, 3-31

addition
- computation, 10-2
- with borrow computation, 10-10
- with carry computation, 10-4, 10-9
- with division computation, 10-6

address
- calculating, 7-17
- addressing
  - and address ranges, 7-19
  - even short words, 7-28
  - gaps in, 7-19
  - odd short words, 7-28
  - short versus long word, 7-19
  - short word, 7-19
- storing top-of-loop addresses, A-10

AF (ALU floating point operation) bit, 3-41, A-19

AI (ALU floating-point invalid operation) bit, 3-41, A-18

AIS (ALU floating-point invalid) bit, 3-41, A-22

aligning data, 7-12

A

about this document, xxxiv
Index

alternate registers, 1-8

See also secondary registers

ALU, 1-5, 3-1, 3-3

ALUSAT (ALU saturation) bit, 3-32
carry (AC) bit, 3-41, A-17
fixed-point overflow (AOS) bit, 3-41, A-22
floating-point operation (AF) bit, 3-41, A-19
floating-point underflow (AUS) bit, 3-41
instructions, 3-4, 3-6, 3-7
operations, 3-4, 10-2, 11-51
overflow (AV) bit, 3-41
result negative (AN) bit, 3-41
result zero (AZ) bit, 3-41, A-17
saturation, 3-32, 10-2 to 10-5, 10-9 to 10-14, 10-36
saturation (ALUSAT) bit, A-5
status, 3-32, 3-38, 3-41
x-input sign (AS) bit, 3-41, A-17

AN (ALU result negative) bit, 3-41, A-17
AN (ALU result negative) flag, 10-7, 10-28
AND, logical, 3-8, 9-20, 9-34, 9-38, 9-46
AND breakpoints (ANDBKP) bit, 8-13, A-46
AND (logical) computation, 10-16
AOS (ALU fixed-point overflow) bit, 3-41, A-22

arithmetic
operations, 3-3, 3-5
shift, 10-58, 10-59
shifts, G-12
status flags, A-28

AS (ALU x-input sign) bit, 3-41, A-17

ASHIFT computation, 10-58, 10-59
ASTATx/y (arithmetic status) registers, 3-39, 3-41, 3-42, 10-4, 10-7, A-29, A-30

asynchronous

clock (external TCK), 8-7
transfers, G-1
AUS (ALU floating-point underflow) bit, 3-41, A-22
automatic breakpoints, 8-19
AV (ALU overflow) bit, 3-41, A-17
average instructions, 3-31
AVS (ALU floating-point overflow) bit, 3-41, A-22
AZ (ALU result zero or floating-point underflow) bit, 3-41, A-17
AZ flag, 10-7, 10-28

B

background registers, 1-8

See also secondary registers

base (Bx) registers, A-25, G-1
BCLR computation, 10-61
BFFWRP (load bit FIFO writer pointer) computation, 10-88
BFFWRP (move bit FIFO write pointer) computation, 10-87
BHO (buffer hang override) bit, 8-13
BITDEP (bit FIFO deposit) computation, 10-83
BITEXT (bit FIFO extract) computation, 10-85

bit FIFO, 3-22
interrupts, 3-24
status flag and bit (SF), 3-23
write pointer instruction (BFFWRP), 3-22

bit manipulation, 3-16, 9-66, G-12
bit-reverse addressing, G-1

bits

ALU carry (AC), 3-41
ALU fixed-point overflow (AOS), 3-41
ALU floating-point overflow (AVS), 3-41
Index

bits (continued)
  ALU floating-point underflow (AUS), 3-41
  ALU result negative (AN), 3-41
  ALU result zero (AZ), 3-41
  ALU x-input sign (AS), 3-41
  AV (ALU overflow), 3-41
  BHO (buffer hang override), 8-13
  bit-reverse address enable (BRx), A-4
  bus master select (CSEL), A-40
  cache disable (CADIS), A-8
  cache freeze (CAFRZ), A-8
  circular buffer x overflow (CBxS), A-23
  compare accumulation (CCAC), 3-41
  illegal input condition detected (IICD), 7-25
  illegal I/O processor register access enable (IIRAE), 7-25
  internal memory data width (IMDWx), 7-20
  nesting multiple interrupt enable (NESTM), A-5
  overwriting, 7-19
  PC stack full (PCFL), A-23
  shifter input sign (SS), C-6
  shifter overflow (SV), C-6
  shifter zero (SZ), C-6
  unaligned 64-bit memory access (U64MA), 7-25, A-8, A-23
  bit stream manipulation instructions, 3-22
  bit test (BTST) instruction, 3-39
  bit test flag (BFT) bit, 9-66, A-20
  block conflicts, memory, 4-83
  boolean operator
    AND, 3-8, 9-20, 9-34, 9-38, 9-46
    OR, 9-29, 10-17, 10-59, 10-67, 10-78, 11-51, 11-58, A-44, A-45
  boundary scan, 8-6, 8-27
  boundary-scan register, 8-7
  branch, direct, G-4

breakpoint
  output (BRKOUT) pin, 8-11
  stop (BKSTOP) bit, 8-11
  triggering mode (xMODE) bit, 8-12
  break point control register (BRKCTL), 8-19
  broadcast load, A-6, G-2
    enable (BDCSTx) bits, A-6
  broadcast loading, 7-53
  BSET (bit set) computation, 10-62
  BTF (bit test flag) bit, A-20
  BTGL (bit toggle) computation, 10-63
  BTST (bit test) computation, 10-64
  BTST (bit test) instruction, 3-39
  buses
    bus exchange register, 2-10
    bus master select (CSEL) bits, A-5, A-40
    bus slave defined, G-2
    data access types, 7-27
    master, A-5
    structure, 7-11
    bus exchange, 2-3
    Bx (base) registers, A-25, G-1
    BYPASS instruction, 8-5

C
  CACCx (compare accumulation) bits, 3-41, A-20
  cache, 4-6, 4-79 to 4-89
    cache disable (CADIS) bit, A-8
    cache freeze (CAFRZ) bit, A-8
  conflict in memory, 7-3
  controlling, 4-79
  disable (external memory), 4-88
  external instruction fetch, 4-82
  flush, 9-70, 11-42
  flushing, 4-86
  freezing, 4-88
  hit, 4-81, 4-82, 4-84
  inefficient use of, 4-89
Index

cache (continued)
- instruction, defined, 1-8
- instruction fetch and, 4-79
- instructions, 4-83
- invalidate instruction, 4-86
- miss, 4-80
- restrictions, 4-87
- CADIS (cache disable) bit, A-8
- CAFRZ (cache freeze) bit, A-8
- calculating starting address (32-bit addresses), 7-17
- CBxI (circular buffer x overflow interrupt) bit, A-35
- CBxS (circular buffer x overflow) bit, A-23
  See also mode control (MODEx) registers
- enable (CBUFEN) bit, A-6
- circular buffering, length and base registers, A-25
- circular buffering enable (CBUFEN) bit, 9-29, 9-69
- circular buffer x overflow interrupt (CBxI), A-35
- Cjump/Rframe (Type 25) instruction, 9-73
- CLIP computation, 10-22, 10-46
- clock input (CLKin) pin, 8-7
- clocks and system clocking
  CLKin pin, 8-7
  external clock (TCK), 8-7
- companding (compressing/expanding), G-3
- compare accumulation (CACC) bits, 3-41
- COMP computation, 10-7, 10-28
- complementary registers, G-5
- complement (Fn = –Fx) computation, 10-29
- complement (Rn = –Rx) computation, 10-13
- COMPU computation, 10-8
- computation
  ABS, 10-14, 10-25, 10-26, 10-30
  addition, 10-2, 10-23
  addition/division ((Rx + Ry)/2), 10-6
  addition with borrow, 10-10
  addition with carry, 10-4, 10-9
  AND (logical), 10-16
  ASHIFT, 10-58, 10-59
  BCLR, 10-61
  BFFWRP, 10-87, 10-88
  BITDEP, 10-83
  BITTEXT, 10-85
  BSET, 10-62
  BTGL, 10-63
  BTST, 10-64
  CLIP, 10-22, 10-46
  COMP, 10-7, 10-28
  complement (Fn = –Fx), 10-29
  complement (Rn = –Rx), 10-13
  COMPU, 10-8
  COPYSIGN, 10-43
  decrement (Rn = Rx – 1), 10-12
  division (Fx + Fy)/2, 10-27
  dual add/subtract, 3-28
  EXP, 10-77, 10-78
  FDEP, 10-65, 10-67, 10-69, 10-71
  FEXT, 10-73, 10-75
  FIX, 10-36
  FLOAT, 10-38
  FPACK, 10-81
  FUNPACK, 10-82
  increment, 10-11
  LEFTO, 10-80
  LEFTZ, 10-79
  LOGB, 10-35
  LSHIFT, 10-56, 10-57
  MANT, 10-34
  MAX, 10-21, 10-45
  MIN, 10-20, 10-44
Index

computation  
multiplication, 10-48, 10-55
multiplication/addition \((Rn = MRF + Rx * Ry \mod 2)\), 10-49
multiplication/subtraction \((Rn = MRF – Rx * Ry \mod 2)\), 10-50
NOT, 10-19
OR (logical), 10-17
PASS, 10-15, 10-31
RECIPS, 10-39
RND, 10-32, 10-52
ROT, 10-60
RSQRTS, 10-41
SAT, 10-51
SCALB, 10-33
subtraction, 10-3
subtraction \((Fn = Fx – Fy)\), 10-24
subtraction with borrow, 10-5
transfer \((MR = RN/Rn = MR)\), 10-54
TRUNC, 10-36
XOR (logical), 10-18
zero \((MRF = 0)\), 10-53
computational mode
setting, 3-32
status, using, 3-38
compute
field, 10-1, 11-49
operations, 10-1, 11-49
compute/dreg«···»DM|PM, immediate modify (Type 4), 9-17
compute/dreg«···»DM/dreg«···»PM (Type 1), 9-7
compute/modify (Type 7), 9-28, 9-29
compute (Type 2), 9-10
compute (Type 2c), 9-10
compute/ureg«···»DM|PM, register modify (Type 3), 9-12
compute/ureg«···»ureg (Type 5), 9-22
conditional
branches, G-3
call, 9-32, 9-36
instructions, 9-11
jump, 9-32, 9-36, 9-40
loop (DO), 9-49
context switch, 1-8
conventions, xli
converting numbers, 3-25
COPYSIGN computation, 10-43
counter-based loops See also non counter-based loops
CSEL bit, A-40
CURLCNTR (current loop counter) register, A-12
current loop counter (CURLCNTR) register, 3-1, 9-48, A-12
customer support, xxxvi

D
DADDR (decode address) register, A-9
DAGs, 2-3
32-bit
modifier, 6-13
64-bit
DM and PM bus transfers, 6-5
addressing
post-modify, pre-modify, modify, bit-reverse, or circular buffer, 6-1
with DAGs, 6-11
addressing with, 6-11
alternate DAG registers, 6-28
base \((Bx)\) registers, 6-2, 6-24
bit-reverse (BITREV) instruction, 6-17
broadcast load, 6-1
buffer, circular, 6-21
buffer overflow, circular, 6-20, 6-23
Bx (base) registers, 6-2, 6-24
CBUFEN (circular buffer enable) bit, 6-20, 6-24
INDEX

DAGs (continued)
circular buffer addressing, 6-20
circular buffer addressing enable (CBUFEN) bit, 6-20, 6-24
circular buffer addressing registers, 6-23
circular buffer addressing setup, 6-21
circular buffer enable (CBUFEN), 6-20, 6-24
circular buffer wrap, 6-23
data alignment, normal word, 6-8
data type, 6-4
defined, G-3
enable, circular buffer, 6-21
examples, long word moves, 6-9
index (Ix) registers, 6-2, 6-23
instructions, 6-15
  BITREV (bit reverse), 6-17
dual data load, 6-24
interpreting, 6-2
modify, 6-12, 6-17
Ix (index) registers, 6-2, 6-23
long word, 6-8
long word, data moves, 6-9
Lx (length) registers, 6-2, 6-24
memory, access types, 6-27
memory, access word size, 6-4
memory, data types, 6-4
modified addressing, 6-11
modify, immediate value, 6-13
modify, instruction, 6-12, 6-17
modify address, 6-1
modify instruction, 6-17
modify (Mx) registers, 6-2, 6-24
Mx (modify) registers, 6-2, 6-24
operations, 6-7
overview, 1-7
PEVLEN (processing element Y enable) bit, SIMD mode, 6-25, 6-26
post-modify addressing, 6-1, 6-16, 6-17
pre-modify addressing, 6-1
DAGs (continued)
  processing element Y enable (PEVLEN) bit, SIMD mode, 6-25, 6-26
  register descriptions, A-24
  registers, 6-1, A-24
  registers, base, 6-24
  registers, neighbor, 6-9
  registers, secondary registers, 6-29
  SIMD and long word accesses, 6-32
  wrap around, buffer, 6-20, 6-23
  wrap around circular buffer addressing, 6-23
data
  access options, 7-27
  alignment, 7-12
  alignment in memory, 7-14
  bus alignment, 2-10
  (Dreg) registers, G-3
  fixed- and floating-point, G-1
  flow paths, 3-2
  format in computation units, 3-3
  numeric formats, C-1
  packing and unpacking, 3-25, C-4
data address generator, See DAGs
data memory breakpoint hit (STATDx) bit, A-48
data move, 1-10, 3-29
to/from PX, 2-10
data move conditional, 4-97
data register file, 2-2
data registers, 1-5, G-11
debug
  JTAG, 8-1
decode address (DADDR) register, A-9
decrement (Rx = Rx – 1) computation, 10-12
delayed branch
  (DB) instruction, 9-32, 9-36, 9-45
  (DB) jump or call instruction, G-4
denormal operands, 3-34, G-4
development tools, 1-10
direct addressing, 9-53
direct jump|call (type 8), 9-32
division (Fx + Fy)/2 computation, 10-27
DMA
  bus priority, A-42
  defined, G-4
  parameter registers, defined, G-4
  sequences, TCB loading, G-5
DO UNTIL
  counter expired (type 12), 9-48
  (type 13), 9-49
dreg···»DM|PM, immediate modify
  (Type 4c), 9-17
dreg···»DM (16-bit), register modify
  (Type 3d), 9-13
dreg···»DM/dreg···»PM (Type 1c), 9-7
dual add/subtract, 3-28, 3-31
dual processing element moves (broadcast
  load mode), 7-53

E
edge-sensitive interrupts, A-7, G-5
EEMUINENS bit, A-49
EEMUINFULLS bit, A-49
EEMUOUIRQENS bit, A-48
EEMUOUTRDY bit, A-48
effect latency, A-27
EMUIDLE (Type 21d), 9-71
EMUI (emulator lower priority interrupt)
  bit, A-34, A-36
emulator
  boundry scan system, 8-6
  enable (EMUENA) bit, 8-11
  interrupt (EMUI) bit, A-34, A-36
  interrupt enable (EIRQENA) bit, 8-11
  TAP pins, 8-3
  TAP reset state, 8-5
emulator lower priority interrupt (EMUI),
  A-34, A-36
emulator registers
  control shift (EMUCTL), 8-11
  EEMUSTAT (emulator status), 8-18
  event count (EMUN), 8-16
  event counter (EMUN), 8-16
  instruction, 8-5
  Nth event counter (EMUN), 8-16
  enable
  alternate registers, 3-36
  breakpoint (ENBx) bit, 8-12, A-46
  (BRKOUT) pin, 8-11
  broadcast loading, 6-24, 6-25
  circular buffering, 6-19
  DAGs, 6-18
  interrupts, 3-38
  timer, 1-8, 5-1
  timer (timing diagram), 5-4
endian format, G-5
enhanced emulation
  feature enable (EEMUENS) bit, A-49
  FIFO status (EEMUOUTFULLS) bit, A-49
  INDATA FIFO status
    (EEMUINFULLS) bit, A-49
  OUTDATA FIFO status
    (EEMUOUTFULLS) bit, A-49
  OUTDATA interrupt enable
    (EEMUOUIRQENS) bit, A-48
  OUTDATA ready (EEMUOUTRDY)
    bit, A-48
examples
  BITDEP instruction (bit deposit), 3-23
  bit FIFO header creation, 3-24
  bit FIFO header extraction, 3-23
  bit FIFO store/restore, 3-24
  clearing FLAG bit (exception detected),
    3-41
  IIR biquad, 3-29
Index

examples (continued)
MAC and parallel read with data dependency, 3-29
multifunction data move, 3-29
programming memory-to-memory DMA, 8-14
shift immediate instruction, SIMD mode, 3-36

EXP (exponent) computation, 10-77, 10-78
explicit versus implicit operations, G-5
exponent
derivation, G-12
unsigned, C-2
extended precision normal word, 7-12
data access, 7-45, 7-46
SISD mode access, 7-48
external port stop (EPSTOP) bit, 8-12
EXTEST instruction, 8-5

F
FADDR (fetch address) register, A-9, A-10
FDEP (field deposit) computation, 10-65, 10-67, 10-69, 10-71
fetch address (FADDR) register, A-9, A-10
FEXT computation, 10-73, 10-75
field alignment, 10-65, 10-69, 10-73
field deposition/extraction, G-12
FIFO, shifter, 3-22
FIX computation, 10-36
fixed-point
ALU instructions, 3-7
ALU operations, 11-51
data, G-1
formats, C-6
multiplier instructions, 3-14, 3-31
multiplier operations, 11-53
operands, 3-5, A-17
operations, ALU, 10-2, 11-51
overflow interrupt (FIXI) bit, A-35
fixed-point (continued)
product, 64-bit, C-6
product, 64-bit unsigned, 3-32
saturation values, 3-13
fixed-point overflow error, B-4
flag
input/output (FLAGx) pins, A-13
input/output value (FLAGS) register, A-13
update, 3-40, 3-43, G-6
use with NAN, C-2
FLAGS
FLAGx pins, A-13
register, A-13
register, stalls in, A-29
FLOAT computation, 10-38
floating-point
ALU instructions, 3-8
ALU operations, 11-51
data, 3-35, G-1
invalid operation interrupt (FLTII) bit, A-36
multiplier instructions, 3-15
multiplier operations, 3-28, 11-53
operations, 3-32
overflow error, B-4
overflow interrupt (FLTII) bit, A-35
underflow interrupt (FLTUI) bit, A-17, A-35
FLTII (floating-point invalid operation interrupt) bit, A-36
flush cache command, 4-86
formats
See also data format
16-bit floating-point, C-4
40-bit floating-point, C-3
64-bit fixed-point, C-6
fixed-point, 3-13, C-6
integer, fractional, 3-5, 3-10
numeric, C-1
Index

formats (continued)
packing (Fpack/Funpack) instructions, 3-25
short word, C-4
Fpack (floating-point pack) computation, 10-81
F-pack/Funpack (floating-point pack/unpack) instructions, C-4
fractional input(s), 3-16
results, 3-11, C-6
freezing the cache, 4-88
Funpack (floating-point unpack) computation, 3-25, 10-82

G
general-purpose IOP Timer 2 interrupt mask (GPTMR2IMSK) bit, A-38
global interrupt enable, A-5
GPTMR2IMSK bit, A-38

H
hardware breakpoints, 8-19
Harvard architecture, 7-2, G-6

I
IDLE instruction, defined, G-7
IDLE (Type 21d), 9-71
idle (type 22), 9-72
IEEE 1149.1 JTAG standard, G-8
IEEE 754/854 standard, 3-33
IEEE floating-point number conversion, 3-25
IEEE standard 754/854, C-1
IIDC (illegal input condition interrupt) bit, 7-25, A-34
IIRAE (illegal IOP register access enable) bit, 7-25, A-8
IIRA (illegal IOP register access) bit, A-23
illegal input condition detected (IICD) bit, 7-25, A-34
illegal IOP register access (IIRA) bit, A-23
illegal I/O processor register access enable (IIRAE) bit, 7-25, A-8
IMASK (interrupt mask) register, A-31
IMASKP (interrupt mask pointer) register, 9-45, A-31
IMDWx (internal memory data width) bits, 2-9, 7-20
immediate data->DM|PM (Type 16), 9-60
immediate data->ureg (Type 17) instruction, 9-62
immediate data (16-bit)->DM|PM (Type 16c), 9-60
immediate data (16-bit)->ureg (Type 17c) instruction, 9-62
immediate shift/dreg->DM|PM (Type 6) instruction, 9-25
immediate shift instruction, 11-57
implicit operations
complementary registers, 2-6
increment (Rn = Rx + 1) computation, 10-11
INDATA interrupt enable (EEMUINENS) bit, A-49
index (Ix) registers, A-24, G-7
indirect addressing, 1-7, 9-60
indirect branch, G-7
indirect jump Call|Compute (Type 9) instruction, 9-35
indirect jump Call (Type 9c) instruction, 9-36
indirect jump or compute/dreg->DM (Type 10), 9-40
inexact flags, G-7
Index

infinity, round-to, 3-34

instruction
ADD, 3-31
AVE, 3-31
BYPASS, 8-5
delayed branch (DB) JUMP or CALL, G-4
FDEP (field deposit), 3-19
FPACK (floating-point pack), C-4
FUNPACK (floating-point unpack), C-4
Group I (Compute and Move), 9-1, 11-3
Group III (Immediate Move), 9-51
Group IV (Miscellaneous), 9-64, 9-74
multiplier, 3-9, 3-13
(Type 10) indirect jump or compute/dreg«···»DM, 9-40
(Type 11d) return from subroutine|interrupt, 9-44
(Type 11) return from subroutine|interrupt/compute, 9-44
(Type 12) do until counter expired, 9-48
(Type 13) do until, 9-49
(Type 14) ureg«···»DM|PM (direct addressing), 9-53
(Type 15c) ureg«···»DM|PM 7-bit data (indirect addressing), 9-56
(Type 15) ureg«···»DM|PM (indirect addressing), 9-56
(Type 16c) immediate data (16-bit) → ureg, 9-62
(Type 16) immediate data → DM|PM, 9-60
(Type 17c) immediate data (16-bit) → ureg, 9-62
(Type 17) immediate data → ureg, 9-62
(Type 18) system register bit manipulation, 9-66

instruction
(Type 19) I register modify/bit-reverse, 9-69
(Type 1c) dreg«···»DM/dreg«···»PM, 9-7
(Type 1) compute/dreg«···»DM/dreg«···»PM, 9-7
(Type 20) Push|Pop Stacks/Flush Cache, 9-70
(Type 21d) Nop | Idle | EMU Idle, 9-71
(Type 21) Nop, 9-71
(Type 22) Idle, 9-72
(Type 25) Cjump/Rframe, 9-73
(Type 25d) Rframe, 9-74
(Type 2c) compute, 9-10
(Type 2) compute, 9-10
(Type 3) compute/ureg«···»DM|PM, register modify, 9-12
(Type 3c) ureg«···»DM|PM, register modify, 9-12
(Type 3d) dreg«···»DM (16-bit), register modify, 9-13
(Type 4c) dreg«···»DM|PM, immediate modify, 9-17
(Type 4) compute/dreg«···»DM|PM, immediate modify, 9-17
(Type 5) compute/ureg«···»ureg, 9-22
(Type 5c) ureg«···»ureg, 9-24
(Type 6) immediate shift/dreg«···»DM|PM, 9-25
(Type 7) compute/modify, 9-28, 9-29
(Type 8) direct jump|call, 9-32
(Type 9c) indirect jump|call, 9-36
(Type 9) indirect jump|call / compute, 9-35

instruction address breakpoint hit
(STATIx) hit, A-48

instruction alignment buffer (IAB), 4-6
instruction cache, 1-6, 1-8, 9-70
instruction register, emulator, 8-5
Index

instruction set
   notation, 9-4
integer
   input(s), 3-16
   results, 3-11, C-6
intended audience, xxxiii
interleaved data, G-7
interleaving data, 7-27
internal buses, 1-9
internal memory, 7-4, 7-23, G-7
  data width (IMDWx) bits, 7-20
interrupt
   nesting, 4-41
   interrupt input x interrupt (IRQxI) bit, A-34
   interrupt latch (IRPTL) register, A-31
   interrupt latch/mask (LIRPTL) registers, A-36
   interrupt mask (IMASK) control register, A-31
interrupts, 1-7, 7-25, G-8
   and floating-point exceptions, 3-38
   enable, global (IRPTEN) bit, A-5
   input x interrupt (IRQxI) bit, A-35
   interrupt sensitivity, A-7, G-8
   interrupt service routine (ISR), A-29
   interrupt x edge/level sensitivity (IRQxE) bits, A-7
   latch (IRPTL) register, A-31
   latch/mask (LIRPTL) register, A-36
   latch status for, A-31
   mask (IMASK) register, A-31
   mask pointer (IMASKP) register, A-31
   nesting, A-5
   sensitivity, interrupts, A-7
   interrupts, response in sequencer, 4-30
INTEST instruction, 8-5
I/O
   and multiplier registers, 2-2
   stop (IOSTOP) bit, 8-11
   I/O address breakpoint hit (STATI0) bit, A-48
I/O processor
   bus priority, A-42
   registers, G-6
   I register modify/bit-reverse (Type 19), 9-69
   IRPTL (interrupt latch) register, A-31
   IRQxE (interrupt sensitivity) bits, A-7
   IRQxI (hardware interrupt) bits, A-35
   ISR
   programming issues, 9-37, 9-45
   IVT (interrupt vector table) bit, 7-25
   lx (index) registers, A-24, G-7

J
JTAG
   port, G-8
   specification, IEEE 1149.1, 8-6, 8-27
JUMP instructions, G-8

L
LADDR (loop address) register, A-11
   latch
      characteristics, 8-7
      status for interrupts, A-31
   latency
      effect, A-27
      in FLAGS register, A-29
      read, A-27
   LCNTR (loop counter) register, 9-48,
      11-31, A-12
   LEFTO computation, 10-80
   LEFTO operation, A-19
   LEFTZ computation, 10-79
   LEFTZ (shifter) operation, A-19
   level sensitive interrupts, A-7, G-8
   LIRPTL (interrupt) registers, A-36
Index

LOGB (floating-point ALU) computation, 10-35
logical operations, 3-3
logical shifts, G-12
long word
data, 7-12
data access, G-10
single data, 7-49
SISD mode, 7-51
loop, G-8
address stack (LADDR) register, A-11
counter setup, 9-48
counter stack, 9-48
counter stack, access to, A-12
count (LCNTR) register, A-12
current counter (CURLCNTR) register, 2-3
loop abort (LA) instruction, 9-32, 9-36
reentry (LR) modifier, 9-45
stack, 9-32, 9-36, 9-49, 11-42
stack empty (LSEM) bit, A-24
stack overflow (LSOV) bit, A-23
termination, A-10, A-11
LSEM (loop stack empty) bit, A-24
LSHIFT (logical shift) computation, 10-56, 10-57
LSOV (loop stack overflow) bit, A-23
Lx (length) registers, A-25, G-8

MAX computation, 10-21, 10-45
memory, G-7
access priority, 7-62
architecture, 7-2
banks of, G-9
blocks, G-8
broadcast loading, 7-53
buses, 7-2
bus structure, 7-11
data bus alignment, 2-10
data width (IMDWx) bits, 2-9
internal memory data width bit (IMDWx), 7-20
mixing 32-bit & 48-bit words, 7-14
mixing 32-bit and 48-bit words, 7-14
mixing 32-bit data and 48-bit instructions, 7-13
mixing 40/48-bit and 16/32/64-bit data, 7-18
mixing instructions and data
two unused locations, 7-17, 7-18
mixing word width in SIMD mode, 7-64
mixing word width in SISD mode, 7-62
program memory bus exchange (PX) register, 2-10
regions, 7-12, ?? to 7-18
register-to-register moves, 2-10
transition from 32-bit/48-bit data, 7-17
writes, 7-23
memory transfers, 2-10
16-bit (short word), 7-29
32-bit (normal word), 7-37
40-bit (extended precision normal word), 7-45
64-bit (long word), 7-49
bus exchange (PX) registers, 2-10
MI (multiplier floating-point invalid) bit, 3-43, A-19
MIN (minimum) computation, 10-20, 10-44

M
mantissa, 10-34
MANT (mantissa) computation, 10-34
manual
audience, xxxiii
contents, xxxiv
conventions, xli
new in this edition, xxxvi
manual revisions, xxxvi
map 1 and 2 registers, 11-66
master, bus, A-5
MIS (multiplier floating-point invalid) bit, 3-43, A-23
MMASK (mode mask) register, A-39
MN (multiplier negative) bit, 3-42, A-18
mode 1 and 2 options and opcodes, 10-47, 11-55, 11-56
MODE1 register, 3-32, 3-34, A-4
mode mask (MMASK) register, A-39
modified addressing, G-9
modify address, G-9
modify (Mx) registers, A-24, G-9
modify/update an I register with a DAG, 9-28, 9-29
modulo addressing, 1-7
MOS (multiplier fixed-point overflow) bit, 3-43, A-22
move data, 3-29
MRF (multiplier foreground) registers, 3-31
MRF (multiplier foreground) registers, 3-29, 9-11
MR (multiplier result) register transfers, 10-1, 11-49
multifunction, multiplier and ALU, 11-64
multifunction, multiplier and dual add and subtract, 11-64
multifunction, parallel add and subtract, 11-60
multifunction computations, 3-27, 3-28, 3-29, G-9
multifunction instructions, 10-1, 10-89, 11-49, 11-60
registers, 11-60
multiplication/addition (Rn = MRF + Rx * Ry mod2) computation, 10-49
multiplication computation, 10-48
multiplication (Fn = Fx * Fy) computation, 10-55
multiplication/subtraction (Rn = MRF – Rx * Ry mod2) computation, 10-50
multiplier, 1-5, G-9
64-bit product, C-6
clear operation, 3-12
fixed-point overflow status (MOS) bit, 3-43, A-22
floating-point invalid (MI) bit, 3-42, A-18
floating-point invalid status (MIS) bit, 3-43, A-23
floating-point overflow status (MVS) bit, 3-43, A-22
floating-point underflow (MU) bit, 3-43, A-19
floating-point underflow status (MUS) bit, 3-43, A-23
input modifiers, 3-16
instructions, 3-9, 3-13
MRF/B (multiplier result foreground/background) registers, 3-10, 3-11
operations, 3-10, 3-42, 10-47, 11-53
overflow (MV) bit, 3-42, A-18
registers, 2-4
rounding, 3-13
saturation, 3-13
status, 3-38, 3-42, 3-43
multiply accumulator, 3-9
multiply accumulator, See also multiplier
MU (multiplier floating-point underflow) bit, 3-43, A-19
MUS (multiplier floating-point underflow) bit, 3-43, A-23
MV (multiplier not overflow) bit, 3-42, A-18
MVS (multiplier floating-point overflow) bit, 3-43, A-22
Mx (modify) registers, A-24, G-9
Index

N
nearest, round-to, 3-34
negate breakpoint (NEGx) bit, 8-12, A-45
nesting interrupts, 4-41
nesting multiple interrupts enable
  (NESTM) bit, A-5
no boot mode (NOBOOT) bit, 8-13
NOP (Type 21), 9-71, 9-72
NOP (Type 21d), 9-71
normal word, 7-12
  accesses with LW, G-10
  mixing 32-bit data and 48-bit
  instructions, 7-13
SIMD mode, 7-41, 7-43
SISD mode, 7-37, 7-39
not-a-number (NAN), 3-34
notation summary, instruction set, 9-2
NOT computation, 10-19
numbers, infinity, C-2

O
opcode acronyms, 11-1 to ??
operands, 3-10, 3-17, G-11
  in ALU, 3-4
operands for multifunction computations,
  3-29
OR, logical, 9-29, 10-17, 10-59, 10-67,
  10-78, 11-51, 11-58, A-44, A-45
OR (logical) computation, 10-17
OSPIDENS (operating system process ID)
  register enable bit, A-49
OSPID (operating system process ID),
  A-49
overflow, ALU (AV) bit, 3-41
overflow and underflow, 3-25, C-5
overwriting bits, 7-19

P
packing (16-to-32 data), C-4

parallel
  add and subtract, 10-89, 10-90, 11-60
  multiplier and ALU, 11-60
  multiplier with add and subtract, 11-64
parallel accesses to data and program
  memory, 9-7
parallel operations, 3-28, G-9
PASS computation, 10-15, 10-31
PCEM (PC stack empty bit, A-23
PCEM (PC stack empty) bit, A-23
PCFL (PC stack full) bit, A-23
PC (program counter) register, G-4
PC (program counter) stack, 9-32, 9-36,
  9-49, 11-42
PCSTKP (PC stack pointer) register, A-11
peripheral clock (PCLK), 7-21
peripherals, described, G-10
PEYEN (processing element Y enable) bit,
  SIMD mode, 3-36
pin
  boundary scan (JTAG), 8-6
  CLkin (clock input), 8-7
  flag, 2-4, A-13
  timer expired (TMREXP), 5-2
  TRST (test reset) pin, 8-3
pipeline use in, 4-6
porting from previous SHARCs
  performance, 3-37
post-modify addressing, 1-7, G-10
precision
  16-bit, 3-25
  defined, G-10
pre-modify addressing, 1-7, G-11
primary registers, 1-8
processing elements, 1-5, 3-1
  data flow, 3-2
  features, 3-2
  shifter, 3-1
processing element Y enable (PEYEN) bit,
  SIMD mode, 3-36
Index

processor
  buses, 1-9
  core, 1-3
  design advantages, 1-1
processor core, 1-3
  access times for the core to any IOP register, 7-8
  block diagram, 1-4
  buses, 1-9
  IOP core registers, 7-7
  memory block conflicts, preventing, 7-22
  register types in, 2-2
  stalls, 7-9
  user status registers (USTAT), 2-8
program counter
  relative address (PC) register, G-4
  stack empty (PCEM) bit, A-23
  stack full (PCFL) bit, A-23
  stack pointer (PCSTKP) register, A-11
programmable interrupt bits, A-36 to A-38
program memory
  breakpoint hit (STATPA) bit, A-48
  bus exchange (PX) register, A-25
program memory bus exchange (PX) register, 1-10, 2-10
program sequencer, 2-3
  absolute address, 4-17
  AC (ALU fixed-point carry) bit, 4-92
  addressing
    storing top-of-loop addresses, 4-12
ALU
  carry (AC) bit, 4-92
  AND, logical, 4-105
  arithmetic
    exception and interrupts, 4-26
    loops, 4-53
  ASTATx/y (arithmetic status) registers, 4-115
  AV (ALU overflow) bit, 4-92
program sequencer (continued)
  bit manipulation, 4-122
  bit test flag (BTF), 4-92
  bit XOR instruction, 4-92
  block conflicts, 4-83
  boolean operator AND, 4-105
  branch conditional, 4-105
  branch delayed, 4-19, 4-22
  branch direct, 4-17
  branch indirect, 4-17
  branching execution, 4-15
  branching execution direct and indirect branches, 4-17
  branch stalls in, 4-112
  BTF (bit test flag) bit, 4-92
  buffer instruction, 4-6
  buses bus and block conflicts, 4-80
  buses conflicts, 4-38
  cache code examples, 4-89
  cache disable (CADI) bit, 4-88
  cache efficient use of, 4-88
  cache freeze (CAFRZ) bit, 4-88
  cache hit, 4-81, 4-88
  cache miss, 4-81, 4-88
  cache restrictions on use, 4-87
  CADIS (cache disable) bit, 4-88
  CAFRZ (cache freeze) bit, 4-88
  CALL instructions, 4-16
  clock cycles and program flow, 4-5
  complementary conditions, 4-105
  conditional branches, 4-105
  conditional complementary conditions, 4-105
  conditional compute operations, 4-96
  conditional conditions list, 4-92, 4-94
  conditional execution summary, 4-95
  conditional instructions, 4-122
  conditional instruction stalls, 4-115
  conditional SIMD mode and conditionals, 4-94
## Index

*program sequencer*  
  condition codes, 4-92  
  conflicts block, 4-80  
  conflicts bus, 4-38, 4-80  
  control, 1-6  
  core stalls, 4-108 to 4-115  
  counter-based loops, 4-60  
  DADDR (decode address) register, 4-4  
  decode address (DADDR) register, 4-3  
  delayed branch (DB) instruction, 4-19, 4-22  
  delayed branch (DB) jump or call instruction, 4-21  
  delayed branch limitations, 4-22  
  delayed interrupt processing, causes, 4-38  
  DO UNTIL instruction, 4-47  
  DO UNTIL loops, 4-47  
  edge-sensitive interrupts, 4-119  
  enable cache, 4-87  
  enable nesting, interrupt, 4-32  
  equals (EQ) condition, 4-92, 4-94  
  examples cache inefficient code, 4-89  
  examples direct branch, 4-17  
  examples DO UNTIL loop, 4-51  
  examples interrupt service routine, 4-36  
  fetch address (FADDR) register, 4-3  
  fetched address, 4-3  
  flag input (FLAGx_IN) conditions, 4-93  
  greater or equals (GE) condition, 4-92  
  greater than (GT) condition, 4-92  
  IDLE instruction, 4-2  
  indirect branch, 4-17  
  instruction (bit), 4-122  
  instruction bit XOR, 4-92  
  instruction CALL, 4-16  
  instruction delayed branch (DB), 4-19, 4-22  
  instruction delayed branch (DB) JUMP or CALL, 4-21  

*program sequencer (continued)*  
  instruction DO UNTIL, 4-47  
  instruction INNER, 4-89  
  instruction loop counter expired (LCE), 4-51  
  instruction OUTER, 4-89  
  instruction pipeline, 4-3  
  instruction pipeline counter-based four instruction loops, 4-67  
  instruction pipeline counter-based single instruction loops, 4-60 to 4-63  
  instruction pipeline counter-based three instruction loops, 4-66  
  instruction pipeline counter-based two instruction loops, 4-64 to 4-65  
  instruction pipeline stalls, data and control, 4-109  
  instruction pipeline stalls, structural, 4-108  
  instruction pipeline stalls in, 4-108  
  interrupt response, 4-30  
  interrupt sources, 4-33  
  interrupts single-cycle instruction latency, 4-31  
  JUMP instructions, 4-1, 4-16  
  JUMP instructions, stalls caused by, 4-116  
  JUMP instructions clear interrupt (CI) register, 4-17  
  JUMP instructions loop abort (LA) register, 4-17  
  JUMP instructions pops status stack with (CI), 4-14  
  LA (loop abort instruction), 4-17  
  latching interrupts, 4-34  
  latency, 4-30, 4-88, 4-122  
  latency effect in MODE2 register, 4-87  
  latency system registers, 4-122  
  LCE (loop counter expired condition), 4-114
program sequencer

(continued)

loop abort (LA) modifier in a jump instruction, 4-17, 4-47, 4-56, 4-116
loop address stack, 4-48, 4-122
loop conditional loops, 4-51
loop counter expired (LCE) instruction, 4-51
loop counter register, defined, 4-51
loop defined, 4-1
loop do/until instruction, 4-51
loop restrictions, 4-55, 4-59
masking interrupts, 4-29
mnemonics evaluation of, 4-92
nested interrupt routines, 4-122
nesting multiple interrupts enable (NESTM) bit, 4-12
not equal (NE), 4-92, 4-94
pop program counter (PC) stack, 4-16
pop status stack, 4-14
processor core stalls, 4-108 to 4-117
program flow branches, 4-15 to 4-25
program flow hardware stacks and, 4-10
program flow nonsequential, 4-10
program flow operating mode, 4-26
program flow stack access and, 4-12
push loop counter stack, 4-50
push program counter (PC) stack, 4-16
push status stack, 4-14
restrictions delayed branch, 4-22
restrictions on ending loops, 4-55
restrictions on short loops, 4-59
return (RTI/RTS) instructions, 4-16
RTI/RTS (return from/to interrupt) instructions, 4-16
sensing interrupts, 4-119
stack overview, 4-11
stacks status, 4-27
stacks status, current values in, 4-14
stalls data and control, 4-109
stalls in branches, 4-112

program sequencer

(continued)
stalls in conditional branches, 4-113
stalls instruction pipeline, 4-108 to 4-118
stalls structural, 4-108
stalls to optimize performance, 4-116
stalls with JUMP(LA) modifier, 4-116
status stack, 4-15
subroutines, 4-1
SV (shifter overflow) bit, 4-93
termination codes, condition codes and loop termination, 4-92
test flag (TF) condition, 4-93
top-of-loop address, 4-46
top-of-PC stack, 4-13
uncomplemented register, 4-96
underflow, multiplier, 4-92
VISA instruction alignment buffer, 4-6
program sequencer bits
cache disable (CADIS), 4-88
cache freeze (CAFRZ), 4-88
least recently used (LRU), 4-89
nesting multiple interrupt enable (NESTM), 4-12
program sequencer interrupts, 4-2
and sequencing, 4-26
delayed, 4-38
hold off, 4-38
inputs (IRQ2-0), 4-26
interrupt service routine (ISR), 4-28
interrupt vector table, 4-27
interrupt vector table (IVT), 4-26
interrupt x edge/level sensitivity (IRQxE) bits, 4-120
latching, 4-34
latch (IRPTL) register, 4-16
latch/mask (LIRPTL) register, 4-29
latency, 4-30
masking and latching, 4-29, 4-34
nested interrupts, 4-14
Index

program sequencer interrupts  (continued)
  nesting enable (NESTM) bit, 4-12
  PC stack full, 4-12
  processing, 4-27
  response, 4-26
  re-using, 4-35

purpose of this manual, xxxiii

PC stack full, 4-12

processing, 4-27

push/pop stacks/flush cache (Type 20), 9-70

PX (program memory bus exchange)
  register, 1-10, 2-10, A-25

R

read latency, A-27

RECIPS (reciprocal) computation, 10-39

Reference Notation Summary, 9-2

register codes, 11-66

register drawings, reading, A-2

register files, G-11

registers
  See also timer registers
  ASTATxy, 3-39, 3-41
  base, A-25, G-1
  boundary, 8-7
  BRKCTL (breakpoint control), 8-19
  complementary, G-5
  DAG, A-24
  data, 1-5, G-11
  decode address (DADDR), A-9
  for multifunction computations, 11-60
  MODE1, 3-32, 3-34
  neighbor, 7-49, 7-51
  program memory bus exchange (PX), 2-10
  restrictions on data registers, 3-28
  STKXy (sticky), 3-39
  universal (Ureg), 1-10, G-13
  user-defined status (USTATx), A-26
  register-to-register
    swaps, G-11

register-to-register data transfers, 2-10

reset interrupt (RSTI) bit, A-34

restrictions
  breakpoints, setting, 8-26
  mixing 32- and 48-bit words, 7-16
  register, 9-7

return from an interrupt service routine (RTI), 9-44

return from a subroutine (RTS), 9-33, 9-37, 9-44, 9-45

return from subroutine | interrupt,
  compute (Type 11), 9-44

return from subroutine | interrupt,
  (Type 11d), 9-44

Rframe/Cjump (Type 25) instruction, 9-73

Rframe (Type 25d) instruction, 9-74

RND (round) computation, 10-32, 10-52

ROT (rotate) computation, 10-60

rounded output, 3-16

rounding, 3-34

rounding 32-bit data (RND32) bit, A-5

rounding mode, 3-34, A-5

round instruction, 3-13

RSQRTS (reciprocal square root)
  computation, 10-41

RSTI (reset interrupt) bit, A-34

S

SAMPLE (emulator) instruction, 8-5

SAT computation, 10-51

saturate instruction, 3-13

saturation (ALU saturation mode), G-11

saturation maximum values, 3-13

saturation mode, 10-2, 10-3, 10-4, 10-5,
  10-9, 10-10, 10-11, 10-12, 10-13,
  10-14, 10-35, 10-36

SCALB (scale) computation, 10-33

secondary processing element, 3-36
secondary registers, 1-8, A-4
for computational units (SRCU) bit, A-4
for DAGs (SRDxH/L) bits, A-4
for register file (SRRFH/L) bit, A-4
secondary registers for DAGs (SRDxH/L) bits, A-4
secondary registers for register file (SRRFH/L) bit, A-4
serial test access port (TAP), 8-6
setting breakpoints, 8-19
SFTx (user software interrupt) bits, A-36
shadow write FIFO, 7-23
SHARC architecture, G-11
background information, 1-11
porting from previous SHARCs, 1-11
shifter, 1-5, 3-16, G-12
bit manipulation operations, 3-16
bit stream manipulation instructions, 3-22
FIFO, 3-22
fixed-point/floating-point conversion, 3-17
immediate operation, 11-57
instructions, 3-25, 3-26, 3-27
operations, 3-17, 3-43, 10-56, 11-57, A-19
results, 3-18
status flags, 3-43
shifter input sign (SS) bit, A-20, C-6
shifter overflow (SV) bit, C-6
shifter zero (SZ) bit, C-6
short (16-bit data) sign extend (SSE) bit, A-5
short float data format, 10-81, 10-82
short word, 7-12
16-bit format, C-4
SIMD mode, 7-33, 7-35, 7-41
SISD mode, 7-29, 7-31
short word sign extension bit (SSE), 3-33
signals
clock, G-7
core clock (CCLK), 7-11
test clock (TCK), 8-3, 8-7
signed
fixed-point product, C-6
input, 3-16
sign extension, A-5
SIMD (single-instruction, multiple-data) mode, 1-5, A-6
broadcast load mode, 6-24
complementary registers, 2-6
complimentary register pairs, 2-6
complimentary registers, 11-65
DAG operations and, 6-26
defined, 3-36
memory access and, 7-28
processing elements, 3-36
register transfers (UREG/SREG), 2-16
shift immediate instruction, 3-36
status flags, 3-40
Type 10 instruction, 9-41
Type 11 instruction, 9-46
Type 12 instruction, 9-48
Type 13 instruction, 9-49
Type 14 instruction, 9-53
Type 15 instruction, 9-57
Type 16 instruction, 9-60
Type 17 instruction, 9-62
Type 18 instruction, 9-67
Type 19 instruction, 9-69
Type 1 instruction, 9-7, 9-9
Type 20 instruction, 9-70
Type 2 instruction, 9-11
Type 3 instruction, 9-14
Type 4 instruction, 9-18
Type 5 instruction, 9-22
Type 6 instruction, 9-25
Type 7 instruction, 9-29
Type 8 instruction, 9-33
Index

SIMD  
Type 9 instruction, 9-37
single-precision format, 3-33
single serial shift register path, 8-7
single-step (SS) bit, 8-11
SISD (single-instruction, single-data) mode defined, 1-5
software breakpoints, 8-19
software interrupt (SFT0x) bit, A-36
software interrupt x, user (SFTxI) bit, A-36
software pipelining, 3-28
software reset (SRST) bit, A-42
software reset (SYSRST) bit, 8-11
SOVFI (stack overflow/full) bit, A-34
SP0I (serial port interrupt) bit, A-38
SP2I (serial port interrupt) bit, A-38
SP4I (serial port interrupt) bit, A-38
SPI receive DMA interrupt mask (SPILIMSKP) bit, A-39
SPORT transmit channel 4 (SP4I), A-38
SSEM (status stack empty) bit, A-23
SS (shifter input sign) bit, A-20, C-6
stack overflow/full interrupt (SOVFI) bit, A-34
stacks
  PC (program counter) latency in, A-27
  SSOV (status stack overflow) bit, A-23
  stalls
    for backward compatibility, A-27
STATDAx (data memory breakpoint hit) bit, A-48
STATI0 (I/O address breakpoint hit) bit, A-48
STATIx (instruction address breakpoint hit) bit, A-48
STATPA (program memory data breakpoint hit) bit, A-48
status stack, 9-45, 9-70
status stack empty (SSEM) bit, A-23
status stack overflow (SSOV) bit, A-23
sticky status (STKYx/y) register, 3-39, A-19, A-20
STKYx/y register, 3-39, A-19, A-20
subroutines, G-12
subtraction instructions, 3-31
subtraction computation, 10-3
subtraction (F_n = F_x – F_y) computation, 10-24
subtraction with borrow computation, 10-5
subtract/multiply, G-9
support, technical or customer, -xxxvi
SV (shifter overflow) bit, A-19, C-6
swap between universal registers, 9-22
swap register operator, G-11
SYSCTL register
  internal memory data width (IMDWx) bits, A-42
  rotating priority bus arbitration (RBPR) bit, A-42
SRST (software reset) bit, A-42
SYSCTL (system control) register, A-41
  system control register. See SYSCTL register
system control register (SYSCTL), 7-12, 7-19, 7-20, 8-11
system register bit manipulation (Type 18), 9-66
system registers (SREG), 2-4
SZ (shifter zero) bit, A-19, C-6

T
TAP registers
  boundary-scan, 8-7
TCK pin, 8-3
TDI (test data in) pin, 8-3
technical support, -xxxvi
termination condition, 9-48, 9-49
test access port, See TAP, emulator
test clock (TCK) pin, 8-3
Index

test data input (TDI) pin, 8-3
test mode
JTAG, 8-1
TIMEN (timer enable) bit, A-8
timer, 1-8, 2-3
timer expired high priority (TMZHI), A-34
timer expired low priority (TMZLI), A-35
TMS (test mode select) pin, 8-3
TMZH (timer expired high priority) bit, A-34
TMZLI (timer expired low priority) bit, A-35
transfer between universal registers, 9-22
transfer (MR = RN/Rn = MR) computation, 10-54
tri-state vs. three-state, G-12
TRST (test reset) pin, 8-3
truncate, rounding (TRUNC) bit, A-5
TRUNC computation, 10-36
two’s-complement data, 3-5

U
U64MA bit, 7-25, A-8, A-23
UMODE (user mode breakpoint function enable) bit, 8-19
unaligned 64-bit memory access (U64MA) bit, A-8
underflow, 10-81, 10-82
underflow exception, 3-34
universal registers (Ureg), 1-10, 2-2, 2-10, 9-53, 9-56, 11-66, G-13
unpacking (32-to-16-bit data), C-4
unsigned
fixed-point product, 3-32
input, 3-16
update an I register with an M register, 9-28
ureg→→DM|PM, register modify (Type 3c), 9-12
Ureg→→DM|PM 7-bit data (indirect addressing) (Type 15c), 9-56
Ureg→→DM|PM (direct addressing) (Type 14), 9-53
Ureg→→DM|PM (indirect addressing) (Type 15), 9-56
ureg→ureg (Type 5c), 9-24
user-defined status registers (USTATx) registers, A-26
user-defined status (USTATx) register, A-26
USTATx registers, A-26

V
valid data registers for input operands, 11-61
values, saturation maximum, 3-13
VISA instructions, 9-7, 9-10, 9-12, 9-13, 9-17, 9-24, 9-36, 9-44, 9-60, 9-62, 9-71, 9-74
Von Neumann architecture, 7-2, G-13

W
wait states, defined, G-13
word rotations, 7-14
write 32-bit immediate data to DM or PM, 9-60
write 32-bit immediate data to register, 9-62
writing memory, 7-23

X
XOR (logical) computation, 10-18

Z
zero, round-to, 3-34
zero (MRF = 0) computation, 10-53