ADSP-21160 SHARC® DSP Instruction Set Reference

Revision 2.0, November 2003

Part Number
82-001967-01

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

ANALOG
DEVICES

Contents

1 INTRODUCTION

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

Purpose

The ADSP-21160 SHARC DSP Instruction Set Reference provides assembly syntax information for the ADSP-21160 Super Harvard Architecture (SHARC) Digital Signal Processor (DSP). The syntax descriptions cover instructions that execute within the DSP's processor core (processing elements, program sequencer, and data address generators). For architecture and design information on the DSP, see the *ADSP-21160 SHARC DSP Hardware Reference*.

Audience

DSP system designers and programmers who are familiar with signal processing concepts are the primary audience for this manual. This manual assumes that the audience has a working knowledge of microcomputer technology and DSP-related mathematics.

DSP system designers and programmers who are unfamiliar with signal processing can use this manual, but should supplement this manual with other texts, describing DSP techniques.

All readers, particularly programmers, should refer to the DSP's development tools documentation software development information. For additional suggested reading, see "For Information About Analog Products" on page 1-6.

Contents Overview

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

- "Instruction Summary" on page 2-1 – This chapter provides a syntax summary of all instructions and describes the conventions that are used on the instruction reference pages.
- "Compute and Move" on page 3-1 – These instruction specify a compute operation in parallel with one or two data moves or an index register modify.
- "Program Flow Control" on page 4-1 – These instructions specify various types of branches, calls, returns, and loops. Some may also specify a compute operation and/or a data move.
- "Immediate Move" on page 5-1 – These instructions use immediate instruction fields as operators for addressing.

- "Miscellaneous Operations" on page 6-1 – These instructions include bit modify, bit test, no operation, idle, and cache manipulation.
- "Computations Reference" on page 7-1 – This chapter describes computation and multifunction computation operations that are available within many instructions' opcodes through a `COMPUTE` field that specifies a compute operation using the ALU, multiplier, or shifter.

Each of the DSP's instructions is specified in this text. The reference page for an instruction shows the syntax of the instruction, describes its function, gives one or two assembly-language examples, and identifies fields of its opcode. The instructions are referred to by type, ranging from 1 to 25. These types correspond to the opcodes that ADSP-21160 DSPs recognize, but are for reference only and have no bearing on programming.

Some instructions have more than one syntactical form; for example, instruction "Type 4: Compute, dreg«…»DM | PM, data modify" on page 3-14 has four distinct forms.

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

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

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

Development Tools

The ADSP-21160 is supported by VisualDSP++TM, an easy-to-use project management environment, comprised of an Integrated Development Environment (IDE) and Debugger. VisualDSP++ lets you 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.

Flexible Project Management. The IDE provides flexible project management for the development of DSP applications. The IDE includes access to all the activities necessary to create and debug DSP projects. You can create or modify source files or view listing or map files with the IDE Editor. This powerful Editor is part of the IDE and includes multiple language syntax highlighting, OLE drag and drop, bookmarks, and standard editing operations such as undo/redo, find/replace, copy/paste/cut, and go to.

Also, the IDE includes access to the SHARC DSP C Compiler, C Run-time Library, Assembler, Linker, Loader, Simulator, and Splitter. You specify options for these SHARC DSP Tools through Property Page dialogs. Property Page dialogs are easy to use, and make configuring, changing, and managing your projects simple. These options control how the tools process inputs and generate outputs, and have a one-to-one correspondence to the tools' command line switches. You can define these options once, or modify them to meet changing development needs. You can also access the SHARC DSP Tools from the operating system command line if you choose.

Greatly Reduced Debugging Time. The Debugger has an easy-to-use, common interface for all processor simulators and emulators available through Analog Devices and third parties or custom developments. The Debugger has many features that greatly reduce debugging time. You can view C source interspersed with the resulting Assembly code. You can profile execution of a range of instructions in a program; set simulated watch points on hardware and software registers, program and data memory; and trace instruction execution and memory accesses. These features enable you to correct coding errors, identify bottlenecks, and examine DSP performance. You can use the custom register option to select any combination of registers to view in a single window. The Debugger can also generate inputs, outputs, and interrupts so you can simulate real world application conditions.

SHARC DSP Software Development Tools. SHARC DSP Software Development Tools, which support the SHARC DSP Family, allow you to develop applications that take full advantage of the SHARC DSP architecture, including multiprocessing, shared memory, and memory overlays. SHARC DSP Software Development Tools include C Compiler, C Runtime Library, DSP and Math Libraries, Assembler, Linker, Loader, Simulator, and Splitter.

C Compiler & Assembler. The C Compiler generates efficient code that is optimized for both code density and execution time. The C Compiler allows you to include Assembly language statements inline. Because of this, you can program in C and still use Assembly for time-critical loops. You can also use pretested Math, DSP, and C Runtime Library routines to help shorten your time to market. The SHARC DSP Family Assembly language is based on an algebraic syntax that is easy to learn, program, and debug. The add instruction, for example, is written in the same manner as the actual equation (for example, `Rx = Ra + Rb`;).

Linker & Loader. The Linker provides flexible system definition through Linker Description Files (`.LDF`). In a single LDF, you can define different types of executables for a single or multiprocessor system. The Linker resolves symbols over multiple executables, maximizes memory use, and easily shares common code among multiple processors. The Loader supports creation of host, link port, and PROM boot images. Along with the Linker, the Loader allows multiprocessor system configuration with smaller code and faster boot time.

Simulator. The Simulator is a cycle-accurate, instruction-level simulator that allows you to simulate your application in real time.

Third-Party Extensible. The VisualDSP++ environment enables third-party companies to add value by using a published set of Application Programming Interfaces (API) provided by Analog Devices. Third-party products—runtime operating systems, emulators, high-level language compilers, multiprocessor hardware—can interface seamlessly with VisualDSP++ thereby simplifying the tools integration task.

VisualDSP++ follows the COM API format. Two API tools, Target Wizard and API Tester, are also available for use with the API set. These tools help speed the time-to-market for vendor products. Target Wizard builds the programming shell based on API features the vendor requires. The API tester exercises the individual features independently of VisualDSP++. Third parties can use a subset of these APIs that meets their application needs. The interfaces are fully supported and backward compatible.

Further details and ordering information are available in the VisualDSP++ Development Tools data sheet. This data sheet can be requested from any Analog Devices sales office or distributor.

For Information About Analog Products

Analog Devices is online on the internet at `http://www.analog.com`. Our Web pages provide information on the company and products, including access to technical information and documentation, product overviews, and product announcements.

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

- Visit our World Wide Web site at `www.analog.com`
- FAX questions or requests for information to 1(781)461-3010.
- Access the Computer Products Division File Transfer Protocol (FTP) site at `ftp ftp.analog.com` or `ftp 137.71.23.21` or `ftp://ftp.analog.com`.

For Technical or Customer Support

You can reach our Customer Support group in the following ways.

- E-mail questions to `dsp.support@analog.com` or `dsp.europe@analog.com` (European customer support)
- Telex questions to 924491, TWX:710/394-6577
- Cable questions to ANALOG NORWOODMASS
- Contact your local ADI sales office or an authorized ADI distributor
- Send questions by mail to:

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

USA
```

What's New in This Manual

This is the second edition of the *ADSP-21160 SHARC DSP Instruction Set Reference*. This edition was updated to correct all open document errata.

Related Documents

For more information about Analog Devices DSPs and development products, see the following documents.

- *ADSP-21160 SHARC DSP Microcomputer Data Sheet*
- *ADSP-21160 SHARC DSP Hardware Reference*
- *Getting Started Guide for SHARC DSPs*
- *User's Guide for SHARC DSPs*
- *C/C++ Compiler and Library Manual for SHARC DSPs*
- *Assembler and Preprocessor Manual for SHARC DSPs*
- *Linker and Utilities Manual for SHARC DSPs*

All the manuals are included in the software distribution CD-ROM. To access these manuals in the VisualDSP++ environment, select **Contents** from the **Help** menu. Then select **Manuals** and open any of the manuals, which are in Adobe Acrobat PDF format.

Conventions

The following conventions apply to all chapters. Note that additional conventions, which apply only to specific chapters, appear throughout this document.

Table 1-1. Notation Conventions

Example	Description
`PC, R1, PX`	Register names appear in UPPERCASE and keyword `font`
`TIMEXP`, $\overline{\texttt{RESET}}$	Pin names appear in UPPERCASE and keyword `font`; active low signals appear with an $\overline{\texttt{OVERBAR}}$.

Table 1-1. Notation Conventions

Example	Description
If, Do/Until	Assembler instructions (mnemonics) appear in initial capitals
Click Here	In the online version of this document, a cross reference acts as a hyper-text link to the item being referenced. Click on blue references (Table, Figure, or section names) to jump to the location.

2 INSTRUCTION SUMMARY

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

Chapter Overview

The following summary topics appear in this chapter.

Compute and Move/Modify Summary

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

"Type 1: Compute, Dreg«…»DM | Dreg«…»PM" on page 3-3

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

"Type 2: Compute" on page 3-7

```
IF  COND compute  ;
```

"Type 3: Compute, ureg«…»DM | PM, register modify" on page 3-9

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

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

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

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

“Type 4: Compute, dreg«···»DM | PM, data modify” on page 3-14

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

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

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

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

“Type 5: Compute, ureg«···»ureg | Xdreg<->Ydreg” on page 3-19

```
IF COND compute,      | ureg1 = ureg2       |  ;

                      | X dreg <-> Y dreg   |
```

“Type 6: Immediate Shift, dreg«···»DM | PM” on page 3-23

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

                        , dreg  =           | DM(Ia, Mb)  ;       |
                                            | PM(Ic, Md)  ;       |
```

“Type 7: Compute, modify” on page 3-28

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

Program Flow Control Summary

Program flow control instructions are classed as Group II instructions, and they let you control program execution flow. The series of tables that follow summarize the Group II instructions. For a complete description of these instructions, see the noted pages.

"Type 8: Direct Jump | Call" on page 4-3

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

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

"Type 9: Indirect Jump | Call, Compute" on page 4-8

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

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

"Type 10: Indirect Jump | Compute, dreg«···»DM" on page 4-15

IF COND Jump	(Md, Ic)	,Else	compute, DM(Ia, Mb) = dreg ;
	(PC, <reladdr6>)		compute, dreg = DM(Ia, Mb) ;

"Type 11: Return From Subroutine | Interrupt, Compute" on page 4-21

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

IF COND RTI	(DB)	, compute	;
		, ELSE compute	

"Type 12: Do Until Counter Expired" on page 4-26

LCNTR =	<data16>	, DO	<addr24>	UNTIL LCE;
	ureg		(PC, <reladdr24>)	

"Type 13: Do Until" on page 4-28

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

Immediate Move Summary

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

"Type 14: Ureg«···»DM | PM (direct addressing)" on page 5-2

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

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

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

DM(<data32>, Ia)	= ureg	(LW);
PM(<data32>, Ic)		

ureg =	DM(<data32>, Ia)
	PM(<data32>, Ic)
		(LW);

"Type 16: Immediate data···»DM | PM" on page 5-9

DM(Ia, Mb)
PM(Ic, Md)
	= <data32> ;

"Type 17: Immediate data···»Ureg" on page 5-12

```
ureg = <data32> ;
```

Miscellaneous Operations Summary

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

"Type 18: System Register Bit Manipulation" on page 6-2

BIT	SET CLR TGL TST XOR	sreg <data32> ;

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

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

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

"Type 20: Push, Pop Stacks, Flush Cache" on page 6-8

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

"Type 21: Nop" on page 6-10

NOP ;

"Type 22: Idle" on page 6-11

IDLE ;

"Type 25: Cjump/Rframe" on page 6-12

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

RFRAME ;

Register Types Summary

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

Table 2-1. Universal Registers (Ureg)

Register Type	Register(s)	Function
Register File (ureg & dreg)	R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15	Processing element X register file locations, fixed-point
	F0, F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, F13, F14, F15	Processing element X register file locations, floating-point
	S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15	Processing element Y register file locations, fixed-point
	SF0, SF1, SF2, SF3, SF4, SF5, SF6, SF7, SF8, SF9, SF10, SF11, SF12, SF13, SF14, SF15	Processing element Y register file locations, floating-point
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

Table 2-1. Universal Registers (Ureg) (Cont'd)

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

Table 2-1. Universal Registers (Ureg) (Cont'd)

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

Table 2-2. I/O and Multiplier Registers

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

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

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

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

Register Type	Register(s)	Function
IOP registers (SPORTs)	STCTL0, SRCTL0, TX0, RX0, TDIV0, RDIV0, MTCS0, MRCS0, MTCCS0, MRCCS0, SPATH0, KEYWD0, KEYMASK0	SPORT 0 registers
	STCTL1, SRCTL1, TX1, RX1, TDIV1, RDIV1, MTCS, MRCS1, MTCCS1, MRCCS1, SPATH1, KEYWD1, KEYMASK1	SPORT 1 registers
Multiplier registers	MR, MR0, MR1, MR2,	Multiplier results
	MRF, MR0F, MR1F, MR2F	Multiplier results, foreground
	MRB, MR0B, MR1B, MR2B	Multiplier results, background

Memory Addressing Summary

ADSP-21160 DSPs support the following types of addressing.

Direct Addressing

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

```
dm(0x000015F0) = astat;
```

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

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

```
call(pc,10), r0=r6+r3;
```

```
do(pc,length) until sz;   {'length' is a variable}
```

Indirect Addressing (using DAG registers):

Post-modify with M register, update I register (Instruction Types 1, 3, 6, 16)

```
f5=pm(i9,m12);

dm(i0,m3)=r3, r1=pm(i15,m10);
```

Pre-modify with M register, no update (Instruction Types 3, 9, 10)

```
r1=pm(m10,i15);

jump(m13,i11);
```

Post-modify with immediate value, update I register (Instruction Type 4)

```
f15=dm(i0,6);

if av r1=pm(i15,0x11);
```

Pre-modify with immediate value, no update (Instruction Types 4, 15)

```
if av r1=pm(0x11,i15);

dm(127,i5)=laddr;
```

Instruction Set Notation Summary

The conventions for ADSP-210xx instruction syntax descriptions appear in Table 2-3 on page 2-16. Other parts of the instruction syntax and opcode information also appear in this section.

Table 2-3. Instruction Set Notation

Notation	Meaning
UPPERCASE	Explicit syntax—assembler keyword (notation only; assembler is case-insensitive and lowercase is the preferred programming convention)
;	Semicolon (instruction terminator)
,	Comma (separates parallel operations in an instruction)
italics	Optional part of instruction

`| option1 |`
`| option2 |`
	List of options between vertical bars (choose one)

compute	ALU, multiplier, shifter or multifunction operation (see the chapter “Computations Reference”).
shiftimm	Shifter immediate operation (see the chapter “Computations Reference”).
cond	Status condition (see condition codes in Table 2-4 on page 2-18)
termination	Loop termination condition (see condition codes in Table 2-4 on page 2-18)
ureg	Universal register
cureg	Complementary universal register (see Table 2-10 on page 2-28)
sreg	System register
csreg	Complementary system register (see Table 2-10 on page 2-28)
dreg	Data register (register file): R15-R0 or F15-F0
cdreg	Complementary data register (register file): R15-R0 or F15-F0 (see Table 2-10 on page 2-28)
creg	One of 32 cache entries, an entry consisting of a CH, CL, & CA

Table 2-3. Instruction Set Notation (Cont'd)

Notation	Meaning
Ia	I7-I0 (DAG1 index register)
Mb	M7-M0 (DAG1 modify register)
Ic	I15-I8 (DAG2 index register)
Md	M15-M8 (DAG2 modify register)
<datan>	n-bit immediate data value
<addrn>	n-bit immediate address value
<reladdrn>	n-bit immediate PC-relative address value
+1	the incremented data, address or register value
(DB)	Delayed branch
(LA)	Loop abort (pop loop and PC stacks on branch)
(CI)	Clear interrupt
(LR)	Loop reentry
(LW)	Long Word (forces Long word access in Normal word range)

Conditional Execution Codes Summary

In a conditional instruction, execution of the entire instruction depends on the specified `condition` (`cond` or `terminate`). Table 2-4 lists the codes that you can use in conditionals (IF and DO UNTIL).

Table 2-4. IF Condition and Do/Until Termination Mnemonics

Condition From	Description	True if…	Mnemonic
ALU	ALU = 0	AZ = 1	EQ
	ALU ≠ 0	AZ = 0	NE
	ALU > 0	footnote[1]	GT
	ALU < zero	footnote[2]	LT
	ALU ≥ 0	footnote[3]	GE
	ALU ≤ 0	footnote[4]	LE
	ALU carry	AC = 1	AC
	ALU not carry	AC = 0	NOT AC
	ALU overflow	AV = 1	AV
	ALU not overflow	AV = 0	NOT AV
Multiplier	Multiplier overflow	MV = 1	MV
	Multiplier not overflow	MV= 0	NOT MV
	Multiplier sign	MN = 1	MS
	Multiplier not sign	MN = 0	NOT MS
Shifter	Shifter overflow	SV = 1	SV
	Shifter not overflow	SV = 0	NOT SV
	Shifter zero	SZ = 1	SZ
	Shifter not zero	SZ = 0	NOT SZ

Table 2-4. IF Condition and Do/Until Termination Mnemonics

Condition From	Description	True if…	Mnemonic
Bit Test	Bit test flag true	BTF = 1	TF
	Bit test flag false	BTF = 0	NOT TF
Flag Input	Flag0 asserted	FI0 = 1	FLAG0_IN
	Flag0 not asserted	FI0 = 0	NOT FLAG0_IN
	Flag1 asserted	FI1 = 1	FLAG1_IN
	Flag1 not asserted	FI1 = 0	NOT FLAG1_IN
	Flag2 asserted	FI2 = 1	FLAG2_IN
	Flag2 not asserted	FI2 = 0	NOT FLAG2_IN
	Flag3 asserted	FI3 = 1	FLAG3_IN
	Flag3 not asserted	FI3 = 0	NOT FLAG3_IN
Mode	Bus master true		BM
	Bus master false		NOT BM
Sequencer	Loop counter expired (Do)	CURLCNTR = 1	LCE
	Loop counter not expired (If)	CURLCNTR ≠ 1	NOT ICE
	Always false (Do)	Always	FOREVER
	Always true (If)	Always	TRUE

1 ALU greater than (GT) is true if: [$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN)] or $\overline{AZ}$ = 0

2 ALU less than (LT) is true if: [$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN and $\overline{AZ}$)] = 1

3 ALU greater equal (GE) is true if: [$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN and $\overline{AZ}$)] = 0

4 ALU lesser or equal (LT) is true if: [$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN)] or $\overline{AZ}$ = 1

SISD/SIMD Conditional Testing Summary

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

- In conditional computation (If … Compute) instructions, each processing element executes the computation based on evaluating the condition in that processing element.
- In conditional program control (If … Jump/Call) instructions, the program sequencer executes the Jump/Call based on a logical AND of the conditions in both processing elements.
- In conditional computation instructions with an Else clause, each processing element executes the Else computation based on evaluating the inverse of the condition (Not Cond) in that processing element.

Table 2-5 on page 2-20 and Table 2-6 on page 2-21 compare SISD and SIMD If-Else conditional execution, which are available in the Type 9, 10, and 11 instructions.

Table 2-5. SISD Mode Conditional Execution

Conditional test	ELSE modifier	Results for Type 11 (RTS)
0 (false)	0 (without else)	rts nops, compute nops
0 (false)	1 (else)	rts nops, compute executes
1 (true)	0 (without else)	rts executes, compute executes
1 (true)	1 (else)	rts executes, compute nops

Table 2-6. SIMD Mode Conditional Execution

Conditional test		Else modifier	Results for Type 11 (RTS)
PEx	PEy		
0	0	0	rts nops, pex compute nops, pey compute nops
0	1	0	rts nops, pex compute nops, pey compute executes
1	0	0	rts nops, pex compute exe., pey compute nops
1	1	0	rts exe., pex compute exe., pey compute exe.
0	0	1	rts nops, pex compute exe., pey compute exe.
0	1	1	rts nops, pex compute exe., pey compute nops
1	0	1	rts nops, pex compute nops, pey compute exe.
1	1	1	rts exe., pex compute nops, pey compute nops

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

- "Type 9: Indirect Jump | Call, Compute" on page 4-8
- "Type 10: Indirect Jump | Compute, dreg«···»DM" on page 4-15
- "Type 11: Return From Subroutine | Interrupt, Compute" on page 4-21

Instruction Opcode Acronym Summary

In ADSP-21160 DSP opcodes, some bits are explicitly defined to be zeros or ones. The values of other bits or fields set various parameters for the instruction. The terms in Table 2-7 define these opcode bits and fields. Unspecified bits are ignored when the processor decodes the instruction, but are reserved for future use.

Table 2-7. Opcode Acronyms

Bit/Field	Description	States	
A	Loop abort code	0	Do not pop loop, PC stacks on branch
		1	Pop loop, PC stacks on branch
ADDR	Immediate address field		
AI	Computation unit register	0000	MR0F
		0001	MR1F
		0010	MR2F
		0100	MR0B
		0101	MR1B
		0110	MR2B
B	Branch type	0	Jump
		1	Call

Table 2-7. Opcode Acronyms (Cont'd)

Bit/Field	Description	States	
BOP	Bit Operation select codes	000	Set
		001	Clear
		010	Toggle
		100	Test
		101	XOR
COMPUTE	Compute operation field (see "Computations Reference" on page 7-1)		
COND	Status Condition codes	0–31	
CI	Clear interrupt code	0	Do not clear current interrupt
		1	Clear current interrupt
CREG	Instruction cache entry	0–31	
CS	Instruction cache register select code	00	Lower half of instruction RAM entry
		01	Upper half of instruction RAM entry
		11	Address CAM entry
CU	Computation unit select codes	00	ALU
		01	Multiplier
		10	Shifter
DATA	Immediate data field		
DEC	Counter decrement code	0	No counter decrement
		1	Counter decrement

Table 2-7. Opcode Acronyms (Cont'd)

Bit/Field	Description	States	
DMD	Memory access direction	0 1	Read Write
DMI	Index (I) register numbers, DAG1	0–7	
DMM	Modify (M) register numbers, DAG1	0–7	
DREG	Register file locations	0–15	
E	ELSE clause code	0 1	No ELSE clause ELSE clause
FC	Flush cache code	0 1	No cache flush Cache flush
G	DAG/Memory select	0 1	DAG1 or Data Memory DAG2 or Program Memory
INC	Counter increment code	0 1	No counter increment Counter increment
J	Jump Type	0 1	Non-delayed Delayed
L	Long Word memory address	0 1	Access size based on memory map Long word (64-bit) access size
LPO	Loop stack pop code	0 1	No stack pop Stack pop
LPU	Loop stack push code	0 1	No stack push Stack push

Table 2-7. Opcode Acronyms (Cont'd)

Bit/Field	Description	States
LR	Loop reentry code	0 No loop reentry 1 Loop reentry
NUM	Interrupt vector	0–7
PMD	Memory access direction	0 Read 1 Write
PMI	Index (I) register numbers, DAG2	8–15
PMM	Modify (M) register numbers, DAG2	8–15
PPO	PC stack pop code	0 No stack pop 1 Stack pop
PPU	PC stack push code	0 No stack push 1 Stack push
RELADDR	PC-relative address field	
S	UREG transfer/instruction cache read-load select	0 instruction cache read-load 1 ureg transfer
SPO	Status stack pop code	0 No stack pop 1 Stack pop
SPU	Status stack push code	0 No stack push 1 Stack push
SREG	System Register code	0–15 (see "Universal Register Codes" on page 2-26)
TERM	Termination Condition codes	0–31
U	Update, index (I) register	0 Pre-modify, no update 1 Post-modify with update

Table 2-7. Opcode Acronyms (Cont'd)

Bit/Field	Description	States
UREG	Universal Register code	0–256 (see "Universal Register Codes" on page 2-26)
RA, RM, RN, RS, RX, RY	Register file locations for compute operands and results	0–15
RXA	ALU x-operand register file location for multifunction operations	8–11
RXM	Multiplier x-operand register file location for multifunction operations	0–3
RYA	ALU y-operand register file location for multifunction operations	12–15
RYM	Multiplier y-operand register file location for multifunction operations	4–7

Universal Register Codes

Table 2-8, Table 2-9 on page 2-27, Table 2-10 on page 2-28, and Table 2-11 on page 2-30 in this section list the bit codes for register that appear within opcode fields.

Table 2-8. Universal Registers

Register	Description
PC	program counter
PCSTK	top of PC stack
PCSTKP	PC stack pointer
FADDR	fetch address
DADDR	decode address
LADDR	loop termination address
CURLCNTR	current loop counter

Table 2-8. Universal Registers (Cont'd)

Register	Description
LCNTR	loop counter
R15–R0	X element register file locations
S15–S0	Y element register file locations
I15–I0	DAG1 and DAG2 index registers
M15–M0	DAG1 and DAG2 modify registers
L15–L0	DAG1 and DAG2 length registers
B15–B0	DAG1 and DAG2 base registers
PX	48-bit PX1 and PX2 combination
PX1	bus exchange 1 (16 bits)
PX2	bus exchange 2 (32 bits)
TPERIOD	timer period
TCOUNT	timer counter

Table 2-9. Universal and System Registers

Register	Description
MODE1	mode control 1
MODE2	mode control 2
IRPTL	interrupt latch
IMASK	interrupt mask
IMASKP	interrupt mask pointer
MMASK	Mode mask
FLAGS	Flag pins input/output state
ASTATx	X element arithmetic status

Table 2-9. Universal and System Registers (Cont'd)

Register	Description
STKYx	X element sticky status
ASTATy	Y element arithmetic status
STKYy	Y element sticky status
USTAT1	user status reg 1
USTAT2	user status reg 2
USTAT3	user status reg 3
USTAT4	user status reg 4

Table 2-10. Complementary Registers (Ureg–Cureg)

Register Type	SIMD Mode Complementary Registers
Data register (dreg & ureg)	R0–S0 R1–S1 R2–S2 R3–S3 R4–S4 R5–S5 R6–S6 R7–S7 R8–S8 R9–S9 R10–S10 R11–S11 R12–S12 R13–S13 R14–S14 R15–S15

Table 2-10. Complementary Registers (Ureg–Cureg)

Register Type	SIMD Mode Complementary Registers
System register (sreg & ureg)	USTAT1–USTAT2 USTAT3–USTAT4 ASTATx–ASTATy STKYx–STKYy
Bus exchange register (ureg)	PX1–PX2

Table 2-11 shows how Ureg register codes appear to PEx.

Table 2-11. Processing Element X Universal Register Codes (SISD/SIMD)

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

Table 2-12 shows how Ureg register codes appear to PEy.

Table 2-12. Processing Element Y Universal Register Codes (SIMD)

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

ADSP-21160 Instruction Opcode Map

Table 2-13. ADSP-21160 DSP Opcodes (Bits 47–27)

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

Table 2-14. ADSP-21160 DSP Opcodes (Bits 26–0)

26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PM DREG				COMPUTE																						
				COMPUTE																						
<UREG				COMPUTE																						
DREG				COMPUTE																						
<DEST UREG				COMPUTE																						
X DREG				COMPUTE																						
DREG				0	SHIFTOP						DATA								RN				RX			
26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table 2-15. ADSP-21160 DSP Opcodes (Bits 47–27)

Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27
(b) "Type 6: Immediate Shift, dreg«···»DM | PM"	000			00010							COND							DATAEX			
"Type 7: Compute, modify"	000			00100						G	COND					I			M		
(a) "Type 8: Direct Jump | Call"	000			00110					B	A	COND										
(b) "Type 8: Direct Jump | Call"	000			00111					B	A	COND										
(a) "Type 9: Indirect Jump | Call, Compute"	000			01000					B	A	COND					PMI			PMM		
(b) "Type 9: Indirect Jump | Call, Compute"	000			01001					B	A	COND					RELADDR					
(a) "Type 10: Indirect Jump | Compute, dreg«···»DM"	110			D	DMI			DMM			COND					PMI			PMM		
Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27

Table 2-16. ADSP-21160 DSP Opcodes (Bits 26–0)

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

Table 2-17. ADSP-21160 DSP Opcodes (Bits 47–27)

Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27
(b) "Type 10: Indirect Jump | Compute, dreg«···»DM"	111			D	DMI			DMM			COND					RELADDR					
(a) "Type 11: Return From Subroutine | Interrupt, Compute"	000			01010							COND										
(b) "Type 11: Return From Subroutine | Interrupt, Compute"	000			01011							COND										
(a) "Type 12: Do Until Counter Expired"	000			01100					DATA>												
(b) "Type 12: Do Until Counter Expired"	000			01101					0	UREG											
"Type 13: Do Until"	000			01110							TERM										
"Type 14: Ureg«···»DM | PM (direct addressing)"	000			100			G	D	L	UREG							ADDR (upper 5 bits)				
Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27

Table 2-18. ADSP-21160 DSP Opcodes (Bits 26–0) (Cont'd)

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

Table 2-19. ADSP-21160 DSP Opcodes (Bits 47–27)

Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27
"Type 15: Ureg«···»DM | PM (indirect addressing)"	101			G	I			D	L	UREG							DATA (upper 5 bits)				
"Type 16: Immediate data···»DM | PM"	100			1	I			M			G						DATA (upper 5 bits)				
"Type 17: Immediate data···»Ureg"	000			01111					0	UREG							DATA (upper 5 bits)				
"Type 18: System Register Bit Manipulation"	000			10100					BOP				SREG				DATA (upper 5 bits)				
(a) "Type 19: I Register Modify | Bit-Reverse"	000			10110					0	G				I			DATA (upper 5 bits)				
(b)"Type 19: I Register Modify | Bit-Reverse"	000			10110					1	G				I			DATA (upper 5 bits)				
"Type 20: Push, Pop Stacks, Flush Cache"	000			10111					LPU	LPO	SPU	SPO	PPU	PPO	FC						
Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27

Table 2-20. ADSP-21160 DSP Opcodes (Bits 26–0) (Cont'd)

26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA (lower 27 bits)																										
DATA (lower 27 bits)																										
DATA (lower 27 bits)																										
DATA (lower 27 bits)																										
DATA (lower 27 bits)																										
DATA (lower 27 bits)																										
26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table 2-21. ADSP-21160 DSP Opcodes (Bits 47–27)

Instruction Type	**47**	**46**	**45**	**44**	**43**	**42**	**41**	**40**	**39**	**38**	**37**	**36**	**35**	**34**	**33**	**32**	**31**	**30**	**29**	**28**	**27**
“Type 21: Nop”	000			00000					0												
“Type 22: Idle”	000			00000					1												
Type 23: Idle16	Not supported on ADSP-21160																				
Type 24: creg«···»ureg	Not documented on ADSP-21160																				
(a) “Type 25: Cjump/Rframe”	0001				1000				0000				0100				0000				0
(b) “Type 25: Cjump/Rframe”	0001				1000				0100				0100				0000				0
(c) “Type 25: Cjump/Rframe”	0001				1001				0000				0000				0000				0
Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27

Table 2-22. ADSP-21160 DSP Opcodes (Bits 26–0)

26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
000			ADDR																							
000			RELADDR																							
000			0000				0000				0000				0000				0000				0000			
26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

3 COMPUTE AND MOVE

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

Group I Instructions

The instructions in this group contain a `COMPUTE` field that specifies a compute operation using the ALU, multiplier, or shifter. Because there are a large number of options available for computations, these operations are described separately in the "Computations Reference" on page 7-1. Note that data moves between the `MR` registers and the register file are considered multiplier operations and are covered in the "Computations Reference" on page 7-1. Group I instructions include the following.

- "Type 1: Compute, Dreg«···»DM | Dreg«···»PM" on page 3-3

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

- "Type 2: Compute" on page 3-7

  Compute operation, optional condition

- "Type 3: Compute, ureg«···»DM | PM, register modify" on page 3-9

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

- “Type 4: Compute, dreg«…»DM | PM, data modify” on page 3-14

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

- “Type 5: Compute, ureg«…»ureg | Xdreg<->Ydreg” on page 3-19

  Transfer between two universal registers, optional condition, optional compute operation

- “Type 6: Immediate Shift, dreg«…»DM | PM” on page 3-23

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

- “Type 7: Compute, modify” on page 3-28

  Index register modify, optional condition, optional compute operation

Type 1: Compute, Dreg«··»DM | Dreg«··»PM

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

Syntax

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

Function (SISD)

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

Function (SIMD)

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

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

The I values are post-modified and updated by the specified M registers. Pre-modify offset addressing is not supported. For more information on register restrictions, see the “Data Address Generators” chapter of the *ADSP-21160 SHARC DSP Hardware Reference*.

The X element uses the specified Dreg registers, and the Y element uses the complementary registers (Cdreg) that correspond to the Dreg registers. For a list of complementary registers, see Table 2-10 on page 2-28.

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

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

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

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

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

Do not use the pseudo code above as instruction syntax.

Examples

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

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

When the ADSP-21160 DSP is in SIMD, the first instruction in this example performs the same computation and performs two writes in parallel on both PEx and PEy. The `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`.

```
R0=DM(I1,M1);
```

When the ADSP-21160 processor is in broadcast from the `BDCST1` bit being 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.

Type 1 Opcode

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
001			DMD	DMI			DMM			PMD	DM DREG				PMI			PMM			PM DREG			

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

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 data and program memory
DMM, PMM	Specify M registers used to update the I registers
COMPUTE	Defines a compute operation to be performed in parallel with the data accesses; if omitted, this is a NOP

Type 2: Compute

Compute operation, optional condition

Syntax

```
IF  COND compute  ;
```

Function (SISD)

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

Function (SIMD)

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

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

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

```
IF  PEx COND compute  ;
```

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

```
IF  PEy COND compute  ;
```

Do not use the pseudo code above as instruction syntax.

Examples

```
IF MV R6=SAT MRF (UI);
```

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

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

Type 2 Opcode

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
000			00001							COND														

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

Bits	Description
COND	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.

Type 3: Compute, ureg«··»DM | PM, register modify

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

Syntax

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

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

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

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

Function (SISD)

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

Function (SIMD)

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

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

For the universal register, the X element uses the specified Ureg register, and the Y element uses the corresponding complementary register (Cureg). For a list of complementary registers, see Table 2-10 on page 2-28. Note that the Ureg may not be from the same DAG (DAG1 or DAG2) as `Ia/Mb` or `Ic/Md`.

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

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

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

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

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

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

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

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

```
IF PEy COND compute    |, DM(Ia+1, 0)      |= cureg (LW);
                       |, PM(Ic+1, 0)      |

                       |, DM(Mb+1, Ia)     |= cureg (LW);
                       |, PM(Md+1, Ic)     |

                        , cureg =          |DM(Ia+1, 0) (LW);    |
                                           |PM(Ic+, 0) (LW);     |

                        , cureg =          |DM(Mb+1, Ia) (LW);   |
                                           |PM(Md+1, Ic) (LW);   |
```

Do not use the pseudo code above as instruction syntax.

Examples

```
R6=R3-R11, DM(I0,M1)=ASTATx;
IF NOT SV F8=CLIP F2 BY F14, F7=PM(I12,M12);
```

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

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

```
IF NOT SV F8=CLIP F2 BY F14, F7=PM(I9,M12);
```

When the ADSP-21160 DSP is in broadcast from the `BDCST9` bit being 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`.

Type 3 Opcode

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
010			U	I			M			COND					G	D	L	UREG						

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

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

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

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

Syntax

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

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

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

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

Function (SISD)

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

Function (SIMD)

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

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

For the data register, the X element uses the specified Dreg register, and the Y element uses the corresponding complementary register (Cdreg). For a list of complementary registers, see Table 2-10 on page 2-28.

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

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

```
SIMD Explicit Operation (PEx Operation Stated in the Instruction Syntax)
IF PEx COND compute     |, DM(Ia, <data6>)        | = dreg ;
                        |, PM(Ic, <data6>)        |

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

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

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

SIMD Implicit Operation (PEy Operation Implied by the Instruction Syntax)
IF PEy COND compute     |, DM(Ia+1, 0)            | = cdreg ;
                        |, PM(Ic+1, 0)            |

                        |, DM(<data6>+1, Ia)      |= cdreg ;
                        |, PM(<data6>+1, Ic)      |

                         , cdreg  =               |DM(Ia+1, 0)  ;            |
                                                  |PM(Ic+1, 0)  ;            |

                         , cdreg  =               |DM(<data6>+1, Ia)  ;      |
                                                  |PM(<data6>+1, Ic)  ;      |
```

Do not use the pseudo code above as instruction syntax.

Examples

```
IF FLAG0_IN F1=F5*F12, F11=PM(I10,6);
R12=R3 AND R1, DM(6,I1)=R6;
```

When the ADSP-21160 is in SISD, the computation and program memory read in the first instruction are performed in PEx if the condition's outcome is true. The second instruction stores the result of the logical AND in `R12` and writes the value within `R6` into data memory.

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

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

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

Type 4 Opcode

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
011			0	I			G	D	U	COND					DATA						DREG			

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

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

Bits	Description
COND	Specifies the test condition; if omitted, COND is TRUE
D	Selects the access Type (read or write)
G	Selects data memory or program memory
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
COMPUTE	Defines a compute operation to be performed in parallel with the data access; if omitted, this is a NOP

Type 5: Compute, ureg«··»ureg | Xdreg<->Ydreg

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

Syntax

```
IF COND compute,        |ureg1 = ureg2          | ;

                        |X dreg <-> Y dreg      |
```

Function (SISD)

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

Function (SIMD)

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

In the transfer (=), the X element transfers between the universal registers Ureg1 and Ureg2, and the Y element transfers between the complementary universal registers Cureg1 and Cureg2. For a list of complementary registers, see Table 2-10 on page 2-28.

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

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

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

```
IF PEx COND compute,    |ureg1 = ureg2           | ;

                        |X dreg <-> Y dreg       |
```

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

```
IF PEy COND compute,    |cureg1 = cureg2         | ;

                        |{no implicit operation} |
```

Do not use the pseudo code above as instruction syntax.

Examples

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

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

When the ADSP-21160 DSP is in SIMD, the condition is evaluated on each processing element, PEx and PEy, independently. The computation executes on both PE's, either one PE, or neither PE dependent on the outcome of the condition. For the register transfer to complete, the condition must be satisfied in both PEx and PEy. The second instruction initializes the loop counter independent of the outcome of the first instruction's condition. The third instruction swaps the register contents between `R0` and `S1`—the SISD and SIMD swap operation is the same.

Type 5 Opcode (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						

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

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

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	1		Y DREG				COND											X DREG			

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

Type 5: Compute, ureg«··»ureg | Xdreg<->Ydreg

Bits	Description
COND	Specifies the test condition; if omitted, COND is TRUE
SRC UREG	Identifies the universal register source. (highest 5 bits of register code)
SU	Identifies the universal register source. (lowest 2 bits of register code)
DEST UREG	Identifies the universal register destination
Y DREG	Identifies the PEy data registers for swap (must appear to right of swap operator)
X DREG	Identifies the PEx data register for swap (must appear to left of swap operator)
COMPUTE	Defines a compute operation to be performed in parallel with the data transfer; if omitted, this is a NOP

Type 6: Immediate Shift, dreg«··»DM | PM

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

Syntax

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

                          , dreg =            |DM(Ia, Mb) ;          |
                                              |PM(Ic, Md) ;          |
```

Function (SISD)

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

For more information on shifter operations, see "Shifter Operations" on page 7-64. For more information on register restrictions, see the "Data Address Generators" chapter of the *ADSP-21160 SHARC DSP Hardware Reference*.

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

Function (SIMD)

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

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

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

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

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

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

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

                          , dreg =          |DM(Ia, Mb)  ;       |
                                            |PM(Ic, Md)  ;       |
```

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

```
IF PEy COND shiftimm    | , DM(Ia+1, 0)     | = cdreg ;
                        | , PM(Ic+1, 0)     |

                          , cdreg =         |DM(Ia+1, 0)  ;      |
                                            |PM(Ic+1, 0)  ;      |
```

Do not use the pseudo code above as instruction syntax.

Examples

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

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

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

```
R2 = LSHIFT R6 BY 0x4, F3=DM(I1,M3);
```

When the ADSP-21160 DSP is in broadcast from the `BDCST1` bit being set in the `MODE1` system register, the computation is performed, the result is stored in `R2`, and the data memory value is read into register `F3` via the `I1` register from DAG1. The `SF3` register is also loaded with the same value from data memory as `F3`.

Type 6: Immediate Shift, dreg«··»DM | PM

Type 6 Opcode (with data access)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
100			0	I			M			COND					G	D	DATAEX				DREG			

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

Type 6 Opcode (without data access)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
000			00010							COND							DATAEX							

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

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

Type 7: Compute, modify

Index register modify, optional condition, optional compute operation

Syntax

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

Function (SISD)

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

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

Function (SIMD)

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

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

The following pseudo code compares the Type 7 instruction’s explicit and implicit operations in SIMD mode.

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

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

SIMD **Implicit** Operation (PEy Operation **Implied** by the InstructionSyntax)

```
IF PEy COND   compute     {no implied MODIFY operation}
```

Do not use the pseudo code above as instruction syntax.

Examples

```
IF NOT FLAG2_IN R4=R6*R12(SUF), MODIFY(I10,M8);
IF NOT LCE MODIFY(I3,M1);
```

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

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

Type 7: Compute, modify

Type 7 Opcode

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
000			00100						G	COND					I			M						

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

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

4 PROGRAM FLOW CONTROL

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

Group II Instructions

The instructions in this group contain a `COMPUTE` field that specifies a compute operation using the ALU, multiplier, or shifter. Because there are a large number of options available for computations, these operations are described separately in the "Computations Reference" on page 7-1. Note that data moves between the `MR` registers and the register file are considered multiplier operations and are covered in the "Computations Reference" on page 7-1. Group II instructions include the following.

- "Type 8: Direct Jump | Call" on page 4-3

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

- "Type 9: Indirect Jump | Call, Compute" on page 4-8

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

- "Type 10: Indirect Jump | Compute, dreg«···»DM" on page 4-15

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

- “Type 11: Return From Subroutine | Interrupt, Compute” on page 4-21

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

- “Type 12: Do Until Counter Expired” on page 4-26

  Load loop counter, do loop until loop counter expired

- “Type 13: Do Until” on page 4-28

  Do until termination

Type 8: Direct Jump | Call

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

Syntax

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

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

Function (SISD)

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

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

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

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

When returning from a reduced subroutine, you must use the (LR) modifier of the RTS if the interrupt occurs during the last two instructions of a loop. For related information, see “Type 11: Return From Subroutine | Interrupt, Compute” on page 4-21.

Function (SIMD)

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

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

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

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

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

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

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

{No explicit PEx operation}

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

{No implicit PEy operation}

Do not use the pseudo code above as instruction syntax.

Examples

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

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

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

Type 8 Opcode (with direct branch)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
000			00110					B	A	COND											J		CI

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

Type 8 Opcode (with PC-relative branch)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
000			00111					B	A	COND											J		CI

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

Bits	Description
COND	Specifies the test condition; if omitted, COND is TRUE
B	Selects the branch type, jump or call. For calls, A and CI are ignored
J	Determines whether the branch is delayed or non-delayed
ADDR	Specifies a 24-bit program memory address
A	Activates loop abort
CI	Activates clear interrupt
RELADDR	Holds a 24-bit, twos-complement value that is added to the current PC value to generate the branch address

Type 9: Indirect Jump | Call, Compute

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

Syntax

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

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

Function (SISD)

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

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

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

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

When returning from a reduced subroutine, you must use the (LR) modifier of the RTS instruction if the interrupt occurs during the last two instructions of a loop. For related information, see "Type 11: Return From Subroutine | Interrupt, Compute" on page 4-21.

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

Function (SIMD)

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

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

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

Note that for the `compute`, the X element uses the specified registers and the Y element uses the complementary registers. For a list of complementary registers, see Table 2-10 on page 2-28.

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

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

IF (PEx AND PEy COND) JUMP	(Md, Ic)	(DB)	, (if PEx COND) compute	;
	(PC, <reladdr6>)	(LA)	, ELSE (if NOT PEx) compute	
		(CI)		
		(DB, LA)		
		(DB, CI)		

IF (PEx AND PEy COND) CALL	(Md, Ic)	(DB)	, (if PEx COND) compute	;
	(PC, <reladdr6>)		, ELSE (if NOT PEx) compute	

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

IF (PEx AND PEy COND) JUMP	(Md, Ic)	(DB)	, (if PEy COND) compute	;
	(PC, <reladdr6>)	(LA)	, ELSE (if NOT PEy) compute	
		(CI)		
		(DB, LA)		
		(DB, CI)		

IF (PEx AND PEy COND) CALL	(Md, Ic)	(DB)	, (if PEy COND) compute	;
	(PC, <reladdr6>)		, ELSE (if NOT PEy) compute	

Do not use the pseudo code above as instruction syntax.

Examples

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

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

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

For a summary of SISD/SIMD conditional testing, see "SISD/SIMD Conditional Testing Summary" on page 2-20.

Type 9 Opcode (with indirect branch)

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

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 9 Opcode (with PC-relative branch)

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

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

Bits	Description
COND	Specifies the test condition; if omitted, COND is true
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
CI	Activates clear interrupt
COMPUTE	Defines a compute operation to be performed in parallel with the data access; if omitted, this is a NOP
RELADDR	Holds a 6-bit, twos-complement value that is added to the current PC value to generate the branch address
PMI	Specifies the I register for indirect branches. The I register is pre-modified but not updated by the M register.
PMM	Specifies the M register for pre-modifies

Type 10: Indirect Jump | Compute, dreg«··»DM

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

Syntax

IF COND Jump	(Md, Ic)	,Else	compute, DM(Ia, Mb) = dreg ;
	(PC, <reladdr6>)		compute, dreg = DM(Ia, Mb) ;

Function (SISD)

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

The PC-relative address for the Jump is a 6-bit, 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, you may not use the delay branch (DB), loop abort (LA), or clear interrupt (CI) modifiers.

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

Function (SIMD)

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

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

For this instruction, the If `condition` and Else keywords are not optional and must be used. If the specified `condition` is true in both processing elements, the Jump is executed. The 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 2-10 on page 2-28. 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.

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

IF (PEx AND PEy COND) Jump	(Md, Ic) (PC, <reladdr6>)	,Else (if NOT PEx)	compute, DM(Ia, Mb) = dreg ; compute, dreg = DM(Ia, Mb) ;

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

IF (PEx AND PEy COND) Jump	(Md, Ic) (PC, <reladdr6>)	,Else (if NOT PEy)	compute, DM(Ia, Mb) = dreg ; compute, dreg = DM(Ia, Mb) ;

Do not use the pseudo code above as instruction syntax.

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 ADSP-21160 processor is in SISD, the indirect jump in the first instruction is performed if the condition tests true. Otherwise, `R6` stores the value of a data memory read. The second instruction is much like the first, however, it also includes an optional compute, which is performed in parallel with the data memory read.

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

The second instruction is much like the first instruction. The second instruction, however, includes an optional compute also performed in parallel with the data memory read independently on either PEx or PEy and based on the negative evaluation of the condition code seen by that processing element.

For a summary of SISD/SIMD conditional testing, see "SISD/SIMD Conditional Testing Summary" on page 2-20.

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

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

Type 10 Opcode (with indirect jump)

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

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

Type 10 Opcode (with PC-relative jump)

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

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

Bits	Description
COND	Specifies the condition to test; not optional
PMI	Specifies the I register for indirect branches. The I register is premodified, but not updated by the M register.
PMM	Specifies the M register for pre-modifies
D	Selects the data memory access Type (read or write)
DREG	Specifies the register file location
DMI	Specifies the I register that is post-modified and updated by the M register

Bits	Description
DMM	Identifies the M register for post-modifies
COMPUTE	Defines a compute operation to be performed in parallel with the data access; if omitted, this is a NOP
RELADDR	Holds a 6-bit, twos-complement value that is added to the current PC value to generate the branch address

Type 11: Return From Subroutine | Interrupt, Compute

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

Syntax

```
IF  COND  RTS        |(DB)        |    |, compute          | ;
                     |(LR)        |    |, ELSE  compute    |
                     |(DB, LR)    |

IF  COND  RTI         (DB)             |, compute          | ;
                                       |, ELSE  compute    |
```

Function (SISD)

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

- Pops status stack if the `ASTAT` and `MODE1` status registers have been pushed—if the interrupt was `IRQ2-0`, the timer interrupt, or the `VIRPT` vector interrupt
- Clears the appropriate bit in the interrupt latch register (`IRPTL`) and the interrupt mask pointer (`IMASKP`)

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

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

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

You must also use the (LR) modifier for RTS when returning from a subroutine that has been reduced from an interrupt service routine with a Jump (CI) instruction. This case occurs when the interrupt occurs during the last two instructions of a loop. For a description of the Jump (CI) instruction, see "Type 8: Direct Jump | Call" on page 4-3 or "Type 9: Indirect Jump | Call, Compute" on page 4-8.

Function (SIMD)

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

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

Note that for the `compute`, the X element uses the specified registers, and the Y element uses the complementary registers. For a list of complementary registers, see Table 2-10 on page 2-28.

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

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

```
IF  (PEx AND PEy COND)  RTS   |(DB)     |  |, (if PEx COND) compute        | ;
                              |(LR)     |  |, ELSE (if NOT PEx) compute    |
                              |(DB, LR) |

IF  (PEx AND PEy COND)  RTI    (DB)        |, (if PEx COND) compute        | ;
                                           |, ELSE (if NOT PEx) compute    |
```

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

```
IF  (PEx AND PEy COND)  RTS   |(DB)     |  |, (if PEy COND) compute        | ;
                              |(LR)     |  |, ELSE (if NOT PEy) compute    |
                              |(DB, LR) |

IF  (PEx AND PEy COND)  RTI    (DB)        |, (if PEy COND) compute        | ;
                                           |, ELSE (if NOT PEy) compute    |
```

Do not use the pseudo code above as instruction syntax.

Examples

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

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

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

For a summary of SISD/SIMD conditional testing, see "SISD/SIMD Conditional Testing Summary" on page 2-20.

Type 11 Opcode (return from subroutine)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
000			01010							COND											J	E	LR	

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

Type 11 Opcode (return from interrupt)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
000			01011							COND											J	E		

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

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

Type 12: Do Until Counter Expired

Load loop counter, do loop until loop counter expired

Syntax

```
LCNTR =   |<data16>   | , DO   |<addr24>              | UNTIL LCE;
          |ureg       |        |(PC, <reladdr24>)     |
```

Function (SISD and SIMD)

In SISD or SIMD modes, the Type 12 instruction sets up a counter-based program loop. The loop counter `LCNTR` is loaded with 16-bit immediate data or from a universal register. The loop start address is pushed on the `PC` stack. The loop end address and the `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 `PC`-relative 24-bit twos-complement address. The `LCNTR` is pushed on the loop counter stack and becomes the `CURLCNTR` value. The loop executes until the `CURLCNTR` reaches zero.

Examples

```
LCNTR=100, DO fmax UNTIL LCE;   {fmax is a program label}
LCNTR=R12, DO (PC,16) UNTIL LCE;
```

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

Type 12 Opcode (with immediate loop counter load)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
000			01100					DATA															

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

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

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
000			01101					0	UREG														

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

Bits	Description
RELADDR	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.
DATA	Specifies a 16-bit value to load into the loop counter (LCNTR) for an immediate load.
UREG	Specifies a register containing a 16-bit value to load into the loop counter (LCNTR) for a load from an universal register.

Type 13: Do Until

Do until termination

Syntax

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

Function (SISD)

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

Function (SIMD)

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

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

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

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

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

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

Do not use the pseudo code above as instruction syntax.

Examples

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

When the ADSP-21160 processor is in SISD, the `end` program label in the first instruction specifies the start address for the loop, and the loop is executed until the instruction's condition tests true. In the second 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 ADSP-21160 processor is in SIMD, the `end` program label in the first instruction specifies the start address for the loop, and the loop is executed until the instruction's condition tests true in both PEx or PEy. In the second instruction, the start address is given in the form of a PC-relative address. The loop executes until the instruction's condition tests true in both PEx or PEy.

Type 13 Opcode (relative addressing)

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			01110							TERM													

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

Bits	Description
RELADDR	Specifies the end-of-loop address relative to the Do Loop instruction address. The Assembler accepts an absolute address as well and converts the absolute address to the equivalent relative address for coding.
TERM	Specifies the termination condition.

5 IMMEDIATE MOVE

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

Group III Instructions

Group III instructions include the following.

- “Type 14: Ureg«···»DM | PM (direct addressing)” on page 5-2

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

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

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

- “Type 16: Immediate data···»DM | PM” on page 5-9

  Immediate data write to data or program memory

- “Type 17: Immediate data···»Ureg” on page 5-12

  Immediate data write to universal register

Type 14: Ureg«··»DM | PM (direct addressing)

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

Syntax

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

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

Function (SISD)

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

Function (SIMD)

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

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

For the universal register, the X element uses the specified Ureg, and the Y element uses the complementary register (Cureg) that corresponds to the Ureg register specified in the instruction. For a list of complementary registers, see Table 2-10 on page 2-28. Note that only the Cureg subset registers which have complimentary registers are effected by SIMD mode.

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

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

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

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

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

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

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

Do not use the pseudo code above as instruction syntax.

Examples

```
DM(temp)=MODE1;   {temp is a program label}
WAIT=PM(0x489060);
```

When the ADSP-21160 processor is in SISD, the first instruction performs a direct memory write of the value in the `MODE1` register into data memory with the data memory destination address specified by the program label, `temp`. The second instruction initializes the `WAIT` register with the value found in the specified address in program memory.

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

Type 14 Opcode

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

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

Bits	Description
D	Selects the access Type (read or write)
G	Selects the memory Type (data or program)
L	Forces a long word (LW) access when address is in normal word address range
UREG	Specifies the number of a universal register
ADDR	Contains the immediate address value

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

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

Syntax

```
|DM(<data32>, Ia)          |= ureg                 (LW);
|PM(<data32>, Ic)          |

 ureg =                    |DM(<data32>, Ia)      |(LW);
                           |PM(<data32>, Ic)      |
```

Function (SISD)

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

Function (SIMD)

In SIMD mode, the Type 15 instruction provides the same access between data or program memory and a universal register, with indirect addressing using I registers, as is available in SISD mode, except that addressing differs slightly, and the transfer occurs in parallel for the X and Y processing elements.

The X processing element uses the specified I register—pre-modified with an immediate value—to address memory. The Y processing element adds one to the pre-modified I value to address memory. The I register is not updated.

The Ureg specified in the instruction is used for the X processing element transfer and may not be from the same DAG (that is, DAG1 or DAG2) as `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 2-10 on page 2-28. Note that only the Cureg subset registers which have complimentary registers are effected by SIMD mode. For more information on register restrictions, see the "Data Address Generators" chapter of the *ADSP-21160 SHARC DSP Hardware Reference*.

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

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

```
|DM(<data32>, Ia)    |=  ureg                        (LW);
|PM(<data32>, Ic)    |

ureg  =              |DM(<data32>, Ia)       |  (LW);
                     |PM(<data32>, Ic)       |
```

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

```
|DM(<data32>+1, Ia)  |=  cureg                       (LW);
|PM(<data32>+1, Ic)  |

cureg  =             |DM(<data32>+1, Ia)     |  (LW);
                     |PM(<data32>+1, Ic)     |
```

Do not use the pseudo code above as instruction syntax.

Examples

```
DM(24,I5)=TCOUNT;
USTAT1=PM(offs,I13);   {"offs" is a user-defined constant}
```

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

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

The second instruction operates differently in SIMD. The `USTAT1` (system) register is included in the Cureg subset. Therefore, a program memory read—using indirect addressing and the system register, `USTAT1` and its complimentary register `USTAT2`—is performed in parallel on 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 1 to address memory on PEy. The `I13` register is not updated after the memory access occurs in SIMD mode.

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

Type 15 Opcode

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

Bits	Description
D	Selects the access Type (read or write)
G	Selects the memory Type (data or program)
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

Type 16: Immediate data ··»DM | PM

Immediate data write to data or program memory

Syntax

```
|DM(Ia, Mb)        |= <data32> ;
|PM(Ic, Md)        |
```

Function (SISD)

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

Function (SIMD)

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

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

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

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+1, 0)    |= <data32> ;
|PM(Ic+1, 0)    |
```

Do not use the pseudo code above as instruction syntax.

Examples

```
DM(I4,M0)=19304;
PM(I14,M11)=count;   {count is user-defined constant}
```

When the ADSP-21160 processor is in SISD, the two immediate memory writes are performed on PEx. The first instruction writes to data memory and the second instruction writes to program memory. DAG1 and DAG2 are used to indirectly address the locations in memory to which values are written. The `I4` and `I14` registers are post-modified and updated by `M0` and `M11` respectively.

When the ADSP-21160 processor is in SIMD, the two immediate memory writes are performed in parallel on PEx and PEy. The first instruction writes to data memory and the second instruction writes to program memory. DAG1 and DAG2 are used to indirectly address the locations in memory to which values are written. The `I4` and `I14` registers are post-modified and updated by `M0` and `M11` respectively.

Type 16 Opcode

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

Bits	Description
I	Selects the I register
M	Selects the M register
G	Selects the memory (data or program)
DATA	Specifies the 32-bit immediate data

Type 17: Immediate data··»Ureg

Immediate data write to universal register

Syntax

```
ureg = <data32> ;
```

Function (SISD)

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

Function (SIMD)

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

The X element uses the specified Ureg, and the Y element uses the complementary Cureg. Note that only the Cureg subset registers which have complimentary registers are effected by SIMD mode. For a list of complementary registers, see Table 2-10 on page 2-28.

The following pseudo 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> ;
```

Do not use the pseudo code above as instruction syntax.

Examples

```
ASTATx=0x0;
M15=mod1;   {mod1 is user-defined constant}
```

When the ADSP-21160 processor is in SISD, 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.

Type 17 Opcode

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

Bits	Description
UREG	Specifies the number of a universal register.
DATA	Specifies the immediate modify value for the I register.

6 MISCELLANEOUS OPERATIONS

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

Group IV Instructions

Group IV instructions include the following.

- "Type 18: System Register Bit Manipulation" on page 6-2

  System register bit manipulation

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

  Immediate I register modify, with or without bit-reverse

- "Type 20: Push, Pop Stacks, Flush Cache" on page 6-8

  Push or Pop of loop and/or status stacks

- "Type 21: Nop" on page 6-10

  No Operation (NOP)

- "Type 22: Idle" on page 6-11

  Idle

- "Type 25: Cjump/Rframe" on page 6-12

  CJUMP/RFRAME (Compiler-generated instruction)

Type 18: System Register Bit Manipulation

System register bit manipulation

Syntax

BIT	SET CLR TGL TST XOR	sreg <data32> ;

Function (SISD)

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

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

For more information on shifter operations, see "Computations Reference" on page 7-1. For more information on system registers, see the "Registers" appendix of the *ADSP-21160 SHARC DSP Hardware Reference*.

Function (SIMD)

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

The X element operation uses the specified Sreg, and the Y element operations uses the complementary Csreg. For a list of complementary registers, see Table 2-10 on page 2-28.

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

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

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

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

```
BIT    | SET  | csreg  <data32> ;
       | CLR  |
       | TGL  |
       | TST  |
       | XOR  |
```

Do not use the pseudo code above as instruction syntax.

Examples

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

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

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

Type 18 Opcode

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			10100					BOP				SREG				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)																							

Bits	Description
BOP	Selects one of the five bit operations.
SREG	Specifies the system register.
DATA	Specifies the data value.

Type 19: I Register Modify | Bit-Reverse

Immediate I register modify, with or without bit-reverse

Syntax

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

Function (SISD & SIMD)

In SISD and SIMD modes, the Type 19 instruction modifies and updates the specified I register by an immediate 32-bit data value. If the address is to be bit-reversed, you must specify a DAG1 Ia register (`I0`–`I7`) or DAG2 Ic register (`I8`–`I15`), and the modified value is bit-reversed before being written back to the I register. No address is output in either case. For more information on register restrictions, see the "Data Address Generators" chapter of the *ADSP-21160 SHARC DSP Hardware Reference*.

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

Examples

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

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

Type 19 Opcode (without bit-reverse)

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			10110					0	G				I			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)																							

Type 19 Opcode (with bit-reverse)

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			10110					1	G				I			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)																							

Bits	Description

G	Selects the data address generator:
	G=0 for DAG1
	G=1 for DAG2

I	Selects the I register:
	I=0–7 for I0–I7 (for DAG1)
	I=0–7 for I8–I15 (for DAG2)

DATA	Specifies the immediate modifier.

Type 20: Push, Pop Stacks, Flush Cache

Push or Pop of loop and/or status stacks

Syntax

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

Function (SISD and SIMD)

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

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

Examples

```
PUSH LOOP, PUSH STS;
POP 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.

Type 20 Opcode

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
000			10111					LPU	LPO	SPU	SPO	PPU	PPO	FC									

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

Bits	Description
LPU	Pushes the loop stacks
LPO	Pops the loop stacks
SPU	Pushes the status stack
SPO	Pops the status stack
PPU	Pushes the PC stack
PPO	Pops the PC stack
FC	Causes a cache flush

Type 21: Nop

No Operation (NOP)

Syntax

```
NOP ;
```

Function (SISD and SIMD)

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

Type 21 Opcode

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

Type 22: Idle

Idle

Syntax

```
IDLE ;
```

Function (SISD and SIMD)

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

Type 22 Opcode

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

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

Type 25: Cjump/Rframe

Cjump/Rframe (Compiler-generated instruction)

Syntax

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

RFRAME  ;
```

Function (SISD and SIMD)

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

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

The different forms of this instruction perform the operations listed in Table 6-1.

Table 6-1. Operations Done by Forms of the Type 25 Instruction

Compiler-Generated Instruction[1]	**Operations Performed in SISD Mode**	**Operations Performed in SIMD Mode**
CJUMP label (DB);	JUMP label (DB), R2=I6, I6=I7;	JUMP label (DB), R2=I6, S2=I6, I6=I7;
CJUMP (PC,raddr)(DB);	JUMP (PC,raddr) (DB), R2=I6, I6=I7;	JUMP (PC,raddr) (DB), R2=I6, S2=I6, I6=I7;
RFRAME;	I7=I6, I6=DM(0,I6);	I7=I6, I6=DM(0,I6), I6=DM(1,I6);

1 In this table, raddr indicates a relative 24-bit address.

Type 25a Opcode (with direct branch)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
0001				1000				0000				0100				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
ADDR																							

Type 25b Opcode (with PC-relative branch)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
0001				1000				0100				0100				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
RELADDR																							

Bits	Description
ADDR	Specifies a 24-bit program memory address for “function”
RELADDR	Specifies a 24-bit, twos-complement value added to the current PC value to generate the branch address

Type 25: Cjump/Rframe

Type 25c Opcode (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			

7 COMPUTATIONS REFERENCE

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

Compute Field

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

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

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

Compute Field

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

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

Bits	Description
CU	Specifies the computation unit for the compute operation, where: 00=ALU, 01=Multiplier, and 10=Shifter
Opcode	Specifies the compute operation
Rn	Specifies register for the compute result
Rx	Specifies register for the compute's x operand
Ry	Specifies register for the compute's y operand

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 7-1 on page 7-4, Table 7-2 on page 7-5, Table 7-3 on page 7-53, Table 7-4 on page 7-54, and Table 7-8 on page 7-65. These tables are organized by computation unit: "ALU Operations" on page 7-3, "Multiplier Operations" on page 7-51, and "Shifter Operations" on page 7-64. Following each table, each compute operation is described in detail.

ALU Operations

This section describes the ALU operations. Table 7-1 and Table 7-2 on page 7-5 summarize the syntax and opcodes for the fixed-point and floating-point ALU operations, respectively.

Fixed-Point ALU Operations

Table 7-1. Fixed-Point ALU Operations

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

ALU Floating-Point Operations

Table 7-2. Floating-Point ALU Operations

Syntax	Opcode	Reference page
Fn = Fx + Fy	1000 0001	on page 7-28
Fn = Fx – Fy	1000 0010	on page 7-29
Fn = ABS (Fx + Fy)	1001 0001	on page 7-30
Fn = ABS (Fx – Fy)	1001 0010	on page 7-31
Fn = (Fx + Fy)/2	1000 1001	on page 7-32
Fn = COMP(Fx, Fy)	1000 1010	on page 7-33
Fn = –Fx	1010 0010	on page 7-34
Fn = ABS Fx	1011 0000	on page 7-35
Fn = PASS Fx	1010 0001	on page 7-36
Fn = RND Fx	1010 0101	on page 7-37
Fn = SCALB Fx BY Ry	1011 1101	on page 7-38
Rn = MANT Fx	1010 1101	on page 7-39
Rn = LOGB Fx	1100 0001	on page 7-40
Rn = FIX Fx BY Ry	1101 1001	on page 7-41
Rn = FIX Fx	1100 1001	on page 7-41
Rn = TRUNC Fx BY Ry	1101 1101	on page 7-41
Rn = TRUNC Fx	1100 1101	on page 7-41
Fn = FLOAT Rx BY Ry	1101 1010	on page 7-43
Fn = FLOAT Rx	1100 1010	on page 7-43
Fn = RECIPS Fx	1100 0100	on page 7-44
Fn = RSQRTS Fx	1100 0101	on page 7-46
Fn = Fx COPYSIGN Fy	1110 0000	on page 7-48
Fn = MIN(Fx, Fy)	1110 0001	on page 7-49

Table 7-2. Floating-Point ALU Operations (Cont'd)

Syntax	Opcode	Reference page
Fn = MAX(Fx, Fy)	1110 0010	on page 7-50
Fn = CLIP Fx BY Fy	1110 0011	on page 7-51

Rn = Rx + Ry

Function

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

Status Flags

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

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

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

Rn = Rx + Ry + CI

Function

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

Status Flags

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

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

Rn = (Rx + Ry)/2

Function

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

Status Flags

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

COMP(Rx, Ry)

Function

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

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

Status Flags

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

COMPU(Rx, Ry)

Function

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

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

Status Flags

AZ	Is set if the operands in registers Rx and Ry are equal, otherwise cleared
AU	Is cleared
AN	Is set if the operand in the Rx register is smaller than the operand in the Ry register, otherwise cleared
AV	Is cleared
AC	Is cleared
AS	Is cleared
AI	Is cleared

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

Rn = Rx + CI – 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

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

Rn = Rx – 1

Function

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

Status Flags

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

Rn = –Rx

Function

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

Status Flags

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

Rn = ABS Rx

Function

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

Status Flags

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

Rn = PASS Rx

Function

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

Status Flags

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

Rn = Rx AND Ry

Function

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

Status Flags

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

Rn = Rx OR Ry

Function

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

Status Flags

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

Rn = Rx XOR Ry

Function

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

Status Flags

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

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

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

Rn = MAX(Rx, Ry)

Function

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

Status Flags

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

Rn = CLIP Rx BY Ry

Function

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

Status Flags

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

Fn = Fx + Fy

Function

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

Status Flags

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

Fn = Fx – Fy

Function

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

Status Flags

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

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

Fn = ABS (Fx – Fy)

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

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 $\overline{AZ}$ and $\overline{AN}$); 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

Fn = –Fx

Function

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

Status Flags

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

Fn = 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

AZ	Set if the result operand is +Zero, otherwise cleared
AU	Cleared
AN	Cleared
AV	Cleared
AC	Cleared
AS	Set if the input operand is negative, otherwise cleared
AI	Set if the input operand is a NAN, otherwise cleared

Fn = PASS Fx

Function

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

Status Flags

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

Fn = RND Fx

Function

Rounds the floating-point operand in register Fx to a 32 bit boundary. Rounding is to nearest (IEEE) or by truncation, as defined by the rounding mode bit in `MODE1`. Post-rounded overflow returns ±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

Fn = SCALB Fx BY Ry

Function

Scales the exponent of the floating-point operand in Fx by adding to it the fixed-point twos-complement integer in Ry. The scaled floating-point result is placed in register Fn. Overflow returns ±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

Rn = MANT Fx

Function

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

Status Flags

AZ	Set if the result is zero, otherwise cleared
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

Rn = LOGB Fx

Function

Converts the exponent of the floating-point operand in register Fx to an unbiased twos-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

Rn = FIX Fx
Rn = TRUNC Fx
Rn = FIX Fx BY Ry
Rn = TRUNC Fx BY Ry

Function

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

If the `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 twos-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 `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 `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.

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

Status Flags

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

Fn = FLOAT Rx BY Ry
Fn = FLOAT Rx

Function

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

Status Flags

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

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 twos complement of the unbiased Fx exponent, decremented by one; that is, if e is the unbiased exponent of Fx, then the unbiased exponent of Fn = –e – 1. The sign of the seed is the sign of the input. A ±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.[1] The result is accurate to one LSB in whichever format mode, 32-bit or 40-bit, is set (32-bit only for ADSP-21010). The following inputs are required: F0=numerator, F12=denominator, F11=2.0. The quotient is returned in F0. (The two highlighted instructions can be removed if only a ±1 LSB accurate single-precision result is necessary.)

```
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.

[1] Cavanagh, J. 1984. Digital Computer Arithmetic. McGraw-Hill. Page 284.

Status Flags

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

Fn = RSQRTS Fx

Function

Creates a 4-bit accurate seed for $1/(Fx)^{½}$, the reciprocal square root of Fx.

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

The unbiased exponent of the seed is calculated as the twos complement of the unbiased Fx exponent, shifted right by one bit and decremented by one; that is, if e is the unbiased exponent of Fx, then the unbiased exponent of Fn = –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 ($1/(x)^{½}$) using a Newton-Raphson iteration algorithm.[1] The result is accurate to one LSB in whichever format mode, 32-bit or 40-bit, is set (32-bit only for ADSP-21010).

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

```
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}
```

[1] Cavanagh, J. 1984. Digital Computer Arithmetic. McGraw-Hill. Page 278.

```
F12=F12*F0;   {F12=C*X1^2}
F4=F1*F4, F12=F8-F12;   {F4=.5*X1, F12=3-C*X1^2}
F4=F4*F12;   {F4=X2=.5*X1(3-C*X1^2)}
F12=F4*F4;   {F12=X2^2}
F12=F12*F0;   {F12=C*X2^2}
F4=F1*F4, F12=F8-F12;   {F4=.5*X2, F12=3-C*X2^2}
F4=F4*F12;   {F4=X3=.5*X2(3-C*X2^2)}
```

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

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

Fn = MAX(Fx, Fy)

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

Fn = CLIP Fx BY Fy

Function

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

Status Flags

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

Multiplier Operations

This section describes the multiplier operations. Table 7-3 and Table 7-4 on page 7-54 summarize the syntax and opcodes for the fixed-point and floating-point multiplier operations, respectively. These tables use the following symbols to indicate location of operands and other features:

- y = y-input (1 = signed, 0 = unsigned)
- x = x-input (1 = signed, 0 = unsigned)

- f = format (1 = fractional, 0 = integer)
- r = rounding (1 = yes, 0 = no)

Multiplier Fixed-Point Operations

Table 7-3. Fixed-Point Multiplier Operations

Syntax	Opcode	Reference Page
Rn = Rx*Ry *mod2*	01yx f00r	on page 7-56
MRF = Rx*Ry *mod2*	01yx f10r	on page 7-56
MRB = Rx*Ry *mod2*	01yx f11r	on page 7-56
Rn = MRF +Rx*Ry *mod2*	10yx f00r	on page 7-57
Rn = MRB +Rx*Ry *mod2*	10yx f01r	on page 7-57
MRF = MRF +Rx*Ry *mod2*	10yx f10r	on page 7-57
MRB = MRB +Rx*Ry *mod2*	10yx f11r	on page 7-57
Rn = MRF –Rx*Ry *mod2*	11yx f00r	on page 7-58
Rn = MRB –Rx*Ry *mod2*	11yx f01r	on page 7-58
MRF = MRF –Rx*Ry *mod2*	11yx f10r	on page 7-58
MRB = MRB –Rx*Ry *mod2*	11yx f11r	on page 7-58
Rn = SAT MRF *mod1*	0000 f00x	on page 7-59
Rn = SAT MRB *mod1*	0000 f01x	on page 7-59
MRF = SAT MRF *mod1*	0000 f10x	on page 7-59
MRB = SAT MRB *mod1*	0000 f11x	on page 7-59
Rn =RND MRF *mod1*	0001 100x	on page 7-60
Rn = RND MRB *mod1*	0001 101x	on page 7-60
MRF = RND MRF *mod1*	0001 110x	on page 7-60
MRB = RND MRB *mod1*	0001 111x	on page 7-60
MRF = 0	0001 0100	on page 7-61
MRB = 0	0001 0110r	on page 7-61
MR = Rn		on page 7-62
Rn = MR		on page 7-62

Multiplier Floating-Point Operations

Table 7-4. Floating-Point Multiplier Operations

Syntax	Opcode	Reference Page
Fn = Fx*Fy	0011 0000	on page 7-64

Mod1 and Mod2 Modifiers

Mod2 in Table 7-3 on page 7-53 is an optional modifier. It is enclosed in parentheses and consists of three or four letters that indicate whether:

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

Table 7-5 lists the options for mod2 and the corresponding opcode values.

Table 7-5. Mod2 Options and Opcodes

Option	Opcode
(SSI)	_ _11 0_ _0
(SUI)	_ _01 0_ _0
(USI)	_ _10 0_ _0
(UUI)	_ _00 0_ _0
(SSF)	_ _11 1_ _0
(SUF)	_ _01 1_ _0
(USF)	_ _10 1_ _0
(UUF)	_ _00 1_ _0

Table 7-5. Mod2 Options and Opcodes (Cont'd)

Option	Opcode
(SSFR)	_ _11 1_ _1
(SUFR)	_ _01 1_ _1
(USFR)	_ _10 1_ _1
(UUFR)	_ _00 1_ _1

Similarly, mod1 in Table 7-3 on page 7-53 is an optional modifier, enclosed in parentheses, consisting of two letters that indicate whether the input is signed (S) or unsigned (U) and whether the input is in integer (I) or fractional (F) format. The options for mod1 and the corresponding opcode values are listed in Table 7-6.

Table 7-6. Mod1 Options and Opcodes

Option	Opcode
(SI) (for SAT only)	_ _ _ _ 0 _ _ 1
(UI) (for SAT only)	_ _ _ _ 0 _ _ 0
(SF)	_ _ _ _ 1 _ _ 1
(UF)	_ _ _ _ 1 _ _ 0

Rn = Rx * Ry mod2
MRF = Rx * Ry mod2
MRB Rx * Ry mod2

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 MR accumulation registers.

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

Status Flags

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

Rn = MRF + Rx * Ry mod2
Rn = MRB + Rx * Ry mod2
MRF = MRF + Rx * Ry mod2
MRB = MRB + Rx * Ry mod2

Function

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

Status Flags

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

Rn = MRF – Rx * Ry mod2
Rn = MRB – Rx * Ry mod2
MRF = MRF – Rx * Ry mod2
MRB = MRB – Rx * Ry mod2

Function

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

Rn = SAT MRF mod1
Rn = SAT MRB mod1
MRF = SAT MRF mod1
MRB = SAT MRB mod1

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

MN	Set if the result is negative, otherwise cleared
MV	Cleared
MU	Set if the upper 48 bits of a fractional result are all zeros (signed or unsigned result) or ones (signed result) and the lower 32 bits are not all zeros; Integer results do not underflow
MI	Cleared

Rn = RND MRF mod1
Rn = RND MRB mod1
MRF = RND MRF mod1
MRB = RND MRB mod1

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

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

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	Cleared
MV	Cleared
MU	Cleared
MI	Cleared

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

```
MR0F = Rn    Rn = MR0F
MR1F = Rn    Rn = MR1F
MR2F = Rn    Rn = MR2F
MR0B = Rn    Rn = MR0B
MR1B = Rn    Rn = MR1B
MR2B = Rn    Rn = MR2B
```

Compute Field

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
100000						T	Ai				Rk											

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

Table 7-7. Ai Values and MR Registers

Ai	MR Register
0000	MR0F
0001	MR1F

Table 7-7. Ai Values and MR Registers (Cont'd)

Ai	MR Register
0010	MR2F
0100	MR0B
0101	MR1B
0110	MR2B

Status Flags

MN	Cleared
MV	Cleared
MU	Cleared
MI	Cleared

Fn = Fx * Fy

Function

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

Status Flags

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

Shifter Operations

Shifter operations are described in this section. Table 7-8 lists the syntax and opcodes for the shifter operations. The succeeding pages provide detailed descriptions of each operation. Some of the instructions in Table 7-8 accept the following modifiers.

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

Shifter Opcodes

The shifter operates on the register file's 32-bit fixed-point fields (bits 38-9). Two-input shifter operations can take their y input from the register file or from immediate data provided in the instruction. Either form uses the same opcode. However, the latter case, called an immediate shift or shifter immediate operation, is allowed only with instruction type

6, which has an immediate data field in its opcode for this purpose. All other instruction types must obtain the y input from the register file when the compute operation is a two-input shifter operation.

Table 7-8. Shifter Operations

Syntax	Opcode	Reference Page
Rn = LSHIFT Rx BY Ry|<data8>	0000 0000	on page 7-66
Rn = Rn OR LSHIFT Rx BY Ry|<data8>	0010 0000	on page 7-67
Rn = ASHIFT Rx BY Ry|<data8>	0000 0100	on page 7-68
Rn = Rn OR ASHIFT Rx BY Ry|<data8>	0010 0100	on page 7-69
Rn = ROT Rx BY Ry|<data8>	0000 1000	on page 7-70
Rn = BCLR Rx BY Ry|<data8>	1100 0100	on page 7-71
Rn =BSET Rx BY Ry|<data8>	1100 0000	on page 7-72
Rn = BTGL Rx BY Ry|<data8>	1100 1000	on page 7-73
BTST Rx BY Ry|<data8>	1100 1100	on page 7-74
Rn = FDEP Rx BY Ry|<bit6>:<len6>	0100 0100	on page 7-75
Rn = Rn OR FDEP Rx BY Ry|<bit6>:<len6>	0110 0100	on page 7-77
Rn = FDEP Rx BY Ry|<bit6>:<len6> (SE)	0100 1100	on page 7-79
Rn = Rn OR FDEP Rx BY Ry|<bit6>:<len6>(SE)	0110 1100	on page 7-81
Rn = FEXT RX BY Ry|<bit6>:<len6>	0100 0000	on page 7-83
Rn = FEXT Rx BY Ry|<bit6>:<len6> (SE)	0100 1000	on page 7-85
Rn = EXP Rx	1000 0000	on page 7-87
Rn = EXP Rx (EX)	1000 0100	on page 7-88
Rn = LEFTZ Rx	1000 1000	on page 7-89
Rn = LEFTO Rx	1000 1100	on page 7-90
Rn = FPACK Fx	1001 0000	on page 7-91
Fn = FUNPACK Rx	1001 0100	on page 7-92

Rn = LSHIFT Rx BY Ry
Rn = 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 twos-complement numbers. Positive values select a left shift, negative values select a right shift. The 8-bit immediate data can take values between –128 and 127 inclusive, allowing for a shift of a 32-bit field from off-scale right to off-scale left.

Status Flags

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

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

Function

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

Status Flags

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

Rn = ASHIFT Rx BY Ry
Rn = ASHIFT Rx BY <data8>

Function

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

Status Flags

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

Rn = Rn OR ASHIFT Rx BY Ry
Rn = Rn OR ASHIFT Rx BY <data8>

Function

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

Status Flags

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

Rn = ROT Rx BY Ry
Rn = ROT Rx BY <data8>

Function

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

Status Flags

SZ	Set if the rotated result is zero, otherwise cleared
SV	Cleared
SS	Cleared

Rn = BCLR Rx BY Ry
Rn = BCLR Rx BY <data8>

Function

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

Status Flags

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

ⓘ This compute operation affects a bit in a register file location. There is also a bit manipulation instruction that affects one or more bits in a system register. This Bit Clr instruction should not be confused with the Bclr shifter operation. For more information on Bit Clr, see "Type 18: System Register Bit Manipulation" on page 6-2.

Rn = BSET Rx BY Ry
Rn = BSET Rx BY <data8>

Function

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

Status Flags

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 18: System Register Bit Manipulation" on page 6-2.

Rn = BTGL Rx BY Ry
Rn = BTGL Rx BY <data8>

Function

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

Status Flags

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

This compute operation affects a bit in a register file location. There is also a bit manipulation instruction that affects one or more bits in a system register. This Bit Tgl instruction should not be confused with the Btgl shifter operation. For more information on Bit Tgl, see "Type 18: System Register Bit Manipulation" on page 6-2.

BTST Rx BY Ry
BTST Rx BY <data8>

Function

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

Status Flags

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

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

For more information on Bit Tst, see "Type 18: System Register Bit Manipulation" on page 6-2.

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

Function

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

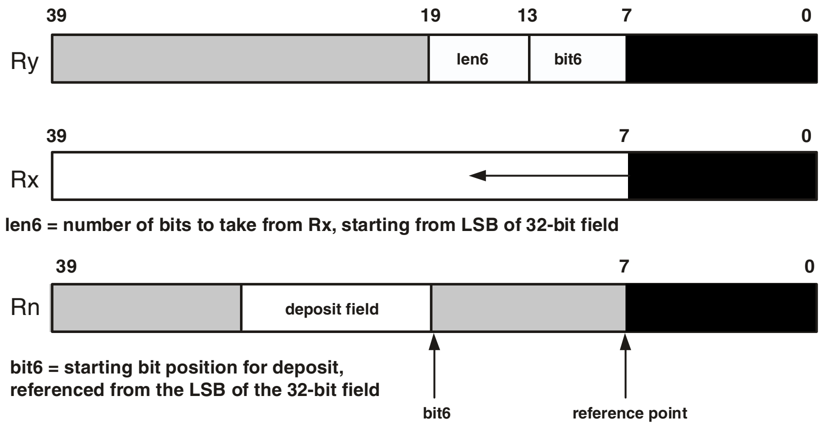
39
19
13
7
0
Ry
len6
bit6
39
7
0
Rx
len6 = number of bits to take from Rx, starting from LSB of 32-bit field
39
7
0
Rn
deposit field
bit6 = starting bit position for deposit, referenced from the LSB of the 32-bit field
bit6
reference point

Figure 7-1. Field Alignment

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

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

Function

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

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

Example

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

39       31       23       15       7        0
|abcdefgh|ijklmnop|qrstuvwx|yzabcdef|ghijklmn|      Rn old
      \--------------/
                     |
                     bit position bit6 (from reference point)

39       31       23       15       7        0
|abcdeopq|rstuvwxy|zabtuvwx|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

Rn = FDEP Rx BY Ry (SE)
Rn = FDEP Rx BY <bit6>:<len6> (SE)

Function

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

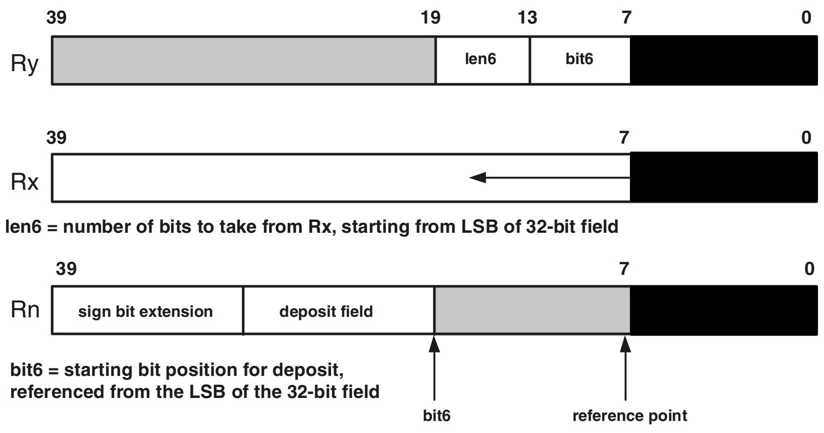
39
19
13
7
0
Ry
len6
bit6
39
7
0
Rx
len6 = number of bits to take from Rx, starting from LSB of 32-bit field
39
7
0
Rn
sign bit extension
deposit field
bit6 = starting bit position for deposit, referenced from the LSB of the 32-bit field
bit6
reference point

Figure 7-2. Field Alignment

Example

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

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

Status Flags

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

Rn = Rn OR FDEP Rx BY Ry (SE)
Rn = Rn OR FDEP Rx BY <bit6>:<len6> (SE)

Function

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

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

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 bit6
                     (from reference point)

39       31       23       15       7        0
|abcdefgh|ijklmnop|qrstuvwx|yzabcdef|ghijklmn|     Rn old
```

```
39       31       23       15       7        0
|vwxyzabc|defghijk|lmntuvwx|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

Rn = FEXT Rx BY Ry
Rn = FEXT Rx BY <bit6>:<len6>

Function

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

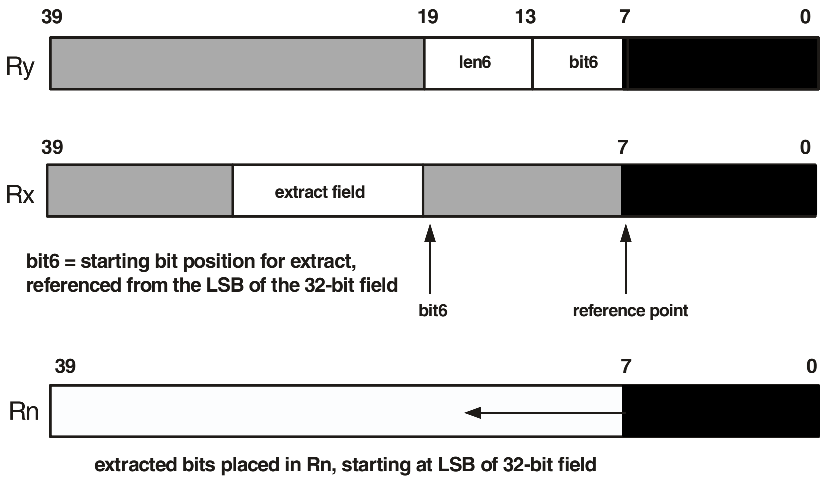
39
19
13
7
0
Ry
len6
bit6
39
7
0
Rx
extract field
bit6 = starting bit position for extract, referenced from the LSB of the 32-bit field
bit6
reference point
39
7
0
Rn
extracted bits placed in Rn, starting at LSB of 32-bit field

Figure 7-3. Field Alignment

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

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

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

Function

Extracts and sign-extends a field from register Rx to register Rn. The output field is placed right-aligned in the fixed-point field of Rn. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction. The field is extracted from the fixed-point field of Rx starting from a bit position determined by the bit6 field in register Ry or by the immediate bit6 field in the instruction. The MSBs of Rn are sign-extended by the MSB of the extracted field, unless the MSB is extracted from off-scale left.

The floating-point extension field of Rn (bits 7–0 of the 40-bit word) is set to all 0s. Bit6 and len6 can take values between 0 and 63 inclusive, allowing for extraction of fields ranging in length from 0 to 32 bits and from bit positions ranging from 0 to off-scale left.

Example

```
39       31       23       15       7        0
|-----abc|defghijk|lmn-----|--------|--------|          Rx
      \--------------/
         len6 bits   |
                     bit position bit6
                     (from reference point)

39       31       23       15       7        0
|aaaaaaaa|aaaaaaaa|aaabcdef|ghijklmn|00000000|          Rn
\-------------------/
    sign extension
```

Status Flags

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

Rn = EXP Rx

Function

Extracts the exponent of the fixed-point operand in Rx. The exponent is placed in the shf8 field in register Rn. The exponent is calculated as the twos-complement of:

```
# leading sign bits in Rx - 1
```

Status Flags

SZ	Set if the extracted exponent is 0, otherwise cleared
SV	Cleared
SS	Set if the fixed-point operand in Rx is negative (bit 31 is a 1), otherwise cleared

Rn = 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 twos-complement of:

```
# leading sign bits in Rx - 1
```

Status Flags

SZ	Set if the extracted exponent is 0, otherwise cleared
SV	Cleared
SS	Set if the 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

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

Rn = LEFTO Rx

Function

Extracts the number of leading 1s from the fixed-point operand in Rx. The extracted number is placed in the bit6 field in Rn.

Status Flags

SZ	Set if the MSB of Rx is 0, otherwise cleared
SV	Set if the result is 32, otherwise cleared
SS	Cleared

Rn = FPACK Fx

Function

Converts the IEEE 32-bit floating-point value in Fx to a 16-bit floating-point value stored in Rn. The short float data format has an 11-bit mantissa with a four-bit exponent plus sign bit. The 16-bit floating-point numbers reside in the lower 16 bits of the 32-bit floating-point field.

The result of the FPACK operation is:

$135 < exp$[1]	Largest magnitude representation
$120 < exp \le 135$	Exponent is MSB of source exponent concatenated with the three LSBs of source exponent. The packed fraction is the rounded upper 11 bits of the source fraction.
$109 < exp \le 120$	Exponent=0. Packed fraction is the upper bits (source exponent – 110) of the source fraction prefixed by zeros and the "hidden" 1. The packed fraction is rounded.
$exp < 110$	Packed word is all zeros.

1 exp = source exponent sign bit remains the same in all cases

The short float type supports gradual underflow. This method sacrifices precision for dynamic range. When packing a number which would have underflowed, the exponent is set to zero and the mantissa (including "hidden" 1) is right-shifted the appropriate amount. The packed result is a denormal which can be unpacked into a normal IEEE floating-point number.

Status Flags

SZ	Cleared
SV	Set if overflow occurs, cleared otherwise
SS	Cleared

Fn = FUNPACK Rx

Function

Converts the 16-bit floating-point value in Rx to an IEEE 32-bit floating-point value stored in Fx.

Result

$0 < \exp^1 \leq 15$	Exponent is the three LSBs of the source exponent prefixed by the MSB of the source exponent and four copies of the complement of the MSB. The unpacked fraction is the source fraction with 12 zeros appended.
exp = 0	Exponent is (120 – N) where N is the number of leading zeros in the source fraction. The unpacked fraction is the remainder of the source fraction with zeros appended to pad it and the “hidden” 1 stripped away.

1 exp = source exponent sign bit remains the same in all cases

The short float type supports gradual underflow. This method sacrifices precision for dynamic range. When packing a number that would have underflowed, the exponent is set to 0 and the mantissa (including “hidden” 1) is right-shifted the appropriate amount. The packed result is a denormal, which can be unpacked into a normal IEEE floating-point number.

Status Flags

SZ	Cleared
SV	Cleared
SS	Cleared

Multifunction Computations

Multifunction computations are operations that occur simultaneously within the DSP's computational unit. The syntax for these operations consists of combinations of instructions, delimited with commas and ended with a semicolon. The three types of multifunction computations appear below. Each type has a different format for the compute field.

- "Parallel Add and Subtract" on page 7-95
- "Parallel Multiplier and ALU" on page 7-98
- "Parallel Multiplier With Add and Subtract" on page 7-101

Operand Constraints

Each of the four input operands for multifunction computations are constrained to a different set of four register file locations, as shown in Figure 7-4. For example, the x-input to the ALU must be `R8`, `R9`, `R10`, or `R11`. In all other compute operations, the input operands can be any register file location.

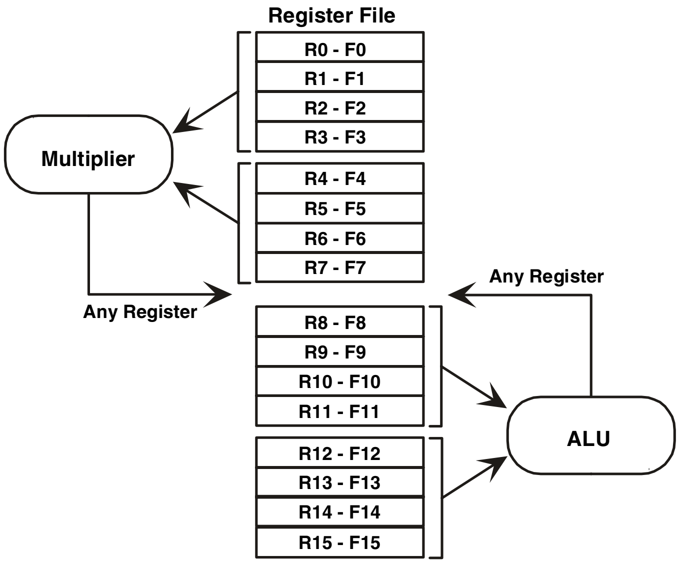

Figure 7-4. Permitted Input Registers for Multifunction Computations

Parallel Add and Subtract

Function (Fixed-Point)

Completes a dual add/subtract of the fixed-point fields in registers Rx and Ry. The sum is placed in the fixed-point field of register Ra and the difference in the fixed-point field of Rs. The floating-point extension fields of Ra and Rs are set to all 0s. In saturation mode (the ALU saturation mode bit in `MODE1` set) positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

Function (Floating-Point)

Completes a dual add/subtract of the floating-point operands in registers Fx and Fy. The normalized results are placed in registers Fa and Fs: the sum in Fa and the difference in Fs. Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in `MODE1`. Post-rounded overflow returns ±Infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). Post-rounded denormal returns ±Zero. Denormal inputs are flushed to ±Zero. A NAN input returns an all 1s result.

Syntax

Table 7-9 shows the fixed-point and floating-point syntax for multifunction add and subtract instructions.

Table 7-9. Multifunction, Parallel Add and Subtract

Syntax	Opcode (bits 19–16)
Ra = Rx + Ry, Rs = Rx – Ry	0111
Fa = Fx + Fy, Fs = Fx – Fy	1111

Compute Field (Fixed-Point)

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	00		0111				Rs				Ra				Rx				Ry			

AZ	Set if an output is 0s, otherwise cleared
AU	Cleared
AN	Set if the most significant output bit is 1 for either of the outputs, otherwise cleared
AV	Set if the XOR of the carries of the two most significant adder stages of either of the outputs is 1, otherwise cleared
AC	Set if the carry from the most significant adder stage for either of the outputs is 1, otherwise cleared
AS	Cleared
AI	Cleared

Compute Field (Fixed-Point)

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	00		1111				Fs				Fa				Fx				Fy			

AZ	Set if either of the post-rounded results is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU	Set if either post-rounded result is a denormal, otherwise cleared
AN	Set if either of the floating-point results is negative, otherwise cleared
AV	Set if a post-rounded result overflows (unbiased exponent > +127), otherwise cleared

AC	Cleared
AS	Cleared
AI	Set if an input is a NAN or if both inputs are Infinities, otherwise cleared

Parallel Multiplier and ALU

Function

The parallel multiplier/ALU operation performs a multiply or multiply/accumulate and one of the following ALU operations: Add, Subtract, Average, Fixed-point to floating-point conversion or floating-point to fixed-point conversion, and/or Floating-point Abs, Min, or Max.

The multiplier and ALU operations are determined by `OPCODE`. The selections for the 6-bit `OPCODE` field are listed in Table 7-11 on page 7-99. The multiplier x and y operands are received from data registers RXM (FXM) and RYM (FYM). The multiplier result operand is returned to data register RM (FM). The ALU x and y operands are received from data registers RXA (FXA) and RYA (FYA). The ALU result operand is returned to data register RA (FA).

The result operands can be returned to any registers within the register file. Each of the four input operands is restricted to a particular set of four data registers.

Table 7-10. Valid Data Registers for Input Operands

Input	**Valid Sources**
Multiplier X	R3-R0 (F3-F0)
Multiplier Y	R7-R4 (F7-F4)
ALU X	R11-R8 (F11-F8)
ALU Y	R15-R12 (F15-F12)

Syntax

Table 7-11 provides the syntax and opcode for each of the parallel multiplier and ALU instructions for both fixed-point and floating-point versions.

Table 7-11. Multifunction, Multiplier and ALU

Syntax	Opcode (bits 22–16)
Rm = R3-0 * R7-4 (SSFR), Ra = R11-8 + R15-12	1000100
Rm = R3-0 * R7-4 (SSFR), Ra = R11-8 – R15-12	1000101
Rm = R3-0 * R7-4 (SSFR), Ra = (R11-8 + R15-12)/2	1000110
MRF = MRF + R3-0 * R7-4 (SSF), Ra = R11-8 + R15-12	1001000
MRF = MRF + R3-0 * R7-4 (SSF), Ra = R11-8 – R15-12	1001001
MRF = MRF + R3-0 * R7-4 (SSF), Ra = (R11-8 + R15-12)/2	1001010
Rm = MRF + R3-0 * R7-4 (SSFR), Ra = R11-8 + R15-12	1001100
Rm = MRF + R3-0 * R7-4 (SSFR), Ra = R11-8 – R15-12	1001101
Rm = MRF + R3-0 * R7-4 (SSFR), Ra =(R11-8 + R15-12)/2	1001110
MRF = MRF – R3-0 * R7-4 (SSF), Ra = R11-8 + R15-12	1010000
MRF = MRF – R3-0 * R7-4 (SSF), Ra = R11-8 – R15-12	1010001
MRF = MRF – R3-0 * R7-4 (SSF), Ra = (R11-8 + R15-12)/2	1010010
Rm = MRF – R3-0 * R7-4 (SSFR), Ra = R11-8 + R15-12	1010100
Rm = MRF – R3-0 * R7-4 (SSFR), Ra = R11-8 – R15-12	1010101
Rm = MRF – R3-0 * R7-4 (SSFR), Ra =(R11-8 + R15-12)/2	1010110
Fm = F3-0 * F7-4, Fa = F11-8 + F15-12	1011000
Fm = F3-0 * F7-4, Fa = F11-8 – F15-12	1011001
Fm = F3-0 * F7-4, Fa = FLOAT R11-8 by R15-12	1011010
Fm = F3-0 * F7-4, Fa = FIX F11-8 by R15-122	1011011
Fm = F3-0 * F7-4, Fa = ABS F11-8	1011101
Fm = F3-0 * F7-4, Fa = MAX (F11-8, F15-12)	1011110
Fm = F3-0 * F7-4, Fa = MIN (F11-8, F15-12)	1011111

Compute Field (Fixed-Point)

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Opcode (Table 7-11)						Rs				Ra				Rxm		Rym		Rxa		Rya	

Compute Field (Floating-Point)

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Opcode (Table 7-11)						Fs				Fa				Fxm		Fym		Fxa		Fya	

Parallel Multiplier With Add and Subtract

Function

The parallel multiplier and dual add/subtract operation performs a multiply or multiply/accumulate and computes the sum and the difference of the ALU inputs.

The multiplier x and y operands are received from data registers RXM (FXM) and RYM (FYM). The multiplier result operand is returned to data register RM (FM). The ALU x and y operands are received from data registers RXA (FXA) and RYA (FYA). The ALU result operands are returned to data register RA (FA) and RS (FS).

The result operands can be returned to any registers within the register file. Each of the four input operands is restricted to a different set of four data registers.

Table 7-12. Valid Sources of the Input Operands

Input	Valid Sources
Multiplier X	R3-R0 (F3-F0)
Multiplier Y	R7-R4 (f7-f4)
ALU X	R11-R8 (F11-F8)
ALU Y	R15-R12 (F15-F12)

Syntax

Table 7-13 provides the syntax and opcode for each of the parallel multiplier and add/subtract instructions for both fixed-point and floating-point versions.

Table 7-13. Multifunction, Multiplier and Dual Add and Subtract

Syntax	Opcode (bits 22–20)
Rm=R3-0 * R7-4 (SSFR), Ra=R11-8 + R15-12, Rs=R11-8 – R15-12	110
Fm=F3-0 * F7-4, Fa=F11-8 + F15-12, Fs=F11-8 – F15-12	111

Compute Field (Fixed-Point)

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	10		Rs				Rm				Ra				RxmM		Rym		Rxa		Rya	

Compute Field (Floating-Point)

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	11		Fs				Fm				Fa				Fxm		Fym		Fxa		Fya	

I INDEX