SHARC+ Core Programming Reference

(Includes ADSP-SC5xx and ADSP-215xx Processors)

Revision 1.5, June 2023

Part Number
82-100131-01

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

Notices

Copyright Information

Disclaimer

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

Trademark and Service Mark Notice

The Analog Devices logo, Blackfin, Blackfin+, CrossCore, EngineerZone, EZ-Board, EZ-KIT Lite, EZ-KIT Mini, EZ-Extender, SHARC, SHARC+, A^{2}B, SigmaStudio, and VisualDSP++ are registered trademarks of Analog Devices, Inc.

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

Contents

Processing Elements

Program Sequencer

L1 Memory Interface

L1 Cache Controller

Safety, Security, Multi-Core, and Low-Power Features

SHARC+ Core Debug Interface

Instruction Set Reference

Group I Conditional Compute and Move or Modify Instruction

Group II Conditional Program Flow Control Instructions

Group III Immediate Data Move Instructions

Group IV Miscellaneous Instructions

Computation Opcode Reference

ALU Fixed-Point Computations

ALU Floating-Point Computations

Multi-Function Instruction Computations

Immediate (imm) and Constant (const) Opcodes

Register (reg) Opcodes

Numeric Formats

SHARC-PLUS REGF Register Descriptions

SHARC-PLUS SHDBG Register Descriptions

SHARC-PLUS SHL1C Register Descriptions

SHARC-PLUS PFM Register Descriptions

Preface

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

Purpose of This Manual

The ***SHARC+ Processor Programming Reference*** provides architectural and programming information about the SHARC+ cores. The cores implement a single-instruction multiple-data (SIMD) architecture with an 11-stage instruction pipeline. The architectural descriptions cover functional blocks and buses, including features and processes that they support. The manual also provides information on the I/O capabilities (flag pins, JTAG) supported by the core. The programming information covers the instruction set and compute operations.

For information about the peripherals associated with these products, see the product family hardware reference. For timing, electrical, and package specifications, see the processor-specific data sheet.

NOTE: Analog Devices is in the process of updating documentation to provide terminology and language that is culturally appropriate. This is a process with a wide scope and will be phased in as quickly as possible. Thank you for your patience.

Intended Audience

The primary audience for this manual is a programmer who is familiar with Analog Devices processors. The manual assumes the audience has a working knowledge of the appropriate processor architecture and instruction set. Programmers who are unfamiliar with Analog Devices processors can use this manual, but should supplement it with other texts, such as hardware and programming reference manuals that describe their target architecture.

Manual Contents

This manual provides detailed information about the processor family in the following chapters. Please note that there are differences in this section from previous manual revisions.

- Chapter 1, ***Introduction.*** Provides an architectural overview of the SHARC+ core.
- Chapter 2, ***Register Files.*** Describes the core register files including the data exchange register (PX).
- Chapter 3, ***Processing Elements.*** Describes the arithmetic/logic units (ALUs), multiplier/accumulator units, and shifter. The chapter also discusses data formats, data types, and register files.
- Chapter 4, ***Program Sequencer.*** Describes the operation of the program sequencer, which controls program flow by providing the address of the next instruction to be executed. The chapter also discusses loops, subroutines, jumps, interrupts, exceptions, and the `IDLE` instruction.
- Chapter 5, ***Timer.*** Describes the operation of the processor's core timer.

- Chapter 6, ***Data Address Generators.*** Describes the Data Address Generators (DAGs), addressing modes, how to modify DAG and pointer registers, memory address alignment, and DAG instructions.
- Chapter 7, ***L1 Memory.*** Describes aspects of processor memory including internal (L1) memory, address and data bus structure, and memory accesses.
- Chapter 8, ***L1-Cache Controller.*** Describes the internal (L1) memory cache controller, including instruction and data cache control and operations. It also discusses the Prefetch Buffer feature that is exclusive to the ADSP-2156x and ADSP-SC59x processors.
- Chapter 9, ***Safety, Security, Multi-Core, and Low-Power Features.*** Describes support for processor saftey and security features, including parity error detection, illegal opcode detection, and memory barrier operation. The chapters also describes some power saving features that are exclusive to the ADSP-2156x and ADSP-SC59x processors.
- Chapter 10, ***Debug Interface.*** Discusses the debug interface and how to use the SHARC processors in a test environment.
- Chapter 11, ***Program Trace Macrocell (PTM).*** Discusses the PTM, which implements a subset of Coresight Program Flow Trace Architecture (CSPFT) specification.
- Chapter 12, ***Instruction Set Reference.*** Provides reference information for the ISA and VISA instruction types, including instruction opcodes.
- Chapter 13, ***Computation Reference.*** Describes each compute operation in detail, including computation opcodes. Compute operations execute in the multiplier, the ALU, and the shifter.
- Appendix A, ***Numeric Formats.*** Provides descriptions of the supported data formats.
- Appendix B, ***Register File and Other Non-Memory Mapped Registers (REGF).*** Provides register descriptions and bit descriptions.
- Appendix C, ***Core Memory-Mapped Registers (CMMR).*** Provides register descriptions and bit descriptions.
- Appendix D, ***Branch Target Buffer Registers (BTB).*** Provides register descriptions and bit descriptions.
- Appendix E, ***L1-Cache Controller Registers (L1C).*** Provides register descriptions and bit descriptions.
- Appendix F, ***Debug-Related Registers (DBG).*** Provides register descriptions and bit descriptions.

What's New in This Manual

The ***Revision History*** table describes the major changes to the ***SHARC+ Core Programming Reference***.

Table 1-1: Revision History

Revision	Description of Changes
1.5	This revision corrects minor typographical errors and the following: • Included an additional document section for the Performance Monitor (PFM)
1.4	This revision corrects minor typographical errors and the following: • Included content for the ADSP-2159x and ADSP-SC59x • Added IIRxI bits in the REGF_IMASKP, REGF_IMASK and REGF_IRPTL registers to support the ADSP-2159x and SC59x processors • Added additional flags in the REGF_FLAGS register to support the ADSP-2159x and SC59x processors • Updated language surrounding the memory fabric infrastructure. The term *master* is replaced with *requester* and the term *slave* is replaced with *completer*. The master port is now the requester port and the slave port is now the completer port. • Added SHARC-PLUS PFM Register Descriptions Registers
1.3	This revision corrects minor typographical errors and the following issues: • Updated code example for the recips functions in 32-bit and 40-bit Operations • Updated code example for Fixed-to-Float Conversion Instructions with Scaling • Updated PFB Invalidation instructions in Prefetch Buffer
1.2	This revision corrects minor typographical errors and the following issues: • Added note to topic Coherency Between DM and PM Caches • Updated definition of IIRI and FIRI bits for REGF_IMASKP, REGF_IMASK and REGF_IRPTL registers • Changed bit field associated with the ID Code register SHDBG_REVID from 4:7 to 0:3 • Updated SHL1C_CFG.PMCASIZ enum 0 description • Updated Type 20a ISA/VISA (push/pop stack/manipulate cache) syntax for cache flushing • Added note to Address-Range Based Invalidation and Write-Back Invalidation topic • Updated topic Further Details on Range Based WBI/Invalidation • Added ADSP-2156x to SHDBG_CORE_ID register description

Table 1-1: Revision History (Continued)

Revision	Description of Changes
1.1	Not released. This revision introduces the ADSP-2156x series of SHARC+ processors and its specific features: • Prefetch Buffer (ADSP-2156x and ADSP-SC59x Only) • Low-Power Features (ADSP-2156x and ADSP-SC59x Only) It also corrects minor typographical errors and the following issues: • *Header Creation* example in the Bit Stream Manipulation Instructions topic • *Storing and Restoring Bit FIFO State* example in the Interrupts Using Bit FIFO Instructions topic • Added description of the ILLOPI and ILLOPIA bit fields in the `SHDBG_DBGREG_ILLOP` register
1.0	Initial release

Technical Support

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

- Post your questions in the processors and DSP support community at *EngineerZone*®:

  http://ez.analog.com/community/dsp

- Submit your questions to technical support directly at:

  http://www.analog.com/support

- E-mail your questions about processors, DSPs, and CrossCore Embedded Studio® (CCES) development software tools:

  Choose ***Help*** > ***Email Support***. This creates an e-mail to processor.tools.support@analog.com and automatically attaches your CCES version information and `license.dat` file.

- E-mail your questions about processors and processor applications to:

  processor.support@analog.com

  processor.tools.support@analog.com

  processor.china@analog.com

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

- Contact your Analog Devices sales office or authorized distributor. Locate one at:

  http://www.analog.com/adi-sales

- Send questions by mail to:

Analog Devices, Inc.

Three Technology Way

P.O. Box 9106

Norwood, MA 02062-9106 USA

Supported Processors

The name "*SHARC+*" indicates a DSP core incorporated into an SoC from a family of high-performance, floating-point embedded processors. Refer to the product data sheet for a complete list of supported processors.

Product Information

Product information can be obtained from the Analog Devices web site and the online help for the CCES development environment.

Analog Devices Web Site

The Analog Devices Web site, http://www.analog.com, provides information about a broad range of products—analog integrated circuits, amplifiers, converters, and digital signal processors.

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

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

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

EngineerZone

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

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

Notation Conventions

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

Example	Description
File > Close	Titles in bold style indicate the location of an item within the CrossCore Embedded Studio IDE's menu system (for example, the ***Close*** command appears on the ***File*** menu).
`{this | that}`	Alternative required items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as `this` or `that`. One or the other is required.
`[this | that]`	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional `this` or `that`.
`[this, …]`	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipsis; read the example as an optional comma-separated list of `this`.
`.SECTION`	Commands, directives, keywords, and feature names are in text with `letter gothic` font.
`filename`	Non-keyword placeholders appear in text with italic style format.
NOTE:	***NOTE:*** For correct operation, ... A note provides supplementary information on a related topic. In the online version of this book, the word ***Note*** appears instead of this symbol.
CAUTION:	***CAUTION:*** Incorrect device operation may result if ... ***CAUTION:*** Device damage may result if ... A caution identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word ***Caution*** appears instead of this symbol.
ATTENTION:	***ATTENTION*** Injury to device users may result if ... A warning identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for devices users. In the online version of this book, the word ***Warning*** appears instead of this symbol.

Register Diagram Conventions

Register diagrams use the following conventions:

- The descriptive name of the register appears at the top, followed by the short form of the name in parentheses.
- If the register is read-only (RO), write-1-to-set (W1S), or write-1-to-clear (W1C), this information appears under the name. Read/write is the default and is not noted. Additional descriptive text may follow.
- If any bits in the register do not follow the overall read/write convention, this is noted in the bit description after the bit name.
- If a bit has a short name, the short name appears first in the bit description, followed by the long name in parentheses.

- The reset value appears in binary in the individual bits and in hexadecimal to the right of the register.
- Bits marked *x* have an unknown reset value. Consequently, the reset value of registers that contain such bits is undefined or dependent on pin values at reset.
- Shaded bits are reserved.

NOTE: To ensure upward compatibility with future implementations, write back the value that is read for reserved bits in a register, unless otherwise specified.

The ***Register Diagram Example*** figure shows an example of these conventions.

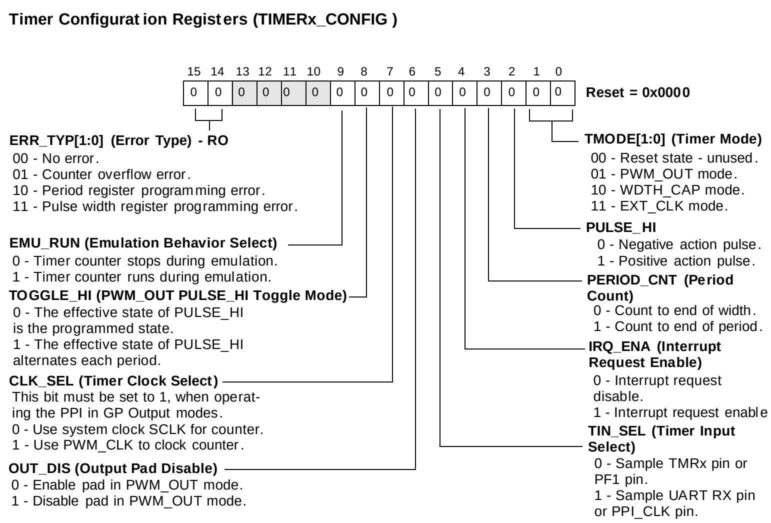

Figure 1-1: Register Diagram Example

1 Introduction

The SHARC® processors are high performance 32-bit/40-bit/64-bit fixed-point/floating-point processors used for applications, such as:

- Medical imaging
- Communications
- Military
- Audio
- Test equipment
- 3D graphics
- Speech recognition
- Motor control
- Imaging
- Automotive

The on-chip SRAM, integrated I/O peripherals, extra processing element for single-instruction, multiple-data (SIMD) support in the SHARC+ core and a rich instruction set builds on the ADSP-21000 family processor core. This combination forms a complete system-on-a-chip (SOC).

The SHARC+ core family includes distinct groups of processors:

- ADSP-215xx processors (single and multiple SHARC+ cores)
- ADSP-SC5xx processors (single and multiple SHARC+ cores with an Arm® core)

These products are differentiated by number of processor cores, on-chip memories, peripheral choices, packaging, and operating speeds. In all SHARC processors, the SHARC+ core operates in the same way. This uniform operation lets this manual apply to all groups. Where differences exist (such as external memory interfacing), they are noted.

SHARC+ Core Design Advantages

The data format used by a digital signal processor determines its ability to handle signals of differing precision, dynamic range, and signal-to-noise ratios. Because floating-point math reduces the need for scaling and probability of overflow, using a floating-point processor can ease algorithm and software development. The extent to which these guidelines are true depends on the architecture of the floating-point processor. Consistency with IEEE workstation simulations and the elimination of scaling are clearly two ease-of-use advantages. High-level-language programmability, large address spaces, and wide dynamic range allow system development time to be spent on algorithms and signal processing concerns. This architecture reduces time spent on coding in assembly language, managing code placement on memory pages, and developing routines to handle errors. The processors are highly integrated, 32-bit/40-bit/64-bit floating-point processors that provide many of these design advantages.

The SHARC processor architecture balances multiple high performance SHARC+ core with four high speed memory L1 blocks and two I/O buses. In the core, every instruction working with 32-bit or 40-bit data can execute in a single cycle. Instructions working with 64-bit floating-point data require multiple cycles.

Architectural Overview

The following sections summarize the features of each functional block.

SHARC Processor

The SHARC processors form a complete system-on-a-chip, integrating the SHARC+ core plus a crossbar including the instruction and data cache control (Internal memory interface), high-speed L1 SRAM blocks, two requester and two completer ports for connection to the system or peripheral world.

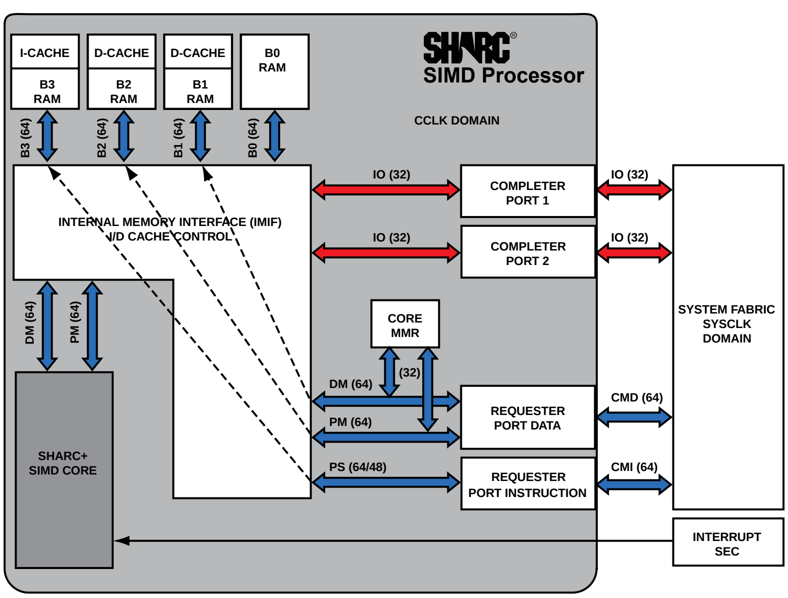

Figure 1-1: SHARC+ SIMD Core Block Diagram

SHARC+ Core

The following sections provide details of the elements in the SHARC+ core.

System Event Controller Input (SEC)

The output of the SEC controller is forwarded to the SHARC+ Core Event Controller (CEC) to respond to all system based interrupts. It also supports nesting including various SEC interrupt channel arbitration options. For all SEC channels the processor automatically stacks the arithmetic status (REGF_ASTATX and REGF_ASTATY) registers and mode (REGF_MODE1) registers in parallel with the interrupt servicing.

Instruction and data caches

The processor includes one instruction cache (block 3) and two data caches (block 2 and block 1) in L1 memory. These caches temporarily store instructions and data located in higher latency system L2 or L3 memories. The blocks 1-3 of L1 memory can be configured as instruction cache, DM data cache and PM data cache. While instruction fetches are completed through the instruction cache, DM and PM data accesses are completed through the DM- and PM-caches. The cache architecture provides a data coherence protocol between DM and PM data caches. The sizes of each of the caches and other attributes are independently configurable.

Core Memory mapped Registers (CMMR)

The core memory mapped registers control L1 I/D cache, BTB, L2 system, parity error, system control, debug and monitor functions.

SHARC+ Core Block Diagram

The SHARC+ core, shown in the ***SHARC+ SIMD Core Block Diagram*** figure, consists of two processing elements, data register files, a program sequencer, conflict cache, a branch target buffer, two DAGs, timers, debug interface and system interface.

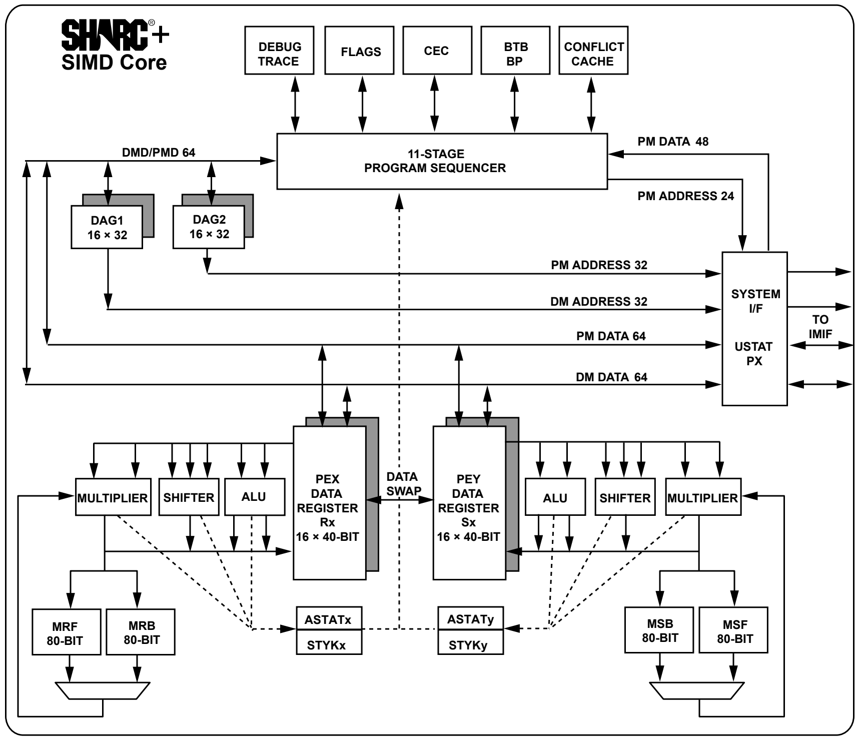

Figure 1-2: SHARC+ SIMD Core Block Diagram

Dual Processing Elements

The processor core contains two processing elements: PEx and PEy. Each element contains a data register file and three independent computation units: an arithmetic logic unit (ALU), a multiplier with an 80-bit fixed-point accumulator, and a shifter. For meeting a wide variety of processing needs, the computation units process data in a number of formats: 32-bit fixed-point integer/fractional formats (twos-complement and unsigned). 32-bit floating-point, 40-bit floating-point, and 64-bit floating-point. The floating-point operations are IEEE compatible.

The 32-bit and 64-bit floating-point compute units follow the standard IEEE format, whereas the 40-bit extended precision format has eight additional least significant bits (LSBs) of mantissa for greater accuracy compared to the 32-bit single precision format.

The ALU performs a set of arithmetic and logic operations on both fixed-point and floating-point formats. The multiplier performs floating-point or fixed-point multiplication and fixed-point multiply/ accumulate or multiply/ cumulative-subtract operations. The shifter performs logical and arithmetic shifts, bit manipulation, bit-wise field deposit and extraction, and exponent derivation operations on 32-bit operands.

Some of the compute operations are pipelined, while others are not. All shifter operations, fixed point operations performed in ALU are single cycle. Output of these operations may serve as input of any other operation in the next cycle. All 32-bit single precision ALU and multiplier operations as fixed point multiplier operations are pipelined by one cycle. A new operation in these units can be started in every cycle unless it requires an operand from one such pipelined operation. The fixed point multiply-accumulate operation is an exception to this rule. This operation can be started every cycle. Double precision floating-point operations are not fully pipelined. These operations stall the pipeline by 1-6 cycles. All units are connected in parallel, rather than serially. In a multifunction computation, the ALU and multiplier perform independent, simultaneous operations.

Complementary Processing Element (PEy)

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

- For information on PEy operations, see the Processing Elements chapter.
- For information on data accesses in SIMD mode, and data addressing in SIMD mode, see Internal Memory Access Listings in the Memory chapter.
- For information on SIMD programming, see the Instruction Set Types and Computation Types chapters.

Data Register File

Each processing element has a general-purpose data register file that transfers data between the computation units and the data buses and stores intermediate results. A register file has two sets (primary and secondary) of 16 general-purpose registers each for fast context switching. All of the registers are 40 bits wide. The ten-port data register file supports:

- write or read two operands to or from the register file,
- supply two operands to the ALU, supply two operands to the multiplier, and
- receive three results from the ALU and multiply accumulator (MAC). For more information, see the Register Files chapter.

Fore/Background Registers

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

Core Buses

The processor core has two buses-PM data and DM data. The PM bus is used to fetch instructions from memory, but may also be used to fetch data. The DM bus can only be used to fetch data from memory.

In conjunction with the instruction-conflict cache, this Super Harvard Architecture allows the core to fetch an instruction and two pieces of data in the same cycle that a data word is moved between memory and a peripheral. This architecture allows dual data fetches, when the instruction is supplied by the conflict cache.

Program Sequencer

The program sequencer supplies instruction addresses to program memory. It controls loop iterations and evaluates conditional instructions. To achieve a high execution rate, the processor employs an eleven stage pipeline to process instructions - four stages of fetch, two stages of decode, 4 stages for memory access and 2 stages for execution. The processors support both delayed and non-delayed branches for more efficient control coding. For more information, see Instruction Pipeline in the Program Sequencer chapter.

Conflict Cache

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

Branch Target Buffer

Implementation of a hardware-based branch predictor (BP) and branch target buffer (BTB) reduce branch delay. The program sequencer supports efficient branching using this branch target buffer (BTB) for conditional and unconditional instructions.

Core Event Controller (CEC)

The SHARC+ core IVT generates various core interrupts (arithmetic and circular buffer instruction flow exceptions) and SEC events (peripherals). The CEC only responds to interrupts which are unmasked (IMASK register).

Loop Sequencer

Zero-overhead loops allow efficient program sequencing. In addition to this, the sequencer allows single cycle set-up of loop. No explicit instruction is needed for counter decrement, counter check, loop-back and loop-termination. Loops are both nest-able (six levels in hardware) and interruptible.

Data Address Generators (DAGs)

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

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

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

Timer

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

Debug Port

The JTAG port supports the IEEE standard 1149.1 Joint Test Action Group (JTAG) standard for system test. This standard defines a method for serially scanning the I/O status of each component in a system. Emulators use the JTAG port to monitor and control the processor during emulation.

Emulators using this port provide full speed emulation with access to inspect and modify memory, registers, and processor stacks. JTAG-based emulation is non-intrusive and does not effect target system loading or timing.

Differences from Previous SHARC Processors

This section identifies differences between the current generation processors and previous SHARC processors: ADSP-2146x/2136x/2126x/2116x and ADSP-2106x. Like the ADSP-2116x family, the current generation is based on the original ADSP-2106x SHARC family. The current products preserve much of the ADSP-2106x architecture and is code compatible to the ADSP-2116x, while extending performance and functionality. For background information on SHARC and the ADSP-2106x Family processors, see the *ADSP-2106x SHARC User's Manual*.

The following tables show the high level differences between the SHARC processor families.

Table 1-1: Architectural Differences between SHARC Core Generations

Features	ADSP-2106x	ADSP-2116x/ 2126x	ADSP-2136x/ 2137x	ADSP-214xx	ADSP-SC5xx/ 215xx
Instruction Pipeline	3 stages	3 stages	5 stages	5 stages	11 stages
Branch Target buffer	No	No	No	No	Yes
VISA	No	No	No	Yes	Yes
DAG2 address/data width	24/48	32/64	32/64	32/64	32/64
Conflict cache 32 entries	Yes	Yes	Yes	Yes	Yes
Conflict cache ext. instruction fetch	Yes	Yes	Yes	Yes	No
L1 Instruction/Data cache	No	No	No	No	Yes
L1 Internal memory blocks	2	2	4	4	4

Table 1-1: Architectural Differences between SHARC Core Generations (Continued)

Features	ADSP-2106x	ADSP-2116x/ 2126x	ADSP-2136x/ 2137x	ADSP-214xx	ADSP-SC5xx/ 215xx
L1 Ports/memory block	Dual port	Dual port	Single Port	Single Port	Single Port
L1 parity	No	No	No	No	Yes
Dual Processing Units PEx/PEy	No	Yes	Yes	Yes	Yes
SEC Interrupt	No	No	No	No	Yes
IRQ 2-0 Interrupts	Yes	Yes	Yes	Yes	No
L1 ROM	Yes	Yes	Yes	Yes	No
Security Control	No	No	No	No	Yes

Table 1-2: L1 Memory Address Aliasing Differences between SHARC Core Generations

Features	ADSP-2106x	ADSP-2116x/ 2126x	ADSP-2136x/ 2137x	ADSP-214xx	ADSP-SC5xx/ 215xx
Byte word 8-bit	No	No	No	No	Yes
Short word 16-bit	Yes	Yes	Yes	Yes	Yes
Normal word 32-bit	Yes	Yes	Yes	Yes	Yes
Normal word 40-bit	Yes	Yes	Yes	Yes	Yes
Normal word 48-bot	Yes	Yes	Yes	Yes	Yes
Long word 64-bit	No	Yes	Yes	Yes	Yes

Table 1-3: Instruction Differences between SHARC Core Generations

Features	ADSP-2106x	ADSP-2116x/ 2126x	ADSP-2136x/ 2137x	ADSP-214xx	ADSP-SC5xx/ 215xx
COMPU(Rx,Ry);	No	Yes	Yes	Yes	Yes
Rn = BFFWRP;	No	No	No	Yes	Yes
BFFWRP = Rn | <data7>;	No	No	No	Yes	Yes
Rn = BITEXT Rx | <bitlen12>; Rn = BITEXT Rx | <bitlen12> (NU);	No	No	No	Yes	Yes
BITDEP Rx by Ry | <bitlen12>;	No	No	No	Yes	Yes
Ia=modify(Ia|<data32>);	No	No	No	Yes	Yes
Ia=bitrev(Ia|<data32>);	No	No	No	Yes	Yes
<64-bit floating-point instruction set>	No	No	No	No	Yes

Table 1-4: Register Differences between SHARC Core Generations

Registers	ADSP-2106x	ADSP-2116x/2126x	ADSP-2136x/2137x	ADSP-214xx	ADSP-SC5xx/215xx
SYSCTL/SYSCON (MMR)	Yes	Yes	Yes	Yes	Yes
SYSCTL1 (MMR)	No	No	No	Yes	Yes
FADDR	Yes	Yes	Yes	Yes	Yes
DADDR	Yes	Yes	Yes	Yes	Yes
PC	Yes	Yes	Yes	Yes	Yes
PCSTK	Yes	Yes	Yes	Yes	Yes
PCSTKP	Yes	Yes	Yes	Yes	Yes
LADDR	Yes	Yes	Yes	Yes	Yes
CURLCNTR	Yes	Yes	Yes	Yes	Yes
LCNTR	Yes	Yes	Yes	Yes	Yes
EMUCLK	Yes	Yes	Yes	Yes	Yes
EMUCLK2	Yes	Yes	Yes	Yes	Yes
PX	Yes	Yes	Yes	Yes	Yes
PX1	Yes	Yes	Yes	Yes	Yes
PX2	Yes	Yes	Yes	Yes	Yes
TPERIOD	Yes	Yes	Yes	Yes	Yes
TCOUNT	Yes	Yes	Yes	Yes	Yes
USTAT1	Yes	Yes	Yes	Yes	Yes
USTAT2	Yes	Yes	Yes	Yes	Yes
USTAT3	No	Yes	Yes	Yes	Yes
USTAT4	No	Yes	Yes	Yes	Yes
MODE1	Yes	Yes	Yes	Yes	Yes
MODE2	Yes	Yes	Yes	Yes	Yes
MMASK	No	Yes	Yes	Yes	Yes
MODE1STK	No	No	No	No	Yes
FLAGS	Yes	Yes	Yes	Yes	Yes
ASTATx	Yes	Yes	Yes	Yes	Yes
ASTATy	No	Yes	Yes	Yes	Yes
STKX	Yes	Yes	Yes	Yes	Yes
STKY	No	Yes	Yes	Yes	Yes
IRPTL	Yes	Yes	Yes	Yes	Yes

Table 1-4: Register Differences between SHARC Core Generations (Continued)

Registers	ADSP-2106x	ADSP-2116x/2126x	ADSP-2136x/2137x	ADSP-214xx	ADSP-SC5xx/215xx
IMASK	Yes	Yes	Yes	Yes	Yes
IMASKP	Yes	Yes	Yes	Yes	Yes
LIRPTL	Yes	Yes	Yes	Yes	No
Foreground					
B0–B15 (base)	Yes	Yes	Yes	Yes	Yes
I0–I15 (index)	Yes	Yes	Yes	Yes	Yes
M0–M15 (modify)	Yes	Yes	Yes	Yes	Yes
L0–L15 (length)	Yes	Yes	Yes	Yes	Yes
R0–R15 (PEx register)	Yes	Yes	Yes	Yes	Yes
S0–S15 (PEy register)	No	Yes	Yes	Yes	Yes
MRF (PEx register)	Yes	Yes	Yes	Yes	Yes
MSF (PEy register)	No	Yes	Yes	Yes	Yes
Background					
B0–B15 (base)	Yes	Yes	Yes	Yes	Yes
I0–I15 (index)	Yes	Yes	Yes	Yes	Yes
M0–M15 (modify)	Yes	Yes	Yes	Yes	Yes
L0–L15 (length)	Yes	Yes	Yes	Yes	Yes
R0–R15 (PEx register)	Yes	Yes	Yes	Yes	Yes
S0–S15 (PEy register)	No	Yes	Yes	Yes	Yes
MRB (PEx register)	Yes	Yes	Yes	Yes	Yes
MSB (PEy register)	No	Yes	Yes	Yes	Yes

Instruction Type Differences from Previous SHARC Processors

The following tables show the differences in instruction types between the current generation processors and previous SHARC processors.

Table 1-5: 48-bit Instruction Set Types

Instruction Types	ADSP-2106x	ADSP-2116x/2126x	ADSP-2136x/2137x	ADSP-214xx	ADSP-SC5xx/215xx
1a	Yes	Yes	Yes	Yes	Yes
2a	Yes	Yes	Yes	Yes	Yes

Table 1-5: 48-bit Instruction Set Types (Continued)

Instruction Types	ADSP-2106x	ADSP-2116x/2126x	ADSP-2136x/2137x	ADSP-214xx	ADSP-SC5xx/215xx
3a	Yes	Yes	Yes	Yes	Yes
3d	No	No	No	No	Yes
4a	Yes	Yes	Yes	Yes	Yes
4d	No	No	No	No	Yes
5a	Yes	Yes	Yes	Yes	Yes
6a	Yes	Yes	Yes	Yes	Yes
7a	Yes	Yes	Yes	Yes	Yes
7d	No	No	No	No	Yes
8a	Yes	Yes	Yes	Yes	Yes
9a	Yes	Yes	Yes	Yes	Yes
10a	Yes	Yes	Yes	Yes	Yes
11a	Yes	Yes	Yes	Yes	Yes
12a	Yes	Yes	Yes	Yes	Yes
13a	Yes	Yes	Yes	Yes	Yes
14a	Yes	Yes	Yes	Yes	Yes
14d	No	No	No	No	Yes
15a	Yes	Yes	Yes	Yes	Yes
16a	Yes	Yes	Yes	Yes	Yes
17a	Yes	Yes	Yes	Yes	Yes
18a	Yes	Yes	Yes	Yes	Yes
19a	Yes	Yes	Yes	Yes	Yes
20a	Yes	Yes	Yes	Yes	Yes
21a	Yes	Yes	Yes	Yes	Yes
22a	Yes	Yes	Yes	Yes	Yes
25a	Yes	Yes	Yes	Yes	Yes
26a	No	No	No	No	Yes

Table 1-6: 32-bit Instruction Set Types

Instruction Types	ADSP-2106x	ADSP-2116x/2126x	ADSP-2136x/2137x	ADSP-214xx	ADSP-SC5xx/215xx
1b	No	No	No	Yes	Yes
2b	No	No	No	Yes	Yes
3b	No	No	No	Yes	Yes

Table 1-6: 32-bit Instruction Set Types (Continued)

Instruction Types	ADSP-2106x	ADSP-2116x/2126x	ADSP-2136x/2137x	ADSP-214xx	ADSP-SC5xx/215xx
4b	No	No	No	Yes	Yes
5b	No	No	No	Yes	Yes
7b	No	No	No	Yes	Yes
9b	No	No	No	Yes	Yes
15b	No	No	No	Yes	Yes
16b	No	No	No	Yes	Yes
17b	No	No	No	Yes	Yes

Table 1-7: 16-bit Instruction Set Types

Instruction Types	ADSP-2106x	ADSP-2116x/2126x	ADSP-2136x/2137x	ADSP-214xx	ADSP-SC5xx/215xx
2c	No	No	No	Yes	Yes
3c	No	No	No	Yes	Yes
11c	No	No	No	Yes	Yes
21c	No	No	No	Yes	Yes
22c	No	No	No	Yes	Yes
25c	No	No	No	Yes	Yes

Development Tools

The SHARC+ core is supported by a complete set of software and hardware development tools, including Analog Devices' emulators and the CCES development environment. The emulator hardware that supports other Analog Devices processors also emulates the SHARC+ core.

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

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

Analog Devices DSP emulators use the IEEE 1149.1 JTAG test access port to monitor and control the target board processor during emulation. The emulator provides full speed emulation, allowing inspection and modification of memory, registers, and processor stacks. Nonintrusive in-circuit emulation is assured by the use of the processor JTAG interface-the emulator does not affect target system loading or timing.

Software tools also include Board Support Packages (BSPs). Hardware tools also include standalone evaluation systems (boards and extenders). In addition to the software and hardware development tools available from Analog Devices, third parties provide a wide range of tools supporting the SHARC+ processors. Third party software tools include DSP libraries, real-time operating systems, and block diagram design tools.

2 Register File Registers and Core Memory-Mapped Registers

The SHARC+ core is controlled by register file based registers (using the instruction set) and by core memory-mapped registers (using addresses).

Features

The register files have the following features.

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

NOTE: The register file based registers and the CMMR register are accessible by the local SHARC+ core only.

Functional Description

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

Register File Registers

The core architecture has the following register categories:

- Register file based registers
- Data registers in the PEx unit (`Dreg`)
- Complementary data register in the PEy unit (`CDreg`)
- Multiplier results registers (`MRx, MSx`)
- Data address generator registers (`Ia`, `Mb`, `Ic`, `Md`, `Ba`, `Bc`)
- System registers (`Sysreg`) in bit manipulation units

- Universal registers (*Ureg*), includes almost all processor core registers

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

Register Types and Classes

The ***SHARC+ Core Register Types and Classes*** table list the SHARC+ core registers.

Table 2-1: SHARC+ Core Register Types and Classes

Register Type	Register Classes	Registers	Function
Data Registers (*Dreg*)	RREG[*1]	`r0 - r15`	Processing element X (PEx) register file locations, fixed-point
	FREG[*2]	`f0 - f15`	PEx register file locations, floating-point
	RFREG	`r0 - r15` `f0 - f15`	PEx register file locations, fixed-point or floating-point
	RFREGDBL[*3]	`r1:0 - r15:14`	PEx 64-bit register file locations, fixed-point
		`f1:0 - f15:14`	PEx 64-bit register file locations, floating-point
Complementary Data Registers (*CDreg*)	SREG, CDREG	`s0 - s15`	Processing element Y (PEy) register file locations, fixed-point
		`sf0 - sf15`	PEy register file locations, floating-point
Multiply Result Registers (*MR*). All Multiply register are NOT part of sys register, they have separate instuctions.	MRXFBREG	`mrf, mr0f, mr1f, mr2f`	Multiplier results PEx, foreground
		`mrb, mr0b, mr1b, mr2b`	Multiplier results PEx, background
Multiply Result Registers (*MS*). All Multiply register are NOT part of sys register, they have separate instuctions.	MSXFBREG	`msf, ms0f, ms1f, ms2f`	Multiplier results PEy, foreground
		`msb, ms0b, ms1b, ms2b`	Multiplier results PEy, background
System Registers (*Sysreg*)	SYSREG	`astat, astatx, astaty`	PE, PEx, PEy arithmetic status flags and bit test flag
		`flags`	Flag pins input/output state
		`imask`	Interrupt mask
		`imaskp`	Interrupt mask pointer (for nesting)
		`irptl`	Interrupt latch
		`mmask`	Mode mask
		`mode1`	Mode 1 control and status
		`mode1stk`	Mode 1 stack (top-most entry)

Table 2-1: SHARC+ Core Register Types and Classes (Continued)

Register Type	Register Classes	Registers	Function
		`mode2`	Mode 2 control and status
		`stky`, `stkyx`, `stkyy`	PE, PEx, PEy sticky status flags and stack status flags
		`ustat1`, `ustat2`, `ustat3`, `ustat4`	User status 1, 2, 3, and 4
Index Registers (*Ia*)	I1REG	`i0` - `i7`	Index registers, Data Address Generator 1 (DAG1)
Modifier Registers (*Mb*)	M1REG	`m0` - `m7`	Modify registers, DAG1
Base Registers (*Ba*)	B1REG	`b0` - `b7`	Base registers, DAG1
Index Registers (*Ic*)	I2REG	`i8` - `i15`	Index registers, DAG2
Modifier Registers (*Md*)	M2REG	`m8` - `m15`	Modify registers, DAG2
Base Registers (*Bc*)	B2REG	`b8` - `b15`	Base registers, DAG2
Universal Register (*Ureg*) Note that *Ureg* includes the registers listed in the Registers column plus all of the registers in the register classes: RFEG, SREG, I1REG, I2REG, M1REG, M2REG, B1REG, B2REG, and SYS-REG.	UREG	`l0` - `l7`	Length registers, DAG1
		`l8` - `l15`	Length registers, DAG2
		`px`	PMD-DMD bus exchange PX1/PX2 (64-bit)
		`px1`	PMD-DMD bus exchange 1 (32 bits)
		`px2`	PMD-DMD bus exchange 2 (32 bits)
		`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
		`tperiod`	Timer period
		`tcount`	Timer counter
		`emuclk`, `emuclk2`	Emulator clocks

Table 2-1: SHARC+ Core Register Types and Classes (Continued)

Register Type	Register Classes	Registers	Function
Universal Register (*Ureg*) (additional register classes)	UREGDBL	`f1:0 - f15:14` `sf1:0 - sf15:14`	PEx and PEy double-precision floating-point data registers
	UREGXDAG1	This is a sub-set of UREG. See Function column.	Same as UREG, but omits DAG1 specific index, modify, base, and length registers
	UREGXDAG1DBL	This is a sub-set of UREG. See Function column.	Same as UREGDBL
	UREGXDAG2	This is a sub-set of UREG. See Function column.	Same as UREG, but omits DAG2 specific index, modify, base, and length registers
	UREGXDAG2DBL	This is a sub-set of UREG. See Function column.	Same as UREGDBL

*1 The RREG register class also contains a number of register sub-classes with restricted usage, including: RXAREG, RXMREG, RYAREG, and RYMREG

*2 The FREG register class also contains a number of register sub-classes with restricted usage, including: FXAREG, FXMREG, FYAREG, and FYMREG

*3 The RFREGDBL register class also contains a number of register sub-classes with restricted usage, including: DBLREG, DBLREG3, DBLXAREG, DBLXMREG, DBLYAREG, and DBLYMREG

Data Registers

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

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

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

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

Data Register Neighbor Pairing

In the long word (LW) address space, the sequencer or DAGs allow the loading and or storing of data to or from a data register pair as shown in the ***Data Register Pairs (Neighbor and Complementary) for Long Word and SIMD Mode Access*** table (see Complementary Data Register Pairs). Every even data register has an associated odd register representing a register pair. For example, `R1:0` are a neighbor data register pair. For more information, see ***DAG Instruction Types*** in the Data Address Generators chapter.

Complementary Data Register Pairs

The computational units (ALU, multiplier, and shifter) in PEx and PEy processing elements are identical. The data bus connections for the dual computational units permit asymmetric data moves to, from, and between the two processing elements. Identical instructions execute on the PEx and PEy units; the difference is the data. The data registers for PEy operations are identified (implicitly) from the PEx registers in the instruction. This implicit relationship between PEx and PEy data registers corresponds to the complementary register pairs in the ***Data Register Pairs (Neighbor and Complementary) for Long Word and SIMD Mode Access*** table. For example, the `R0` and `S0` data registers are a complementary data register pair.

NOTE: Data moves directly to the complementary registers are possible in SISD mode. For PEy computations SIMD mode is required. The instruction modifer (LW) overrides SIMD Mode. SIMD mode is not supported in LW space.

Table 2-2: Data Register Pairs (Neighbor and Complementary) for Long Word and SIMD Mode Access[1]

PEx Neighbor Pairs (side-by-side in a PE) for LW **R1:0 is a neighbor register pair** Complementary Pairs (match across PE's) for SIMD **R0 and S0 are a complementary register pair**		PEy Pairs Neighbor Pairs (side-by-side in a PE) for LW **S1:0 is a neighbor register pair** Complementary Pairs (match across PE's) for SIMD **S0 and R0 are a complementary register pair**	
R0	R1	S0	S1
R2	R3	S2	S3
R4	R5	S4	S5
R6	R7	S6	S7
R8	R9	S8	S9
R10	R11	S10	S11
R12	R13	S12	S13
R14	R15	S14	S15

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

Data and Complementary Data Register Transfers

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

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

Data and Complementary Data Register Access Priorities

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

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

It should be noted to avoid using multifunction instructions with multiple destination registers for the same source. Examples:

- Rx = any compute, `Rx = dm/pm();`
- Rx = any compute, `Ry = dm/pm () (LW);` (Rx longword pair for Ry)
- Rx = any compute, `Sx = Ry;` (SIMD enabled)
- `Rx = ALU (), Rx = MUL ();`
- Rx = 64-bit-ALU, `Ry = dm/pm();` (Rx and Ry are pairs)

Data and Complementary Data Register Swaps

Registers swaps use the special swap operator, `<->`. A register-to-register swap occurs when registers in different processing elements exchange values; for example `R0 <-> S1`. Only single, 40-bit register-to-register swaps are supported. Double register operations are also supported as shown in the example below.

```
R7 <-> S7;
R2 <-> S0;
```

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

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

System Register Bit Manipulation

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

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

However the following example is more efficient.

```
BIT SET MODE1 BITM_REGF_MODE1_PEYEN | BITM_REGF_MODE1_CBUFEN;    /* change both
modes */
/* these macros are defined in the platform header, see #include <sys/platform.h>
to get the definitions */
NOP;                           /* effect latency */
```

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

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

The core has four user status registers also classified as system registers but for general-purpose use. These registers allow flexible manipulation/testing of single or multiple individual bits in a register without affecting neighbor bits as shown in the following example.

```
USTAT1=dm(SYSCTL);
BIT SET USTAT1 BITM_SHDBG_SYSCTL_IMDWBLK2 | BITM_SHDBG_SYSCTL_IMDWBLK3; /* sets
bits 12-11 */
dm(SYSCTL)=USTAT1;
USTAT1=dm(SYSCTL);
BIT TST USTAT1 BITM_SHDBG_SYSCTL_IMDWBLK2 | BITM_SHDBG_SYSCTL_IMDWBLK3; /* test
bits 12-11 */

IF TF r15=r15+1;            /* BTF = 1 PEx OR PEy */
```

Combined Data Bus Exchange Register

The two 64-bit data DMD and PMD buses allow programs to transfer the contents of any register in the processor to any other register or to any internal memory location in a single cycle. As shown in the ***Bus Exchange (PX, PX1, and PX2) Registers*** figure, the bus exchange (`REGF_PX`) register permits data to flow between the PMD and DMD buses.

The `REGF_PX` register can work as one combined 64-bit register or as two 32-bit registers (`REGF_PX1` and `REGF_PX2`).

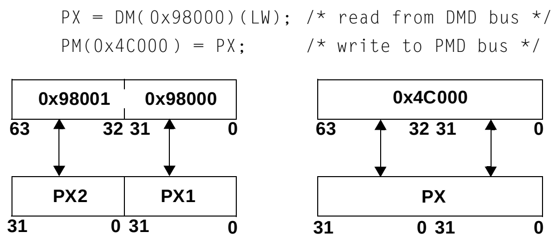

Figure 2-1: Bus Exchange (PX, PX1, and PX2) Registers

The `REGF_USTAT1`, `REGF_USTAT4`, `REGF_PX1`, and `REGF_PX2` registers allow load and store operations from memory. However, direct computations using universal registers is not supported and therefore a data move to the data register is required.

The alignment of `REGF_PX1` and `REGF_PX2` within `REGF_PX` appears in the ***PX to Dreg Transfers*** figure. The combined `REGF_PX` register is an universal register (`UREG`) that is accessible for register-to-register or memory-to-register transfers.

PX to Data Register Transfers

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

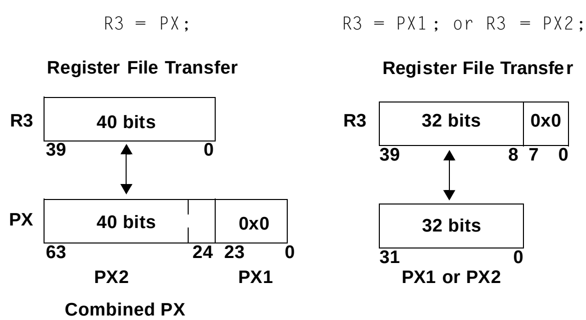

Figure 2-2: PX to Dreg Transfers

All transfers between the `PX` register (or any other internal register or memory) and any I/O processor register are 32-bit transfers (least significant 32 bits of `PX`). All transfers between the `PX` register and *`Dreg`*/*`CDreg`* (`R0-R15` or `S0-S15`) are 40-bit transfers. The most significant 40 bits are transferred as shown in the ***PX to Dreg Transfers*** figure.

Immediate 40-bit Data Register Load

Extended precision data cannot be loaded immediately by using the following code.

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

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

```
PX2 = 0x12345678;      /* load data 39-8*/
PX1 = 0x9A000000;      /* load data 7-0*/
R1 = PX;               /* R1 load with 40-bit*/
```

PX to Memory Transfers

The PX register-to-internal memory transfers over the DMD or PMD bus are either 48-bit transfers for the combined PX or 32-bit transfers (on bits 31-0 of the bus) for PX1 or PX2. The ***PX, PX1, PX2 Register-to-Memory Transfers on DM or PM Data Bus*** figure shows these transfers.

The figure also shows that during a transfer between PX1 or PX2 and internal memory, the bus transfers the lower 32 bits of the register. During a transfer between the combined PX register and internal memory, the bus transfers the upper 48 bits of PX and zero-fills the lower 16 bits.

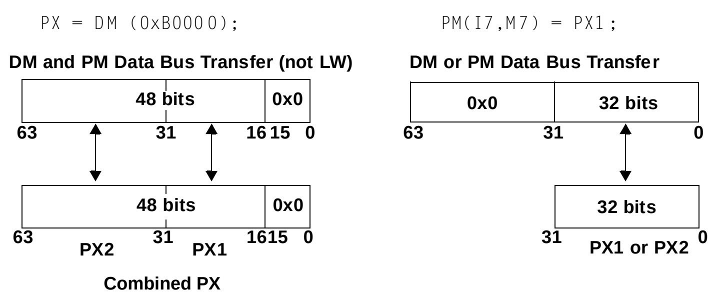

Figure 2-3: PX, PX1, PX2 Register-to-Memory Transfers on DM or PM Data Bus

PX to Memory LW Transfers

The ***PX Register-to-Memory Transfers on PM Data Bus (LW)*** figure shows the transfer size between PX and internal memory over the PMD or DMD bus when using the long word (LW) option.

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

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

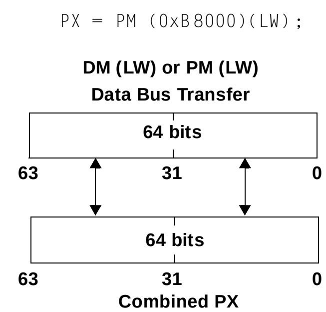

Figure 2-4: PX Register-to-Memory Transfers on PM Data Bus (LW)

Uncomplementary Ureg to Memory LW Transfers

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

```
I0 = 0x4F800;
M0 = 0x1;
L0 = 0x0;
DM(I0,M0) = 0xabbaabba;
```

Long word accesses using the `USTATx` registers is shown below.

```
USTAT1 = DM (LW address);   /* Loads only USTAT1 in SISD mode */
DM (LW address) = USTAT1;   /* Stores both USTAT1 and USTAT2 */
```

Core Memory Mapped Registers (CMMR)

The SHARC+ SoC supports a core-based address range to control the following modules.

- System control MMR. This register is used for system control, 32/40 bit IEEE floating data transfer and SW reset + Shared L2 Arm cache for data CMMR available for shared Arm L2 cache with SHARC+ data port.
- Miscellaneous Core MMRs. These registers include L1 Parity Control that are used for control and status of L1 Instruction, data and IO. See SHARC-PLUS CMMR Register Descriptions chapter.
- Branch Target buffer MMRs. These registers are used for control and status of L1 Branch target buffer and branch prediction. See SHARC-PLUS SHBTB Register Descriptions chapter.
- L1 Instruction/Data cache MMRs. These registers available for control and status of L1 Instruction and data caches. See SHARC-PLUS SHL1C Register Descriptions chapter.
- Emulation/Debug control MMRs. These registers are used for debugging the SHARC+ core. See SHARC-PLUS SHDBG Register Descriptions chapter.

For the valid address range refer to the product data sheet.

NOTE: The CMMR registers are only accessible by the local SHARC+ core.

Operating Modes

The following sections detail the operation of the register files.

Alternate (Secondary) Data Registers

Each data register file has an alternate data register set. To facilitate fast context switching, the processor includes alternate register sets for data, results, and data address generator registers. Bits in the `REGF_MODE1` register control when alternate registers become accessible. While inaccessible, the contents of alternate registers are not affected by processor operations.

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

The alternate register sets for data and results are shown in the ***Alternate (Secondary) Data Register File*** figure. For more information on alternate data address generator registers, see ***Alternate (Secondary) DAG Registers*** in the Registers appendix. Bits in the `REGF_MODE1` register can activate independent alternate data register sets: the lower half (R0-R7) and the upper half (R8-R15). To share data between contexts, a program places the data to be shared in one half of either the current processing element's register file or the opposite processing element's register file and activates the alternate register set of the other half. The register files consist of a primary set of 16 x 40-bit registers and an alternate set of 16 x 40-bit registers.

Alternate (Secondary) Data Registers SIMD Mode

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

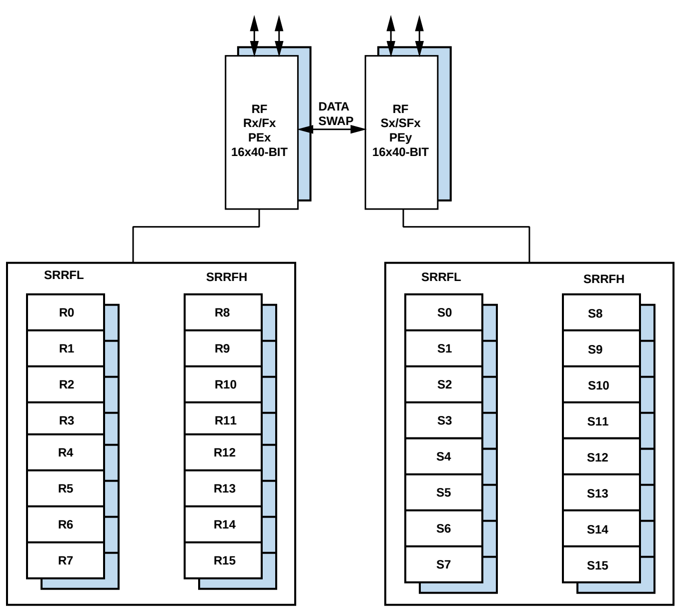

Figure 2-5: Alternate (Secondary) Data Register File

Ureg/Sysreg SIMD Mode Transfers

The ***Complementary Register Pairs*** table shows the PX registers (*Ureg*), USTATx registers (*Sysreg*), and their complementary registers (*CUreg* and *CSysreg*) relationships.

Table 2-3: Universal and System Register Complementary Pairs (CUreg and CSysreg)

USTAT1	USTAT2
USTAT3	USTAT4
PX1	PX2

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

```
PX1 = R0; /* R0 32-bit explicit move to PX1, and S0 32-bit implicit move to PX2 */
PX = R0;  /* R0 40-bit explicit move to PX, but no implicit move for S0 */
```

However, the following exceptions should be noted:

- Transfers between USTATx and PX registers as in the following example and figure (***Transfers Between USTATx and PX Registers***). Note that all user status registers behave in this manner.

  ```
  PX = USTAT1; /* loads PX1 with USTAT1 and PX2 with USTAT2 */
  USTAT1 = PX; /* loads only PX1 to USTAT1 */
  ```

- Transfers between DAG and other system registers and the PX register as shown in the following example:

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

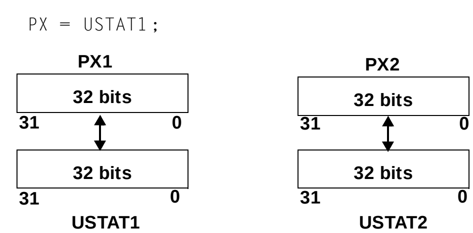

Figure 2-6: Transfers Between USTATx and PX Registers

Interrupt Mode Mask

On the SHARC+ cores, programs can automatically mask individual operating mode bits of the REGF_MODE1 register when entering into an ISR by setting bits in the REGF_MMASK register. This improves interrupt handling performance and helps ensure that interrupt handler code runs with operating modes set consistently.

For the data registers the alternate registers (SRRFH/L) are optional masks in use. For more information, see the Program Sequencer chapter.

3 Processing Elements

The PEx and PEy processing elements perform numeric algorithm processing. Each element contains a data register file and three computation units; an arithmetic/logic unit (ALU), a multiplier, and a barrel shifter. Computational instructions for these elements include both fixed-point and floating-point operations, and each computational instruction operating on 32-bit or 40-bit data executes in a single cycle. Single precision floating-point/multiplier instructions are 2 cycle computes that include register file storage. Computational instructions operating on 64-bit floating-point data (64-bits) require multiple cycles.

Features

The processing elements have the following features.

- ***Data Formats.*** The units support 32-bit fixed-point and floating-point single precision data (IEEE 32-bit), extended precision data (40-bit), and 64-bit data (IEEE 64-bit).
- ***Arithmetic/logic unit.*** The ALU performs arithmetic and logic operations on fixed-point and floating-point data.
- ***Multiplier.*** The multiplier performs floating-point and fixed-point multiplication and executes fixed-point multiply/add and multiply/subtract operations.
- ***Barrel Shifter.*** The barrel shifter performs bit shifts, bit field, and bit stream manipulation on 32-bit operands. The shifter can also derive exponents.
- ***Multifunction.*** The ALU and Multiplier support simultaneous operations for fixed- and floating-point data formats. The fixed-point multiplier can return results as 32 or 80 bits.
- ***One Cycle Arithmetic Pipeline.*** All computation instructions operating on 32-bit data and 40-bit data execute in one cycle. Computational instructions operating on 64-bit data execute over multiple cycles.
- ***Multi Precision Arithmetic.*** The ALU and multiplier support instructions/options for 64-bit precision.

Functional Description

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

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

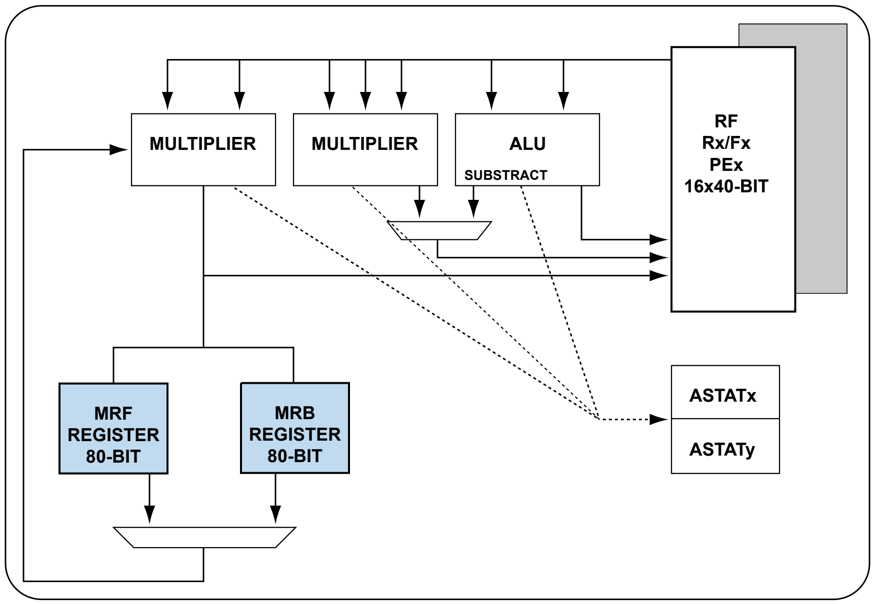

Figure 3-1: Computational Block

Single Cycle Processing

All the unconditional fixed-point compute (excluding multiply) instructions take a single cycle to complete and the results can be used in the immediately following instruction (compute or otherwise) without incurring any stalls.

For the conditional compute instruction, if the condition used was set in the immediately preceding instruction, then the compute is treated as a double cycle compute.

Data Forwarding in Processing Units

Splitting the execute phases into two stages in the SHARC+ core results in additional dependencies and forwarding logic paths. The fixed-point ALU compute is a single-cycle execute where forwarding across the immediately next dependent compute instruction is possible (M4 to M3 pipeline stage). In cases of compute-compute dependency involving either floating-point or multiply computations, a stall is generated if the dependent instruction immediately follows or data is forwarded in case of dependency across two cycles.

Data forwarding can occur across cycles, for example from the E2 to the M3 pipeline stage. The following sequence of instructions illustrates the need for forwarding across cycles.

```
r0 = r1 + r2;
nop;
r3 = r0 + r4;
```

The 3rd instruction operand `r0` is forwarded from the first instruction result from its E2 phase. The compute instructions can be Fixed/Floating, ALU or multiply.

Special Considerations

In the following code sequence the second register move instruction is dependent on the first multiply instruction, and the 3rd multiply instruction is dependent on the 2nd register transfer instruction. A forwarding path from the E2 to the M4 to the M3 pipeline stage is introduced by forwarding the first multiplier result directly to the third multiplier input. Two data forwarding paths are introduced, E2 to M4 and E2 to M3 pipeline stages.

```
r0 = r1*r2;
r3=r0;
r4= r3*r1;
```

Data Format for Computation Units

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

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

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

Arithmetic Status

The multiplier and ALU each provide exception information when executing floating-point or fixed-point operations.. Each unit updates overflow, underflow, and invalid operation flags in the processing element's arithmetic status (`REGF_ASTATX` and `REGF_ASTATY`) registers and sticky status (`REGF_STKYX` and `REGF_STKYY`) registers. An underflow, overflow, or invalid operation from any unit also generates a maskable interrupt. There are three ways to use floating-point or fixed-point exceptions from computations in program sequencing.

- Enable interrupts and use an interrupt service routine (ISR) to handle the exception condition immediately. This method is appropriate if it is important to correct all exceptions as they occur.
- Use conditional instructions to test the exception flags in the `REGF_ASTATX` or `REGF_ASTATY` registers after the instruction executes. This method permits monitoring each instruction's outcome.
- Use the bit test (`BTST`) instruction to examine exception flags in the `REGF_STKYY` register after a series of operations. If any flags are set, some of the results are incorrect. Use this method when exception handling is not critical.

Computation Status Update Priority

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

```
R0=R1+R2, ASTATx=R6; /* R6 overrides ALU status */
F0=F1*F2, STKYx=F6;  /* F6 overrides MUL status */
```

For information on conditional instruction execution based on arithmetic status, see Conditional Instruction Execution in the Program Sequencer chapter.

SIMD Computation and Status Flags

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

Table 3-1: Computation Status Register Pairs

ASTATx	ASTATy
STKYx	STKYy

Arithmetic Logic Unit (ALU)

The ALU performs arithmetic operations on fixed-point or floating-point data and logical operations on fixed-point data. ALU fixed-point instructions operate on 32-bit or 64-bit fixed-point operands and output 32-bit or 64-bit fixed-point results. ALU floating-point instructions operate on 32-bit, 40-bit, or 64-bit floating-point operands and output 32-bit, 40-bit, or 64-bit floating-point results. ALU instructions include:

- Floating-point addition, subtraction, add/subtract, average
- Fixed-point addition, subtraction, add/subtract, average
- Floating-point manipulation - binary log, scale, mantissa
- Fixed-point multi precision arithmetic (add with carry, subtract with borrow)
- Logical AND, OR, XOR, NOT
- Functions - ABS, PASS, MIN, MAX, CLIP, COMPARE
- Format conversion (fixed-point to/from floating-point, single-precision to/from 64-bit)
- Floating-point iterative reciprocal and reciprocal square root functions

Functional Description

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

If the ALU operation is fixed-point, the inputs are treated as 32-bit fixed-point operands. The ALU transfers the upper 32 bits from the source location in the register file. For fixed-point operations, the result(s) are 32-bit fixed-point values. Some floating-point operations (`LOGB`, `MANT` and `FIX`) can also yield fixed-point results.

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

ALU Instruction Types

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

Compare Accumulation Instruction

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

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

Fixed-to-Float Conversion Instructions

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

```
Fn = FLOAT Rx; /* floating-point */
Rn = FIX Fx;   /* fixed-point */
```

Fixed-to-Float Conversion Instructions with Scaling

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

```
Ry = -31;
Fn = FLOAT Rx BY Ry; /* floating-point [-1.0 to 1.0) */
Ry = 31;
Rn = FIX Fx BY Ry;   /* fixed-point 1.31 format */
```

Reciprocal/Square Root Instructions

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

```
Fn = RECIPS Fx; /* creates seed for reciprocal */
Fn = RSQRTS Fx; /* creates seed for reciprocal square root */
```

Divide Instruction

The SHARC+ core supports a multi-cycle floating-point divide instruction. The RECIPS instruction is used to simplify the divide implementation instruction by using an iterative convergence algorithm. For more information, see the Computation Types chapter.

Clip Instruction

The clip instruction (CLIP) is very similar to the multiplier saturate (SAT) instruction, however the clipping (saturation) level is an operand within the instruction.

```
Rn = CLIP Rx by Ry; /* clip level stored in Ry register */
```

Multiprecision Instructions

The add with carry and the subtract with borrow allows the implementation of 64-bit operations.

```
Rn = Rx + Ry + CI;     /* adds with carry from status register */
Rn = Rx - Ry + CI -1; /* subtracts with borrow from status register */
```

Arithmetic Status

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

- ALU result zero or floating-point underflow, (AZ)
- ALU overflow, (AV)
- ALU result negative, (AN)
- ALU fixed-point carry, (AC)
- ALU input sign for ABS, MANT operations, (AS)
- ALU floating-point invalid operation, (AI)
- Last ALU operation was a floating-point operation, (AF)
- Compare accumulation register results of last eight compare operations, (CACC)

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

- ALU floating-point underflow, (AUS)
- ALU floating-point overflow, (AVS)
- ALU fixed-point overflow, (AOS)
- ALU floating-point invalid operation, (AIS)

ALU Instruction Summary

The ***Fixed-Point ALU Instruction Summary (AF Flag = 0)*** and ***Floating-Point ALU Instruction Summary*** tables list the ALU instructions and show how they relate to the ASTATx/ASTATy and STKYx/STKYy flags. For more information on assembly language syntax, see the ***Instruction Set Types*** chapter and the ***Computation Types*** chapter. In these tables, note the meaning of the following symbols.

- Rn, Rx, Ry indicate any register file location; treated as fixed-point
- Fn, Fx, Fy indicate any register file location; treated as floating-point
- * indicates that the flag may be set or cleared, depending on the results of instruction
- ** indicates that the flag may be set (but not cleared), depending on the results of the instruction
- - indicates no effect
- In SIMD mode all instructions use the complement data registers

Table 3-2: Fixed-Point ALU Instruction Summary (AF Flag = 0)

Instruction	ASTATx, ASTATy Status Flags							STKYx, STKYy Status Flags			
Fixed-Point:	AZ	AV	AN	AC	AS	AI	CACC	AUS	AVS	AOS	AIS
RN = RX + RY;	*	*	*	*	0	0	-	-	-	**	-
RN = RX – RY;	*	*	*	*	0	0	-	-	-	**	-
RN = RX + RY + ci;	*	*	*	*	0	0	-	-	-	**	-
RN = RX – RY + ci – 1;	*	*	*	*	0	0	-	-	-	**	-
RN = (RX + RY) / 2;	*	0	*	*	0	0	-	-	-	-	-
comp (RX, RY);	*	0	*	0	0	0	*	-	-	-	-
compu (RX, RY);	*	0	*	0	0	0	*	-	-	-	-
RN = RX + ci;	*	*	*	*	0	0	-	-	-	**	-
RN = RX + ci – 1;	*	*	*	*	0	0	-	-	-	**	-
RN = RX + 1;	*	*	*	*	0	0	-	-	-	**	-
RN = RX – 1;	*	*	*	*	0	0	-	-	-	**	-
RN = –RX;	*	*	*	*	0	0	-	-	-	**	-
RN = abs RX;	*	*	0	0	*	0	-	-	-	**	-
RN = pass RX;	*	0	*	0	0	0	-	-	-	-	-
RN = RX and RY;	*	0	*	0	0	0	-	-	-	-	-
RN = RX or RY;	*	0	*	0	0	0	-	-	-	-	-
RN = RX xor RY;	*	0	*	0	0	0	-	-	-	-	-
RN = not RX;	*	0	*	0	0	0	-	-	-	-	-
RN = min (RX, RY);	*	0	*	0	0	0	-	-	-	-	-
RN = max (RX, RY);	*	0	*	0	0	0	-	-	-	-	-
RN = clip RX by RY;	*	0	*	0	0	0	-	-	-	-	-

Table 3-3: Floating-Point ALU Instruction Summary (AF Flag = 1)

Instruction	ASTATx, ASTATy Status Flags							STKYx, STKYy Status Flags			
Floating-Point:	AZ	AV	AN	AC	AS	AI	CACC	AUS	AVS	AOS	AIS
FN = FX + FY;	*	*	*	0	0	*	-	**	**	-	**
FN = FX – FY;	*	*	*	0	0	*	-	**	**	-	**
FN = abs (FX + FY);	*	*	0	0	0	*	-	**	**	-	**
FN = abs (FX – FY);	*	*	0	0	0	*	-	**	**	-	**
FN = (FX + FY) / 2;	*	0	*	0	0	*	-	**	-	-	**
comp (FX, FY);	*	0	*	0	0	*	*	-	-	-	**
FN = –FX;	*	0	*	0	0	*	-	-	-	-	**
FN = abs FX;	*	0	0	0	*	*	-	-	-	-	**
FN = pass FX;	*	0	*	0	0	*	-	-	-	-	**
FN = rnd FX;	*	*	*	0	0	*	-	-	**	-	**
FN = scalb FX by RY;	*	*	*	0	0	*	-	**	**	-	**
RN = mant FX;	*	*	0	0	*	*	-	-	**	-	**
RN = logb FX;	*	*	*	0	0	*	-	-	**	-	**
RN = fix FX;	*	*	*	0	0	*	-	**	**	-	**
RN = fix FX by RY;	*	*	*	0	0	*	-	**	**	-	**
RN = trunc FX;	*	0	*	0	0	*	-	**	-	-	**
RN = trunc FX by RY;	*	0	*	0	0	*	-	**	-	-	**
FN = float RX;	*	0	*	0	0	0	-	-	-	-	-
FN = float RX by RY;	*	*	*	0	0	0	-	**	**	-	-
FN = recips FX;	*	*	*	0	0	*	-	**	**	-	**
FN = rsqrts FX;	*	*	*	0	0	*	-	-	**	-	**
FN = FX copysign FY;	*	0	*	0	0	*	-	-	-	-	**
FN = min (FX, FY);	*	0	*	0	0	*	-	-	-	-	**
FN = max (FX, FY);	*	0	*	0	0	*	-	-	-	-	**
FN = clip FX by FY;	*	0	*	0	0	*	-	-	-	-	**

Table 3-4: 64-bit Floating-Point ALU Instruction Summary (AF Flag = 1)

Instruction	ASTATx, ASTATy Status Flags							STKYx, STKYy Status Flags			
64-bit Floating-Point:	AZ	AV	AN	AC	AS	AI	CACC	AUS	AVS	AOS	AIS
FM:N = FX:Y + FZ:W;	*	*	*	0	0	*	-	**	**	-	**
FM:N = FX:Y - FZ:W;	*	*	*	0	0	*	-	**	**	-	**

Table 3-4: 64-bit Floating-Point ALU Instruction Summary (AF Flag = 1) (Continued)

Instruction	ASTATx, ASTATy Status Flags							STKYx, STKYy Status Flags			
64-bit Floating-Point:	AZ	AV	AN	AC	AS	AI	CACC	AUS	AVS	AOS	AIS
comp (FX:Y, FZ:W);	*	0	*	0	0	*	*	-	-	-	**
FM:N = - FX:Y;	*	0	*	0	0	*	-	-	-	-	**
FM:N = abs FX:Y;	*	0	0	0	*	*	-	-	-	-	**
FM:N = pass FX:Y;	*	0	*	0	0	*	-	-	-	-	**
FM:N = scalb FX:Y by RY;	*	*	*	0	0	*	-	**	**	-	**
RN=fix FX:Y;	*	*	*	0	0	*	-	**	**	-	**
RN = fix FX:Y by RY;	*	*	*	0	0	*	-	**	**	-	**
RN = trunc FX:Y;	*	*	*	0	0	*	-	**	**	-	**
RN = trunc FX:Y by RY;	*	*	*	0	0	*	-	**	**	-	**
FM:N = float RX;	*	0	*	0	0	0	-	-	-	-	-
FM:N = float RX by RY;	*	*	*	0	0	0	-	**	**	-	-
FM:N = cvt FX;	*	0	*	0	0	*	-	-	-	-	**
FN = cvt FX:Y;	*	*	*	0	0	*	-	**	**	-	**

Multiplier

The multiplier performs fixed-point or floating-point multiplication and fixed-point multiply/accumulate operations. Fixed-point multiply/accumulates are available with cumulative addition or cumulative subtraction. Multiplier floating-point instructions operate on 32-bit, 40-bit, or 64-bit floating-point operands and output 32-bit, 40-bit, or 64-bit floating-point results. Multiplier fixed-point instructions operate on 32-bit fixed-point data and produce 80-bit results. Inputs are treated as fractional or integer, unsigned or two's-complement. Multiplier instructions include:

- Floating-point multiplication
- Fixed-point multiplication
- Fixed-point multiply/accumulate with addition, rounding optional
- Fixed-point multiply/accumulate with subtraction, rounding optional
- Rounding multiplier result register
- Saturating multiplier result register
- Fixed point multi-precision arithmetic (signed/signed, unsigned/unsigned or unsigned/signed options)

Functional Description

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

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

Multiplier Inputs

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

Multiplier Result Register

Fixed-point operations place 80-bit results in the multiplier's foreground register (REGF_MRF/REGF_MSF) or background register (REGF_MRB/REGF_MSB), depending on which is active. For more information on selecting the result register, see ***Alternate (Secondary) Data Registers*** in the Registers File chapter.

The location of a result in the MRF register's 80-bit field depends on whether the result is in fractional or integer format, as shown in the ***Multiplier Fixed-Point Result Placement*** figure. If the result is sent directly to a data register, the 32-bit result with the same format as the input data is transferred, using bits 6332 for a fractional result or bits 310 for an integer result. The eight LSBs of the 40-bit register file location are zero-filled.

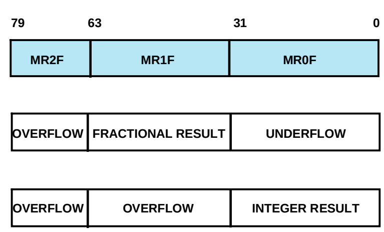

Figure 3-2: Multiplier Fixed-Point Result Placement

Fractional results can be rounded-to-nearest before being sent to the register file. If rounding is not specified, discarding bits 310 effectively truncates a fractional result (rounds to zero). For more information on rounding, see Rounding Mode.

The REGF_MRF register (see the ***MR to Data Register Transfers Formats*** figure) is comprised of the REGF_MR2F, REGF_MR1F, and REGF_MR0F registers, which individually can be read from or written to the register file. Each of these registers has the same format. When data is read from the REGF_MR2F register (guard bits), it is sign-extended to 32 bits. The processor core zero-fills the eight LSBs of the 40-bit register file location when data is written from the REGF_MR2F, REGF_MR1F, or REGF_MR0F registers to a register file location. When data is written into the REGF_MR2F, REGF_MR1F, or REGF_MR0F registers from the 32 MSBs of a register file location, the eight LSBs are ignored. Data written to the REGF_MR1F register is sign-extended to REGF_MR2F, repeating the MSB of REGF_MR1F in the 16 bits of the REGF_MR2F register. Data written to the REGF_MR0F register is not sign-extended.

Note that the multiply result register (REGF_MRF, REGF_MRB) is not an orthogonal register in the instruction set. Only specific instructions decode it as an operand or as a result register (no universal register). For more information, see ***Multiplier Fixed-Point Computations*** in the Computation Types chapter.

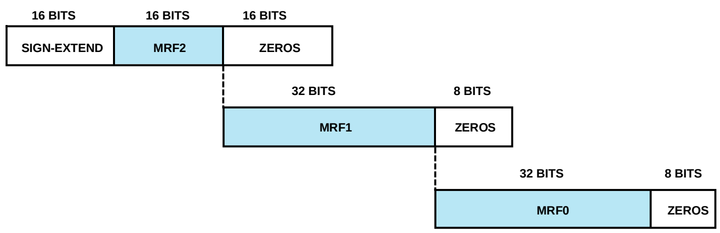

Figure 3-3: MR to Data Register Transfers Formats

Multiply Register Instruction Types

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

Clear MRx Instruction

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

Round MRx Instruction

The RND operation (MRF = RND MRF) applies only to fractional results, integer results are not effected. This operation performs a round to nearest of the 80-bit MRF value at bit 32, for example, the MR1F- MR0F boundary. Rounding a fixed-point result occurs as part of a multiply or multiply/ accumulate operation or as an explicit operation on the MRF register. The rounded result in MR1F can be sent to the register file or back to the same MRF register. To round a fractional result to zero (truncation) instead of to nearest, a program transfers the unrounded result from MR1F, discarding the lower 32 bits in MR0F.

Multi Precision Instructions

The multiplier supports the following data operations for 64-bit data.

```
MRF = Rx * Ry (SSF); /* signed x signed/fractional */
MRF = Rx * Ry (SUF); /* signed x unsigned/fractional */
MRF = Rx * Ry (USF); /* unsigned x signed/fractional */
MRF = Rx * Ry (UUF); /* unsigned x unsigned/fractional */
```

Saturate MRx Instruction

The SAT operation (MRF = SAT MRF) sets MRF to a maximum value if the MRF value has overflowed. Overflow occurs when the MRF value is greater than the maximum value for the data format-unsigned or two's-complement and integer or fractional-as specified in the saturate instruction. The six possible maximum values appear in the ***Fixed-Point Format Maximum Values (Saturation)*** table. The result from MRF saturation can be sent to the register file or back to the same REGF_MRF register.

Table 3-5: Fixed-Point Format Maximum Values (Saturation)

Maximum Number	(Hexadecimal)		
	MR2F	MR1F	MR0F
Two's-complement fractional (positive)	0000	7FFF FFFF	FFFF FFFF
Two's-complement fractional (negative)	FFFF	8000 0000	0000 0000
Two's-complement integer (positive)	0000	0000 0000	7FFF FFFF
Two's-complement integer (negative)	FFFF	FFFF FFFF	8000 0000
Unsigned fractional number	0000	FFFF FFFF	FFFF FFFF
Unsigned integer number	0000	0000 0000	FFFF FFFF

Arithmetic Status

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

- Multiplier result negative (MN)
- Multiplier overflow, (MV)
- Multiplier underflow, (MU)
- Multiplier floating-point invalid operation, (MI)

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

- Multiplier fixed-point overflow, (MOS)
- Multiplier floating-point overflow, (MVS)
- Multiplier underflow, (MUS)
- Multiplier floating-point invalid operation, (MIS)

Multiplier Instruction Summary

The ***Fixed-Point Multiplier Instruction Summary*** and ***Floating-Point Multiplier Instruction Summary*** tables list the multiplier instructions and describe how they relate to the ASTATX/ASTATY and STKYX /STKYY flags. For more information on assembly language syntax, see the Instruction Set Types chapter and the Computation Types chapter. In the tables, note the meaning of the following symbols:

- `Rn`, `Rx`, `Ry` indicate any register file location; treated as fixed-point
- `Fn`, `Fx`, `Fy` indicate any register file location; treated as floating-point
- * indicates that the flag may be set or cleared, depending on results of instruction
- ** indicates that the flag may be set (but not cleared), depending on results of instruction
- - indicates no effect
- The Input Mods column indicates the types of optional modifiers that can be applied to the instruction inputs. For a list of modifiers, see the ***Input Modifiers for Fixed-Point Multiplier Instruction*** table.
- In SIMD mode all instruction uses the complement data/multiply result registers.

Table 3-6: Multiplier Result (MR) Register Data Move Operations Summary

Instruction	ASTATx, ASTATy Flags				STKYx, STKYy Flags			
Fixed-Point	**MU**	**MN**	**MV**	**MI**	**MUS**	**MOS**	**MVS**	**MIS**
(mrf | mrb) = RN;	0	0	0	0	-	-	-	-
RN = (mrf | mrb);	0	0	0	0	-	-	-	-

Table 3-7: Fixed-Point Multiplier Instruction Summary

Instruction	ASTATx, ASTATy Flags				STKYx, STKYy Flags			
Fixed-Point	**MU**	**MN**	**MV**	**MI**	**MUS**	**MOS**	**MVS**	**MIS**
(mrf | mrb) = MRF + RX * RY MOD1;	*	*	*	0	-	**	-	-
RN = (mrf | mrb) + RX * RY MOD1;	*	*	*	0	-	**	-	-
(mrf | mrb) = (mrf | mrb) – RX * RY MOD1;	*	*	*	0	-	**	-	-
RN = (mrf | mrb) – RX * RY MOD1;	*	*	*	0	-	**	-	-
(RN | mrf | mrb) = RX * RY MOD1;	*	*	*	0	-	**	-	-
(RN | mrf | mrb) = rnd (mrf | mrb) MOD3;	*	*	*	0	-	**	-	-
(RN | mrf | mrb) = sat (mrf | mrb) MOD2;	*	*	0	0	-	-	-	-
(mrf | mrb) = 0;	0	0	0	0	-	-	-	-

Table 3-8: Input Modifiers for Fixed-Point Multiplier Instruction

Input Modes (1-2-3) from the Fixed-Point Multiplier Instruction Summary table	Input Mods-Options For Fixed-Point Multiplier Instructions
1	(SSF), (SSI), (SSFR), (SUF), (SUI), (SUFR), (USF), (USI), (USFR), (UUF), (UUI), or (UUFR)

Table 3-8: Input Modifiers for Fixed-Point Multiplier Instruction (Continued)

Input Modes (1-2-3) from the Fixed-Point Multiplier Instruction Summary table	Input Mods-Options For Fixed-Point Multiplier Instructions
2	(SF), (SI), (UF), or (UI) saturation only
3	(SF) or (UF) rounding only
Note the meaning of the following symbols in this table: *Signed input — S* *Unsigned input — U* *Integer input — I* *Fractional input — F* *Fractional inputs, Rounded output — FR* *Note that (SF) is the default format for one-input operations, and (SSF) is the default format for two-input operations.*	

Table 3-9: Floating-Point Multiplier Instruction Summary

Instruction	ASTATx, ASTATy Flags				STKYx, STKYy Flags			
Floating-Point	**MU**	**MN**	**MV**	**MI**	**MUS**	**MOS**	**MVS**	**MIS**
FN = FX * FY;	*	*	*	*	**	-	**	**

Table 3-10: 64-bit Floating-Point Multiplier Instruction Summary

Instruction	ASTATx, ASTATy Flags				STKYx, STKYy Flags			
64-bit Floating-Point	**MU**	**MN**	**MV**	**MI**	**MUS**	**MOS**	**MVS**	**MIS**
FM:N = FX:Y * FZ:W;	*	*	*	*	**	-	**	**
FM:N = FX:Y * FY;	*	*	*	*	**	-	**	**
FM:N = FX * FY;	*	*	*	*	**	-	**	**

Barrel Shifter

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

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

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

- Optional immediate data for one input within the instruction

Functional Description

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

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

The shifter does not make use of the ALU carry bit, it uses its own status bits.

Shifter Instruction Types

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

Shift Compute Category

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

Shift Immediate Category

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

Bit Manipulation Instructions

In the following example, `Rx` is the X input, `Ry` is the Y input, and `Rn` is the Z input. The shifter returns one output (`Rn`) to the register file.

```
Rn = Rn OR LSHIFT Rx BY Ry;
```

As shown in the ***Register File Fields for Shifter Instructions*** figure, the shifter fetches input operands from the upper 32 bits of a register file location (bits 39-8) or from an immediate value in the instruction.

The X input and Z input are always 32-bit fixed-point values. The Y input is a 32-bit fixed-point value or an 8-bit field (`SHF8`), positioned in the register file. These inputs appear in the ***Register File Fields for Shifter Instructions*** figure.

Some shifter operations produce 8 or 6-bit results. As shown in the ***Register File Fields for Shifter Instructions*** figure, the shifter places these results in the `SHF8` field or the `bit6` field and sign-extends the results to 32 bits. The shifter always returns a 32-bit result.

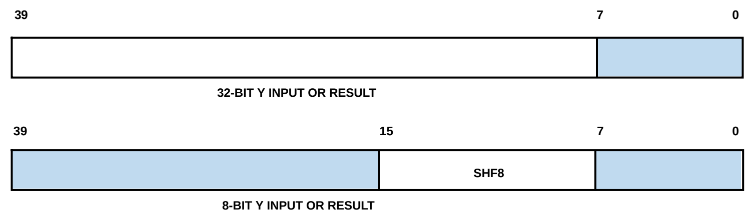

Figure 3-4: Register File Fields for Shifter Instructions

Bit Field Manipulation Instructions

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

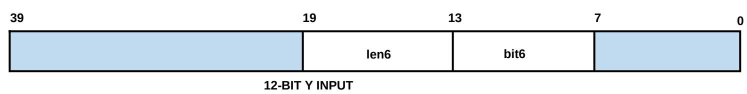

Figure 3-5: Register File Fields for FDEP, FEXT Instructions

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

```
Rn = FDEP Rx By Ry
```

The ***Bit Field Deposit Example*** figure shows bit placement for the following field deposit instruction.

```
R0 = FDEP R1 By R2;
```

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

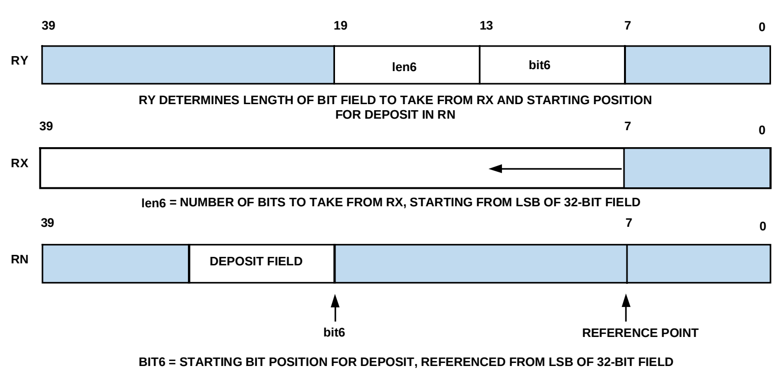

Figure 3-6: Bit Field Deposit Instruction

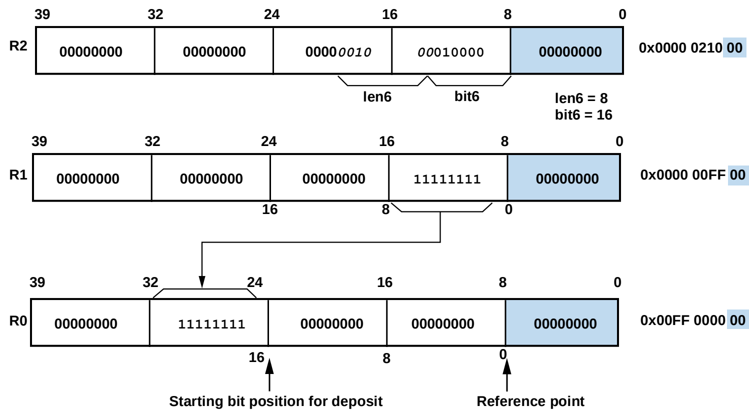

Figure 3-7: Bit Field Deposit Example

The ***Bit Field Extract Instruction*** figure shows bit placement for the following field extract instruction.

```
R3 = FEXT R4 By R5;
```

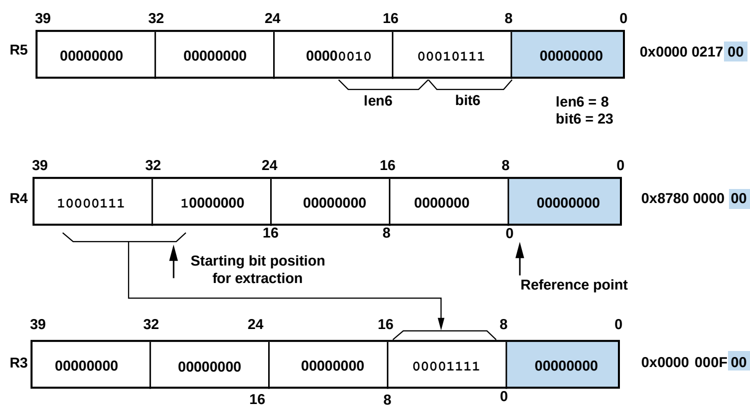

Figure 3-8: Bit Field Extract Instruction

NOTE: The `FEXT` instruction bits to the left of the extracted field are cleared in the destination register. The `FDEP` instruction bits to the left and to the right of the deposited field are cleared in the destination register. Therefore programs can use the (SE) option, which sign extends the left bits, or programs can use a logical OR instruction with the source register which does not clear the bits across the shifted field.

Bit Stream Manipulation Instructions

The bit stream manipulation operations, in conjunction with the bit FIFO write pointer (`BFFWRP`) instruction, implement a bit FIFO used for modifying the bits in a contiguous bit stream.

NOTE: For meaningful results, only use SISD mode to execute all bit FIFO instructions.

The shifter supports bit stream manipulation to access the bit FIFO as follows:

- The `BITDEP` instruction deposits bit field from an input stream into the bit FIFO
- The `BITEXT` instruction extracts bit field from the bit FIFO into an output stream

The bit FIFO consists of a 64-bit register internal to the shifter and an associated write pointer register which keeps track of the number of valid bits in the FIFO. When the bit FIFO is empty, the write pointer is 0, when the FIFO is full, the write pointer is 64. The bit FIFO register and write pointer can be accessed only through the `BITDEP` and `BITEXT` instructions. For more information, see ***Shifter/Shift Immediate Computations*** in the Computation Types chapter.

The ***Example of Header Extraction*** and ***Header Creation*** examples demonstrate the `BITDEP` instruction where 32-bit words are appended to the bit FIFO whenever the total number of bits falls below 32. A variable number of bits are read.

Example of Header Extraction

```
I13 = buffer_base;
M13 = 1;
BFFWRP = 0x0;                    /* initialize Bit FIFO */
R10 = pm(I13,M13);
If NOT SF BITDEP R10 by 32,
R10 = PM(I13,M13);               /* appends R10 to BFF */

R6 = BITEXT (6);                 /* extracts 6 bits from head of BFF
and left-shifts BFF by that amount */
DM(Var_1) = R6;
If NOT SF BITDEP R10 by 32, R10 = PM(I13,M13);
R6 = BITEXT(3);                  /* extracts 3 bits */
DM(Var_2) = R6;
```

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

Header Creation

```
I13 = buffer_base;
M13 = 1;
BFFWRP=0x0;
R10 = dm(_var1);                   /* get the variable */
BITDEP R10 by 6;                   /* append it to BFF */
If SF R10 = BITEXT(32),
pm(I13,M13) = R10;                 /* if the balance > 32, transfer a word */
R10 = dm(Var_1);
BITDEP R10 by 3;
If SF R10 = BITEXT(32), pm(I13,M13) = R10;
```

The bit deposits are in variable quantities. However, the extract is always in 32-bits whenever the total number of bits in the bit FIFO increases beyond 32.

Interrupts Using Bit FIFO Instructions

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

Storing and Restoring Bit FIFO State

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

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

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

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

Floating-Point Data Pack and Unpack Instructions

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

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

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

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

Arithmetic Status

Shifter operations update four status flags in the processing element's arithmetic status registers (`REGF_ASTATX` or `REGF_ASTATY`) where a 1 indicates the condition. The bits that indicate shifter status for the most recent ALU operation are as follows.

- Shifter overflow of bits to left of MSB, (`SV`)
- Shifter result zero, (`SZ`)

- Shifter input sign for exponent extract only, (SS)
- Shifter bit FIFO status (SF)

Note that the shifter does not generate an exception handle.

Bit FIFO Status

The bit FIFO contains a status flag (shifter FIFO, SF) which reflects the current value of the write pointer - SF is set when the write pointer is greater than or equal to 32, it is cleared otherwise. Another status flag SV, indicates the exception condition such as overflow or underflow.

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

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

Shifter Instruction Summary

Tables ***Shifter Instruction Summary*** and ***Shifter Bit FIFO Instruction Summary*** list the shifter instructions and shows how they relate to the flags in the ASTATX or ASTATY registers. For more information on assembly language syntax, see the Instruction Set Types chapter and the Computation Types chapter. In these tables, note the meaning of the following symbols:

- The Rn, Rx, Ry operands indicate any register file location; bit fields used depend on instruction
- The Fn, Fx operands indicate any register file location; floating-point word
- The * symbol indicates that the flag may be set or cleared, depending on data
- In SIMD mode all instruction uses the complement data registers, immediate data are valid for both units

Table 3-11: Shifter Instruction Summary

Instruction	ASTATx, ASTATy Flags		
	SZ	SV	SS
RN = lshift RX by (RY | DATA8);	*	*	0
RN = RN or lshift RX by (RY | DATA8);	*	*	0
RN = ashift RX by (RY | DATA8);	*	*	0
RN = RN or ashift RX by (RY | DATA8);	*	*	0
RN = rot RX by (RY | DATA);	*	0	0
RN = bclr RX by (RY | DATA8);	*	*	0
RN = bset RX by (RY | DATA8);	*	*	0
RN = btgl RX by (RY | DATA8);	*	*	0

Table 3-11: Shifter Instruction Summary (Continued)

Instruction	ASTATx, ASTATy Flags		
	SZ	SV	SS
btst RX by (RY | DATA8);	*	*	0
RN = fdep RX by (RY | BIT6:LEN6);	*	*	0
RN = RN or fdep RX by (RY | BIT6:LEN6);	*	*	0
RN = fdep RX by (RY | BIT6:LEN6) (se);	*	*	0
RN = RN or fdep RX by (RY | BIT6:LEN6) (se);	*	*	0
RN = fext RX by (RY | BIT6:LEN6);	*	*	0
RN = fext RX by (RY | BIT6:LEN6) (se);	*	*	0
RN = exp RX;	*	0	*
RN = exp RX (ex);	*	0	*
RN = leftz RX;	*	*	0
RN = lefto RX;	*	*	0
RN = fpack FX;	0	*	0
FN = funpack RX;	0	0	0

The SHARC+ cores support the instructions in the ***Shifter Instruction Summary*** table. Additionally these processors support the shifter bit FIFO instructions shown in the ***Shifter Bit FIFO Instruction Summary*** table.

NOTE: SIMD mode must be disabled during bit FIFO operations.

Table 3-12: Shifter Bit FIFO Instruction Summary

Instruction	ASTATx, ASTATy Flags			
	SZ	SV	SS	SF
bitdep RX by (RY | BITLEN12);	0	*	0	*
RN = bitext (RX | BITLEN12) (nu);	*	*	0	*
bffwrp = (RN | DATA7);	0	*	0	*
RN = bffwrp;	0	0	0	*

Multifunction Computations

The processor core supports multiple parallel (multifunction) computations by using the parallel data paths within its computational units. These instructions complete in a single cycle (except fixed-point multiply which is a two cycle compute), and they combine parallel operation of the multiplier and the ALU or they perform dual ALU functions. The multiple operations work as if they were in corresponding single function computations. Multifunction computations also handle flags in the same way as the single function computations, except that in the dual add/subtract computation, the ALU flags from the two operations are ORed together.

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

Software Pipelining for Multifunction Instructions

Multifunction instructions are parallel operations of both the ALU and multiplier units where each unit has new data available after 1 cycle. However, for floating-point MAC operations, the processor core needs to emulate the MAC instruction with a multifunction instruction. Results from the 32-bit floating-point multiplier unit are available in 2 cycles for the ALU unit. Coding these instructions requires interleaved software pipelining to avoid the computation stall as shown below.

```
F8=0;                           /* clear acc0 */
F9=0;                      /* clear acc1
F12=F3*F7;                      /* first MUL */
lcntr=N/2, do MAC until lce;
F12=F3*F7, F8=F8+F12;           /* first ALU, loop body */
MAC:     F13=F3*F7, F9=F9+F13;  /* first ALU, loop body */
         F8=F8+F12;             /* last ALU */
         F10 = F8+F9;           /* add both MACs */
```

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

Multifunction and Data Move

Another type of multifunction operation combines transfers between the results and data registers and transfers between memory and data registers. These parallel operations complete in a single cycle. For example, the core can perform the following MAC and parallel read of data memory. However if data dependency exists, software pipeline coding is required as shown in the ***MAC and Parallel Read With Software Pipeline Coding*** example.

MAC and Parallel Read With Software Pipeline Coding

```
MRF=0, R5 = DM(I1,M2), R6 = PM(I9,M9);              /* first data */
lcntr=N-1, do (pc,loopend) until lce;
loopend: MRF = MRF-R5*R6, R5 = DM(I1,M2), R6 = PM(I9,M9);    /* loop body */
MRF = MRF-R5*R6;                                    /* last MAC*/
```

Multifunction Input Operand Constraints

Each of the four input operands for multifunction computations are constrained to a different set of four register file locations, as shown in the ***Permitted Input Registers for Multifunction Computations*** figure. For example, the X input to the ALU must be `R8`, `R9`, `R10`, or `R11`. In all other compute operations, the input operands can be any register file location.

The multiport data register file can normally be read from and written to without restriction. However, in multifunction instructions, the ALU and multiplier input are restricted to particular sets of registers while the outputs are unrestricted.

For any instruction with multiple operations executing in parallel, the destination registers should not be the same.

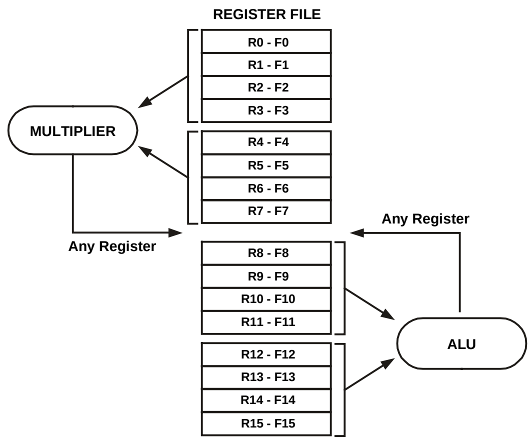

Figure 3-9: Permitted Input Registers for Multifunction Computations

Multifunction Input Modifier Constraints

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

Multifunction Instruction Summary

The processors support the following multifunction instructions.

- Fixed-Point ALU (dual Add and Subtract)
- Floating-Point ALU (dual Add and Subtract)
- Fixed-Point Multiplier and ALU
- Floating Point Multiplier and ALU (dual Add and Subtract)
- Floating-Point Multiplier and ALU
- Fixed-Point Multiplier and ALU (dual Add and Subtract)

For more information see the Computation Types chapter. Note that these computations can be combined with dual data move (type 1 instruction) or single data move with conditions (Group I instruction set types). For more detail refer to the Instruction Set Types chapter.

64-bit Instruction Overview

The SHARC+ core supports 64-bit instruction set, based on ALU, Multiplier and Multifunction instructions. Additional information provided about number of execution cycles consumed by the instructions and the number of unconditional stalls that these instructions impose.

Table 3-13: 64-bit Floating-Point Instruction set for SHARC+ Core

Syntax	No. of Execution Cycles	No. of Stalls (Unconditional)	Basic Function
ALU			
Fm:n = Fx:y + Fz:w;	7	5	Addition
Fm:n = Fx:y - Fz:w;	7	5	Subtraction
COMP(Fx:y , Fz:w);	7	5	Compares the operands and sets flags.
Fm:n = Fx:y;	2	0	Complements the sign bit.
Fm:n = ABS Fx:y;	2	0	Returns the absolute value of the operand.
Fm:n = PASS Fx:y;	2	0	Passes operand in Fx:y through the ALU, to the 64-bit floating point registers Fm:n.
Rn = FIX Fx:y;	4	2	Converts the operand in Fx:y to a twos-complement 32-bit fixed-point integer result.
Rn = FIX Fx:y BY Rz;	4	2	Converts the operand in Fx:y to a twos-complement 32-bit fixed-point integer result. Rz is added to the exponent of the operand in Fx:y before the conversion.
Rn = TRUNC Fx:y;	4	2	Converts the operand in Fx:y to a twos-complement 32-bit fixed-point integer result. The trunc operation always truncates towards 0.
Rn = TRUNC Fx:y BY Rz;	4	2	Converts the operand in Fx:y to a twos-complement 32-bit fixed-point integer result. The trunc operation always truncates toward 0. Rz is added to the exponent of the operand in Fx:y before the conversion.
Fm:n = FLOAT Rx;	2	0	Converts the fixed-point operand in Rx to a floating-point result.
Fm:n = FLOAT Rx BY Ry;	4	2	Converts the fixed-point operand in Rx to a floating-point result. Ry is added to the exponent of the floating-point result.
Fm:n = CVT Fx;	2	0	Converts the 32/40-bit floating-point operand to 64-bit floating point format.
Fn = CVT Fx:y;	4	2	Converts the 64-bit floating-point operand to single precision floating point format.
Fm:n = SCALB Fx:y BY Rz;	2	0	Scales the exponent of the floating-point operand in Fx:y by adding to it the fixed-point twos complement integer in Rz.
Multiplier (uses multiplier result register for temp processing)			
Fm:n = Fx:y * Fz:w;	7	5	Multiplication of two 64-bit operands.
Fm:n = Fx:y * Fz;	7	5	64-bit operand Fx:y multiplied with single precision operand Fz.
Fm:n = Fx * Fy;	7	5	32/40-bit operand Fx multiplied with single precision operand Fy and produces a double precision result.
Multifunction (uses multiplier result register for temp processing)			

Table 3-13: 64-bit Floating-Point Instruction set for SHARC+ Core (Continued)

Syntax	No. of Execution Cycles	No. of Stalls (Unconditional)	Basic Function
Fm:n = Fx:y * Fz:w, Fa:b = Fp:q + Fr:s;	7	5	Multiply and Add in parallel
Fm:n = Fx:y * Fz:w, Fa:b = Fp:q - Fr:s;	7	5	Multiply and Subtract in parallel

WARNING: 64-bit multiplier and multifunction instructions use the multiplier result register during execution. If the multiplier result register contains valid data which may be required by the application, the program must save the data from multiplier result register before executing this instruction (multiplier result register = REGF_MRF or REGF_MRB register depending on the section of the REGF_MODE1.SRCU bit).

64-bit Data Register Coding

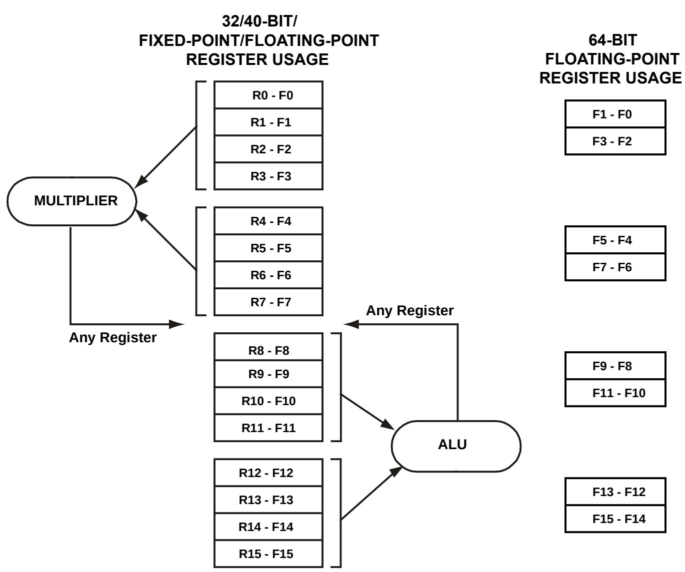

Figure 3-10: Permitted Input Registers for Multifunction Computations

The 64-bit floating-point registers are denoted as "Fx:y". Neighboring data registers are used to construct 64-bit data registers for 64-bit operations.

For example, in the F5:4 = F1:0 + F3:2 operation, the {R1, R0} register pair constitute F1:0; the {R3, R2} register pair constitute F3:2; and the result loaded into the {R5, R4} register pair, for example F5:4.

The first index "x" of "Fx:y" has to be an odd register index and the second index "y" should be its neighboring register with an even index.

NOTE: F4:5, F2:0, F10:7, F7:10 are examples of illegal DP registers.

The complementary 64-bit registers of Processing Element Y (PEY) are named "SFx:y". For example, SF1:0 represent the register pair {S1, S0}.

The following figure shows how the registers R1 and R0 constitute a 64-bit register F1:0.

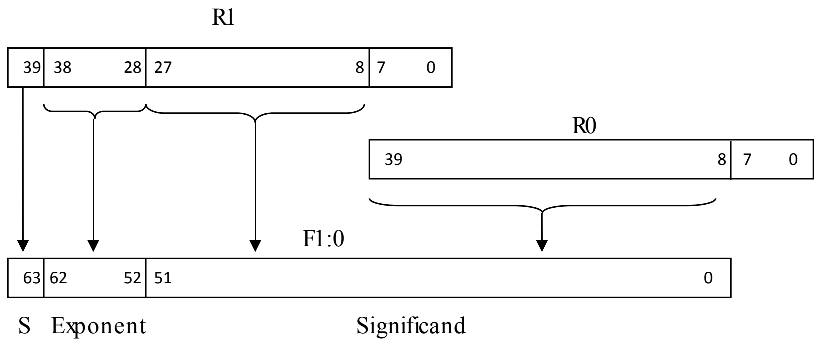

Figure 3-11: 64-bit Register Construction

64-bit Floating-Point Computation Data Hazards

The 64-bit instructions require the registers to be updated in the register-file (RF) before performing the computations. The 64-bit operations cannot be performed on forwarded results of previous compute instructions. There are several data-hazard conditions, which require stalls.

This section explains all the data hazard conditions that are explicitly related to 64-bit floating-point instructions. Such data-hazard conditions are data dependencies from one COMPUTE instruction to the next COMPUTE instructions, when one of them is a 64-bit Compute instruction.

All the other RF related data hazard conditions that affect 32/40-bit floating-point instructions are applicable to 64-bit instructions as well.

The backward and forward data dependencies are explained through instruction pipeline illustrations for each case.

Case A - 64-bit Instruction SRC Operands are DST Operands Of Previous Compute Instructions

This case describes data hazard (instruction pipeline stall) issues that occur when a 64-bit floating-point instruction uses source (SRC) operands that are destination operands (DST) from the previously occuring two compute instructions.

```
< Instruction N-2 ; >
< Instruction N-1 ; >
F5:4 = F3:2 + F1:0; // Instruction N
```

In the Add, Subtract or Compare instructions above, the upper halves of the source 64-bit instruction's register F3:2 and F1:0 (for example, R3 and R1 respectively) are read in the first execution cycle and lower halves (R2 and R0) are read in the second execution cycle. The applicable stalls are listed below in the descending order of priority.

- If the instruction N1 updates either R3 or R1, the instruction N is stalled for 2 cycles.
- If the instruction N1 updates either R2 or R0, the instruction N is stalled for 1 cycle.
- If the instruction N2 updates either R3 or R1, the instruction N is stalled for 1 cycle. This stall is not visible if the instruction N1 also imposes one or more stalls.
- If the instruction N2 updates either R2 or R0, the instruction N is not stalled.

```
< Instruction N-2 ; >
< Instruction N-1 ; >
F5:4 = F3:2 * F1:0 ; // Instruction N
```

In the Fm:n = Fx:y * Fz:w instruction above, the data registers are read in the following sequence.

- Execution Cycle-1: R2, R0
- Execution Cycle-2: R3, R0
- Execution Cycle-3: R2, R1
- Execution Cycle-4: R3, R1

The applicable stalls are listed below in the descending order of priority:

- If the instruction N1 updates either R2 or R0, the instruction N is stalled for 2 cycles.
- If the instruction N1 updates R3, the instruction N is stalled for 1 cycle.
- If the instruction N2 updates either R2 or R0, the instruction N is stalled for 1 cycle. This stall is not visible if the instruction N1 also imposes one or more stalls.
- If the instruction N1 updates R1, or if the instruction N-2 updates either R3 or R1, the instruction N does not stall.

```
< Instruction N-2 ; >
< Instruction N-1 ; >
F5:4 = F3:2 * F0 ; // Instruction N
```

In the Fm:n = Fx:y * Fz instruction above, the data registers are read in the following sequence.

- Execution Cycle-1: R2, R0
- Execution Cycle-2: R3, R0

The applicable stalls are listed below in the descending order of priority.

- If the instruction N1 updates either R2 or R0, the instruction N is stalled for 2 cycles.
- If the instruction N1 updates R3, the instruction N is stalled for 1 cycle.
- If the instruction N2 updates either R2 or R0, the instruction N is stalled for 1 cycle. This stall is not visible if the instruction N1 also imposes one or more stalls.
- If the instruction N2 updates R3, the instruction N is not stalled.

```
< Instruction N-2 ; >
< Instruction N-1 ; >
F5:4 = F2 * F0 ; // Instruction N
```

In the Fm:n = Fx * Fy instruction above, the data registers are read in the following sequence.

- Execution Cycle-1: R2, R0

The stall conditions are listed below in the descending order of priority:

- If the instruction N1 updates either R2 or R0, the instruction N is stalled for 2 cycles.
- If the instruction N2 updates either R2 or R0, the instruction N is stalled for 1 cycle. This stall is not visible if the instruction N1 also imposes one or more stalls.

```
< Instruction N-2 ; >
< Instruction N-1 ; >
 F5:4 =SCALB F3:2 By R0 ; // Instruction N
```

In all the other 64-bit instructions (SCALB is shown above), all the involved source registers are read in the first execution cycle where the following occur.

- If the instruction N1 updates any of the source registers, the instruction N is stalled for 2 cycles.
- If the instruction N2 updates any of the source registers, the instruction N is stalled for 1 cycle. This stall is not visible if the instruction N1 also imposes one or more stalls.

```
Ry = destination of any compute instruction ; //Instruction
N-2, Rx = Ry ; //Instruction N-1
64-bit instruction that uses Rx in first execution cycle; // Instruction N
```

For any 64-bit instruction (N), if the source registers, which are being used in first execution cycle of the 64-bit instruction, are updated by *Dreg*-to-*Dreg* transfer in previous instruction (N1); and the source register of the *Dreg*-to-*Dreg* transfer instruction (N1) is updated in its previous instruction (N2) which is a compute instruction, then:

- The instruction N is stalled for 1-cycle.

Case B - 64-bit Instruction SRC Operands are DST Operands of Previous Cond Register Load

This case describes data hazard (instruction pipeline stall) issues that occur when a 64-bit floating-point instruction uses source (SRC) operands that are destination operands (DST) from a previously occuring conditional register load instruction.

```
If eq R0 = dm(I0,M0); // N-1
F5:4 = F3:2 * F1:0;   // N
```

In the DM, PM, Immediate, or Ureg to Ureg Load or Swap instruction above, if the N1 instruction is a conditional register update instruction and if the N1 instruction updates one or all of the source operands of the instruction N (which is a 64-bit instruction), the following occur.

- Instruction N is stalled for 1 cycle, if the source operand is read in first execution-cycle of the 64-bit instruction.
- Instruction N is not stalled, if the source operand is read in the second or later execution-cycles.

Case C - 64-bit Instruction DST Operand acts as SRC Operands of the Next non-DP Compute Instruction

This case describes data hazard (instruction pipeline stall) issues that occur when a 64-bit floating-point instruction uses destination (DST) operands that are source operands (SRC) for the next occuring non-64-bit compute instruction.

This case applies only if the N instruction is a non-64-bit instruction and if instruction N1 is a 64-bit instruction. If the instruction N is also a 64-bit instruction, this case is same as Case A. (See Case A - 64-bit Instruction SRC Operands are DST Operands Of Previous Compute Instructions.)

```
(1) F5:4 = F3:2 * F1:0 ; // N-1
    R11 = R5 + R4;       //N
(2) F5:4 = F3:2 + F1:0 ; // N-1
    R11 = R5 + R4;       //N
(3) F5 = CVT F3:2 ;      // N-1
    R11 = R5 + R4;       //N
(4) F5 = CVT F3:2 ;      // N-1
    R0 = ASTATX ;        //N
```

When any destination register of a 64-bit instruction (N-1) is a source operand of the next non-64-bit instruction, the instruction N is stalled for 1 cycle. This stall occurs in addition to the stalls imposed by 64-bit instruction N-1. The example code demonstrates these stalls as follows:

- In example line (1), instruction N1 (MULTIPLY) inherently imposes 5-stalls on instruction N. There is 1 additional stall on instruction N, because the source operands of instruction N are the destination operands of instruction N-1. Hence, instruction N is stalled for 6 cycles.
- Similarly, in example line (2), instruction N is stalled for 6-cycles.
- However, in example line (3), instruction N1 inherently imposes 2-stalls on instruction N. There is 1 additional stall on instruction N, because of dependency. Hence, instruction N is stalled for 3-cycles.
- In example line (4), the instruction N1 inherently imposes 2-stalls on instruction N. There is 1 additional stall on instruction N, because of dependency on the status flags. Instruction N is then stalled for 3-cycles.

Combined Data Hazards (Combinations of Cases A, B, C)

In all the described 64-bit data hazard cases (A, B, C), if multiple data hazard conditions arise simultaneously, the number of stalls imposed is the maximum of the number of stalls imposed by each condition.

Example (1) : Case A Combination

In example (1), instruction N2 updates R1, which can stall instruction N for 1-cycle. Instruction N1 updates R3, which can stall instruction N for 2-cycles. In this case, instruction N is stalled for 2-cycles.

```
R1 = R12 + R13; // Instruction N-2
R3 = R10 + R11; // Instruction N-1
F5:4 = F3:2 + F1:0 ; // Instruction N
```

Example (2) : Case A Combination

In example (2), instruction N2 updates R1, which can stall instruction N for 1-cycle. And, instruction N1 updates R0, which can also stall instruction N for 1-cycle. In this case, instruction N is stalled for 1-cycle.

```
R1 = R12 + R13; // Instruction N-2
R0 = R10 + R11; // Instruction N-1
F5:4 = F3:2 + F1:0 ; // Instruction N
```

Example (3) : Case A-C Combination

In example (3), instruction N2 updates R1, which can stall instruction N for 1-cycle. And, instruction N1 is an instruction which unconditionally stalls N for 2-cycles, since it is a 64-bit CVT instruction. In this case, the instruction N is stalled for 2-cycle.

```
R1 = R12 + R13; // Instruction N-2
F15 = CVT F11:10; // Instruction N-1
F5:4 = F3:2 + F1:0 ; // Instruction N
```

64-bit Floating-Point Instruction Execution Cycles

7-cycle Execution of 64-bit Instructions (Add/Subtract/Compare Instructions)

Applies for 64-bit floating-point instructions:

```
Fm:n = Fx:y + Fz:w
Fm:n = Fx:y - Fz:w
COMP ( Fx:y, Fz:w )
```

Table 3-14: 7-cycle Execution of 64-bit Instructions (Add/Subtract/Compare Instructions)

Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
E2									n-2	n-1						n (dp)	n+1
M4/E1								n-2	n-1	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n+1	n+2
M3							n-2	n-1	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n+1	n+2	
M2						n-2	n-1	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n+1	n+2		
M1					n-2	n-1	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n+1	n+2			
D2				n-2	n-1	n (dp)	n+1	n+1	n+1	n+1	n+1	n+1	n+2				
D1			n-2	n-1	n (dp)	n+1	n+2	n+2	n+2	n+2	n+2	n+2					
F4		n-2	n-1	n (dp)	n+1	n+2											
F3	n-2	n-1	n (dp)	n+1	n+2			5-Cycles Stall									
F2	n-1	n (dp)	n+1	n+2													

Table 3-14: 7-cycle Execution of 64-bit Instructions (Add/Subtract/Compare Instructions) (Continued)

Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
F1	n (dp)	n+1	n+2														
NOTE: Rm, Rn, ASTAT, STKY available on cycle 17.																	

7-cycle Execution of 64-bit Instructions (Multiply Instructions)

Applies for 64-bit floating-point instructions:

```
Fm:n = Fx:y * Fz:w
Fm:n = Fx:y * Fz:w
Fm:n = Fx * Fy
```

4-cycle Execution of 64-bit Instructions

Applies for 64-bit floating-point instructions:

```
Rn  = FIX Fx:y
Rn = FIX Fx:y BY Rz
Rn = TRUNC Fx:y
Rn = TRUNC Fx:y BY Rz
Fm:n = FLOAT Rx BY Ry
Fn = CVT Fx:y
```

Table 3-15: 4-cycle Execution of 64-bit Instructions

Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
E2	E2									n-2	n-1			n (dp)	n+1
M4/E1	M4/E1								n-2	n-1	n (dp)	n (dp)	n (dp)	n+1	n+2
M3	M3							n-2	n-1	n (dp)	n (dp)	n (dp)	n+1	n+2	
M2	M2						n-2	n-1	n (dp)	n (dp)	n (dp)	n+1	n+2		
M1	M1					n-2	n-1	n (dp)	n (dp)	n (dp)	n+1	n+2			
D2	D2				n-2	n-1	n (dp)	n+1	n+1	n+1	n+2				
D1	D1			n-2	n-1	n (dp)	n+1	n+2	n+2	n+2					

Table 3-15: 4-cycle Execution of 64-bit Instructions (Continued)

Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
F4	F4		n-2	n-1	n (dp)	n+1	n+2								
F3	F3	n-2	n-1	n (dp)	n+1	n+2			2-Cycles Stall						
F2	F2	n-1	n (dp)	n+1	n+2										
F1	E2	n (dp)	n+1	n+2											
NOTE: Rm, Rn, ASTAT, STKY available on cycle 15.															

2-cycle Execution of 64-bit Instructions with Backward Dependency Stalls

Applies for 64-bit floating-point instructions:

```
Fm:n = - Fx:y
Fm:n = ABS Fx:y
Fm:n = PASS Fx:y
Fm:n = FLOAT Rx
Fm:n = CVT Fx
Fm:n = SCALB Fx:y BY Rz
```

Table 3-16: 2-cycle Execution of 64-bit Instructions with Backward Dependency Stalls

Cycles	1	2	3	4	5	6	7	8	9	10	11	12
E2									n-2	n-1	n (dp)	n+1
M4/E1								n-2	n-1	n (dp)	n+1	n+2
M3							n-2	n-1	n (dp)	n (dp)	n+2	
M2						n-2	n-1	n (dp)	n (dp)	n (dp)		
M1					n-2	n-1	n (dp)	n (dp)	n (dp)	n+1		
D2				n-2	n-1	n (dp)	n+1	n+1	n+1	n+2		
D1			n-2	n-1	n (dp)	n+1	n+2	n+2	n+2			
F4		n-2	n-1	n (dp)	n+1	n+2						
F3	n-2	n-1	n (dp)	n+1	n+2							
F2	n-1	n (dp)	n+1	n+2								
F1	n (dp)	n+1	n+2									
NOTE: Rm, Rn, ASTAT, STKY available on cycle 12.												

1-cycle Execution of 64-bit Instructions with Backward Dependency Stalls

Applies for 64-bit floating-point instructions with backward dependency (CASE-A and CASE-B of Data Hazards):

```
Fx = Fa + Fb; //Instruction n-1
Fm:n = Fx:y + Fz:w ; // Instruction N
```

Table 3-17: 1-cycle Execution of 64-bit Instructions with Backward Dependency Stalls

Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
E2									n-2	n-1	n-1	n-1						n (dp)	n+1
M4/E1								n-2	n-1	n-1	n-1	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n+1	n+2
M3							n-2	n-1	n-1	n-1	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n+1	n+2	
M2						n-2	n-1	n-1	n-1	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n+1	n+2		
M1					n-2	n-1	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n+1	n+2			
D2				n-2	n-1	n (dp)	n+1	n+1	n+1	n+1	n+1	n+1	n+1	n+1	n+2				
D1			n-2	n-1	n (dp)	n+1	n+2	n+2	n+2	n+2	n+2	n+2	n+2	n+2					
F4		n-2	n-1	n (dp)	n+1	n+2													
F3	n-2	n-1	n (dp)	n+1	n+2			2-Cycles Stall		5-Cycles Stall									
F2	n-1	n (dp)	n+1	n+2															
F1	n (dp)	n+1	n+2																
NOTE: Rm, Rn, ASTAT, STKY available on cycle 19.																			

1-cycle Execution of 64-bit Instructions with Forward Dependency Stalls

Applies for 64-bit floating-point instructions with forward dependency (Case C of Data Hazards):

```
Fy = Fa + Fb; //Instruction n-1
Fm:n = Fx:y + Fz:w ; // Instruction N
```

Table 3-18: 1-cycle Execution of 64-bit Instructions with Forward Dependency Stalls

Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
E2									n-2	n-1	n-1						n (dp)	n+1
M4/ E1								n-2	n-1	n-1	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n+1	n+2
M3							n-2	n-1	n-1	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n+1	n+2	
M2						n-2	n-1	n-1	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n+1	n+2		
M1					n-2	n-1	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n+1	n+2			
D2				n-2	n-1	n (dp)	n+1	n+1	n+1	n+1	n+1	n+1	n+1	n+2				
D1			n-2	n-1	n (dp)	n+1	n+2	n+2	n+2	n+2	n+2	n+2	n+2					
F4		n-2	n-1	n (dp)	n+1	n+2												
F3	n-2	n-1	n (dp)	n+1	n+2			1-Cycle Stall	5-Cycles Stall									
F2	n-1	n (dp)	n+1	n+2														
F1	n (dp)	n+1	n+2															
NOTE: Rm, Rn, ASTAT, STKY available on cycle 18.																		

1-cycle Excution of 64-bit Instructions with Forward Dependency Stalls

Applies for 64-bit floating-point instructions with forward dependency (Case C of Data Hazards):

```
Fm:n = Fx:y + Fz:w ;; //Instruction N
Rs = Rm + Rn ; // Instruction n+1
```

Table 3-19: 1-cycle Excution of 64-bit Instructions with Forward Dependency Stalls

Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
E2									n-2	n-1						n (dp)	n (dp)	n+1

Table 3-19: 1-cycle Excution of 64-bit Instructions with Forward Dependency Stalls (Continued)

Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
M4/ E1								n-2	n-1	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n+1	n+2
M3							n-2	n-1	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n+1	n+2	
M2						n-2	n-1	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n+1	n+2		
M1					n-2	n-1	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n (dp)	n+1	n+2			
D2				n-2	n-1	n (dp)	n+1	n+1	n+1	n+1	n+1	n+1	n+1	n+2				
D1			n-2	n-1	n (dp)	n+1	n+2	n+2	n+2	n+2	n+2	n+2	n+2					
F4		n-2	n-1	n (dp)	n+1	n+2												
F3	n-2	n-1	n (dp)	n+1	n+2			5-Cycles Stall					1-Cycle Stall					
F2	n-1	n (dp)	n+1	n+2														
F1	n (dp)	n+1	n+2															
NOTE: Rm, Rn, ASTAT, STKY available on cycle 18.																		

64-bit Floating-Point Register Aliases in Long Word Memory Addressing

The 64-bit registers are made up of neighbor register pairs (for example, pair F1:0 consists of R1 and R0). The DP registers can be loaded using Long-Word memory accesses.

The SHARC+ core assembler supports the following aliases of memory access instructions with 64-bit floating-point registers:

Table 3-20: Alias of 64-bit Registers in Long Word Memory Access Instructions

Register Usage in SHARC+ Core Instruction	Alias to 64-bit Floating-Point Register
Ry = dm()	Fx:y = dm()
Ry = pm()	Fx:y = pm()
dm() = Ry	dm() = Fx:y

Table 3-20: Alias of 64-bit Registers in Long Word Memory Access Instructions (Continued)

Register Usage in SHARC+ Core Instruction	Alias to 64-bit Floating-Point Register
pm() = Ry	pm() = Fx:y

For these aliased instructions, the assembler uses the even address of the neighboring registers (for example, Ry) in place of a 64-bit register (for exmple Fx:y). This aliasing simplifies the long-word accesses because they are aligned to a 64-bit boundary. The even memory locations are mapped to even registers and the odd memory locations are mapped to odd registers. If the above register aliases are used with LW addressing, they can be used to transfer DP data to or from memory. For example:

```
F1:0 = dm(I0,M0) (LW) // would behave same as R0 = dm(I0,M0) (LW).
SF1:0 = dm(I0,M0) (LW) // would behave same as S0 = dm(I0,M0) (LW).
```

Note the following:

- If the LW attribute is not applied, these instructions are same as NW/SW/BW instructions, and the register updated is the even-address register.
- These are only aliases of the existing instructions, not new instructions. All the restrictions and recommendations of using these existing instructions also are applicable to their aliases.

64-bit Floating-Point SIMD Mode

SIMD mode long-word accesses are not supported. In SIMD, the 64-bit registers can be loaded in one of these ways:

- Using two 32-bit normal-word addressing of dual-data in SIMD mode, or
- Using two long-word addressing of dual-data in SISD mode by using the appropriate complementary registers in both accesses.

In both the cases, the alignment of the DP data in memory could be very different. Moreover, there can be many derived methods of updating 64-bit registers for SIMD, from existing methods discussed in the Memory chapter.

64-bit Floating-Point Computation Register Load Priorities

This section describes the register file (RF) bus conflicts that can arise for multifunction 64-bit floating-point (64-bit) operations or 64-bit operations with register load instructions. The SHARC+ core uses the following rules, in cases of RF bus conflicts.

1. Explicit or implicit destinations of DM register-load instructions have the highest priority over all other RF Bus.
2. Explicit or implicit destinations of PM register-load instructions have the second highest priority.
    - For long word addressing on the PM bus, if the explicit or implicit destination of PM bus is same as the explicit or implicit destination of the DM bus, all the writes on the PM bus are blocked.
3. The result of a single ALU operation has the third highest priority.

  - For 64-bit ALU operations, if any of the destination registers (Fa or Fb in case of Fa:b) conflicts with explicit or implicit DM or PM destinations, the result of the DP ALU operation is blocked.
4. The result of multiplier (64-bit or non-64-bit) operations has the fourth highest priority.
  - For 64-bit multiply operations, if any of the destination registers (Fm or Fn in case of Fm:n) conflicts with the explicit/implicit DM or PM destinations or any of the DP ALU destinations, the result of a 64-bit multiply is blocked.
5. The result of shifter and the result of subtract operations for dual-add-sub instructions have the least priority

Note:

- In all of these cases, only the writes to RF registers are blocked by higher priority buses, but the status registers reflect the status of all operations that have occurred.
- For multiplication instructions, even if the results are not updated either fully or partially because of RF bus conflicts, the MR registers are affected because of the execution of multiplication instructions.

Operating Modes

The `MODE1` register controls the operating mode of the processing elements. The ***MODE1 Register Bit Descriptions (RW)*** table in the Registers appendix lists the bits in the `MODE1` register. The bits are described in the following sections.

ALU Saturation

When the `REGF_MODE1.ALUSAT` bit is set (= 1), the ALU is in saturation mode. In this mode, positive fixed-point overflows return the maximum positive fixed-point number (0x7FFF FFFF), and negative overflows return the maximum negative number (0x8000 0000).

When the `REGF_MODE1.ALUSAT` bit is cleared (= 0), fixed-point results that overflow are not saturated, the upper 32 bits of the result are returned unaltered.

Short Word Sign Extension

In short word space, the upper 16-bit word is not accessed. If the `REGF_MODE1.SSE` bit is set (1), the core sign-extends the upper 16 bits. If the bit is cleared (0), the core zeros the upper 16 bits.

Floating-Point Boundary Mode

In the default boundary mode at reset, (`REGF_MODE1.RND32` = 0), a 40-bit extended-precision floating-point mode is supported. This mode has eight additional LSBs of the mantissa and is otherwise compliant with the IEEE 754/854 standards. Results when using this format are more precise than the IEEE single-precision standard will achieve. Extended-precision floating-point data uses a 31-bit mantissa with a 8-bit exponent plus sign a bit.

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

IEEE single-precision floating-point data (REGF_MODE1.RND32 = 1) uses a 23-bit mantissa with an 8-bit exponent plus sign bit. In this case, the computation unit sets the eight LSBs of floating-point inputs to zeros before performing the operation. The mantissa of a result rounds to 23 bits (not including the hidden bit), and the 8 LSBs of the 40-bit result clear to zeros to form a 32-bit number, which is equivalent to the IEEE standard result.

NOTE: In fixed-point to floating-point conversion, the rounding boundary is always 40 bits, even if the REGF_MODE1.RND32 bit is set.

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

- The core does not provide inexact flags. An inexact flag is an exception flag whose bit position is inexact. The inexact exception occurs if the rounded result of an operation is not identical to the exact (infinitely precise) result. Thus, an inexact exception always occurs when an overflow or an underflow occurs.
- NAN (Not-A-Number) inputs generate an invalid exception and return a quiet NAN (all 1s).
- Denormal operands, using denormalized (or tiny) numbers, flush to zero when input to a computational unit and do not generate an underflow exception. A denormal operand is one of the floating-point operands with an absolute value too small to represent with full precision in the significant. The denormal exception occurs if one or more of the operands is a denormal number. This exception is never regarded as an error.
- The core supports round-to-nearest and round-toward-zero modes, but does not support round to +infinity and round-to-infinity.
- The sign bit of output NAN x NAN is a sign bit as the OR of two input sign bits.

Rounding Mode

The REGF_MODE1.TRUNCATE bit determines the rounding mode for all ALU operations, all floating-point multiplies, and fixed-point multiplies of fractional data. The core supports two rounding modes- round-toward-zero and round-toward-nearest. The rounding modes comply with the IEEE 754 standard and have the following definitions.

- Round-toward-zero (REGF_MODE1.TRUNCATE = 1). If the result before rounding is not exactly representable in the destination format, the rounded result is the number that is nearer to zero. This is equivalent to truncation.
- Round-toward-nearest (REGF_MODE1.TRUNCATE = 0). If the result before rounding is not exactly representable in the destination format, the rounded result is the number that is nearer to the result before rounding. If the result before rounding is exactly halfway between two numbers in the destination format (differing by an LSB), the rounded result is the number that has an LSB equal to zero.

Statistically, rounding up occurs as often as rounding down, so there is no large sample bias. Because the maximum floating-point value is one LSB less than the value that represents infinity, a result that is halfway between the maximum floating-point value and infinity rounds to infinity in this mode.

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

Multiplier Result Register Swap

Each multiplier has a primary or foreground register (REGF_MR0F, REGF_MR2F) and alternate or background results register (REGF_MR0B, REGF_MR2B). The REGF_MODE1.SRCU bit selects which result register receives the result from the multiplier operation, swapping which register is the current MRF or MRB. This swapping facilitates context switching.

Unlike other registers that have alternates, both the MRF and MRB registers are coded into instructions, without regard to the state of the REGF_MODE1 register as shown in the following example.

```
MRB = MRB - R3 * R2 (SSFR);
MRF = MRF + R4 * R12 (UUI);
```

With this arrangement, programs can use the result registers as primary and alternate accumulators, or programs can use these registers as two parallel accumulators. This feature facilitates complex math. The REGF_MODE1 register controls the access to alternate registers. In SIMD mode, swapping also occurs with the PEY unit based registers (REGF_MS0F, REGF_MS2F, and REGF_MS0B, REGF_MS2B).

SIMD Mode

The SHARC+ core contains two sets of computational units and associated register files. As shown in the ***SHARC+ SIMD Core Block Diagram***, these two processing elements (PEx and PEy) support Single Instruction Multiple Data (SIMD) operation.

The REGF_MODE1 register controls the operating mode of the processing elements. The REGF_MODE1.PEYEN bit (bit 21) enables or disables the PEy processing element. When REGF_MODE1.PEYEN is cleared (0), the core operates in SISD mode, using only PEx. When the REGF_MODE1.PEYEN bit is set (1), the core operates in SIMD mode, using both the PEx and PEy processing elements. There is a one cycle delay after REGF_MODE1.PEYEN is set or cleared, before the mode change takes effect.

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

Compute Instructions in SIMD Mode

```
bit set MODE1 BITM_REGF_MODE1_PEYEN;    /* enable SIMD */
R0 = R1 + R2;                          /* explicit ALU instruction */
S0 = S1 + S2;                          /* implicit ALU instruction */

F0 = F1 *  F2;                         /* explicit MUL instruction */
SF0 = SF1 * SF2;                       /* implicit MUL instruction */

MRB = MRB - R3 * R2 (SSFR);            /* explicit MUL instruction */
```

```
MSB = MSB - S3 * S2 (SSFR);       /* implicit MUL instruction */

R5 = LSHIFT R6 by <data8>; /* explicit shift imm instruction */
S5 = LSHIFT S6 by <data8>; /* implicit shift imm instruction */
```

To support SIMD, the core performs these parallel operations:

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

NOTE: Using SIMD mode's parallelism, it is possible to double the performance over similar algorithms running in SISD (ADSP-2106x processor compatible) mode.

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

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

Conditional Computations in SIMD Mode

Conditional computations allows the computation units to make computations conditional in SIMD mode. For more information, see ***Conditional Instruction Execution*** in the Program Sequencer chapter.

Interrupt Mode Mask

On the SHARC+ cores, programs can mask automated individual operating mode bits in the `REGF_MODE1` register when entering into an ISR by setting bits in the `REGF_MMASK` register. This improves interrupt handling performance and helps ensure that interrupt handler code runs with operating modes set consistently.

For the processing units, the short word sign extension (`REGF_MODE1.SSE`) the truncation (`REGF_MODE1.TRUNCATE`) the ALU saturation (`REGF_MODE1.ALUSAT`) the floating-point boundary rounding (`REGF_MODE1.RND32`) and the multiply register swap (`REGF_MODE1.SRCU`) bits can be masked. For more information, see the Program Sequencer chapter.

Arithmetic Exceptions

The following sections describe how the processor core handles arithmetic exceptions. Note that the shifter does not generate interrupts for exception handling. For a complete list of interrupts, see the Interrupt Priority and Vector Table.

NOTE: Interrupt processing starts two cycles after an arithmetic exception occurs because of the one cycle delay between an arithmetic exception and the REGF_STKYX/REGF_STKYY register update

Table 3-21: Arithmetic Exceptions

Interrupt Source	Interrupt Condition	Return Register	Return Instruction	IVT level
PEx/PEy	Fixed-Point ALU/MUL overflow	STKYx/y	RTI	23, FIXI
	Floating-Point ALU/MUL overflow	STKYx/y	RTI	24, FLTOI
	Floating-Point ALU/MUL underflow	STKYx/y	RTI	25, FLTUI
	Floating-Point ALU/MUL invalid	STKYx/y	RTI	26, FLTII

Arithmetic Exception Acknowledge

After an exception has been detected the ISR routine needs to clear the flag bit as shown in the following example.

```
ISR_ALU_Exception:
bit tst STKYx AVS; /* check condition */
IF TF jump ALU_Float_Overflow;
bit tst STKYx AOS; /* check condition */
IF TF jump ALU_Fixed_Overflow;
ALU_Fixed_Overflow:
bit clr STKYx AOS; /* clear sticky bit */
rti;
ALU_Float_Overflow:
bit clr STKYx AVS; /* clear sticky bit */
rti;
```

NOTE: Interrupt service routines for arithmetic interrupts (FIXI, FLTOI, FLTUI and FLTII) must clear the appropriate STKYx or STKYy bits to clear the interrupt. If the bits are not cleared, the interrupt is still active after the return from interrupt (RTI).

SIMD Computation Exceptions

If one of the four fixed-point or floating-point exceptions is enabled, an exception condition on one or both processingelements generates an exception interrupt. Interrupt service routines (ISRs) must determine which of the processingelements encountered the exception. Returning from a floating-point interrupt does not automatically clear the STKY state. Program code must clear the sticky bits in both processing element's sticky status (REGF_STKYX and REGF_STKYY) registers as part of the exception service routine. For more information, see ***Interrupt Branch Mode*** in the Program Sequencer chapter.

4 Program Sequencer

The program sequencer is responsible for the control flow of programs and data within the processor. The sequencer controls nonsequential program flows such as jumps, calls, and loop instructions. The sequencer is closely connected to the system interface, DAGs, and a special type of cache, called conflict instruction cache.

NOTE: The SHARC+ core provides instruction and data caches, which are not available on previous SHARC processors. The instruction and data caches reduce average latency of instruction and data accesses from system L2 memory or from external memories. By comparison, the conflict instruction cache reduces latency of instruction access due only to instruction accesses conflicting with a data access over PM bus. For more information, see Instruction-Conflict Cache Control.

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

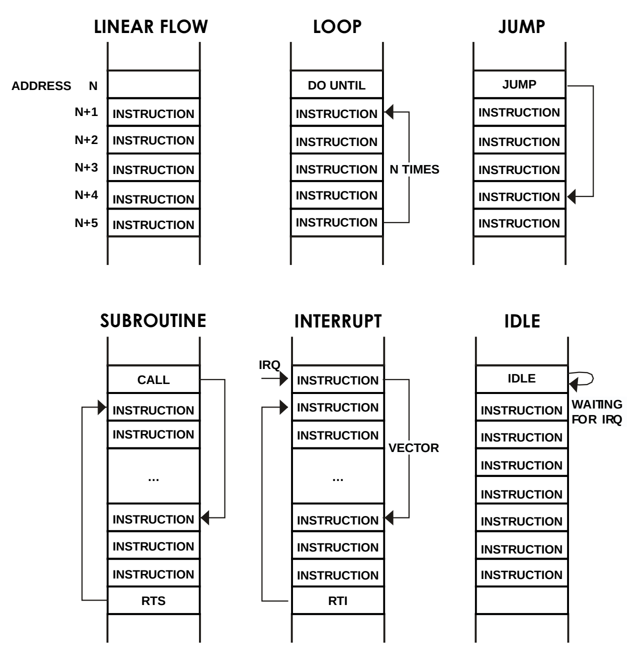

Figure 4-1: Program Flow

Features

The sequencer controls the following operations.

- ***Loops.*** One sequence of instructions executes several times with zero overhead or significantly reduced pipeline overhead (when compared to software loop).
- ***Subroutines.*** The processor temporarily breaks sequential flow to execute instructions from another part of program memory.
- ***Jumps.*** Program flow is permanently transferred to another part of program memory.
- ***Interrupts.*** Subroutines in which a runtime event (not an instruction) triggers the execution of the routine.
- ***Idle.*** An instruction that causes the processor to cease operations and hold its current state until an interrupt occurs. Then, the processor services the interrupt and continues normal execution.
- ***ISA or VISA*** instruction fetches. The fetch address is interpreted as an ISA (NW address, traditional) or VISA instruction (SW address) this allows fast switching between both instruction types.
- ***Direct Addressing.*** Provides data address specified as absolute value in instruction.

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

- Increments the fetch address
- Maintains stacks
- Evaluates conditions
- Decrements the loop counter
- Calculates new addresses
- Maintains a special instruction cache known as instruction-conflict cache
- Predicts branches using the branch target buffer
- Interrupt control

To accomplish these tasks, the sequencer uses the blocks shown in the ***Sequencer Control Diagram*** figure. The sequencer's address multiplexer selects the value of the next fetch address from several possible sources. The fetch address enters the instruction pipeline. The fetch address is the 24-bit address of the instruction currently being fetched, decoded, and executed. The program counter, coupled with the program counter stack, stores return addresses and top-of-loop addresses. All addresses generated by the sequencer are 24-bit program memory instruction addresses.

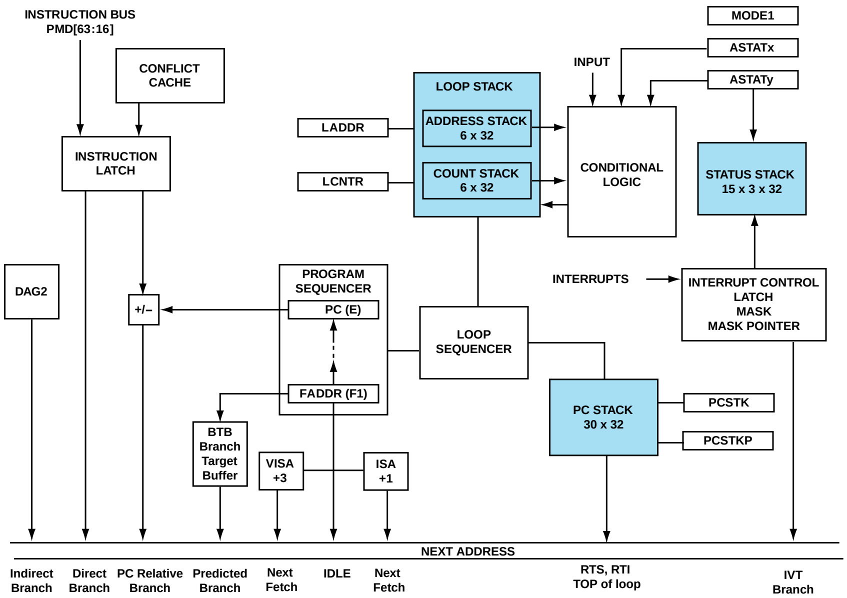

Figure 4-2: Sequencer Control Diagram

Functional Description

The sequencer uses the blocks shown in the ***Sequencer Control Diagram*** figure to execute instructions. The sequencer's address multiplexer selects the value of the next fetch address from several possible sources. These registers contain the 24-bit addresses of the instructions currently being fetched, decoded, and executed.

Instruction Pipeline

The program sequencer determines the next instruction address by examining both the current instruction being executed and the current state of the processor. The branch predictor unit examines each fetch address to determine whether it is a branch instruction. If the unit detects a branch instruction, the unit provides an address of the likely next instruction. If no conditions require otherwise, the processor fetches and executes instructions from memory in sequential order.

To achieve a high execution rate while maintaining a simple programming mode, the processor employs an 11 stage interlocked pipeline, shown in the ***Instruction Pipeline Processing Stages*** table, to process instructions and simplify programming models. All possible hazards are controlled by hardware.

The legacy Instruction Set Architecture (ISA) instructions are addressed using normal word (NW) address space, whereas Variable Instruction Set Architecture (VISA) instructions are addressed using short word (SW) address space. Switching between traditional ISA and VISA instruction spaces occurs automatically when branches (`JUMP`/`CALL` or interrupts) take the execution from ISA address space to VISA address space or vice versa; no changes to mode registers are required.

NOTE: The processor always emerges from reset in ISA mode, so the interrupt vector table must always reside in ISA address space.

The processor controls the fetch address, decode address, and program counter (`REGF_FADDR`, `REGF_DADDR`, and `REGF_PC`) registers which store the Fetch1, decode, and execution phase addresses of the pipeline.

Table 4-1: Instruction Pipeline Processing Stages

Stage	Stage	ISA
Fetch1	F1	In this stage, the appropriate instruction address is chosen from various sources and driven out to memory. The instruction address is matched with the instruction-conflict cache to generate a condition for cache miss/hit in case the PM bus is busy for a data access. The next NW address is auto incremented by one. **NOTE:** *VISA Extension:* Next SW address is auto incremented by three for every 48-bit fetch
Fetch2	F2	Memory data and instruction/conflict cache access stages.
Fetch3	F3	
Fetch4	F4	This stage is the data phase of the instruction fetch-memory access wherein the data address generator (DAG) performs some amount of pre-decode. Based on a hit or miss in the conflict cache, the instruction is read from conflict cache/driven from the memory instruction data bus. **NOTE:** *VISA Extension:* Stores 3 x 16-bit instruction data into the IAB buffer and presents 1 instruction/cycle to the decoder
Decode1	D1	The instruction is decoded and various conditions that control instruction execution are generated. The main active units in this stage are the DAGs, which generate the addresses for various types of functions like data accesses (load/store) and indirect branches. DAG pre-modify (M+I) operation is performed. For a cache miss, instruction data read from memory are loaded into the instruction-conflict cache. **NOTE:** *VISA Extension:* Decode Visa instruction; store its length information in short words.
Decode2	D2	
Memory access 1 (address)	M1	The addresses generated by the DAGs in the previous stage are driven to the memory through memory interface logic. The addresses for the branch operation are made available to the fetch unit. The target address predicted by BP/BTB is validated for unconditional branch instructions. For instruction branches (Call/Jump) the address is forwarded to the Fetch1 stage.
Memory access 2	M2	
Memory access 3	M3	
Memory access 4 (data/execute 1)	M4/E1	Memory access returns data for load operation. All the fixed point ALU and shifter instructions complete operations. First half of the floating point operations and multiplication operations complete.
Execute2	E2	Second half of the two cycle compute operations complete. Results of computations, memory read operations are written back to destination registers. For conditional branch instructions, predictions made by BP/BTB are validated. **NOTE:** *VISA Extension:* Executing VISA instructions the PC value is incremented by 1, 2, or 3; depending on length information from the Instruction decode.

VISA Instruction Alignment Buffer (IAB)

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

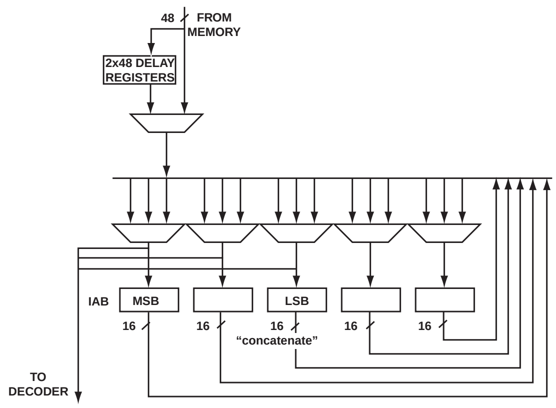

Figure 4-3: Instruction Alignment Buffer

A decode of the instruction indicates the length of the instruction in unit of short words. At the end of the current decode cycle, the short words that are part of the current instruction are discarded and the remaining bits are shifted left to align at the MSB of IAB. The three fetched short words in the following cycle are concatenated to the existing bits of IAB. The next instruction, therefore, is always available in MSB aligned fashion. Because the fetch operations being processed must complete (even after the sequencer stalls the fetch engine), added instruction storage is provided through two 48-bit delay registers.

Linear Program Flow

In the sequential program flow, when one instruction is being executed, the next ten instructions that follow are being processed in other stages of the instruction pipeline. Sequential program flow usually has a throughput of one instruction per cycle.

The ***ISA/VISA Linear Flow 48-bit Instructions Only*** table illustrates how the instructions starting at address n are processed by the pipeline. While the instruction at address n is being executed, the subsequent instructions from n +1 to n+10 are being processed in the subsequent stages of instruction pipeline from M4 to F1 stages respectively. Note that---when executing ISA code---the instruction addresses are NW addresses.

Table 4-2: ISA Linear Flow 48-bit Instructions Only

cycles	1	2	3	4	5	6	7	8	9	10	11	12
E2											n	n+1
M4										n	n+1	n+2

Table 4-2: ISA Linear Flow 48-bit Instructions Only (Continued)

cycles	1	2	3	4	5	6	7	8	9	10	11	12
M3									n	n+1	n+2	n+3
M2								n	n+1	n+2	n+3	n+4
M1							n	n+1	n+2	n+3	n+4	n+5
D2						n	n+1	n+2	n+3	n+4	n+5	n+6
D1					n	n+1	n+2	n+3	n+4	n+5	n+6	n+7
F4				n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8
F3			n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9
F2		n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10
F1	n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11

When executing VISA instructions, the instruction addresses are SW addresses. The sequencer always fetches 48-bits (3 short words) in each fetch operation. The fetch addresses always increment by 3. But, the PC may increment by 1, 2, or 3 based on the length of the instructions.

NOTE: On memory space boundaries, the instruction fetch does not halt and continues to fetch next address.

Direct Addressing

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

```
R0 = DM(NW_Address);   /* sequencer generated data address */
PM(NW_Address) = R7:   /* sequencer generated data address */
```

as compared to the DAG:

```
R0 = DM(I0,M0);   /* DAG1 generated data address */
PM(I8,M8) = R7:   /* DAG2 generated data address */
```

For more information, see the Data Address Generators chapter.

Illegal System Accesses Conditions

If the SHARC+ core as requester performs a system completer access (to peripherals, memories) via the system crossbar, the requests traverse through the system crossbar as follows.

1. The completer receives the request, grants and forwards it to the system crossbar.
2. The system fabric acknowledges and forwards the grant to the system core requester.

Once the core accepts the grant it executes the next instruction. In ADSP-SC58x product based systems illegal conditions may be caused by:

- access to disabled peripherals

- access to enabled unpopulated peripherals
- access to unavailable/private SMMR addresses
- access to secure completers (handled by the SPU/SMPU)

These violation conditions may lead to halt the entire requester-completer path because the completer does not grant the request (system crossbar and few cycles later the core requester is stalled). To prevent illegal access conditions, use the SMPU instances for exception handling and the system watchdogs for stall recognition. See also Completer Ports Warning.

Variation In Program Flow

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

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

Functional Description

To manage these variations, the processor uses several mechanisms, primarily branch prediction and hardware stacks, which are described in the following sections.

Hardware Stacks

If the programmed flow varies (nonsequential and interrupted), the processor requires hardware or software mechanisms (stacks; see the ***Core Stack Overview*** table) to support changes of the regular program flow. The SHARC core supports three hardware stack types which are implemented outside of the memory space and are used and accessed for any nonsequential process. The stack types are:

- Program count stack Used to store the return address (call, IVT branch, do until).
- Status stack Used to store some context of status registers.
- Loop Stack for address and count Used for hardware looping (unnested and nested). This stack is described in Loop Sequencer section.

The SHARC+ core does not have a hardware stack (a memory area dedicated to the sole purpose of stack storage). The DAG architecture allows programmers to implement a software stack, using the DAG instruction types: push (post-modify) and pop (pre-modify).

NOTE: The stacks are fully controlled by hardware. Manipulation of these stacks by using explicit `PUSH`/`POP` instructions and explicit writes to the `REGF_PCSTK`, `REGF_LADDR`, and `REGF_CURLCNTR` registers may affect the correct functioning of the loop.

Table 4-3: Core Stack Overview

Attribute	PC Stack	Loop Address Stack	Loop Count Stack	Status Stack
Stack Size	30 x 32 bits	6 x 32 bits	6 x 32 bits	15 x 3 x 32 bits
Top Entry	Return Address, top of loop address	Loop End Address	Loop iteration count	MODE1 ASTATx/ASTATy
Empty Flag	PCEM	LSEM		SSEM
Full Flag	PCFL	LSOV		SSOV
Stack Pointer	PCSTKP	No		No
Exception IRQ	SOVFI	SOVFI		SOVFI
Automated Access				
Push Condition	CALL, IVT branch DO UNTIL	DO UNTIL		IVT Branch (for all interrupts except EMUI and RSTI)
Pop Condition	RTS, RTI	CURLCNTR = 1 or COND = true		RTI (for all interrupts except EMUI and RSTI)
Manual Access				
Register Access	PCSTK	LADDR	CURLCNTR	MODE1STK
Explicit Push	Push PCSTK	Push Loop		Push STS
Explicit Pop	Pop PCSTK	Pop Loop		Pop STS

PC Stack Access

The sequencer includes a program counter (PC) stack pointer, as shown in the ***Sequencer Control Diagram*** figure. At the start of a subroutine or loop, the sequencer pushes return addresses for subroutines (`CALL` instructions with `RTI`/`RTS`) and top-of-loop addresses for loops (`DO`/`UNTIL` instructions) onto the PC stack. The sequencer pops the PC stack during a return from interrupt (`RTI`), return from subroutine (`RTS`), and a loop termination.

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

> **NOTE:** Compared to ADSP-214xx processors, the PC stack register size on the SHARC+ processor has been enlarged to 32-bits. Additional bits store various other information required for proper instruction sequencing.

PC Stack Status

The PC stack is 30 locations deep. The stack is full when all entries are occupied, is empty when no entries are occupied, and is overflowed if a push occurs when the stack is full.

The following bits in the `REGF_STKYX` registers indicate the PC stack full and empty states.

- ***PC stack full.*** Bit 21 (REGF_STKYX.PCFL) indicates that the PC stack is full (=1) or not full (=0). This bit is not sticky and is cleared by a pop.
- ***PC stack empty.*** Bit 22 (REGF_STKYX.PCEM) indicates that the PC stack is empty (=1) or not empty (=0). This bit is not sticky and is cleared by a push.

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

PC Stack Manipulation

The REGF_PCSTK register contains the top entry on the PC stack. This register is readable and writable by the core. Reading from and writing to the REGF_PCSTK register does not move the PC stack pointer. Only a stack push or pop performed with explicit instructions moves the stack pointer. The REGF_PCSTK register contains the value 0x7FFF FFFF when the PC stack is empty. A write to the REGF_PCSTK register has no effect when the PC stack is empty. The ***Program Counter Stack Register (PCSTK)*** section in the Registers appendix lists the bits in this register.

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

PC Stack Access Priorities

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

Status Stack Access

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

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

For all interrupts except EMUI and RSTI, the sequencer automatically pushes the REGF_ASTATX, REGF_ASTATY, and REGF_MODE1 registers onto the status stack. When the sequencer pushes an entry onto the status stack, the processor uses the MMASK register to clear the corresponding bits in the REGF_MODE1 register. All other bit settings remain the same. See the example in Interrupt Mask Mode.

NOTE: The REGF_MODE1STK register provides access to the REGF_MODE1 data in the top-level entry of the status stack.

The sequencer automatically pops the REGF_ASTATX and REGF_ASTATY registers from the status stack during the return from interrupt instruction (RTI). In one other case, JUMP (CI), the sequencer pops the stack. For more information, see Interrupt (Pseudo) Self-Nesting.

Pushing the REGF_ASTATX, REGF_ASTATY, and REGF_MODE1 registers preserves the status and control bit settings. This allows a service routine to alter these bits with the knowledge that the original settings are automatically restored upon return from the interrupt.

The top of the status stack contains the current values of the REGF_ASTATX, REGF_ASTATY, and REGF_MODE1 registers. Explicit PUSH or POP instructions (not reading and writing these registers) are used to move the status stack pointer.

Status Stack Status

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

- ***Status stack overflow.*** Bit 23 (REGF_STKYX.SSOV) indicates that the status stack is overflowed (=1) or not overflowed (=0). This is a sticky bit.
- ***Status stack empty.*** Bit 24 ()REGF_STKYX.SSEM indicates that the status stack is empty (=1) or not empty (=0). This bit is not sticky, cleared by a push.

Both REGF_ASTATX and REGF_ASTATY register values are pushed/popped regardless of SISD/SIMD mode.

Instruction Driven Branches

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

- A JUMP or a CALL instruction transfers program flow to another memory location. The difference between a JUMP and a CALL is that a CALL automatically pushes the return address (the next sequential address after the CALL instruction) onto the PC stack. This push makes the address available for the CALL instruction's matching return instruction, (RTS) in the subroutine, allowing an easy return from the subroutine.
- A RTS instruction causes the sequencer to fetch the instruction at the return address, which is stored at the top of the PC stack. The two types of return instructions are return from subroutine (RTS) and return from interrupt (RTI). While the RTS instruction only pops the return address off the PC stack, the RTI pops the return address and:

    1. Clears the interrupt's bit in the interrupt latch register (REGF_IRPTL) and the interrupt mask pointer register (REGF_IMASKP).

This action lets another interrupt be latched in the REGF_IRPTL register and the interrupt mask pointer (REGF_IMASKP) register.

2. Pops the status stack

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

- JUMP and CALL instructions can be conditional. The program sequencer can evaluate the status conditions to decide whether or not to execute a branch. If no condition is specified, the branch is always taken.
- JUMP and CALL instructions can be immediate or delayed. Because of the instruction pipeline, an immediate branch incurs a number of lost (overhead) cycles, which is dependent on depth of the pipeline. The 11-deep pipeline in the SHARC+ core core incorporates a branch predictor and a branch target buffer (BP/BTB) to reduce or in some cases, completely eliminate overhead cycles.

  As shown in the Table 4-5 Pipelined Execution Cycles for Immediate Branch (Jump or Call) and Table 4-6 Pipelined Execution Cycles for Immediate Branch (RTI) tables the processor may abort the six instructions after the branch, which are in the Fetch1 through Decode stages, while instructions are fetched from the branched address. Due to presence of BP/BTB in the SHARC+ core, the overhead is 2 cycles in cases involving nondelayed branches. A delayed branch reduces the overhead by two cycles by allowing the two instructions following the branch to propagate through the instruction pipeline and execute, reducing the overhead to zero cycles. For more information, see Delayed Branches (DB).
- JUMP instructions that appear within a loop or within an interrupt service routine have additional options. For information on the loop abort (LA) option, see Functional Description. For information on the clear interrupt (CI) option, see Interrupt (Pseudo) Self-Nesting.

Branch Prediction

The SHARC+ core pipeline contains 11 stages. As the pipeline stages increase, the data hazards may also increase by directly impacting branch operations (mainly conditional branches). To lessen this effect, a hardware based Branch-Predictor (BP) and Branch-Target-Buffer (BTB) are added to the SHARC+ core. The branch predictor is generally used for conditional branches and it determines whether the branch is to be taken or not and provides the branch target address. When the branch is predicted correctly, several stalls are prevented. An incorrect prediction causes the same number of stalls as operation without the branch predictor.

For all branches except hardware loops, RTI and jump (CI), the BP/BTB also provides the branch target address. As it encounters branches, the BP/BTB builds history in the BTB RAM for that instruction, and it uses this history to predict the outcome of that branch when encountering the branch again. The sequencer verifies the prediction for a conditional branch at the final stage of the pipeline. If the prediction is found to be incorrect, then the entire pipeline is flushed and the correct target instruction is fetched. For an unconditional branch, the sequencer verifies the correctness of the target address in the address (M1) stage of pipeline. If the target address is found to be incorrect, then the six stages of the pipeline from the Fetch (F1) to Decode (D2) stages are flushed and the correct target instruction is fetched.

BTB Function

The BP/BTB RAM contains storage for 256 entries organized as 2-way x 128 entries with associated VALID and LRU bits and has a 2-bit saturating counter for each entry. Each fetch address generated by the sequencer is checked for a HIT. When a HIT occurs, the counter value determines the conditional prediction of a branch. The branch is predicted taken when the counter value is 10 or 11, and not taken otherwise.

The counter value is updated when the prediction is validated. For a taken branch, it is incremented, otherwise it is decremented. If a branch instruction was not a HIT in BP/BTB at the final stage of the pipeline, one of the entries is updated with its relevant PC.

The target address and other relevant attributes follow much of the same principles as a traditional instruction or data cache. LRU based replacement policy is followed. The ***Logical Organization of BP/BTB*** figure shows the structure of the BP/BTB. In order to ensure there is only one branch in the fetch stage, the last short word of two branches should not fall in one 48-bit window when using VISA mode.

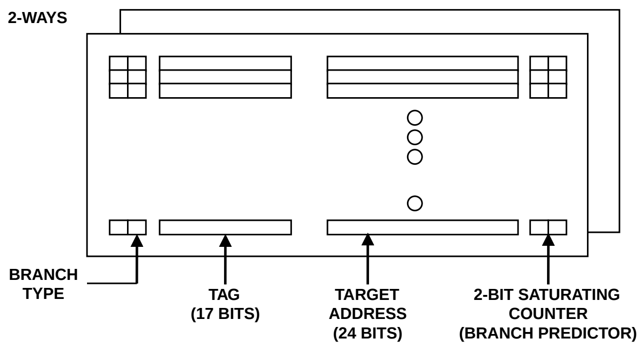

Figure 4-4: Logical Organization of BP / BTB

BTB Features

Disable and Freeze: The entire functionality of the BP/BTB can be disabled by clearing the `SHBTB_CFG.DIS` bit. The contents of the BP/BTB can also be frozen by setting the `SHBTB_CFG.FRZ` bit. When frozen, the BP/BTB continues to make predictions and provide target addresses, but its contents are not changed.

Lock: When the relevant bit is set in the `SHBTB_CFG` register, fetch addresses for all of the branches that fall within a range of addresses and their target addresses are recorded in the BP/BTB and are protected from being overwritten. The range of the addresses are programmed in the `SHBTB_LOCK_START` and `SHBTB_LOCK_END` registers.

Predicting return from subroutine: Return from subroutines (target address is provided from top of PC stack) constitute a large portion of all branches. Return addresses that are predicted based on history are generally incorrect because the same subroutine is called from many places in the code. The BP/BTB attempts to improve the prediction accuracy of this class of branches by taking target addresses from other more relevant sources than the BTB. These features are controlled by setting of relevant bits in the `SHBTB_CFG` register, which are enabled by default. If disabled, the target address is provided by BTB.

BTB Scenarios When Prediction Is Ignored

There are situations when the BP/BTB does not look up the fetch address and/or does not provide a predicted address.

1. If the BTB predicts any branch as taken, the next two fetch addresses are ignored by the BTB for prediction.
2. If a BTB update occurs during the E2 pipeline stage, the instruction in the F1 pipeline stage is ignored by the BTB for prediction.
3. For instructions inside hardware loops, BTB masking occurs in the following situations.
   a. Up to 10 instructions that occur in the pipeline after a do-until instruction are ignored by the BTB for prediction.
   b. Branch instructions that occur in the last three instructions of the loop are ignored by the BTB for prediction.
   c. Branch instructions whose target address falls within the last 10 instructions of a loop may be ignored by the BTB for prediction. But, a JUMP to last 10 instructions of an E2-active loop is not masked from BTB prediction, provided there is no RTS instruction in the pipeline above JUMP.
4. In the pipeline vicinity of stack updates, BTB masking occurs:
   a. RTS instructions that occur while a CALL or another RTS are in the pipeline are ignored by the BTB for prediction.
   b. While a loop stack manipulation instruction is in the pipeline, all instructions are ignored by the BP/BTB for prediction.

When a predictable branch appears at the 1st stage of pipeline, the predicted target address appears in the pipeline after two cycles. These two cycles are to facilitate the execution of branches with delayed slots. For branches without delayed slots, these two cycles are added. The ***Stalls in the Presence of Branch Target Buffer*** table shows the positions of branch and its related target instruction in the pipeline in the presence of the BP/BTB.

Table 4-4: Stalls in the Presence of Branch Target Buffer

Branch	Condition		Target Prediction	Loss of cycles with BTB (Non delayed branch/delayed branch)	Maximum # of loss of cycles without BTB (Non delayed branch/delayed branch)
	Prediction	Actual			
Conditional	Taken	Taken	HIT	2/0	11/9
Conditional	Not Taken	Not Taken	HIT	0	0
Conditional	Taken	Not Taken	MISS	11/11	0
Conditional	Not Taken	Taken	MISS	11/9	11/9
Conditional	Taken	Taken	HIT	6/4	11/9
Unconditional	Always Taken	Always Taken	HIT	2/0	6/4

BTB Registers

The BTB registers include the SHBTB_CFG, SHBTB_LOCK_START, and SHBTB_LOCK_END. Details can be found in the Register Descriptions section

WARNING: After a write operation to the SHBTB_CFG, SHBTB_LOCK_START or SHBTB_LOCK_END registers, there must be at least twelve 48-bit (ISA) instructions, which do not involve any change of flow. Similarly, after a branch there must be at least twelve 48-bit instructions. These twelve instructions should not cross memory boundary.

Restrictions Related to the Branch Predictor

Note the following restrictions related to the branch predictor.

1. After every branch here should be at least 12 48-bit valid instructions in the code. This extra code should not cross a memory boundary
2. In case of VISA encoding, two branches (partially or fully) should not come in any 48-bit window.

Direct Versus Indirect Branches

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

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

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

Indirect branches are `JUMP` or `CALL` instructions that use a dynamic address that comes from the DAG2. Note that this is useful for reconfigurable routines and jump tables. For more information refer to the instruction set types (9a/b and 10a). Two instruction examples that cause an indirect branch are:

```
JUMP (M8, I12); /* Where (M8, I12) are DAG2 registers */
CALL (M9, I13); /* Where (M9, I13) are DAG2 registers */
```

Restrictions for VISA Operation

The following should be noted for VISA operation:

- The program counter (PC) now points to short word address space. The PC increments by one, two or three in each cycle depending on the actual size of an instruction (16-bit, 32-bit, or 48-bit).
- Any source files that use hard-coded numbers (as opposed to labels) for branch offsets in the relative offset field may not function correctly. What used to be N 48-bit instructions could be a different number of VISA instructions.

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

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

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

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

When the instructions take up more than two 16-bit units, the jump could go to an invalid memory location (not to the start of a valid VISA instruction). Good programming practices suggest discouraging such usage of "absolute addressing".

Delayed Branches (DB)

The instruction pipeline influences how the sequencer handles delayed branches (tables ***Pipelined Execution Cycles for Immediate Branch (Jump or Call)*** through ***Pipelined Execution Cycles for Delayed Branch (RTS(db))*** in Branch Listings). For immediate branches in which `JUMP` and `CALL` instructions are not specified as delayed branches `(DB)`, some instruction cycles are lost (`NOP`) as the instruction pipeline empties and refills with instructions from the new branch.

Branch Listings

As shown in the ***Pipelined Execution Cycles for Immediate Branch (Jump or Call)*** and ***Pipelined Execution Cycles for Immediate Branch (RTI)*** tables, the processor aborts the six instructions after the branch, which are present from fetch1 to decode2 stages. For a `CALL` instruction, the address of the instruction after the `CALL` is the return address.

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

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

cycles	1	2	3	4	5	6	7	8	9	10	11	12
E2	n-4	n-3	n-2	n-1	n	n+1	n+2	n+3	n+4	n+5	n+6	j
M4	n-3	n-2	n-1	n	n+1	n+2	n+3	n+4	n+5	n+6	j	j+1
M3	n-2	n-1	n	n+1	n+2	n+3	n+4	n+5	n+6	j	j+1	j+2
M2	n-1	n	n+1	n+2	n+3	n+4	n+5	n+6	j	j+1	j+2	j+3
M1	n	n+1	n+2	n+3	n+4	n+5	n+6	j	j+1	j+2	j+3	
D2	n+1	n+2	n+3	n+4	n+5	n+6	j	j+1	j+2	j+3		

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

cycles	1	2	3	4	5	6	7	8	9	10	11	12
D1	n+2	n+3	n+4	n+5	n+6	j	j+1	j+2	j+3			
F4	n+3	n+4	n+5	n+6	j	j+1	j+2	j+3				
F3	n+4	n+5	n+6	j	j+1	j+2	j+3					
F2	n+5	n+6	j	j+1	j+2	j+3						
F1	n+6	j	j+1	j+2	j+3							
n is the branching instruction and j is the instruction branch address												
cycle 1: n+1 instruction is suppressed												
cycle 2: n+2 instruction is suppressed												
cycle 3: n+3 instruction is suppressed												
cycle 4: n+4 instruction is suppressed												
cycle 5: n+5 instruction is suppressed and for call , n+1 address is pushed on to PC stack												
cycle 6: n+6 instruction is suppressed												

Table 4-6: Pipelined Execution Cycles for Immediate Branch (RTI)

Cycles	1	2	3	4	5	6	7	8	9	10	11	12
E2	n-4	n-3	n-2	n-1	n	n+1	n+2	n+3	n+4	n+5	n+6	r
M4	n-3	n-2	n-1	n	n+1	n+2	n+3	n+4	n+5	n+6	r	r+1
M3	n-2	n-1	n	n+1	n+2	n+3	n+4	n+5	n+6	r	r+1	r+2
M2	n-1	n	n+1	n+2	n+3	n+4	n+5	n+6	r	r+1	r+2	r+3
M1	n	n+1	n+2	n+3	n+4	n+5	n+6	r	r+1	r+2	r+3	
D2	n+1	n+2	n+3	n+4	n+5	n+6	r	r+1	r+2	r+3		
D1	n+2	n+3	n+4	n+5	n+6	r	r+1	r+2	r+3			
F4	n+3	n+4	n+5	n+6	r	r+1	r+2	r+3				
F3	n+4	n+5	n+6	r	r+1	r+2	r+3					
F2	n+5	n+6	r	r+1	r+2	r+3						
F1	n+6	r	r+1	r+2	r+3							
n is the branching instruction and r is the instruction at the return address												
cycle 1: n+1 instruction is suppressed.												
cycle 2: n+2 instruction is suppressed												
cycle 3: n+3 instruction is suppressed												
cycle 4: n+4 instruction is suppressed												
cycle 5: n+5 instruction is suppressed and r address is popped from PC stack												

Table 4-6: Pipelined Execution Cycles for Immediate Branch (RTI) (Continued)

Cycles	1	2	3	4	5	6	7	8	9	10	11	12
cycle 6: n+6 instruction is suppressed												

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

cycles	1	2	3	4	5	6	7	8	9	10	11	12
E2	n-4	n-3	n-2	n-1	n	n+1	n+2	n+3	n+4	n+5	n+6	j
M4	n-3	n-2	n-1	n	n+1	n+2	n+3	n+4	n+5	n+6	j	j+1
M3	n-2	n-1	n	n+1	n+2	n+3	n+4	n+5	n+6	j	j+1	j+2
M2	n-1	n	n+1	n+2	n+3	n+4	n+5	n+6	j	j+1	j+2	j+3
M1	n	n+1	n+2	n+3	n+4	n+5	n+6	j	j+1	j+2	j+3	
D2	n+1	n+2	n+3	n+4	n+5	n+6	j	j+1	j+2	j+3		
D1	n+2	n+3	n+4	n+5	n+6	j	j+1	j+2	j+3			
F4	n+3	n+4	n+5	n+6	j	j+1	j+2	j+3				
F3	n+4	n+5	n+6	j	j+1	j+2	j+3					
F2	n+5	n+6	j	j+1	j+2	j+3						
F1	n+6	j	j+1	j+2	j+3							
n is the branching instruction and j is the instruction branch address												
cycle 2: branch target address "j" is fetched in fetch1 stage												
cycle 3: n+3 instruction is suppressed												
cycle 4: n+4 instruction is suppressed												
cycle 5: n+5 instruction is suppressed and for call, n+1 address is pushed on to PC stack												
cycle 6: n+6 instruction is suppressed												

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

Cycles	1	2	3	4	5	6	7	8	9	10	11	12
E2	n-4	n-3	n-2	n-1	n	n+1	n+2	n+3	n+4	n+5	n+6	r
M4	n-3	n-2	n-1	n	n+1	n+2	n+3	n+4	n+5	n+6	r	r+1
M3	n-2	n-1	n	n+1	n+2	n+3	n+4	n+5	n+6	r	r+1	r+2
M2	n-1	n	n+1	n+2	n+3	n+4	n+5	n+6	r	r+1	r+2	r+3
M1	n	n+1	n+2	n+3	n+4	n+5	n+6	r	r+1	r+2	r+3	
D2	n+1	n+2	n+3	n+4	n+5	n+6	r	r+1	r+2	r+3		
D1	n+2	n+3	n+4	n+5	n+6	r	r+1	r+2	r+3			
F4	n+3	n+4	n+5	n+6	r	r+1	r+2	r+3				

Table 4-8: Pipelined Execution Cycles for Delayed Branch (RTS(db)) (Continued)

Cycles	1	2	3	4	5	6	7	8	9	10	11	12
F3	n+4	n+5	n+6	r	r+1	r+2	r+3					
F2	n+5	n+6	r	r+1	r+2	r+3						
F1	n+6	r	r+1	r+2	r+3							
n is the branching instruction and r is the instruction at the return address												
cycle 2: branch return address "r" is fetched in fetch1 stage												
cycle 3: n+3 instruction is suppressed												
cycle 4: n+4 instruction is suppressed												
cycle 5: n+5 instruction is suppressed and r address is popped from PC stack												
cycle 6: n+6 instruction is suppressed												

In `JUMP` and `CALL` instructions that use the delayed branch `(DB)` modifier, four instruction cycles are lost in the instruction pipeline. This is because the processor executes the two instructions after the branch and the rest (four) are aborted while the instruction pipeline fills with instructions from the new location. This is shown in the sample code below.

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

As shown in the ***Pipelined Execution Cycles for Delayed Branch (JUMP or Call)*** and ***Pipelined Execution Cycles for Delayed Branch (RTS(db))*** tables, the processor executes the two instructions after the branch and the rest (four) are aborted, while the instruction at the branch address is being processed at the M1 to E2 stages of the instruction pipeline. In the case of a `CALL` instruction, the return address is the seventh address after the branch instruction. While delayed branches use the instruction pipeline more efficiently than immediate branches, delayed branch code can be harder to implement because of the instructions between the branch instruction and the actual branch. This is described in the Restrictions When Using Delayed Branches section.

Atomic Execution of Delayed Branches

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

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

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

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

IDLE Instruction in Delayed Branch

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

Restrictions When Using Delayed Branches

Besides being more challenging to code, delayed branches impose some limitations that stem from the instruction pipeline architecture. Because the delayed branch instruction and the two instructions that follow it must execute sequentially, the instructions in the two locations that follow a delayed branch instruction cannot be any of those described below.

NOTE: Development software for the processor should always flag the operations described below as code errors in the two locations after a delayed branch instruction.

Two Subsequent Delayed Branch Instructions

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

```
If gt jump (pc, 7) (db);
If le jump (pc, 11) (db);
```

As a general rule, if a branch is taken with a (DB) modifier (an unconditional branch or condition being true) then it's (DB) slot instructions should not have any branch evaluating to true or be a unconditional branch.

Other Jumps or Branches

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

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

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

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

```
if eq jump label1 (db);
if ne jump label1 (db);
```

```
nop;
nop;
```

Explicit Pushes or Pops of the PC Stack

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

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

The following instructions are executed prior to executing the RTS to return the to instruction foo.

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

If pushing the PC stack, a stack pop must be performed first, followed by an RTS instruction. If a PCSTK is popped inside a delayed call, the return address is lost. The program control returns to an unpredictable instruction when the RTS is executed at the end of the subroutine.

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

Writes to the PCSTK or PCSTKP Registers Inside a Delayed Call

If a program writes to the PC stack in the delay slots of a call, the value that is pushed onto the PC stack (due to the call) is overwritten by the value that the program writes to the PC stack. When a program performs an RTS, the program returns to the address written to the PC stack and does not return to the address pushed while branching to the subroutine. The following example demonstrates this operation.

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

The value 0x90103 is pushed onto the PC stack, while the value 0x90200 is written to the REGF_PCSTK register. Accordingly, the value 0x90103 is overwritten by the value 0x90200 in the PC stack. When the program executes an RTS, the return address is 0x90200 and not 0x90103.

Operating Mode

This section provides information on the operating modes (branching, masking, and nesting) that occur during interrupt-related variations in program flow.

These descriptions of branching, masking, and nesting variations all assume that the SHARC+ core is operating in interrupts enabled mode; the REGF_MODE1.IPERREN bit is set, enabling the interrupt controller.

Interrupt Branch Mode

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

The core event controller (CEC) is enabled by setting the global `REGF_MODE1.IRPTEN` bit. An internal interrupt can occur due to arithmetic exceptions, stack overflows, or circular data buffer overflows. Several factors control the processor's response to an interrupt. When an interrupt occurs, the interrupt is synchronized and latched in the interrupt latch register (`REGF_IRPTL`).

The processor responds to an interrupt request if:

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

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

The following example uses delayed branches to reduce latency.

```
ISR_PARI: rti;
          rti;
          rti;
          rti;
ISR_ILOPI: instruction; /* IVT branch address */
          jump ISR (db);
          instruction;
          instruction;
ISR_CB7I: rti;
          rti;
          rti;
          rti;
```

Interrupt Processing Stages

To process an interrupt, the program sequencer:

1. Outputs the appropriate interrupt vector address.

2. Pushes the current PC value (the return address) onto the PC stack.
3. Pushes the current value of the REGF_ASTATX/REGF_ASTATY and REGF_MODE1 registers onto the status stack.
4. Resets the appropriate bit in the interrupt latch register (REGF_IRPTL register).
5. Alters the interrupt mask pointer bits (REGF_IMASKP register) to reflect the current interrupt nesting state, depending on the nesting mode. The REGF_MODE1.NESTM bit determines whether all the interrupts or only the lower priority interrupts are masked during the service routine.

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

1. Returns to the address stored at the top of the PC stack.
2. Pops this value off the PC stack.
3. Pops the status stack.
4. Clears the appropriate bit in the interrupt mask pointer register (REGF_IMASKP).

Interrupt Categories

The three categories of interrupts are listed below and shown in the ***Interrupt Process Flow*** figure.

- Non maskable interrupts ($\overline{\text{RESET}}$ or emulator)
- Maskable interrupts (core or system)
- Software interrupts (core)

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

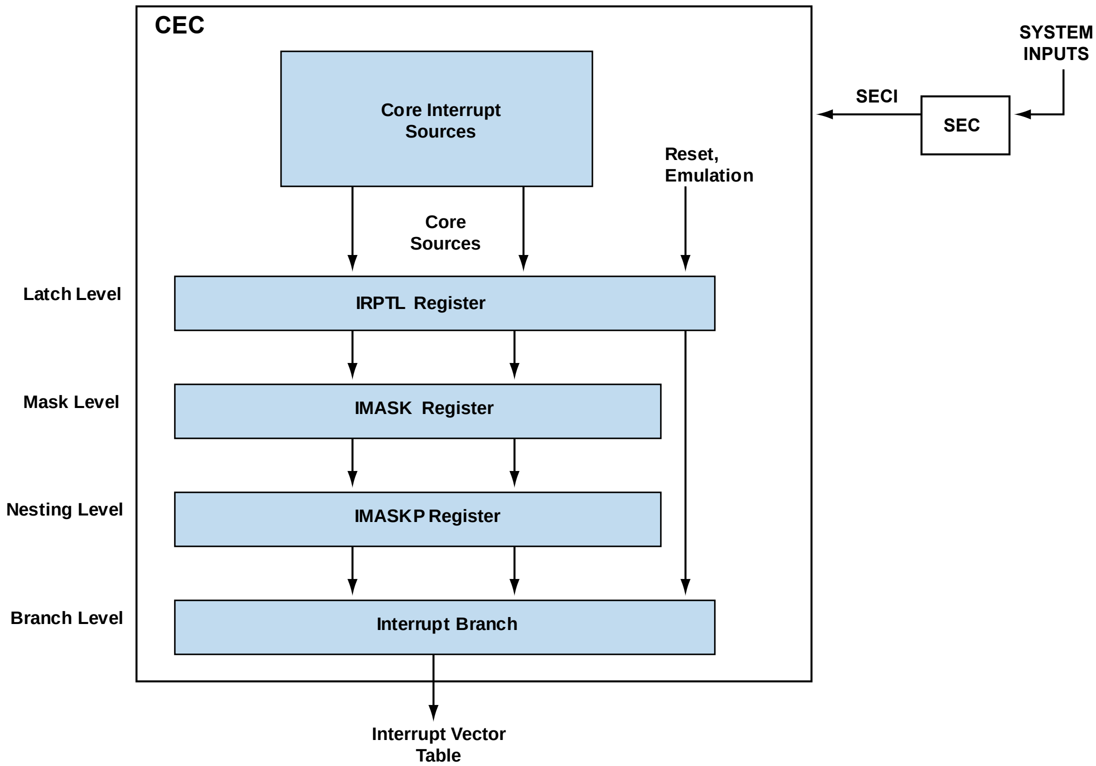

Figure 4-5: Interrupt Process Flow

The sequencer supports masking an interrupt or latching an interrupt, but does not support responding to it. Except for the `RESET` and `EMU` interrupts, all interrupts are maskable. If a masked interrupt is latched, the processor responds to the latched interrupt if it is later unmasked. Interrupts can be masked globally or selectively. Bits in the `REGF_MODE1` and `REGF_IMASK` registers control interrupt masking.

All interrupts are masked at reset except for the non-maskable reset and emulator.

Sequencer Interrupt Response

The processor responds to interrupts in three stages:

1. Synchronization (1 cycle)
2. Latching and recognition (1 cycle)
3. Branching to the interrupt vector table (11 instruction cycles)

If the branch is taken from internal memory, the 11 instruction cycles corresponds to 11 core clock cycles. If the branch is taken from external memory, the 11 instruction cycles may span over many more clock cycles depending on the actual source of the instruction and the state and configuration of the system.

The ***Pipelined Execution Cycles for Interrupt Based During Single Cycle Instruction*** table shows the pipelined execution cycles for interrupt processing.

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

cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
E2	n-2	n-1	n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	v	v+1	v+2
M4	n-1	n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	v	v+1	v+2	
M3	n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	v	v+1	v+2		
M2	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	v	v+1	v+2			
M1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	v	v+1	v+2				
D2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	v	v+1	v+2					
D1	n+4	n+5	n+6	n+7	n+8	n+9	n+10	v	v+1	v+2						
F4	n+5	n+6	n+7	n+8	n+9	n+10	v	v+1	v+2							
F3	n+6	n+7	n+8	n+9	n+10	v	v+1	v+2								
F2	n+7	n+8	n+9	n+10	v	v+1	v+2									
F1	n+8	n+9	n+10	v	v+1	v+2										
cycle1: Interrupt occurs																
cycle2: interrupt is latched and recognized, but not processed																
cycle3: n is pushed onto PC stack																
cycle4: fetch of vector address "v" starts																

NOTE: If the sequencer is executing one of the uninterruptable sequences when an interrupt occurs, a variable amount of delay occurs before the interrrupt vector starts executing.

For most interrupts, both internal (core) and external (system), only one instruction is executed after the interrupt occurs (and 11 instructions are aborted), before the processor fetches and decodes the first instruction of the service routine.

If nesting is enabled and a higher priority interrupt occurs immediately after a lower priority interrupt, the service routine of the higher priority interrupt is delayed until the first instruction of the lower priority interrupt's service routine is executed. For more information, see Interrupt Nesting Mode.

Interrupt Processing

The next several sections discuss the ways in which the SHARC+ core processes interrupts.

Core Interrupt Sources

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

- Reset - hardware/software
- emulator - debugger, breakpoints
- core timer - high, low priority

- illegal memory access and other illegal conditions - unaligned forced long word, SMMR space, illegal opcode, parity error, and others
- stack exceptions - PC, Loop, Status
- SECI - interrupts generated by system (SEC allows local system channel priority)
- DAGs - Circular buffer wrap around
- Arithmetic exceptions - fixed-point, floating-point
- Software interrupts - programmed exceptions

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

The interrupt latch bits in the REGF_IRPTL register correspond to interrupt mask bits in the REGF_IMASK register. In both registers, the interrupt bits are arranged in order of priority. The interrupt priority is from 0 (highest) up to 31 (lowest). Interrupt priority determines which interrupt must be serviced first, when more than one interrupt occurs in the same cycle. Priority also determines which interrupts are nested when the processor has interrupt nesting enabled. For more information, see Interrupt Nesting Mode and the Core Interrupt Control appendix.

Latching Interrupts

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

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

If necessary, an interrupt can be reused while it is being serviced by disabling this automatic clearing of the latch bit.

Interrupt Acknowledge

Every software routine that services core/system interrupts must clear the signaling interrupt request in the respective interrupt channel. The individual channels provide customized mechanisms for clearing interrupt requests.

For system interrupts, refer to the processor-specific hardware reference manual.

Interrupt (Pseudo) Self-Nesting

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

NOTE: A different way of self-nesting is employed only for SECI (system event controller interrupt). For more information, see Self-Nesting for the System Event Controller Interrupt (SECI).

The JUMP (CI) instruction reduces an interrupt service routine to a normal subroutine, clearing the appropriate bit in the interrupt latch and interrupt mask pointer registers and popping the status stack. After the JUMP (CI) instruction, the processor stops automatically clearing the interrupt's latch bit, allowing the interrupt to latch again. See the ***Pipelined Execution Cycles for Immediate Branch (Jump or Call)*** table in Branch Listings.

When returning from a subroutine that was entered with a JUMP (CI) instruction, a program must use a return subroutine instruction (RTS), instead of using an RTI instruction. The following example shows an interrupt service routine that is reduced to a subroutine with the (CI) modifier.

```
INSTR1;              /*  Interrupt entry from main program*/
JUMP(PC,4) (DB,CI); /*  Clear interrupt status*/
INSTR3;
INSTR4;
INSTR5;
INSTR6;
RTS;           /* Return from subroutine */
```

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

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

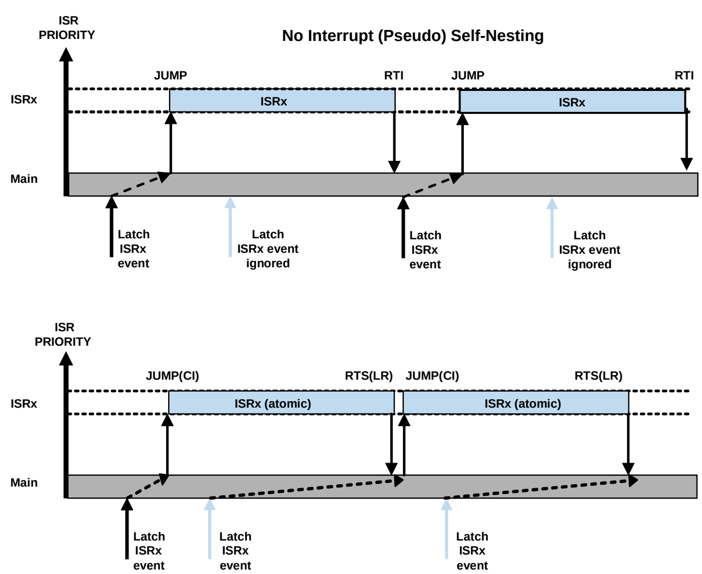

Figure 4-6: Interrupt (Pseudo) Self-Nesting

Self-Nesting for the System Event Controller Interrupt (SECI)

The mode bit, REGF_MODE2.SNEN bit, enables self-nesting interrupt mode for the SECI interrupt only. Self-nesting requires an additional bit, REGF_MODE1.SINEST.

NOTE: The System event controller (SECI) supports *true* interrupt nesting

1. The REGF_MODE2.SNEN bit enables self-nesting for SECI only.
   - When REGF_MODE2.SNEN bit =1, REGF_IMASKP.SECI bit can latch even when SECI is currently being serviced (is set in REGF_IMASKP register).
   - If REGF_MODE1.IRPTEN =1, REGF_MODE1.NESTM =1 and REGF_MODE2.SNEN =1, and the REGF_IMASKP.SECI bit is currently being serviced, the REGF_IMASKP.SECI bit is not masked but lower priority interrupts are. If a higher priority interrupt interrupts the REGF_IMASKP.SECI bit then it becomes masked.
2. The REGF_MODE1.SINEST and REGF_MODE1STK.SINEST bits controls whether REGF_IMASKP.SECI bit is cleared and the interrupts that are implicitly masked in NESTM mode.
   - When REGF_MODE2.SNEN, on vectoring to the SECI ISR, after automatically pushing the previous value of the REGF_MODE1 resister, the REGF_MODE1.SINEST bit is automatically set.
   - On executing RTI, when the current interrupt is SECI and REGF_MODE1STK.SINEST bit is set, the REGF_IMASKP register and interrupt mask are not changed. Otherwise, the REGF_IMASKP and the masked interrupts are modified as normal. After REGF_MODE1STK is tested, the RTI instruction pops the mode stack as normal.

The interrupts masked implicitly in NESTM mode can always be calculated from the REGF_IMASKP register and the REGF_MODE2.SNEN bit. When REGF_MODE2.SNEN =1 and the lowest numbered interrupt set in the REGF_IMASKP register is SECI, all interrupts down to but not including SECI are masked. Otherwise, all interrupts down to and including the lowest numbered bit set in the REGF_IMASKP register are masked, unless no bit is set in the REGF_IMASKP register, indicating no interrupts are implicitly masked.

The global interrupt enable bit, REGF_MODE1.IRPTEN, and interrupt nesting enable bit, REGF_MODE1.NESTM, take precedence over REGF_MODE2.SNEN. The SECI ISR is only interrupted by another incoming SECI if REGF_MODE1.IRPTEN =1, REGF_MODE1.NESTM =1, and REGF_MODE2.SNEN =1.

Table 4-10: SNEN and NESTM Combination and its Effect

SNEN	NESTM	Effect	
		SECI Self Nesting[*1]	Higher Priority Interrupt Nesting
0	0	NO	NO
0	1	NO	YES
1	1	YES	YES

*1 SECI is not stored in IRPTL if already in an SEC IVR. So to avoid missing any SECI when already in an SEC IVR, self-nesting of SECI must be enabled by setting SNEN bit in MODE2.

Release from IDLE

The sequencer supports placing the processor in a low power halted state called idle. The processor is in this state until an interrupt occurs. The execution of the ISR releases the processor from the idle state. When executing an `IDLE` instruction (see the ***ISA/VISA Linear Flow 48-bit Instructions Only*** figure in Linear Program Flow and the ***Pipelined Execution Cycles for IDLE Instruction*** table), the sequencer fetches six more instruction at the current fetch address and then suspends operation. The processor's internal clock and core timer (if enabled) continue to run while in the idle state. When an interrupt occurs, the processor responds normally after an eleven cycle latency to fetch the first instruction of the interrupt service routine.

The processor's DMA engines are not affected by the `IDLE` instruction. DMA transfers to or from internal memory continue uninterrupted.

NOTE: Idle instruction reduces the DMA bandwidth by 50% if executed from the same L1 bank in which DMA operation happens.

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

Table 4-11: Pipelined Execution Cycles for IDLE Instruction

<table>
<tr><th>cycles</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th></tr>
<tr><td>E2</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="8">n(idle)</td><td>n+1</td><td></td><td></td></tr>
<tr><td>M4</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="8">n(idle)</td><td>n+1</td><td>n+2</td><td></td><td></td></tr>
<tr><td>M3</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="8">n(idle)</td><td>n+1</td><td>n+2</td><td>n+3</td><td></td><td></td></tr>
<tr><td>M2</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="8">n(idle)</td><td>n+1</td><td>n+2</td><td>n+3</td><td>n+4</td><td></td><td></td></tr>
<tr><td>M1</td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="8">n(idle)</td><td>n+1</td><td>n+2</td><td>n+3</td><td>n+4</td><td>n+5</td><td></td><td></td></tr>
<tr><td>D2</td><td></td><td></td><td></td><td></td><td></td><td>n(idle)</td><td colspan="8">n+1</td><td>n+2</td><td>n+3</td><td>n+4</td><td>n+5</td><td>n+6</td><td></td><td></td></tr>
<tr><td>D1</td><td></td><td></td><td></td><td></td><td>n(idle)</td><td>n+1</td><td colspan="8">n+2</td><td>n+3</td><td>n+4</td><td>n+5</td><td>n+6</td><td>n+7</td><td></td><td></td></tr>
<tr><td>F4</td><td></td><td></td><td></td><td>n(idle)</td><td>n+1</td><td>n+2</td><td colspan="8">n+3</td><td>n+4</td><td>n+5</td><td>n+6</td><td>n+7</td><td>v</td><td></td><td></td></tr>
<tr><td>F3</td><td></td><td></td><td>n(idle)</td><td>n+1</td><td>n+2</td><td>n+3</td><td colspan="8">n+4</td><td>n+5</td><td>n+6</td><td>n+7</td><td>v</td><td>v+1</td><td></td><td></td></tr>
<tr><td>F2</td><td></td><td>n(idle)</td><td>n+1</td><td>n+2</td><td>n+3</td><td>n+4</td><td colspan="8">n+5</td><td>n+6</td><td>n+7</td><td>v</td><td>v+1</td><td>v+2</td><td></td><td></td></tr>
<tr><td>F1</td><td>n(idle)</td><td>n+1</td><td>n+2</td><td>n+3</td><td>n+4</td><td>n+5</td><td colspan="8">n+6</td><td>n+7</td><td>v</td><td>v+1</td><td>v+2</td><td>v+3</td><td></td><td></td></tr>
<tr><td colspan="22">cycle1:idle instruction is fetched at n</td></tr>
<tr><td colspan="22">cycle14 : interrupt is latched and recognized</td></tr>
</table>

Table 4-11: Pipelined Execution Cycles for IDLE Instruction (Continued)

cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
cycle16:interrupt branches to v																					

Causes of Delayed Interrupt Processing

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

- During the start and termination of short loops encoded as F1 type.
  - Up to four instructions after execution of DO..UNTIL.
  - Up to nine instructions when loop terminates, that is, L-9 to L-1 instructions of unrolled short loop.
- Two instructions in delay slot of a delayed branch are uninterruptible.
- The last but one instruction in arithmetic loop is uninterruptible during the last iteration of the loop.

All cycles during a pipeline flush remain uninterruptible.

Interrupt Mask Mode

The SHARC+ core supports many different operating modes (SIMD, bit reversal, circular buffer, rounding). Interrupt mask mode provides a mechanism that lets the core change its operating mode without performing an explicit operation to perform masking through setting the `REGF_MODE1` register bits. To accomplish this, a copy of the `REGF_MODE1` register is used to mask specific operating modes across interrupts.

Bits that are set in the `REGF_MMASK` register are used to clear bits in the `REGF_MODE1` register when the processor's status stack is pushed. This effectively disables different modes when servicing an interrupt, or when executing a `PUSH STS` instruction. The processor's status stack is pushed in two cases:

1. When executing a `PUSH STS` instruction explicitly in code.
2. When any interrupt occurs.

For example:

Before the `PUSH  STS` instruction, the `REGF_MODE1` register enabled the following bit configurations:

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

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

The value in the REGF_MODE1 register after PUSH STS instruction is:

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

The other settings that were previously set in the REGF_MODE1 register remain the same. The only bits that are affected are those that are set both in the REGF_MMASK and REGF_MODE1 registers. These bits are cleared after the status stack is pushed.

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

Interrupt Nesting Mode

The sequencer supports interrupt nesting-responding to another interrupt while a previous interrupt is being serviced. Bits in the REGF_MODE1and REGF_IMASKPregisters control interrupt nesting as described below.

The REGF_MODE1.NESTM bit directs the processor to enable (if 1) or disable (if 0) interrupt nesting.

When interrupt nesting is enabled, a higher priority interrupt can interrupt a lower priority interrupt's service routine (see ***Interrupt Nesting*** figure). Lower priority interrupts are latched as they occur, but the processor processes them according to their priority after the nested routines finish.

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

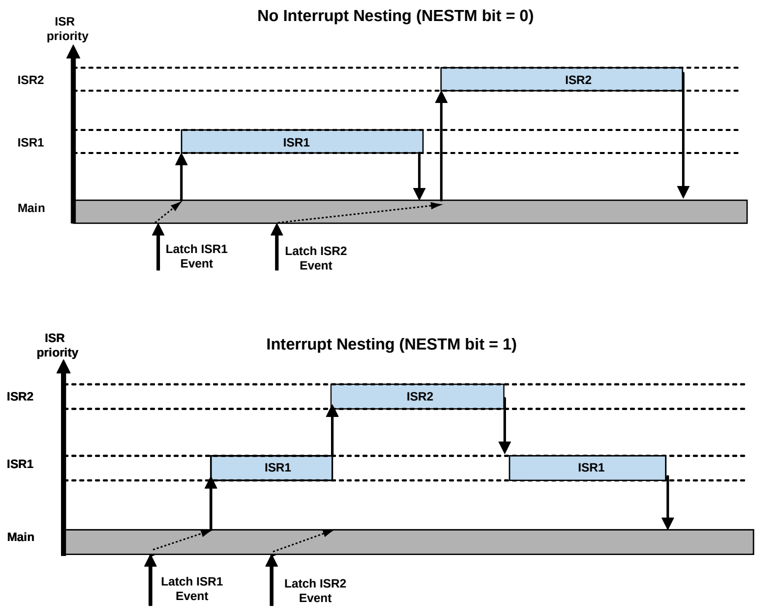

Figure 4-7: Interrupt Nesting

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

Programs should change the interrupt nesting enable bit (REGF_MODE1.NESTM) only while outside of an interrupt service routine or during the reset service routine.

ATTENTION: If nesting is enabled and a higher priority interrupt occurs immediately after a lower priority interrupt, the service routine of the higher priority interrupt is delayed. This delay allows the first instruction of the lower priority interrupt routine to be executed, before it is interrupted (see ***Interrupt Nesting*** figure).

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

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

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

ATTENTION: The entire set of the REGF_IMASKP registers are for interrupt controller use only. Modifying these bits interferes with the proper operation of the interrupt controller.

Loop Sequencer

The program sequencer includes special hardware to execute zero or low-overhead loops. The relevant state machine is activated when a DO..UNTIL instruction executes. The state machine manages all the resources associated with hardware loops such as loop counters, stacks and others. The number of times a loop iterates can be controlled by a hardware counter (LCNTR) or by a flag in the REGF_ASTATX and REGF_ASTATY registers.

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

- Updates the PC stack with the top of loop address.
- Updates the loop stack with the address of the last instruction of the loop.
- Initializes the REGF_LCNTR and REGF_CURLCNTR registers and updates the loop counter stack, if the loop is counter based (do until lce).
- Generates the loop-back (go to the beginning of loop) and loop abort (come out of loop, fetch next instruction from “last instruction of loop plus one” address) signals, according to defined termination condition.
- Generates the abort signals to suppress some of the extra fetched instructions (in certain cases of loops, some unwanted instructions may get fetched).
- Handles interrupts without distorting the intended loop-sequencing (until or unless interrupt service routine deliberately manipulates the status of loop-sequencer resources).

A loop occurs when a DO/UNTIL instruction instructs the processor to repeat a sequence of instructions until a condition tests true or indefinite by using FOREVER as termination condition. The SHARC+ core automatically evaluates the loop termination condition and modify the program counter (REGF_PC) register appropriately. This significantly speeds up execution of loops by eliminating flushed cycles in a pipelined processor. In many cases, the number of lost cycles are completely eliminated.

Loop Categories

Based on the termination criteria of a loop, loops are categorized as follows:

- *Counter based loop* – These are started by a `DO...UNTIL LCE` instruction. Counter based loops are comprised of instructions that are set to run a specified number of iterations. These iterations are controlled by a loop counter register (REGF_LCNTR). The REGF_LCNTR register is a non memory-mapped universal register that is initialized to the count value and the loop counter expired (LCE) instruction is used to check the termination condition. Expiration of LCE signals that the loop has completed the number of iterations as per the count value in the REGF_LCNTR register.

- *Arithmetic Loops* – these loops are started with conditions other than LCE. The sequencer iterates the instructions in the loop body until the specified condition tests true.

Counter based loops are handled by the loop state machine in one of the following modes:

- *E2-active mode*: REGF_CURLCNTR is decremented and is tested for zero when last instruction of the loop is in E2 stage of the pipeline (default loop). Any loop is by default of E2-active type. Because the pipeline already contains the instructions from loop for the next iteration, on completion of loop, the entire pipeline is flushed, and fetch of instructions beyond loop body is started. Consequently, these loops have the overhead of an eleven-cycle pipeline flush on completion of the loop.
- *F1-active mode*: REGF_CURLCNTR is decremented and is tested for zero when the last instruction of the loop is in F1 stage of the pipeline. On expiry of the counter (completion of the loop), the fetch of instruction beyond the loop is started in next cycle. Consequently, loops executed in this mode do not waste any cycles on completion.

The F1-active mode of execution is preferred due to its zero overhead. However, presence of other branches in the pipeline interfere with working of the loop state machine. So, for proper functioning, only loops that do not contain branch or IDLE in last eleven instructions of the loop body are executed in F1-active mode. The mode in which a counter based loop executes is determined by the opcode of the DO...UNTIL LCE instruction.

NOTE: The assembler generates appropriate opcode after examination of the loop body.

Counter-Based F1-Active Loop

For F1-active counter-based loop, the current loop counter decrement (REGF_CURLCNTR) and termination conditions check happens in F1 stage of pipe.

Entering Loop Execution

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

The processor tests the termination condition and decrements the counter when the end-of-loop address is in F1 stage, so that the next fetch either exits the loop or returns to the top. If the termination condition is not satisfied, the processor re-fetches the instruction from the top-of-loop address stored on the top of PC stack.

Table 4-12: Loop Length 11, Entering into Loop Execution

Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
E2											n(DO)	n+1	n+2	n+3	n+4
M4										n(DO)	n+1	n+2	n+3	n+4	n+5
M3									n(DO)	n+1	n+2	n+3	n+4	n+5	n+6
M2								n(DO)	n+1	n+2	n+3	n+4	n+5	n+6	n+7
M1							n(DO)	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8

Table 4-12: Loop Length 11, Entering into Loop Execution (Continued)

Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D2						n(DO)	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9
D1					n(DO)	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10
F4				n(DO)	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11
F3			n(DO)	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+1
F2		n(DO)	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+1	n+2
F1	n(DO)	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+1	n+2	n+3
cycle 1: DO UNTIL enters F1 stage															
cycle 11: DO UNTIL reaches E2 stage and pushes loop state machine information on to the loop stack and the PC stack															
cycle 12: end-of-loop address "n+11" appears in F1 stage, triggering loop back logic to occur in the next cycle															
cycle 13: loop back occurs, resulting in top-of-loop address "n+1" appearing in F1 stage															

Terminating Loop Execution

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

For F1-active counter-based loop, termination condition is checked whenever a valid end-of-loop address appears in F1-stage of pipe. And termination condition is considered true when the `REGF_CURLCNTR` register value is one and valid end-of-loop address is present in F1 stage of pipe. Since the termination condition is checked in F1-stage of pipe, F1-active counter-based loop causes zero cycle overhead.

Table 4-13: Loop Length 11, Terminating Loop Execution

Cycles	1	2	3	4	5
E2	n(DO)	n+1	n+2	n+3	n+4
M4	n+1	n+2	n+3	n+4	n+5
M3	n+2	n+3	n+4	n+5	n+6
M2	n+3	n+4	n+5	n+6	n+7
M1	n+4	n+5	n+6	n+7	n+8
D2	n+5	n+6	n+7	n+8	n+9
D1	n+6	n+7	n+8	n+9	n+10
F4	n+7	n+8	n+9	n+10	n+11
F3	n+8	n+9	n+10	n+11	n+12
F2	n+9	n+10	n+11	n+12	n+13
F1	n+10	n+11	n+12	n+13	n+14

Table 4-13: Loop Length 11, Terminating Loop Execution (Continued)

Cycles	1	2	3	4	5
cycle 2: end-of-loop address "n+11" appears in F1 stage. Loop termination condition is checked, and (if true) loop abort happens. Then, the next consecutive address "n+12" (which is next to the end-of-loop) is fetched.					

Counter-Based E2-Active Loop

E2-active loop is similar to F1-active loop in terms use of Loop stack, PC stack and loopback. But the counter decrement and checking of expiry of the counter is performed when the last instruction of the loop body is in E2-stage of pipe.

Entering Loop Execution

Similar to F1-active loop, E2-active loop also saves information on Loop stack and PC stack.

The processor tests the termination condition and decrements the counter when the end-of-loop address is in E2 stage. The loop back of E2-active loop also happens in F1 stage of pipe similar to F1-active loop. Whenever last-of-loop address appears in F1 stage of pipe and loop has not yet terminated, loopback happens.

Table 4-14: Loop Length 11, Entering into Loop Execution

Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
E2											n(DO)	n+1	n+2	n+3	n+4				
M4										n(DO)	n+1	n+2	n+3	n+4	n+5				
M3									n(DO)	n+1	n+2	n+3	n+4	n+5	n+6				
M2								n(DO)	n+1	n+2	n+3	n+4	n+5	n+6	n+7				
M1							n(DO)	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8				
D2						n(DO)	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9				
D1					n(DO)	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10				
F4				n(DO)	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11				
F3			n(DO)	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+1				
F2		n(DO)	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+1	n+2				

Table 4-14: Loop Length 11, Entering into Loop Execution (Continued)

Cy-cles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
F1	n(DO)	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+1	n+2	n+3				
cycle 1: DO UNTIL enters F1 stage																			
cycle 11: DO UNTIL reaches E2 stage and pushes loop state machine information on to the loop stack and PC stack																			
cycle 12: end-of-loop address "n+11" appears in F1 stage, triggers loop back logic to occur on the next cycle																			
cycle 13: loop back occurs, and top-of-loo address "n+1" appears in F1 stage																			

Terminating Loop Execution

If the termination condition is true, the sequencer pops the loop stack and PC stack, and immediately fetches instruction which is next to end-of-loop address, in the next cycle.

For E2-active C-Loop, termination condition is checked whenever a valid end-of-loop address appears in E2-stage of pipe. And termination condition is considered true when CURLCNTR value is one and valid end-of-loop address is present in E2 stage of pipe. Since the termination condition is checked in E2 stage of pipe, instructions present in the pipe from M4 to F1 stages are flushed if the termination condition is found true. Consequently all E2-active loops have this overhead of eleven lost cycles.

Table 4-15: Loop Length 11, Entering into Loop Execution

Cy-cles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
E2	n(DO)	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+1	n+2	n+3	n+4	n+5	n+6	n+7
M4	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8
M3	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9
M2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10
M1	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11
D2	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+12
D1	n+6	n+7	n+8	n+9	n+10	n+11	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+12	n+13
F4	n+7	n+8	n+9	n+10	n+11	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+12	n+13	
F3	n+8	n+9	n+10	n+11	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+12	n+13		
F2	n+9	n+10	n+11	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+12	n+13			
F1	n+10	n+11	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+12	n+13				
cycle 12: end-of-loop address appears in E2 stage. The counter decrements, and the termination conditions is checked. If the termination condition tests true, loop termination happens, and the loop stack and PC stack are popped.																			

Table 4-15: Loop Length 11, Entering into Loop Execution (Continued)

Cy-cles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
cycle 14: after loop termination, the next address "n+12" to the end-of-lop address is fetched in the F1 stage.																			

Loop Categorization into F1-Active or E2-Active

Determination of F1-active or E2-active mode for a hardware counter based loop is based on the opcode. The assembler identifies loops which are safe to execute in F1-active mode and uses the F1-active opcode. Loops not having a change of flow (jump, call etc.) or IDLE in last eleven instructions of the loop body are safe to execute as F1-active mode.

Short loops with few iterations where the total number of instructions of fully unrolled loop is less than eleven, always execute as E2-active mode irrespective of opcode. The `REGF_MODE2.SLOWLOOP` bit can be set to override the opcode of F1-active loop.

NOTE: With the `REGF_MODE2.SLOWLOOP` bit =1, all counter based loops execute in E2-active mode. This mode bit is intended to be primarily used by the debugger.

Arithmetic Loops

Arithmetic loops are loops where the termination condition in the DO/UNTIL loop is anything other than LCE. In this type of loop, where the body has more than one instruction, the termination condition for loop length 3 and above is checked when L-2nd instruction is in E2 stage of pipe. And for loop length 1 and 2, the termination condition is checked when the last instruction is in E2 stage of pipe. An example of an arithmetic loop is given below.

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

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

If the termination condition tests false, the next instruction is fetched. If the termination condition tests true, one more instruction (which is loop's 1st instruction) is allowed to execute and all the rest of the instructions in the below stages of pipe are flushed. Also, the end-of-loop instruction is fetched in the F1 stage in the next cycle and subsequent instructions (which are next to end-of-loop) are fetched in subsequent cycles.

NOTE: In nested arithmatic loops when the terminating condition is set for the outer loop during the execution of a call instruction by the inner loop, the SHARC+ core iterates an arithmetic loop one additional time in comparison to the 5-stage SHARC.

The ***Arithmetic Loop Length 11, Terminating Loop Execution*** table shows the execution cycles for an arithmetic loop with eleven instructions.

Table 4-16: Arithmetic Loop Length 11, Terminating Loop Execution

Cycles	1	2	3	4	5	6	7	8	9	10	11	12
E2	n(DO)	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11
M4	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+1
M3	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+1	n+2
M2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+1	n+2	n+3
M1	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+1	n+2	n+3	n+4
D2	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+1	n+2	n+3	n+4	n+5
D1	n+6	n+7	n+8	n+9	n+10	n+11	n+1	n+2	n+3	n+4	n+5	n+6
F4	n+7	n+8	n+9	n+10	n+11	n+1	n+2	n+3	n+4	n+5	n+6	n+7
F3	n+8	n+9	n+10	n+11	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8
F2	n+9	n+10	n+11	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9
F1	n+10	n+11	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+11
cycle 1: Do Until executes and pushes loop related information on the loop stack and PC stack												
cycle 3: loop back occurs and top-of-loop address "=1" is fetched in F1 stage												
cycle 10: A-loop termination condition is checked when loop 2nd instruction (for example, "n+9") is in E2 stage												
cycle 11: after termination condition tests true, loop 1st instruction (for example, "n+10") is allowed to execute												
cycle 12: endo-of-loop address (for example, "n+11") is fetched in F1 stage												

Table 4-17: Arithmetic Loop Length 11, Terminating Loop Execution (continued)

Cycles	13	14	15	16	17	18	19	20	21
E2	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9
M4	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+11
M3	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+11	n+12
M2	n+4	n+5	n+6	n+7	n+8	n+9	n+11	n+12	n+13
M1	n+5	n+6	n+7	n+8	n+9	n+11	n+12	n+13	n+14
D2	n+6	n+7	n+8	n+9	n+11	n+12	n+13	n+14	n+15
D1	n+7	n+8	n+9	n+11	n+12	n+13	n+14	n+15	n+16
F4	n+8	n+9	n+11	n+12	n+13	n+14	n+15	n+16	n+17
F3	n+9	n+11	n+12	n+13	n+14	n+15	n+16	n+17	n+18
F2	n+11	n+12	n+13	n+14	n+15	n+16	n+17	n+18	n+19
F1	n+12	n+13	n+14	n+15	n+16	n+17	n+18	n+19	n+20

NOTE: For single instruction loops, the termination condition is checked in every cycle. For two-instruction loops, the termination condition is checked when the end-of-loop instruction is executed. Since two more instructions are always allowed to execute after the termination condition tests true, a single-instruction loop executes for two more iterations and a two-instruction loop executes one more iteration before exiting the loop.

Indefinite Loops

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

```
DO label UNTIL FOREVER; /* pushed LCNTR onto Loop count stack */
R6 = DM(I0,M0);         /* pushed to PC stack */
R6 = R6 - 1;
IF EQ CALL SUB;
nop;
label: nop;             /* pushed to loop address stack */
```

Loop Resources

The sequencer provides a number of resources that support stack management and manipulation.

These resources include the following:

- Loop stack
- Loop address stack access
- Loop address stack status
- Loop address stack manipulation
- Loop counter stack access
- Loop counter stack status
- Loop counter stack manipulation
- Loop counter expired condition (for terminating counter-based loops)

Loop Stack

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

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

Loop Address Stack Access

The sequencer's loop support, as shown in the ***Sequencer Control Diagram*** figure, includes a loop address stack. The sequencer pushes the termination address, termination code and the loop type information (Cloop/Aloop/Forever) onto the loop address stack when executing a `DO`/`UNTIL` instruction. For an F1-active loop the sequencer tests the termination condition when end-of-loop address is in F1 stage of pipe, the loop stack pops before the end-of-loop address is excuted in E2-stage. If a program reads the REGF_LADDR register in the last ten instructions when loop has terminated, the value is already the termination address for the next loop stack entry. For an E2-active loop, since the termination condition is checked in E2 stage, the read REGF_LADDR value is always current loop stack entry.

Loop Address Stack Status

The loop address stack is six levels deep by 32 bits wide. A stack overflow occurs if a seventh entry (one more than full) is pushed onto the loop stack. The stack is empty when no entries are occupied. Because the sequencer keeps the loop stack and loop counter stack synchronized, the same overflow and empty status flags apply to both stacks. These flags are in the sticky status register (REGF_STKYX). For more information on this register, see the ***STKYx and STKYy Register Bit Descriptions (RW)*** table in the Registers appendix. For more information on how these flags work with the loop stacks, see Loop Counter Stack Access. Note that a loop stack overflow causes a maskable interrupt.

Loop Address Stack Manipulation

The REGF_LADDR register contains the top entry (tge) on the loop address stack. This register is readable and writable over the DM data bus. Reading from and writing to the REGF_LADDR register does not move the loop address stack pointer. Only a stack push or pop performed with explicit instructions moves the stack pointer. The REGF_LADDR register contains the value 0xFFFF FFFF when the loop address stack is empty. The ***Loop Address Stack Register (LADDR)*** table in the Registers appendix lists the bits in this register.

The PUSH LOOP instruction pushes the stack by changing the pointer only. It does not alter the contents of the loop address stack. Therefore, the PUSH LOOP instruction should be usually followed by a write to the REGF_LADDR register.

Loop Counter Stack Access

The sequencer's loop support, shown in the ***Sequencer Control Diagram*** figure in Features, also includes a loop counter stack. The loop counter stack is six locations deep by 32 bits wide. The stack is full when all entries are occupied, is empty when no entries are occupied, and is overflowed if a push occurs when the stack is already full. Bits in the REGF_STKYX register indicate the loop counter stack full and empty states.

NOTE: A value of zero in the REGF_LCNTR register causes a loop to execute 2^{32} times.

Loop Counter Stack Status

The loop counter stack is six locations deep by 32 bits wide. The stack is full when all entries are occupied, is empty when no entries are occupied, and is overflowed if a push occurs when the stack is already full. Bits in the REGF_STKYX register indicate the loop counter stack full and empty states. The ***Loop Address Stack Register (LADDR)*** table in the Registers appendix lists the bits in the REGF_STKYX register. The following bits in the REGF_STKYX register indicate the loop counter stack full and empty states.

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

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

NOTE: The sequencer keeps the loop counter stack synchronized with the loop address stack. Both stacks always have the same number of locations occupied. Because these stacks are synchronized, the same empty and overflow status flags from the REGF_STKYX register apply to both stacks.

Loop Counter Stack Manipulation

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

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

Loop Counter Expired (If Not LCE Condition) in Counter-Based Loops

Since a counter based loop can be either F1-active loop or E2-active loops, the REGF_CURLCNTR register value is changed based on the presence of end-of-loop address either in F1-stage or in E2-stage. For a deterministic behavior of IF NOT LCE condition it is advisable not to use this condition in the last eleven instruction of a counter based loop.

Restrictions on Ending Loops

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

- For SHARC+ core pipeline increase, the natural extension of the LR rule is that it should be used if the call is one of the last five instructions inside a loop. To keep the rule backward compatible, any RTS without LR will also be treated as a RTS with a LR. This ensures that even if a call is placed at last 4th or 5th instruction inside a loop and RTS for that call is not paired with LR, the loop counter is not decremented twice. (In 5-stage pipeline SHARC products if a call is one of the last three instructions inside a loop, a RTS for that call had to be paired with LR modifier to prevent the Loop counter from decrementing twice for the same iteration.)
- There is a one cycle latency between a multiplier status change and arithmetic loop abort (LA). This extra cycle is a machine cycle, not an instruction cycle. Therefore, if there is a pipeline stall (due to external memory access for example), then the latency is not applicable.
- An IF NOT LCE conditional instruction cannot be used as the instruction that follows a write to the REGF_CURLCNTR register.
- The loop controller uses both the loop stack and the program control stack for its operation. Manipulation of these stacks by using PUSH/POP instructions and explicit writes to these stacks may affect the correct functioning of the loop.

- The IDLE and EMUIDLE instructions should not be used in the last three instructions of any arithmetic loop.

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

VISA-Related Restrictions on Hardware Loops

The last 11 instruction of a hardware loop must be encoded as legacy Instruction Set Architecture (ISA) instructions. These loop end instructions may not be encoded as Variable Instruction Set Architecture (VISA) instructions.

This restriction *against VISA encoded instructions at the end of a loop* is required for two reasons:

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

NOTE: As the last 11 instructions of a hardware loop must be encoded as ISA (traditional 48-bit) instructions, the CrossCore Embedded Studio code-generation tools from Analog Devices automatically do encode these as ISA instructions. For more information about ISA and VISA instructions, see Instruction Pipeline.

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

In cases of short loops (loops with a body shorter than 11 instructions), the above rule extends to state that all the instructions in the loop are encoded as ISA instructions (left uncompressed).

Nested Loops

Signal processing algorithms like FFTs and matrix multiplications require nested loops. Nested loop constructs are built using multiple DO/UNTIL instructions. If using counter based instructions, within the loop sequencer, two separate loop counters operate:

- Loop counter (REGF_LCNTR) register has top level entry to loop counter stack
- Current loop counter (REGF_CURLCNTR) iterates in the current loop

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

The loop counter stack is popped on termination of the loop. The cycle in which a loop is effectively terminated depends on the type (F1- or E2-active) of the loop. When the loop counter stack is popped, the new top entry of the stack becomes the REGF_CURLCNTR value—the count in effect for the executing loop.

Two examples of nested loops are shown in the ***Nested Counter-Based Loop*** and ***Nested Mixed-Base Loop*** examples.

Nested Counter-Based Loop

```
LCNTR = S, DO the_end UNTIL LCE;    /*outer Loop*/
Instruction;
Instruction;
```

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

Nested Mixed-Based Loop

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

Example For Six Nested Loops

A DO/UNTIL instruction pushes the value of LCNTR onto the loop counter stack, making that value the new CURLCNTR value. The following procedure and the ***Pushing the Loop Counter Stack for Nested Loops*** figure demonstrate this process for a set of nested loops. The previous CURLCNTR value is preserved one location down in the stack.

1. The processor is not executing a loop, and the loop counter stack is empty (LSEM bit =1). The program sequencer loads the REGF_LCNTR register with AAAA AAAA.
2. The processor is executing a single loop. The program sequencer loads LCNTR with the value BBBB BBBB (LSEM bit =0).
3. The processor is executing two nested loops. The program sequencer loads the REGF_LCNTR register with the value CCCC CCCC.
4. The processor is executing three nested loops. The program sequencer loads the REGF_LCNTR register with the value DDDD DDDD.
5. The processor is executing four nested loops. The program sequencer loads the REGF_LCNTR register with the value EEEE EEEE.
6. The processor is executing five nested loops. The program sequencer loads the REGF_LCNTR register with the value FFFF FFFF.
7. The processor is executing six nested loops. The loop counter stack (LCNTR) is full (REGF_STKYX.LSOV bit =1).

A read of the REGF_LCNTR register when the loop counter stack is full results in invalid data. When the loop counter stack is full, the processor discards any data written to LCNTR.

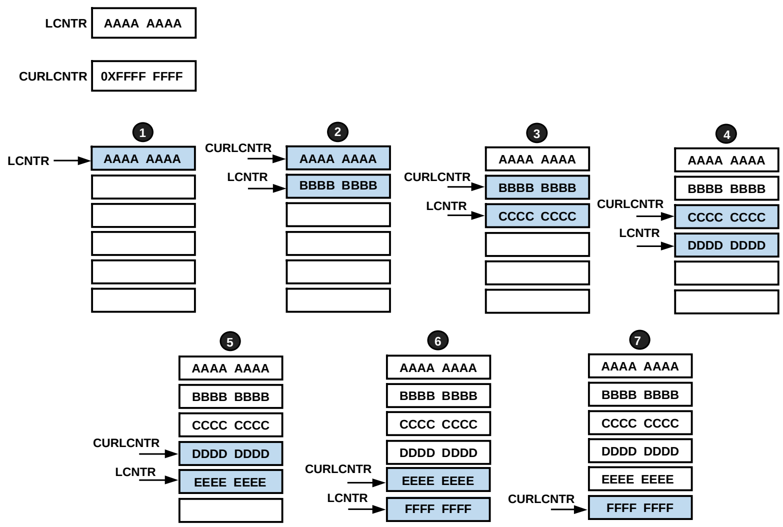

Figure 4-8: Pushing the Loop Counter Stack for Nested Loops

Restrictions on Ending Nested Loops

The sequencer's loop features (which optimize performance in several ways) limit the types of instructions that may appear at or near the end of the loop.

These restrictions include the following:

- Nested loops cannot use the same end-of-loop instruction address. The sequencer resolves whether to loop back or not, based on the termination condition. If multiple nested loops end on the same instruction, the sequencer exits all the loops when the termination condition for the current loop tests true. There may be other sequencing errors.
- Nested loops with an arithmetic loop as the outer loop must place the end address of the outer loop at least two addresses after the end address of the inner loop.
- Nested loops with an arithmetic based loop as the outer loop that use the loop abort instruction, `JUMP (LA)`, to abort the inner loop, may not use `JUMP (LA)` to the last instruction of the outer loop.

Loop Abort

The hardware loop state machine maintains and manages various state information. Normally branches are allowed within a loop body. It is allowed for these branches to even transfer control outside of the loop body. A `CALL` is an example of this. For the purposes of the looped execution, these instructions executed even outside of loop body are effectively part of the loop. Loops normally terminate when the specified loop termination condition tests true.

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

```
/* Example: Loop Abort Instruction, JUMP (LA) */

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

NOTE: In 5-stage SHARC products and earlier, if a `CALL` is one of the last three instructions inside a loop, the `RTS` instruction for that call had to be paired with a LR. This prevents the loop counter from decrementing twice for the same iteration. The LR (loop re-entry) modifier for `RTS` has been deprecated in the SHARC+ core. The situations where use of `RTS(LR)` was required have been eliminated by introducing E2-active mode of execution of some of the loops.

Interrupt Driven Loop Abort

For servicing the interrupt, eleven instructions in the various stages of the pipeline are replaced with `NOP` instructions. Accordingly, the hardware loop logic freezes the `REGF_CURLCNTR` for the required fetch cycles on return from an ISR. The hardware determines this based on the sequencer executing a `RTI` instruction.

The ***Pipeline Interrupt in a Loop*** table shows a pipeline where an interrupt is being serviced in a loop. e = end-of-loop instruction, b = top-of-loop instruction. e-1 is the return address.

NOTE: There is one situation where an ISR returns into the loop body using the `RTS` instruction. This situation occurs when `JUMP (CI)` is used to convert an ISR to a normal subroutine. Therefore, an `RTS` cannot be used to determine that the sequencer branched off to a subroutine or ISR. For this reason, the hardware sets an additional bit in the `REGF_PCSTK` register, before branching off to an ISR so that on return, either with a `RTI` or `JUMP (CI) + RTS CURLCNTR` instruction can be frozen for required number of cycles.

Table 4-18: Pipeline Interrupt in a Loop

Cycles	1	2	3	4	5	6	7	8	…	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+12
E2				e-2	e-1	e							RTI								e-1
M4			e-2	e-1	e							RTI								e-1	e
M3		e-2	e-1	e							RTI								e-1	e	b
M2	e-2	e-1	e							RTI								e-1	e	b	b+1
M1	e-1	e															e-1	e	b	b+1	b+2
D2	e															e-1	e	b	b+1	b+2	
D1															e-1	e	b	b+1	b+2		

Table 4-18: Pipeline Interrupt in a Loop (Continued)

Cycles	1	2	3	4	5	6	7	8	~	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n +10	n +11	n +12
F4														e-1	e	b	b+1	b+2			
F3								v(ISR)					e-1	e	b	b+1	b+2				
F2							v(ISR)	v+1				e-1	e	b	b+1	b+2					
F1						v(ISR)	v+1	v+2			e-1	e	b	b+1	b+2						
cycle 4: interrupt is recognized; e-1 is pushed to PC stack; pipeline flushed																					
cycle 6: instruction from ISR is fetched in F1 stage																					
cycle n+1: PC returns back from ISR and e-1 is fetched in F1 stage. From CURLCNTR is frozen for required number of cycles.																					

Loop Resource Manipulation

The SHARC+ core prohibits any modification of loop resources, such as the REGF_PCSTK, REGF_LADDR, and REGF_CURLCNTR registers within the loop (including subroutines and ISRs starting from a loop) as doing this may adversely affect the proper function of the looping operation. The manipulation of these resources are allowed only when it is done in accordance with the loop restrictions (for example, restrictions on ending loops and other restrictions).

The loop hardware state machine maintains certain information of the ongoing loops on loop stack and PC stack. The processor relies on this information for correct execution of loops under various conditions. Popping and pushing REGF_LADDR/REGF_CURLCNTR and REGF_PCSTK registers with new values generally interferes with proper loop function. However, popping and pushing the loop and PC stack to temporarily vacate the stacks can still be performed from an ISR by following the procedure described in Popping and Pushing Loop and PC Stack From an ISR.

NOTE: A fundamental requirement for processors using a real-time operating system (RTOS) is support for context switching. A context switch of the processor forces a save all core registers on the software stack, including the core stack registers.

Popping and Pushing Loop and PC Stack From an ISR

Use the following sequence to pop and push REGF_LADDR/REGF_CURLCNTR and REGF_PCSTK to temporarily vacate the stacks.

1. Pop `LOOP` and `PCSTK` after storing the value of the REGF_CURLCNTR, REGF_LADDR, and REGF_PC registers.
2. Use the empty entry/entries of stacks.
3. Recreate the loops by performing the following steps in the proscribed sequence.
   a. Push `LOOP` stack.

b. Load the value of REGF_CURLCNTR.

c. Load the REGF_LADDR.

d. Push the PCSTK.

e. Load the REGF_PC with the stored value.

The ***Sequence for Pop and Push of Two-deep Nested Loops*** code listing provides an example of the sequence of operations. The sequence of operations is critical and must be followed exactly. Any number of unrelated instructions may be executed during step 2 of the sequence ("Use the empty entry/entries of stacks").

Interrupts should not be triggered during this sequence of operations. Disable the interrupts by clearing the REGF_MODE1.IRPTEN bit. Consider the two cycles of effect latency before the restoration sequence, starting with the first instruction of the sequence and ending with setting the REGF_MODE1.IRPTEN bit after the last instruction in the sequence (following completion of restoration).

In the ***Sequence for Pop and Push of Two-deep Nested Loops*** example, REGF_LADDR is restored after REGF_CURLCNTR. This order of restoration ensures that when REGF_LADDR is restored, the correct value of loop count is available. At the time of REGF_LADDR restoration, the hardware recreates the information about the exact characterization of the loop.

Sequence for Pop and Push of Two-deep Nested Loops

```
/* --- Step 1: Pop and Store --- */
R1 = LADDR;
R2 = CURLCNTR;
R3 = PCSTK;
POP LOOP;
POP PCSTK;
NOP;
R4 = LADDR;
R5 = CURLCNTR;
R6 = PCSTK;
POP LOOP;
POP PCSTK;
NOP;
/* --- Store the registers to memory here --- */
/* --- Step 2: Miscellaneous instruction or instructions related or unrelated to
hardware loops --- */
/* --- Load the registers from memory here --- */
/* --- Step 3: Push and Load --- */
PUSH LOOP;
CURLCNTR = R5;
LADDR = R4;
PUSH PCSTK;
PCSTK = R6;
PUSH LOOP;
CURLCNTR = R2;
LADDR = R1;
```

```
PUSH PCSTK;
PCSTK = R3;
```

Instruction-Conflict Cache Control

This section on instruction-conflict cache control describes a special type of cache that is related to instruction fetch.

Functional Description

The SHARC+ core has a traditional instruction conflict cache (Support for Super Harvard Architecture), and new instruction/data caches. This section describes the traditional instruction conflict cache which affects internal memory only. For information about the instruction/data caches refer to the L1 Cache Controller chapter.

NOTE: The instruction conflict cache is the "cache" which is available as part of all generations of the SHARC and SHARC+ cores. The instruction cache and data cache are available on the SHARC+ cores.

Instruction Data Bus Conflicts

A bus conflict occurs when the PM data bus, normally used to fetch an instruction in each cycle, is used to fetch an instruction and to access data in the same cycle. If an instruction at the M1 stage uses the PM bus to access data, it creates a conflict with the instruction fetch at the Fetch1 stage, assuming sequential executions.

In the event of such bus conflict, the bus operations are serialized. The instruction conflict cache stores only those instructions whose fetch operation involves a bus conflict. In subsequent instance of fetch of these stored instructions, conflict-cache supplies the instruction, avoiding the bus conflict altogether.

Cache Miss

In the instruction `PM(Ip,Mq) = Ureg`, the data access over the PMD bus conflicts with the fetch of instruction n+5 (shown in the ***PM Access Conflict*** table). In this case the data access completes first. This is true of any program memory data access type instruction. This stall occurs only when the instruction to be fetched is not cached.

Table 4-19: PM Access Conflict

cycles	1	2	3	4	5	6	7
E2						pm(ip,mq)=ureg	
M4					pm(ip,mq)=ureg		n
M3				pm(ip,mq)=ureg		n	n+1
M2			pm(ip,mq)=ureg		n	n+1	n+2
M1		pm(ip,mq)=ureg		n	n+1	n+2	n+3
D2	pm(ip,mq)=ureg	n		n+1	n+2	n+3	n+4
D1		n+1		n+2	n+3	n+4	n+5
F4		n+2		n+3	n+4	n+5	n+6
F3		n+3		n+4	n+5	n+6	n+7
F2		n+4		n+5	n+6	n+7	n+8

Table 4-19: PM Access Conflict (Continued)

cycles	1	2	3	4	5	6	7
F1		n+5	n+5	n+6	n+7	n+8	n+9
cycle2:n+5 instruction fetch postponed							
cycle3:stall cycle							

Note that the instruction-conflict cache stores the fetched instruction (n+5), not the instruction requiring the program memory data access.

When the processor first encounters a bus conflict, it must stall for one cycle while the data is transferred, and then fetch the instruction in the following cycle. To prevent the same delay from happening again, the processor automatically writes the fetched instruction to the instruction-conflict cache. The sequencer checks the instruction cache on every data access using the PM bus. If the instruction needed is in the cache, a ***cache hit*** occurs. The instruction fetch from the cache happens in parallel with the program memory data access, without incurring a delay.

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

The ***Instruction Cache Architecture*** figure shows a block diagram of the 2-way set associative instruction cache. The instruction-conflict cache holds 32 instruction-address pairs. These pairs (or cache entries) are arranged into 16 (15-0) cache sets according to the four least significant bits (3-0) of their address. The two entries in each set (entry 0 and entry 1) have a valid bit, indicating whether the entry contains a valid instruction. The least recently used (LRU) bit for each set indicates which entry was not placed in the cache last (0 = entry 0 and 1 = entry 1).

The cache places instructions in entries according to the four LSBs of the instruction's address. When the sequencer checks for an instruction to fetch from the cache, it uses the four address LSBs as an index to a cache set. Within that set, the sequencer checks the addresses of the two entries as it looks for the needed instruction. If the cache contains the instruction, the sequencer uses the entry and updates the `LRU` bit (if necessary) to indicate the entry did not contain the needed instruction.

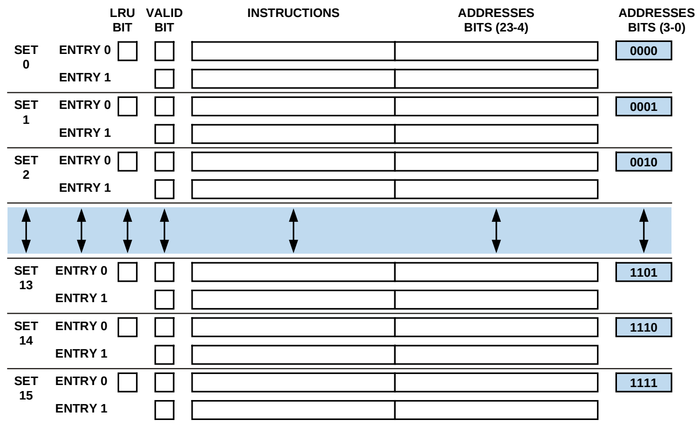

Figure 4-9: Instruction Cache Architecture

When the instruction-conflict cache does not contain a needed instruction, it loads a new instruction and address and places them in the least recently used entry of the appropriate cache set. The cache then toggles the LRU bit, if necessary.

Cache Invalidate Instruction

The FLUSH CACHE instruction allows programs to explicitly invalidate the cache content by clearing all valid bits. The execution of the FLUSH CACHE instruction is independent of the cache enable bit in the REGF_MODE2.CADIS register.

The FLUSH CACHE instruction has a 1 cycle instruction latency while executing from internal memory and has a 2 cycle instruction latency while executing from external memory. Using an instruction that contains a PM data access immediately following a FLUSH CACHE instruction is prohibited.

This instruction is required in systems using software overlay programming techniques. With these overlays, software functions are loaded via DMA during runtime into the internal RAM. Since the cache entries are still valid from any previous function, it is essential to flush all the valid cache entries to prevent system crashes.

Operating Modes

The following sections describe the instruction-conflict cache operating modes.

NOTE: After power-up and or reset, the cache content is not predicable in that it may contain valid/invalid instructions, be unfrozen and enabled. However, all LRU and valid bits are cleared. So after a processor power-up or reset, the cache performs only cache miss/cache entry until the same entry causes later hits.

Cache Restrictions

The following restrictions on instruction-conflict cache usage should be noted.

- If the REGF_MODE2.CAFRZ is set by instruction *n*, then this feature is effective from the n+2 instruction onwards. This results from the effect latency of the REGF_MODE2 register.
- When a program changes the instruction-conflict cache mode, an instruction containing a program memory data access must not be placed directly after a cache enable or cache disable instruction. This is because the

processor must wait at least one cycle before executing the PM data access. A program should have a NOP (no operation) or other non-conflicting instruction inserted after the cache enable or cache disable instruction.

Cache Disable

The cache disable bit (bit 4, REGF_MODE2.CADIS) directs the sequencer to disable the instruction-conflict cache (if 1) or enable the instruction-conflict cache (if 0).

Note that the FLUSH CACHE instruction has a 1 cycle instruction latency while executing next Instruction/data from internal memory and a 2 cycle instruction latency while executing next instruction/data from external memory.

Cache Freeze

The cache freeze bit (REGF_MODE2.CAFRZ) directs the sequencer to freeze the contents of the instruction-conflict cache (if 1) or let new entries displace the entries in the cache (if 0).

Freezing the cache prevents any changes to its contents-a cache miss does not result in a new instruction being stored in the instruction-conflict cache. Disabling the cache stops its operation completely-all instruction fetches conflicting with program memory data accesses are delayed. These functions are selected by the REGF_MODE2.CADIS (cache enable/disable) and REGF_MODE2.CAFRZ bits.

GPIO Flags

The SHARC+ core has a number of general-purpose I/O flags. The I/O flags provide direct instruction support for setting, resetting, or reading the state of these FLAG pins. The SHARC+ core Flag pins based on SHARC instruction set, (shown in the ***IF Condition Mnemonics*** table) are multiplexed with other peripheral pins in the Peripheral Port block. Refer to the General-Purpose Port chapter in the product related hardware reference manual or the product data sheet for the number of flag pins supported.

NOTE: Programs cannot change the output selects of the FLAGS register and provide a new value in the same instruction. Instead, programs must use two write instructions—the first to change the output select of a particular FLAG pin, and the second to provide the new value as shown.

```
bit set flags FLG2O; /* set flag2 as output */
bit clr flags FLG2;  /* set flag2 output low */
```

The FLAGS register is used to control all FLAGx pins. Based on FLAGS register effect latency and internal timings there must be at least 4 wait states in order to toggle the same flag correctly as shown in the following example. The total number of wait cycles can be more than four cycles, depending on the product. For total number of wait cycles, refer to the specific product data sheet.

```
bit tgl flags FLG2;
nop; nop; nop; nop;  /* wait 4 cycles */
bit tgl flags FLG2;
nop; nop; nop; nop;  /* wait 4 cycles */
bit tgl flags FLG2;
```

Conditional Instruction Execution

Conditional instructions provide many options for program execution which are discussed in this section. There are three types of conditional instructions:

- Conditional compute (ALU/Multiplier/Shifter)
- Conditional data move (reg-to-reg, reg-to-memory)
- Conditional branch (direct branch, indirect branch)

If the condition is evaluated as true, the operation is performed, if it is false, it gets aborted as shown in the example below.

```
R10 = R12-R13;
If LT R0=R1+R2; /* if ALU less than zero, do computation */
```

If an if-then-else construct is used, the else evaluates the inverse of the if condition:

```
R10 = R12-R13;
If LT CALL SUB, ELSE R0=R1+R2; /* do computation if condition is false */
```

The processor records status for the PEx element in the REGF_ASTATX and REGF_STKYX registers and the PEy element in the REGF_ASTATY and REGF_STKYY registers.

IF Conditions with Complements

Each condition that the processor evaluates has an assembler mnemonic. The condition mnemonics for conditional instructions appear in the ***IF Condition Mnemonics*** table. For most conditions, the sequencer can test both true and false (complement) states. For example, the sequencer can evaluate ALU equal-to-zero (EQ) and its complement ALU not-equal-to-zero (NE).

Note that since the IF condition is optional, and if the condition is omitted from the instruction, the condition is always true.

Table 4-20: IF Condition Mnemonics

Condition From	Description	True If	Mnemonic
ALU or exclusive access	ALU = 0 or exclusive access successful	AZ = 1[*1]	EQ
	ALU ≠ 0 or exclusive access failed	AZ = 0	NE
ALU	ALU > 0 or unordered	footnote[*2] footnote[*3]	GT
	ALU < 0	footnote[*4]	LT
	ALU ≥ 0 or unordered	footnote[*5] footnote[*6]	GE

Table 4-20: IF Condition Mnemonics (Continued)

Condition From	Description	True If	Mnemonic
	ALU ≤ 0	footnote[*7]	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
	Shifter bit FIFO overflow[*8]	SF = 1	SF
	Shifter bit FIFO not overflow	SF = 0	NOT SF
System Register	Bit test flag true	BTF = 1	TF
	Bit test flag false	BTF = 0	NOT TF
Flag 3-0 Input	Flag0 asserted	Flag0 = 1	FLAG0_IN
	Flag0 not asserted	Flag0 = 0	NOT FLAG0_IN
	Flag1 asserted	Flag1 = 1	FLAG1_IN
	Flag1 not asserted	Flag1 = 0	NOT FLAG1_IN
	Flag2 asserted[*9]	Flag2 = 1	FLAG2_IN
	Flag2 not asserted [*10]	Flag2 = 0	NOT FLAG2_IN
	Flag3 asserted	Flag3 = 1	FLAG3_IN
	Flag3 not asserted	Flag3 = 0	NOT FLAG3_IN
Loop Sequencer	Loop counter not expired	CURLCNTR 1	NOT LCE[*11]

*1 Instruction type 3d/14d support exclusive access (modifier EX)

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

*3 The unordered condition arises from floating-point instructions in which an operand is NaN (Not a Number). Note that by the inclusion of this case, the GT and GE conditions (GT and GE) differ from the IEEE 754 definitions.

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

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

*6 The unordered condition arises from floating-point instructions in which an operand is NaN (Not a Number). Note that by the inclusion of this case, the GT and GE conditions (GT and GE) differ from the IEEE 754 definitions.

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

*8 For ADSP-214xx processors and beyond.

*9 Support for conditional selection of PEx or PEy.

*10 Support for conditional selection of PEx or PEy.

*11 Does not have a complement.

DO/UNTIL Terminations Without Complements

Programs should use FOREVER and LCE to specify loop (DO/UNTIL) termination. A DO FOREVER instruction executes a loop indefinitely, until an interrupt or reset intervenes. There are some restrictions on how programs may use conditions in DO UNTIL loops. For more information, see Restrictions on Ending Loops.

Table 4-21: DO/UNTIL Termination Mnemonics

Condition From	Description	True If	Mnemonic
Loop Sequencer	Loop counter expired	REGF_CURLCNTR = 1	LCE
	Always false (Do)	Always	FOREVER

Operating Modes

The following sections describe the operating modes for conditional instruction execution.

Conditional Instruction Execution in SIMD Mode

Because the two processing elements can generate different outcomes, the sequencer must evaluate conditions from both elements (in SIMD mode) for conditional (IF) instructions and loop (DO/UNTIL) terminations. The processor records status for the PEx element in the REGF_ASTATX and REGF_STKYX registers and PEy element in theREGF_ASTATY and REGF_STKYY registers.

NOTE: Even though the processor has dual processing elements PEx and PEy, the sequencer does not have dual sets of stacks.

The sequencer has one PC stack, one loop address stack, and one loop counter stack. The status bits for stacks are in the REGF_STKYX register and are not duplicated in the REGF_STKYY register.

The processor handles conditional execution differently in SISD versus SIMD mode. There are a number of ways that conditionals differ in SIMD mode. These are described below and in the ***Conditional SIMD Execution Summary*** table.

- In conditional computation and data move (IF ... compute/move) instructions, each processing element executes the computation/move based on evaluating the condition in that processing element. See the Instruction Set Types chapter for coding information.

- In conditional branch (if ... jump/call) instructions, the program sequencer executes the jump/call based on a logical AND of the conditions in both processing elements.

- In conditional indirect branch (if ... pc, reladdr/Md, Ic) instructions with an ELSE clause, each processing element executes the ELSE computation/data move based on evaluating the inverse of the condition (NOT IF) in that processing element.
- Enhanced conditions for FLAG2_IN/NOT FLAG2_IN. These instruction conditions together with SISD/SIMD modes enables selective condition for PEx or PEy unit. For more information, see Conditional Execution by Selection of Processing Unit X or Y.

Table 4-22: Conditional SIMD Execution Summary

Conditional Operation		Conditional Outcome Depends On
Compute Operations		Executes in each PE independently depending on condition test in each PE
Register-to-register Move	Ureg/CUreg to Ureg/CUreg (from complementary pair[*1] to complementary pair)	Executes move in each PE (and/or memory) independently depending on condition test in each PE
	Ureg to Ureg/CUreg (from uncomplementary register to complementary pair)	Executes move in each PE (and/or memory) independently depending on condition test in each PE; *Ureg* is source for each move
	Ureg/CUreg to Ureg (from complementary pair to uncomplementary register)[*2])	Executes explicit move to uncomplementary universal register depending on the condition test in PEx only; no implicit move occurs
Register-to-memory Move	DAG post-modify	Executes memory move depending on OR'ing condition test on both PE's
	DAG pre-modify	Pre-modify operations always occur independent of the conditions
Branches and Loops		Executes in sequencer depending on AND'ing condition test on both PEs

*1 Complementary universal register pairs (`CUreg`) are registers with SIMD complements, include PEx/y data registers and USTAT1/2, USTAT3/4, ASTATx/y, STKYx/y, and PX1/2 Uregs.

*2 Uncomplementary registers are Uregs that do not have SIMD complements.

NOTE: SIMD must be disabled during bit FIFO operations.

Bit Test Flag in SIMD Mode

In SIMD mode, two independent bit tests can occur from individual registers as shown in the following example.

```
bit set mode1 PEYEN;
r2=0x80000000;
ustat1=r2;
bit TST ustat1 BIT_31; /* test bit 31 in ustat1/ustat2 */
if TF call SUB;        /* branch if both cond are true */
if TF r10=r10+1;       /* compute on any cond */
```

Conditional Compute

While in SIMD mode, a conditional compute operation can execute on both processing elements, either element, or neither element, depending on the outcome of the status flag test. Flag testing is independently performed on each processing element.

Conditional Data Move

The execution of a conditional (`IF`) data move (register-to-register and register-to/from-memory) instruction depends on three factors:

- The explicit data move depends on the evaluation of the conditional test in the PEx processing element.
- The implicit data move depends on the evaluation of the conditional test in the PEy processing element.
- Both moves depend on the types of registers used in the move.
- For conditional broadcast instructions, the condition depends on the PEx status only.

Conditional Execution by Selection of Processing Unit X or Y

An application can select which execution unit X or Y should be active while executing a conditional computation. The execution unit is selected using the `REGF_MODE1.SELPE` bit and using both conditional instructions `IF FLAG2_IN` and `IF NOT FLAG2_IN` (or IF PEX and `IF NOT PEX`). If the `REGF_MODE1.SELPE` bit is set then the instruction `IF FLAG2_IN` always performs the computation in PEx only, and the instruction `IF NOT FLAG2_IN` is always performed in PEy only.

In the following example the `REGF_MODE1.SELPE` bit is set and only the R0 value is updated and not the S0 value.

```
IF FLAG2_IN, R0=R1+R2;
```

In the next example the `REGF_MODE1.SELPE` bit is set only S0 value is updated and not the R0 value.

```
IF NOT FLAG2_IN, R0=R1+R2;
```

In this mode when the `REGF_MODE1.SELPE` bit set, the instruction `IF FLAG2_IN` and `IF NOT FLAG2_IN` should only be used for data move/compute operations where only one of the execution units need to be active as shown in the above example. These two conditional instructions, if used for any purpose other than data move/compute operations, will not have desired effect, because these condition is always true for other related instruction, such as:

`IF FLAG2_IN jump` or `IF NOT FLAG2_IN jump`, both execute irrespective of the state of FLAG2.

SISD versus SIMD Operating Mode

SISD If PEx tests true and if PEy tests false: Execution in PEX No operation in PEy

SIMD If PEx tests true and if PEy tests false: Execution in PEx. If PEy tests true and if PEx tests false: Execution in PEy

Table 4-23: SIMD Modes and IF FLAG2 Conditions Truth Table

MODE1.PEYEN	MODE1.SELPE	Condition	Execution PEx	Execution PEy
ON	ON	If FLAG2_IN	yes	no
ON	ON	If NOT_FLAG2_IN	no	yes
ON	OFF	If FLAG2_IN	yes	yes
ON	OFF	If NOT_FLAG2_IN	yes	yes
OFF	ON	If FLAG2_IN	yes	no
OFF	ON	If NOT_FLAG2_IN	no	no
OFF	OFF	If FLAG2_IN	yes	no
OFF	OFF	If NOT_FLAG2_IN	yes	no

Listings for Conditional Register-to-Register Moves

In this section the various register files move types are listed and illustrated with examples.

Listing 1 - Dreg/CDreg to Dreg/CDreg Register Moves/Swaps

When register-to-register swaps are unconditional, they operate the same in SISD mode and SIMD mode. If a condition is added to the instruction in SISD mode, the condition tests only in the PEx element and controls the entire operation. If a condition is added in SIMD mode, the condition tests in both the PEx and PEy elements separately and the halves of the operation are controlled as detailed in the ***Dreg/CDreg Register Moves Summary (SISD Versus SIMD)*** table.

Table 4-24: Dreg/CDreg Register Moves Summary (SISD Versus SIMD)

Mode	Instruction	Explicit Transfer Executed According to PEx	Implicit Transfer Executed According to PEy
SISD*1	IF condition Rx = Ry;	Rx loaded from Ry	None
	IF condition Rx = Sy;	Rx loaded from Sy	None
	IF condition Sx = Ry;	Sx loaded from Ry	None
	IF condition Sx = Sy;	Sx loaded from Sy	None
	IF condition Rx <-> Sy;	Rx loaded from Sy	Sy loaded from Rx
SIMD*2	IF condition Rx = Ry;	Rx loaded from Ry	Sx loaded from Sy
	IF condition Rx = Sy;	Rx loaded from Sy	Sx loaded from Ry
	IF condition Sx = Ry;	Sx loaded from Ry	Rx loaded from Sy
	IF condition Sx = Sy;	Sx loaded from Sy	Rx loaded from Ry
	IF condition Rx <-> Sy;	Rx loaded from Sy	Sy loaded from Rx

*1 In SISD mode, the conditional applies only to the entire operation and is only tested against PEx's flags. When the condition tests true, the entire operation occurs.

*2 In SIMD mode, the conditional applies separately to the explicit and implicit transfers. Where the condition tests true (PEx for the explicit and PEy for the implicit), the operation occurs in that processing element.

Listing 2 - Ureg/CUreg to Ureg/CUreg Register Moves

For the following instructions, the processors are operating in SIMD mode and registers in the PEx data register file are used as the explicit registers. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements is shown in the ***Register-to-Register Moves - Complementary Pairs*** table.

```
IF EQ R9 = R2;
IF EQ PX1 = R2;
IF EQ USTAT1 = R2;
```

Table 4-25: Register-to-Register Moves - Complementary Pairs

Condition in PEx	Condition in PEy	Result	
AZx	AZy	Explicit	Implicit
0	0	No data move occurs	No data move occur
0	1	No data move to registers r9, px1, and ustat1 occurs	s2 transfers to registers s9, px2 and ustat2
1	0	r2 transfers to registers r9, px1, and ustat1	No data move to s9, px2, and ustat2 occurs
1	1	r2 transfers to registers r9, px1, and ustat1	s2 transfers to registers s9, px2, and ustat2

Listing 3 - CUreg/Ureg to Ureg/CUreg Registers Moves

For the following instructions, the processors are operating in SIMD mode and registers in the PEy data register file are used as explicit registers. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements is shown in the ***Register-to-Register Moves - Complementary Pairs*** table.

```
IF EQ R9 = S2;
IF EQ PX1 = S2;
IF EQ USTAT1 = S2;
```

Table 4-26: Register-to-Register Moves - Complementary Pairs

Condition in PEx	Condition in PEy	Result	
AZx	AZy	Explicit	Implicit
0	0	No data move occurs	No data move occur
0	1	No data move to registers r9, px1, and ustat1 occurs	r2 transfers to registers s9, px2 and ustat2
1	0	s2 transfers to registers r9, px1, and ustat1	No data move to s9, px2, or ustat2 occurs
1	1	s2 transfers to registers r9, px1, and ustat1	r2 transfers to registers s9, px2, and ustat2

Listing 4 - Ureg to Ureg/CUreg Register Moves

In this case, data moves from an uncomplementary register (`Ureg` without a SIMD complement) to a complementary register pair. The processor executes the explicit move depending on the evaluation of the conditional test in the PEx processing element. The processor executes the implicit move depending on the evaluation of the conditional test in the PEy processing element. In each processing element where the move occurs, the content of the source register is duplicated in the destination register.

Note that while `REGF_PX1` and `REGF_PX2` registers have complements, the `REGF_PX` register has no complementary register.

For the following instruction the processors are operating in SIMD mode. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements is shown in the ***Uncomplementary-to-Complementary Register Move*** table.

```
IF EQ R1 = PX;
```

Table 4-27: Uncomplementary-to-Complementary Register Move

Condition in PEx	Condition in PEy	Result	
AZx	AZy	Explicit	Implicit
0	0	r1 remains unchanged	s1 remains unchanged
0	1	r1 remains unchanged	s1 gets px value
1	0	r1 gets px value	s1 remains unchanged
1	1	r1 gets px value	s1 gets px value

Listing 5 - Ureg/CUreg to Ureg Register Moves

In this case data moves from a complementary register pair to an uncomplementary register. The processor executes the explicit move to the un complemented universal register, depending on the condition test in the PEx processing element only. The processor does not perform an implicit move.

For all of the following instructions, the processors are operating in SIMD mode. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements for all of the example code samples are shown in the ***Complementary-to-Uncomplementary Register Move*** table.

```
IF EQ R1 = PX;
```

Uncomplementary register to DAG move:

```
if EQ m1 = PX;
```

DAG to uncomplementary register move:

```
if EQ PX = m1;
```

For more information, see the Register Files chapter.

Note that the `REGF_PX1` and `REGF_PX2` registers have complements, but `REGF_PX` as a register is uncomplementary.

DAG to DAG move:

```
if EQ m1 = i15;
```

Complementary register to DAG move:

```
if EQ i6 = r9;
```

In all the cases described above, the behavior is the same. If the condition in PEx is true, then only the transfer occurs.

Table 4-28: Complementary-to-Uncomplementary Register Move

Condition in PEx	Condition in PEy	Result	
AZx	**AZy**	**Explicit**	**Implicit**
0	0	px remains unchanged	No implicit move
0	1	px remains unchanged	No implicit move
1	0	r1 40-bit explicit move to px	No implicit move
1	1	r1 40-bit explicit move to px	No implicit move

Listing 6 - UREG to UREG Register Moves

In this case data moves from an uncomplementary register to an uncomplementary register. The processor executes the explicit move, depending on the condition test in the PEx or PEy processing. The processor does not perform an implicit move.

```
if lt tperiod = dm(i3, m3);
```

Listings for Conditional Register-to-Memory Moves

Conditional post-modify DAG operations update the DAG register based on OR'ing of the condition tests on both processing elements. Actual data movement involved in a conditional DAG operation is based on independent evaluation of condition tests in PEx and PEy. Only the post-modify update is based on the OR'ing of these conditional tests.

NOTE: Conditional pre-modify DAG operations behave differently. The DAGs always pre-modify an index, independent of the outcome of the condition tests on each processing element.

Listing 1 - Dreg to Memory

For this instruction, the processors are operating in SIMD mode, a register in the PEx data register file is the explicit register, and `I0` is pointing to an even address in internal memory or external memory. Indirect addressing is shown in the instructions in the example. However, the same results occur using direct addressing. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements is shown in the ***Register-to-Memory Moves-Complementary Pairs (PEx Explicit Register)*** table.

```
IF EQ DM(I0,M0) = R2;
```

Table 4-29: Register-to-Memory Moves-Complementary Pairs (PEx Explicit Register)

Condition in PEx	Condition in PEy	Result	
AZx	AZy	Explicit	Implicit
0	0	No data move occurs	No data move occurs
0	1	No data move occurs from r2 to location I0	s2 transfers to location (I0+n*1)
1	0	r2 transfers to location I0	No data move occurs from s2 to location (I0+n1)
1	1	r2 transfers to location I0	s2 transfers to location (I0+n1)

*1 In NW space n = 1, in SW space n = 2, in BW space, n = 4.

Listing 2 - CDreg to Memory

For the following instruction, the processors are operating in SIMD mode, a register in the PEy data register file is the explicit register and `I0` is pointing to an even address in internal memory. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements is shown in the ***Register-to-Memory Moves - Complementary Pairs (PEy Explicit Register)*** table.

```
IF EQ DM(I0,M0) = S2;
```

Table 4-30: Register-to-Memory Moves - Complementary Pairs (PEy Explicit Register)

Condition in PEx	Condition in PEy	Result	
AZx	AZy	Explicit	Implicit*1
0	0	No data move occurs	No data move occurs
0	1	No data move occurs from s2 to location I0	r2 transfers to location I0+n
1	0	s2 transfers to location I0	No data move occurs from r2 to location I0 + n
1	1	s2 transfers to location I0	r2 transfers to location I0 + n

*1 In NW space n = 1, in SW space n = 2, in BW space, n=4.

Listing 3 - Dreg/CDreg to SMMR Memory Space

For the following instructions the processors are operating in SIMD mode and the explicit register is either a PEx register or PEy register. `I0` points to SMMR memory space. This example shows indirect addressing. However, the same results occur using direct addressing.

```
IF EQ DM(I0,M0) = R2;
IF EQ DM(I0,M0) = S2;
```

Listing 4 - Ureg to SMMR Memory Space

In the case of memory-to-DAG register moves, the transfer does not occur when both PEx and PEy are false. Otherwise, if either PEx or PEy is true, transfers to the DAG register occur. For example:

```
if EQ m13 = dm(i0,m1);
```

NOTE: Conditional data moves from a complementary register pair to an uncomplementary register with an access to IOP memory space results in unexpected behavior and should not be used.

Conditional Branches

The processor executes a conditional branch (`JUMP` or `CALL` with `RTI/RTS`) or loop (`DO/UNTIL`) based on the result of AND'ing the condition tests on both PEx and PEy. A conditional branch or loop in SIMD mode occurs only when the condition is true in PEx and PEy.

Using complementary conditions (for example `EQ` and `NE`), programs can produce an OR'ing of the condition tests for branches and loops in SIMD mode. A conditional branch or loop that uses this technique must consist of a series of conditional compute operations. These conditional computes generate `NOP`s on the processing element where a branch or loop does not execute. For more information on programming in SIMD mode, see the Instruction Set Types and Computation Types chapters.

IF Conditional Branch Instructions

The `IF` conditional direct branch instruction is available in Type 8 instruction. The `IF` conditional indirect branch instruction is available in the Type 9, 10, and 11 instructions. The instructions are shown in the ***IF Conditional Branch Execution (SISD mode)*** and ***If Conditional Branch Instruction (SIMD Mode)*** tables.

Table 4-31: IF Conditional Branch Execution (SISD mode)

Conditional Test	Execution for Instruction Types 8-11
0 (false)	IF not exe
1 (true)	IF exe

Table 4-32: If Conditional Branch Instruction (SIMD Mode)

Conditional Test		Execution for Instruction Types 8-11
PEx	PEy	
0 (false)	0 (false)	IF not exe
0 (false)	1 (true)	IF not exe
1 (true)	0 (false)	IF not exe
1 (true)	1 (true)	IF exe

IF Then ELSE Conditional Indirect Branch Instructions

The conditional `IF` then `ELSE` construct for indirect branch instructions is available in the Type 9, 10, and 11 instructions. The instructions are shown in the ***IF then ELSE Conditional Branch Execution (SISD mode)*** and ***IF Then ELSE Conditional Branch Instruction (SIMD Mode) tables.***

Table 4-33: IF then ELSE Conditional Branch Execution (SISD mode)

Conditional Test	Execution for Instruction Types 9-11	
0 (false)	IF not exe	ELSE exe
1 (true)	IF exe	ELSE not exe

Table 4-34: IF Then ELSE Conditional Branch Instruction (SIMD Mode)

Conditional Test		Execution for Instruction Types 9-11	
PEx	PEy		
0 (false)	0 (false)	IF not exe	ELSE PEx exe - PEY exe
0 (false)	1 (true)	IF not exe	ELSE PEx exe - PEY not exe
1 (true)	0 (false)	IF not exe	ELSE PEx not exe - PEY exe
1 (true)	1 (true)	IF exe	ELSE PEx not exe - PEY not exe

For more information and examples, see the following instruction reference pages in the Instruction Set Types chapter.

- *Type 8a ISA/VISA (cond + branch)*
- *Type 9a ISA/VISA (cond + Branch + comp/else comp)*
- *Type 10a ISA (cond + branch + else comp + mem data move)*
- *Type 11a ISA/VISA (cond + branch return + comp/else comp) and Type 11c VISA (cond + branch return)*

IF Conditional Branch Limitations in VISA

Type 10 instructions are the most infrequently used instructions in the Instruction Set Architecture:

```
/* Template: */
IF COND JUMP (Md, Ic), ELSE compute, DM(Ia, Mb) = dreg ;
```

To make maximum use of available opcode combinations, the SHARC+ core uses the Type 10 instruction opcode to encode a simpler and more commonly used compute instructions such as:

```
Rm = Rn + Rm;
```

NOTE: Code generated by the CrossCore Embedded Studio C compiler does not use the Type 10 instruction.

If assembly code containing Type 10 instructions is run through the code generation tools, the assembler issues an error message stating that a Type 10 instruction is not supported while in VISA short word space.

Pipeline Flushes and Stalls

The SHARC+ core uses pipeline flush and pipeline stalls to ensure correct and efficient program execution. It is helpful for programmers to be aware of different scenarios that result in flushes or stalls of pipeline stages.

The sequencer uses pipeline stalls in the following situations:

- Stalls occur in case of structural hazards. Such stalls are incurred when different instructions at various stages of the instruction pipeline attempt to use the same processor resources simultaneously. For example, when processor issues data access on PM bus, it conflicts with instruction request being issued by the sequencer on the same bus.
- Stalls occur in case of data and control hazards. Such stalls are incurred when an instruction attempts to read a value from a register or from a condition flag that has been updated by an earlier instruction, before the value becomes available. For example, an index register based data address generation, which happens in early stage of pipeline, stalls when index register is being loaded in previous instruction.
- Stalls occur in some cases to achieve high performance when the processor executes a certain sequence of instructions. For example, when both the input operands are forwarded to the multiplier from previous compute instruction, this scenerio causes stall to accommodate additional operations.
- Stalls occur in some cases to retain effect latency. These cases provide operation that is compatible with earlier SHARC processors. One such example is 64-bit compute in the newer versions of SHARC+ core.

The sequencer uses pipeline flushes when the processor branches to new location due to an interrupt, a jump, a call, a loop-abort, or other branch situations. These situation leave some extra instructions in the pipeline (from previous flow) that must be flushed.

For complete information on stalls, refer to the Engineer-to-Engineer Note ***EE-375: Migrating Legacy SHARC to ADSP-SC58x/2158x SHARC+ Processors*** on the Analog Devices web site.

Stalls Related to Memory Access

Table 4-35: Stalls Related to Memory Access

Details	Example	Stall type	SHARC+ Stalls
Conflict cache miss on PM data access	`r0 = pm(Addr);`	Structural	1
Two accesses on same block in same cycle	`r0 = dm(Block0-addr1);` background DMA access to Block0	Structural	1
Conditional store –to- any load	`If eq DM(A) = Fz;` Fa = DM(A/B)	Timing	1

Stalls Related to Compute Operations

Table 4-36: Stalls Related to Compute Operations - Non 64-bit Floating Point Computations

Details		Example	Stall type	SHARC+ Stalls
1	Data forwarding to compute operation			
	Floating point compute/multiply operationto-next compute dependency	`Fx = PASS Fy;` `Fz = Fx + Fa;`	Data dependence	1

	If previous instruction is conditional fixed point compute or conditional register read and condition set is just before	`Rx = PASS Ry;`
		`IF eq Rz = Ra + Rb;`
		`Fc = Fz + Fd;`		

	ASTATx/y register update to- carry or overflow dependent instruction	`ASTATx = DM(..);` `Rx = Ry + Rz + CI;`		
2	Dual forwarding to multiplier			

	Dual forwarding to multiply operation from N –2 to Nth location.	`Fx=Fa+Fb, Fy=FaFb;`
		`[unrelated instr];`
		`Fz = Fx * Fy;`
			Timing	1

	Dual forwarding to multiply operation from N –1 to Nth location	`Fx=Fa+Fb,Fy=FaFb;` `Fz = Fx * Fy;`		2
3	Floating point multiply operation to next fixed point ALU	`Fz = Fx * Fy;` `Ra = Rz + Rb;`		0

Table 4-37: Stalls Related to Compute Operations - 64-bit Floating Point Computations with Data Forwarding from N-2/N-1 to Nth Instruction

Case	Example	Stall type	SHARC+ Stalls
Data forwarding to/from any compute, load and 64-bit compute	`Fx:y = .;` `Fa:b = Fx:y + 1;`	Data dependence	2

Data forwarding to/from any compute, load and 64-bit compute	`Fx:y = .;`
	`[Unrelated instr];`
	`Fa:b = Fx:y + 1;`
			1

Stalls Related to DAG Operations

Table 4-38: Stalls Related to DAG Operations

#	Details	Example	Stall type	SHARC+ Stalls*
1	Unconditional DAG register load -to- use	`Ix = DM(..);` `DM(Ix…) = …;`	Data dependence	4

2	Conditional DAG register load (with condition set just before) -to- use	`Rx = PASS Ry;`
		`IF eq Ix = DM(..);`
		`DM(Ix…) = …;`
				5

3	Condition set -to- conditional post modify DAG operation on Ix -to- any DAG operation on same Ix	`Rx = PASS Ry;`
		`IF eq DM(Ix,..);`
		`DM(Ix…) = …;`
				5

4	Load of DAG register with immediate value –to- use	`Ix = [IMM VALUE];` `DM(Ix..)..`		0

*An additional stall occurs if :

- the condition that is set happens through a write to the `REGF_ASTATX` register, and
- a register load is used with the sign extension modifier.

Stalls and Flushes Related to Branch and Prediction Operations

These stalls and pipeline flushes include those related to jumps, calls, returns, rframe, and cjump.

In most of the cases, any branch instruction flushes the pipeline and some cycles are lost. Branch prediction attempts to minimize the loss of cycles. The following table describes the number of lost cycles when BTB is disabled or the branch entry is not present in BTB (for example, a BTB miss). If the branch is the one with a delay slot of two instructions, the number of flushed instruction is lesser by 2.

Table 4-39: Stalls and Flushes Related to Branch and Prediction Operations - Jump and Pass

#	Details	Example	Stall type	SHARC+ Stalls (non-delayed vs. delayed)
1	Unconditional branch	`Jump(My,Ix);`	Pipeline flush	6/4**
2	Condition set -to- conditional branch	`Rx = PASS Ry;` `IF eq Jump(My,Ix);`	Control dependence	11/9*

In addition to the above stalls, there are other data and control dependence stalls in relation to branch instructions. The cycles in the table below are additive to the cycles incurred due to other reasons as described in the tables above in this section.

Table 4-40: Stalls and Flushes Related to Branch and Prediction Operations - Cjump, Rframe, Pass, and Jump

#	Details	Example	Stall type	SHARC+ Stalls
1	CJUMP/RFRAME –to- use of I6	`CJUMP;` `DM(I6,..) = …;`	Data dependence	6
2	CJUMP/RFRAME –to- read of I6/7	`RFRAME;` `R0 = I6;`		6
3	Unconditional DAG register load -to- use in indirect branch	`Ix = DM(..);` `Jump(Ix…) = …;`		4†

4	Conditional DAG register load (with condition set just before) -to- use in indirect jump	`Rx = PASS Ry;`
		`IF eq Ix = DM(..);`
		`Jump(Ix…) = …;`
			Data and control dependence	5*†

† One additional cycle of stall if register load is used with sign extension modifier

* One additional cycle of stall if condition set happens through write to the REGF_ASTATX or REGF_ASTATY register.

** As an exception, RTI (DB) and RTI causes 7 cycles of pipeline flush.

***Total of two cycles of stall for multiplier generated conditions

Table 4-41: Stalls and Flushes Related to Branch and Prediction Operations - Hardware Loops

Case	Example	Stall type	SHARC+ Stalls
On termination of E2 active and short loops	`LCNTR = 4, DO (PC,2) UNTIL LCE;`	Pipeline flush	11
On termination of arithmetic condition based loops	`DO (PC,2) UNTIL EQ;`		11
Write to CCNTR to LCE based instruction	`CCNTR = 4;` `If not LCE R0 = R1;`	Timing	1
Start of 1,2,4 instruction loop	`LCNTR = 4, DO (PC,2) UNTIL LCE;`		0

Table 4-42: Stalls and Flushes Related to Branch and Prediction Operations - Miscellaneous

Case	Example	Stall type	SHARC+ Stalls
During the execution of first four instructions of an unrolled loop, when COF (change of flow) is at Nth position in loop from top, where N = 0-3		Loop state machine	4-N

Table 4-42: Stalls and Flushes Related to Branch and Prediction Operations - Miscellaneous (Continued)

Case	Example	Stall type	SHARC+ Stalls
If RTS/RTI is returning to a loop at "Last-Addr"-N, where N = 0-3		Data dependence	4-N
Jump with loop abort	`Jump <Target> (LA);`		4
Target/next-to-target of CALL/RTS/RTI itself being an RTS/RTI			3
Target of CALL/RTS/RTI itself being a Jump			1
Loop-stack modification followed by RTS/RTI/Jump			5

SREG or SYSCTL update to N+2 instruction	`Bit set MODE1 CBUFEN;`
	`[Instr];`
	`DM(I0…);`
		Control dependence	5

Bit set/clear MODE1 PEYEN to N+2 instruction	`Bit set MODE1 PEYEN;`
	`[Instr];`
	`DM(I0…);`
			0

Stalls Related to Data Move Operations

Table 4-43: Stalls Related to Data Move Operations

Case	Example	Stall type	SHARC+ Stalls
Floating point compute or any multiplier operation followed by move of the result to any register outside the relevant PE	`F1 = F2+F3;` `USTAT1 = F1;`	Timing	1

Condition set followed by a conditional load of a DAG reg followed by move of that reg to any other Ureg	`R0 = R1 + 1;`
	`IF EQ I0=PM(<Addr>);`
	`USTAT1=I0;`
			1

Access of any Timer core register	`TCOUNT = USTAT1;`		1
Read of these registers: IRPTL, IMASKP, MODE1STK, LPSTK, CCNTR, LCNTR, PCSTK, PCSTKP, MODE1, FLAGS, ASTATx/y, STKYx/y, FADDR, DADDR	`R0 = IRPTL;`		1
Write Followed by Read of these registers: IMASK, USTAT, MMASK, MODE2	`USTAT1=DM(<Addr>);` `R0= USTAT1;`		1
Read and write of any CMMR	`R0 = dm(SYSCTL);`		0-4

Table 4-44: DAG Register Loading for SHARC Product Families

Model	DAG Stall Condition	Stall Examples	Stall Cycles
ADSP-2106x[*1]	Any DAG registers in same DAG	i0=>i5, b3=>b3; m12=>l15	1
ADSP-2116x[1]	Any same DAG register number in same DAG	i0=>b0, b3=>b3; m12=>l12	1
ADSP-2126x[1]	Any same DAG register number in same DAG (except M regs, stall only if same register is reused)	i0=>b0, b3=>b3; i10=>l10, (m2=>l2 no stall)	1
ADSP-2136x[*2]ADSP-2137x[2] ADSP-214xx[2] ADSP-SC58x			2

*1 Three stage pipeline. These products are not included in this manual.

*2 Five stage pipeline. These products are not included in this manual.

Core Event Controller Exceptions

The SHARC+ core uses the system bus infrastructure that appears on many processors from Analog Devices. In this bus architecture, external interrupts are managed by the System Event Controller (SEC).

NOTE: If porting code from previous SHARC processors, modify the code to interface with the SEC and remove existing support for external interrupts.

Table 4-45: Core Event Controller Exceptions

Interrupt Source	Interrupt Condition	Return Register	Return Instruction	IVT level
HW stack	PC stack overflow	STKYx	RTI	5, SOVFI
HW stack	Loop stack overflow	STKYx	RTI	5, SOVFI
HW stack	Status stack overflow	STKYx	RTI	5, SOVFI
HW stack	Restricted instruction sequence	N/A	RTI	20, RINSEQI
L1 Memory	Parity error[*1]	N/A	RTI	3, PARI
Sequencer	Illegal opcode detect[*2]	N/A	RTI	4, ILOPI
System	System event interrupt[*3]	SEC_ID	RTI	15, SECI
SW	Bit set IRPTL SFT0I	N/A	RTI	28, SFT0I
SW	Bit set IRPTL SFT1I	N/A	RTI	29, SFT1I
SW	Bit set IRPTL SFT2I	N/A	RTI	30, SFT2I
SW	Bit set IRPTL SFT3I	N/A	RTI	31, SFT3I

*1 See Parity Error Detection for L1 Accesses.

*2 See Illegal Opcode Error Detection for Instruction Fetch.

*3 See the ADSP-SC58x SHARC Processor Hardware Reference.

Hardware Stack Exceptions

The hardware stack (status stack, loop stack and PC stack) conditions trigger a maskable interrupt shown in the ***Hardware Stack Interrupt Overview*** table. The overflow and full flags provide diagnostic aid only. Programs should not use these flags for runtime recovery from overflow. The empty flags can ease stack saves to memory. Programs can monitor the empty flag when saving a stack to memory to determine when the processor has transferred all the values. For a complete interrupt list, see the Interrupt Priority and Vector Table.

HW Loop Stack Exceptions (RINSEQI)

Because of re-timing in the 11 stage pipeline, the following are situations when the restricted instruction sequence interrupt (RINSEQI) is generated.

1. In a nested loop, where the outer loop is an arithmetic loop and the inner loop is a counter based loop. Also the LADDR of the inner loop coincides with LADDR-2 of the outer loop. Then the LADDR of the inner counter based loop cannot be a branch instruction.
2. Last five instructions of an Arithmetic Loop cannot be a delayed branch.

Software Interrupts

Software interrupts (or programmed exceptions) are instructions which explicitly generate an exception. The interrupt overview is shown in the ***Software Interrupt Overview*** table. For a complete interrupt list, see the Interrupt Priority and Vector Table.

The `REGF_IRPTL` register provides four software interrupts. When a program sets the latch bit for one of these interrupts (`REGF_IRPTL.SFT0I`, `REGF_IRPTL.SFT1I`, `REGF_IRPTL.SFT2I`, or `REGF_IRPTL.SFT3I`), the sequencer services the interrupt, and the processor branches to the corresponding interrupt routine. Software interrupts have the same behavior as all other maskable interrupts. For more information, see the Core Interrupt Control appendix.

If programs force an interrupt by writing to a bit in the `REGF_IRPTL` register, the processor recognizes the interrupt in the following cycle, and eleven cycles of branching to the interrupt vector follow the recognition cycle.

Interrupt Priority and Vector Table

There are 32 core interrupts supported by SHARC+ core. The various interrupts caused by external events on previous SHARC processors have been replaced by the single SECI interrupt. The relative priorities of the remaining interrupts are unchanged, except for CB7I. As the interrupt numbers are different the vector offsets are also changed from previous SHARC processors.

CB7I is used to trap software stack overflow. Having this interrupt at a high priority enables stack overflow to be detected in high priority handlers.

NOTE: Any reset asserted to SHARC+ core is not honored if execution control is in Emulation space. Also the reset asserted is not latched if the core is in Emulation space, so if the program wants to reset the core, then the core should be first brought out of Emulation space and then reset should be asserted.

NOTE: Interrupt numbers 3, 4, 8, 15, 20 have been added for the SHARC+ core.

Table 4-46: Interrupt Priority and Vectors

Interrupt Number	Vector Offset	Interrupt Name	Function
0	0x00	EMUI	Emulator (HIGHEST PRIORITY)
1	0x04	RSTI	Reset
2	0x08	Reserved	Reserved
3	0x0C	PARI	L1 Parity Error
4	0x10	ILOPI	Illegal opcode detected
5	0x14	CB7I	Software stack (Circular Buffer 7) Overflow
6	0x18	IICDI	Unaligned LW/BW access + unintentional CMMR/SMMR access
7	0x1C	SOVFI	Status loop or mode stack overflow; or PC stack full
8	0x20	ILADI	Illegal Address Space detected
9	0x24	IIR2 (For ADSP-SC59x)	IIR2 Interrupt (For ADSP-SC59x)
		Reserved (ADSP-SC57x, ADSP-SC58x, and ADSP-2156x)	Reserved (ADSP-SC57x, ADSP-SC58x, and ADSP-2156x)
10	0x28	IIR3 (For ADSP-SC59x)	IIR3 Interrupt (For ADSP-SC59x)
		Reserved (ADSP-SC57x, ADSP-SC58x, and ADSP-2156x)	Reserved (ADSP-SC57x, ADSP-SC58x, and ADSP-2156x)
11	0x2C	TMZHI	Core Timer (high priority option)
12	0x30	BKPI	User Hardware Breakpoint

Table 4-46: Interrupt Priority and Vectors (Continued)

Interrupt Number	Vector Offset	Interrupt Name	Function
13	0x34	FIR (For ADSP-2156x)	FIR Interrupt (For ADSP-2156x)
		FIR0 (For ADSP-SC59x)	FIR0 Interrupt (For ADSP-SC59x)
		Reserved (For ADSP-SC57x and ADSP-SC58x)	Reserved (For ADSP-SC57x and ADSP-SC58x)
14	0x38	IIR (For ADSP-2156x)	IIR Interrupt (For ADSP-2156x)
		IIR0 (For ADSP-SC59x)	IIR0 Interrupt (For ADSP-SC59x)
		Reserved (For ADSP-SC57x and ADSP-SC58x)	Reserved (For ADSP-SC57x and ADSP-SC58x)
15	0x3C	SECI	System event controller interrupt
16	0x40	IIR1 (For ADSP-SC59x)	IIR1 Interrupt (For ADSP-SC59x)
		Reserved (ADSP-SC57x, ADSP-SC58x, and ADSP-2156x)	Reserved (ADSP-SC57x, ADSP-SC58x, and ADSP-2156x)
17	0x44	Reserved	Reserved
18	0x48	Reserved	Reserved
19	0x4C	Reserved	Reserved
20	0x50	RINSEQI	Restricted Instruction Sequence
21	0x54	CB15I	Circular Buffer 15 Overflow
22	0x58	TMZLI	Core Timer (Low Priority Option)
23	0x5C	FIXI	Fixed-point overflow exception
24	0x60	FLTOI	Floating-point overflow exception
25	0x64	FLTUI	Floating-point underflow exception
26	0x68	FLTII	Floating-point invalid exception
27	0x6C	EMULI	Emulator low priority interrupt
28	0x70	SFTOI	User software interrupt 0

Table 4-46: Interrupt Priority and Vectors (Continued)

Interrupt Number	Vector Offset	Interrupt Name	Function
29	0x74	SFT1I	User software interrupt 1
30	0x78	SFT2I	User software interrupt 2
31	0x7C	SFT3I	User software interrupt 3 (LOWEST PRIORITY)

Internal Interrupt Vector Table Location

The default location of the SHARC processor's interrupt vector table (IVT) depends on control bits: `CMMR_SYSCTL.IIVT` and `CMMR_SYSCTL.EIVT` bits determine the IVT location. The following table summarizes the selection of IVT location.

Table 4-47: IVT Location Selections

EIVT	IIVT	IVT location
1	0	L3
0	0	L2 ROM (default)
0	1	L1

Exact address in each memory can be found in detailed address map. Reset routine address is always fixed for a given product. This address is also available in detailed address map.

The internal interrupt vector table `CMMR_SYSCTL.IIVT` bit in the register overrides the default placement of the vector table. If `CMMR_SYSCTL.IIVT` is set (=1), the interrupt vector table starts at internal RAM regardless of the booting mode. If `CMMR_SYSCTL.IIVT` is cleared (=0), the interrupt vector table starts in the L2 ROM.

For information about processor booting, see the processor-specific hardware manual.

Core Interrupt Registers

All core interrupts are programmed through the `REGF_IRPTL`, `REGF_IMASK`, and `REGF_IMASKP` registers. The bit for each interrupt in these registers is indexed by interrupt number.

NOTE: Unlike previous SHARC processors, the SHARC+ core does not have an LIRPTL register.

All Interrupts Automatically Push Status

On the SHARC+ core, all interrupts push the status stack.

NOTE: This functionality is an extension of the push of the status stack which was provided by only the IRQ and core timer interrupts on previous SHARC processors.

The sequencer automatically pushes the current value of the `REGF_ASTATX`, `REGF_ASTATY`, and `REGF_MODE1` registers on the status stack. Then, the sequencer clears the bits in the `REGF_MODE1` that are set in the REGF_MMASK register, before branching to an interrupt vector. If the REGF_MMASK.IRPTEN bit is cleared by this operation, interrupts are disabled globally before another (higher priority) interrupt can preempt the current interrupt.

The JUMP(CI) and RTI instructions always automatically pop the status stack.

Self-Nesting Mode for System Event Controller Interrupt (SECI)

The SHARC+ core provides a mode bit, REGF_MODE2.SNEN. This bit enables self-nesting interrupt mode for the SECI interrupt only. Self-nesting operation also uses the nesting bit, REGF_MODE1STK.NESTM.

1. The REGF_MODE2.SNEN bit enables self-nesting for SECI only.
   - When REGF_MODE2.SNEN =1, the REGF_IMASKP.SECI bit can latch even when it is currently being serviced.
   - If REGF_MODE1.IRPTEN =1, REGF_MODE1.NESTM =1 and REGF_MODE2.SNEN =1 and the REGF_IMASKP.SECI bit is currently being serviced, the REGF_IMASKP.SECI bit is not masked but lower priority interrupts are. If a higher priority interrupt interrupts the REGF_IMASKP.SECI bit then it becomes masked.
2. The REGF_MODE1.NESTM and REGF_MODE1STK.NESTM bits control whether the REGF_IMASKP.SECI bit is cleared and controls whether the interrupts are implicitly masked in NESTM mode.
   - When REGF_MODE2.SNEN =1, on vectoring to the SECI ISR, after automatically pushing the previous value of the REGF_MODE1 register, the REGF_MODE1.NESTM bit is automatically set.
   - On executing RTI, when the current interrupt is SECI and the REGF_MODE1STK.NESTM bit is set, the REGF_IMASKP register and interrupt mask are not changed. Otherwise, the REGF_IMASKP register and the masked interrupts are modified as normal. After the REGF_MODE1STK register is tested, the RTI instruction pops the mode stack as normal.

The interrupts masked implicitly in NESTM mode can always be calculated from the REGF_IMASKP register and REGF_MODE2.SNEN bit. When REGF_MODE2.SNEN =1 and the lowest numbered interrupt set in the REGF_IMASKP register is SECI, all interrupts down to but not including SECI are masked. Otherwise, all interrupts down to and including the lowest numbered bit set in the REGF_IMASKP register are masked, unless no bit is set in the REGF_IMASKP register, indicating no interrupts are implicitly masked.

The global interrupt enable bit, REGF_MODE1.IRPTEN, and interrupt nesting enable bit, REGF_MODE1.NESTM, take precedence over REGF_MODE2.SNEN. The SECI ISR is only interrupted by another incoming SECI if REGF_MODE1.IRPTEN =1, REGF_MODE1.NESTM =1, and REGF_MODE2.SNEN =1.

Table 4-48: SNEN and NESTM Combination and its Effect

SNEN	NESTM	Effect	
		SECI Self Nesting*1	Higher Priority Interrupt Nesting
0	0	NO	NO
0	1	NO	YES
1	0	YES	NO
1	1	YES	YES

*1 SECI is not stored in IRPTL if already in an SEC ISR. So to avoid missing any SECI when already in an SEC ISR, self-nesting of SECI must be enabled by setting SNEN bit in MODE2.

Interrupt Control Latencies

The latency for changes to the REGF_IMASK to take effect is up to one cycle.

The latency for changes to the REGF_MODE1.IRPTEN is up to one cycle.

The latency for changes to the REGF_MODE1.NESTM is up to one cycle.

The latency for changes to the REGF_MODE2.SNEN is up to one cycle.

Hardware Status Stack Access Register

It is possible to read and write the MODE1 value at the top of the status stack using the REGF_MODE1STK universal register.

NOTE: The REGF_MODE1STK register is not available on previous SHARC processors.

This makes it possible to save the top of the status stack to memory without enabling any undesired modes when the REGF_MODE1 register is popped.

REGF_MODE1STK can be an operand of the Type 18 register bit manipulation instructions.

For example, this code sequence copies the top of the status stack to the software stack without re-enabling interrupts as would be the case had the value pushed on entry to an interrupt handler been popped.

```
DM(I7,M7) = MODE1STK;       //save original MODE1
MODE1STK = RND32|TRUNC|NESTM;
POP STS;                    // MODE1 set to known safe value
DM(I7,M7) = ASTATX;         // save original ASTATX
DM(I7,M7) = ASTATY;         // save original ASTATY
```

Core Interface to SEC

The interface to the System Exception Controller (SEC) is similar to the interface used on other processors from Analog Devices. A core memory mapped register, CEC_SID, is provided. This register is read within the SECI interrupt handler to identify the external event that caused the interrupt. It is also written, with any value, to acknowledge the interrupt and allow the SEC to raise a new interrupt.

When CEC_SID is read by the core, the SID value from the SEC is returned. When it is written, it sends the ACK signal to the SEC.

Example SEC Handler Using Pseudo Self-Nesting

This handler, or something like it, must be added to the verification suite to ensure system interrupts can be serviced at a higher priority than low priority core interrupts while being preemptable by higher priority core and system interrupts. The system exception controller prioritizes external interrupts destined for the core. SECI is raised again if a higher priority interrupt than the one being serviced comes in.

As SHARC+ processors do not allow a core interrupt to latch while it is being handled, this handler exits interrupt level with a JUMP (CI) instruction and manipulates the `REGF_IMASK` explicitly to prevent lower priority core interrupts from preempting the system interrupt while allowing higher priority system interrupts to do so.

```
/* prior to interrupt IRPTEN=1, NESTM=1, SNEN=0 */
sec_ivt:            /*IVT+0x3c*/ /* status pushed automatically */
DM(I7,M7) = PX1;
JUMP sec_handler (DB);  /* Jump to avoid next IVT entry. */
DM(I7,M7) = PX2;
PX = R0;        /* Use PX to save all 40-bits of R-registers on 32-bit stack. */
sec_handler:
DM(I7,M7) = PX1;
DM(I7,M7) = PX2;
DM(I7,M7) = I8;
DM(I7,M7) = M8;
/* Save imask somewhere other than s/w stack as it is not part of
thread context. We only want to save the mask when leaving
thread level. Use a counter to see when that happens. */
I8 = saved_imask;
R0 = PM(count_imask_saves);
R0 = PASS R0;
IF EQ PM(M13,I8) = IMASK;
R0 = R0 + 1;
PM(count_imask_saves) = R0;
BIT CLR IMASK 0xffff0000; /* Mask lower priority core interrupts. */
M8 = DM(SEC_ID); /* Source Interrupt identifier (SID) from SEC */
/* Save pc and status on software stack with rest of thread context,
and leave interrupt level so SECI can latch again. */
DM(I7,M7) = PCSTK;
DM(I7,M7) = MODE1STK;
MODE1STK = safe_mode1_value;
JUMP sec_handler_at_thread_level (CI, DB); /* Jump to exit interrupt level. */
POP PCSTK;
I8 = sec_id_vector;
sec_handler_at_thread_level:
DM(SEC_ID) = M8; /* Tell SEC interrupts can latch again */
```

```
/* Higher priority system interrupts may latch from here. */
DM(I7,M7) = ASTATX; /* Save rest of thread status */
DM(I7,M7) = ASTATY;
I8 = PM(M8,I8); /* Call 2nd level handler based on SID */
CALL (M13,I8) (DB);
R0 = M8; /* Pass SEC_ID */
NOP;
/* Assume 2nd level handler returns in same state as it is called:
interrupts globally disabled and SID in r0. */
DM(SEC_END) = R0; /* Tell SEC interrupt is handled. */
/* Lower priority system interrupts may latch from here. */
ASTATY = DM(1,I7); /* Push status and pc back on h/w stacks. */
ASTATX = DM(2,I7);
PUSH STS, PUSH PCSTK;
MODE1STK = DM(3,I7);
PCSTK = DM(4,I7);
I8 = saved_imask; /* Restore IMASK if returning to thread level */
R0 = PM(count_imask_saves);
R0 = R0 - 1;
IF EQ IMASK = PM(M13,I8);
PM(count_imask_saves) = R0;
M8 = DM(5,I7);
I8 = DM(6,I7);
PX2 = DM(7,I7);
PX1 = DM(8,I7);
R0 = PX;
PX2 = DM(9,i7);
PX1 = DM(10,I7);
RTS (DB);
MODIFY(I7, 10);
POP STS;
```

Example SEC Handler in Self-Nesting Interrupt Mode

This handler, or something like it, must also be added to the verification suite to test self-nesting interrupt mode.

```
/* prior to interrupt IRPTEN=1, NESTM=1, SNEN=1 */
sec_ivt: /*IVT+0x3c*/
/* status pushed automatically */
DM(I7,M7) = PX1;
JUMP sec_handler (DB); /* Jump to avoid next IVT entry. */
DM(I7,M7) = PX2;
PX = R0; /* Use PX to save all 40-bits of R-registers on 32-bit stack. */
sec_handler:
DM(I7,M7) = PX1;
DM(I7,M7) = PX2;
DM(I7,M7) = I8;
DM(I7,M7) = M8;
M8 = DM(SEC_ID); /* Source Interrupt identifier (SID) from SEC */
DM(SEC_ID) = M8; /* Acknowledge */
```

```
/* Higher priority system interrupts may latch from here. */
/* Save pc and status on software stack with rest of thread context */
DM(I7,M7) = PCSTK;
DM(I7,M7) = MODE1STK;
MODE1STK = safe_mode1_value;
POP PCSTK, POP STS;
DM(I7,M7) = ASTATX; /* Save rest of thread status */
DM(I7,M7) = ASTATY;
I8 = sec_id_vector;
I8 = PM(M8,I8); /* Call 2nd level handler based on SID */
CALL (M13,I8) (DB);
R0 = M8; /* Pass SID */
NOP;
/* Assume 2nd level handler returns in same state as it is called:
interrupts globally disabled and SID in r0. */
DM(SEC_END) = R0; /* Tell SEC interrupt is handled. */
/* Lower priority system interrupts may latch from here. */
ASTATY = DM(1,I7); /* Push status and pc back on h/w stacks. */
ASTATX = DM(2,I7);
PUSH STS, PUSH PCSTK;
MODE1STK = DM(3,I7);
PCSTK = DM(4,I7);
M8 = DM(5,I7); /*Restore context and return */
I8 = DM(6,I7);
PX2 = DM(7,I7);
PX1 = DM(8,I7);
R0 = PX;
PX2 = DM(9,i7);
RTI (DB);
PX1 = DM(10, I7);
MODIFY(I7, 10);
```

5 Timer

The SHARC+ core includes a programmable interval timer. The REGF_MODE2, REGF_TCOUNT, and REGF_TPERIOD registers control timer operations.

Features

The timer has the following features:

- Simple programming model of three registers for interval timer
- Provides high or low priority interrupt
- If counter expired timer expired pin is asserted
- If core is in emulation space timer halts

Functional Description

The bits that control the timer are given as follows:

- ***Timer Enable*** (REGF_MODE2.TIMEN): This bit directs the processor to enable (if 1) or disable (if 0) the timer.
- ***Timer Count*** (REGF_TCOUNT): This register contains the decrementing timer count value, counting down the cycles between timer interrupts.
- ***Timer Period*** (REGF_TPERIOD): This register contains the timer period, indicating the number of cycles between timer interrupts. The REGF_TCOUNT register contains the timer counter.

To start and stop the timer, programs use the REGF_MODE2.TIMEN bit. With the timer disabled (REGF_MODE2.TIMEN = 0), the program loads REGF_TCOUNT with an initial count value and loads REGF_TPERIOD with the number of cycles for the desired interval. Then, the program enables the timer (REGF_MODE2.TIMEN=1) to begin the count.

On the core clock cycle after REGF_TCOUNT reaches zero, the timer automatically reloads REGF_TCOUNT from the REGF_TPERIODregister. The REGF_TPERIOD value specifies the frequency of timer interrupts.

The number of cycles between interrupts is TPERIOD + 1. The maximum value of TPERIOD is 2^{32} - 1.

The timer decrements the REGF_TCOUNT register during each clock cycle. When the REGF_TCOUNT value reaches zero, the timer generates an interrupt and asserts the TMREXP output pin high for several cycles (when the timer is enabled), as shown in the ***Core Timer Block Diagram*** figure. For more information about TMREXP pin muxing refer to system design chapter in the processor-specific hardware reference.

Programs can read and write the REGF_TPERIOD and REGF_TCOUNT registers by using universal register transfers. Reading the registers does not effect the timer. Note that an explicit write to REGF_TPERIODtakes priority over the sequencer's loading REGF_TCOUNT from REGF_TPERIOD and the timer's decrementing of REGF_TCOUNT. Also note that REGF_TCOUNT and REGF_TPERIOD are not initialized at reset. Programs should initialize these registers before enabling the timer.

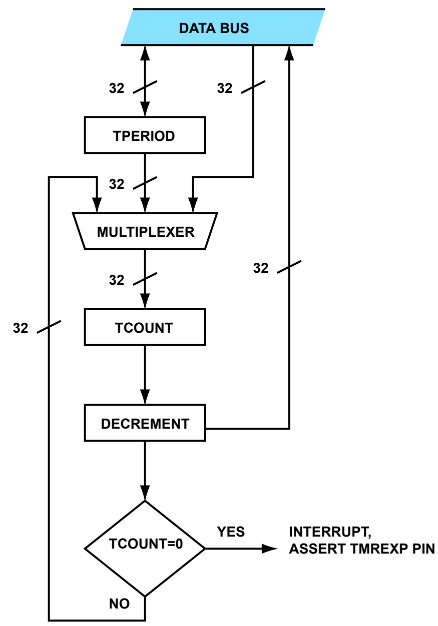

Figure 5-1: Core Timer Block Diagram

To start and stop the timer, the REGF_MODE2.TIMEN has to be set or cleared respectively. The latency of this bit is two core clock cycles at the start of the counter and one core clock cycle at the stop of the counter shown in the ***Timer Enable and Disable*** figure.

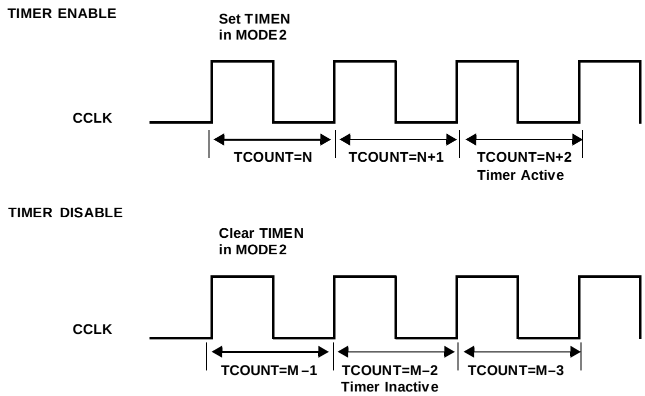

Figure 5-2: Timer Enable and Disable

Timer Exceptions

The timer expired event `REGF_TCOUNT` decrements to zero) generates two interrupts, TMZHI and TMZLI. For information on latching and masking these interrupts to select timer expired priority, see ***Latching Interrupts*** section in the Program Sequencer chapter.

The Timer exception overview is shown in the ***Timer Exceptions*** table. For a complete interrupt list, see Interrupt Priority and Vector Table.

One event can cause multiple exceptions. The timer decrementing to zero causes two timer expired interrupts to be latched, TMZHI (high priority) and TMZLI (low priority). This feature allows selection of the priority for the timer interrupt. Programs should unmask the timer interrupt with the desired priority and leave the other one masked. If both interrupts are unmasked, the processor services the higher priority interrupt first and then services the lower priority interrupt.

Table 5-1: Timer Exceptions

Interrupt Source	Interrupt Condition	Return Register	Return Instruction	IVT level
Core Timer	Timer Priority high	n/a	RTI	11, TMZHI
	Timer Priority low	n/a	RTI	22, TMZLI

6 Data Address Generators

The data address generators (DAGs) generate addresses for data moves to and from data memory (DM) and program memory (PM). By generating addresses, the DAGs let programs refer to addresses indirectly, using a DAG register instead of an absolute address. The DAG's architecture, which appears in the ***Data Address Generator (DAG) Block Diagram*** figure (see Features), supports several functions that minimize overhead in data access routines.

Features

The data address generators have the following features.

- ***Supply address and post-modify.*** Provides an address during a data move and auto-increments the stored address for the next move.
- ***Supply pre-modified (indexed) address.*** Provides a modified address during a data move without incrementing the stored address.
- ***Modify address.*** Increments the stored address without performing a data move.
- ***Bit-reverse address.*** Provides a bit-reversed address during a data move without reversing the stored address, as well as an instruction to explicitly bit-reverse the supplied address.
- ***Byte/Normal Word Space Conversion.*** Converts byte space address to normal word space address and vice versa.
- ***Broadcast data loads.*** Performs dual data moves to complementary registers in each processing element to support single-instruction multiple-data (SIMD) mode.
- ***Circular Buffering.*** Supports addressing a data buffer at any address with predefined boundaries, wrapping around to cycle through this buffer repeatedly in a circular pattern.
- ***Indirect Branch Addressing.*** DAG2 supports indirect branch addressing which provides index and modify address registers used for dynamic instruction driven branch jumps (Md,Ic) or calls (Md,Ic). For more information, see ***Direct Versus Indirect Branches*** in the Program Sequencer chapter.
- ***Semaphores.*** Semaphores are essential for shared memory multi-core systems where multiple cores are competing for the same shared resource and the access needs to be atomic. DAGs support issuing of exclusive accesses on the AXI channel to support semaphores.

- *Scaled Address Arithmetic.* When addressing byte address space, the access size options (SW, NW, LW) provide address scaling for modify, load, and store operations.

Functional Description

As shown in the *Data Address Generator (DAG) Block Diagram* figure, each DAG has four types of registers. These registers hold the values that the DAG uses for generating addresses. The four types of registers are:

- *Index registers (I0-I7 for DAG1 and I8-I15 for DAG2).* An index register holds an address and acts as a pointer to memory. For example, the DAG interprets `DM(I0,0)` and `PM(I8,0)` syntax in an instruction as addresses.
- *Modify registers (M0-M7 for DAG1 and M8-M15 for DAG2).* A modify register provides the increment or step size by which an index register is pre- or post-modified (indexed) during a register move. For example, the `DM(I0,M1)` instruction directs the DAG to output the address in register `I0` then modify the contents of `I0` using the `M1` register.
- *Length and base registers (L0-L7 and B0-B7 for DAG1 and L8-L15 and B8-B15 for DAG2).* Length and base registers set the range of addresses and the starting address for a circular buffer. For more information on circular buffers, see Circular Buffering Mode.

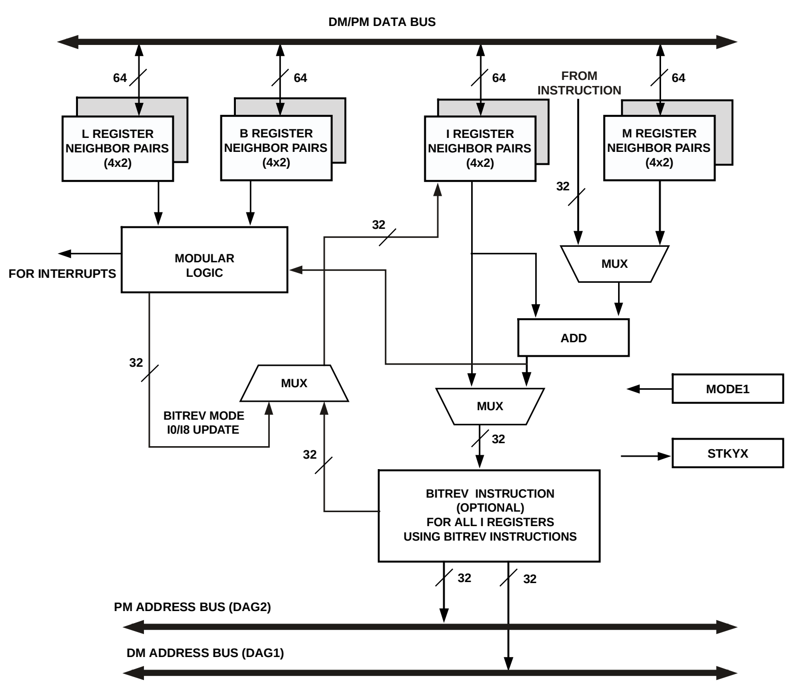

Figure 6-1: Data Address Generator (DAG) Block Diagram

NOTE: The DAG provides scaling for the value from a Modify register if making a long word, word, or short word access to byte space address. The same scaling factor is used for long word and word accesses.

DAG Address Output

The following sections describe how the DAGs output addresses.

Address Versus Word Size

The internal memory accommodates the following word sizes:

- 64-bit long word data (lw)
- 40-bit extended-precision normal word data (nw, 48-bit)
- 32-bit normal word data (nw, 32-bit)
- 16-bit short word data (sw, 16-bit)
- 8-bit byte data (bw, 8-bit)

NOTE: For short word, normal word, or long word accesses, the address space determines which memory word size is accessed. An important item to note is that the DAG automatically adjusts the output address per the word size of the address location. An exception to this rule is that the (lw) qualifier allows 64-bit access using a normal word address.

The address space *does not* select the memory word size for byte addresses. Accesses to byte addresses obey the size of the opcode (lw, unqualified, sw, or bw) not the address space.

The address adjustment allows internal memory to use the address directly as shown in the following example.

```
I15=LW_addr;
pm(i15,0)=r0;    /* 64-bit transfer */
I7=NW_addr;
dm(i7,0)=r8;     /* 32-bit transfer */
I7=SW_addr;
dm(i7,0)=r14;    /* 16-bit transfer */
I7=BW_addr;
dm(i7,0)=r14;    /* byte transfer */
```

DAG Register-to-Bus Alignment

There are a number of word alignment types for DAG registers and PM or DM data buses:

- Byte word (8-bit)
- Short word (16-bit)
- Normal word (32-bit)
- Extended-precision normal word (40-bit)
- Long word (64-bit)

32-Bit Alignment

The DAGs align normal word (32-bit) addressed transfers to the low order bits of the buses. These transfers between memory and 32-bit DAG1 or DAG2 registers use the 64-bit DM and PM data buses. The ***Normal Word (32-Bit) DAG Register Memory Transfers*** figure illustrates these transfers.

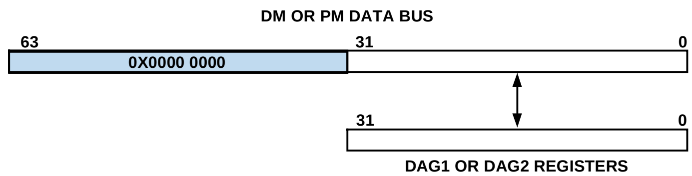

Figure 6-2: Normal Word (32-Bit) DAG Register Memory Transfers

40-Bit Alignment

The DAGs align register-to-register transfers to bits 39-8 of the buses. These transfers between a 40-bit data register and 32-bit DAG1 or DAG2 registers use the 64-bit DM and PM data buses. The ***DAG Register-to-Data Register Transfers*** figure illustrates these transfers.

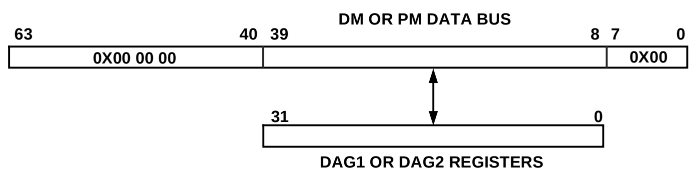

Figure 6-3: DAG Register-to-Data Register Transfers

64-Bit Alignment

Long word (64-bit) addressed transfers between memory and 32-bit DAG1 or DAG2 registers target double DAG registers and use the 64-bit DM and PM data buses. The ***Long Word DAG Register-to-Data Register Transfers*** figure illustrates how the bus works in these transfers.

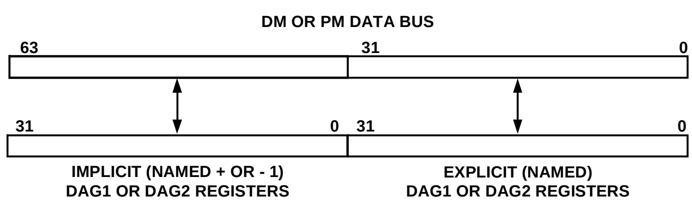

Figure 6-4: Long Word DAG Register-to-Data Register Transfers

DAG1 Versus DAG2

DAG registers are part of the universal register (`Ureg`) set. Programs may load the DAG registers from memory, from another universal register, or with an immediate value. Programs may store the DAG registers' contents to memory or to another universal register.

Both DAGs are identical in their operation modes and can access the entire memory-mapped space. However, the following differences should be noted.

- Only DAG1 is capable of supporting compiler specific instructions like RFRAME and CJUMP.
- Only DAG2 is capable of supporting flow control instruction for indirect branches. Additionally DAG2 access can cause instruction-conflict cache miss/hits for internal memory execution.

Instruction Types

The DAGs perform several types of operations to generate data addresses. As shown in the ***Data Address Generator (DAG) Block Diagram*** figure, the DAG registers and the MODE1 and MODE2 registers contribute to DAG operations. The STKYx registers may be affected by the DAG operations and are used to check the status of a DAG operation.

NOTE: SISD/SIMD mode, access word size, and data location (internal) all influence data access operations.

Long Word Memory Access Restrictions

If the long word transfer specifies an even numbered DAG register (REGF_I[n]0 or 2), then the even numbered register value transfers on the lower half of the 64-bit bus, and the even numbered register + 1 value transfers on the upper half (bits 63-32) of the bus as shown below.

```
I8 = DM(I2,M2);   /* I2 loads to I8/9 pair */
PM(I14,M14) = M5; /* stores M5/4 pair to I14*/
```

If the long word transfer specifies an odd numbered DAG register (REGF_I[n]1 or REGF_B[n] 3), the odd numbered register value transfers on the lower half of the 64-bit bus, and the odd numbered register 1 value (REGF_I[n]0 or REGF_B[n] 2 in this example) transfers on the upper half (bits 63-32) of the bus.

In both the even and odd numbered cases, the explicitly specified DAG register sources or sinks bits 31-0 of the long word addressed memory.

Table 6-1: Neighbor DAG Register for Long Word Accesses (x = B, I, L, M)

DAG Neighbor Registers	
x0 and x1	x8 and x9
x2 and x3	x10 and x11
x4 and x5	x12 and x13
x6 and x7	x14 and x15

Alignment requirements in byte space are summarized in the ***Sizes and Alignment Restrictions in SISD and SIMD Modes*** table in the Byte Address Space Overview of Data Accesses section.

Forced Long Word (lw) Memory Access Instructions

When data is accessed using long word addressing, the data is always long word aligned on 64-bit boundaries in internal memory space. When data is accessed using normal word addressing and the lw mnemonic, the program should maintain this alignment by using an even normal word address (least significant bit of address = 0 for lw and lower 3 bits = 0 for bw addresses). This register selection aligns the normal word or byte word address with a 64-bit boundary (long word address). For more information, see ***Unaligned Forced Long Word Access*** in the Memory chapter.

NOTE: The forced long word (`lw`) access only effects normal word address and byte address accesses and overrides all other factors (`REGF_MODE1.PEYEN`, `CMMR_SYSCTL.IMDWBLK3`).

All long word accesses load or store two consecutive 32-bit data values. The register file source or destination of a long word access is a set of two neighboring data registers (the ***Neighbor DAG Register for Long Word Accesses*** table) in a processing element. In a forced long word access (using the `lw` mnemonic), the even (normal word address) location moves to or from the explicit register in the neighbor-pair, and the odd (normal word address) location moves to or from the implicit register in the neighbor-pair. In the ***Long Word Move Options*** example, the following long word moves can occur.

Long Word Move Options

```
DM(NW_Address) = R0 (lw);

/* The data in R0 moves to location DM(NW_Address), and the data in R1 moves to
location DM(NW_Address) */

R15 = DM(NW_Address)(lw);

/* The data at location DM(NW_Address) moves to R14, and the data at location
DM(NW_Address) moves to R15 */
```

Byte Word (bw) (bwse) and Short Word (sw) (swse) Memory Access Instructions

When data is accessed in byte space, 8-bit data may be accessed with the `bw` and `bwse` modifiers, and 16-bit data with `sw` and `swse` modifiers. Unmodified and `lw` modified loads and stores behave as they do in normal word space. This is summarised in the ***Byte Address Access Modifiers*** table.

The `bw`, `bwse`, `sw` and `swse` modifiers may only be used when byte space is addressed. Attempts to access other address spaces with these instructions cause an Illegal address space access interrupt. See Byte Address Space Overview of Data Accesses for details of byte space.

Modifier	Size in memory	Value loaded to register	Value stored
(bw)	8-bits	Zero extended to 32-bits	Low 8-bits of 32-bit register value
(bwse)	8-bits	Sign extend to 32-bits	*Not allowed, use (bw)*
(sw)	16-bits	Zero extended to 32-bits	Low 16-bits of 32-bit register value
(swse)	16-bits	Sign extended to 32-bits	*Not allowed, use (sw)*
	32-bits or 40-bits	Value in memory	32-bit or 40-bit register value
(lw)	64-bits	Value in memory	64-bit value in register pair

Pre-Modify Instruction

As shown in the ***Pre-Modify and Post-Modify Operations*** figure, the DAGs support two types of modified addressing, pre- and post-modify. Modified addressing is used to generate an address that is incremented by a value or a register.

When addressing byte space, address scaling affects the output, but does not change the values in the registers. For more information about address scaling arithmetic, see Enhanced Modify Instruction for Address Scaling.

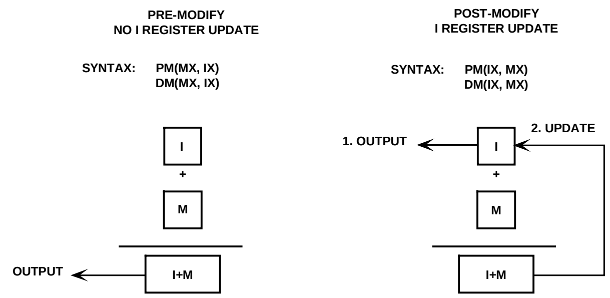

Figure 6-5: Pre-Modify and Post-Modify Operations

In pre-modify (indexed) addressing, the DAG adds an offset (modifier), which is either an `M` register or an immediate value, to an `I` register and outputs the resulting address. Pre-modify addressing does not change or update the `I` register.

NOTE: Pre-modify addressing operations must not change the memory space of the address.

Post-Modify Instruction

The DAGs support post-modify addressing. Modified addressing is used to generate an address that is incremented by a value or a register. In post-modify addressing, the DAG outputs the `I` register value unchanged, then adds an `M` register or immediate value, updating the `I` register value.

When addressing byte space, address scaling affects the output, but does not change the values in the registers. For more information about address scaling arithmetic, see Enhanced Modify Instruction for Address Scaling.

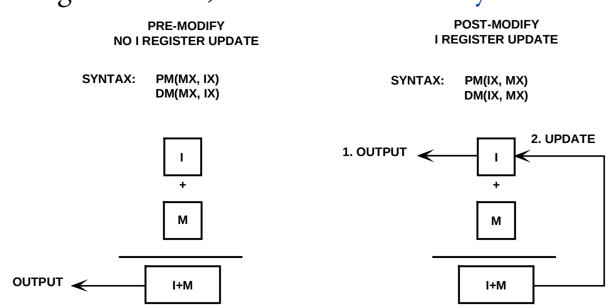

Figure 6-6: Pre-Modify and Post-Modify Operations

The DAG post-modify addressing type can be used to emulate the push (save of registers) to a sw stack.

Post-Modify Addressing

```
BIT CLR MODE1 CBUFEN; /* clear circular buffer*/
nop;
I1 = buffer;          /* Index Pointer */
M1 = 1;               /* Modify */
instruction;          /* stall, any non-DAG instruction */
instruction;          /* stall, any non-DAG instruction */
R3 = dm(I1,M1);       /* 1st access */
R3 = dm(I1,M1);       /* 2nd access */
```

Modify Instruction

The DAGs support two operations that modify an address value in an index register without outputting an address. These two operations, address bit-reversal and address modify, are useful for bit-reverse addressing and maintaining pointers.

The `MODIFY` instruction modifies addresses in any DAG index register (`I0-I15`) without accessing memory.

The syntax for the `MODIFY` instruction is similar to post-modify addressing (index, then modifier). The `MODIFY` instruction accepts either a 32-bit immediate value or an `M` register as the modifier. The following example adds 4 to `I1` and updates `I1` with the new value.

```
MODIFY(I1,4);
```

NOTE: If the `I` register's corresponding `B` and `L` registers are set up for circular buffering, a `MODIFY` instruction performs the specified buffer wraparound (if needed).

The `MODIFY` instruction executes independent of the state of the `REGF_MODE1.CBUFEN` bit. The `MODIFY` instruction always performs circular buffer modify of the index registers if the corresponding `B` and `L` registers are configured, independent of the state of the `REGF_MODE1.CBUFEN` bit.

Enhanced Modify Instruction

`Ib = MODIFY(Ia,Mc);` is an enhanced version of the MODIFY instruction. This instruction loads the modified index pointer into another index register. If the source and destination registers are different, then:

- The source register (`Ia`) is not updated.
- The destination register (Ib) receives the result of the modify.

If the `B` and `L` registers corresponding to the source `I` register (`Ia`) are set up for circular buffering, the `MODIFY` instruction performs specified buffer wraparound if it is needed.

The following example assumes that the `La` and `Ba` registers that correspond to the source `Ia` register are set up for circular buffering, the modify operation executes circular buffer wraparound if it is needed, and the `Ib` register is updated with the value after wraparound.

```
B0 = 0x40000;
L0 = 0x10000;
I0 = 0x4ffff;
I1 = modify(I0, 2); // I1 == 0x40001
```

Immediate Modify Instruction

Instructions can also use a number (immediate value), instead of an `M` register, as the modifier. The size of an immediate value that can modify an `I` register depends on the instruction type. For all single data access operations, modify immediate values can be up to 32 bits wide. Instructions that combine DAG addressing with computations limit the size of the modify immediate value. In these instructions (multifunction computations), the modify immediate values can be up to 6 bits wide. The following example instruction accepts up to 32-bit modifiers:

```
R1 = DM(0x40000000,I1);    /* DM address = I1 + 0x4000 0000 */
```

The following example instruction accepts up to 6-bit modifiers:

```
R0 = R1 + R2, PM(I8, 0x0B) = R3;       /* PM address = I8, I8 = I8 + 0x0B */
```

Bit-Reverse Instruction

The `BITREV` instruction modifies and bit-reverses addresses in any DAG index register (`I0-I15`) without accessing memory. This instruction is independent of the bit-reverse mode. The `BITREV` instruction adds a 32-bit immediate value to a DAG index register, bit-reverses the result, and writes the result back to the same index register. The following example adds 4 to `I1`, bit-reverses the result, and updates `I1` with the new value:

```
BITREV(I1,4);
```

NOTE: Bit-reverse mode is supported. See Operating Modes. However bit-reverse mode does ***NOT*** support address scaling for byte space accesses. For more information, see Enhanced Modify Instruction for Address Scaling.

Enhanced Bit-Reverse Instruction

An enhanced version of the `BITREV` instruction, that loads the bit reversed index pointer into another index register is shown below:

```
I6 = BITREV(I1,0);
```

Enhanced Modify Instruction for Address Scaling

When addressing byte address space, the access size options (for example, short word `(sw)`) provide address scaling for modify, load, and store operations.

The need to scale indices of arrays of words by 4 for byte-addressed pointer arithmetic necessitates a modify by a scaled increment. Likewise, scaling is supported in the addressing modes of loads and stores.

Scaling occurs automatically for pointers in the byte-addressed space. In the word-addressed space, scaling of the offset does not occur. For loads and stores, scaling is by the size of the access (except in the case of `(lw)`), while for modifies it is dependent on the instruction, specified in the `(sw)` or `(nw)` flag. Circular buffering interprets the length in terms of the unit size too, so the value in the length register is also scaled.

Byte or short word access to any space other than byte address space results in an illegal address space (ILAD) interrupt and the access ignores the size information. Modify instructions with `(sw)` flag with the I-register in non-byte addressed space and `(nw)` flag in any long word or short word address space also results in ILAD interrupt.

The existing modify instruction is defined to do no scaling on its index, regardless of whether the I-register is in the byte- or word-address space. This is for backwards compatibility. The `(sw)` or `(nw)` version performs conditional scaling of the index if the I-register is in the byte address space (see Type 7a ISA/VISA (cond + comp + index modify).

For example:

```
Existing enhanced MODIFY instruction (ADSP-214xx)
Ia = MODIFY(Ib,Mc);     /* Add Mc bytes, Ia=Ib+Mc */
Does not scale the modifier, whatever the address space

Enhanced MODIFY instructions
Ia = MODIFY(Ib,Mc) (sw);     /* Add Mc shorts, Ia=Ib+(2xMc) */
Ia = MODIFY(Ib,Mc) (nw);     /* Add Mc words, Ia=Ib+(4xMc) */
Scales the modifier if Ib contains byte address
```

There is a complication for bit-reverse addressing. Note that the base address is tested to see which address space it is in after any bit-reversing that is necessary due to the mode1 register. The scaling of the offset for bit-reverse addressing is in the opposite direction (shift down) than for normal addressing. This is because the offset is itself reversed, so the extra zero bits are required at the top of the offset word. This is applicable for modify instruction also. The ***Legal and Illegal Accesses to Byte Space With or Without Address Scaling*** table illustrates all conditions for address scaling. If scaling applies to 40-bit NW space (extended precision) refer to the Table 6-3 Operand Addressed in Non-Byte Space or Byte Space for Extended Precision Accesses (40-bit) table.

Table 6-2: Legal and Illegal Accesses to Byte Space With or Without Address Scaling

Modify Instruction	Operand Ireg in ... Non-Byte-Addressed Space	Operand Ireg in ... Byte-Addressed Space

`Im = modify(In, mod)`	`if ((In + data) >= (Bn + Ln))`
	`Im ← In + mod - Ln`
	`else if ((In + data) < Bn)`
	`Im ← In + mod + Ln`
	`else`
	`Im ← In + mod`
		`if ((In + mod) >= (Bn + Ln))`
		`Im ← In + mod - Ln`
		`else if ((In + mod) < Bn)`
		`Im ← In + mod + Ln`
		`else`
		`Im ← In + mod`

`Im = modify(In, mod) (sw)`	`Illegal address space interrupt`	`if ( (In == I0 && BR0)`
		`|| (In == I8 && BR8))`
		`{`
		`scaled_mod = mod >> 1;`
		`scaled_len = Ln >> 1;`
		`}`
		`else`
		`{`
		`scaled_mod = mod << 1;`
		`scaled_len = Ln << 1;`
		`}`

Table 6-2: Legal and Illegal Accesses to Byte Space With or Without Address Scaling (Continued)

<table>
<tr><th>Modify Instruction</th><th>Operand Ireg in ...
Non-Byte-Addressed Space</th><th>Operand Ireg in ...
Byte-Addressed Space</th></tr>
<tr><td></td><td></td><td>if ((In + scaled_mod) >= (Bn +
scaled_len))
 Im ← In + scaled_ mod - scaled_len
else if ((In + scaled_mod) < Bn)
 Im ← In + scaled_ mod + scaled_len
else
 Im ← In + scaled_ mod</td></tr>
<tr><td>Im = modify(In, mod) (nw)</td><td>if ((In + mod) >= (Bn + Ln))
 Im ← In + mod - Ln
else if ((In + mod) < Bn)
 Im ← In + mod + Ln
else
 Im ← In + mod</td><td>if ((In == I0 && BR0)
 || (In == I8 && BR8))
 {
 scaled_mod = mod >> 2;
 scaled_len = Ln << 2;
 }
else
 {
 scaled_mod = mod << 2;
 scaled_len = Ln << 2;
 }
if ((In + scaled_mod) >= (Bn +
scaled_len))
 Im ← In + scaled_ mod - scaled_len
else if ((In + scaled_mod) < Bn)
 Im ← In + scaled_ mod + scaled_len
else
 Im ← In + scaled_ mod</td></tr>
<tr><td colspan="3">Data Move Instructions</td></tr>
<tr><td>Rm = dm(mod, In)
Rm = dm(mod, In) (lw)</td><td>if ((In == I0 && BR0)
 || (In == I8 && BR8)) {
 Rm ← dm(reverse(In + mod))
} else {
 Rm ← dm(In + mod)
}</td><td>if ((In == I0 && BR0)
 || (In == I8 && BR8)) {
 scaled_mod = mod >> 2;
 Rm ← dm(reverse(In + scaled_mod))
} else {
 scaled_mod = mod << 2;
 Rm ← dm(In + scaled_mod)
}</td></tr>
<tr><td>Rm = dm(In, mod)
Rm = dm(In, mod) (lw)</td><td>if ((In == I0 && BR0)
 || (In == I8 && BR8)) {
 Rm ← dm(reverse(In))
} else {
 Rm ← dm(In)
}if ((In + mod) >= (Bn +
Ln))
 I n ← In + mod - Ln
else if ((In + mod) < Bn)
 In ← In + mod + Ln
else
 In ← In + mod</td><td>if ((In == I0 && BR0)
 || (In == I8 && BR8)) {
 Rm ← dm(reverse(In))
 scaled_mod = mod >> 2;
 scaled_len = Ln >> 2;
} else {
 Rm ← dm(In)
 scaled_mod = mod << 2;
 scaled_len = Ln << 2;
}
if ((In + scaled_mod) >= (Bn +
scaled_len))
 Im ← In + scaled_ mod - scaled_len
else if ((In + scaled_mod) < Bn)
 Im ← In + scaled_ mod + scaled_len
else</td></tr>
</table>

Table 6-2: Legal and Illegal Accesses to Byte Space With or Without Address Scaling (Continued)

<table>
<tr><th>Modify Instruction</th><th>Operand Ireg in ...
Non-Byte-Addressed Space</th><th>Operand Ireg in ...
Byte-Addressed Space</th></tr>
<tr><td></td><td></td><td>Im ← In + scaled_ mod</td></tr>
<tr><td>Rm = dm(mod, In) (bw)</td><td>Illegal address space interrupt</td><td>if ((In == I0 && BR0)
 || (In == I8 && BR8)) {
 Rm ← dm(reverse(In + mod))
} else {
 Rm ← dm(In + mod)
}</td></tr>
<tr><td>Rm = dm(In, mod) (bw)</td><td>Illegal address space interrupt</td><td>if ((In == I0 && BR0)
 || (In == I8 && BR8)) {
 Rm ← dm(reverse(In))
} else {
 Rm ← dm(In)
}
if ((In + mod) >= (Bn + Ln))
 I n ← In + mod - Ln
else if ((In + mod) < Bn)
 In ← In + mod + Ln
else
 In ← In + mod</td></tr>
<tr><td>Rm = dm(mod, In) (sw)</td><td>Illegal address space interrupt</td><td>if ((In == I0 && BR0)
 || (In == I8 && BR8)) {
 scaled_mod = mod >> 1;
 Rm ← dm(reverse(In + scaled_mod))
} else {
 scaled_mod = mod << 1;
 Rm ← dm(In + scaled_mod)
}</td></tr>
<tr><td>Rm = dm(In, mod) (sw)</td><td>Illegal address space interrupt</td><td>if ((In == I0 && BR0)
 || (In == I8 && BR8)) {
 Rm ← dm(reverse(In))
 scaled_mod = mod >> 1;
 scaled_len = Ln >> 1;
} else {
 Rm ← dm(In)
 scaled_mod = mod << 1;
 scaled_len = Ln << 1;
}
if ((In + scaled_mod) >= (Bn +
scaled_len))
 I n ← In + scaled_ mod - scaled_len
else if ((In + scaled_mod) < Bn)
 I n ← In + scaled_ mod + scaled_len
else
 I n ← In + scaled_ mod</td></tr>
</table>

Table 6-3: Operand Addressed in Non-Byte Space or Byte Space for Extended Precision Accesses (40-bit)

<table>
<tr><th>Data Move Instructions</th><th>Operand I-reg in non-byte-addressed space*1</th><th>Operand I-reg in byte-addressed space</th></tr>
<tr><td>Rm = dm(mod, In)</td><td>if ((In == I0 && BR0)
|| (In == I8 && BR8)) {
offset = reverse(In+mod) % 2;
Rm ← dm((3/2 * reverse(In + mod)) << 1 + offset)
} else {
Offset = (In + data) % 2;
Rm ← dm((3/2 * (In + mod)) << 1 + offset)
}</td><td>if ((In == I0 && BR0)
|| (In == I8 && BR8)) {
scaled_mod = mod >> 2;
byteoffset = reverse(In+scaled_mod) % 4;
wordoffset = (reverse(In+scaled_mod)>>2) % 2;
Rm ← dm((3/2 * reverse(In + scaled_mod) >> 2 + wordoffset)
<< 2 + byteoffset)
} else {
scaled_mod = data << 2;
byteoffset = (In+scaled_mod) % 4;
wordoffset = ((In+scaled_mod) >> 2) % 2;
Rm ← dm(((3/2 * (In + scaled_mod) >> 2 + wordoffset)
<< 2) + byteoffset)
}</td></tr>
<tr><td>Rm = dm(In, mod)</td><td>if ((In == I0 && BR0)
|| (In == I8 && BR8)) {
offset = reverse(In) % 2;
Rm ← dm((3/2 * reverse(In) << 1) + offset)
} else {
offset = In % 2;
Rm ← dm((3/2 * In) << 1 + offset)
}
if ((In + mod) >= (Bn + Ln))
In ← In + mod - Ln
else if ((In + mod) < Bn)
In ← In + mod + Ln
else
In ← In + mod</td><td>if ((In == I0 && BR0)
|| (In == I8 && BR8)) {
byteoffset = reverse(In) % 4;
wordoffset = (reverse(In) >> 2) % 2;
Rm ← dm(((3/2 * reverse(In) >> 2 + wordoffset) << 2)
+ byteoffset)
scaled_mod = mod >> 2;
scaled_len = Ln >> 2;
} else {
byteoffset = In % 4;
wordoffset = (In >> 2) % 2;
Rm ← dm(((3/2 * In >> 2 + wordoffset) << 2)
+ byteoffset)
scaled_mod = mod << 2;
scaled_len = Ln << 2;
}
if ((In + scaled_mod) >= (Bn + scaled_len))
In ← In + scaled_mod scaled_len
else if ((In + scaled_mod) < Bn)
In ← In + scaled_mod + scaled_len
else
In ← In + scaled_mod</td></tr>
<tr><td>Rm = dm(mod, In) (bw)</td><td>Illegal address space interrupt</td><td>if ((In == I0 && BR0)
|| (In == I8 && BR8)) {
byteoffset = reverse(In+mod) % 4;
wordoffset = (reverse(In+mod) >> 2) % 2;
Rm ← dm((3/2 * reverse(In + mod) >> 2 + wordoffset)
<< 2 + byteoffset + 2)
} else {
byteoffset = (In+mod) % 4;
wordoffset = (In+mod) >>2 % 2;</td></tr>
</table>

Table 6-3: Operand Addressed in Non-Byte Space or Byte Space for Extended Precision Accesses (40-bit) (Continued)

<table>
<tr><th>Data Move Instructions</th><th>Operand I-reg in non-byte-addressed space*1</th><th>Operand I-reg in byte-addressed space</th></tr>
<tr><td></td><td></td><td>Rm ← dm(((3/2 * (In + mod) >> 2 +
wordoffset) << 2)
+ byteoffset + 2)
}</td></tr>
<tr><td>Rm = dm(In, mod) (bw)</td><td>Illegal address space interrupt</td><td>if ((In == I0 && BR0)
|| (In == I8 && BR8)) {
byteoffset = reverse(In) % 4;
wordoffset = reverse(In) >> 2 % 2;
Rm ← dm(((3/2 * reverse(In) >> 2 +
wordoffset) << 2) + byteoffset + 2)
} else {
byteoffset = In % 4;
wordoffset = In >> 2 % 2;
Rm ← dm(((3/2 * In >> 2 + wordoffset) <<
2) + byteoffset + 2)
}
if ((In + mod) >= (Bn + Ln))
In ← In + mod Ln
else if ((In + mod) < Bn)
In ← In + mod + Ln
else
In ← In + mod</td></tr>
<tr><td>Rm = dm(mod, In) (sw)</td><td>Illegal address space interrupt</td><td>if ((In == I0 && BR0)
|| (In == I8 && BR8)) {
scaled_mod = mod >> 1;
byteoffset = reverse(In+scaled_mod) % 4;
wordoffset = reverse(In+scaled_mod) >> 2 %
2;
Rm ← dm((3/2 * reverse(In + scaled_mod) >>
2 + wordoffset)
<< 2 + byteoffset + 2)
} else {
scaled_mod = data << 2;
byteoffset = (In+scaled_mod) % 4;
wordoffset = (In+scaled_mod)>> 2 % 2;
Rm ← dm(((3/2 * (In + scaled_mod) >> 2 +
wordoffset)
<< 2) + byteoffset + 2)
}</td></tr>
<tr><td>Rm = dm(In, mod) (sw)</td><td>Illegal address space interrupt</td><td>if ((In == I0 && BR0)
|| (In == I8 && BR8)) {
byteoffset = reverse(In) % 4;
wordoffset = reverse(In) >> 2 % 2;
Rm ← dm(((3/2 * reverse(In) >> 2 +
wordoffset) << 2)
+ byteoffset)
scaled_mod = mod >> 1;
scaled_len = Ln >> 1;
} else {
byteoffset = In % 4;
wordoffset = In >> 2 % 2;</td></tr>
</table>

Table 6-3: Operand Addressed in Non-Byte Space or Byte Space for Extended Precision Accesses (40-bit) (Continued)

<table>
<tr><th>Data Move Instructions</th><th>Operand I-reg in non-byte-addressed space*1</th><th>Operand I-reg in byte-addressed space</th></tr>
<tr><td></td><td></td><td>Rm ← dm(((3/2 * In >> 2 + wordoffset) << 2) + byteoffset)
scaled_mod = mod << 1;
scaled_len = Ln << 1;
}
if ((In + scaled_mod) >= (Bn + scaled_len))
In ← In + scaled_mod scaled_len
else if ((In + scaled_mod) < Bn)
In ← In + scaled_mod + scaled_len
else
In ← In + scaled_mod</td></tr>
<tr><td>Rm = dm(mod, In) (lw)</td><td>if ((In == I0 && BR0)
|| (In == I8 && BR8)) {
Rm ← dm(reverse(In + mod))
} else {
Rm ← dm(In + mod)
}</td><td>if ((In == I0 && BR0)
|| (In == I8 && BR8)) {
scaled_mod = mod >> 2;
Rm ← dm(reverse(In + scaled_mod))
} else {
scaled_mod = mod << 2;
Rm ← dm(In + scaled_mod)
}</td></tr>
<tr><td>Rm = dm(In, mod) (lw)</td><td>if ((In == I0 && BR0)
|| (In == I8 && BR8)) {
Rm ← dm(reverse(In))
} else {
Rm ← dm(In)
}if ((In + mod) >= (Bn + Ln))
In ← In + mod - Ln
else if ((In + mod) < Bn)
In ← In + mod + Ln
else
In ← In + mod</td><td>if ((In == I0 && BR0)
|| (In == I8 && BR8)) {
Rm ← dm(reverse(In))
scaled_mod = mod >> 2;
scaled_len = Ln >> 2;
} else {
Rm ← dm(In)
scaled_mod = mod << 2;
scaled_len = Ln << 2;
}
if ((In + scaled_mod) >= (Bn + scaled_len))
Im ← In + scaled_mod scaled_len
else if ((In + scaled_mod) < Bn)
Im ← In + scaled_mod + scaled_len
else
Im ← In + scaled_mod</td></tr>
</table>

*1 Access addresses shown are short word addresses i.e. Rm <- dm(shortword address), 40/48 bits will be fetched starting from the short word address

Switch Address Instruction

New instructions are provided to convert pointers between the *byte* and the legacy term *word*.

```
IF COND  compute  Id = B2W(Is);
   compute  Id = W2B(Is);
   compute  Bd = B2W(Bs);
   compute  Bd = W2B(Bs);
```

NOTE: In case of B2W or W2B instruction on any B register, the corresponding I register is not implicitly updated. These instructions have the following semantics:

Table 6-4: Switch Address Instruction Semantics

Instruction	Base addr in word-addressed space	Base addr in byte-addressed space
Id = B2W(Is)	Id = Is	Convert byte pointer to word pointer. Likely semantics Id <- Is >> 2 Exact semantics depend on address map and must work correctly for all addresses in both internal and external memory. In case of byte addresses not having word space equivalent Is will be retained as is i.e. Id = Is and illegal address space (ILAD) interrupt is generated.
Id = W2B(Is)	Convert word pointer to byte pointer. Likely semantics Id <- Is << 2 Exact semantics depend on address map and must work correctly for all addresses in both internal and external memory. In case of word addresses not having byte space equivalent Is will be retained as is i.e. Id = Is and illegal address space (ILAD) interrupt is generated.	Id = Is
Bd = B2W(Bs)	Bd = Bs	Convert byte pointer to word pointer. Likely semantics Bd <- Bs >> 2 Exact semantics depend on address map and must work correctly for all addresses in both internal and external memory. In case of byte addresses not having word space equivalent Bs are retained as is i.e. Bd = Bs and illegal address space (ILAD) interrupt is generated.
Bd = W2B(Bs)	Convert word pointer to byte pointer. Likely semantics Bd <- Bs << 2 Exact semantics depend on address map and must work correctly for all addresses in both internal and external memory. In case of word addresses not having byte space equivalent Bs are retained as is i.e. Bd = Bs and illegal address space (ILAD) interrupt is generated.	Bd = Bs

Dual Data Move Instructions

The number of transfers that occur in a clock cycle influences the data access operation. As described in ***Internal Memory Space*** in the Memory chapter, the processor core supports single cycle, dual-data accesses to and from internal memory for register-to-memory and memory-to-register transfers. Dual-data accesses occur over the PM and DM bus and act independently of SIMD/SISD mode setting. Though only available for transfers between memory and data registers, dual-data transfers are extremely useful because they double the data throughput over single-data transfers.

Note that the explicit use of complementary registers (*CDreg*) is not supported for dual data access.

Examples:

```
f0=f3*f4, f8=f8+f10, f3=dm(i2,m2), f4= pm(i9,m9); /* DREG*/
f0=f3*f4, f8=f8+f10, s3=dm(i2,m2), s4= pm(i9,m9); /* asm error*/
f0=f3*f4, f8=f8+f10, s3=dm(i2,m2); /* SDREG */
```

ATTENTION: On SHARC+ cores, it is illegal to use the DAGs in Type 1 instructions to access MMR space. External memory space access is legal.

```
        R8 = DM(I4,M3), PM(I12,M13) = R0; /* Dual access */
    R0 = DM(I5,M5);                       /* Single access */
```

For examples of data flow paths for single and dual-data transfers, see the Register Files chapter.

The processor core can use its complementary registers explicitly. They support single data access as shown in the example below.

```
S8 = DM(I4,M3);
PM (I12,M13) = S12;
COMP, S8 = DM(I5,M5);
COMP, DM(I5,M5) = S14;
```

Conditional DAG Transfers

Conditions with DAG transfers allows programs to make memory accesses conditional. For more information, see the ***Program Sequencer*** chapter.

DAG Breakpoint Units

Both DAGs are connected to the breakpoint units used for hardware breakpoints. They are used if user breakpoints are enabled. For more information, see the ***Program Trace Macrocell (PTM)*** chapter.

DAG Instruction Restrictions

Modify (M) registers can work with any index (I) register in the same DAG (DAG1 or DAG2).

The DAGs do allow transfers involving registers on the two DAG, as in the following example.

```
DM(M2,I1) = I12;
```

```
L7 = PM(M12,I12);
```

However, transfers using registers on one DAG are not allowed, as in the following example. In this case, the assembler returns an error message.

```
DM(M2,I1) = I0; /* generates asm error */
```

Instruction Summary

The ***DAG Instruction Types Summary*** table lists the instruction types associated with DAG transfer instructions. Note that instruction set types may have more options (conditions or compute). For more information see the Instruction Set Types chapter. In these tables, note the meaning of the following symbols:

- *Ia* indicates a DAG1 index register (I7-0), *Ic* indicates a DAG2 index register (I15-8)
- *Mb* indicates a DAG1 modify register (M7-0), *Md* indicates a DAG2 modify register (M15-8)
- *Ba* indicates a DAG1 base register (B7-0), *Bc* indicates a DAG2 base register (B15-8)
- *Ureg* indicates any universal register, *Dreg* indicates any data register

Table 6-5: DAG Instruction Types Summary

Instruction Type	DAG Instruction Syntax	Description
1a/b	DM(Ia,Mb)=Dreg, PM(Ic,Md)=Dreg; Dreg=DM(Ia,Mb), Dreg=PM(Ic,Md); Dreg=DM(Ia,Mb), PM(Ic,Md)=Dreg; DM(Ia,Mb)=Dreg, Dreg=PM(Ic,Md);	DAG1/2, post-modify, Dreg, Dual data move
3a	DM(Ia,Mb)=Ureg (lw); PM(Ic,Md)=Ureg (lw); Ureg=DM(Ia,Mb) (lw); Ureg=PM(Ic,Md) (lw); DM(Mb,Ia)=Ureg (lw); PM(Md,Ic)=Ureg (lw); Ureg=DM(Mb,Ia) (lw); Ureg=PM(Mc,Id) (lw);	DAG1/2, post/pre modify, Ureg, forced long word access
3b	DM(Ia,Mb)=Ureg (bw/sw); PM(Ic,Md)=Ureg (bw/sw); Ureg=DM(Ia,Mb) (bw/bwse/sw/swse); Ureg=PM(Ic,Md) (bw/bwse/sw/swse); DM(Mb,Ia)=Ureg (bw/sw); PM(Md,Ic)=Ureg (bw/sw); Ureg=DM(Mb,Ia) (bw/bwse/sw/swse); Ureg=PM(Mc,Id) (bw/bwse/sw/swse);	DAG1/2, post/pre modify, Ureg, byte (bw), byte with sign extend (bwse), short word (sw), short word with sign extend (swse)
3c	DM(Ia,Mb)=Dreg; Dreg=DM(Ia,Mb);	DAG1, Post modify, Dreg

Table 6-5: DAG Instruction Types Summary (Continued)

Instruction Type	DAG Instruction Syntax	Description

3d	`Ureg=DM(Ia,Mb) (lw/nw/sw/bw,ex);`
	`Ureg=PM(Ic,Md) (lw/nw/sw/bw,ex);`
	`Ureg=DM(Ia,Mb) (bwse/swse,ex);`
	`Ureg=PM(Ic,Md) (bwse/swse,ex);`
	
	`DM(Ia,Mb)=Ureg (lw/nw/sw/bw,ex);`
	`PM(Ic,Md)=Ureg (lw/nw/sw/bw,ex);`
	`DM(Ia,Mb)=Ureg`
	`PM(Ic,Md)=Ureg`
	
	`Ureg=DM(Mb,Ia) (lw/nw/sw/bw,ex);`
	`Ureg=PM(Md,Ic) (lw/nw/sw/bw,ex);`
	`Ureg=DM(Mb,Ia) (bwse/swse,ex);`
	`Ureg=PM(Md,Ic) (bwse/swse,ex);`
	
	`DM(Mb,Ia)=Ureg (lw/nw/sw/bw,ex);`
	`PM(Md,Ic)=Ureg (lw/nw/sw/bw,ex);`
	`DM(Mb,Ia)=Ureg`
	`PM(Md,Ic)=Ureg`
		DAG1/2, pre/post modify, exclusive access, Ureg

4a/b	`Dreg=DM(Ia,data6);`
	`Dreg=PM(Ic,data6);`
	`DM(Ia,data6)=Dreg;`
	`PM(Ic,data6)=Dreg;`
	
	`Dreg=DM(data6,Ia);`
	`Dreg=PM(data6,Ic);`
	`DM(data6,Ia)=Dreg;`
	`PM(data6,Ic)=Dreg;`
		DAG1/2, pre/post modify, Dreg, immediate modify

4d	`Dreg=DM(Ia,data6) (bw/bwse/sw/swse);`
	`Dreg=PM(Ic,data6) (bw/bwse/sw/swse);`
	`DM(Ia,data6)=Dreg (bw/sw);`
	`PM(Ic,data6)=Dreg (bw/sw);`
	
	`Dreg=DM(data6, Ia) (bw/bwse/sw/swse);`
	`Dreg=PM(data6, Ic) (bw/bwse/sw/swse);`
	`DM(data6, Ia)=Dreg (bw/sw);`
	`PM(data6, Ic)=Dreg (bw/sw);`
		DAG1/2, pre/post modify, Dreg, immediate modify, byte (bw), byte with sign extend (bwse), short word (sw), short word with sign extend (swse)

6a	`Dreg=DM(Ia,Mb);`
	`Dreg=PM(Ic,Md);`
	`DM(Ia,Mb)=Dreg;`
	`PM(Ic,Md)=Dreg;`
	
	`Dreg=DM(Mb,Ia);`
	`Dreg=PM(Md,Ic);`
	`DM(Mb,Ia)=Dreg;`
	`PM(Md,Ic)=Dreg;`
		DAG1/2, pre/post modify, Dreg

7a/b	`MODIFY(Ia,Mb);`
	`MODIFY(Ic,Md);`
	`Ia=MODIFY(Ia,Mb);`
		DAG1/2, Index Modify, short word (sw) or normal word (nw).

Table 6-5: DAG Instruction Types Summary (Continued)

Instruction Type	DAG Instruction Syntax	Description

	`Ic=MODIFY(Ic,Md);`
	`Ia=MODIFY(Ia,Mb) (sw);`
	`Ic=MODIFY(Ic,Md) (sw);`
	`Ia=MODIFY(Ia,Mb) (nw);`
	`Ic=MODIFY(Ic,Md) (nw);`	

7d	`Ia=B2W(Ia);`
	`Ic=B2W(Ic);`
	`Ia=W2B(Ia);`
	`Ic=W2B(Ic);`
	`Ba=B2W(Ba);`
	`Bc=B2W(Bc);`
	`Ba=W2B(Ba);`
	`Bc=W2B(Bc);`
		DAG1/2, scaled address arithmetic

10a	`DM(Ia,Mb)=Dreg;` `Dreg=DM(Ia,Mb);`	DAG1, post modify, Dreg

14a	`DM(addr32)=Ureg (lw);`
	`PM(addr32)=Ureg (lw);`
	
	`Ureg=DM(addr32) (lw);`
	`Ureg=PM(addr32) (lw);`
		DAG1/2, direct address, Ureg, LW option

14d	`Dreg=DM(addr32) (lw/nw/sw/bw/ex);`
	`Dreg=DM(addr32) (nwse/swse/bwse/ex);`
	`DM(addr32)=Dreg (lw/nw/sw/bw/ex);`
		DAG1, direct address, Dreg, byte (bw), byte with sign extend (bwse), short word (sw), short word with sign extend (swse), exclusive access (ex)

15a	`DM(data32,Ia)=Ureg (lw);`
	`PM(data32,Ic)=Ureg (lw);`
	
	`Ureg=DM(data32,Ia) (lw);`
	`Ureg=PM(data32,Ic) (lw);`
		DAG1/2, pre modify, Ureg, LW option, immediate modify

15b	`DM(data7,Ia)=Ureg (lw);`
	`PM(data7,Ic)=Ureg (lw);`
	
	`Ureg=DM(data7,Ia) (lw);`
	`Ureg=PM(data7,Ic) (lw);`
		DAG1/2, pre modify, Ureg, LW option, immediate modify

16a	`DM(Ia,Mb)=data32;` `PM(Ic,Md)=data32;`	DAG1/2, post modify, immediate data
16b	`DM(Ia,Mb)=data16;` `PM(Ic,Md)=data16;`	DAG1/2, post modify, immediate data

19a	`MODIFY(Ia,data32);`
	`MODIFY(Ic,data32);`
	`Ia=MODIFY(Ia,data32);`
	`Ic=MODIFY(Ic,data32);`
	`Ia=MODIFY(Ia,data32) (sw);`
	`Ic=MODIFY(Ic,data32) (sw);`
		DAG1/2, Index Modify, with optional scaled address arithmetic: short word (sw) or normal word (nw), immediate modify

Table 6-5: DAG Instruction Types Summary (Continued)

Instruction Type	DAG Instruction Syntax	Description
	`Ia=MODIFY(Ia,data32) (nw);` `Ic=MODIFY(Ic,data32) (nw);`	

19a	`BITREV(Ia,data32);`
	`BITREV(Ic,data32);`
	`Ia=BITREV(Ia,data32);`
	`Ic=BITREV(Ic,data32);`
		DAG1/2, Bit reverse

Operating Modes

This section describes all modes related to the DAG which are enabled by a control bit in the REGF_MODE1, REGF_MODE2, and CMMR_SYSCTL registers.

Normal Word (40-Bit) Accesses

A program makes an extended-precision normal word (40-bit) access to internal memory using an access to a normal word address when that internal memory block's IMDWx bit is set (=1) for 40-bit words. The address ranges for internal memory accesses appear in the product-specific data sheet. For more information on configuring memory for extended-precision normal word accesses, see ***Extended-Precision Normal Word Addressing of Single-Data*** in the Memory chapter.

The processor core transfers the 40-bit data to internal memory as a 48-bit value, zero-filling the least significant 8 bits on stores and truncating these 8 bits on loads. The register file source or destination of such an access is a single 40-bit data register as shown in the ***Normal Word (40-Bit) Accesses*** example.

Normal Word (40-Bit) Accesses

```
bit clr MODE1 CBUFEN;
nop;
I9=0x90500;                /* start of 40-bit block 0 */
M9=1;
I5=0xB8000;                /* start of 32-bit block 1 */
M5=1;
USTAT1 = dm(SYSCTL);
bit set USTAT1 IMDW0;      /* Blk0 access 40-bit precision */
dm(SYSCTL) = USTAT1;
NOP;                       /* effect latency */
DM(I5,M5)=R0, PM(I9,M9)=R4;   /* DAG1 32-bit, DAG2 40-bit */
```

The sequencer uses 48-bit memory accesses for instruction fetches. Programs can make 48-bit accesses with the REGF_PX register moves, which default to 48 bits.

Input Sections Definition for 32/40-bit Data Access in LDF File

```
/* block 0 */
seg_pmco       /* TYPE(PM RAM) START(0x00090200) END(0x000904FF) WIDTH(48) */
seg_pmda_40    /* TYPE(PM RAM) START(0x00090500) END(0x00090FFF) WIDTH(48) */

/* block 1 */
seg_dmda_32    /* TYPE(DM RAM) START(0x000B8000) END(0x000B87FF) WIDTH(32)*/
```

Processing Unit versus Memory Load/Store Precision Accesses

The REGF_MODE1.RND32 bit and the CMMR_SYSCTL.IMDWBLK3-0 bits control how floating-point data are treated by the processing units versus L1 memory depending on the REGF_MODE1.PEYEN bit.

- REGF_MODE1.RND32 =0, CMMR_SYSCTL.IMDWBLK3-0 =0 (default). See Figure 7-17 Normal Word Addressing of Single-Data in SIMD Mode.
  - Processing Units: 40-bit boundary to/from register file (SIMD)
  - Load/Store: 32-bit floating to/from memory (SIMD)
- REGF_MODE1.RND32 =0, CMMR_SYSCTL.IMDWBLK3-0 =1. See Figure 7-21 Extended-Precision Normal Word Addressing of Dual-Data in SISD Mode.
  - Processing Units: 40-bit boundary to/from register file (SIMD)
  - Load/Store: 40-bit floating to/from memory (SISD)
- REGF_MODE1.RND32 =1, CMMR_SYSCTL.IMDWBLK3-0 =1. See Figure 7-21 Extended-Precision Normal Word Addressing of Dual-Data in SISD Mode.
  - Processing Units: 32-bit boundary to/from register file (SIMD)
  - Load/Store: 40-bit floating to/from memory (SISD)

Extended Precision Access

All 3/2* operations in the ***Operand Addressed in Non-Byte Space or Byte Space for Extended Precision Accesses*** table are assumed to implicitly perform a floor operation on the result, by rounding off the result to the lowest non-fractional value.

Note that the lw mnemonic overrides the IMDW setting as can be seen from the ***Operand Addressed in Non-Byte Space or Byte Space for Extended Precision Accesses*** table. The addresses calculated using the formulae in the above tables will be subject to force alignment as per alignment restrictions listed previously.

Also, SIMD accesses to a bank with the IMDW bit set results in the explicit access occurring irrespective of the size of the access only (consistent with legacy behavior for extended precision accesses in normal word space).

The data accessed by extended precision normal word accesses is shown in the ***Extended Precision Normal Word Access (Byte address or normal word address space)*** table, showing how 48-bit data elements are laid out contiguously in memory. By contrast, when Short Word or Byte Word accesses are performed, the low 16 bits of each 48-bit word are skipped, as shown in the ***Extended Precision Byte Word Access (Byte Address Space) and Extended Precision Short Word Access (Byte Address Space)*** tables.

Table 6-6: Extended Precision Normal Word Access (Byte address or normal word address space)

<table>
<tr><th>63-56</th><th>55-48</th><th>47-40</th><th>39-32</th><th>31-24</th><th>23-16</th><th>15-8</th><th>7-0</th></tr>
<tr><td colspan="6">EP WORD X3</td><td colspan="2">EP WORD X2 ...</td></tr>
<tr><td colspan="4">... EP WORD X2</td><td colspan="4">EP WORD X1 ...</td></tr>
<tr><td colspan="2">... EP WORD X1</td><td>EP WORD X0</td><td colspan="5"></td></tr>
</table>

Table 6-7: Extended Precision Byte Word Access (Byte Address Space)

63-56	55-48	47-40	39-32	31-24	23-16	15-8	7-0
BYTE WORD X15	BYTE WORD X14	BYTE WORD X13	BYTE WORD X12			BYTE WORD X11	BYTE WORD X10
BYTE WORD X9	BYTE WORD X8			BYTE WORD X7	BYTE WORD X6	BYTE WORD X5	BYTE WORD X4
		BYTE WORD X3	BYTE WORD X2	BYTE WORD X1	BYTE WORD X0		

Table 6-8: Extended Precision Short Word Access (Byte Address Space)

<table>
<tr><th>63-56</th><th>55-48</th><th>47-40</th><th>39-32</th><th>31-24</th><th>23-16</th><th>15-8</th><th>7-0</th></tr>
<tr><td colspan="2">SHORT WORD X7</td><td colspan="2">SHORT WORD X6</td><td></td><td></td><td colspan="2">SHORT WORD X5</td></tr>
<tr><td colspan="2">SHORT WORD X4</td><td></td><td></td><td colspan="2">SHORT WORD X3</td><td colspan="2">SHORT WORD X2</td></tr>
<tr><td></td><td></td><td colspan="2">SHORT WORD X1</td><td colspan="2">SHORT WORD X0</td><td></td><td></td></tr>
</table>

Circular Buffering Mode

The `REGF_MODE1.CBUFEN` bit enables circular buffering-a mode where the DAG supplies addresses that range within a constrained buffer length (set with an `L` register). Circular buffers start at a base address (set with a `B` register), and increment addresses on each access by a modify value (set with an `M` register).

The circular buffer enable bit (`CBUFEN`) in the `MODE1` register is cleared (= 0) at reset.

NOTE: It is recommended to statically enable the `REGF_MODE1.CBUFEN` bit. During processing the individual DAG length registers enable (L>0) or disable (L=0) circular buffering.

When using circular buffers, the DAGs can generate an interrupt on buffer overflow (wraparound). For more information, see DAG Status.

Circular buffering is defined as addressing a range of addresses which contain data that the DAG steps through repeatedly, *wrapping around* to repeat stepping through the range of addresses in a circular pattern. To address a circular buffer, the DAG steps the index pointer (`I` register) through the buffer, post-modifying and updating the index on each access with a positive or negative modify value (M register or immediate value). If the index pointer falls outside the buffer, the DAG subtracts or adds the buffer length to the index value, wrapping the index pointer back within the start and end boundaries of the buffer. The DAG's support for circular buffer addressing appears in the ***Data Address Generator (DAG) Block Diagram*** figure (see Features), and an example of circular buffer addressing appears in Circular Buffer Programming Model.

The starting address that the DAG wraps around is called the buffer's base address (B register). There are no restrictions on the value of the base address for a circular buffer.

NOTE: Circular buffering starting at any address may only use post-modify addressing.

It is important to note that the DAGs do not detect memory map overflow or underflow. If the address post-modify produces I - M < 0 or I + M > 0xFFFFFFFF, circular buffering may not function correctly. For byte space accesses, the M value in the I+M or the I-M is the scaled M value. Also, the length of a circular buffer should not let the buffer straddle the top of the memory map. For more information on the core memory map, see ***Internal Memory Space*** in the Memory chapter and the product-specific data sheet.

Circular Buffer Programming Model

As shown in the ***Circular Data Buffers With Positive Modifier*** figure, programs use the following steps to set up a circular buffer:

1. Enable circular buffering (`BIT SET MODE1 CBUFEN;`). This operation is only needed once in a program. This operation is done by default when setting up the C runtime.
2. Load the buffer's base address into the B register. This operation automatically loads the corresponding I register. If an offset is required the I register can be changed accordingly.
3. Load the buffer's length into the corresponding L register. For example, L0 corresponds to B0.
4. Load the modify value (step size) into an M register in the corresponding DAG. For example, M0 through M7 correspond to B0. Alternatively, the program can use an immediate value for the modifier.

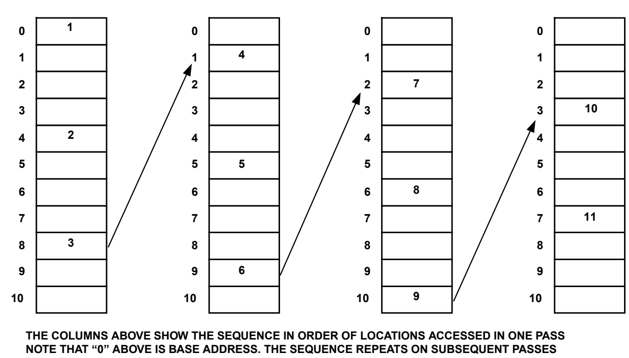

Figure 6-7: Circular Data Buffers With Positive Modifier

The ***Circular Data Buffers With Negative Modifier*** figure shows a circular buffer with the same syntax as in the ***Circular Data Buffers With Positive Modifier*** figure, but with a negative modifier (M1=-4).

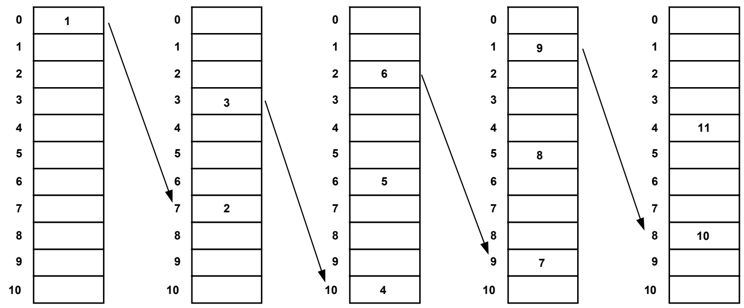

Figure 6-8: Circular Data Buffers With Negative Modifier

After circular buffering is set up, the DAGs use the modulus logic in the ***Data Address Generator (DAG) Block Diagram*** figure (in Features) to process circular buffer addressing.

NOTE: Using circular buffering with odd length in SIMD mode allows the implicit move to exceed the circular buffer limits. For example if the circular buffer requires an odd length, add one location (zero init) to the SW buffer (even count).

Wraparound Addressing

When circular buffering is enabled, on the first post-modify access to the buffer, the DAG outputs the `I` register value on the address bus then modifies the address by adding the modify value. If the updated index value is within limits of the buffer, the DAG writes the value to the `I` register. If the updated value is outside the buffer limits, the DAG subtracts (for positive `M`) or adds (for negative `M`) the `L` register value before writing the updated index value to the `I` register. In equation form, these post-modify and wraparound operations work as follows.

- If M is positive:
  - $I_{new} = I_{old} + M$ if $I_{old} + M <$ Buffer base + length (end of buffer)
  - $I_{new} = I_{old} + M - L$ if $I_{old} + M \geq$ buffer base + length
- If M is negative:
  - $I_{new} = I_{old} + M$ if $I_{old} + M \geq$ buffer base (start of buffer)
  - $I_{new} = I_{old} + M + L$ if $I_{old} + M <$ buffer base (start of buffer)

NOTE: Scaled M and L values are used for byte space access.

The DAGs use all four types of DAG registers for addressing circular buffers. These registers operate as follows for circular buffering.

- The index (`I`) register contains the value that the DAG outputs on the address bus.
- The modify (`M`) register contains the post-modify value (positive or negative) that the DAG adds to the `I` register at the end of each memory access. The `M` register can be any `M` register in the same DAG as the `I` register and does not have to have the same number. The modify value can also be an immediate value instead

of an M register. The size of the modify value, whether from an M register or immediate, must be less than the length (L register) of the circular buffer.

- The length (L) register sets the size of the circular buffer and the address range that the DAG circulates the I register through. The L register must be positive and cannot have a value greater than 2^{31} - 1. For byte accesses, the scaled length value cannot have a value greater than 2^{31} - 1. If an L register's value is zero, its circular buffer operation is disabled.
- The DAG compares the base (B) register, or the B register plus the L register, to the modified I value after each access. When the B register is loaded, the corresponding I register is simultaneously loaded with the same value. When I is loaded, B is not changed. Programs can read the B and I registers independently.

Clearing the CBUFEN bit disables circular buffering for all data load and store operations. The DAGs perform normal post-modify load and store accesses, ignoring the B and L register values. Note that a write to a B register modifies the corresponding I register, independent of the state of the CBUFEN bit.

DAG Status

The DAGs can provide buffer overflow information when executing circular buffer addressing for the I7 or I15 registers. When a buffer overflow occurs (a circular buffering operation increments the I register past the end of the buffer or decrements below the start of the buffer), the appropriate DAG updates a buffer overflow flag in a sticky status (STKYx) register. Use the BITTST instruction to examine overflow flags in the STKY register after a series of operations. If an overflow flag is set, the buffer has overflowed or wrapped around at least once. This method is useful when overflow handling is not time sensitive.

Broadcast Load Mode

The REGF_MODE1.BDCST1 and REGF_MODE1.BDCST9 bits in the control broadcast register loading. When broadcast loading is enabled, the processor core writes to complementary registers or complementary register pairs in each processing element on writes that are indexed with DAG1 register I1 (if REGF_MODE1.BDCST1 =1) or DAG2 register I9 (if REGF_MODE1.BDCST9 =1). Broadcast load accesses are similar to SIMD mode accesses in that the core transfers both an explicit (named) location and an implicit (unnamed, complementary) location. However, broadcast loading only influences writes to registers and writes identical data to these registers.

Broadcast mode is independent of SIMD mode. Broadcast load mode is a hybrid between SISD and SIMD modes that transfers dual-data under special conditions.

NOTE: Broadcast Load Mode performs memory reads only. Broadcast mode only operates with data registers (*Dreg*) or complement data registers (*CDreg*). Enabling either DAG register to perform a broadcast load has no effect on register stores or loads to universal registers (Ureg). For example

```
R0=DM(I1,M1);  /* I1 load to R0 and S0 */
S10=PM(I9,M9); /* I9 load to S10 and R10 */
```

The ***Instruction Summary Broadcast Load*** table shows examples of Broadcast load instructions.

Table 6-9: Instruction Summary Broadcast Load

Explicit, PEx Operation	Implicit, PEy operation

`Rx = dm(i1,ma);`
`Rx = pm(i9,mb);`
`Rx = dm(i1,ma), Ry = pm(i9,mb);`
	`Sx = dm(i1,ma);`
	`Sx = pm(i9,mb);`
	`Sx = dm(i1,ma), Sy = pm(i9,mb);`

NOTE: The `REGF_MODE1.PEYEN` bit (SISD/SIMD mode select) does not influence broadcast operations. Broadcast loading is particularly useful in SIMD applications where the algorithm needs identical data loaded into each processing element. For more information on SIMD mode (in particular, a list of complementary data registers), see *Data Register Neighbor Pairing* in the Register Files chapter.

Bit-Reverse Mode

The bit reserve mode is useful for FFT calculations, if using a DIT (decimation in time) FFT, all inputs must be scrambled before running the FFT, thus the output samples are directly interpretable. For DIF (decimation in frequency) FFT the process is reversed. This mode automates bit reversal, no specific instruction is required.

The `REGF_MODE1.BR0` and `REGF_MODE1.BR8` bits in the enable the bit-reverse addressing mode where addresses are output in reverse bit order. When `REGF_MODE1.BR0` is set (= 1), DAG1 bit-reverses 32-bit addresses output from `I0`. When `REGF_MODE1.BR8` is set (= 1), DAG2 bit-reverses 32-bit addresses output from `I8`. The DAGs bit-reverse only the address output from `I0` or `I8`; the contents of these registers are not reversed.

The ***Bit Reverse Addressing*** example demonstrates how bit-reverse mode effects address output.

Bit Reverse Addressing

```
BIT SET MODE1 BR0;   /* Enables bit-rev. addressing for DAG1 */
IO = 0x83000         /* Loads I0 with the bit reverse of the
                        buffer's base address DM(0xC1000) */
M0 = 0x4000000;      /* Loads M0 with value for post-modify, which
                        is the bit reverse value of the modifier
                        value M0 = 32 */
R1 = DM(I0,M0);      /* Loads R1 with contents of DM address
                        DM(0xC1000), which is the bit-reverse of 0x83000,
                        then post-modifies I0 for the next access with
                        (0x83000 + 0x4000000) = 0x4083000, which is the
                        bit-reverse of DM(0xC1020) */
```

SIMD Mode

When the `REGF_MODE1.PEYEN` bit is set (=1), the processors are in single-instruction, multiple-data (SIMD) mode. In SIMD mode, many data access operations differ from the default single-instruction, single-data (SISD) mode. These differences relate to doubling the amount of data transferred for each data access.

For example, processing two channels in parallel requires a more complex data layout. This complexity stems from the need for all inputs and outputs for the two channels have to be interleaved. The layout lets the even array elements represent one channel, while all odd elements represent the other channel.

DAG Transfers in SIMD Mode

Accesses in SIMD mode transfer both an explicit (named) location and an implicit (unnamed, complementary) location (the ***DAG Address vs. Access Modes*** table). The explicit transfer is a data transfer between the explicit register and the explicit address, and the implicit transfer is between the implicit register and the implicit address.

Table 6-10: DAG Address vs. Access Modes

DAG Instruction	Post-Modify		Pre-Modify (M+I, no I update)	
	Explicit Access	Implicit Access	Explicit Access	Implicit Access
SISD/40-bit	DM(Ia, Mb) PM(Ic, Md)	–	DM(Mb, Ia) PM(Md, Ic)	–
SIMD k=1 NW k=2 SW k=4 BW		DM(Ia+k, Mb) PM(Ic+k, Md)		DM(Ia+k, Mb) PM(Ic+k, Md)
Broadcast		DM(Ia, Mb) PM(Ic, Md)		DM(Mb, Ia) PM(Md, Ic)

NOTE: In SIMD mode, both aligned (explicit even address) and unaligned (explicit odd address) transfers are supported.

```
R0=DM(I1,M1);       /* I1 points to nw space */
S0=DM(I1+1,M1);     /* implicit instruction */
R10=PM(I10,M11);    /* I1 points to sw space */
S10=PM(I10+2,M11); /* implicit instruction */
```

NOTE: SIMD mode can be overridden with 40-bit mode, broadcast mode, byte word or with the long word modifier. Refer to the instruction types for more information.

The DAG registers support the bidirectional register-to-register transfers that are described in SIMD Mode. When the DAG register is a source of the transfer, the destination can be a register file data register. This transfer results in the contents of the single source register being duplicated in complementary data registers in each processing element as shown below.

```
BIT SET MODE1 PEYEN;    /* SIMD */
R5 = I8;                /* Loads R5 and S5 with I8 */
```

In SIMD mode, if the DAG register is a destination of a transfer from a register file data register source, the core executes the explicit move only on the condition in PEx becoming true, whereas the implicit move is not performed. This is also true when both the source and the destination is a DAG register.

```
BIT SET MODE1 PEYEN;   /* SIMD */
I8 = R5;               /* Loads I8 with R5 */
```

Conditional DAG Transfers in SIMD Mode

Conditions in SIMD allows programs to make memory accesses conditional. For more information, see the Program Sequencer chapter.

```
IF EQ S8 = DM(I4,M3);         /* S8 load with I4, R8 load with I4+1*/
IF NOT AV PM(I12,M13) = S12; /* I12 load with S12, I12+1 load with R12*/
```

Alternate (Secondary) DAG Registers

To facilitate fast context switching, the processor core has alternate register sets for all DAG registers. Bits in the REGF_MODE1 register control when alternate registers become accessible. While inaccessible, the contents of alternate registers are not affected by core operations. Note that there is a one cycle latency between writing to REGF_MODE1 and being able to access an alternate register set. The alternate register sets for the DAGs are described in this section. For more information on alternate data and results registers, see ***Alternate (Secondary) Data Registers*** in the Register Files chapter.

Bits in the REGF_MODE1 register can activate alternate register sets within the DAGs: the lower half of DAG1 (I, M, L, B0-3), the upper half of DAG1 (I, M, L, B4-7), the lower half of DAG2 (I, M, L, B8-11), and the upper half of DAG2 (I, M, L, B12-15). The ***DAG Primary and Alternate Registers*** figure shows the primary and alternate register sets of the DAGs.

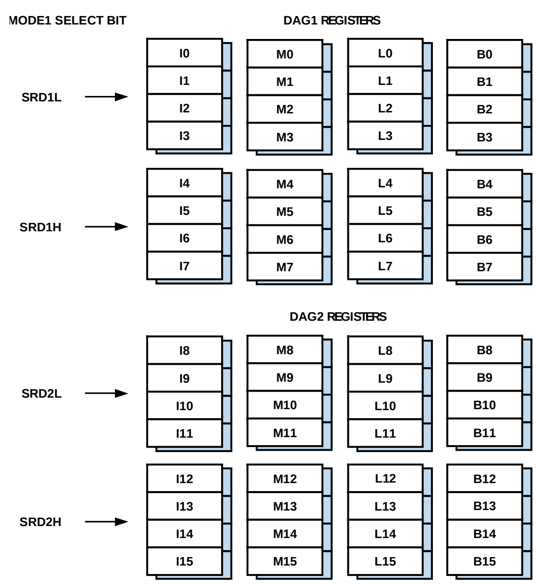

Figure 6-9: DAG Primary and Alternate Registers

To share data between contexts, a program places the data to be shared in one half of either the current data address generator's registers or the other DAG's registers and activates the alternate register set of the other half. The following examples demonstrate how the code handles the one cycle latency from the instruction that sets the bit in REGF_MODE1 to when the alternate registers may be accessed. Note that programs can use a NOP instruction or any other instruction not related to the DAG to take care of this latency.

Example 1

```
BIT SET MODE1 SRD1L; /* Activate alternate dag1 lo regs */
NOP;                 /* Wait for access to alternates */
R0 = DM(i0,m1);
```

Example 2

```
BIT SET MODE1 SRD1L; /*activate alternate dag1 lo registers */
R13 = R12 + R11;     /* Any unrelated instruction */
R0 = DM(I0,M1);
```

Interrupt Mode Mask

On the SHARC+ cores, programs can mask automated individual operating mode bits in the REGF_MODE1 register when entering into an ISR by setting bits in the REGF_MMASK register. This improves interrupt handling performance and helps ensure that interrupt handler code runs with operating modes set consistently.

For the DAGs, the alternate registers (REGF_MODE1.SRD1H/REGF_MODE1.SRD1L and REGF_MODE1.SRD2H/REGF_MODE1.SRD2L), circular buffer (REGF_MODE1.CBUFEN), bit-reverse (REGF_MODE1.BR0/REGF_MODE1.BR8) and broadcast (REGF_MODE1.BDCST1/REGF_MODE1.BDCST9) are optional masks in use. For more information, see the Program Sequencer chapter.

DAG Exceptions

The DAG exceptions are shown in the following sections. For a complete list, see the Interrupt Priority and Vector Table.

Table 6-11: DAG Exceptions

Interrupt Source	Interrupt Condition	Return Register	Return Instruction	IVT level
DAG	Circular Buffer 7 overflow	STKYx	RTI	5, CB7I
	Circular Buffer 15 overflow	STKYx	RTI	21, CB15I
	Unintentional CMMR/SMMR access	STKYx	RTI	6, IICDI
	Illegal Input Condition Detect	STKYx	RTI	6, IICDI
	Illegal Address Switch	n/a	RTI	8, ILADI

Circular Buffer Exceptions

There is one set of registers (I7 and I15) in each DAG that can generate an interrupt on circular buffer overflow (address wraparound). See DAG Status.

When a program needs to use I7 or I15 without circular buffering, and circular buffer overflow interrupts are unmasked, the program should disable the generation of these interrupts by setting the B7/B15 and L7/L15 registers to values that prevent the interrupts from occurring. If, for example, I7 is accessing the address range 0x1000 - 0x2000, the program could set B7 = 0x0000 and L7 = 0xFFFF. Because the circular buffer interrupt is based on the wraparound equations (see Wraparound Addressing), setting the `L` register to zero does not necessarily achieve the desired results. If the program is using either of the circular buffer overflow interrupts, it should avoid using the corresponding `I` register(s) (`I7` or `I15`) where interrupt branching is not needed.

There are two special situations to be aware of when using circular buffers:

1. In the case of circular buffer overflow interrupts, if `REGF_MMASK.CBUFEN` = 1 and register `L7` = 0 (or `L15` = 0), then the `CB7I` (or `CB15I`) interrupt occurs at every change of `I7` (or `I15`), after the index register (`I7` or `I15`) crosses the base register (`B7` or `B15`) value. This behavior is independent of the context of both primary and alternate DAG registers.
2. When a lw access, SIMD access, or normal word access with the `lw` option crosses the end of the circular buffer, the processor core completes the access before responding to the end of buffer condition.

Enable interrupts and use an interrupt service routine (ISR) to handle the overflow condition immediately. This method is appropriate if it is important to handle all overflows as they occur; for example in a "ping-pong" or swap I/O buffer pointers routine.

Illegal Address Space Access Exceptions

The ***Accesses Causing Illegal Address Space Interrupt*** table lists all the scenarios which results in Illegal Address Space (ILAD) interrupt.

Table 6-12: Accesses Causing Illegal Address Space Interrupt

Instruction/Quantifier	Source Address Space			
	Long Word Space	Normal Word Space	Short Word Space	Byte Word Space
Memory Access Instruction				
Byte word (example: `dm(i0,m0)=r0(bw);`)	***Illegal***	***Illegal***	***Illegal***	Legal
Short word (example: `dm(i0,m0)=r0(sw);`)	***Illegal***	***Illegal***	***Illegal***	Legal
Normal Word (example: `dm(i0,m0)=r0(nw);`)*1	Legal	Legal	Legal	Legal
Extended Precision (example: `dm(i0,m0)=r0(nw);`)	Legal	Legal	Legal	Legal
Long Word (example: `dm(i0,m0)=r0(lw);`)	Legal	Legal	***Illegal***	Legal
Modify Instruction				
Byte word (example: `i5=modify (i2,m3);`)*2	Legal	Legal	Legal	Legal
Short word (example: `i5=modify (i2,m3) (sw);`)	***Illegal***	***Illegal***	***Illegal***	Legal
Normal Word (example: `i5=modify (i2,m3) (nw);`)	***Illegal***	Legal*3	***Illegal***	Legal

Table 6-12: Accesses Causing Illegal Address Space Interrupt (Continued)

Instruction/Quantifier	Source Address Space			
	Long Word Space	Normal Word Space	Short Word Space	Byte Word Space
Address Switch Instructions				
Byte to word (example: `i5=B2W(i3);`)	***Illegal***	Legal	***Illegal***	***Illegal*** if no eqiva-lent
Word to byte (example: `i5=W2B(i3);`)	***Illegal***	Legal	***Illegal***	Legal

*1 Normal word is the default access size. No interrupt will be raised for any address space. Normal word sized access would be done in byte address space. In other address spaces, access would be as per the address space.

*2 No interrupt raised as this is the default modifier.

*3 Behavior is same as BW modifier in NW space.

Unintentional CMMR/SMMR Space Access Exceptions

Execution or data access from SMMR space can create problems, as many peripheral FIFOs are mapped in this space. To help programs detect any such accesses, the processor provides the illegal MMR access interrupt. This logic detects accesses both to core MMRs and to system MMRs. Setting the `REGF_MODE2.IIRAE` bit enables this interrupt.

Unaligned Forced Long Word Access Exceptions

The processor monitors for unaligned 64-bit memory accesses (access from two successive rows) if the unaligned 64-bit memory accesses (`REGF_MODE2.U64MAE`) bit is set (=1). Accesses not following alignment in the ***Sizes and Alignment in SISD and SIMD Modes*** table cause this interrupt. When detected, this condition is an input that can cause an illegal input condition detected interrupt if the interrupt is enabled in the `REGF_IMASK.IICDI`. For more information, see ***Mode Control 2 Register (MODE2)*** in the Registers appendix.

The following code example shows the access for even and odd addresses. When accessing an odd address, the sticky bit is set to indicate the unaligned access.

```
bit set mode2 U64MAE;    /* set bit for aligned or unaligned 64-bit access*/
r0 = 0x11111111;
r1 = 0x22222222;
pm(NW_Address1) = r0(lw);    /* even address in 32-bit, access is aligned */
pm(NW_Address2) = r0(lw);    /* odd address in 32-bit, sticky bit is set */
```

Unaligned Byte Word Access Exceptions

The following table details all the alignment requirements. Any access which does not adhere to applicable restrictions will cause IICDI (Illegal Input Condition Detected) interrupt if unaligned memory access (`REGF_MODE2.U64MAE`) is set. Such accesses are force-aligned to the immediately lower legally aligned address for the given data size.

Table 6-13: Sizes and Alignment Restrictions in SISD and SIMD Modes

Access Size	Alignment Restriction		Exclusive Accesses Restrictions	
	SISD	SIMD	SISD	SIMD
Byte	None	None	None	Short word boundary
Short Word	None	None[*1]	Short word boundary	Normal word boundary
Normal Word	None	Normal word boundary	Normal word boundary	Long word boundary
Long Word	Long word boundary	Long word boundary[*2]	None	None
External memory space short word and IMDW mode	Short word boundary	Short word boundary	None	None
External memory space normal word and IMDW mode	Normal word boundary	Normal word boundary	None	None

*1 Note that SIMD accesses using short word (SW) address space behave differently than using byte address space.

*2 Behavior similar to those in any other address space after forced alignment.

7 L1 Memory Interface

The SHARC processors contain from to 3 to 5M bits of internal RAM. This memory is organized into four independent single ported memory blocks. This organization allows greater system flexibility in regards to code, data and stack or heap allocation. For information and a block diagram about the the exact size and maximum number of data or instruction words that can fit into internal memory, see the processor-specific data sheet.

Features

The following are the memory interface features.

- Four independent internal memory blocks comprised of RAM. Contents of all the four banks can be parity protected. There is one parity bit for each byte.
- Each block can be configured for different combinations of code and data storage.
- Each block consists of eight columns and each column is 8 bits wide.
- Each block maps to separate regions in memory address space and can be accessed as 8-bit, 16-bit, 32-bit, 48-bit or 64-bit words.
- Memory aliasing allows access of same space from different word sizes.
- Block 0 has 256 addresses reserved for internal interrupt vector table (IVT). Controller jumps after interrupt latch to a specific IVT address.
- Unified memory space (both DAGs can support the same address).
- Only the end address regions of blocks are assigned to I/D cache if enabled.

While each memory block can store combinations of code and data, accesses are most efficient when the DM bus accesses data from block 1, the PM bus accesses data and instructions from block 2 and two I/O buses access data from blocks 3 and 4. Using the DM and PM buses in this way assures single-cycle execution with two data transfers where the instruction must be available in the instruction-conflict cache.

NOTE: The address map between the L1 memory blocks is not sequential.

Von Neumann Versus Harvard Architectures

Most microprocessors use a single address and a single-data bus for memory accesses. This type of memory architecture is referred to as the Von Neumann architecture. Because processors require greater data throughput than the Von Neumann architecture provides, many processors use memory architectures that have separate data and address buses for instruction and data storage. These two sets of buses let the processor retrieve data and instructions simultaneously. This type of memory architecture is called Harvard architecture.

Super Harvard Architecture

SHARC processors go a step further by using a Super Harvard architecture. This four bus architecture has two address buses and two data buses, but provides a single, unified address space for program and data storage. While the data memory (DM) bus only carries data, the program memory (PM) bus handles both instructions and data, allowing dual-data accesses in a single.

The following code examples and the ***Pipelined Execution Cycles*** table illustrate the differences between Harvard and Super Harvard capabilities.

Standard Harvard Architecture

```
Compute, r0=dm(i0,m0); /* instruction performs 2 accesses */
  /* cycle 6: Instruction Fetch conflict cache, PM Data fetch F1 */
```

Super Harvard Architecture

```
Compute, r0=dm(i0,m0), r1=pm(i8,m8); /* instruction performs 3 accesses */
       /* cycle 6: Instruction Fetch conflict cache, PM Data fetch F1 and DM Data
Fetch D2 */
       /* cycle 6: See d2 and f1 in cycle 6 in the table below */
```

The ***Pipelined Execution Cycles*** table illustrates multiple accesses in the instruction pipeline.

Table 7-1: Pipelined Execution Cycles

cycles	1	2	3	4	5	6	7	8	9	10	11	12
e2											n	n+1
m4										n	n+1	n+2
m3									n	n+1	n+2	n+3
m2								n	n+1	n+2	n+3	n+4
m1							n	n+1	n+2	n+3	n+4	n+5
d2						***n***	n+1	n+2	n+3	n+4	n+5	n+6
d1					n	n+1	n+2	n+3	n+4	n+5	n+6	n+7
f4				n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8
f3			n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9
f2		n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10

Table 7-1: Pipelined Execution Cycles (Continued)

cycles	1	2	3	4	5	6	7	8	9	10	11	12
f1	n	n+1	n+2	n+3	n+4	*n+5*	n+6	n+7	n+8	n+9	n+10	n+11

When instructions and data passing over the PM bus cause a conflict, the instruction-conflict cache resolves them using hardware that act as a third bus feeding the sequencer's pipeline with instructions.

Processor core and CMMR/SMMR accesses to internal memory are completely independent and transparent to one another. Each block of memory can be accessed by the processor core and DMA in every cycle provided the accesses are to different blocks of the memory.

Functional Description

The following sections provide detail about the processor's memory function.

The SHARC processor’s memory map appears in the product-specific data sheet. See the data sheet for address decoding of memory space.

Memory Access Types

The memory interface of processor is responsible for servicing all of the accesses that are generated by core or coming to core from outside system. The ***Access Types*** figure shows summary of all the accesses serviced by this interface and the associated ports.

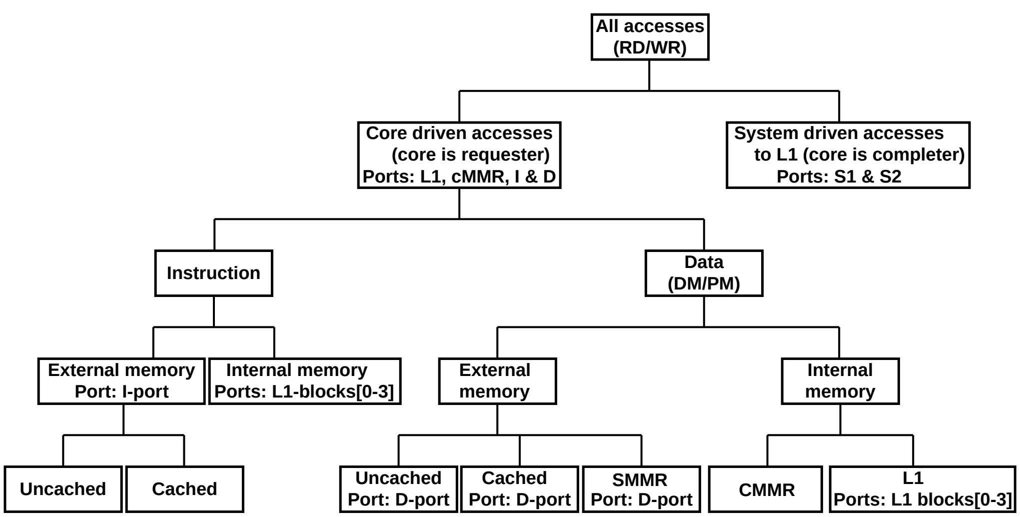

Figure 7-1: Access Types

Other than instruction accesses, all the accesses can be both read and write. Access to L1 blocks can be serviced without any pipeline stall if block conflicts are avoided. All other accesses can cause a pipeline stall. Precise number of stall depends on delay in outside system and type of completer port used. Though system related delay cannot be predicted by the core but port related components are predictable.

Byte Address Space Overview of Data Accesses

The byte address space is universal address space for the core and SoC. It has the following properties.

1. Data access of all sizes can be done using byte address space.
2. The size of an access is determined by the instruction encoding.
   - The `CMMR_SYSCTL.IMDWBLK0` and `CMMR_SYSCTL.IMDWBLK3` bits must be referred to in order to select between 32-bit and 40-bit accesses.
3. The entire system memory-map is byte-addressed. A cores byte space address map matches very closely with the address map view of other cores and access requesters (for example, DMAs).
   - All physical memory can be addressed using the byte addressable memory space. By contrast, there is some physical memory which has no corresponding normal word or short word alias.
4. Sign extension OR zero filling is based on the sign extension modifier of the instruction encoding. As a result, the sign extension mode bit is ignored for short word and byte accesses in byte space.
5. In byte space SIMD pairs are *contents of explicit address* and *contents of next location*, while in short word space SIMD pairs are alternate locations.
6. The impact of the IMDW bit on byte space accesses are slightly different than normal word space accesses. For more information, see the ***Normal-Word Access in SISD Mode*** section and the ***Normal-Word Access in SIMD Mode*** section.
7. Alignment requirements in byte space are summarized in the ***Sizes and Alignment Restrictions in SISD and SIMD Modes*** table.

The following sections describe how all sizes of internal memory accesses can be accomplished in byte space and the corresponding valid data alignments. Note each column supports 16 bits of data.

Byte Access in SISD Mode

All alignments are allowed in this mode.

	Column-3		Column-2		Column-1		Column-0	
Direction of address increment ↑	Addr=n							
				Addr=4	Addr=3	Addr=2	Addr=1	Addr=0
	← Direction of address increment							

Byte Access in SIMD Mode

Where byte access in byte space with SIMD mode is enabled, the PEX and PEY units take data from consecutive locations. The explicit register is updated with the content of the explicit address location while its SIMD pair is updated with the content of the explicit address + 1-byte memory location. Accesses of all alignments are allowed in this mode.

	Column-3		Column-2		Column-1		Column-0	
Direction of address increment (↑)				Implicit access for addr=11	Explicit access for addr=11			Implicit access for addr=7
	Explicit access for addr=7						Implicit access for addr=0	Explicit access for addr=0
	← Direction of address increment							

Short-Word Access in SISD Mode

Accesses of all alignments are allowed in this mode.

	Column-3		Column-2		Column-1		Column-0	
Direction of address increment (↑)				Access for addr=11	Access for addr=11			Access for addr=7
	Access for addr=7						Access for addr=0	Access for addr=0
	← Direction of address increment							

Short-Word Access in SIMD Mode

Where short word access in byte space with SIMD mode is enabled, the PEX and PEY unit take data from consecutive locations. Explicit register is updated with content of explicit address location while its SIMD pair gets updated with content of explicit address + 2-byte memory location. Accesses of all alignments are allowed in this mode.

	Column-3		Column-2		Column-1		Column-0	
Direction of address increment (↑)			Implicit access for addr=26	Implicit access for addr=26	Explicit access for addr=26	Explicit access for addr=26		
		Implicit access for addr=19	Implicit access for addr=19	Explicit access for addr=19	Explicit access for addr=19			
						Implicit access for addr=7	Implicit access for addr=7	Explicit access for addr=7
	Explicit access for addr=7				Implicit access for addr=0	Implicit access for addr=0	Explicit access for addr=0	Explicit access for addr=0
	← Direction of address increment							

Normal-Word Access in SISD Mode

Accesses of all the alignments are allowed in this mode.

	Column-3		Column-2		Column-1		Column-0	
Direction of address increment (↑)			Access for addr=19	Access for addr=19	Access for addr=19	Access for addr=19		
		Access for addr=19	Access for addr=19	Access for addr=19	Access for addr=19			
						Access for addr=7	Access for addr=7	Access for addr=7
	Access for addr=7				Access for addr=0	Access for addr=0	Access for addr=0	Access for addr=0
	← Direction of address increment							

32-Bit Normal-Word Access in SIMD Mode

Where normal word access in byte space with SIMD mode is enabled, the X and Y unit take data from consecutive locations. The explicit register is updated with the content of the explicit address location while its SIMD pair gets updated with the content of the “explicit address + 4-byte” memory location. In this case accesses must be aligned on normal word boundaries (byte space address = 4 n, where n = 0, 1, 2, 3 and so on).

	Column-3		Column-2		Column-1		Column-0	
Direction of address increment ↑					Implicit access for addr=24	Implicit access for addr=24	Implicit access for addr=24	Implicit access for addr=12
	Explicit access for addr=20	Explicit access for addr=20	Explicit access for addr=20	Explicit access for addr=12				
	Implicit access for addr=0	Implicit access for addr=0	Implicit access for addr=0	Implicit access for addr=0	Explicit access for addr=0	Explicit access for addr=0	Explicit access for addr=0	Explicit access for addr=0
	← Direction of address increment							

Long-Word Accesses

Long word accesses in byte space must be aligned on long word boundaries (byte space address = 8 n, where n = 0, 1, 2,and so on).

	Column-3		Column-2		Column-1		Column-0	
Direction of address increment ↑	Explicit access for addr=0	Explicit access for addr=0	Explicit access for addr=0	Explicit access for addr=0	Explicit access for addr=0	Explicit access for addr=0	Explicit access for addr=0	Explicit access for addr=0
	← Direction of address increment							

Byte Accesses to a 3 column (40-bit) enabled Block

Byte space access to an internal memory block having its `CMMR_SYSCTL.IMDWBLK0` and `CMMR_SYSCTL.IMDWBLK3` bit set behaves differently than 32-bit accesses discussed in previous sections. A summary of byte addressed accesses with IMDW set are as follows:

1. Address arithmetic on byte addresses using normal word accesses or using the modify (nw) instruction scales the modifier by 6 as there are 6 bytes per word.
2. Only 4 bytes (out of 6) can be accessed using the byte modifier.
   a. Byte n accesses the 3rd byte of a given 6-byte chunk. Similarly n + 1 goes to the 4th byte and finally n+3 goes to the 6th byte.
   b. This way least significant (unused) byte and the 2nd byte remain inaccessible in this mode.
   c. However each and every byte becomes accessible to byte access as soon as IMDW is turned off (as shown in previous sub-section).

Table 7-2: Extended Precision Normal Word Access (Byte address or normal word address space)

63-56	55-48	47-40	39-32	31-24	23-16	15-8	7-0
EP WORD X3						EP WORD X2 ...	
... EP WORD X2				EP WORD X1 ...			
... EP WORD X1		EP WORD X0					

3. Only 4 bytes (out of 6) can be accessed using the SW modifier.
   a. A short word access starting with byte address n accesses the 3rd and 4th bytes of a given 6-byte chunk. A short word access starting with byte address n + 2 accesses the 5th and 6th bytes of a given 6-byte chunk.
   b. This way least significant (unused) byte and 2nd byte remain inaccessible in this mode.
   c. However each and every byte is accessible to byte access as soon as IMDW is turned off (as shown in previous sub-section).
4. Normal word accesses the entire 6-bytes and ignores the least significant unused byte to create 40-bit data.
5. Long word accesses in byte space memory override the IMDW setting (see Byte Accesses to a 3 column (40-bit) enabled Block.
6. SIMD accesses to bank with the IMDW bit set results only in the explicit access occurring irrespective of the size of the access.

Internal Memory Space

The SHARC processor's internal memory address space is divided into four SRAM banks and one Core-MMR group. See the memory-map in the product specific data sheet for exact address details.

Internal Memory Interface

The internal memory interface is responsible for all address and strobe generation for internal memory accesses. It also performs the necessary 48-bit address rotation, pin multiplexing and other interface tasks for instruction fetch or 40-bit data access. All data writes to the internal memory blocks pass through a shadow write FIFO logic. Apart from performing a memory access, the interface also performs bus-switching for the various buses. The crossbar switches between the data memory bus (DM), program memory bus (PM), completer 1 (S1) and completer 2 bus (S2) to the single ported memory blocks.

Requester Ports

The SHARC core has two 32-bit bidirectional requester ports: a 64-bit PM port is used to fetch instruction or data and a 64-bit DM port used for data transfers.

The requester ports are used when the core performs a system access into the cross bar.

Completer Ports

The SHARC core has two 32-bit bidirectional completer ports. The ports can be used by any external requester to access any amount of data from the core's L1 memory. Some important points related to the completer ports.

- Both the ports are 32-bit wide and run at system clock speed (SYSCLK).
- L1 memory accesses cannot be performed when the core is in reset.
- Read and write requests that occur at the same time on the same port causes arbitration within that completer port.
- Both of the completer ports share same arbitration logic with core accesses. When one completer access collides with a core access on any of the internal memory banks, the other completer port also sees the bandwidth reduction.
- Completer ports do not return an error response for unpopulated spaces within the address range. Accesses to unpopulated memory space should be avoided because the access may be mapped to some other space.

NOTE: For ADSP-SC58x based products two system requesters can concurrently access two completer ports (for address map refer to product DS)

For ADSP-SC57x based products completer port 2 is hard wired for the Max BW MDMA. Therefore any system requester on completer port 1 can have concurrent access with the Max BW MDMA

WARNING: Speculative read accesses launched on a pipeline flush may lead the system to hang under certain conditions.

The SHARC+ core launches all non MMR reads speculatively based on the current values of the index and modifier registers. Speculative accesses are the accesses which are launched ahead of their execute stage. These accesses can be killed when the pipe is flushed or during an abort such as when a condition is false. When a pipeline flush occurs after a branch or a loop for example, MMR reads can launch extra accesses based on a Ix = Ix + My operation or on the stale value of index registers. If a MMR read lands on a memory interface that is not functional (either not initialized or is blocked by the SMPU), then such accesses may hang the system.

Programs should use SMPU instances to disable accesses to system memory that may not be populated or needs to be initialized before being accessed. That way, when attempting to speculatively access non-populated/non-activated memory completers, the system receives a protection-violation response rather than hanging the system. This avoids the possibility of an infinite stall in the system due to a speculative access to a disabled or uninitialized memory. The preload and init code executables provided in CCES for the ADSP-SC584 and ADSP-SC589 EZ-Board have code that disables unused DMC, PCIe and SMC memory using the SMPU. See Illegal System Accesses Conditions for more information.

Internal Memory Block Architecture

The internal memory of the processor is organized as four 16-bit columns. The organization further divides each column into two bytes to support byte access. The size of the data access can be from one byte to up to 64-bits as follows:

- 0.5 column = 8-bit words (byte)
- 1 column = 16-bit words

- 2 columns = 32-bit words
- 3 columns = 48- or 40-bit words
- 4 columns = 64-bit words

Each block is physically comprised of four 16-bit columns. Wrapping, as shown in the ***Short Word Addressing of Single-Data in SISD Mode*** figure (see Short Word Addressing of Dual-Data in SISD Mode), is a method where memory can efficiently store different combinations of 8-bit, 16-bit, 32-bit, 48-bit or 64-bit wide words.

The width of the data word fetched from memory depends on:

- Type of address space used,
- Type of access size modifier used in instruction encoding, and
- Instruction mode (SISD or SIMD mode)

The same physical location in memory can be accessed using four different addresses.

NOTE: The memory data width access is only address space dependent and NOT on instruction type. This is very unique for SHARC processors. Memory aliasing allows to access the same physical location via different memory aliases.

Extended-precision normal word (40-bit) data is only accessible if the CMMR_SYSCTL.IMDWBLK0 and CMMR_SYSCTL.IMDWBLK3 bits are set. It is left-justified within a three column location using bits 478 of the location.

Normal Word Space 48-bit or 40-Bit Word Rotations

When the processor core addresses memory, the word width of the access determines how many columns within the memory are accessed. For instruction word (48 bits) or extended-precision normal word data (40 bits), the word width is 48 bits, and the processor accesses the memory's 16-bit columns in groups of three. Because these sets of three column accesses are packed into a 4 column matrix, there are four possible rotations of the columns for storing 40- or 48-bit data. The three column word rotations within the four column matrix appear in the ***48-Bit Word Rotations*** figure.

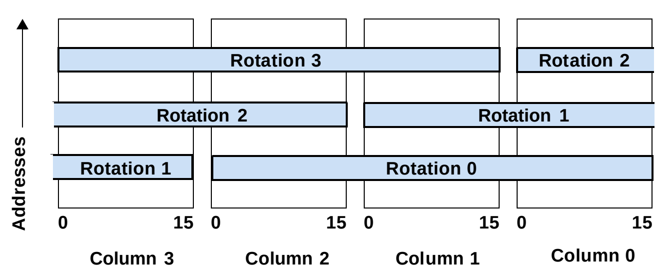

Figure 7-2: 48-Bit Word Rotations

Extended precision floating-point (40-bit) data and instruction fetches (48-bit) need a different type of manipulation of their addresses to derive the corresponding row addresses. Since each row contains 4 columns while 48-bit words span across 3 columns, the address is multiplied by 3/2 (add address to its left-shifted version, right-shift the result by two bit-positions) to derive the first row address. The next address is the incremented version of the first one. Note that this assumes that the starting address of the 48-bit/32-bit/64-bit addresses are aligned.

For long word (64 bits), normal word (32 bits) and short word (16 bits) memory accesses accomplished using LW/NW/SW address space of memory map, the processor selects from fixed columns in memory. No rotations of words within columns occur for these data types. 16-bit and 32-bit accesses that the processor performs using the byte space of the address map may result in rotation, depending on the starting point of the accessed data.

The ***Mixed Instructions and Data with No Unused Locations*** figure in Mixing Words in Normal Word Space shows the memory ranges for each data size in the processor's internal memory.

Rules for Wrapping Memory Layout

The following sections describe memory *wrapping*, a method where programs can efficiently store different combinations of 16-bit, 32-bit, 48-bit or 64-bit wide words.

Mixing Words in Normal Word Space

The processor's memory organization lets programs freely place memory words of all sizes (see Internal Memory Block Architecture) with few restrictions (see Mixing 32-Bit Words and 48-Bit Words). This memory organization also lets programs mix (place in adjacent addresses) words of all sizes. This section discusses how to mix odd (three column) and even (four column) data words in the processor's memory.

Transition boundaries between 48-bit (three column) data and any other data size can occur only at any 64-bit address boundary within the internal memory block. Depending on the ending address of the 48-bit words, there are zero, one, or two empty locations at the transition between the 48-bit (three column) words and the 64-bit (four column) words. These empty locations result from the column rotation for storing 48-bit words. The three possible transition arrangements appear in figures ***Mixed Instructions and Data with No Unused Locations***, ***Mixed Instructions and Data With One Unused Location***, and ***Mixed Instructions and Data With Two Unused Locations***.

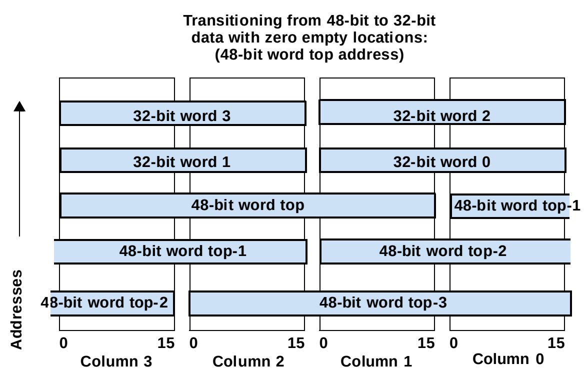

Figure 7-3: Mixed Instructions and Data with No Unused Locations

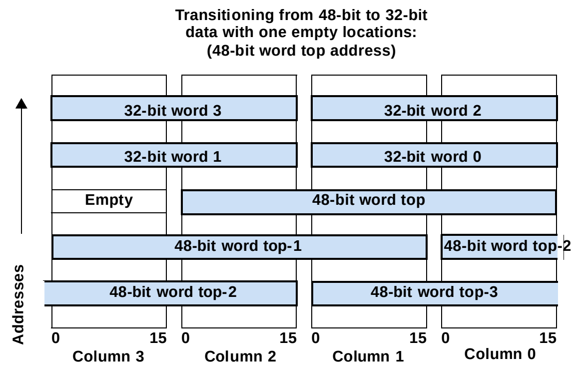

Figure 7-4: Mixed Instructions and Data With One Unused Location

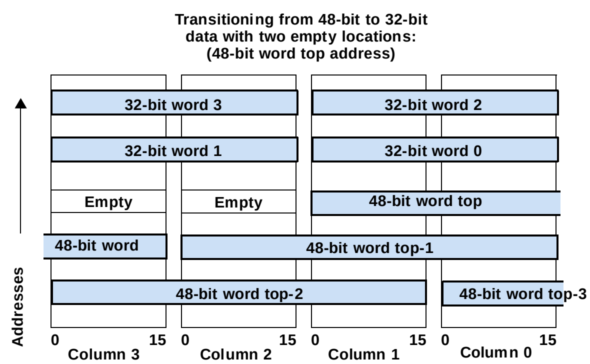

Figure 7-5: Mixed Instructions and Data With Two Unused Locations

Mixing 32-Bit Words and 48-Bit Words

There are some restrictions that stem from the memory column rotations for three column data (48 or 40-bit words) and they relate to the way that three column data can mix with two column data (32-bit words) in memory. These restrictions apply to mixing 48 and 32-bit words because the processor uses a normal word address to access both of these types of data even though 48-bit data maps onto three columns of memory and 32-bit data maps onto two columns of memory.

When a system has a range of three column (48-bit) words followed by a range of two column (32-bit) words, there is often a gap of empty 16-bit locations between the two address ranges. The size of the address gap varies with the ending address of the range of 48-bit words. Because the addresses within the gap alias to both 48 and 32-bit words, a 48-bit write into the gap corrupts 32-bit locations, and a 32-bit write into the gap corrupts 48-bit locations. The locations within the gap are only accessible with short word (16-bit) accesses.

32-Bit Word Allocation

Calculating the starting address for two column data that minimizes the gap after three column data is useful for programs that are mixing three and two column data. Given the last address of the three column (48-bit) data, the starting address of the 32-bit range that most efficiently uses memory can be determined by the equation:

```
m = B + (3/2 (n - B)) + 1)
```

where:

- ***n*** is the first unused address after the end of 48-bit words
- B is the base normal word / 48-bit address of the internal memory block
- ***m*** is the first 32-bit normal word address to use after the end of 48-bit words.

ATTENTION: Note that the linker verifies the wrapping rules of different output sections and returns an overlap error message during project build if the rules are violated.

Example: Calculating a Starting Address for 32-Bit Addresses

If, in the SHARC address map for example, the block 0 starting point of a normal word and 48-bit address is 0x90000, and given a block of words in the range 0x90000 to 0x92694, the next valid address is 0x92695. The number of 48-bit words (n) is:

```
n = 0x92695 - 0x90000 = 0x02695
```

When 0x12695 is converted to decimal representation, the result is 9877.

The base (B) normal word address of the internal memory block is 0x80000. The first 32-bit normal word address to use after the end of the 48-bit words is given by:

```
m = 0x90000 + (3/2 (9877)) + 1
m = 0x90000 + 0x039E0
m = 0x90000 + 0x039E0 = 0x939E0
```

The first valid starting 32-bit address is 0x9B9E0.

48-Bit Word Allocation

Another useful calculation for programs that are mixing two and three column data is to calculate the amount of three column data that minimizes the gap before starting four column data. Given the starting address of the two column (32-bit) data, the number of 48-bit words that most efficiently uses memory can be determined by the equation:

```
n = B + (2/3 (m - B)) - 1
```

where:

- ***m*** is the first 32-bit normal word address after the end of 32-bit words (m values falls in the valid normal word address space)
- B is the base normal word / 48-bit address of the internal memory block
- ***n*** is the address of the first 48-bit word to use after the end of 32-bit words

Memory Block Arbitration

A memory access conflict can occur when the processor attempts two or more accesses to the same internal memory block in the same cycle. When this conflict, known as a block conflict occurs, the memory interface logic resolves it according the following rules. The instruction that causes this conflict may take two or three core clock cycles to complete execution.

1. DMA access completer ports1-2 (highest priority)*
2. Core access to L1 over DM bus
3. Core access to L1 over PM bus
4. Core instruction access to L1
5. Core access (D-Cache) external memory over DM bus
6. Core access (D-Cache) external memory over PM bus
7. Core access (I-Cache) external memory over Instr. bus (lowest priority)

   * In case both DMA completer ports access the same block completer port 1 is given higher priority

During a single-cycle, dual-data access, the processor core uses the independent PM and DM buses to simultaneously access data from two memory blocks. Though dual-data accesses provide greater data throughput, it is important to note some limitations on how programs may use them. The limitations on single cycle, dual-data accesses are:

- The two pieces of data must come from different memory blocks.
- If the core accesses two words from the same memory block in a single instruction, an extra cycle is needed.
- The data access execution may not conflict with an instruction fetch operation. The PM data bus tries to fetch an instruction in every cycle. If a data fetch is also attempted over the PM bus, an extra cycle may be required depending on the availability of victim instruction in conflict cache.
- If the conflict cache contains the conflicting instruction, the data access completes in a single cycle and the sequencer uses the cached instruction. If the conflicting instruction is not in the instruction-conflict cache, an extra cycle is needed to complete the data access and cache the conflicting instruction. For more information, see ***Instruction-Conflict Cache for External Instruction Fetch*** in the Program Sequencer chapter.

For more information on how the buses access memory blocks, see Requester Ports.

VISA Instruction Arbitration

With standard arbitration processes, 48-bits of data are fetched at a time. In VISA operation, this data may either be 1, 2, or 3 instructions. This is an advantage of VISA operation-during the execution of a typical VISA application there are fewer accesses to internal memory from the core, causing less conflict on the internal buses with other peripheral DMAs or dedicated hardware accelerators using the same bus.

Using Single Ported Memory Blocks Efficiently

Because the SHARC+ cores are designed with four single-ported memory blocks, software needs to be designed so that data is continuously being processed and there are no memory block conflicts.

Typically data is pushed into memory using the DMA infrastructure. The core loads the data from memory, performs a computation, and stores the data back into memory. Then the DMA drives this data off-chip.

To ensure continuous data streams, mechanisms like ping-pong buffers, together with chained DMA transfers, can be implemented as shown in the ***DMA Flow*** figure. Designs should ensure that while the DMA moves data to the primary memory block, the core processes the secondary block's data. Then, after the DMA interrupt is generated, the memory block processing between core and DMA is flipped which prevents memory block conflicts between the core and DMA.

For complete information on using DMA, see the product-specific hardware reference, "Direct Memory Access (DMA)" chapter.

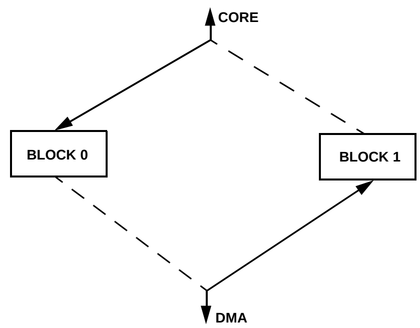

Figure 7-6: DMA versus Core Flow

Internal Memory Data Access Options (8-, 16-, 32-, 40-bit)

The processor's DM and PM buses support many combinations of register-to-memory data access options in byte-word, short-word, normal-word, and long-word address spaces. The following factors influence the data access type:

- Size of words - short word, normal word, extended-precision normal word, or long word
- Number of words - single or dual-data move
- Processor mode - SISD, SIMD, or broadcast load
- Instruction modifiers, such as long word (LW), short word (SW or SWSE) and byte word (BW or BWSE)

The following list shows the processor's possible memory transfer modes and provides a cross-reference to examples of each memory access option that stems from the processor's data access options.

These modes include the transfer options that stem from the following data access options:

- The mode of the processor: SISD, SIMD, or Broadcast Load
- The size of access words: long, extended-precision normal word, normal word, short word, or byte word
- The number of transferred words

To take advantage of the processor's data accesses to three and four column locations, programs must adjust the interleaving of data into memory locations to accommodate the memory access mode. The following guidelines provide overviews of how programs should interleave data in memory locations. For more information and examples, see ***Instruction Set Types*** in the Instruction Set Types chapter, and ***Computation Types*** in the Computation Types chapter.

- Programs can use odd or even modify values (1, 2, 3,) to step through a buffer in single- or dual-data, SISD or broadcast load mode regardless of the data word size (long word, extended-precision normal word, normal word, short word, or byte word).
- Programs should use a multiple of 2 modify values (2, 4, 6,) to step through buffers of 8-, 16- or 32-bit data using the byte address space.
- Programs should use a multiple of 4 modify values (4, 8, 12,) to step through a buffer of short word data in single- or dual-data, SIMD mode. Programs must step through a buffer twice, once for addressing even short word addresses and once for addressing odd short word addresses.
- Programs should use a multiple of 2 modify values (2, 4, 6,) to step through a buffer of normal word data in single- or dual-data SIMD mode.
- Programs can use odd or even modify values (1, 2, 3,) to step through a buffer of long word or extended-precision normal word data in single- or dual-data SIMD modes.

NOTE: Where a cross (†) appears in the `PEx` registers in any of the following figures, it indicates that the processor zero-fills or sign-extends the most significant bits of the data register while loading the byte/short word value into a 40-bit data register. Zero-filling or sign-extending depends on the state of the `SSE` bit in the `MODE1` system register. For byte/short word transfers, the least significant 8 bits of the data register are always zero.

Byte Addressing of Single-Data in SISD Mode

The ***Byte Addressing of Single-Data in SISD Mode*** figure shows the SISD single-data, byte word addressed access mode. For byte addressing, the processor treats the data buses as eight 8-bit short word lanes. The 8-bit value for the byte access is transferred using the least significant byte lane of the PM or DM data bus. The processor drives the other byte lanes of the data buses with zeros.

In SISD mode, the instruction accesses the PEx registers to transfer data from memory. This instruction accesses BYTE X0, whose short word address has "00" for its least significant two bits of address. Other locations within this row have addresses with least significant two bits of "01", "10", or "11" and select BYTE X1, BYTE X2, or BYTE X3 from memory respectively. The syntax targets register RX in PEx.

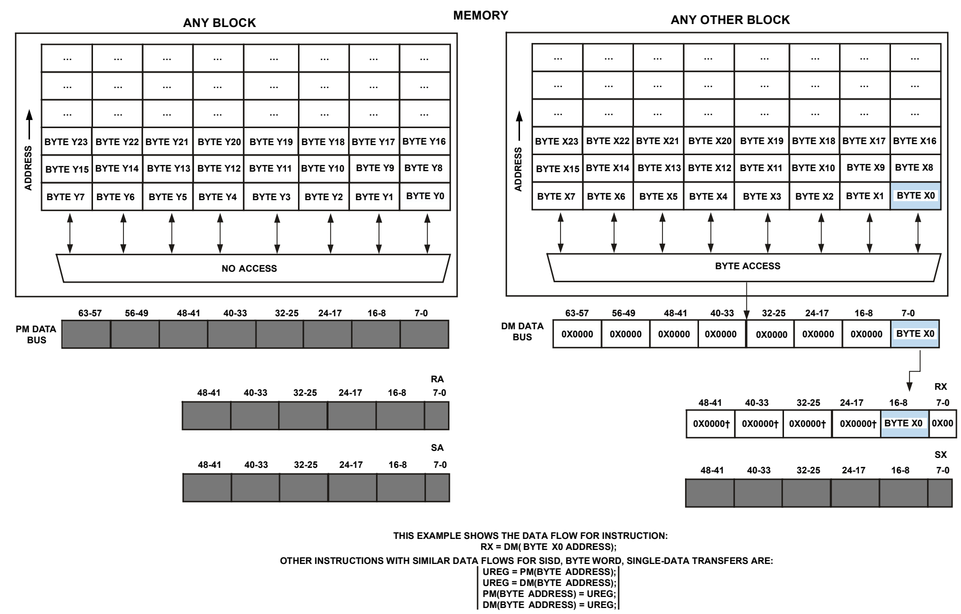

Figure 7-7: Byte Addressing of Single-Data in SISD Mode

Byte Addressing of Dual-Data in SISD Mode

The ***Byte Addressing of Dual-Data in SISD Mode*** figure shows the SISD, dual-data, byte addressed access mode. For byte addressing, the processor treats the data buses as eight 8-bit short word lanes. The 8-bit values for byte accesses are transferred using the least significant byte lanes of the PM and DM data buses. The processor drives the other byte lanes of the data buses with zeros.

In SISD mode, the instruction explicitly accesses PEx registers. This instruction accesses BYTE X0 in any block and BYTE Y0 in any other block. Each of these words has a short word address with "00" for its least significant two bits of address. Other accesses within these four column locations have addresses with their least significant two bits as "01", "10", or "11" and select BYTE X1/Y1, BYTE X2/Y2, or BYTE X3/Y3 from memory respectively. The syntax explicitly accesses registers RX and RA in PEx.

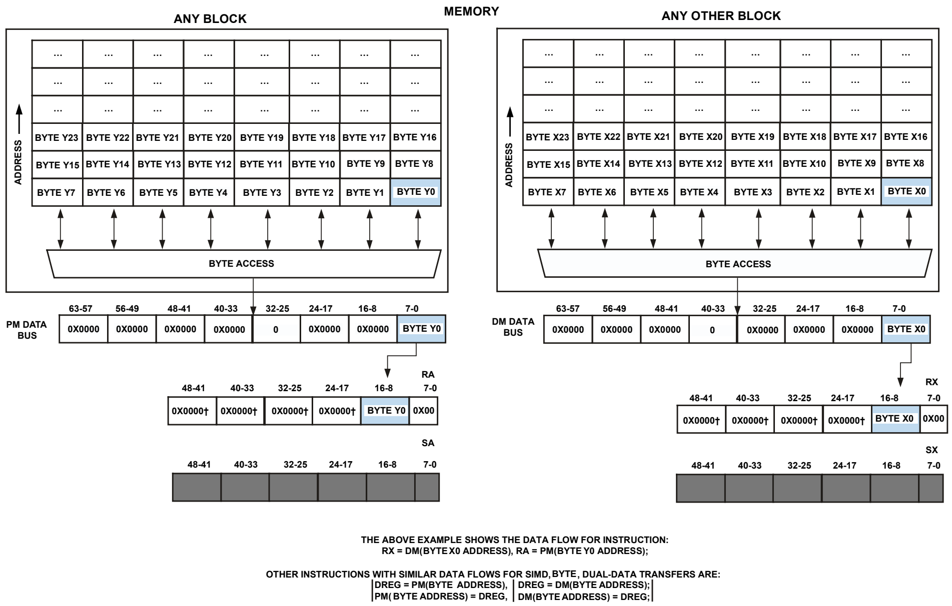

Figure 7-8: Byte Addressing of Dual-Data in SISD Mode

Byte Word Addressing of Single-Data in SIMD Mode

The *Byte Addressing of Single-Data in SIMD Mode* figure shows the SIMD, single-data, byte addressed access mode. For byte addressing, the processor treats the data buses as eight four 16-bit byte lanes. The explicitly addressed (named in the instruction) 16-bit value is transferred using the least significant byte lane of the PM or DM data bus. The implicitly addressed (not named in the instruction, but inferred from the address in SIMD mode) byte value is transferred using the 47-32 bit byte lane of the PM or DM data bus. The processor drives the other byte lanes of the PM or DM data buses with zeros (31-16 bit lane and 63-48 bit lane). The instruction explicitly accesses the register RX and implicitly accesses that register's complementary register, SX. This instruction uses a PEx register with an RX mnemonic. If the syntax named the PEy register SX as the explicit target, the processor uses that register's complement RX as the implicit target.

For more information on complementary registers, see SIMD Mode in the Processing Elements chapter.

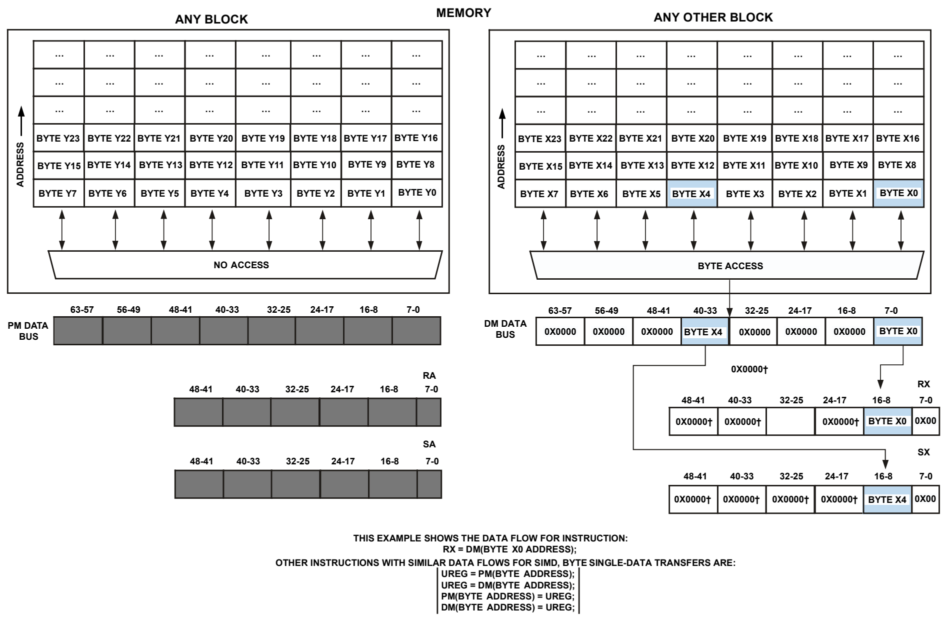

Figure 7-9: Byte Addressing of Single-Data in SIMD Mode

Byte Addressing of Dual-Data in SIMD Mode

The ***Byte Addressing of Dual-Data in SIMD Mode*** figure shows the SIMD, dual-data, byte addressed access. For byte addressing, the processor treats the data buses as four 16-bit byte lanes. The explicitly addressed 16-bit values are transferred using the least significant byte lanes of the PM and DM data bus. The implicitly addressed byte values are transferred using the 47-32 bit byte lanes of the PM and DM data buses. The processor drives the other byte lanes of the PM and DM data buses with zeros. The instruction explicitly accesses registers RX and RA, and implicitly accesses the complementary registers, SX and SA. This instruction uses PEx registers with the RX and RA mnemonics. The second word from any other block is shown as x2 on the data bus and in the Sx register. It is shown as Y2 and Y0 respectively in the left side of the block. The Sx and SA registers are transparent and look similar to Rx and RA. All bits should be shown as in Rx and RA. For more information on arranging data in memory to take advantage of byte addressing of dual-data in SIMD mode, see the ***Long Word Addressing of Dual-Data in Broadcast Load*** figure in Broadcast Load Access.

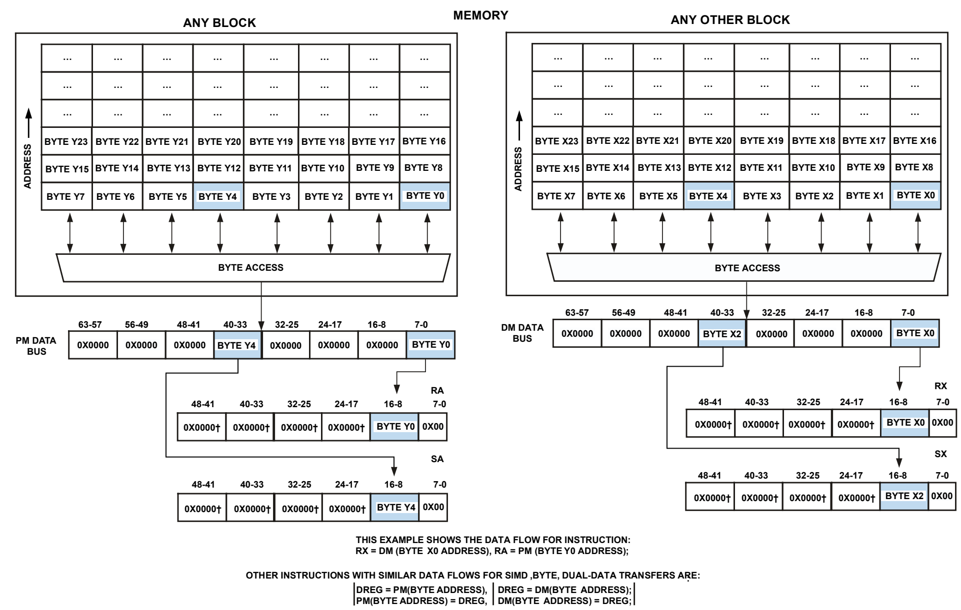

Figure 7-10: Byte Addressing of Dual-Data in SIMD Mode

Short Word Addressing of Single-Data in SISD Mode

The ***Short Word Addressing of Single-Data in SISD Mode*** figure shows the SISD single-data, short word addressed access mode. For short word addressing, the processor treats the data buses as four 16-bit short word lanes. The 16-bit value for the short word access is transferred using the least significant short word lane of the PM or DM data bus. The processor drives the other short word lanes of the data buses with zeros.

In SISD mode, the instruction accesses the `PEx` registers to transfer data from memory. This instruction accesses `WORD X0`, whose short word address has "00" for its least significant two bits of address. Other locations within this row have addresses with least significant two bits of "01", "10", or "11" and select `WORD X1`, `WORD X2`, or `WORD X3` from memory respectively. The syntax targets register `RX` in `PEx`.

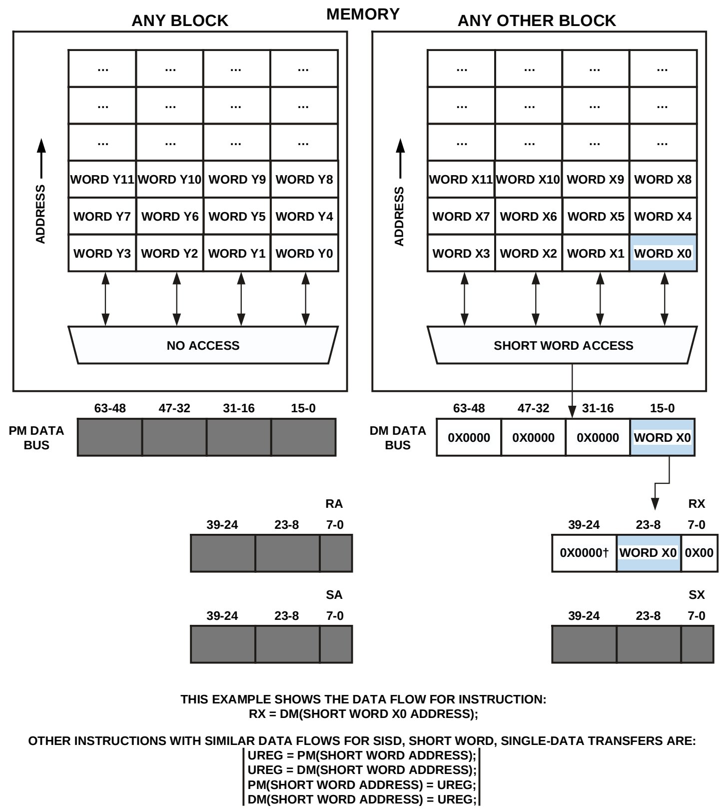

Figure 7-11: Short Word Addressing of Single-Data in SISD Mode

Short Word Addressing of Dual-Data in SISD Mode

The *Short Word Addressing of Dual-Data in SISD Mode* figure shows the SISD, dual-data, short word addressed access mode. For short word addressing, the processor treats the data buses as four 16-bit short word lanes. The 16-bit values for short word accesses are transferred using the least significant short word lanes of the PM and DM data buses. The processor drives the other short word lanes of the data buses with zeros.

In SISD mode, the instruction explicitly accesses `PEx` registers. This instruction accesses `WORD X0` in any block and `WORD Y0` in any other block. Each of these words has a short word address with "00" for its least significant two bits of address. Other accesses within these four column locations have addresses with their least significant two bits as "01", "10", or "11" and select `WORD X1/Y1`, `WORD X2/Y2`, or `WORD X3/Y3` from memory respectively. The syntax explicitly accesses registers `RX` and `RA` in `PEx`.

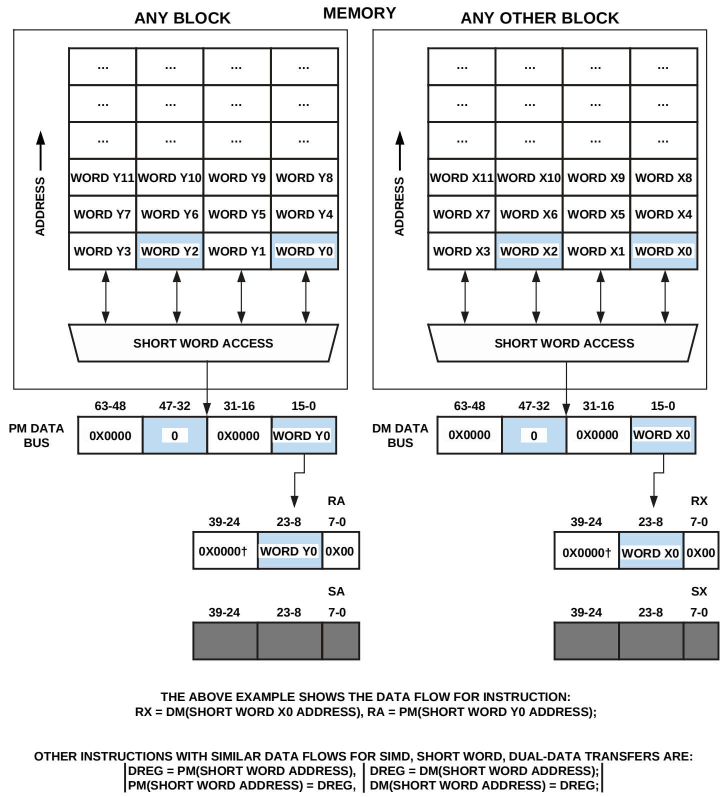

Figure 7-12: Short Word Addressing of Dual-Data in SISD Mode

Short Word Addressing of Single-Data in SIMD Mode

The ***Short Word Addressing of Single-Data in SIMD Mode*** figure shows the SIMD, single-data, short word addressed access mode. For short word addressing, the processor treats the data buses as four 16-bit short word lanes. The explicitly addressed (named in the instruction) 16-bit value is transferred using the least significant short word lane of the PM or DM data bus. The implicitly addressed (not named in the instruction, but inferred from the address in SIMD mode) short word value is transferred using the 47-32 bit short word lane of the PM or DM data bus. The processor drives the other short word lanes of the PM or DM data buses with zeros (31-16 bit lane and 63-48 bit lane).

The instruction explicitly accesses the register `RX` and implicitly accesses that register's complementary register, `SX`. This instruction uses a `PEx` register with an `RX` mnemonic. If the syntax named the `PEy` register `SX` as the explicit target, the processor uses that register's complement `RX` as the implicit target. For more information on complementary registers, see ***SIMD Mode*** in the Processing Elements chapter.

The ***Short Word Addressing of Single-Data in SIMD Mode*** figure shows the data path for one transfer. The processor accesses short words sequentially in memory. For more information on arranging data in memory to take advantage of this access pattern, see the ***Long Word Addressing of Single-Data in Broadcast Load*** figure in Broadcast Load Access.

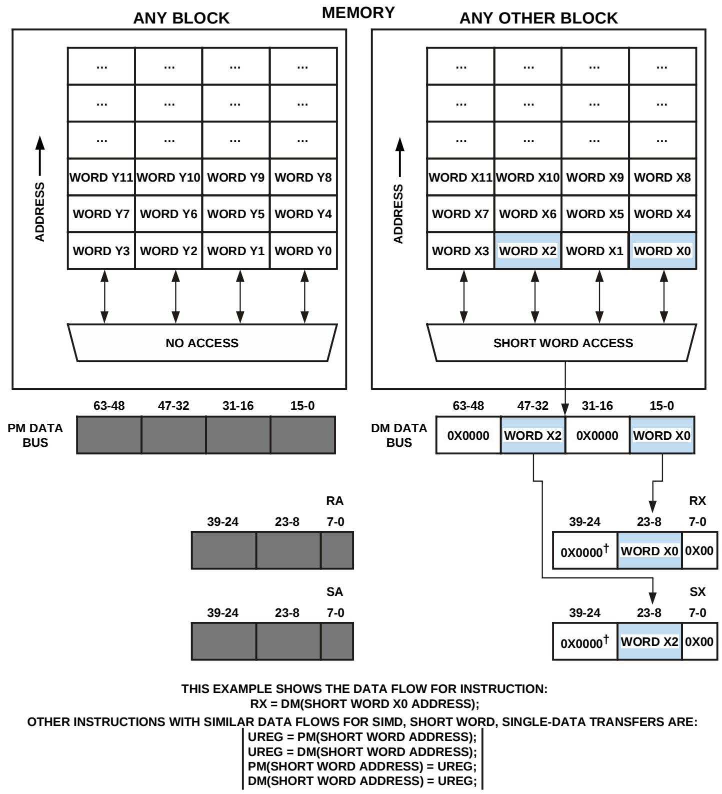

Figure 7-13: Short Word Addressing of Single-Data in SIMD Mode

Short Word Addressing of Dual-Data in SIMD Mode

The ***Short Word Addressing of Dual-Data in SIMD Mode*** figure shows the SIMD, dual-data, short word addressed access. For short word addressing, the processor treats the data buses as four 16-bit short word lanes. The explicitly addressed 16-bit values are transferred using the least significant short word lanes of the PM and DM data bus. The implicitly addressed short word values are transferred using the 47-32 bit short word lanes of the PM and DM data buses. The processor drives the other short word lanes of the PM and DM data buses with zeros.

The instruction explicitly accesses registers `RX` and `RA`, and implicitly accesses the complementary registers, `SX` and `SA`. This instruction uses `PEx` registers with the `RX` and `RA` mnemonics.

The second word from any other block is shown as `x2` on the data bus and in the `Sx` register. It is shown as `Y2` and `Y0` respectively in the left side of the block. The `Sx` and `SA` registers are transparent and look similar to `Rx` and `RA`. All bits should be shown as in `Rx` and `RA`. For more information on arranging data in memory to take advantage of short word addressing of dual-data in SIMD mode, see the ***Long Word Addressing of Dual-Data in Broadcast Load*** figure in Broadcast Load Access.

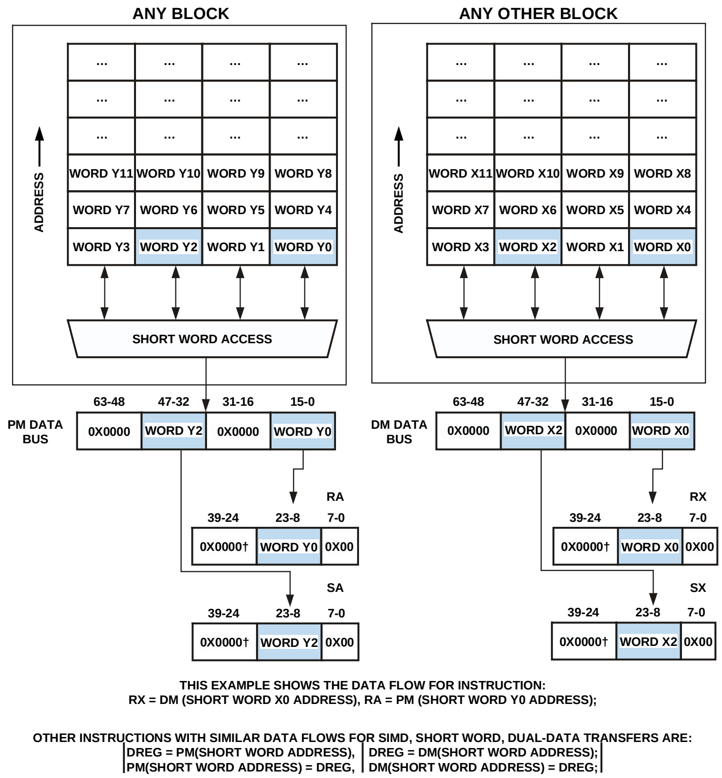

Figure 7-14: Short Word Addressing of Dual-Data in SIMD Mode

32-Bit Normal Word Addressing of Single-Data in SISD Mode

The *Normal Word Addressing of Single-Data in SISD Mode* figure shows the SISD, single-data, 32-bit normal word addressed access mode. For normal word addressing, the processor treats the data buses as two 32-bit normal word lanes. The 32-bit value for the normal word access completes a transfer using the least significant normal word lane of the PM or DM data bus. The processor drives the other normal word lanes of the data buses with zeros.

In SISD mode, the instruction accesses a `PEx` register. This instruction accesses `WORD X0` whose normal word address has "0" for its least significant address bit. The other access within this four column location has an address with a least significant bit of "1" and selects `WORD X1` from memory. The syntax targets register `RX` in `PEx`.

NOTE: For normal word accesses, the processor zero-fills the least significant 8 bits of the data register on loads and truncates these bits on stores to memory.

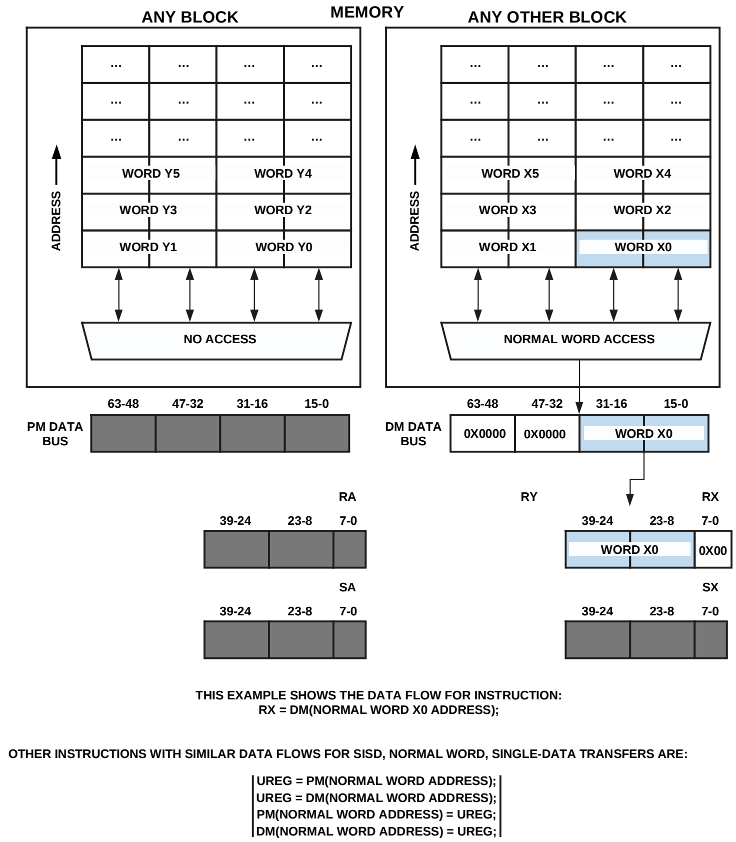

Figure 7-15: Normal Word Addressing of Single-Data in SISD Mode

32-Bit Normal Word Addressing of Dual-Data in SISD Mode

The ***Normal Word Addressing of Dual-Data in SISD Mode*** figure shows the SISD dual-data, 32-bit normal word addressed access mode. For normal word addressing, the processor treats the data buses as two 32-bit normal word lanes. The 32-bit values for normal word accesses transfer using the least significant normal word lanes of the PM and DM data buses. The processor drives the other normal word lanes of the data buses with zeros.

In the ***Normal Word Addressing of Dual-Data in SISD Mode*** figure, the access targets the `PEx` registers in a SISD mode operation. This instruction accesses `WORD X0` in any other block and `WORD Y0` in any block. Each of these words has a normal word address with 0 for its least significant address bit. Other accesses within these four column locations have addresses with the least significant bit of 1 and select `WORD X1/Y1` from memory. The syntax targets registers `RX` and `RA` in `PEx`.

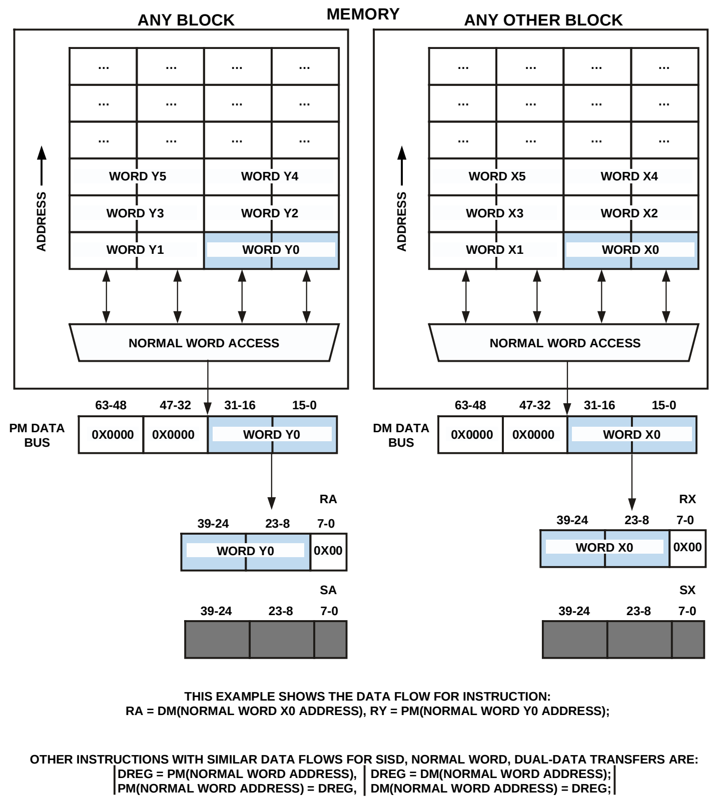

Figure 7-16: Normal Word Addressing of Dual-Data in SISD Mode

32-Bit Normal Word Addressing of Single-Data in SIMD Mode

The ***Normal Word Addressing of Single-Data in SIMD Mode*** figure shows the SIMD, single-data, normal word addressed access mode. For normal word addressing, the processor treats the data buses as two 32-bit normal word lanes. The explicitly addressed (named in the instruction) 32-bit value completes a transfer using the least significant normal word lane of the PM or DM data bus. The implicitly addressed (not named in the instruction, but inferred from the address in SIMD mode) normal word value completes a transfer using the most significant normal word lane of the PM or DM data bus.

In the ***Normal Word Addressing of Single-Data in SIMD Mode*** figure, the explicit access targets the named register `RX`, and the implicit access targets that register's complementary register, `SX`. This instruction uses a `PEx` register with an `RX` mnemonic. If the syntax named the `PEy` register `SX` as the explicit target, the processor would use that register's complement, `RX`, as the implicit target. For more information on complementary registers, see ***SIMD Mode*** in the Processing Elements chapter.

The ***Normal Word Addressing of Single-Data in SIMD Mode*** figure shows the data path for one transfer. The processor accesses normal words sequentially in memory. For more information on arranging data in memory to take advantage of this access pattern, see the ***Long Word Addressing of Dual-Data in Broadcast Load*** figure in Broadcast Load Access.

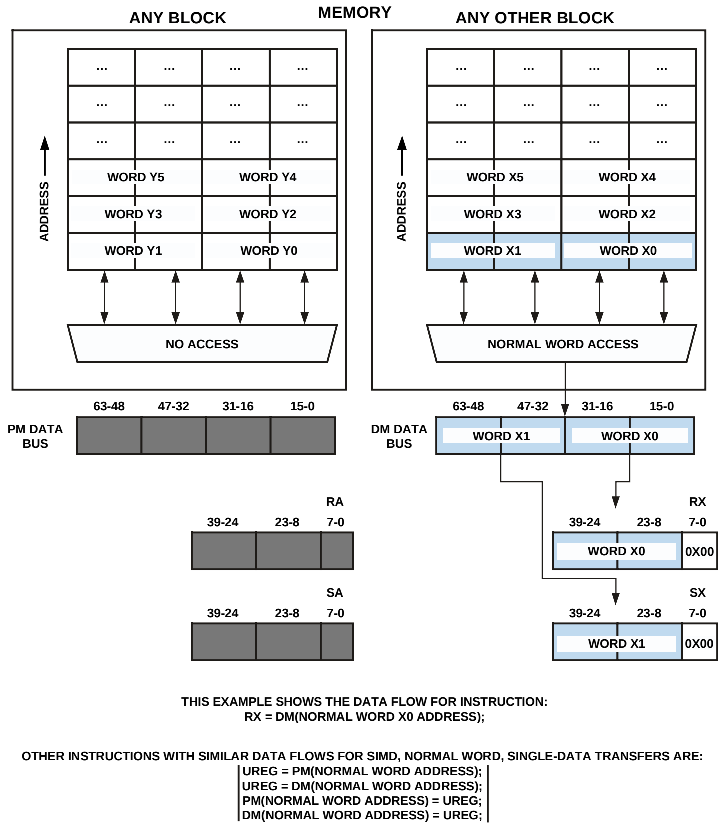

Figure 7-17: Normal Word Addressing of Single-Data in SIMD Mode

32-Bit Normal Word Addressing of Dual-Data in SIMD Mode

The ***Normal Word Addressing of Dual-Data in SIMD Mode*** figure shows the SIMD, dual-data, 32-bit normal word addressed access mode. For normal word addressing, the processor treats the data buses as two 32-bit normal word lanes. The explicitly addressed (named in the instruction) 32-bit values are transferred using the least significant normal word lane of the PM or DM data bus. The implicitly addressed (not named in the instruction, but inferred from the address in SIMD mode) normal word values are transferred using the most significant normal word lanes of the PM and DM data bus.

In the ***Normal Word Addressing of Dual-Data in SIMD Mode*** figure, the explicit access targets the named registers RX and RA, and the implicit access targets those register's complementary registers SX and SA. This instruction uses the PEx registers with the RX and RA mnemonics.

The ***Normal Word Addressing of Dual-Data in SISD Mode*** figure in 32-Bit Normal Word Addressing of Dual-Data in SIMD Mode shows the data path for one transfer. The processor accesses normal words sequentially in memory. For more information on arranging data in memory to take advantage of this access pattern, see the ***Long Word Addressing of Dual-Data in Broadcast Load*** figure in Broadcast Load Access.

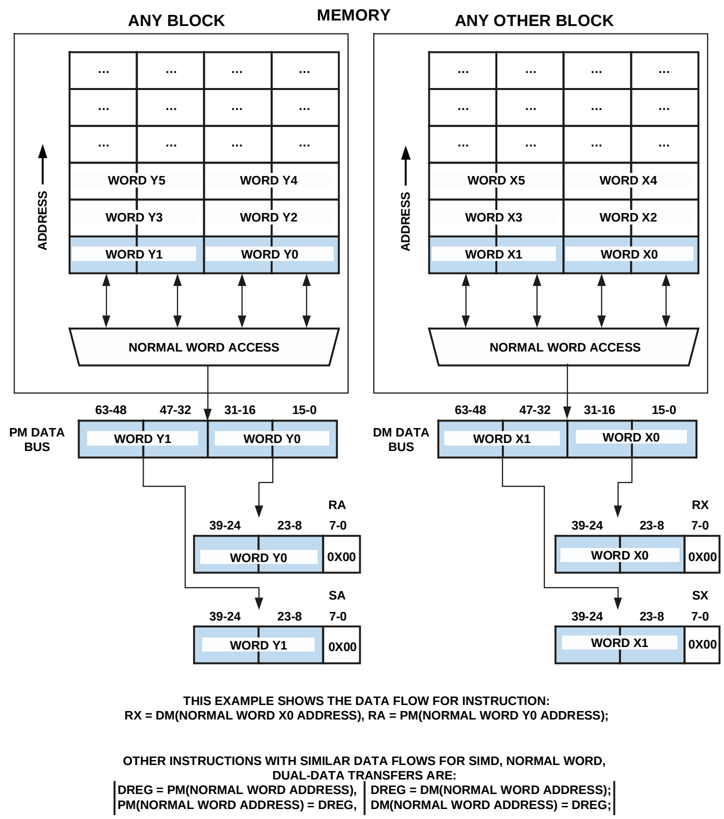

Figure 7-18: Normal Word Addressing of Dual-Data in SIMD Mode

Long Word Addressing of Single-Data

The ***Long Word Addressing of Single-Data*** figure displays one possible single-data, long word addressed access. For long word addressing, the processor treats each data bus as a 64-bit long word lane. The 64-bit value for the long word access completes a transfer using the full width of the PM or DM data bus.

In the ***Long Word Addressing of Single-Data*** figure, the access targets a `PEx` register in a SISD or SIMD mode operation. Long word single-data access operate the same in SISD or SIMD mode. This instruction accesses `WORD X0` with syntax that explicitly targets register `RX` and implicitly targets its neighbor register, `RY`, in `PEx`. The processor zero-fills the least significant 8 bits of both the registers. The example targets `PEy` registers when using the syntax `SX`. For more information on how neighbor registers work, see ***Data Register Neighbor Pairing*** in the Register Files chapter.

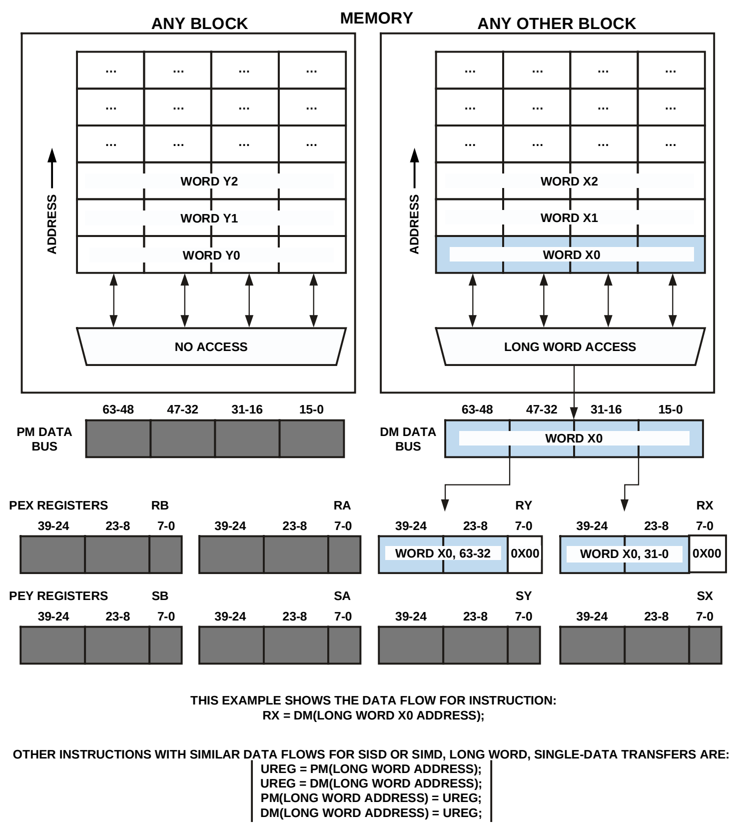

Figure 7-19: Long Word Addressing of Single-Data

Extended-Precision Normal Word Addressing of Single-Data

The ***Extended-Precision Normal Word Addressing of Single-Data*** figure displays a possible single-data, 40-bit extended-precision normal word addressed access. For extended-precision normal word addressing, the processor treats each data bus as a 40-bit extended-precision normal word lane. The 40-bit value for the extended-precision normal word access is transferred using the most significant 40 bits of the PM or DM data bus. The processor drives the lower 24 bits of the data buses with zeros.

In the ***Extended-Precision Normal Word Addressing of Single-Data*** figure, the access targets a `PEx` register in a SISD or SIMD mode operation; extended-precision normal word single-data access operate the same in SISD or SIMD mode. This instruction accesses `WORD X0` with syntax that targets register `RX` in `PEx`. The example targets a `PEy` register when using the syntax `SX`.

NOTE: Extended precision cannot be supported in SIMD mode. The PM and DM data buses are limited to 64-bits, but would require 80-bits to support this format and mode.

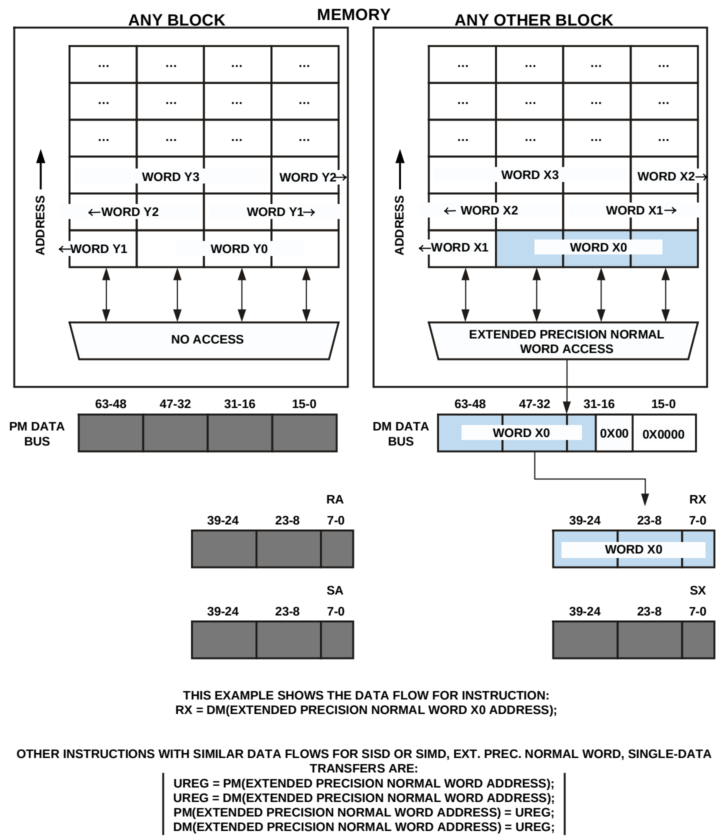

Figure 7-20: Extended-Precision Normal Word Addressing of Single-Data

Extended-Precision Normal Word Addressing of Dual-Data

The ***Extended-Precision Normal Word Addressing of Dual-Data in SISD Mode*** figure shows the SISD, dual-data, 40-bit extended-precision normal word addressed access mode. For extended-precision normal word addressing, the processor treats each data bus as a 40-bit extended-precision normal word lane. The 40-bit values for the extended-precision normal word accesses are transferred using the most significant 40 bits of the PM and DM data bus. The processor drives the lower 24 bits of the data buses with zeros.

In the ***Extended-Precision Normal Word Addressing of Dual-Data in SISD Mode*** figure, the access targets the `PEx` registers in a SISD mode operation. This instruction accesses `WORD X0` in block 1 and `WORD Y0` in block 0 with syntax that targets registers `RX` and `RY` in `PEx`. The example targets a `PEy` register when using the syntax `SX` or `SY`.

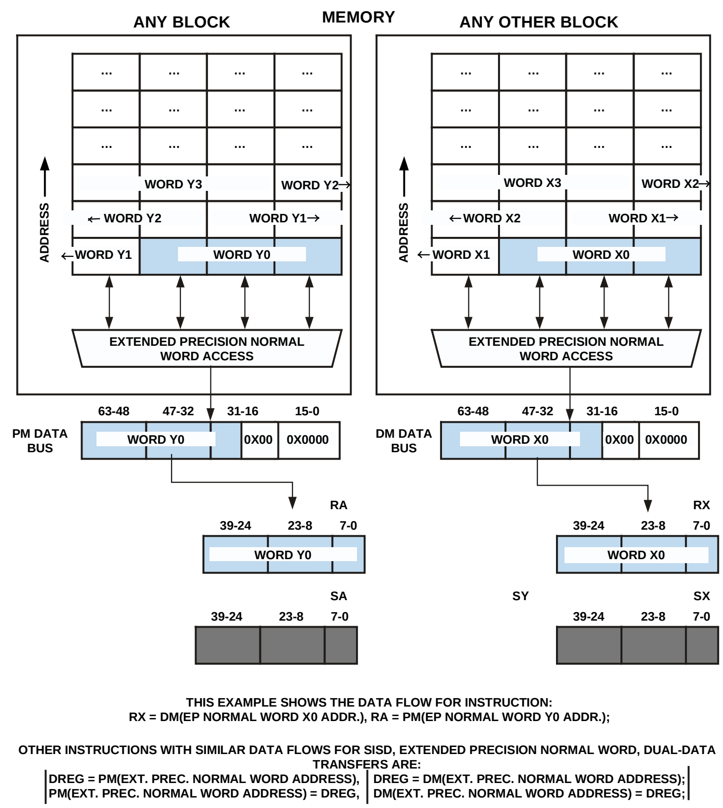

Figure 7-21: Extended-Precision Normal Word Addressing of Dual-Data in SISD Mode

Broadcast Load Access

Figures ***Short Word Addressing of Single-Data in Broadcast Load*** through ***Long Word Addressing of Dual-Data in Broadcast Load*** provide examples of broadcast load accesses for single and dual-data transfers. These read examples show that the broadcast load's to register access from memory is a hybrid of the corresponding non-broadcast SISD and SIMD mode accesses. The exceptions to this relation are broadcast load dual-data, extended-precision normal word and long word accesses. These broadcast accesses differ from their corresponding non-broadcast mode accesses.

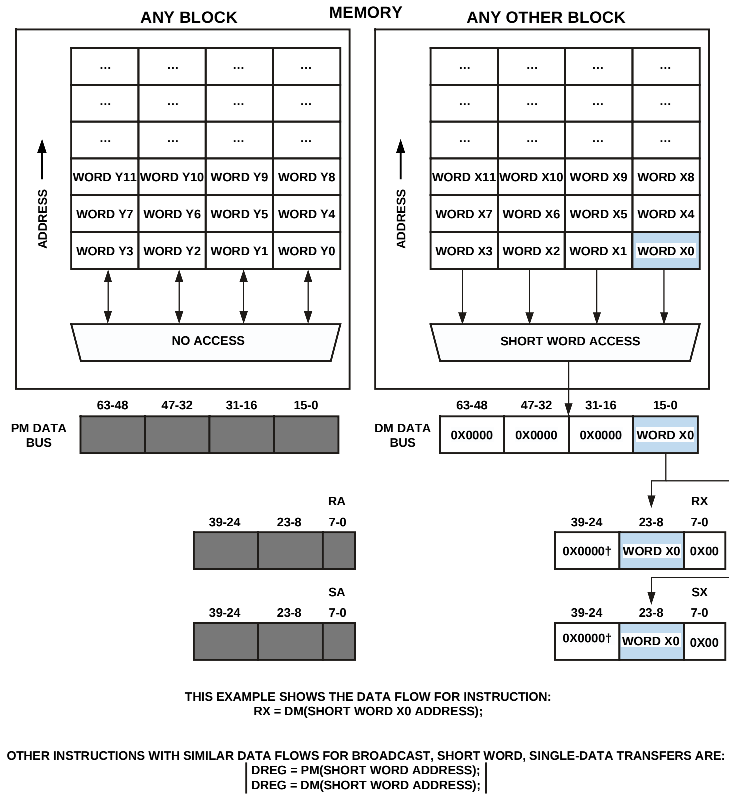

Figure 7-22: Short Word Addressing of Single-Data in Broadcast Load

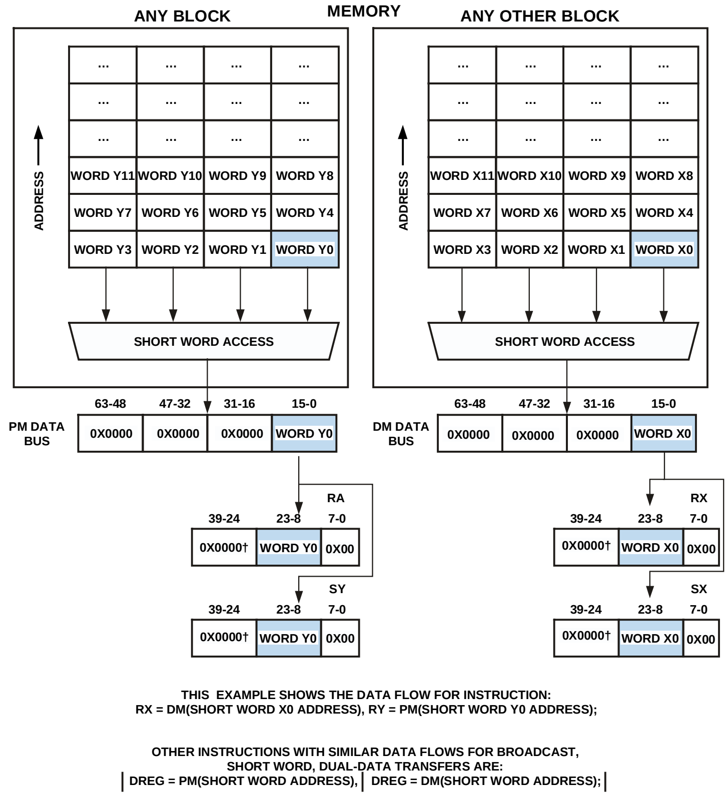

Figure 7-23: Short Word Addressing of Dual-Data in Broadcast Load

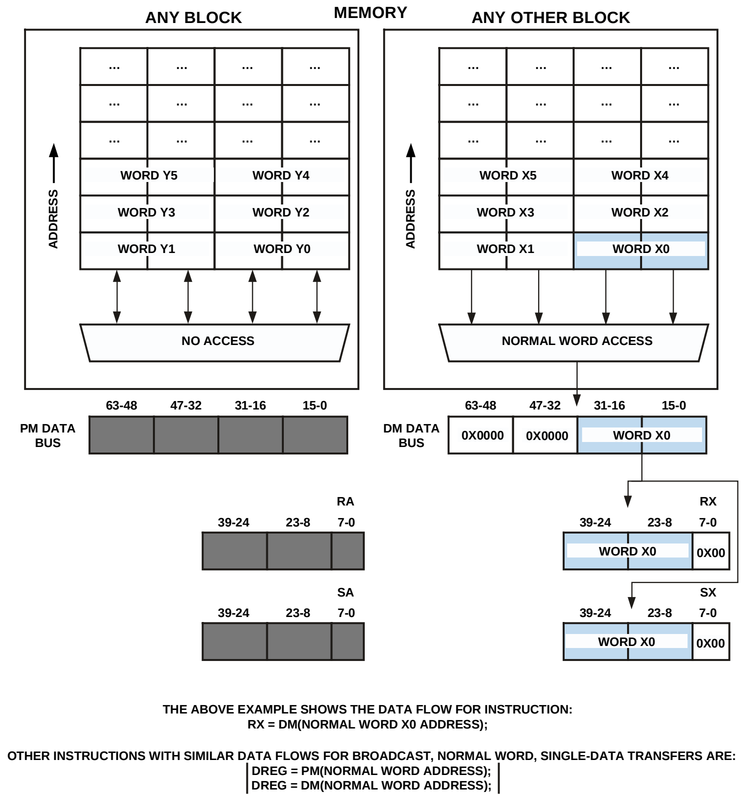

Figure 7-24: Normal Word Addressing of Single-Data in Broadcast Load

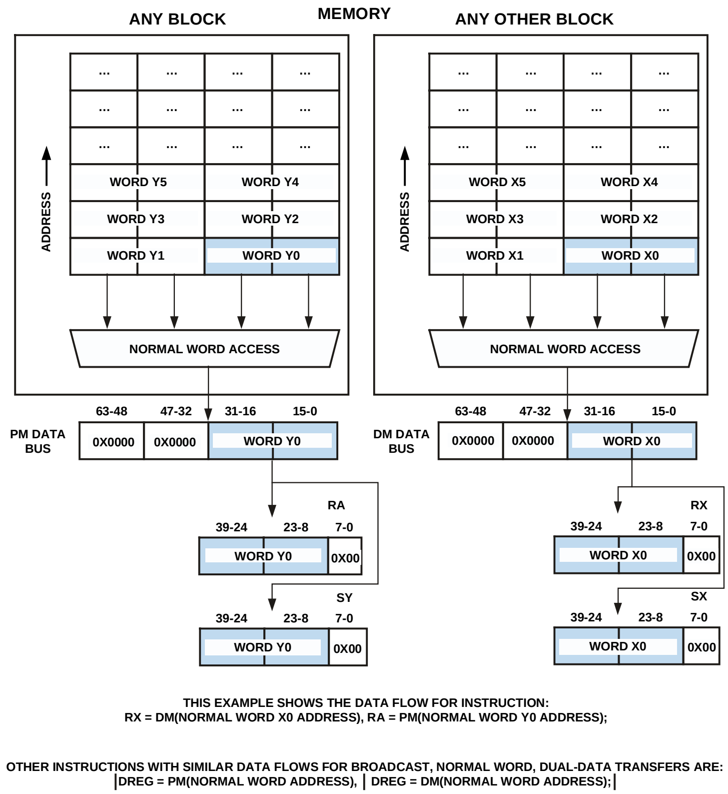

Figure 7-25: Normal Word Addressing of Dual-Data in Broadcast Load

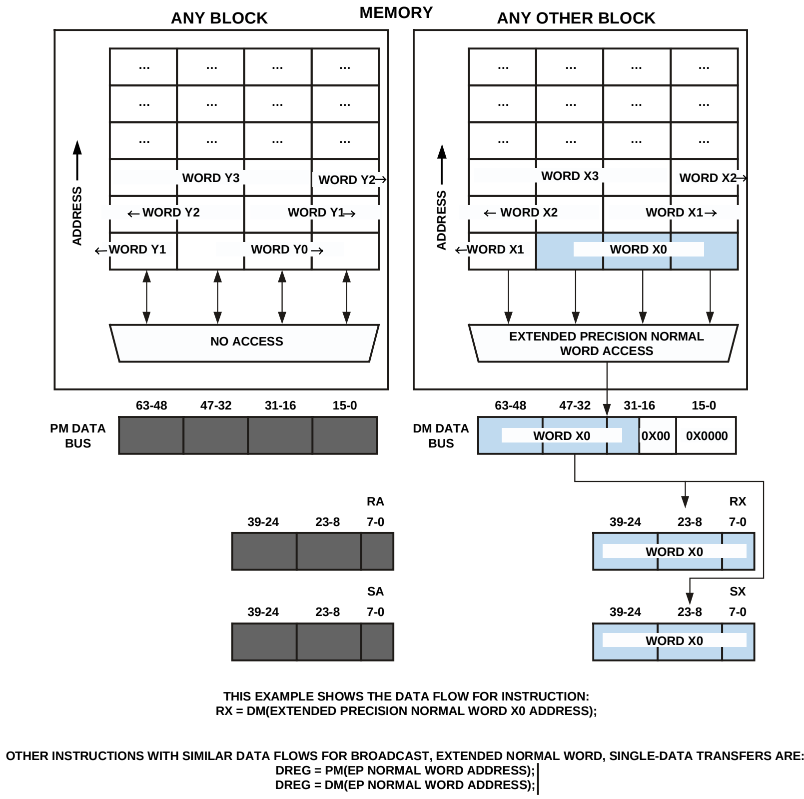

Figure 7-26: Extended-Precision Normal Word Addressing of Single-Data in Broadcast Load

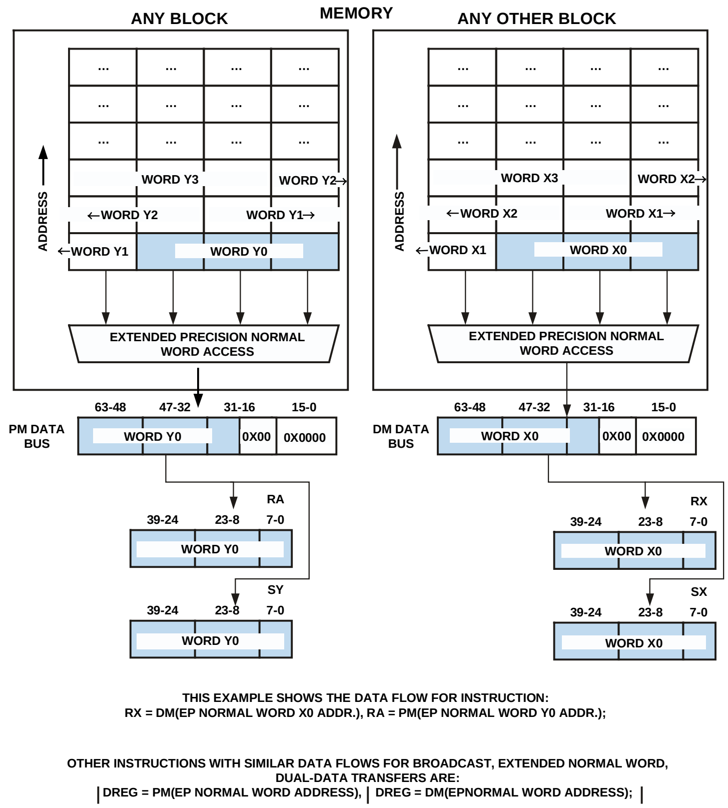

Figure 7-27: Extended-Precision Normal Word Addressing of Dual-Data in Broadcast Load

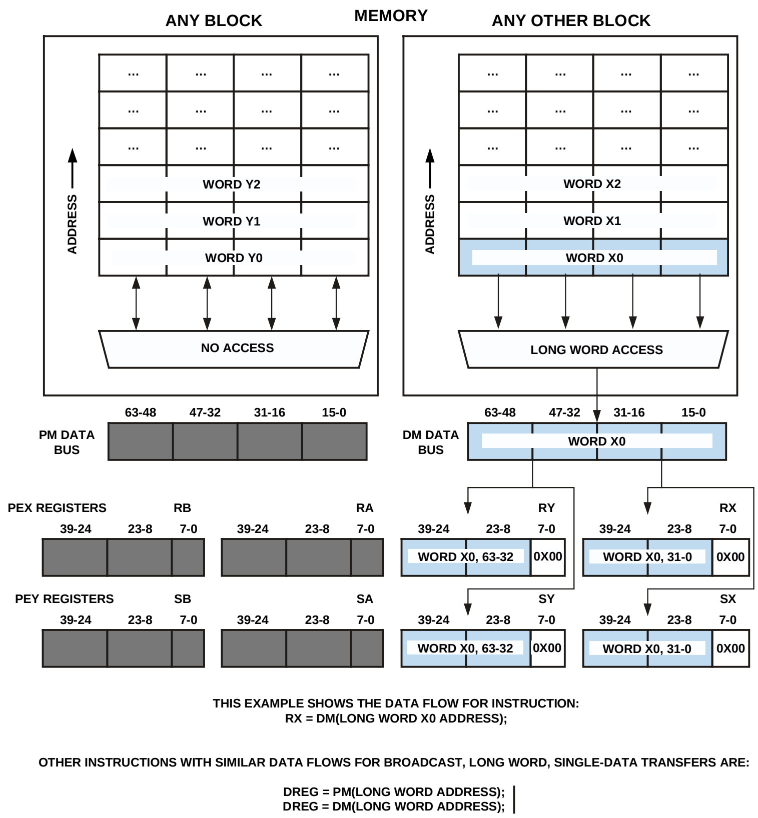

Figure 7-28: Long Word Addressing of Single-Data in Broadcast Load

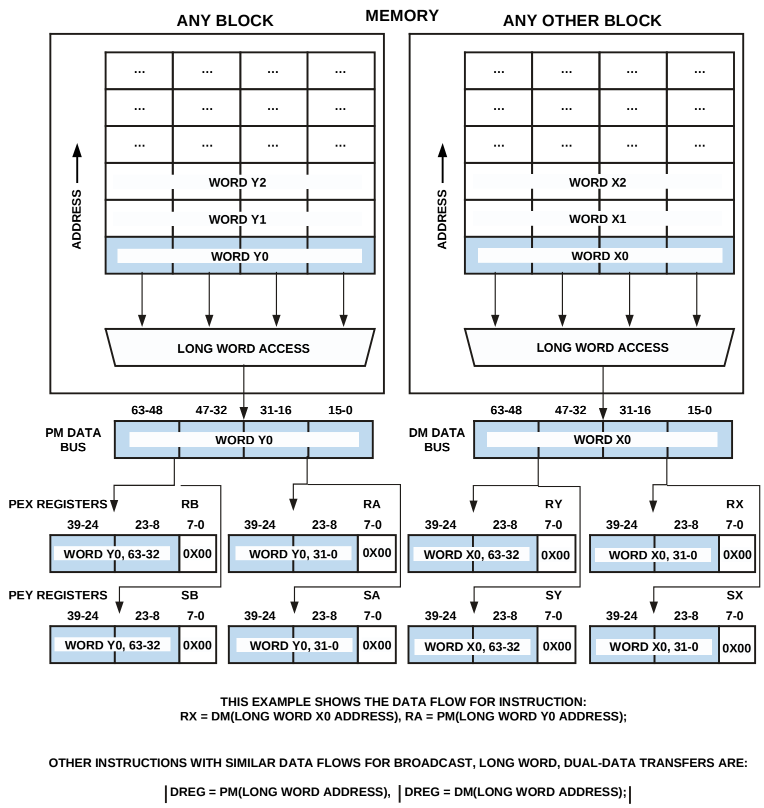

Figure 7-29: Long Word Addressing of Dual-Data in Broadcast Load

Mixed-Word Width Addressing of Long Word with Short Word

The mixed mode requires a dual data access in all cases. Modes like SISD, SIMD and Broadcast in conjunction with the address types LW, NW-40, NW-32 and SW will result in many different mixed word width access types to use in parallel between the two memory blocks.

The ***Mixed-Word Width Addressing of Dual-Data in SISD Mode*** figure shows an example of a mixed-word width, dual-data, SISD mode access. This example shows how the processor transfers a long word access on the DM bus and transfers a short word access on the PM bus.

NOTE: The assembler generates an error if the same register is written by both memory accesses in the instruction. For more information on how the processor prioritizes accesses, see *Register Files* in the Register Files chapter.

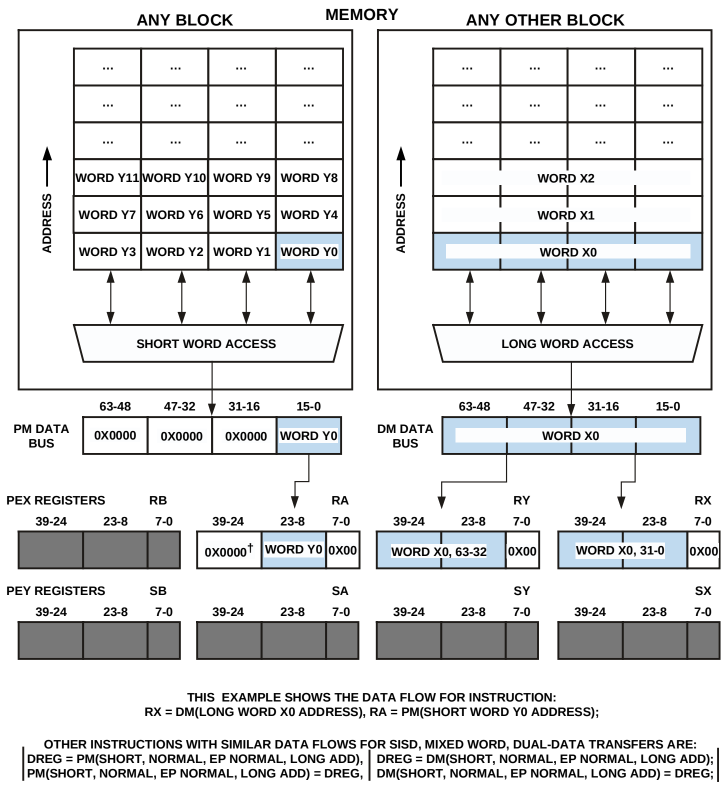

Figure 7-30: Mixed-Word Width Addressing of Dual-Data in SISD Mode

Mixed-Word Width Addressing of Long Word with Extended Word

The ***Mixed-Word Width Addressing of Dual-Data in SIMD Mode*** figure shows an example of a mixed-word width, dual-data, SISD mode access. This example shows how the processor transfers a long word access on the DM bus and transfers an extended-precision normal word access on the PM bus.

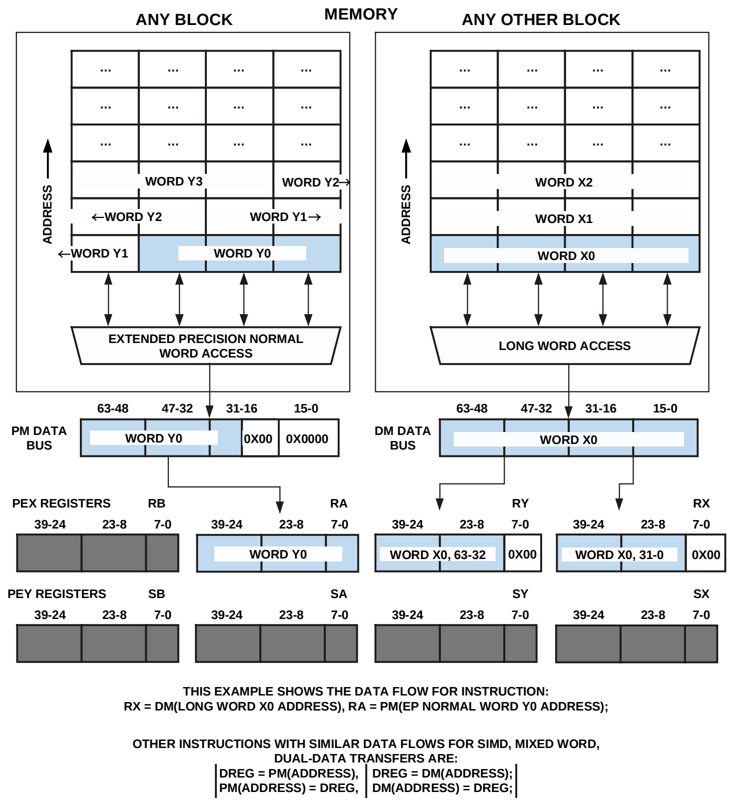

Figure 7-31: Mixed-Word Width Addressing of Dual-Data in SIMD Mode

Internal Memory Access Listings (64-bit Floating-Point)

SIMD mode for long-word or 64-bit accesses are not supported. In SIMD, the 64-bit registers can be loaded in one of these ways:

- Using two 32-bit normal-word addressing of dual-data in SIMD mode, or
- Using two long-word addressing of dual-data in SISD mode by using appropriate complementary registers in both accesses.

In both the cases, the alignment of the 64-bit data in memory could be very different.

64-bit Floating-Point Addressing of Single Data

The ***Long Word Addressing of Single-Data*** figure displays one possible single-data, long word addressed access. For long word addressing, the processor treats each data bus as a 64-bit long word lane. The 64-bit value for the long word access completes a transfer using the full width of the PM or DM data bus.

In the ***Long Word Addressing of Single-Data*** figure, the access targets a `PEx` register in a SISD or SIMD mode operation. Long word single-data access operate the same in SISD or SIMD mode. This instruction accesses `WORD X0` with syntax that explicitly targets register `RX` and implicitly targets its neighbor register, `RY`, in `PEx`. The processor zero-fills the least significant 8 bits of both the registers. The example targets `PEy` registers when using the syntax

SX. For more information on how neighbor registers work, see ***Data Register Neighbor Pairing*** in the Register Files chapter.

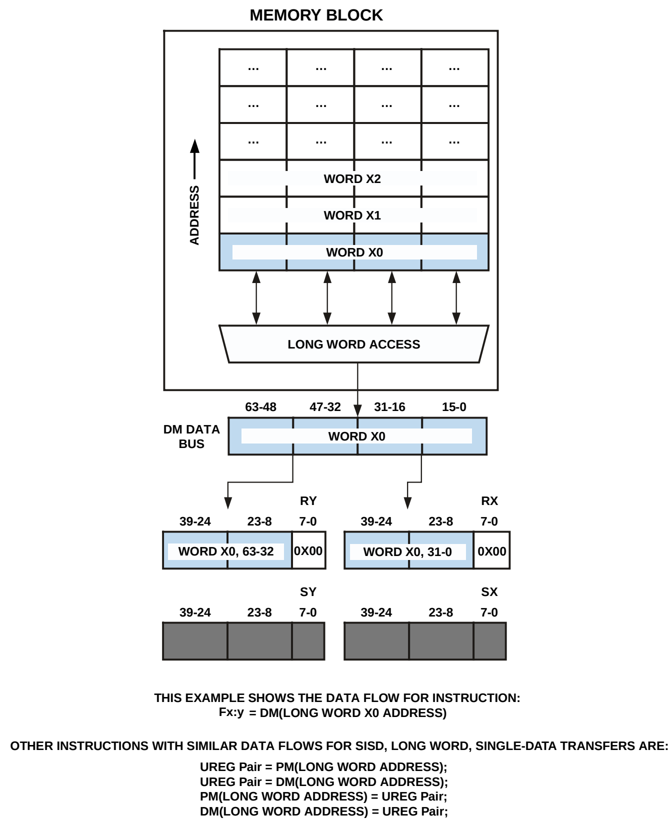

Figure 7-32: 64-bit Floating-Point Addressing of Single-Data

64-bit Floating-Point Addressing of Dual-Data in SISD Mode

The ***64-bit Floating-Point Addressing of Dual-Data*** figure shows the SISD, dual-data, long word addressed access mode. For long word addressing, the processor treats each data bus as a 64-bit long word lane. The 64-bit values for the long word accesses completes a transfer using the full width of the PM or DM data bus.

In the ***64-bit Floating-Point Addressing of Dual-Data*** figure, the access targets PEx registers in SISD mode operation. This instruction accesses WORD X0 and WORD Y0 with syntax that explicitly targets registers RX and RA and implicitly targets their neighbor registers RY and RB in PEx. The processor zero-fills the least significant 8 bits of all the registers. For more information on how neighbor registers work, see the ***Neighbor DAG Register for Long Word Accesses*** table in ***Long Word Memory Access Restrictions***, the Data Address Generators chapter.

Programs must be careful not to explicitly target neighbor registers in this instruction. While the syntax lets programs target these registers, one of the explicit accesses targets the implicit target of the other access. The processor resolves this conflict by performing only the access with higher priority. For more information on the priority order of data register file accesses, see the Register Files chapter.

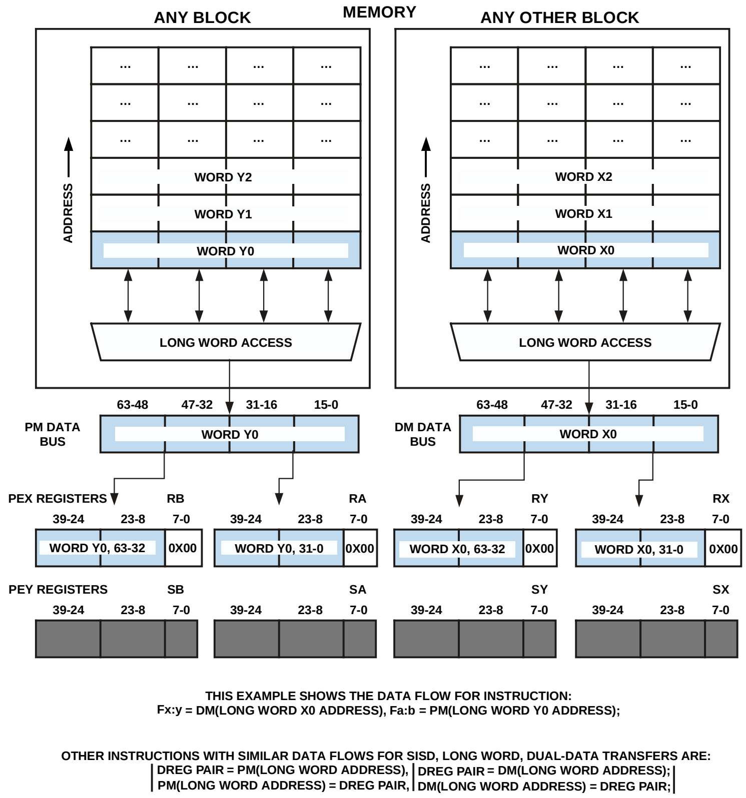

Figure 7-33: 64-bit Floating-Point Addressing of Dual-Data

64-bit Floating-Point Addressing of Dual-Data in SIMD Mode

```
Fx:y = dm(long-word address) (LW), Fa:b = pm(long-word address) (LW);
```

The following figure shows the SISD, dual-data, long word addressed access mode.

The access targets PEx registers Fx:y and Fa:b in SISD or SIMD mode. This instruction accesses WORD X0 and WORD Y0 and targets registers Fx:y and Fa:b in PEx. The least significant 8 bits of all the registers are zero-filled.

NOTE: Programs must be careful not to target the same register as destinations of both buses. The processor resolves this conflict by performing only the access with higher priority.

8 L1 Cache Controller

The SHARC+ core supports code and data storage within itself (L1), on-chip memories outside core (L2) and external memories (L3) as well. Access to L1 memories takes a single cycle whereas access to external memories (L2 or L3) takes multiple cycles. The highest performance from the SHARC+ core is achieved when the code and data storage is in on-chip L1 memory. The SHARC+ core adds on-chip data and instruction caches (D-cache and I-cache respectively) to eliminate need for software controlled overlay-based data and code management.

Features

L1 Cache gives significant performance advantage as in most of DSP applications data is located in close vicinity and the same data is reused several times (such as coefficients). In this document, both L2 and L3 accesses are referred to as external accesses.

Table 8-1: L1 Instruction Cache Operations Features

Parameter	Description
Block Size*1	Configurable – 128K bits, 256K bits, 512K bits, or 1024K bits
Associativity	Two-way
Line size	512 bits
Write policy	N/A
Replacement policy	LRU based
Supported accesses	Misaligned even/odd memory separation, ISA/VISA instructions
Additional features	Full-cache and address range based locking range based non-cacheable

*1 L1 cache uses upper portion of L1 memory block. Do not configure the cache size bigger than the block size. Cache size more than block size can not be used in product generics that have smaller memory blocks than the original product.

Table 8-2: L1 Data Cache Operation Features

Parameter	Description
Block Size*1	Configurable – 128K bits, 256K bits, 512K bits, or 1024K bits
Associativity	Two-way
Line size	512 bits

Table 8-2: L1 Data Cache Operation Features (Continued)

Parameter	Description
Write policy	Write allocate – Write back (Default policy) No write allocate – Write through*2
Replacement policy	LRU based
Supported accesses	Misaligned access additional stalls Byte, short, normal and long word accesses SISD and SIMD modes
Additional features	Full-cache and address range based Locking, write-back and invalidation. Range-based Non-cacheable and write through DM-PM cache coherencey

*1 L1 cache uses upper portion of the L1 memory block. Do not configure the cache size bigger than the block size. Cache size more than block size cannot be used in product generics that have smaller memory blocks than the original product.

*2 Non-burst/special access zones should be marked as non-cacheable.

Functional Description

The ***Memory Interface Buses and Ports*** figure shows the typical memory hierarchy of the SHARC+ core. Note that the arrows refer to the address bus only. Their direction describes the source and destination of that address.

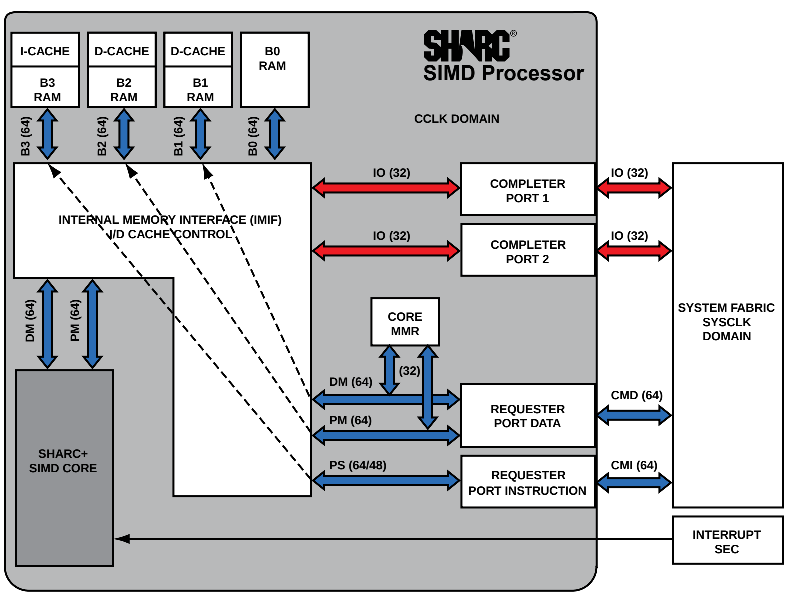

Figure 8-1: Memory Interface Buses and Ports

The cache controller uses part of on chip L1-SRAM as cache memory for its operation. Since the on-chip L1-SRAM has effectively single cycle read/write latencies, cache stores only L2 and L3 based code and data.

The *Memory Cache Signal Flow* figure shows the on chip memory system of the SHARC+ core. There are two data caches (D-cache) and one instruction cache (I-cache) per SHARC+ core. The data cache is shared with (and uses) block1 which caches all the external memory access requests from the DM bus. Similarly the other data cache is shared with (and uses) block2 which caches all external memory data access requests from the PM bus. Instruction cache is shared with (and uses) block3. In this chapter the data cache used by the DM bus is referred to as DM cache and the data cache used by the PM bus is referred to as PM cache.

The SHARC+ core supports two combinations of these caches:

1. Instruction cache mode: I-cache is enabled but DM- and PM- caches are disabled
2. Data cache mode: All three caches are enabled

NOTE: These are the only configuration options as the PM and DM caches cannot be configured independently.

Cache shares the physical memories of block1, block2 and block3. Cache sizes can be selected from 1/4 to 1 Mbits in four steps. Regular L1 accesses in those blocks should be outside of those regions used by the cache controller.

NOTE: Usage of remaining L1 space may be impacted in the following ways: Code segments should not be placed in block1 and block2 when data caches are enabled in those blocks. During certain cache operations DMAs/system requests may be delayed.

The size of each cache can be set independently.

L1 instruction and data access types are spread over four stages of core pipeline; address preprocessing and conflict generation, address to memory block, data from memory block and data merging. L1 Cache operation has to fit within this four stages of core pipeline. In case of cache hit, all the operations complete in four core clock cycles. In case of a cache miss, the fourth stage of the access takes multiple cycles.

The operation of the data cache and the instruction cache are similar. The most important difference is that instruction cache does not support writes to the cached content for the simple reason that the core can only read the instruction. The operation of the data cache is described in detail in next section. A brief description of instruction cache that highlights the differences follows.

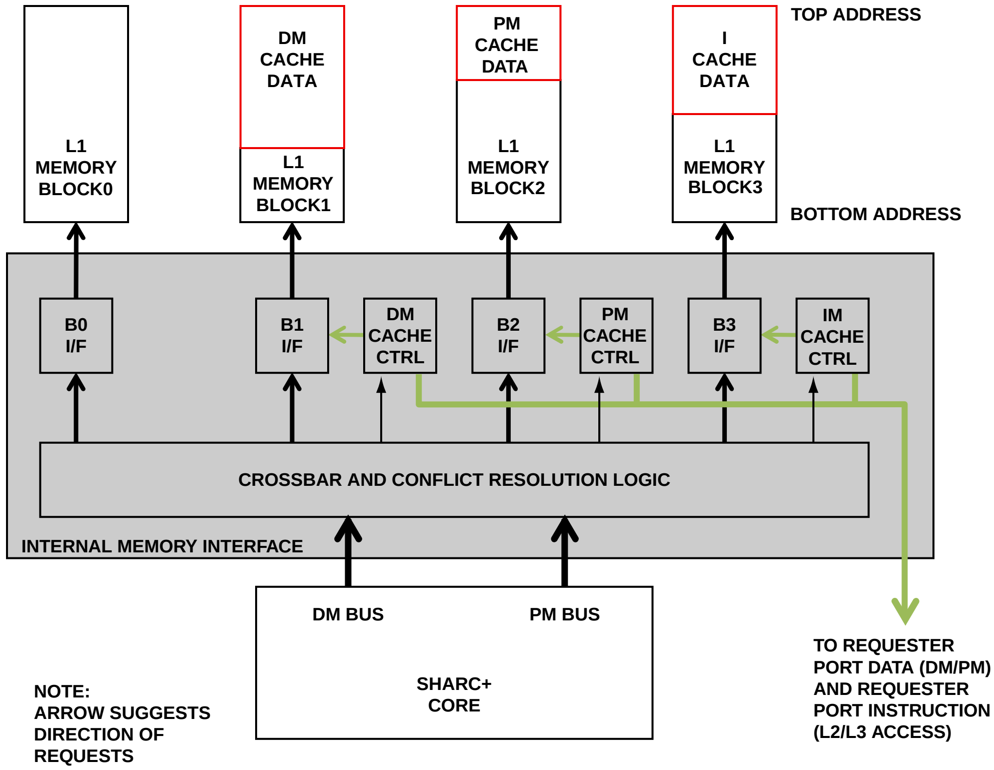

Figure 8-2: Memory Cache Signal Flows

Tag Memories

Besides access to L1 memory (instruction/data), the L1 cache controller enables additional SRAM blocks to store tag and state bits. The size of these Tag SRAM is dependent on the configuration of the cache controller.

NOTE: The Tag RAMs do not support parity protection for the ADSP-SC58x/2158x processors (unlike data/instruction SRAM).

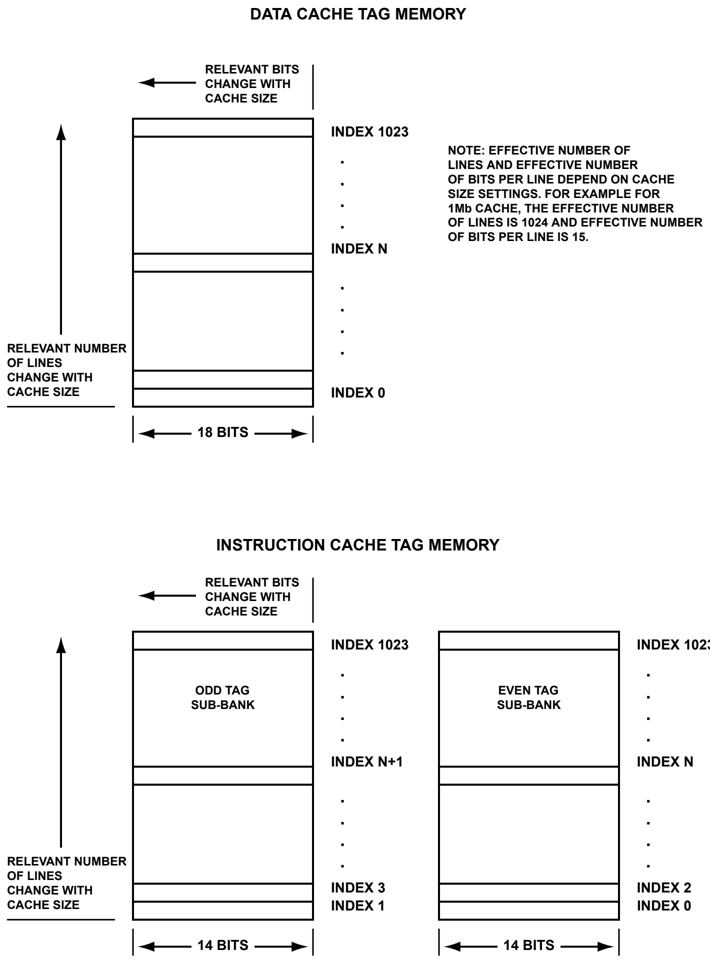

Figure 8-3: Tag Memory Blocks

Basic Cache Functionality

The address of an incoming access is first converted to an equivalent byte address (also called normalization) and then decomposed into Index, Tag and Offset, depending on Cache size.

The table shows the decomposition of the data address after it has been normalized. Since the size of lines is a fixed 512 bits, the offset field remains the same 6 bits. The size of the index field varies depending on the size of the cache – bigger caches have more sets. For a cache size of 128K bits, the line size is 512 and there are 2 ways so the number of lines per way (indexes) is 128. Therefore, the index needs 7 bits and other 19 bits comprise the tag. D Cache maintains four entries for each index.

1. Tag
2. Valid bit
3. Dirty (or modified)
4. LRU

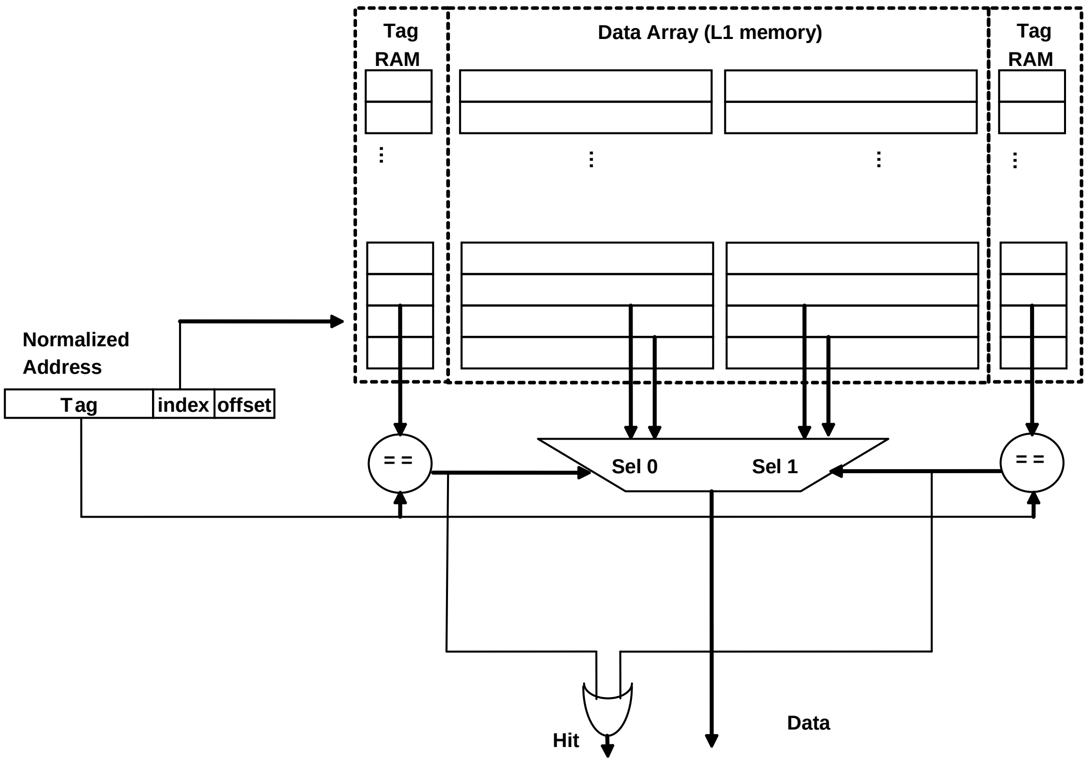

Figure 8-4: Data Cache Hit / Miss Generation Process for a Single Bank

Table 8-3: Data Cache Address Decompositions

Cache size	31...............16	15	14	13	12.....6	5....0
128K bits	Tag (Addr[31:13] - 19 bits)				Index (7 bits)	Offset
256K bits	Tag (Addr[31:14] - 18 bits)			Index (8 bits)		Offset
512K bits	Tag (Addr[31:15] - 17 bits)		Index (9 bits)			Offset
1024K bits	Tag (Addr[31:16] - 16 bits)	Index (10 bits)				Offset

Instruction Cache Features

Instruction Cache Operation

The Instruction cache (I-cache) functions like the data cache for read accesses. The SHARC+ core supports two modes of instruction: VISA and non-VISA. A VISA address increments for every short word and a non-VISA address increments for every 48-bit word. The instruction cache supports misaligned accesses for instructions straddling two cache lines.

Misaligned accesses are treated as a single access in cases of a cache hits. No stall is generated for a misaligned hit. Miss-processing may take longer if fetching two lines is required.

Data Cache Features

The following sections provide a descriptions of L1 Cache controller.

Data Cache Operations

As discussed in the previous section, the core has two data caches to support two external memory data requests per cycle. The DM cache handles access requests over the DM bus while the PM cache handles access requests over the PM bus. The DM and PM caches are functionally identical. This section discusses the details of these caches.

NOTE: D-cache does not support 40-bit floating point data.

The size of DM-cache and PM-cache can be independently set in the configuration register. Some of the additional features such as locking/invalidation cannot be independently exercised for the DM and PM cache separately but can be on the D-cache as a whole.

Cache Hit Cases

The cache hit cases are read hit and write hit.

Read Hit

For external memory data accesses data is simultaneously read from both of the cache ways in L1 RAM. Tag and valid bits are also read simultaneously. Based on the source of a hit, data is selected from one of the ways of cache. Due to this simultaneous data reads and tagging both cache ways, cache hits do not stall the pipe. L1 memory is structured to support simultaneous reads from both the ways.

Write Hit

Similar to read hits, write operations also do not stall the pipe on cache hits. The only difference is that write operations set the relevant dirty bits at the end of the accesses.

Cache Miss Cases

The cache miss cases are read miss and write miss.

Read Miss

A read miss occurs on an attempt to read data that is not available in the cache. In such cases tag matching fails and the cache returns a miss. Various actions are initiated by the cache controller to service the request and cache the relevant line for future accesses.

The following is the sequence of events that takes place after read miss detection:

1. If lines in both of the cache ways of the identified set are valid then the least recently used line is written back to the external memory (if the line is dirty) or discarded.
2. Read requests are sent to external memory for complete line fill. A request sequence starts with a critical word (the first word of a complete line (burst access)).

3. Once the complete line is received, the tag, valid, LRU and dirty bits are updated for the new line.
4. Stall release requests are sent to the core and requested data is serviced from L1 memory.
5. Cache events like misses and hits are all uninterruptible. Interrupt processing may be delayed when these events occur.

Write Miss

Write misses are processed same way as the read misses except that the write operation sets the dirty bit for the relevant line at the end of the access.

Coherency Between DM and PM Caches

The SHARC+ core has two data caches per core – one cache for an access request over the DM bus and another for an access request over the PM bus. Because both buses share same address range, both caches may cache overlapping regions of external memory. To avoid this potential overlap leading to an incoherent view of memory, where two copies of a single piece of data have different values, the L1-cache controller supports a DM-PM data coherence mechanism (also called cross-check) in hardware.

Once a cache miss is detected in the native cache, the access is launched to the remote cache. Cross checking requires two extra stall cycles in cross-check-hit (cc-hit) cases.

If DM and PM both try to access data belonging to same cache line in any type-1 instruction, and if it returns misses from both the data caches, then that access is converted to a through access to avoid creation of two copies of the line in two caches. For type-1 instructions, if the DM and PM buses try to access the same address which is already cached by a previous access, a self hit occurs first for the native cache where the data exists and then a cc-hit access occurs in the other cache.

NOTE: Unlike the SHARC core in older SHARC processors, the SHARC+ core allows the usage of the same L1 memory as both a cache memory and a regular L1 memory. This dual role introduces the potential for a bank conflict between a cached and regular L1 memory access when accessing the same resource. There are various mechanisms in the cache controller such as line fill, write back and cross check that can halt the DMA accesses to L1 memory. If there are sequential cross-check events, the DMA-to-L1 memory access can be halted for long time.

Stalling DMA accesses for peripherals without flow control for a long time can cause buffer underflow or overflow. To avoid this scenario, use the same bus (DM/PM) to access the cacheable memory regions as much as possible. If it is not possible, minimize the usage of PM accesses when the data cache is enabled (minimize the DM-PM cache overlap). Use DM accesses when possible – except in the case of type 1 accesses (or any other restrictions as applicable).

Misaligned Accesses in Data Cache

Misaligned accesses (accesses straddling two cache lines) are supported in the data cache. Eight stall cycles are generated in cases of cache hits. Miss-processing can take longer as two lines may be required to be fetched in the worst case (both lines miss). Interrupts are delayed until the entire miss-processing is completed.

Programming Model

The configuration registers enable the cache, select the cacheable area, control cache locking and select other cache features.

Programming Model for Changing Cache Configuration

- The addresses specified in the range registers (SHL1C_RANGE_START0 through SHL1C_RANGE_START7 and SHL1C_RANGE_END0 through SHL1C_RANGE_END7) must be in byte form for data caches and it should be native address for instruction cache. The property should be set in the SHL1C_CFG2 register first and range values should be filled in selected registers later.
- Range register addresses must start or end at cache line boundary.
  - Last six bits of the range register must be 0 for data caches.
  - Last five bits of the range register must be 0 for instruction cache.
- Twelve instructions after any cache MMR access should be unrelated and uncompressed (48-bit).
  - After writing to the SHL1C_CFG or SHL1C_CFG2 registers, the next twelve instructions should not contain any cache operations. This guideline includes access to non L1 locations, access to cache MMRs, or any other cache operation.
  - However separation of one instruction is sufficient while configuring start and end registers of any range register pair.
- Cache configuration or range registers should not be accessed when executing from external memory. Accesses should be unconditional and should be done through the DM bus. [*1] [*2]
- Address spaces should not be mixed in one range.
- Programs should write-back and invalidate the cache before changing its size or any other property.
- When Range registers are filled with values A and B, the effective range starts with A and finishes before B, in other words A <= range < B.

NOTE: If the system only requires the data cache, both the data cache and the instruction cache (DM and PM cache) need to be enabled.

*1 This guideline requires that range-based WBI or Invalidation operations should be done from internal memory.

*2 A loop using a cache invalidate (Range Based Write-Back-Invalidation) also accesses the SHL1C_CFG and range registers and should execute from L1 memory.

Configurable Range Registers

L1-cache controller contains a number of range register pairs to specify ranges for non-cacheability, write-through write policy selection, locking and range-based invalidation/WBI. One range register pair consists of start address register and end address register. See the *SHARC+ L1C Register Descriptions*.

Write Through Accesses

In some cases the cache controller does not perform a line-fill after detecting a miss and services the miss directly from the requested memory location (for example, L2 or DDR). The following list identifies situations when a miss access is serviced as a through access.

1. Accesses belonging to a non-cachable range
2. Write accesses belong to a write-through range
3. System MMR and exclusive accesses
4. If both cache ways are locked and have valid entries
5. For a Type 1 instruction, both DM and PM caches encounter a miss and belong to same cache line

For through accesses, the access request is forwarded to external memory only if it is found to be un-cached. Through accesses are launched after checking the conditions listed above and take more cycles than un-cached accesses.

Write Through Accesses

Write through is supported by the L1C controller for particular ranges that are defined using the range registers. If an address range that is specified by a range register is write through, all writes falling in that range reflect to the external or L2 memory.

Four range register pairs, SHL1C_RANGE_START4 through SHL1C_RANGE_START7 and SHL1C_RANGE_END4 through SHL1C_RANGE_END7 can be used to specify write through ranges.

If an access is to a write through address range, the following can occur:

- If the write access is a hit or a cc hit, the write updates both the cached copy and the external memory. The pipe is held until the write access completes.
- If the write access is a miss, the write becomes a through access and only external memory is updated. The line-fill does not occur.

Non-Cacheable Accesses

Cache controller supports non-cacheable ranges. Such ranges can be defined by using six range register pairs (SHL1C_RANGE_START2 through SHL1C_RANGE_START7 and SHL1C_RANGE_END2 through SHL1C_RANGE_END7).

Locking

The cache supports way-based locking and range-based locking.

Way-Based Locking

Locking is useful to avoid thrashing and to ensure availability of useful buffers in cache. Two ways of DM/PM/I-cache can be independently locked by setting appropriate bits in the SHL1C_CFG register. While these bits are set, a valid line in the respective way is not replaced. However, invalid cache lines of a locked way can still be filled. Priority is given to the invalid line of a locked way over LRU status while filling a cache line. This property ensures that after cache invalidation, the *needed-to-be-locked* buffer goes to *locked way only*. If both the ways are locked and invalid then way0 gets the priority.

For example if a specific code or data section needs to be locked in the cache (so that section cannot be replaced) invalidate the cache and lock one or both the ways just prior to executing that code or accessing the data section. As the instructions or data are accessed the cache is filled. Once the relevant ways are completely filled additional accesses result in a miss but no replacement.

When both ways of a cache contain valid data, are locked and there is a read/write miss, that request is directly serviced from external memory.

Address-Range-Based Locking

A data buffer or a section of code can also be locked using range registers. A pair of range registers can be selected to define lockable data and code ranges. Once a range is set, any data or code of this range cannot be replaced. A locking eligible data or code cannot be cached if required lines/ways are already locked with valid data or code.

Cache Invalidation and Write Back Invalidation

The SHARC+ core wakes up in a cache disabled state after reset is removed. This action prevent false hits with uninitialized tag memory. After completing the necessary booting sequence and before enabling the cache, all cache entries must be invalidated.

A cache can be invalidated at any other time also. For example, once DMA updates the buffer in L3, the stale copy in L1-cache must be invalidated. There are times when invalidation is required to clear the cache unconditionally and other times when the cache must be cleared while ensuring that any updated copy is not cleared without writing back to L2 or L3. This is called write-back invalidation.

Cache invalidation and write-back invalidation occurs the following ways:

- Full-cache
- Address-range based

Full Cache Invalidation and Write-Back Invalidation

Write-back-invalidation (WBI/flush) ensures that all the modified data is written back to L3. This operation can be initiated by setting the appropriate bits in the `SHL1C_CFG` register.

A WBI requires the WB and Invalidation bits to be simultaneously set for the corresponding caches while invalidation requires only invalidation bit to be set for the corresponding caches.

NOTE: Invalidation and WBI should not be mixed. Any cache operation requires that that cache is enabled.

Both invalidation and write-back-invalidation takes multiple core clock cycles. The core pipeline is stalled during this time and interrupt servicing is delayed. Write-back takes more cycles than invalidation as this operation occurs line by line with possible L3 accesses. The exact number of cycles to complete a WB/WBI operation depends on number of dirty lines in a given data cache and available L3 throughput. Invalidation for both D-cache and I-cache takes about 32 cycles.

Address-Range Based Invalidation and Write-Back Invalidation

In some situations it is more appropriate and efficient to invalidate or write-back-invalidate only a buffer of data. For example when DMA updates a data buffer in L3, L1-cache copy becomes stale and should be invalidated. Similarly when an output buffer has been created on L1-cache, to perform DMA it must be written back to L3. In such cases address range based invalidation and write-back-invalidation is more efficient.

Pairs of range registers can be filled with the start and end address of the data/code segment to be invalidated or written back. Once these registers are filled and properties selected, the cache controller internally computes the *starting index* corresponding to start address and the *number of indexes to be invalidated* based on end address. These values are available in the `SHL1C_INV_IXSTART0` and `SHL1C_INV_CNT0` registers. These registers are running registers, which means that after clearing one index, the value of the index register increments and the value of the count register decrements. Do not access these registers when invalidation or flushing is in progress.

NOTE: The range register start address and end address must be cache line aligned (64 bytes). To make the cache line align, program the end address with the address of the last byte of the last cache line to be flushed plus one or the start address of the next line.

Example Range Based Write-Back Validation/Invalidation

The following sequence is the range of instructions that invalidate or flush the cache based on a given address range.

```
CAINV_START0 = <Start address of data/code segment>;
CAINV_END0 = <End address of data/code segment>;
12 un-compressed NOPs
LCNTR = dm(CAINV_COUNTER0);
Do … until LCE;
FLUSH (ICINV|DMINV|PMINV|DMWB|PMWB);
```

This sequence goes through each and every index of a specified cache/caches and whenever a matching entry is found, the line is invalidated and/or written back. Write-back operations occur only for dirty lines.

Further Details on Range Based WBI/Invalidation

- The value of the `SHL1C_INV_CNT0`/`SHL1C_INV_IXSTART0` registers is automatically determined by the cache controller based on the start and end address and the count.

  NOTE: To reload these registers, reload the `SHL1C_CFG2` and range registers. A write path to the `SHL1C_INV_IXSTART0` register has been provided to resume the process where it has been interrupted and left out.

- Mix of invalidation and write-back-invalidation options are not allowed.
- DM/PM caches should be WBI/invalidated together.
- Use full cache invalidation after the core reset removal.
- Program either the instruction cache or the data cache pair (DM/PM) at one time as a WBI/invalidated pair.
- Locking is not honored

Prefetch Buffer (ADSP-2156x and ADSP-SC59x Only)

A prefetch buffer (PFB) works with L1 cache memory to expedite the filling of cache lines. It prefetches the instructions and/or data for the next cache line in the background when there are no cache misses. This feature is enabled using the CMMR_SYSCTL.DPORT_PFB_EN and CMMR_SYSCTL.IPORT_PFB_EN bits for the data and instruction caches, respectively.

Features

The figure shows the block diagram of a prefetch buffer in the system.

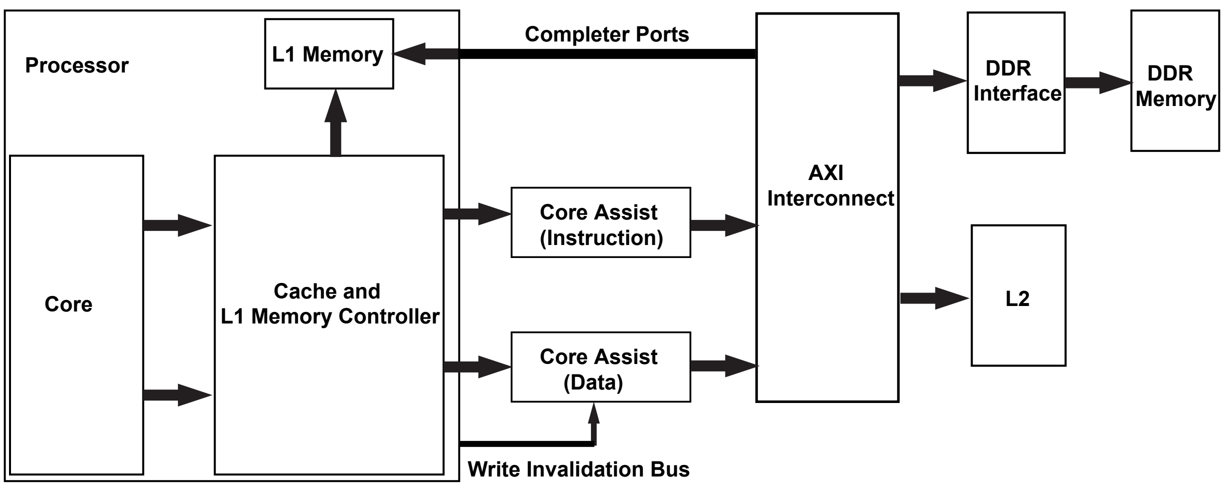

Figure 8-5: Prefetch Buffer

The following are the additional features of the prefetch buffer:

- Tiny cache: if the PFB is enabled with L1 cache off, it starts working as a tiny L1 cache (8 lines, single way) and reduces the external instruction and data latency.
- Range-based return zero: when this feature is enabled (CMMR_SYSCTL.UNINT_RET_0 = 1), the prefetch buffer returns all zeros if the requested line address falls in the range specified in the range start (CMMR_PFB_NOCHRT0_ST) and range end (CMMR_PFB_NOCHRT0_END) registers, thus reducing the cache line fill time for output and all zero input buffers.
- Invalidation:

  - The PFB is manually invalidated by setting the invalidate prefetch buffer bit (CMMR_SYSCTL.PFB_INVAL = 1) and then clearing it (CMMR_SYSCTL.PFB_INVAL = 0), when the data and instruction port pre-fetch buffers are enabled (CMMR_SYSCTL.DPORT_PFB_EN = CMMR_SYSCTL.IPORT_PFB_EN = 1).
  - The PFB is invalidated upon core reset.
  - A line in the PFB is invalidated whenever the same line is invalidated by L1 cache.

    NOTE: Explicitly invalidate the prefetch buffer, after range based invalidation, write-back, or both, by setting the invalidate prefetch buffer bit CMMR_SYSCTL.PFB_INVAL = 1 and then clearing it CMMR_SYSCTL.PFB_INVAL = 0.

- Range-based prefetching:
  - Prefetching can be disabled for up to sixteen address ranges by clearing the appropriate bit in the Prefetch Range Selection Register (REG_MISCREG_PFB_RANGE_SELECT). The IPORT_RANGE_SELECT bits control the range selection for the instruction cache. The DPORT_RANGE_SELECT bits control the range selection for the data cache.
  - The ***Prefetch Address Range*** table provides the address range associated with each bit in the REG_MISCREG_PFB_RANGE_SELECT register. By default, all ranges are enabled. Any specific range can be disabled by clearing the associated input bit.

Table 8-4: Prefetch Address Range

Input Bit	Prefetch Address Range
REG_MISCREG_PFB_RANGE_SELECT[0]	0x00000000 – 0x0FFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[1]	0x10000000 – 0x1FFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[2]	0x20000000 – 0x2FFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[3]	0x30000000 – 0x3FFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[4]	0x40000000 – 0x4FFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[5]	0x50000000 – 0x5FFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[6]	0x60000000 – 0x6FFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[7]	0x70000000 – 0x7FFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[8]	0x80000000 – 0x8FFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[9]	0x90000000 – 0x9FFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[10]	0xA0000000 – 0xAFFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[11]	0xB0000000 – 0xBFFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[12]	0xC0000000 – 0xCFFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[13]	0xD0000000 - 0xDFFFFFFF

Table 8-4: Prefetch Address Range (Continued)

Input Bit	Prefetch Address Range
REG_MISCREG_PFB_RANGE_SELECT[14]	0xE0000000 – 0xEFFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[15]	0xF0000000 – 0xFFFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[16]	0x00000000 – 0x0FFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[17]	0x10000000 – 0x1FFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[18]	0x20000000 – 0x2FFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[19]	0x30000000 – 0x3FFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[20]	0x40000000 – 0x4FFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[21]	0x50000000 – 0x5FFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[22]	0x60000000 – 0x6FFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[23]	0x70000000 – 0x7FFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[24]	0x80000000 – 0x8FFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[25]	0x90000000 – 0x9FFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[26]	0xA0000000 – 0xAFFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[27]	0xB0000000 – 0xBFFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[28]	0xC0000000 – 0xCFFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[29]	0xD0000000 - 0xDFFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[30]	0xE0000000 – 0xEFFFFFFF
REG_MISCREG_PFB_RANGE_SELECT[31]	0xF0000000 – 0xFFFFFFFF

Prefetch Range Selection Register

Prefetching can be disabled for up to sixteen address ranges by clearing the appropriate bit in this register.

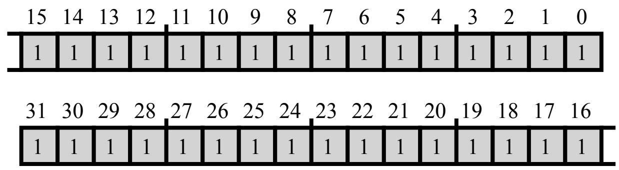

Figure 8-6: MISCREG_PFB_RANGE_SELECT Register Diagram

Table 8-5: REG_MISCREG_PFB_RANGE_SELECT

Bit No. (Access)	Bit Name	Description/Enumeration
31:16 (R/W)	IPORT_RANGE_SELECT	Prefetch Range Selection for IPORT. It controls the range selection for the instruction cache.
15:0 (R/W)	DPORT_RANGE_SE-LECT	Prefetch Range Selection for DPORT. It controls the range selection for the data cache.

9 Safety, Security, Multi-Core, and Low-Power Features

The SHARC+ core provides features related to application safety and security, and the processors that feature multiple cores have provisions for multi-core management. This chapter discusses these features, as well as some power-saving features that are exclusive to the ADSP-2156x and ADSP-SC59x processors:

- Parity Error Detection for L1 Accesses
- Illegal Opcode Error Detection for Instruction Fetch
- Security Operations
- Memory Barrier (SYNC) Instruction
- Semaphores (ADSP-SC57x, ADSP-SC58x, and ADSP-SC59x Only)
- Resetting in Multicore Systems (ADSP-SC57x, ADSP-SC58x, and ADSP-SC59x Only)
- Arm L2 Cache Sharing Address Range Registers (ADSP-SC57x, ADSP-SC58x, and ADSP-SC59x Only)
- Low-Power Features (ADSP-2156x and ADSP-SC59x Only)

Parity Error Detection for L1 Accesses

Detection of single-bit soft errors is important for overall system security. The SHARC+ core uses a hardware-based single-bit soft error detection scheme. In the event of a single-bit error in L1, tag, LRU and dirty cache memories, a read generates an error interrupt that is routed to both the REGF_IRPTL register and the system parity error controller. This error condition can be handled using a local interrupt service rotuine (ISR) or via system-level control.

Parity Operations Programming Model

The REGF_MODE1.SPERREN, REGF_MODE1.DPERREN, and REGF_MODE1.IPERREN bits enable parity checking for instruction fetch, data read (through DM/PM bus) and DMA read, respectively. For all applications, these parity enable bits must all be configured in a single write to the REGF_MODE1 register. When any of the REGF_MODE1.SPERREN, REGF_MODE1.DPERREN, or REGF_MODE1.IPERREN bits are enabled, the parity bit updates whenever the corresponding RAM is written, and parity checking occurs whenever any RAM is read. If an error is detected, the parity debug register is updated, and the error is routed to the REGF_IRPTL register (if it is enabled) and to the system-level parity error controller (if it is enabled).

When the parity interrupt in the REGF_IRPTL register is enabled, it will be latched upon detection.

CAUTION: Parity error interrupts can occur during core cycles that are typically not interruptible. As a result, the RTI register may be unreliable as the continuation point after the parity ISR executes.

Error Generation:

Parity errors are indicated in the CMMR_GPERR_STAT register. Once a bit is set, it is locked until an explicit write of 0x0 clears the CMMR_GPERR_STAT register. Register writes should be done with care, as writing an artificial error also causes an interrupt. The CMMR_GPERR_STAT register is cleared by a core reset.

Error Handling

There are two ways to handle parity errors:

- Through the system-level parity error controller.
- Through a local ISR serving interrupts latched in the REGF_IRPTL register.

Either of the two methods can be used to clear the error condition.

Parity Error Registers

Parity status registers are core MMR registers.

Once an error condition is registered in the debug registers, it gets locked and remains locked until all the bits are written with zeros (clearing the error status manually) or the core is reset.

For more information on the Parity Error register, see the SHARC-PLUS CMMR Register Descriptions chapter.

Illegal Opcode Error Detection for Instruction Fetch

Illegal opcode detection is similar to instruction parity error detection. The primary difference is that parity error detection works for L1 instruction and data accesses, while illegal opcode detection works for all kinds of instruction fetches. When an opcode which is not defined in the supported instruction set is encountered, an illegal opcode interrupt is generated.

NOTE: For double-precision floating-point compute operations, unused register bits are not checked by this logic.

The Illegal Opcode Error Status register (SHDBG_DBGREG_ILLOP) captures the status of the opcode error. On core reset, all the bits in the register are set to zero. Once an error is detected, the content of the register remains locked until it is manually cleared by writing zero to the register or the core is reset.

For more information on the Opcode Error Status Register, see the SHARC-PLUS CMMR Register Descriptions chapter.

Security Operations

The SHARC+ core does not generate nor change its security status. The following scheme has been used to implement a security gate on the SHARC+ processor core:

- The system protection unit (SPU) and the System Memory Protection Unit (SMPU) maintain the security status of each SHARC+ core in the SoC.
- This security status signal is fed to all the ports of the core: requester instruction/data port and both completer ports.
- If the SHARC+ core is secure, all the accesses originating from the requester instruction/data ports carry that status so they can access secure system resources.
- All accesses coming to the completer ports are checked for security status. Decode error responses are sent when there is a security mismatch.

NOTE: Enabling and disabling of debug and trace feature access from the debug port is controlled by the system-level DBGEN input to the SHARC+ core. If DBGEN is asserted, all the debug and trace configuration registers can be accessed. If DBGEN is deasserted, only a limited set of debug and trace configuration registers can be accessed. This limited set provides information on debug and trace features supported by the SHARC+ core, but none of the debug and trace features can be enabled and used.

Memory Barrier (SYNC) Instruction

On shared-memory multi-requester systems, data written by one requester must be visible to other requesters before proceeding with the rest of the communication task. For example, core A may write to an external shared space and signal core B that it can now read the data from the shared space. However, core A needs to have a mechanism to ensure that the write has actually been completed before it sends the signal to core B.

Even though the SHARC+ core does not reorder any transactions, it has write buffers on both the system SCB interface and the L1 IMIF. The system interface operates on the bus protocol, where any write is deemed to have been completed only when the write response is returned. This could take many cycles, depending on the number of register slices in the system fabric. Waiting for the write response for each write transaction results in numerous multi-cycle stalls. Instead, these writes are posted on the bus channel, and the pipeline is allowed to move without waiting for the response. The SHARC+ memory barrier (SYNC) instruction flushes these write buffers and awaits the write responses.

NOTE: Executing the SYNC instruction does not flush dirty data cache lines and does not invalidate the instruction cache. Write-back-invalidation can be used to flush dirty data cache lines, and invalidation can be used to invalidate the instruction cache (if required) before using the SYNC instruction.

Upon executing the SYNC instruction, the core is stalled until all pending writes on the System Interface and all pending writes in the write FIFO on the Internal Memory (L1) interface have completed.

SYNC instructions are required before starting any read or write DMA targeting L1 memory. This also applies to sharing data with other processors. It is furthermore advised to use the SYNC instruction between writing and subsequently reading any peripheral or core MMR.

Example Pipeline Behavior for Memory Barrier (SYNC) Instruction

In the ***Pipeline View of Parity Error Detection During Core Data Read*** table, note the STALL cycles and the CMP (complete) cycle. The STALL label indicates the pipeline stalls that occur while waiting for the system interace or for the L1 interface to complete pending writes. The CMP label indicates the cycle in which the processor completes pending writes and releases the stall.

Table 9-1: Pipeline View of Parity Error Detection During Core Data Read

							STALL						CMP	
e2							SYNC	SYNC	SYNC	SYNC	SYNC	SYNC		
m4						SYNC	SYNC	SYNC	SYNC	SYNC	SYNC	SYNC		
m3					SYNC	SYNC	SYNC	SYNC	SYNC	SYNC	SYNC	SYNC		
m2				SYNC	SYNC	SYNC	SYNC	SYNC	SYNC	SYNC	SYNC	SYNC		N+1
m1			SYNC	SYNC	SYNC	SYNC	SYNC	SYNC	SYNC	SYNC	SYNC	SYNC	N+1	
d2		SYNC	N+1	N+1	N+1	N+1	N+1	N+1	N+1	N+1	N+1	N+1		
d1	SYNC	N+1	N+2	N+2	N+2	N+2	N+2	N+2	N+2	N+2	N+2			

SYNC Instruction and Interrupts

Once a memory barrier (SYNC) instruction reaches the execution (E2) stage of the pipeline, it can no longer be interrupted. This can make the SYNC instruction and the previous or next instruction uninterruptible. This can also potentially make a larger number of cycles uninterruptible as the SYNC instruction awaits all pending system and L1 interface writes to complete.

Flushing the Pipeline

To ensure instructions are refetched from memory after a SYNC instruction, software must arrange for the processor pipeline to be flushed, which is done by placing a JUMP instruction after the SYNC instruction. This operation is most conveniently achieved by putting the SYNC instruction in a subroutine. There is, however, no need to disable the BTB.

Semaphores (ADSP-SC57x, ADSP-SC58x, and ADSP-SC59x Only)

Semaphores are essential for shared memory multi-core systems where multiple cores are competing for the same shared resource and the access needs to be atomic. Semaphores are supported in SHARC+ using the exclusive access feature of the system fabric. Load and store exclusive instructions can be used to implement software semaphores to control interaction among multiple cores.

Exclusive reads and writes are supported with the following addressing modes.

- Indirect addressing (register modify) Type 3d (see Instruction Summary)
- Direct addressing Type 14d (see Instruction Summary)

NOTE: Exclusive load/store operation is not supported in multifunction compute instructions.

As can be seen from the table in Instruction Summary, all possible access sizes (byte, short-word, word and long word) are supported with exclusive accesses. Success of the exclusive store/load access sets the `REGF_ASTATX.AZ` bit for SISD and the `REGF_ASTATY.AZ` bit for SIMD. Programs must ensure that exclusive stores are only attempted to locations from which there has been a successful exclusive load, indicated by a zero AZ flag

ATTENTION: Application code should abort when this operation fails.

Example Usage

Typical exclusive access instruction usage (spin-lock) is shown below:

```
R1 = 0x1;
 SPIN:
 R0 = DM(M0,I0) (EX);
 IF NE JUMP _abort;      // Unrecoverable error
 R0 = PASS R0;             // is semaphore unlocked?
 IF NE JUMP SPIN;          // no - try again
 DM(M0,I0) = R1 (EX);      // try to lock
 IF NE JUMP SPIN ;         // failed - try again

 // CRITICAL SECTION
 R1 = 0;                   // unlocked value
 DM(M0,I0) = R1;           // unlock
```

Exclusive Access Usage Restrictions

The following are restrictions applying to exclusive load and store accesses:

- Exclusive accesses are not supported to local L1 and multi-memory space.
- When performing an exclusive access to L2/L3 space, the region being accessed should be marked as non-cacheable, otherwise the access is not seen by the memory peripheral as an exclusive access.
- Refer to the product-specific hardware reference manual for memory regions supporting exclusive accesses. When an exclusive access is attempted to a region that does not support exclusive accesses, the failed access is indicated by a set AZ flag.
- Refer to the ***Sizes and Alignment Restrictions in SISD and SIMD Modes*** in Byte Address Space Overview of Data Accesses for exclusive access alignment restrictions.

Resetting in Multicore Systems (ADSP-SC57x, ADSP-SC58x, and ADSP-SC59x Only)

The SHARC+ core has support for a RCU disable request and acknowledge mechanism. On receiving a disable request from the Reset Control Unit (RCU), the SHARC+ stops all further accesses on the requester port and stalls the completer port on a clean access boundary. It then sends an acknowledge back to the RCU. This activity can be polled by another core to decide when it is safe to reset the first core. This ensures that any ongoing DMA to the L1 (when the first core was reset) can seamlessly resume when it is brought out of reset.

Refer to the RCU chapter in the product specific hardware reference manual for details on this mechanism.

Arm L2 Cache Sharing Address Range Registers (ADSP-SC57x, ADSP-SC58x, and ADSP-SC59x Only)

To achieve data coherency between the Arm and SHARC+ core at the L2 cache level, the processor provides a connection between the SHARC and the L2 cache of the Arm.

As shown in the ***Data Cache Hit / Miss Generation Process for a Single Bank*** figure, the SHARC core can access system memory directly or via Arm L2 cache. When the L2 cache address register pair is cleared, the SHARC+ data port read data directly from system memories (L2/L3) through the system fabric. However when the L2 cache address register pair have L2/L3 addresses configured the SHARC+ data port reads data via the A5 L2 cache from the system memories.

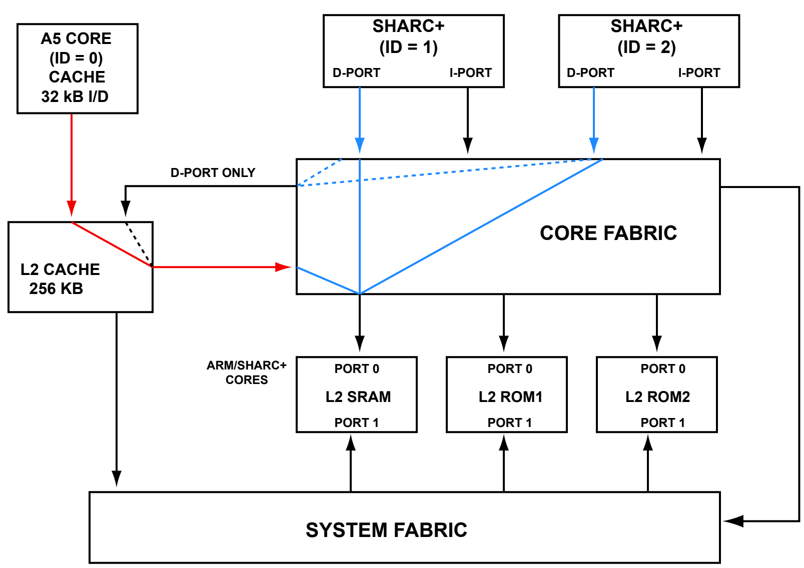

Figure 9-1: Arm L2 Cache Sharing

Secure versus Non-secure Access through A5 L2 Cache

The Arm L2 cache treats secure and non-secure data as being part of two different memory spaces. System designs should ensure that all completers that share data are using the same level of security.

When using the L2 cache to perform an access through the L2 cache, the SHARC+ core always performs the access in read/write allocate and write-back mode.

If a non-secure SHARC core attempts a write access on a secure completer, the core receives an OK response. If a write miss occurs, the core first performs a read on the completer, then performs a line fill and finally attempts to perform a write. Note that a read error is not returned to the SHARC core for the write attempt. If data is currently in the cache the L2 cache treats secure and non-secure accesses as different and doesn't generate an error (an OK response is returned). A L2 cache read response interrupt must be used to see the write error.

NOTE:

- When the SHARC performs an access through the L2 cache it is always configured in write-back and read/write allocate mode. Configure the Arm A5 accordingly.
- Core memory-mapped registers (CMMRs) are not accessible by rest of the system.
- Performing an access through the L2 cache does not guarantee data coherency between the Arm A5 and SHARC+ cores. However it eases the coherency implementation on software.
- If the Arm A5 is in reset, this feature should not be used.
- Programs should perform a L2 cache write-back invalidation before changing the value of the L2 cache address register pair.
- All completers that are accessed through L2 cache should be either read/write both secure or both non-secure. In case of partial security coherency is not guaranteed and behavior is unexpected.
- Insert a `SYNC` instruction before and after modifying a core memory-mapped register. Otherwise any transactions pending in the FIFOs in the requester bridge may see the effect of the writes.
- Programs must not perform a write if a data access occurs on ROM through the L2 cache because a write may cause corruption in the cache.

For more information, see the *ADSP-SC58x SHARC Processor Hardware Reference* "L2 Memory Controller (L2CTL)" chapter.

Low-Power Features (ADSP-2156x and ADSP-SC59x Only)

The ADSP-2156x and ADSP-SC59x SHARC+ processors provide additional power-saving features to reduce power consumption in on-chip L1 memory and when the core is idle, as discussed in the following sections.

Low-Power Memory Features

Many applications may not require the entire 5 Mb of L1 memory. If certain banks are unused or predictably not being accessed, they can be put into either of two low-power modes, Memory Sleep or Memory Shutdown. Depending on the application requirements, a suitable mix of both modes can be configured on a bank-wise basis, as discussed in the following sections.

Memory Sleep Mode

Memory Sleep mode reduces idle memory power consumption while retaining the memory content. Banks 0 through 3 can individually be put into and taken out of Memory Sleep mode by manipulating the corresponding BANK0 through BANK3 bits of the CMMR_PWR_L1_LS_CTL register. Setting (=1) any of these bits places the associated bank into Memory Sleep mode, while clearing (=0) any of these bits takes the associated bank out of Memory Sleep mode. As the reset states of these bits are 0, Memory Sleep mode is disabled by default for all banks.

There is an 11-cycle latency associated with removing a bank from Memory Sleep mode. If any read/write access to a bank being removed from Memory Sleep mode takes place prior to this delay, it may result in an unexpected outcome (reads are always 0 and writes do not take place).

NOTE: Initial 0.25 MB of Bank 0 is always fully enabled and is unaffected by changes to the Bank 0 power mode.

CAUTION: Do not enable both Memory Shutdown and Memory Sleep modes on the same bank of memory.

Memory Shutdown Mode

If memory content does not need to be maintained for the banks that are being placed into a low-power state, the Memory Shutdown mode can be used instead of Memory Sleep to reduce power even further. This mode is controlled exactly as with the Memory Sleep mode, except the bank-wise control over enabling and disabling Memory Shutdown mode is via the CMMR_PWR_L1_SD_CTL register.

As Memory Shutdown feature can cause entire banks of memory to lose content, access to this feature must be enabled separately by setting the L1 shutdown enable bit (CMMR_SYSCTL.L1_SD_EN = 1).

There is an 11-cycle latency associated with removing a bank from Memory Shutdown mode. If any read/write access to a bank being removed from Memory Shutdown mode takes place prior to this delay, it may fail.

NOTE: Initial 0.25 MB of Bank 0 is always fully enabled and is unaffected by changes to the Bank 0 power mode.

CAUTION: Do not enable both Memory Shutdown and Memory Sleep modes on the same bank of memory.

Low-Power Idle Mode (Core Light Sleep)

When the core goes into the idle state upon executing the IDLE instruction, there is still active/switching power associated with all active clocks despite the fact that there is no activity inside the core. To reduce this power, a core clock gating mechanism (Core Light Sleep) can be used to switch off these clocks when the core becomes idle by programming the Core Light Sleep Enable field (CMMR_PWR_GLB_CTL.CORE_SLP = 1).

NOTE: To ensure backwards compatibility with previous ADSP-SC5xx processors, the disable CCLK gating bit can be used to disable this feature (`CMMR_SYSCTL.DIS_CCLKG` = 1).

The following are associated with use of the Core Light Sleep feature:

- The IMIF, completer ports, and requester ports are awake, but the remaining clocks become gated when the IDLE instruction reaches the E2 stage of the pipeline. Therefore, DMA reads and writes are allowed to all four banks.
- The core timer freezes.
- An interrupt-driven wakeup event (SEC interrupt arriving to the core) disables gating of the clocks.
- DBGEN can also be used to wake from the Core Light Sleep mode to serve debug requests.

10 SHARC+ Core Debug Interface

The Analog Devices Tools JTAG emulator is a development tool for debugging programs running in real time on target system hardware.

Because the JTAG emulator controls the target system's processor through the processor's debug interface, non-intrusive in-circuit emulation is assured.

Features

The debug interface has the following features.

- Standard emulation-start stop and single step
- Enhanced standard emulation with instruction and data breakpoints, event count, valid and invalid address range detection
- Statistical profiling for benchmarking
- Support for setting user breakpoints

Functional Description

The following sections provide descriptions about debug functionality.

Debug Interface

The core provides a peripheral bus completer interface to access the debug functionality. Debug registers are memory-mapped and accessible over the peripheral bus.

Breakpoints

This section explains the different types of breakpoint and conditions to hit breakpoints.

Software Breakpoints

Software breakpoints are implemented by the processor as a special type of instruction. The instruction, `EMUIDLE` is not a public instruction, and is only decoded by the processor when specific bits are set in emulation control. If the processor encounters the EMUIDLE instruction and the specific bits are not set in emulation control, then the processor executes a NOP instruction. The EMUIDLE instruction triggers a high emulator interrupt. When EMUIDLE is executed, the emulation clock counter halts immediately.

General Restrictions on Software Breakpoints

Based on the 11-stage instruction pipeline, programs can not set software breakpoints at the following locations.

- If a breakpoint interrupt comes at a point when a program is coming out of an interrupt service routine of a prior breakpoint, then in some cases the breakpoint status does not reflect that the second breakpoint interrupt has occurred.
- If an instruction address breakpoint is placed just after a short loop, a spurious breakpoint is generated.
- Delay slots of delayed branch instructions.
- Counter based loops of length one two and three
- Last three instructions of any arithmetic loop

Automatic Breakpoints

The IDDE (tools environment) places software breakpoints automatically at the labels _main and __lib_prog_term. For example, if the program places the (_main) label at the beginning of user code, it simplifies halting the start of code execution after reset (for example, in a DDR2/SDRAM initialization or a runtime environment).

For more information, refer to the tools documentation.

Hardware Breakpoints

Hardware breakpoints allow much greater flexibility than the software breakpoints provided by the EMUIDLE instruction. At the simplest level, hardware breakpoints are helpful when debugging ROM code where the emulation software can not replace instructions with the EMUIDLE instruction. At a minimum, an effective hardware breakpoint unit has the capability to trigger a break on a load, store, and fetch activity.

Additionally, address ranges, both inclusive (bounded) and exclusive (unbounded) can be specified.

Operating Modes

The following sections detail the operation of the debug interface.

Emulation Space Mode

The processor emulation features halt the processor at a predefined point to examine the state of the processor, execute arbitrary code, restore the original state, and continue execution. If the processor hits a valid breakpoint it triggers an emulator interrupt which puts the processor into *emulation space* (core halt). In this state, the processor waits until the emulator continues to scan new instructions into the processor over the debug interface. If the emulator scans an RTI instruction into the processor, it is released back into *user space* (core run).

The emulator uses the debug interface to access the internal space of the processor, allowing the developer to:

- Load code
- Set SW/HW breakpoints
- Set user breakpoints
- Observe variables
- Observe memory
- Examine registers
- Perform cycle counting

The processor must be halted to send data and commands, but once an operation is completed by the emulator, the system is set running at full speed with no impact on system timing. The emulator does not impact target loading or timing. The emulator's in-circuit probe connects to a variety of host computers (USB or PCI) with plug-in boards.

Emulation Control

The processor is free running. In order to observe the state of the core, the emulator must first halt instruction execution and enter emulation mode. In this mode, the emulation software sets up a halt condition by setting the HALT bit in the Run-Control-Status (RCS) register.

Instruction and Data Breakpoints

The SHARC processors contain sets of emulation breakpoint registers. Each set consists of a start and an end register which describe an address range, with the start register setting the lower end of the address range. Each breakpoint set monitors a particular address bus. When a valid address is in the address range, then a breakpoint signal is generated. The address range includes start and end addresses.

Instruction breakpoints monitor the program memory address bus while data breakpoints monitor the data or program memory address bus.

Address Breakpoint Registers

The address breakpoint registers are described in the SHARC-PLUS SHDBG Register Descriptions chapter. These registers are used by the emulator and the user breakpoint control to specify address ranges to verify if specific conditions become true. The reset values are not defined.

Conditional Breakpoints

The breakpoint sets are grouped into four types:

- 4x instruction breakpoints (IA)
- 2x data breakpoints for DM bus (DA)
- 1x data breakpoints for PM bus (PA)

The individual breakpoint signals in each group are logically OR'ed together to create a composite breakpoint signal per group.

Each breakpoint group has an enable bit in the SHDBG_BRKCTL register. When set, these bits add the specified breakpoint group into the generation of the effective breakpoint signal. If cleared, the specified breakpoint group is not used in the generation of the effective breakpoint signal. This allows the user to trigger the effective breakpoint from a subset of the breakpoint groups.

These composite signals can be optionally AND'ed or OR'ed together to create the effective breakpoint event signal used to generate an emulator interrupt. The SHDBG_BRKCTL.ANDBKP bit register selects the function used.

> **NOTE:** The SHDBG_BRKCTL.ANDBKP bit has no impact within the same group of breakpoints (DA group, IA group). It has significance when the program uses different groups of breakpoints (IA, DM, PM) and the resultant breakpoint is logically AND'ed of all those breakpoints which are enabled.

To provide further flexibility, each individual breakpoint can be programmed to trigger if the address is in range AND one of these conditions is met: READ access, WRITE access, or ANY access. The control bits for this feature are also located in DBG_BRKCTL register.

> **NOTE:** Note the following restrictions on breakpoints.
>
> 1. At least two breakpoints must be enabled prior to enabling the SHDBG_BRKCTL.ANDBKP bit.
> 2. Enabling of the SHDBG_BRKCTL.ANDBKP bit should not be done in the same instruction.

For index range violations in user code, the address ranges of the emulation breakpoint registers are negated (twos complement) by setting the appropriate SHDBG_BRKCTL register.

Each breakpoint can be disabled by setting the start address larger than the end address.

> **NOTE:** The instruction address breakpoints monitor the address of the instruction being executed, not the address of the instruction being fetched.

If the current execution is aborted, the breakpoint signal does not occur even if the address is in range. Data address breakpoints (DA and PA only) are also ignored during aborted instructions.

The breakpoint sets can be found in Programming Model User Breakpoints.

Event Count Register

The SHDBG_EMUN register is a 32-bit memory-mapped I/O register and can be accessed in user space. The core can write to it in user space. This register is used to detect the Nth breakpoint. This SHDBG_EMUN register allows the breakpoint to occur at Nth count. If the register is loaded with N, the processor is interrupted only after the detection of N breakpoint conditions. At every breakpoint occurrence the processor decrements the SHDBG_EMUN register and it generates an interrupt when the contents of the SHDBG_EMUN register is zero and a breakpoint event occurs.

Note that programs must load this register with a value greater or equal to zero for proper breakpoint generation under the condition that bit 25 (SHDBG_BRKCTL.UMODE bit) is set.

Emulation Cycle Counting

The emulation clock counter consists of a 32-bit count register, REGF_EMUCLK and a 32 bit scaling register, REGF_EMUCLK2. The REGF_EMUCLK register counts clock cycles while the user has control of the chip and stops counting when the emulator gains control. This allows a user to gauge the amount of time spent executing a particular section of code. The REGF_EMUCLK2 register is used to extend the time REGF_EMUCLK can count by incrementing itself each time the EMUCLK value rolls over to Zero. Both REGF_EMUCLK and REGF_EMUCLK2 are emulation registers,which can only be written in emulation space. Reads of REGF_EMUCLK and REGF_EMUCLK2 can be performed in user space. This allows simple benchmarking of code.

Statistical Profiling

Statistical profiling allows the emulation software to sample the processors PC value while the processor is running. By sampling at random intervals, a profile can be created which can aid the developer in tuning performance critical code sections. As a second use, statistical profiling can also aid in finding dead code as well as being used to make code partition decisions. Fundamentally, statistical profiling is supported by the debug register called EMUPC. The EMPUC register is a 24-bit register which samples the program counter whenever any transaction (read or write) happens on the debug interface. This register is used for statistical profiling.

User Space Mode

The following sections describe user space mode operation.

User Breakpoint Control

By default, the emulator has control over the breakpoint unit. However, if there is a need for faster system debug without the delay incurred when the core halts and enters emulations space, then the core can gain control by setting the SHDBG_BRKCTL.UMODE bit.

Conversely, if the SHDBG_BRKCTL.UMODE (bit 25) is cleared, only the emulator has breakpoint control over the TAP.

NOTE: If the SHDBG_BRKCTL.UMODE bit is set, all address breakpoint registers can be written in user space.

For more information, see SHARC-PLUS SHDBG Register Descriptions chapter.

User Breakpoint Status

The DBG_BRKSTAT register acts as the breakpoint status register for the SHARC+ processors. This register is a memory-mapped IOP register. The processor core can access this register if the DBG_BRKCTL.UMODE bit (bit 25) is set.

The DBG_BRKSTAT register indicates which breakpoint hit occurred. All the breakpoint status bits are cleared when the program exits the ISR with an RTI instruction. Such interrupts may contain error handling if the processor accesses any of the addresses in the address range defined in the breakpoint registers.

NOTE: Status update of the DBG_BRKSTAT register does not work in single step mode for user break points.

For more information, see SHARC-PLUS SHDBG Register Descriptions chapter.

User Breakpoint System Exception Handling

Through the proper configuration of the SHDBG_BRKCTL and SHDBG_BRKSTAT registers, and by using different logical combined address breakpoint regions in conjunction with event count registers for core or DMA operations, programs can take advantage of system specific exception handling based on specified conditions which trigger the low priority emulator interrupt (BKPI).

User to Emulation Space Breakpoint Comparison

The primary difference between user and emulation space breakpoints are that user breakpoints are user instruction driven while emulation space breakpoints happen via the debug interface.

Programming Model User Breakpoints

To set up the user controlled breakpoint functionality use the following steps.

1. Unmask the BKPI interrupt (low priority interrupt).
2. Set the SHDBG_BRKCTL.UMODE bit.
3. Set the breakpoint count in the SHDBG_EMUN register to the required value.
4. Initialize the breakpoint address registers with required address ranges.
5. Enable the breakpoint conditions as required in the SHDBG_BRKCTL register.
6. Enable the logical AND'ing of breakpoints if required in the SHDBG_BRKCTL register.

Programming Examples

The *Trigger an Exception for a Valid Address* example shows how to trigger an exception for a valid address.

Trigger an Exception for a Valid Address

```
bit set IMASK BKPI;     /* unmask BKPI */
bit set MODE1 IRPTEN;   /* enable global int */
r5 = ADDR_S;            /* valid start addr for the break */
r6 = ADDR_E;            /* valid end addr for the break */
r3 = UMODE | DA1MODE;   /* set the user mode and dm access functionality for r/w
access */
dm(BRKCTL) = r3;
dm(DMA1S) = r5;         /* start addr for break */
dm(DMA1E) = r6;         /* end addr for break */
r5 = 0x15;
dm(EMUN) = r5;          /* set event count */

USTAT1 = dm(BRKCTL);
BIT SET USTAT1 ENBDA;   /* enable the dm access break points */
dm(BRKCTL) = USTAT1;

ISR_BKPI:
```

```
r4 = dm(BRKSTAT);      /* read status bits */
rti;                   /* status register cleared */
```

The *Trigger an Exception for an Invalid Address Range* example shows how to trigger an exception for an invalid address range.

Trigger an Exception for an Invalid Address Range

```
bit set IMASK BKPI;                   /* unmask BKPI */
bit set MODE1 IRPTEN;                 /* enable global int */
r4 = ADDR_S;                          /* valid start address for the break */
r5 = ADDR_E;                          /* valid end address for the break */

USTAT1 = UMODE | DA2MODE | NEGDA2; /* set the user mode and negate dm access
                                      functionality for r/w access */
dm(BRKCTL) = USTAT1;

dm(DMA2S) = r4;
dm(DMA2E) = r5;
r5 = 0x0;                             /* no event count */

dm(EMUN) = r5;

USTAT1 = dm(BRKCTL);
BIT SET USTAT1 ENBDA;                 /* enable the dm access break points */
dm(BRKCTL) = USTAT1;
ISR_BKPI:
r4 = dm(BRKSTAT);                     /* read status bits */
rti;                                  /* status register cleared */
```

Single Step Mode

When the single step bit in the emulation control register is set, single step mode is enabled. In single step mode, the processor executes a single instruction, and then automatically generates an internal emulator interrupt to return to emulation space. While in emulation space the emulator can execute a RTI instruction to do a single step again. Each user instruction execution in single step mode clears the instruction pipeline when the part reenters user space.

Instruction Pipeline Fetch Inputs

The instruction pipeline is fed by four inputs:

1. Instruction fetch from memory, this is the user mode (also known as user space) and described in the sequencer chapter
2. Instruction fetch from boot channel, during boot operation (256 instruction words) the pipeline is fed with the `IDLE` instruction until the peripheral's interrupt is generated

3. Instruction fetch from an emulator register, by using tools (debugger) in single step mode (also known as emulation space) the instruction pipeline is deactivated. In this mode, each instruction is fetched from an emulation register over the JTAG interface (rather from memory) and executed in isolation. The process is repetitive for all the next instructions in single step mode.
4. Instruction fetched from instruction-conflict cache during an cache hit. If a hit occurs, the instruction is loaded from instruction-conflict cache and not from memory.

Differences Between Emulation and User Space Modes

The primary difference between user space and emulation space operation is that in emulation space, the processor holds while the instruction is scanned in, while in user space, the instruction is taken from an emulation instruction register, rather than from the PMD bus. In emulation space, the program counter also stops incrementing. All other aspects of instruction execution are the same in both modes.

Debug Interrupts

The ***Debug Interrupt Overview*** table provides an overview of the interrupts associated with the debug interface. For a complete list of interrupts, see the Interrupt Priority and Vector Table.

Table 10-1: Debug Interrupt Overview

Interrupt Source	Interrupt Condition	Return Register	Return Instruction	IVT Level
JTAG	Emulation Space	N/A	N/A	0, EMUHI
Instruction/Data Address	HW Breakpoint Hit	BRKSTAT	RTI	12, BKPI

Interrupt Types

The following different types of interrupts/breakpoints are generated.

- External emulator generates EMUI interrupt via Halt bit (highest priority)
- Breakpoint generates an internal EMUI interrupt (highest priority)
- User space breakpoint generates an internal BKPI interrupt (lower priority)

Entering Into Emulation Space

When the core receives emulator interrupt, the following sequence occurs:

1. The PC stack is pushed and the PC vectors to reset location
2. The core is idle, waiting for an emulator instruction
3. The core timer and emulation counter stop counting
4. The instruction-conflict cache is disabled
5. DMA operation may be optionally stalled

6. The core notifies emulation space via the HALTED bit in RCS register.

Debug Register Effect Latency

Instruction address and program memory breakpoint negates have an effect latency of four core clock cycles.

References

- IEEE Standard 1149.1-1990. Standard Test Access Port and Boundary-Scan Architecture. To order a copy, contact the IEEE society.
- Maunder, C.M. and R. Tulloss. Test Access Ports and Boundary Scan Architectures. IEEE Computer Society Press, 1991.

11 Performance Monitor (PFM)

The SHARC+ architecture provides a built-in performance monitor (PFM) to non-intrusively monitor core resources by counting the number of L1-cache hit or miss events. Monitoring these event counts is useful for debug purposes and to enable improved execution efficiency, for example, by suggesting where to modify code to increase L1C hits.

NOTE: Note: When a sequence of instructions aborts and then appears in the pipeline again due to sequencing requirements, the counters record more than once for the same instruction.

Functional Description

The PFM has a control register (PFM_CFG) that enables performance counting in the I-cache, PM-cache or DM cache. Counting may only be enabled in one cache memory mode at a time. There are 4 sets of 16-bit counter registers (PFM_CNTR3-PFM_CNTR6). The performance metric for each counter depends on the cache mode enabled in the PFM_CFG register. The *DM-Cache Mode*, *PM-Cache Mode*, and *I-Cache Mode* tables show the register descriptions for each cache mode.

Table 11-1: DM-Cache Mode

Count Register	Function	Description
PFM_CNTR3	DM-Cache Hit	Records the total number of occurrences of DM-cache hits
PFM_CNTR4	Crosscheck Hit	Records the total number of occurrences of crosscheck hits
PFM_CNTR5	DM-Cache Miss without Writeback	Records the total number of occurrences of DM-cache misses without writeback
PFM_CNTR6	DM-Cache Miss with Writeback	Records the total number of occurrences of DM-cache misses with writeback

Table 11-2: PM-Cache Mode

Count Register	Function	Description
PFM_CNTR3	PM-Cache Hit	Records the total number of occurrences of PM-cache hits
PFM_CNTR4	Crosscheck Hit	Records the total number of occurrences of crosscheck hits

Table 11-2: PM-Cache Mode (Continued)

Count Register	Function	Description
PFM_CNTR5	PM-Cache Miss without Writeback	Records the total number of occurrences of PM-cache misses without writeback
PFM_CNTR6	PM-Cache Miss with Writeback	Records the total number of occurrences of PM-cache misses with writeback

Table 11-3: I-Cache Mode

Count Register	Function	Description
PFM_CNTR3	I-Cache Hit	Records the total number of occurrences of I-cache hits
PFM_CNTR4	I-Cache Miss	Records the total number of occurrences of I-cache misses
PFM_CNTR5	Reserved	N/A
PFM_CNTR6	Reserved	N/A

Each counter register has a Start (PFM_CNTR3START, PFM_CNTR4START, PFM_CNTR5START, PFM_CNTR6START), Pause (PFM_CNTR3PAUSE, PFM_CNTR4PAUSE, PFM_CNTR5PAUSE, PFM_CNTR6PAUSE), and Clear (PFM_CNTR3CLR, PFM_CNTR4CLR, PFM_CNTR5CLR, PFM_CNTR6CLR) register associated with it. A write access to one of the start registers will initiate the recording of the count in the corresponding count register. Similarly, a write access to a pause register will pause or resume the count, and a write access to a clear register will clear the value in the corresponding counter register.

NOTE: The start, pause, and clear registers are non-resettable MMR registers. To avoid pipeline related write latencies, put 11 NOP instructions before a write to one of these registers.

12 Program Trace Macrocell (PTM)

The processor core implements Program Trace Macrocell (PTM) which implements a subset of Coresight Program Flow Trace Architecture (CSPFT) specification by Arm and provides instruction trace capability. For Cortex A5 trace unit features refer to the *Embedded Trace Macrocell (ETM)* chapter the hardware reference manual.

Features

The trace module has the following features

- Address comparators and Context ID comparators for filtering trace data and use as event resources.
- External inputs and outputs for use as event resources.
- Events can be created using address comparators, context ID comparators and external inputs.
- Counters to count events occurrences.

Functional Description

The following section describes the features available in the trace module.

Address Comparators

The trace module provides 4 address comparators. Program the Address Comparator Value register with the address to be matched and the corresponding Address Comparator Access Type register with additional information about the required comparison shown in the following list.

- Include or exclude range
- Linking the address comparison with Context ID comparator

Address comparators can be used

- Individually, as single address comparators (SACs)
- In pairs, as address range comparators (ARCs), in which case two adjacent address comparators form an ARC.

Context ID Comparators

The trace module provides 1 Context ID comparator.

The Context ID comparator consists of a Context ID Comparator Value Register which can hold a Context ID value, for comparison with the current Context ID and a Context ID Comparator Mask Register which can hold a mask value, which is used to mask all Context ID comparisons. If Context ID Comparator Mask Register is programmed to zero then no mask is applied to the Context ID comparisons.

Events

The trace module includes a number of event resources, address comparators, context ID comparators and external inputs.

Event resources can be used to define events. Event register can be programmed to define the corresponding event as the result of a logical operation involving one or two event resources.

Each event resource is either active or inactive, active event resource generates a logical TRUE signal and an inactive event resource generates a logic FALSE signal. An event is logical combinational of event resources, therefore at any given time each event is either TRUE or FALSE.

Counters

The trace module provides 2 counters that are controlled using events. Each 16-bit counter can count from 0 to 65535. Counter behavior is controlled by the following registers.

Counter Enable Event Register

Enables the counter and counts down while the counter enable event is TRUE.

Counter Reload Event Register

Reloades the counter from the Counter Reload Value Register when a counter reload event occurs.

Counter Reload Value Register

Holds the value that is loaded into the counter when the counter reload event is TRUE.

Counter Value Register

Finds the current value of the counter at any time through a read and writes a new value into the counter when programming the trace module.

Trace Security

The trace module supports that is controlled by the Debug Enable input signal. It controls whether the trace module is allowed to trace instructions. If this signal is deasserted, all tracing will stop, all internal resources are disabled and trace module's state is held.

Programming Model

The trace module registers are memory-mapped in a 4KB region as per CoreSight programmers model.

References

- CoreSight™ Program Flow Trace™ Architecture Specification - Arm IHI 0035B – Available at http://infocenter.arm.com
- CoreSight™ Architecture Specification - Arm IHI 0029B – Available at http://infocenter.arm.com

13 Instruction Set Reference

In the SHARC+ core family two different instruction types are supported.

- Instruction Set Architecture (ISA) is the traditional instruction set and is supported by all the SHARC and SHARC+ processors.
- Variable Instruction Set Architecture (VISA) is supported by the newer (ADSP-214xx and beyond) processors.

The instruction types linked into normal word space are valid ISA instructions (48-bit). When linked into short word space they become valid VISA instructions (48/32/16 bits).

Many ISA instruction types have conditions and compute/data move options. However, as programmer there may be situations where options in an instruction are not required. Moreover, many instructions have spare bits which are unused. For ISA instructions the opcode always consumes 48 bits, which results in wasted memory space. For VISA instruction types, all possible options have been extracted to generate new sub instructions resulting in 32-bit or 16-bit instructions.

This chapter provides information on the instructions associated with the SHARC+ core. Each instruction group has an overview table of its instruction types. The opcodes relating to the instruction types are shown with each instruction. For information on computation types and their associated opcodes (ALU, multiplier, shifter, multi-function) see the Computation Reference chapter.

Instruction Groups

The instruction groups are:

- Group I Conditional Compute and Move or Modify Instruction
- Group II Conditional Program Flow Control Instructions
- Group III Immediate Data Move Instructions
- Group IV Miscellaneous Instructions

The following tables provide an overview of the Group I-IV instructions. The letter after the instruction type denotes the instruction size as follows: a = 48-bit, b = 32-bit, c = 16-bit, d = 48-bit. Note that items in italics are optional. In the Introduction chapter the differences in instruction set are listed versus previous SHARC processor generations.

Instruction Set Notation Summary

The conventions for instruction syntax descriptions appear in the ***Instruction Set Notation*** table. Other parts of the instruction syntax and opcode information also appear in this section.

Table 13-1: Instruction Set Notation

<table>
<tr><th>Notation</th><th>Meaning</th></tr>
<tr><td>UPPERCASE</td><td>Explicit syntax—assembler keyword (notation only; assembler is case-insensitive and lowercase is the preferred programming convention)</td></tr>
<tr><td>;</td><td>Semicolon (instruction terminator)</td></tr>
<tr><td>,</td><td>Comma (separates parallel operations in an instruction)</td></tr>
<tr><td>italics</td><td>Optional part of instruction</td></tr>
<tr><td><code>| option1 | | option2 |</code></td><td>List of options between vertical bars (choose one)</td></tr>
<tr><td>compute</td><td>ALU, multiplier, shifter or multifunction operation. (See the Computation Reference chapter.)</td></tr>
<tr><td>shiftimm</td><td>Shifter immediate operation. (See the Computation Reference chapter.)</td></tr>
<tr><td>cond</td><td>Status condition (see condition codes in the Program Sequencer chapter)</td></tr>
<tr><td>termination</td><td>Loop termination condition (see condition codes in the Program Sequencer chapter)</td></tr>
<tr><td>ureg</td><td>Universal register</td></tr>
<tr><td>cureg</td><td>Complementary universal register (see the Register Files chapter)</td></tr>
<tr><td>sreg</td><td>System register</td></tr>
<tr><td>csreg</td><td>Complementary system register (see the Register Files chapter)</td></tr>
<tr><td>dreg</td><td>Data register (register file): R15–R0 or F15–F0</td></tr>
<tr><td>cdreg</td><td>Complementary data register (register file): S15–S0 or SF15–SF0 (see the Register Files chapter)</td></tr>
<tr><td>Ia</td><td>I7–I0 (DAG1 index register)</td></tr>
<tr><td>Mb</td><td>M7–M0 (DAG1 modify register)</td></tr>
<tr><td>Ic</td><td>I15–I8 (DAG2 index register)</td></tr>
<tr><td>Md</td><td>M15–M8 (DAG2 modify register)</td></tr>
<tr><td><datan></td><td>n-bit immediate data value</td></tr>
<tr><td><addrn></td><td>n-bit immediate address value</td></tr>
<tr><td><reladdrn></td><td>n-bit immediate PC-relative address value</td></tr>
<tr><td>+k</td><td>the implicit incremental address depending on SISD, SIMD or Broadcast mode</td></tr>
<tr><td>RTS</td><td>Return from subroutine</td></tr>
<tr><td>RTI</td><td>Return from interrupt</td></tr>
</table>

Table 13-1: Instruction Set Notation (Continued)

Notation	Meaning
(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)
(nw)	Normal word
(sw)	Short word
(bw)	Byte word
(se)	Sign extension
(ex)	Exclusive access

The list of `UREGs` (universal registers) can be found in the Register Files chapter.

14 Group I Conditional Compute and Move or Modify Instruction

The group I instructions contain a condition, a computation, and a data move operation.

The COND field selects whether the operation specified in the COMPUTE field and a data move is executed. If the COND is true, the compute and data move are executed. If no condition is specified, COND is true condition, and the compute and data move are executed.

The COMPUTE field specifies a compute operation using the ALU, multiplier, or shifter. Because there are a large number of options available for computations, these operations are described separately in the Computation Reference chapter.

Table 14-1: Group I Instructions by Instruction Type

Type	Addr	Option1	Option2	Operation (Option)	Modifier (Option)
1a	ISA VISA		*compute*,	`DM(Ia,Mb) = Dreg, PM(Ic,Md) = Dreg;` `Dreg = PM(Ic,Md), Dreg = DM(Ia,Mb);`	
1b	VISA			`DM(Ia,Mb) = Dreg, PM(Ic,Md) = Dreg;` `Dreg = PM(Ic,Md), Dreg = DM(Ia,Mb);`	
2a	ISA VISA	`IF` *cond*		*compute;*	
2b	VISA			*compute;*	
2c	VISA			*short compute;*	
3a	ISA VISA	`IF` *cond*	*compute*,	DM(*Ia*,*Mb*) = *Ureg* DM(*Mb*,*Ia*) = *Ureg* PM(*Ic*,*Md*) = *Ureg* PM(*Md*,*Ic*) = *Ureg* *Ureg* = DM(*Ia*,*Mb*) *Ureg* = DM(*Mb*,*Ia*) *Ureg* = PM(*Ic*,*Md*) *Ureg* = PM(*Md*,*Ic*)	`(lw);`

Table 14-1: Group I Instructions by Instruction Type (Continued)

Type	Addr	Option1	Option2	Operation (Option)	Modifier (Option)
3b	VISA	IF *cond*		DM(*Ia*,*Mb*) = *Ureg* DM(*Mb*,*Ia*) = *Ureg* PM(*Ic*,*Md*) = *Ureg* PM(*Md*,*Ic*) = *Ureg*	(bw/sw);
				Ureg = DM(*Ia*,*Mb*) *Ureg* = DM(*Mb*,*Ia*) *Ureg* = PM(*Ic*,*Md*) *Ureg* = PM(*Md*,*Ic*)	(bw/sw); (bwse,swse);
3c	VISA			*Dreg* = DM(*Ia*,*Mb*); DM(*Ia*,*Mb*) = *Dreg*;	
3d	ISA VISA	IF *cond*		DM(*Ia*,*Mb*) = *Ureg* DM(*Mb*,*Ia*) = *Ureg* PM(*Ic*,*Md*) = *Ureg* PM(*Md*,*Ic*) = *Ureg*	(bw/sw/lw,ex);
				Ureg = DM(*Ia*,*Mb*) *Ureg* = DM(*Mb*,*Ia*) *Ureg* = PM(*Ic*,*Md*) *Ureg* = PM(*Md*,*Ic*)	(bw/sw/lw,ex); (bwse/swse,ex);
4a	ISA VISA	IF *cond*	*compute*,	DM(*Ia*, <*data6*>) = *Dreg*; DM(<*data6*>,*Ia*) = *Dreg*; PM(*Ic*, <*data6*>) = *Dreg*; PM(<*data6*>,*Ic*) = *Dreg*; *Dreg* = DM(*Ia*, <*data6*>); *Dreg* = DM(<*data6*>,*Ia*); *Dreg* = PM(*Ic*, <*data6*>); *Dreg* = PM(<*data6*>,*Ic*);	
4b	VISA	IF *cond*		DM(*Ia*, <*data6*>) = *Dreg*; DM(<*data6*>,*Ia*) = *Dreg*; PM(*Ic*, <*data6*>) = *Dreg*; PM(<*data6*>,*Ic*) = *Dreg*; *Dreg* = DM(*Ia*, <*data6*>); *Dreg* = DM(<*data6*>,*Ia*); *Dreg* = PM(*Ic*, <*data6*>); *Dreg* = PM(<*data6*>,*Ic*);	
4d	ISA VISA	IF *cond*		DM(*Ia*, <*data6*>) = *Dreg*; DM(<*data6*>,*Ia*) = *Dreg*; PM(*Ic*, <*data6*>) = *Dreg*; PM(<*data6*>,*Ic*) = *Dreg*;	(bw/sw);
				Dreg = DM(*Ia*, <*data6*>); *Dreg* = DM(<*data6*>,*Ia*); *Dreg* = PM(*Ic*, <*data6*>); *Dreg* = PM(<*data6*>,*Ic*);	(bw/sw); (bwse/swse);
5a (move)	ISA VISA	IF *cond*	*compute*,	*Ureg1* = *Ureg2*;	
5a (swap)	ISA VISA	IF *cond*	*compute*,	*Dreg* <-> *CDreg*;	

Table 14-1: Group I Instructions by Instruction Type (Continued)

Type	Addr	Option1	Option2	Operation (Option)	Modifier (Option)
5b (move)	VISA	IF *cond*		*Ureg1* = *Ureg2*;	
5b (swap)	VISA	IF *cond*		*Dreg* <-> *CDreg*;	
6a (mem)	ISA VISA	IF *cond*	*shif-timm*,	DM(*Ia*,*Mb*) = *Dreg*; PM(*Ic*,*Md*) = *Dreg*; *Dreg* = DM(*Ia*,*Mb*); *Dreg* = PM(*Ic*,*Md*);	
6a (no-mem)	ISA VISA	IF *cond*		*shiftimm*;	
7a	ISA VISA	IF *cond*	*compute*,	MODIFY(*Ia*,*Mb*); MODIFY(*Ic*,*Md*); Ia = MODIFY(Ia,Mb) Ic = MODIFY(Ic,Md)	(nw/sw);
7b	VISA	IF *cond*		MODIFY(*Ia*,*Mb*); MODIFY(*Ic*,*Md*); Ia = MODIFY(Ia,Mb) Ic = MODIFY(Ic,Md)	
7d	ISA VISA	IF *cond*	*compute*,	*Ia* = B2W(*Ia*); *Ic* = B2W(*Ic*); *Ia* = W2B(*Ia*); *Ic* = W2B(*Ic*); *Ba* = B2W(*Ba*); *Bc* = B2W(*Bc*); *Ba* = W2B(*Ba*); *Bc* = W2B(*Bc*);	

Type 1a ISA/VISA (compute + mem dual data move)

Syntax Summary

Table 14-2: Group I Instructions by Instruction Type

Type	Addr	Option1	Option2	Operation (Option)	Operation (Option)
1a	ISA VISA		compute,	DM(Ia,Mb) = Dreg, PM(Ic,Md) = Dreg; Dreg = PM(Ic,Md), Dreg = DM(Ia,Mb);	

The following table provides the opcode field values (compute) and the instruction syntax overview (Syntax)

Compute	Syntax
00000000000000000000000	DMACCESS (Type 1a) , PMACCESS (Type 1a) ;

Compute	Syntax
-----------------------	COMPUTE , DMACCESS (Type 1a) , PMACCESS (Type 1a) ;

For more information about compute syntax, see the Computation Reference chapter.

Abstract

Compute with a parallel memory (data and program) transfer

Description

It is important to understand how this instruction operates differently in SISD, SIMD, and Broadcast modes.

SISD Mode

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

SIMD Mode

In SIMD mode, the Type 1 instruction provides the same parallel accesses to data and program memory from the register file as are available in SISD mode, but provides these operations simultaneously for the X and Y processing elements. The X element uses the specified I registers to address data and program memory, and the Y element adds one to the specified I registers to address data and program memory. The I values are post-modified and updated by the specified M registers. Pre-modify offset addressing is not supported. For more information on register restrictions, see the Data Address Generators chapter. 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 the Complementary Data Register Pairs description in the Register Files chapter.

Broadcast Mode

If the broadcast control read bits—`REGF_MODE1.BDCST1` (for I1) or `REGF_MODE1.BDCST9` (for I9)—are set, both processing units (PEx/PEy) share the same index address. The following code compares the Type 1 instruction's explicit and implicit operations in SIMD and Broadcast modes.

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

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

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

```
compute, DM(Ia+k, 0) = cdreg , PM(Ic+k, 0) = cdreg ;
```

```
compute, cdreg = DM(Ia+k, 0) , cdreg = PM(Ic+k, 0) ;
```

If Broadcast Load Mode memory read k=0. If SIMD mode NW access k=1, SW access k=2, BW access k=4.

Example

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

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

When the processors are in SIMD mode, the first instruction in this example performs the same computation and performs two writes in parallel on both PEx and PEy. The R7 register on PEx and S7 on PEy both store the results of the Bset computations. Also, simultaneous dual memory writes occur with DM and PM, writing in values from R5, S5 (DM) and R4, S4 (PM) respectively. In the second instruction, values are simultaneously read from data memory to registers R8 and S8 and written to program memory from registers R0 and S0.

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

When the processors are in broadcast mode (the BDCST1 bit is set in the MODE1 system register), the R0 (PEx) data register in this example is loaded with the value from data memory utilizing the I1 register from DAG1, and S0 (PEy) is loaded with the same value.

Type 1a Instruction Opcode

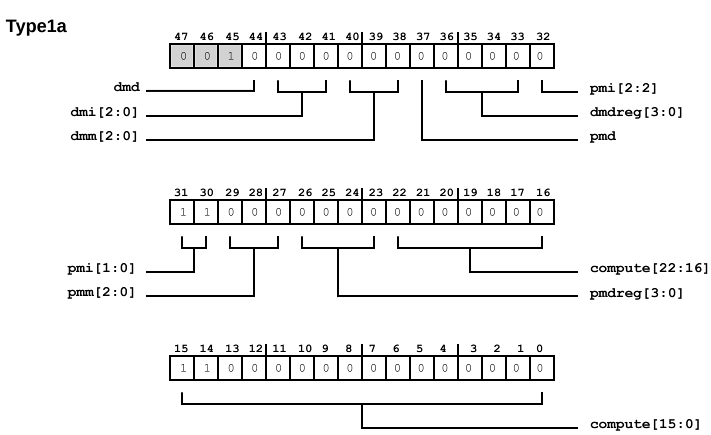

Figure 14-1: Type1a Instruction

DMACCESS (Type 1a)

DMACCESS Encode Table

dmd	Syntax
0	RFREG Register Class = dm(I1REG Register Class, M1REG Register Class)

dmd	Syntax
1	dm(I1REG Register Class, M1REG Register Class) = RFREG Register Class

PMACCESS (Type 1a)

PMACCESS Encode Table

pmd	Syntax
0	RFREG Register Class = pm(I2REG Register Class, M2REG Register Class)
1	pm(I2REG Register Class, M2REG Register Class) = RFREG Register Class

Type 1b VISA (mem dual data move)

Syntax Summary

Table 14-3: Group I Instructions by Instruction Type

Type	Addr	Option1	Option2	Operation (Option)	Modifier (Option)
1b	VISA			`DM(Ia,Mb) = Dreg, PM(Ic,Md) = Dreg;` `Dreg = PM(Ic,Md), Dreg = DM(Ia,Mb);`	

The following table provides the instruction syntax overview (Syntax)

Syntax
DMACCESS (Type 1b) , PMACCESS (Type 1b) ;

Abstract

Parallel memory (data and program) transfer

Description

It is important to understand how this instruction operates differently in SISD, SIMD, and Broadcast modes.

SISD Mode

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

SIMD Mode

In SIMD mode, the Type 1 instruction provides the same parallel accesses to data and program memory from the register file as are available in SISD mode, but provides these operations simultaneously for the X and Y processing elements. The X element uses the specified I registers to address data and program memory, and the Y element adds one to the specified I registers to address data and program memory. The I values are post-modified and updated by the specified M registers. Pre-modify offset addressing is not supported. For more information on register restrictions, see the Data Address Generators chapter. 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 the Complementary Data Register Pairs description in the Register Files chapter.

Broadcast Mode

If the broadcast control read bits—`REGF_MODE1.BDCST1` (for I1) or `REGF_MODE1.BDCST9` (for I9)—are set, both processing units (PEx/PEy) share the same index address. The following code compares the Type 1 instruction's explicit and implicit operations in SIMD and Broadcast modes.

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

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

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

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

If Broadcast Load Mode memory read k=0. If SIMD mode NW access k=1, SW access k=2, BW access k=4.

Type1b Instruction Opcode

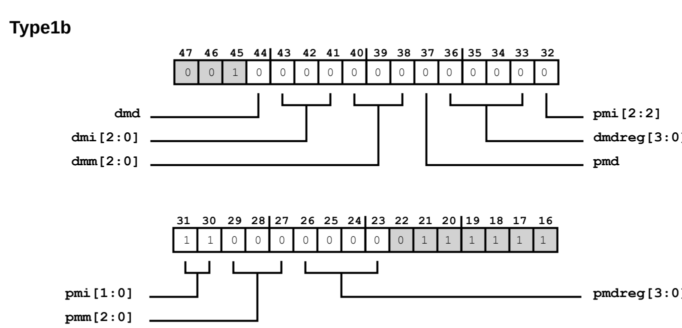

Figure 14-2: Type1b Instruction

DMACCESS (Type 1b)

DMACCESS Encode Table

dmd	Syntax
0	RFREG Register Class = dm(I1REG Register Class, M1REG Register Class)
1	dm(I1REG Register Class, M1REG Register Class) = RFREG Register Class

PMACCESS (Type 1b)

PMACCESS Encode Table

pmd	Syntax
0	RFREG Register Class = pm(I2REG Register Class, M2REG Register Class)
1	pm(I2REG Register Class, M2REG Register Class) = RFREG Register Class

Type 2a ISA/VISA (cond + compute)

Syntax Summary

Table 14-4: Group I Instructions by Instruction Type

Type	Addr	Option1	Option2	Operation (Option)	Modifier (Option)
2a	ISA VISA	`IF` *`cond`*		`compute;`	

The following table provides the opcode field values (cond) and the instruction syntax overview (Syntax)

cond	Syntax
11111	COMPUTE ;
-----	IFCOND COMPUTE ;

Abstract

Compute operation, condition

Description

SISD Mode

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

SIMD Mode

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

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

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

```
IF PEx COND compute ;
```

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

```
IF PEy COND compute ;
```

Example

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

When the processors are in SISD mode, the condition is evaluated in the PEx processing element. If the condition is true, the computation is performed and the result is stored in register `R6`.

When the processors are in SIMD mode, the condition is evaluated on each processing element, PEx and PEy, independently. The computation executes on both PEs, either one PE, or neither PE dependent on the outcome of the condition. If the condition is true in PEx, the computation is performed and the result is stored in register `R6`. If the condition is true in PEy, the computation is performed and the result is stored in register `S6`.

Type2a Instruction Opcode

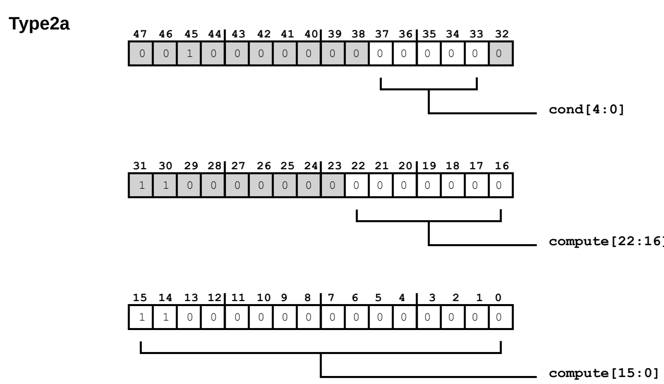

Figure 14-3: Type2a Instruction

Type 2b VISA (compute)

Syntax Summary

Table 14-5: Group I Instructions by Instruction Type

Type	Addr	Option1	Option2	Operation (Option)	Modifier (Option)
2b	VISA			*compute;*	

The following table provides the instruction syntax overview (Syntax)

Syntax
COMPUTE ;

Abstract

Compute operation, *without* the Type 2 condition

Description

SISD Mode

In SISD mode, the Type 2 instruction provides a `compute` instruction.

SIMD Mode

In SIMD mode, the Type 2 instruction provides the same `compute` instruction as is available in SISD mode, but provides the operation simultaneously for the X and Y processing elements.

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)

```
compute ;
```

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

```
compute ;
```

Type2b Instruction Opcode

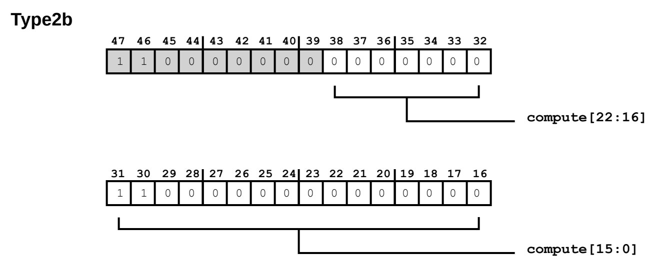

Figure 14-4: Type2b Instruction

Type 2c VISA (short compute)

Syntax Summary

Table 14-6: Group I Instructions by Instruction Type

Type	Addr	Option1	Option2	Operation (Option)	Modifier (Option)
2c	VISA			*short compute;*	

The following table provides the instruction syntax overview (Syntax)

Syntax
SHORTCOMPUTE ;

Syntax

Short (16-bit) compute operation, *without* the Type 2 condition

Description

SISD Mode

In SISD mode, the Type 2 instruction provides a `compute` instruction.

SIMD Mode

In SIMD mode, the Type 2 instruction provides the same `compute` instruction as is available in SISD mode, but provides the operation simultaneously for the X and Y processing elements.

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)

```
COND compute ;
```

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

```
COND compute ;
```

Type2c Instruction Opcode

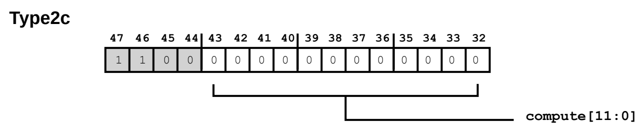

Figure 14-5: Type2c Instruction

Type 3a ISA/VISA (cond + comp + mem data move)

Syntax Summary

Table 14-7: Group I Instructions by Instruction Type

Type	Addr	Option1	Option2	Operation (Option)	Modifier (Option)
3a	ISA VISA	`IF` *cond*	*compute;*	DM(*Ia*,*Mb*) = *Ureg* DM(*Mb*,*Ia*) = *Ureg* PM(*Ic*,*Md*) = *Ureg* PM(*Md*,*Ic*) = *Ureg* *Ureg* = DM(*Ia*,*Mb*) *Ureg* = DM(*Mb*,*Ia*) *Ureg* = PM(*Ic*,*Md*) *Ureg* = PM(*Md*,*Ic*)	(lw);

The following table provides the opcode field values (cond, compute) and the instruction syntax overview (Syntax)

cond	compute	Syntax
11111	00000000000000000000000	ACCESS (Type 3a) ;
11111	-----------------------	COMPUTE , ACCESS (Type 3a) ;
-----	00000000000000000000000	IFCOND ACCESS (Type 3a) ;
-----	-----------------------	IFCOND COMPUTE , ACCESS (Type 3a) ;

Abstract

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

Description

SISD Mode

In SISD mode, the Type 3a and 3b instruction provides access between data or program memory and a universal register. The specified I register addresses data or program memory. The I value is either pre-modified

(M, I order) or post-modified (I, M order) by the specified M register. If it is post-modified, the I register is updated with the modified value. If a `compute` operation is specified, it is performed in parallel with the data access. The optional `(LW)` in this syntax lets programs specify long word addressing, overriding default addressing from the memory map. If a `condition` is specified, it affects the entire instruction. Note that the *`Ureg`* may not be from the same DAG (that is, DAG1 or DAG2) as `Ia/Mb` or `Ic/Md`.

SIMD Mode

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

The X element uses the specified I register to address data or program memory. The I value is either pre-modified (M, I order) or post-modified (I, M order) by the specified M register. The Y element adds one/two (for normal/short word access) to the specified I register (before pre-modify or post-modify) to address data or program memory. If the I value post-modified, the I register is updated with the modified value from the specified M register. The optional `(LW)` in this syntax lets programs specify long word addressing, overriding default addressing from the memory map.

For the universal register, the X element uses the specified *`Ureg`* register, and the Y element uses the corresponding complementary register (*`Cureg`*). Note that the *`Ureg`* may not be from the same DAG (DAG1 or DAG2) as `Ia/Mb` or `Ic/Md`.

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

Broadcast Mode

If the broadcast read bits—`BDCST1` (for `I1`) or `BDCST9` (for `I9`)—are set, the Y element uses the specified I and M registers without implicit address addition.

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

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

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

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

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

IF PEx COND compute, ureg = DM(Mb, Ia) (LW);
```

```
IF PEx COND compute, ureg = PM(Md, Ic) (LW);
```

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

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

IF PEy COND compute, DM(Mb+k, Ia) = cureg (LW);
IF PEy COND compute, PM(Md+k, Ic) = cureg (LW);

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

IF PEy COND compute, cureg = DM(Mb+k, Ia) (LW);
IF PEy COND compute, cureg = PM(Md+k, Ic) (LW);
```

If Broadcast Load Mode memory read k=0. If SIMD mode NW access k=1, SW access k=2, BW access k=4.

Example

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

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

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

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

When the processors are in broadcast mode (the `BDCST9` bit is set in the `MODE1` system register) and the condition tests true, the computation is performed and the result is stored in register `F8`. Also, the result of the program memory read via the `I9` register from DAG2 is stored in `F7`. The `SF7` register is loaded with the same value from program memory as `F7`.

Type3a Instruction Opcode

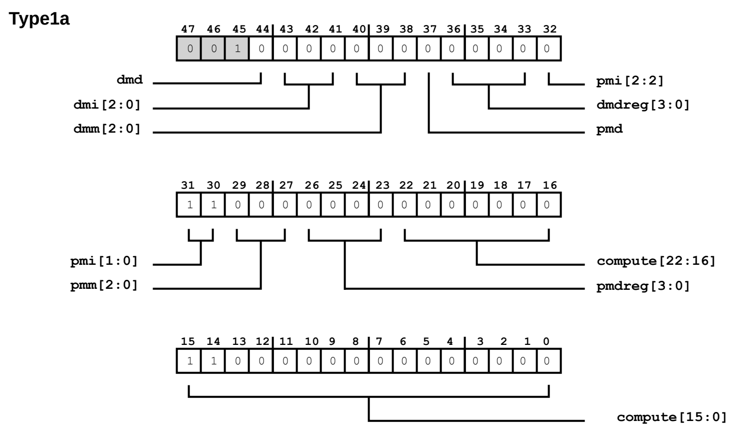

Figure 14-6: Type3a Instruction

ACCESS (Type 3a)

ACCESS Encode Table

u	g	d	l	Syntax
0	0	0	0	UREG Registers Class = dm(M1REG Register Class, I1REG Register Class)
0	0	1	0	dm(M1REG Register Class, I1REG Register Class) = UREGXDAG1 Register Class
0	1	0	0	UREG Registers Class = pm(M2REG Register Class, I2REG Register Class)
0	1	1	0	pm(M2REG Register Class, I2REG Register Class) = UREGXDAG2 Register Class
1	0	0	0	UREGXDAG1 Register Class = dm(I1REG Register Class, M1REG Register Class)
1	0	1	0	dm(I1REG Register Class, M1REG Register Class) = UREGXDAG1 Register Class
1	1	0	0	UREGXDAG2 Register Class = pm(I2REG Register Class, M2REG Register Class)
1	1	1	0	pm(I2REG Register Class, M2REG Register Class) = UREGXDAG2 Register Class
0	0	0	1	UREG Registers Class = dm(M1REG Register Class, I1REG Register Class) (lw)
0	0	1	1	dm(M1REG Register Class, I1REG Register Class) = UREGXDAG1 Register Class (lw)
0	1	0	1	UREG Registers Class = pm(M2REG Register Class, I2REG Register Class) (lw)
0	1	1	1	pm(M2REG Register Class, I2REG Register Class) = UREGXDAG2 Register Class (lw)
1	0	0	1	UREGXDAG1 Register Class = dm(I1REG Register Class, M1REG Register Class) (lw)
1	0	1	1	dm(I1REG Register Class, M1REG Register Class) = UREGXDAG1 Register Class (lw)
1	1	0	1	UREGXDAG2 Register Class = pm(I2REG Register Class, M2REG Register Class) (lw)
1	1	1	1	pm(I2REG Register Class, M2REG Register Class) = UREGXDAG2 Register Class (lw)

Type 3b VISA (cond + mem data move)

Syntax Summary

Table 14-8: Type 3b VISA (cond + mem data move)

Type	Addr	Option1	Option2	Operation (Option)	Modifier (Option)
3b	VISA	IF *cond*		DM(*Ia*,*Mb*) = *Ureg* DM(*Mb*,*Ia*) = *Ureg* PM(*Ic*,*Md*) = *Ureg* PM(*Md*,*Ic*) = *Ureg*	(bw/sw);
				Ureg = DM(*Ia*,*Mb*) *Ureg* = DM(*Mb*,*Ia*) *Ureg* = PM(*Ic*,*Md*) *Ureg* = PM(*Md*,*Ic*)	(bw/bwse/sw/swse);

The following table provides the opcode field values (cond) and the instruction syntax overview (Syntax)

cond	Syntax
11111	ACCESS (Type 3b) ;
-----	IFCOND ACCESS (Type 3b) ;

Abstract

Transfer operation between data or program memory and universal register, optional condition, *without* the Type 3 optional compute operation

Description

SISD Mode

In SISD mode, the Type 3a and 3b instruction provides access between data or program memory and a universal register. The specified I register addresses data or program memory. The I value is either pre-modified (M, I order) or post-modified (I, M order) by the specified M register. If it is post-modified, the I register is updated with the modified value. The optional (LW) in this syntax lets programs specify long word addressing, overriding default addressing from the memory map. If a condition is specified, it affects the entire instruction. Note that the *Ureg* may not be from the same DAG (that is, DAG1 or DAG2) as Ia/Mb or Ic/Md.

The optional (BW), (BWSE), (SW), and (SWSE), may only be used when the I-register addresses byte space. (BW) specifies a byte access; the 8-bit value loaded into a register is zero extended to 32-bits and the value stored is the low order 8-bits of the 32-bit value in the register. (SW) specifies a short word access; the 16-bit value loaded into a register is zero extended to 32-bits and the value stored is the low order 16-bits of the 32-bit value in the register. (BWSE) and (SWSE) may only be used on loads and specify the 8-bit value is sign extended or 16-bit value is sign extended respectively.

SIMD Mode

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

The X element uses the specified I register to address data or program memory. The I value is either pre-modified (M, I order) or post-modified (I, M order) by the specified M register. The Y element adds one/two (for normal/short word access) to the specified I register (before pre-modify or post-modify) to address data or program memory. If the I value post-modified, the I register is updated with the modified value from the specified M register.

The optional (LW) in this syntax lets programs specify long word addressing, overriding default addressing from the memory map and overriding SIMD mode, so these loads always operate in SISD mode. The optional (BW), (BWSE), (SW), and (SWSE) work in SIMD mode but may only be used when the I-register addresses byte space. In each case the memory loaded or stored from the complementary register appears in memory immediately after the location explicitly addressed. So a (BW) load loads the addressed byte to the named register and next byte to its complementary register. (SW) and (SWSE) accesses do not work like SIMD access to short word address space.

For the universal register, the X element uses the specified *Ureg* register, and the Y element uses the corresponding complementary register (*Cureg*). Note that the *Ureg* may not be from the same DAG (DAG1 or DAG2) as Ia/Mb or Ic/Md.

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

Broadcast Mode

If the broadcast read bits—BDCST1 (for I1) or BDCST9 (for I9)—are set, the Y element uses the specified I and M registers without implicit address addition.

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

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

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

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

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

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

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

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

IF PEy COND DM(Mb+k, Ia) = cureg (LW);
IF PEy COND PM(Md+k, Ic) = cureg (LW);

IF PEy COND cureg = DM(Ia+k, 0) (LW);
IF PEy COND cureg = PM(Ic+k, 0) (LW);

IF PEy COND cureg = DM(Mb+k, Ia) (LW);
IF PEy COND cureg = PM(Md+k, Ic) (LW);
```

If Broadcast Load Mode memory read k=0. If SIMD mode NW access k=1, SW access k=2, BW access k=4.

Type3b Instruction Opcode

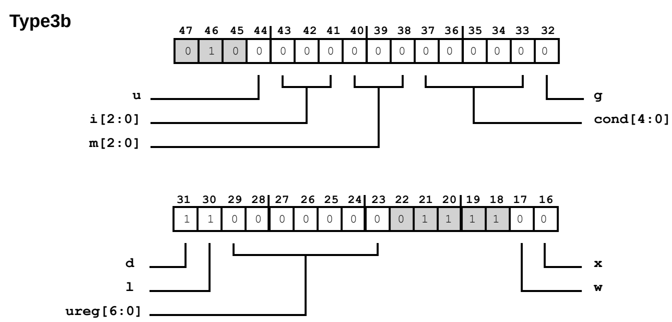

Figure 14-7: Type3b Instruction

ACCESS (Type 3b)

ACCESS Encode Table

u	g	d	l	w	x	Syntax
0	0	0	1	1	1	UREG Registers Class = dm(M1REG Register Class, I1REG Register Class) (lw)
0	0	0	-	-	-	UREG Registers Class = dm(M1REG Register Class, I1REG Register Class) BHSE (Type 3b)
0	0	1	1	1	1	dm(M1REG Register Class, I1REG Register Class) = UREGXDAG1 Register Class (lw)
0	0	1	-	-	-	dm(M1REG Register Class, I1REG Register Class) = UREGXDAG1 Register Class BH (Type 3b)
0	1	0	1	1	1	UREG Registers Class = pm(M2REG Register Class, I2REG Register Class) (lw)

u	g	d	l	w	x	Syntax
0	1	0	-	-	-	UREG Registers Class = pm(M2REG Register Class, I2REG Register Class) BHSE (Type 3b)
0	1	1	1	1	1	pm(M2REG Register Class, I2REG Register Class) = UREGXDAG2 Register Class (lw)
0	1	1	-	-	-	pm(M2REG Register Class, I2REG Register Class) = UREGXDAG2 Register Class BH (Type 3b)
1	0	0	1	1	1	UREGXDAG1 Register Class = dm(I1REG Register Class, M1REG Register Class) (lw)
1	0	0	-	-	-	UREGXDAG1 Register Class = dm(I1REG Register Class, M1REG Register Class) BHSE (Type 3b)
1	0	1	1	1	1	dm(I1REG Register Class, M1REG Register Class) = UREGXDAG1 Register Class (lw)
1	0	1	-	-	-	dm(I1REG Register Class, M1REG Register Class) = UREGXDAG1 Register Class BH (Type 3b)
1	1	0	1	1	1	UREGXDAG2 Register Class = pm(I2REG Register Class, M2REG Register Class) (lw)
1	1	0	-	-	-	UREGXDAG2 Register Class = pm(I2REG Register Class, M2REG Register Class) BHSE (Type 3b)
1	1	1	1	1	1	pm(I2REG Register Class, M2REG Register Class) = UREGXDAG2 Register Class (lw)
1	1	1	-	-	-	pm(I2REG Register Class, M2REG Register Class) = UREGXDAG2 Register Class BH (Type 3b)

BH (Type 3b)

BH Encode Table

l	x	w	Syntax
0	1	1	
0	0	0	(bw)
1	0	0	(sw)

BHSE (Type 3b)

BHSE Encode Table

l	x	w	Syntax
0	1	1	
0	0	0	(bw)
0	1	0	(bwse)
1	0	0	(sw)

l	x	w	Syntax
1	1	0	(swse)

Type 3c VISA (mem data move)

Syntax Summary

Table 14-9: Group I Instructions by Instruction Type

Type	Addr	Option1	Option2	Operation (Option)	Modifier (Option)
3c	VISA			`Dreg = DM(Ia,Mb);` `DM(Ia,Mb) = Dreg;`	

The following table provides the instruction syntax overview (Syntax)

Syntax
ACCESS (Type 3c) ;

Abstract

Transfer operation between data memory and data register, *without* the Type 3 optional condition, *without* the Type 3 optional compute operation, without (LW) modifier.

Description

SISD Mode

In SISD mode, the Type 3d instruction provides access between data or program memory and a universal register. The specified I register addresses data or program memory. The DAG1 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.

SIMD Mode

In SIMD mode, the Type 3d 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 DAG1 I value is either pre-modified (M, I order) or post-modified (I, M order) by the specified M register. The Y element adds one/two (for normal/short word access) to the specified I register (before pre-modify or post-modify) to address data or program memory. If the I value post-modified, the I register is updated with the modified value from the specified M register.

For the universal register, the X element uses the specified *Ureg* register, and the Y element uses the corresponding complementary register (*Cureg*).

Broadcast Mode

If the broadcast read bits—BDCST1 (for I1) or BDCST9 (for I9)—are set, the Y element uses the specified I and M registers without implicit address addition.

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

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

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

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

```
DM(Ia+k, 0) = cureg;
PM(Ic+k, 0) = cureg;
cureg = DM(Ia+k, 0);
cureg = PM(Ic+k, 0);
```

If Broadcast Load Mode memory read k=0. If SIMD mode NW access k=1, SW access k=2, BW access k=4.

Type3c Instruction Opcode

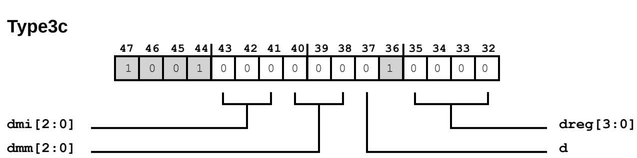

Figure 14-8: Type3c Instruction

ACCESS (Type 3c)

ACCESS Encode Table

d	Syntax
0	RFREG Register Class = dm(I1REG Register Class, M1REG Register Class)
1	dm(I1REG Register Class, M1REG Register Class) = RFREG Register Class

Type 3d ISA/VISA (cond + exclusive mem data move)

Syntax Summary

NOTE: The 48-bit 3d instruction type is an extension to 3a instruction (exclusive access without compute option).

Table 14-10: Group I Instructions by Instruction Type

Type	Addr	Option1	Option2	Operation (Option)	Modifier (Option)

3d	ISA
	VISA
		`IF cond`		`DM(Ia,Mb) = Ureg`
				`DM(Mb,Ia) = Ureg`
				`PM(Ic,Md) = Ureg`
				`PM(Md,Ic) = Ureg`
					`(bw/sw/lw,ex);`

				`Ureg = DM(Ia,Mb)`
				`Ureg = DM(Mb,Ia)`
				`Ureg = PM(Ic,Md)`
				`Ureg = PM(Md,Ic)`
					`(bw/sw/lw,ex);`
					`(bwse/swse,ex);`

The following table provides the opcode field values (w, cond) and the instruction syntax overview (Syntax)

w	cond	Syntax
0	11111	ACCESS (Type 3d) ;
1	11111	WACCESS (Type 3d) ;
0	-----	IFCOND ACCESS (Type 3d) ;
1	-----	IFCOND WACCESS (Type 3d) ;

Abstract

Transfer operation between data or program memory and universal register with options for byte address sub-word access and exclusive access, condition, compute operation

Description

The type 3d exclusive instruction exists to provide a 48-bit encoding of some options of the Type 3 instruction that is not supported by Type 3a. The `(EX)` specifies an exclusive access. The optional `(BW)`, `(BWSE)`, `(SW)`, and `(SWSE)`, may only be used when the I-register addresses byte space. `(BW)` specifies a byte access; the 8-bit value loaded into a register is zero extended to 32-bits and the value stored is the low order 8-bits of the 32-bit value in the register. `(SW)` specifies a short word access; the 16-bit value loaded into a register is zero extended to 32-bits and the value stored is the low order 16-bits of the 32-bit value in the register. `(BWSE)` and `(SWSE)` may only be used on loads and specify the 8-bit value is sign extended or 16-bit value is sign extended respectively. These options may be used in SISD and SIMD mode. This option may be combined with `(LW)`, `(BW)`, `(BWSE)`, `(SW)`, `(SWSE)` which is written `(LW, EX)` etc., or used alone to specify a normal word exclusive access.

Example

```
R1=R1+1, R3 = DM(I0,M4) (SWSE,EX);
```

(SWSE) may only be used with byte addresses so the contents of the I0 register is first checked and if it does not address byte space the Illegal address space interrupt is raised. Otherwise execution proceeds as follows.

SISD Mode

When the processor is in SISD mode, the computation is performed on PEx and result written to R1. The load reads 16-bits from the two byte addressed memory locations at I0 and I0+1 in little endian order. This 16-bit value is sign extended to 32-bits and the 32-bit value deposited left justified in the 40-bit R3 register. So the 8-bits at the address in I0 are copied to bits 15 to 8 of R3 and the 8-bits at the address in I0 copies to bits 23 to 16 of R3. Bits 39 to 24 are all set to the same value as bit 23 and bits 7 to 0 to zero. I0 is then updated as for Scaled Address Arithmetic (see Enhanced Modify Instruction for Address Scaling) the value in M4 is multiplied by two and added to I0.

SIMD Mode

When the processor is in SIMD mode, the computation is performed on both PEx and PEy. The result of the computation on PEX is written to R1 and the result of the computation on PEy is written to S1. The load reads 32-bits from the 4 byte addressed memory locations at I0 and I0+1, I0+2, I0+3. The 16-bit little endian value at I0 is sign extended and deposited in R3 as described above, and the 16-bit little endian value at I0+2 is similarly sign extended and deposited in S3. I0 is then updated as for Scaled Address Arithmetic (see Enhanced Modify Instruction for Address Scaling) the value in M4 is multiplied by two and added to I0.

Broadcast Mode

If the broadcast read bits—BDCST1 (for I1) or BDCST9 (for I9) are set, the Y element uses the specified I and M registers without implicit address addition.

Type3d Instruction Opcode

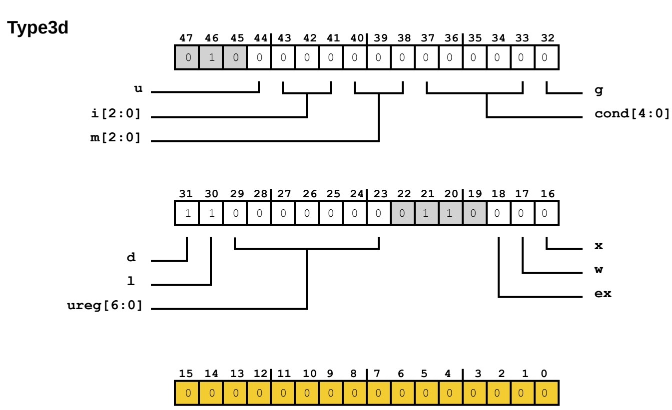

Figure 14-9: Type3d Syntax

ACCESS (Type 3d)

ACCESS Encode Table

u	g	d	Syntax
0	0	0	UREG Registers Class = dm(M1REG Register Class, I1REG Register Class) BHSE (Type 3d)
0	0	1	dm(M1REG Register Class, I1REG Register Class) = UREGXDAG1 Register Class BH (Type 3d)
0	1	0	UREG Registers Class = pm(M2REG Register Class, I2REG Register Class) BHSE (Type 3d)
0	1	1	pm(M2REG Register Class, I2REG Register Class) = UREGXDAG2 Register Class BH (Type 3d)
1	0	0	UREGXDAG1 Register Class = dm(I1REG Register Class, M1REG Register Class) BHSE (Type 3d)
1	0	1	dm(I1REG Register Class, M1REG Register Class) = UREGXDAG1 Register Class BH (Type 3d)
1	1	0	UREGXDAG2 Register Class = pm(I2REG Register Class, M2REG Register Class) BHSE (Type 3d)
1	1	1	pm(I2REG Register Class, M2REG Register Class) = UREGXDAG2 Register Class BH (Type 3d)

BH (Type 3d)

BH Encode Table

l	ex	Syntax
0	1	(bw,ex)
1	1	(sw,ex)

BHSE (Type 3d)

BHSE Encode Table

l	x	ex	Syntax
0	0	1	(bw,ex)
1	0	1	(sw,ex)
0	1	1	(bwse,ex)
1	1	1	(swse,ex)

EX (Type 3d)

EX Encode Table

Syntax
(ex)

LWEX (Type 3d)

LWEX Encode Table

Syntax
(lw,ex)

WACCESS (Type 3d)

WACCESS Encode Table

u	g	d	l	ex	Syntax
0	0	0	0	1	UREG Registers Class = dm(M1REG Register Class, I1REG Register Class) EX (Type 3d)
0	0	1	0	1	dm(M1REG Register Class, I1REG Register Class) = UREGXDAG1 Register Class EX (Type 3d)
0	1	0	0	1	UREG Registers Class = pm(M2REG Register Class, I2REG Register Class) EX (Type 3d)
0	1	1	0	1	pm(M2REG Register Class, I2REG Register Class) = UREGXDAG2 Register Class EX (Type 3d)
1	0	0	0	1	UREGXDAG1 Register Class = dm(I1REG Register Class, M1REG Register Class) EX (Type 3d)
1	0	1	0	1	dm(I1REG Register Class, M1REG Register Class) = UREGXDAG1 Register Class EX (Type 3d)
1	1	0	0	1	UREGXDAG2 Register Class = pm(I2REG Register Class, M2REG Register Class) EX (Type 3d)
1	1	1	0	1	pm(I2REG Register Class, M2REG Register Class) = UREGXDAG2 Register Class EX (Type 3d)
0	0	0	1	1	UREG Registers Class = dm(M1REG Register Class, I1REG Register Class) LWEX (Type 3d)
0	0	1	1	1	dm(M1REG Register Class, I1REG Register Class) = UREGXDAG1 Register Class LWEX (Type 3d)
0	1	0	1	1	UREG Registers Class = pm(M2REG Register Class, I2REG Register Class) LWEX (Type 3d)
0	1	1	1	1	pm(M2REG Register Class, I2REG Register Class) = UREGXDAG2 Register Class LWEX (Type 3d)
1	0	0	1	1	UREGXDAG1 Register Class = dm(I1REG Register Class, M1REG Register Class) LWEX (Type 3d)
1	0	1	1	1	dm(I1REG Register Class, M1REG Register Class) = UREGXDAG1 Register Class LWEX (Type 3d)
1	1	0	1	1	UREGXDAG2 Register Class = pm(I2REG Register Class, M2REG Register Class) LWEX (Type 3d)
1	1	1	1	1	pm(I2REG Register Class, M2REG Register Class) = UREGXDAG2 Register Class LWEX (Type 3d)

Type 4a ISA/VISA (cond + comp + mem data move with 6-bit immediate modifier)

Syntax Summary

Table 14-11: Group I Instructions by Instruction Type

Type	Addr	Option1	Option2	Operation (Option)	Modifier (Option)

4a	ISA
	VISA
		`IF cond`	`compute,`	`DM(Ia, <data6>) = Dreg;`
				`DM(<data6>,Ia) = Dreg;`
				`PM(Ic, <data6>) = Dreg;`
				`PM(<data6>,Ic) = Dreg;`
				`Dreg = DM(Ia, <data6>);`
				`Dreg = DM(<data6>,Ia);`
				`Dreg = PM(Ic, <data6>);`
				`Dreg = PM(<data6>,Ic);`	

The following table provides the opcode field values (cond, compute) and the instruction syntax overview (Syntax)

cond	compute	Syntax
11111	00000000000000000000000	ACCESS (Type 4a) ;
11111	-----------------------	COMPUTE , ACCESS (Type 4a) ;
-----	00000000000000000000000	IFCOND ACCESS (Type 4a) ;
-----	-----------------------	IFCOND COMPUTE , ACCESS (Type 4a) ;

Abstract

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

Description

SISD Mode

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

SIMD Mode

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

The X element uses the specified I register to address data or program memory. The I value is either pre-modified (data, I order) or post-modified (I, data order) by the specified immediate data. The Y element adds one/two (for normal/short word access) to the specified I register (before pre-modify or post-modify) to address data or program memory. If the I value post-modified, the I register is updated with the modified value from the specified M register.

For the data register, the X element uses the specified `Dreg` register, and the Y element uses the corresponding complementary register (`Cdreg`).

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

Broadcast Mode

If the broadcast read bits—`BDCST1` (for `I1`) or `BDCST9` (for `I9`)—are set, the Y element uses the specified I and M registers without adding one.

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

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

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

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

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

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

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

```
IF PEy COND compute, DM(Ia+k, 0) = cdreg ;
IF PEy COND compute, PM(Ic+k, 0) = cdreg ;

IF PEy COND compute, DM(<data6>+k, Ia)= cdreg ;
IF PEy COND compute, PM(<data6>+k, Ic)= cdreg ;

IF PEy COND compute, cdreg = DM(Ia+k, 0) ;
```

```
IF PEy COND compute, cdreg = PM(Ic+k, 0) ;

IF PEy COND compute, cdreg = DM(<data6>+k, Ia) ;
IF PEy COND compute, cdreg = PM(<data6>+k, Ic) ;
```

If Broadcast Load Mode memory read k=0. If SIMD mode NW access k=1, SW access k=2, BW access k=4.

Example

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

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

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

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

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

Type4a Instruction Opcode

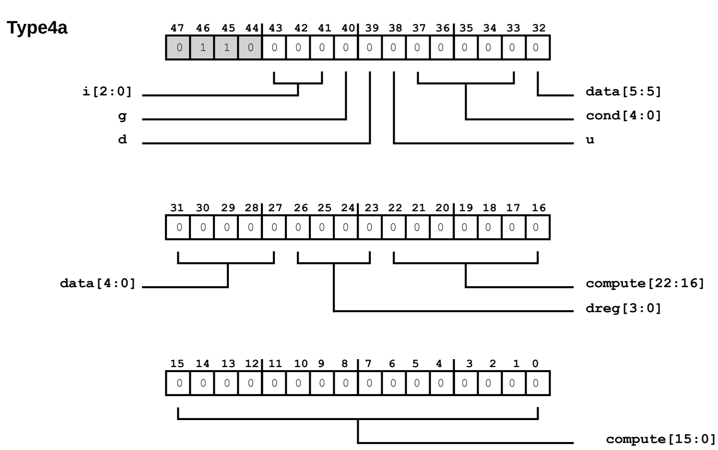

Figure 14-10: Type4a Instruction

ACCESS (Type 4a)

ACCESS Encode Table

g	d	u	Syntax
0	0	0	RFREG Register Class = dm(imm6 Register Type, I1REG Register Class)
0	0	1	RFREG Register Class = dm(I1REG Register Class, imm6 Register Type)
0	1	0	dm(imm6 Register Type, I1REG Register Class) = RFREG Register Class
0	1	1	dm(I1REG Register Class, imm6 Register Type) = RFREG Register Class
1	0	0	RFREG Register Class = pm(imm6 Register Type, I2REG Register Class)
1	0	1	RFREG Register Class = pm(I2REG Register Class, imm6 Register Type)
1	1	0	pm(imm6 Register Type, I2REG Register Class) = RFREG Register Class
1	1	1	pm(I2REG Register Class, imm6 Register Type) = RFREG Register Class

Type 4b VISA (cond + mem data move with 6-bit immediate modifier)

Syntax Summary

Table 14-12: Group I Instructions by Instruction Type

Type	Addr	Option1	Option2	Operation (Option)	Modifier (Option)

4b	VISA	`IF cond`		`DM(Ia, <data6>) = Dreg;`
				`DM(<data6>,Ia) = Dreg;`
				`PM(Ic, <data6>) = Dreg;`
				`PM(<data6>,Ic) = Dreg;`
				`Dreg = DM(Ia, <data6>);`
				`Dreg = DM(<data6>,Ia);`
				`Dreg = PM(Ic, <data6>);`
				`Dreg = PM(<data6>,Ic);`	

The following table provides the opcode field values (cond) and the instruction syntax overview (Syntax)

cond	Syntax
11111	ACCESS (Type 4b) ;
-----	IFCOND ACCESS (Type 4b) ;

Abstract

Index-relative transfer between data or program memory and register file, optional condition, without the Type 4a optional compute operation.

Description

SISD Mode

In SISD mode, the Type 4b 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 condition is specified, it affects the entire instruction. For more information on register restrictions, see the Register Files chapter and the Data Address Generator chapter.

SIMD Mode

In SIMD mode, the Type 4b instruction provides the same access between data or program memory and the register file as is available in SISD mode, but provides the operation simultaneously for the X and Y processing elements. The X element uses the specified I register to address data or program memory. The I value is either pre-modified (data, I order) or post-modified (I, data order) by the specified immediate data. The Y element adds one/two (for normal/short word access) to the specified I register (before pre-modify or post-modify) to address data or program memory. If the I value post-modified, the I register is updated with the modified value from the specified M register. For the data register, the X element uses the specified Dreg register, and the Y element uses the corresponding complementary register (Cdreg). If a condition is specified, it affects the entire instruction. The instruction is executed in a processing element if the specified condition tests true in that element independent of the condition result for the other element.

Broadcast Mode

If the broadcast read bits—`BDCST1` (for `I1`) or `BDCST9` (for `I9`)—are set, the Y element uses the specified I and M registers without adding one.

The following 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 DM(Ia, <data6>) = dreg ;
IF PEx COND PM(Ic, <data6>) = dreg ;

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

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

IF PEx COND dreg = DM(<data6>, Ia) ;
```

```
IF PEx COND dreg = PM(<data6>, Ic) ;
```

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

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

IF PEy COND DM(<data6>+k, Ia)= cdreg ;
IF PEy COND PM(<data6>+k, Ic)= cdreg ;

IF PEy COND cdreg = DM(Ia+k, 0) ;
IF PEy COND cdreg = PM(Ic+k, 0) ;

IF PEy COND cdreg = DM(<data6>+k, Ia) ;
IF PEy COND cdreg = PM(<data6>+k, Ic) ;
```

If Broadcast Load Mode memory read k=0. If SIMD mode NW access k=1, SW access k=2, BW access k=4.

Type4b Instruction Opcode

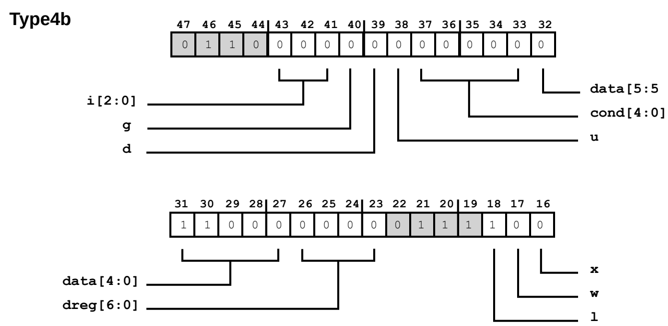

Figure 14-11: Type4b Instruction

ACCESS (Type 4b)

ACCESS Encode Table

g	d	u	Syntax
0	0	0	RFREG Register Class = dm(imm6visa Register Type, I1REG Register Class) BHSE (Type 4b)
0	0	1	RFREG Register Class = dm(I1REG Register Class, imm6visa Register Type) BHSE (Type 4b)
0	1	0	dm(imm6visa Register Type, I1REG Register Class) = RFREG Register Class BH (Type 4b)
0	1	1	dm(I1REG Register Class, imm6visa Register Type) = RFREG Register Class BH (Type 4b)
1	0	0	RFREG Register Class = pm(imm6visa Register Type, I2REG Register Class) BHSE (Type 4b)
1	0	1	RFREG Register Class = pm(I2REG Register Class, imm6visa Register Type) BHSE (Type 4b)
1	1	0	pm(imm6visa Register Type, I2REG Register Class) = RFREG Register Class BH (Type 4b)
1	1	1	pm(I2REG Register Class, imm6visa Register Type) = RFREG Register Class BH (Type 4b)

BH (Type 4b)

BH Encode Table

l	x	w	Syntax
1	1	1	
0	0	0	(bw)
1	0	0	(sw)

BHSE (Type 4b)

BHSE Encode Table

l	x	w	Syntax
1	1	1	
0	0	0	(bw)
1	0	0	(sw)
0	1	0	(bwse)
1	1	0	(swse)

Type 4d ISA/VISA (cond + mem data move with 6-bit immediate modifier)

Syntax Summary

NOTE: The 48-bit 4d instruction type is an extension to 4a instruction but does not support compute option.

Table 14-13: Group I Instructions by Instruction Type

Type	Addr	Option1	Option2	Operation (Option)	Modifier (Option)
4d	ISA VISA	`IF` *`cond`*		`DM(`*`Ia, <data6>`*`) = `*`Dreg;`* `DM(`*`<data6>,Ia`*`) = `*`Dreg;`* `PM(Ic, <data6>) = `*`Dreg;`* `PM(`*`<data6>,Ic`*`) = `*`Dreg;`*	`(bw/sw);`
				`Dreg`` = DM(`*`Ia, <data6>`*`);` *`Dreg`*` = DM(`*`<data6>,Ia`*`);` *`Dreg`*` = PM(`*`Ic, <data6>`*`);` *`Dreg`*` = PM(`*`<data6>,Ic`*`);`	`(bwse/swse);`

The following table provides the opcode field values (cond) and the instruction syntax overview (Syntax)

cond	Syntax
11111	ACCESS (Type 4d) ;
-----	IFCOND ACCESS (Type 4d) ;

Abstract

Index-relative transfer between data or program memory and register file, optional condition, with optional compute operation supporting options for byte address sub-word access and exclusive access.

Description

The type 4d instruction exists to provide a 48-bit encoding of some options of the Type 4 instruction that is not supported by Type 4a.

The optional `(BW)`, `(BWSE)`, `(SW)`, and `(SWSE)`, may only be used when the I-register addresses byte space. (BW) specifies a byte access; the 8-bit value loaded into a register is zero extended to 32-bits and the value stored is the low order 8-bits of the 32-bit value in the register. `(SW)` specifies a short word access; the 16-bit value loaded into a register is zero extended to 32-bits and the value stored is the low order 16-bits of the 32-bit value in the register. `(BWSE)` and `(SWSE)` may only be used on loads and specify the 8-bit value is sign extended or 16-bit value is sign extended respectively.

These options may be used in SISD and SIMD mode.

Example

```
IF SV R4=R5*R6, R3 = DM(I0,2) (SWSE);
```

`(SWSE)` may only be used with byte addresses so the contents of the `I0` register is first checked and if it does not address byte space the Illegal address space interrupt is raised. Otherwise execution proceeds as follows.

When the processor is in SISD mode, if the `SV` flag in `ASTATX` is unset no operation is performed. If the `SV` flag in `ASTATX` is set the multiply is executed on PEx and result written to R4. The load reads 16-bits from the two byte addressed memory locations at I0 and `I0+1` in little endian order. This 16-bit value is sign extended to 32-bits and the 32-bit value deposited left justified in the 40-bit R3 register. So the 8-bits at the address in `I0` are copied to bits 15 to 8 of R3 and the 8-bits at the address in `I0` copies to bits 23 to 16 of R3. Bits 39 to 24 are all set to the same value as bit 23 and bits 7 to 0 to zero. `I0` is then updated as for Scaled Address Arithmetic (see Enhanced Modify Instruction for Address Scaling) the literal modifier, 2, is multiplied by the size of a short word, 2, and added to the value in I0. This advances the address in `I0` by 4.

When the processor is in SIMD mode, the computation and load are performed on PEx if the SV flag is set in `ASTATX` and on PEy if the `SV` flag is set in `ASTATY`. The result of the multiply on PEx if executed is written to R4 and the result the multiply on PEy to S4. The load on PEx reads 16-bits from the 2-bytes addressed by `I0`, sign extends the 16-bit little endian to 32-bits and deposits in R3 as described above. The load on PEy reads 16-bits from the 2-bytes addressed by `I0+2`, sign extends the 16-bit little endian to 32-bits and deposits in S3. `I0` is then updated as for Scaled Address Arithmetic (see Enhanced Modify Instruction for Address Scaling) the literal modifier, 2, is multiplied by the size of a short word, 2, and added to the value in `I0`. This advances the address in I0 by 4.

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 Broadcast Load Mode memory read k=0. If SIMD mode NW access k=1, SW access k=2, BW access k=4.

Type4d Instruction Opcode

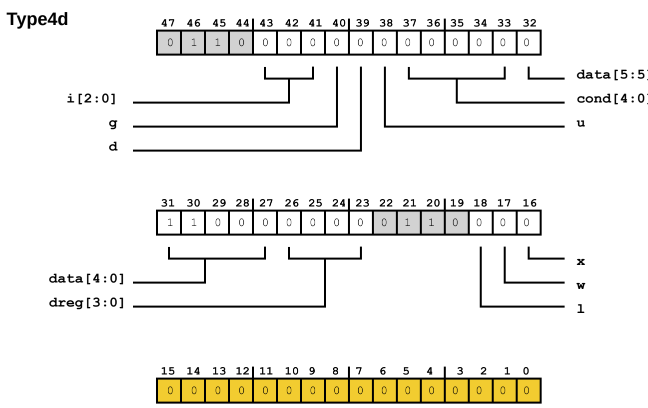

Figure 14-12: Type4d Instruction

ACCESS (Type 4d)

ACCESS Encode Table

g	d	u	Syntax
0	0	0	RFREG Register Class = dm(imm6 Register Type, I1REG Register Class) BHSE (Type 4d)
0	0	1	RFREG Register Class = dm(I1REG Register Class, imm6 Register Type) BHSE (Type 4d)
0	1	0	dm(imm6 Register Type, I1REG Register Class) = RFREG Register Class BH (Type 4d)
0	1	1	dm(I1REG Register Class, imm6 Register Type) = RFREG Register Class BH (Type 4d)
1	0	0	RFREG Register Class = pm(imm6 Register Type, I2REG Register Class) BHSE (Type 4d)
1	0	1	RFREG Register Class = pm(I2REG Register Class, imm6 Register Type) BHSE (Type 4d)
1	1	0	pm(imm6 Register Type, I2REG Register Class) = RFREG Register Class BH (Type 4d)
1	1	1	pm(I2REG Register Class, imm6 Register Type) = RFREG Register Class BH (Type 4d)

BH (Type 4d)

BH Encode Table

l	Syntax
0	(bw)
1	(sw)

BHSE (Type 4d)

BHSE Encode Table

l	x	Syntax
0	0	(bw)
1	0	(sw)
0	1	(bwse)
1	1	(swse)

Type 5a ISA/VISA (cond + comp + reg data move)

Syntax Summary

Table 14-14: Group I Instructions by Instruction Type

Type	Addr	Option1	Option2	Operation (Option)	Modifier (Option)
5a	ISA VISA	`IF cond`	`compute,`	`Ureg1 = Ureg2;`	

The following table provides the opcode field values (cond, compute) and the instruction syntax overview (Syntax)

cond	compute	Syntax
11111	00000000000000000000000	UREG Registers Class = UREG Registers Class ;
11111	-----------------------	COMPUTE , UREG Registers Class = UREG Registers Class ;
-----	00000000000000000000000	IFCOND UREG Registers Class = UREG Registers Class ;
-----	-----------------------	IFCOND COMPUTE , UREG Registers Class = UREG Registers Class ;

Abstract

Transfer between two universal registers in each processing element, optional condition, optional compute operation

Description

SISD Mode

In SISD mode, the Type 5 instruction provides transfer (=) from one universal register to another. If a `compute` operation is specified, it is performed in parallel with the data access. If a `condition` is specified, it affects the entire instruction.

SIMD Mode

In SIMD mode, the Type 5 instruction provides the same transfer (=) from one register to another as is available in SISD mode, but provides this operation simultaneously for the X and Y processing elements.

In the transfer (=), the X element transfers between the universal registers *Ureg*1 and *Ureg*2, and the Y element transfers between the complementary universal registers *Cureg*1 and *Cureg*2. For a list of complementary registers, see the Register Files chapter.

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 ;
```

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

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

Example

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

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

When the processors are in SIMD mode, the condition is evaluated on each processing element, PEx and PEy, independently. The computation executes on both PEs, either one PE, or neither PE dependent on the outcome of the condition. For the register transfer to complete, the condition must be satisfied in both PEx and PEy. The second instruction initializes the loop counter independent of the outcome of the first instruction's condition.

Type5a_move Instruction Opcode

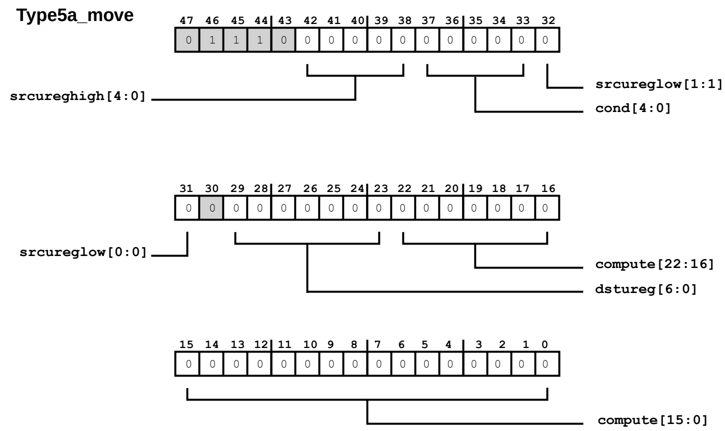

Figure 14-13: Type5a_move Instruction

Type 5a ISA/VISA (cond + comp + reg data swap)

Syntax Summary

Table 14-15: Group I Instructions by Instruction Type

Type	Addr	Option1	Option2	Operation (Option)	Modifier (Option)
5a (swap)	ISA VISA	`IF cond`	`compute,`	`Dreg1 <-> CDreg2;`	

The following table provides the opcode field values (cond, compute) and the instruction syntax overview (Syntax)

cond	compute	Syntax
11111	00000000000000000000000	RFREG Register Class <-> SREG Register Class ;
11111	-----------------------	COMPUTE , RFREG Register Class <-> SREG Register Class ;
-----	00000000000000000000000	IFCOND RFREG Register Class <-> SREG Register Class ;
-----	-----------------------	IFCOND COMPUTE , RFREG Register Class <-> SREG Register Class ;

Abstract

Swap between a data register in each processing element, optional condition, optional compute operation

Description

SISD Mode

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

SIMD Mode

In SIMD mode, the swap (<->) operation does the same operation in SISD and SIMD modes; no extra swap operation occurs in SIMD mode.

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, dreg <-> cdreg ;
```

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

```
IF PEy COND compute ;/* no implicit operation */
```

Example

```
R0 <-> S1;
```

The instruction swaps the register contents between `R0` and `S1`—the SISD and SIMD swap operation is the same.

Type5a (swap) Instruction Opcode

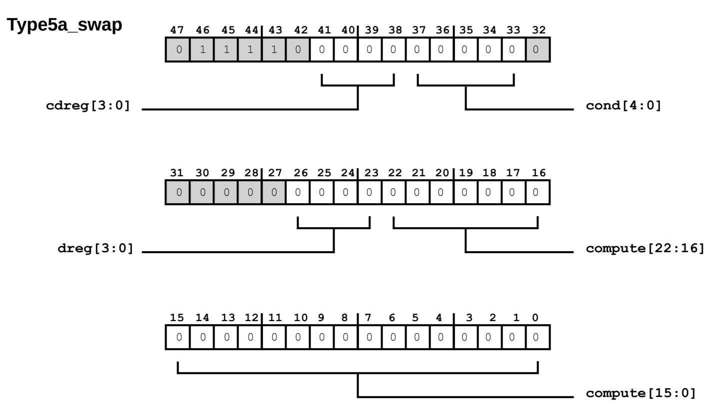

Figure 14-14: Type5a (swap) Instruction

Type 5b VISA (cond + reg data move)

Syntax Summary

Table 14-16: Group I Instructions by Instruction Type

Type	Addr	Option1	Option2	Operation (Option)	Modifier (Option)
5b (move)	VISA	`IF` *`cond`*	*`compute,`*	*`Ureg1 = Ureg2;`*	

The following table provides the opcode field values (cond) and the instruction syntax overview (Syntax)

cond	Syntax
11111	UREG Registers Class = UREG Registers Class ;
-----	IFCOND UREG Registers Class = UREG Registers Class ;

Abstract

Transfer between two universal registers optional condition.

Description

SISD Mode

In SISD mode, the Type 5 instruction provides transfer (=) from one universal register to another.

SIMD Mode

In SIMD mode, the Type 5 instruction provides the same transfer (=) from one register to another as is available in SISD mode, but provides this operation simultaneously for the X and Y processing elements.

In the transfer (=), the X element transfers between the universal registers *`Ureg`*1 and *`Ureg`*2, and the Y element transfers between the complementary universal registers *`Cureg`*1 and *`Cureg`*2. For a list of complementary registers, see the Register Files chapter.

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 ureg1 = ureg2 ;
```

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

```
IF PEy COND cureg1 = cureg2 ;
```

Example

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

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

When the processors are in SIMD mode, the condition is evaluated on each processing element, PEx and PEy, independently. The computation executes on both PEs, either one PE, or neither PE dependent on the outcome of the condition. For the register transfer to complete, the condition must be satisfied in both PEx and PEy. The second instruction initializes the loop counter independent of the outcome of the first instruction's condition.

Type5b (move) Instruction Opcode

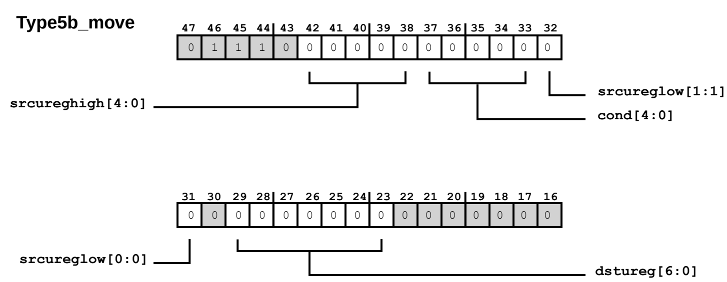

Figure 14-15: Type5b (move) Instruction

Type 5b VISA (cond + reg data swap)

Syntax Summary

Table 14-17: Group I Instructions by Instruction Type

Type	Addr	Option1	Option2	Operation (Option)	Modifier (Option)
5b (swap)	VISA	`IF` *`cond`*		*`Dreg1 = CDreg2;`*	

The following table provides the opcode field values (cond) and the instruction syntax overview (Syntax)

cond	Syntax
11111	RFREG Register Class <-> SREG Register Class ;
-----	IFCOND RFREG Register Class <-> SREG Register Class ;

Abstract

Swap between a data register in each processing element, optional condition.

Description

SISD Mode

In SISD mode, the Type 5 instruction provides a swap (<->) between a data register in the X processing element and a data register in the Y processing element. If a `condition` is specified, it affects the entire instruction.

SIMD Mode

In SIMD mode the swap (<->) operation does the same operation in SISD and SIMD modes; no extra swap operation occurs in SIMD mode.

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 dreg <-> cdreg ;
```

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

```
IF PEy COND ;/* no implicit operation */
```

Example

```
R0 <-> S1;
```

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

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

Type5b (swap) Instruction Opcode

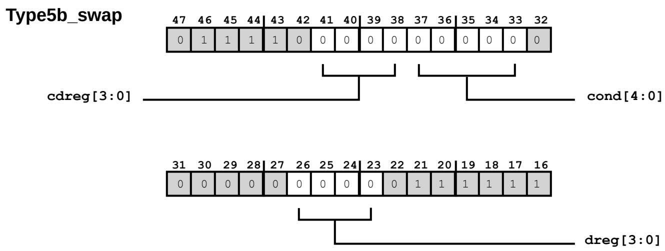

Figure 14-16: Type5b (swap) Instruction

Type 6a ISA/VISA (cond + shift imm + mem data move)

Syntax Summary

Table 14-18: Group I Instructions by Instruction Type

Type	Addr	Option1	Option2	Operation (Option)	Modifier (Option)
6a (mem)	ISA VISA	`IF` *`cond`*	*`shiftimm,`*	`DM(`*`Ia`*`,`*`Mb`*`) = `*`Dreg;`* `PM(`*`Ic`*`,`*`Md`*`) = `*`Dreg;`* *`Dreg`*` = DM(`*`Ia`*`,`*`Mb`*`);` *`Dreg`*` = PM(Ic,Md);`	

The following table provides the opcode field values (cond) and the instruction syntax overview (Syntax)

cond	Syntax
11111	SHIFTIMM , ACCESS (Type 6a) ;
-----	IFCOND SHIFTIMM , ACCESS (Type 6a) ;

Abstract

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

Description

SISD Mode

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

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

SIMD Mode

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

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

The X element uses the specified I register to address data or program memory. The I value is post-modified by the specified M register and updated with the modified value.The Y element adds one/two (for normal/short word access) to the specified I register to address data or program memory.

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

Broadcast Mode

If the broadcast read bits—`BDCST1` (for `I1`) or `BDCST9` (for `I9`)—are set, the Y element uses the specified I and M registers without adding one.

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

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

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

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

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

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

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

If Broadcast Load Mode memory read k=0. If SIMD mode NW access k=1, SW access k=2, BW access k=4.

Example

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

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

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

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

When the processors are in broadcast mode (the `BDCST1` bit is set in the `MODE1` system register), the computation is performed, the result is stored in `R2`, and the data memory value is read into register `F3` via the `I1` register from DAG1. The `SF3` register is also loaded with the same value from data memory as `F3`.

Type6a (mem) Instruction Opcode

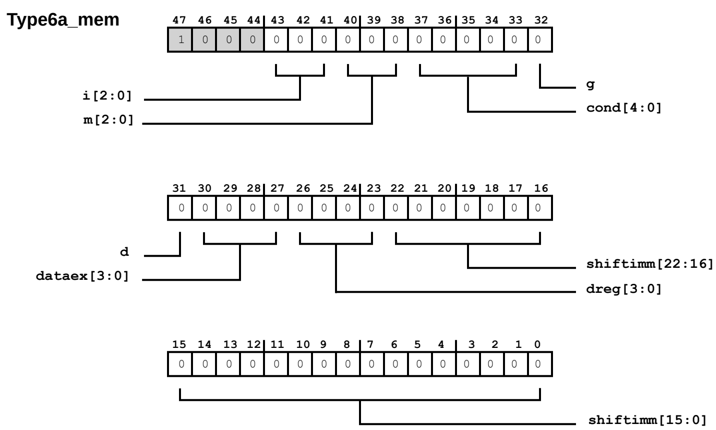

Figure 14-17: Type6a (mem) Instruction

ACCESS (Type 6a)

ACCESS Encode Table

g	d	Syntax
0	0	RFREG Register Class = dm(I1REG Register Class, M1REG Register Class)
0	1	dm(I1REG Register Class, M1REG Register Class) = RFREG Register Class
1	0	RFREG Register Class = pm(I2REG Register Class, M2REG Register Class)
1	1	pm(I2REG Register Class, M2REG Register Class) = RFREG Register Class

Type 6a ISA/VISA (cond + shift imm)

Syntax Summary

Table 14-19: Group I Instructions by Instruction Type

Type	Addr	Option1	Option2	Operation (Option)	Modifier (Option)
6a (no-mem)	ISA VISA	`IF cond`		`shiftimm;`	

The following table provides the opcode field values (cond) and the instruction syntax overview (Syntax)

cond	Syntax
11111	SHIFTIMM ;
-----	IFCOND SHIFTIMM ;

Abstract

Immediate shift operation, optional condition.

Description

SISD Mode

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

If a `condition` is specified, it affects the entire instruction.

SIMD Mode

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

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.

```
IF PEx COND shiftimm, ;
IF PEx COND shiftimm, ;
IF PEx COND shiftimm,  ;
IF PEx COND shiftimm, ;
```

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

```
IF PEy COND shiftimm, ;
IF PEy COND shiftimm, ;
IF PEy COND shiftimm, ;
IF PEy COND shiftimm, ;
```

Type6a (nomem) Instruction Opcode

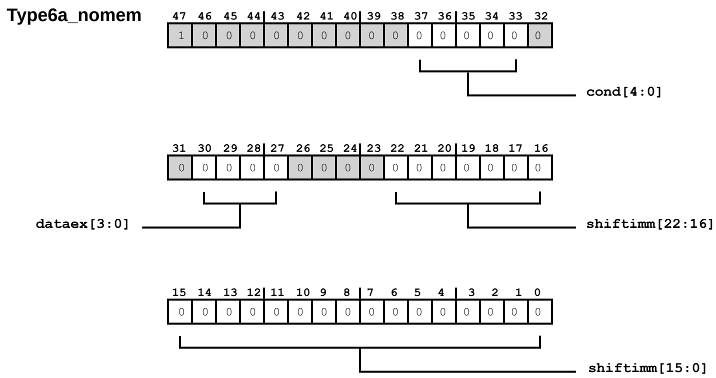

Figure 14-18: Type6a (nomem) Instruction

Type 7a ISA/VISA (cond + comp + index modify)

Syntax Summary

Table 14-20: Group I Instructions by Instruction Type

Type	Addr	Option1	Option2	Operation (Option)	Modifier (Option)

7a	ISA
	VISA
		`IF cond`	`compute,`	`MODIFY(Ia,Mb);`
				`MODIFY(Ic,Md);`
				`Ia = MODIFY(Ia,Mb)`
				`Ic = MODIFY(Ic,Md)`
					`(nw);`
					`(sw);`

The following table provides the opcode field values (cond, compute) and the instruction syntax overview (Syntax)

cond	compute	Syntax
11111	00000000000000000000000	MODIFY (Type 7a) ;
11111	-----------------------	COMPUTE , MODIFY (Type 7a) ;
-----	00000000000000000000000	IFCOND MODIFY (Type 7a) ;
-----	-----------------------	IFCOND COMPUTE , MODIFY (Type 7a) ;

Abstract

Index register modify, optional condition, optional compute operation

Description

SISD Mode

In SISD mode, the Type 7 instruction provides an update of the specified Ia/Ic register by the specified Mb/Md register. If the destination register is not specified, Ia/Ic is used as destination register. Unless destination I register is specified or implied to be the same as the source I register, the source I register is left unchanged. M register is always left unchanged. If a `compute` operation is specified, it is performed in parallel with the data access. If a `condition` is specified, it affects the entire instruction.

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

An optional `(SW)` specifies the `MODIFY` instruction should perform scaled address arithmetic for short words in byte address space. The value in the M-register is multiplied by 2 before adding to the value in the I-register. If the address in the I-register is not to the byte space then MODIFY `(SW)` will raise the Illegal address space interrupt.

An optional `(NW)` specifies the `MODIFY` instruction should perform scaled address arithmetic for normal words. If the value in the I-register addresses the normal word space then this instruction just adds the value in the M-register to the value in the I-register, but if the I-register addresses byte space then the value in the M-register is multiplied by 4 before adding to the value in the I-register. Thus the instruction increments the address by Mb/Md normal words in both address spaces which helps to write address-space neutral code. `MODIFY (NW)` raises the Illegal address space interrupt if the I-register addresses the long word or short word spaces.

SIMD Mode

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

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

The index register modify operation, in SIMD mode, occurs based on the logical ORing of the outcome of the conditions tested on both PEs. In the second instruction, the index register modify also occurs based on the logical ORing of the outcomes of the conditions tested on both PEs. Because both threads of a SIMD sequence may be dependent on a single DAG index value, either thread needs to be able to cause a modify of the index.

Example

```
IF NOT FLAG2_IN R4=R6*R12(SUF), MODIFY(I10,M8);
IF FLAG2_IN R4=R6*R12(SUF), I9 = MODIFY(I10,M8);
IF NOT LCE MODIFY(I3,M1);
IF NOT LCE I0 = MODIFY(I3,M1);
MODIFY(I10,M9);
I15 = MODIFY(I11,M12);
I0 = MODIFY(I2,M2);
```

```
I3 = MODIFY(I3,M5); /* Semantically same as MODIFY(I3,M5) */;
```

Type7a Instruction Opcode

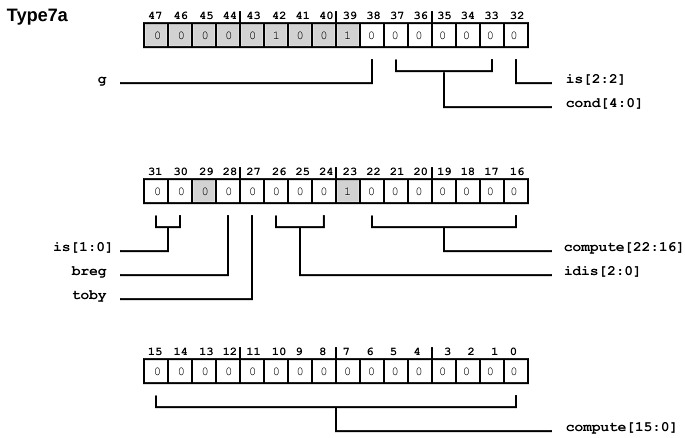

Figure 14-19: Type7a Instruction

BH (Type 7a)

BH Encode Table

w	l	Syntax
0	0	
0	1	(sw)
1	0	(nw)

MODIFY (Type 7a)

MODIFY Encode Table

g	Syntax
0	modify(I1REG Register Class, M1REG Register Class)
0	I1REG Register Class = modify(I1REG Register Class, M1REG Register Class) BH (Type 7a)
1	modify(I2REG Register Class, M2REG Register Class)
1	I2REG Register Class = modify(I2REG Register Class, M2REG Register Class) BH (Type 7a)

Type 7b VISA (cond + index modify)

Syntax Summary

Table 14-21: Group I Instructions by Instruction Type

Type	Addr	Option1	Option2	Operation (Option)	Modifier (Option)

7b	VISA	`IF cond`		`MODIFY(Ia,Mb);`
				`MODIFY(Ic,Md);`
				`Ia = MODIFY(Ia,Mb)`
				`Ic = MODIFY(Ic,Md)`
					`(nw);`
					`(sw);`

The following table provides the opcode field values (cond) and the instruction syntax overview (Syntax)

cond	Syntax
11111	MODIFY (Type 7b) ;
-----	IFCOND MODIFY (Type 7b) ;

Abstract

Index register modify, optional condition

Description

SISD Mode

In SISD mode, the Type 7 instruction provides an update of the specified Ia/Ic register by the specified Mb/Md register. If the destination register is not specified, Ia/Ic is used as destination register. Unless destination I register is specified or implied to be the same as the source I register, the source I register is left unchanged. M register is always left unchanged. If a `condition` is specified, it affects the entire instruction.

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

SIMD Mode

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

Type7b Instruction Opcode

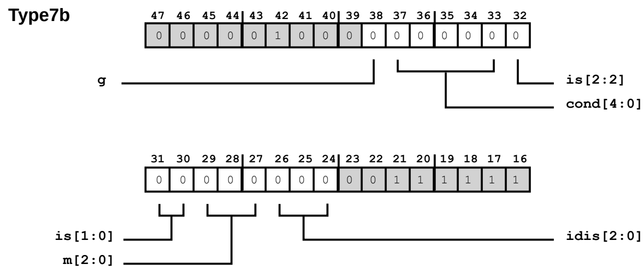

Figure 14-20: Type7b Instruction

MODIFY (Type 7b)

MODIFY Encode Table

g	Syntax
0	I1REG Register Class = modify(I1REG Register Class, M1REG Register Class)
1	I2REG Register Class = modify(I2REG Register Class, M2REG Register Class)

Type 7d ISA/VISA (cond + comp + address switch)

Syntax Summary

NOTE: The 48-bit 7d instruction type is an extension (address switch) to 7a instruction.

Table 14-22: Group I Instructions by Instruction Type

Type	Addr	Option1	Option2	Operation (Option)	Modifier (Option)
7d	ISA VISA	`IF` *`cond`*	`compute,`	*`Ia`* `= B2W(`*`Ia`*`);` *`Ic`* `= B2W(`*`Ic`*`);` *`Ia`* `= W2B(`*`Ia`*`);` *`Ic`* `= W2B(`*`Ic`*`);` *`Ba`* `= B2W(`*`Ba`*`);` *`Bc`* `= B2W(`*`Bc`*`);` *`Ba`* `= W2B(`*`Ba`*`);` *`Bc`* `= W2B(`*`Bc`*`);`	

The following table provides the opcode field values (cond, compute) and the instruction syntax overview (Syntax)

cond	compute	Syntax
11111	00000000000000000000000	ACONV (Type 7d) ;

cond	compute	Syntax
11111	-----------------------	COMPUTE , ACONV (Type 7d) ;
-----	00000000000000000000000	IFCOND ACONV (Type 7d) ;
-----	-----------------------	IFCOND COMPUTE , ACONV (Type 7d) ;

Abstract

Switch address unit, optional compute, and optional condition.

SISD Mode

In SISD mode, the Type 7d instruction converts an address in an I register or B register between normal word and byte address spaces. The B2W instruction converts a byte address to a normal word and the W2B instruction converts a normal word address to a byte address. If the input value is already an address of the requested type it is copied to the result register unchanged. If a compute operation is specified, it is performed in parallel with the switch address unit. If a condition is specified, it affects the entire instruction.

SIMD Mode

In SIMD mode, the Type 7d instruction provides the same switch address as is available in SISD mode, but provides additional features for the optional compute operation. If a compute operation is specified, it is performed simultaneously on the X and Y processing elements in parallel with the transfer. If a condition is specified, it affects the entire instruction. The instruction is executed in a processing element if the specified condition tests true in that element independent of the condition result for the other element.

The index register modify operation, in SIMD mode, occurs based on the logical ORing of the outcome of the conditions tested on both PEs. In the second instruction, the index register modify also occurs based on the logical ORing of the outcomes of the conditions tested on both PEs. Because both threads of a SIMD sequence may be dependent on a single DAG index value, either thread needs to be able to cause a modify of the index.

Example

```
I0 = B2W(I2);

IF AV B4 = W2B(B1);

R1=R2-R3, I0 = B2W(I0);

IF AZ R1=R2-R3, I0 = B2W(I0);
```

Type7d Instruction Opcode

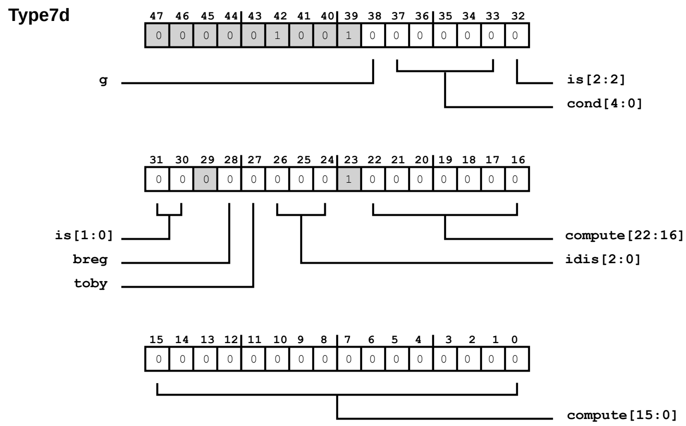

Figure 14-21: Type7d Instruction

ACONV (Type 7d)

ACONV Encode Table

breg	toby	g	Syntax
0	0	0	I1REG Register Class = b2w(I1REG Register Class)
0	0	1	I2REG Register Class = b2w(I2REG Register Class)
0	1	0	I1REG Register Class = w2b(I1REG Register Class)
0	1	1	I2REG Register Class = w2b(I2REG Register Class)
1	0	0	B1REG Register Class = b2w(B1REG Register Class)
1	0	1	B2REG Register Class = b2w(B2REG Register Class)
1	1	0	B1REG Register Class = w2b(B1REG Register Class)
1	1	1	B2REG Register Class = w2b(B2REG Register Class)

15 Group II Conditional Program Flow Control Instructions

The group II instructions contain data move operation and `COMPUTE/ELSE COMPUTE` operation.

The `COND` field selects whether the operation specified in the `COMPUTE` field and branch are executed. If the `COND` is true, the compute and branch are executed. If no condition is specified, `COND` is true condition, and the compute and branch are executed.

The `ELSE` field selects whether the condition is not true, in this case the computation is performed. The `ELSE` condition always requires an condition.

The `COMPUTE` field specifies a compute operation using the ALU, multiplier, or shifter. Because there are a large number of options available for computations, these operations are described separately in in the Computation Reference chapter.

Type	Addr	Option 1	Operation	Modifier 2	Option 3	SHARC 5 Stage Core
8a	ISA VISA	`IF` *cond*	`CALL` *<addr24>* `CALL (PC,`*<reladdr24>*`)`	`(DB);`		Yes
			`JUMP` *<addr24>* `JUMP (PC,`*<reladdr24>*`)`	`(DB)(LA)(CI)(DB,LA) (DB,CI);`		Yes
9a	ISA VISA	`IF` *cond*	`CALL (`*Md*`,` *Ic*`)` `CALL (PC,<reladdr6>)`	`(DB)`	`,ELSE` *compute;* `,` *compute;*	Yes
			`JUMP (`*Md*`,` *Ic*`)` `JUMP (PC,`*<reladdr6>*`)`	`(DB)(LA)(CI) (DB,LA)(DB,CI)`		Yes
9b	VISA	`IF` *cond*	`CALL (`*Md*`,`*Ic*`)` `CALL (PC,`*<reladdr6>*`)`	`(DB);`		Yes
			`JUMP (`*Md*`,`*Ic*`)` `JUMP (PC,`*<reladdr6>*`)`	`(DB)(LA)(CI)(DB,LA) (DB,CI);`		Yes
10a	ISA	`IF` *cond*	`JUMP (`*Md*`,`*Ic*`)`	`,ELSE` *compute;*		Yes

Type	Addr	Option 1	Operation	Modifier 2	Option 3	SHARC 5 Stage Core
			JUMP (PC,*<reladdr6>*)	,ELSE DM(*Ia*,*Mb*) = *Dreg*; ,ELSE *Dreg* =DM(*Ia*,Mb); ,ELSE *compute*, DM(*Ia*,*Mb*) = *Dreg*; ,ELSE *compute*, *Dreg* =DM(*Ia*,Mb);		
11a	ISA VISA	IF *cond*	RTS	(DB)(LR)(DB,LR)	,ELSE*compute*; ,*compute*;	Yes
			RTI	(DB)		Yes
11c	VISA	IF *cond*	RTS	(DB)(LR)(DB,LR);		Yes
			RTI	(DB);		Yes
12a (imm)	ISA VISA		LCNTR = *<data16>*, DO *<addr24>* UNTIL LCE; LCNTR =*<data16>*, DO (PC,*<reladdr24>*) UNTIL LCE;			Yes
12a (ureg)	ISA VISA		LCNTR = *Ureg*, DO *<addr24>* UNTIL LCE; LCNTR = *Ureg*, DO(PC,*<reladdr24>*) UNTIL LCE;			Yes
13a	ISA VISA		DO *<addr24>* UNTIL termination; DO (PC,*<reladdr24>*) UNTIL termination;			Yes

Type 8a ISA/VISA (cond + branch)

Syntax Summary

Type	Addr	Option 1	Operation	Option 2	Option 3	
8a	ISA VISA	IF *cond*	CALL *<addr24>* CALL (PC,*<reladdr24>*)	(DB);		Yes
			JUMP *<addr24>* JUMP (PC,*<reladdr24>*)	(DB)(LA)(CI)(DB,LA)(DB,CI);		Yes

The following table provides the opcode field values (cond) and the instruction syntax overview (Syntax)

cond	Syntax
11111	JUMP(Type 8a) ;
-----	IFCOND JUMP(Type 8a) ;

Abstract

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

Description

SISD Mode

In SISD mode, the Type 8 instruction provides a jump or call to the specified address or PC-relative address. The PC-relative address is a 24-bit, 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 programs reuse an interrupt while it is being serviced

Normally, the processors ignore and do not latch an interrupt that reoccurs while its service routine is already executing. Jump (CI) clears the status of the current interrupt without leaving the interrupt service routine, This feature reduces the interrupt routine to a normal subroutine and allows the interrupt to occur again, as a result of a -different event or task in the SHARC processor system. The jump (CI) instruction should be located within the interrupt service routine.

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

When returning from a reduced subroutine, programs must use the (LR) modifier of the RTS if the interrupt occurs during the last two instructions of a loop.

SIMD Mode

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

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

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

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

```
IF (PEx AND PEy COND) JUMP <addr24>  (options);
IF (PEx AND PEy COND) JUMP (PC, <reladdr24>) (options);
IF (PEx AND PEy COND) CALL <addr24> (options);
IF (PEx AND PEy COND) CALL (PC, <reladdr24>) (options);
```

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

```
/* No implicit PEy operation */
```

Example

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

When the processors are in SIMD mode, the first instruction performs a jump to the PC-relative address depending on the logical ANDing of the outcomes of the conditions tested in both PEs. In SIMD mode, the second and third instructions operate the same as in SISD mode. In the second instruction, a jump to the program label `init` occurs. A PC-relative jump takes place in the third instruction.

Type8a Instruction Syntax

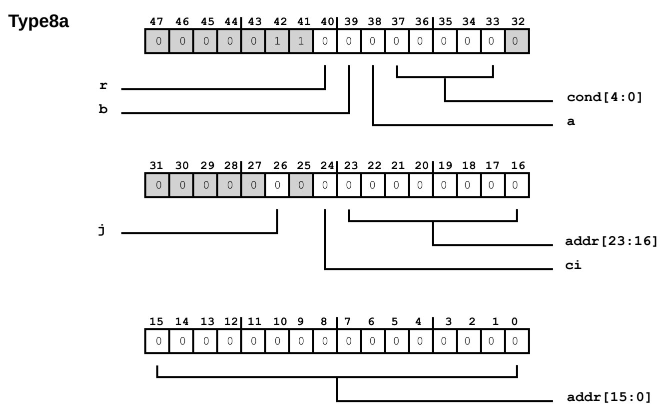

Figure 15-1: Type8a Instruction

ADDR (Type 8a)

ADDR Encode Table

r	Syntax
0	imm24 Register Type
1	(pc,imm24pc Register Type)

JUMP(Type 8a)

JUMP Encode Table

b	a	j	ci	Syntax
0	0	0	0	jump ADDR (Type 8a)

b	a	j	ci	Syntax
0	0	0	1	jump ADDR (Type 8a) (ci)
0	0	1	0	jump ADDR (Type 8a) (db)
0	0	1	1	jump ADDR (Type 8a) (db,ci)
0	1	0	0	jump ADDR (Type 8a) (la)
0	1	1	0	jump ADDR (Type 8a) (db,la)
1	0	0	0	call ADDR (Type 8a)
1	0	1	0	call ADDR (Type 8a) (db)

Type 9a ISA/VISA (cond + Branch + comp/else comp)

Syntax Summary

Type	Addr	Option 1	Operation	Modifier 2	Option 3	
9a	ISA VISA	IF *cond*	CALL (*Md*, *Ic*) CALL (PC,<reladdr6>)	(DB)	,ELSE *compute*; , *compute*;	Yes
			JUMP (*Md*, *Ic*) JUMP (PC,*<reladdr6>*)	(DB)(LA)(CI)(DB,LA) (DB,CI)		Yes

The following table provides the opcode field values (cond) and the instruction syntax overview (Syntax)

cond	Syntax
11111	JUMPCLAUSE (Type 9a) COMPUTECLAUSE (Type 9a) ;
-----	IFCOND JUMPCLAUSE (Type 9a) COMPUTECLAUSE (Type 9a) ;

Abstract

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

Description

SISD Mode

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

- `(DB)`—delayed branch—starts a delayed branch

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

Normally, the processor ignores and does not latch an interrupt that reoccurs while its service routine is already executing. Jump (CI) clears the status of the current interrupt without leaving the interrupt service routine. This feature reduces the interrupt routine to a normal subroutine and allows the interrupt to occur again, as a result of a different event or task in the system. The jump (CI) instruction should be located within the interrupt service routine.

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

When returning from a reduced subroutine, programs must use the (LR) modifier of the RTS instruction if the interrupt occurs during the last two instructions of a loop.

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

SIMD Mode

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

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

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

Note that for the compute, the X element uses the specified registers and the Y element uses the complementary registers.

The following 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) (options), (if PEx COND) compute ;
IF (PEx AND PEy COND) JUMP (PC, <reladdr6>) (options), (if PEx COND) compute ;
IF (PEx AND PEy COND) JUMP (Md, Ic) (options), ELSE (if NOT PEx) compute ;
IF (PEx AND PEy COND) JUMP (PC, <reladdr6>) (options), ELSE (if NOT PEx)
compute ;
```

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

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

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

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

Example

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

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

When the processors are in SIMD mode, the indirect jump in the first instruction occurs in parallel with both processing elements executing computations. In PEx, `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 ANDing 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.

Type9a Instruction Syntax

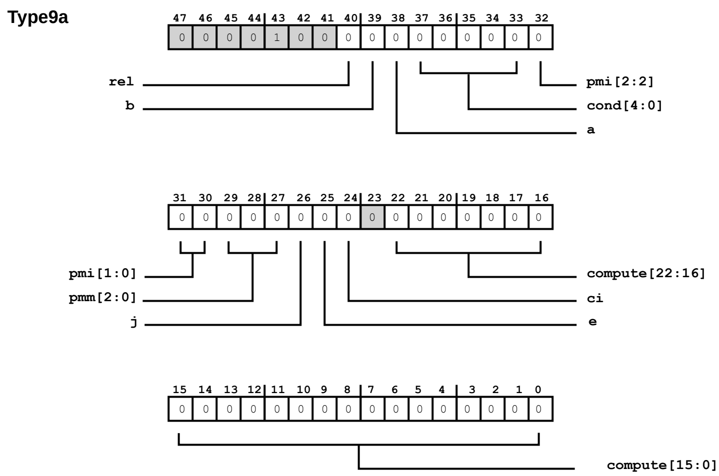

Figure 15-2: Type9a Instruction

ADDRCLAUSE (Type 9a)

ADDRCLAUSE Encode Table

rel	Syntax
0	(M2REG Register Class,I2REG Register Class)
1	(pc,imm6pc Register Type)

COMPUTECLAUSE (Type 9a)

COMPUTECLAUSE Encode Table

e	compute	Syntax
0	00000000000000000000000	
0	-----------------------	, COMPUTE
1	-----------------------	, else COMPUTE

JUMPCLAUSE (Type 9a)

JUMPCLAUSE Encode Table

b	a	j	ci	Syntax
0	0	0	0	jump ADDRCLAUSE (Type 9a)
0	0	0	1	jump ADDRCLAUSE (Type 9a) (ci)

b	a	j	ci	Syntax
0	0	1	0	jump ADDRCLAUSE (Type 9a) (db)
0	0	1	1	jump ADDRCLAUSE (Type 9a) (db,ci)
0	1	0	0	jump ADDRCLAUSE (Type 9a) (la)
0	1	1	0	jump ADDRCLAUSE (Type 9a) (db,la)
1	0	0	0	call ADDRCLAUSE (Type 9a)
1	0	1	0	call ADDRCLAUSE (Type 9a) (db)

Type 9b VISA (cond + Branch + comp/else)

Syntax Summary

Type	Addr	Option 1	Operation	Modifier 2	Option 3	
9b	VISA	`IF` *`cond`*	`CALL (`*`Md`*`,`*`Ic`*`)` `CALL (PC,<`*`reladdr6`*`>)`	`(DB);`		Yes
			`JUMP (`*`Md`*`,`*`Ic`*`)` `JUMP (PC,<`*`reladdr6`*`>)`	`(DB)(LA)(CI)(DB,LA)(DB,CI);`		Yes

The following table provides the opcode field values (cond) and the instruction syntax overview (Syntax)

cond	Syntax
11111	JUMPCLAUSE (Type 9b) ;
-----	IFCOND JUMPCLAUSE (Type 9b) ;

Abstract

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

Description

SISD Mode

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

- `(DB)`—delayed branch—starts a delayed branch
- `(LA)`—loop abort—causes the loop stacks and PC stack to be popped when the jump is executed. Use the (LA) modifier if the jump transfers program execution outside of a loop. Do not use `(LA)` if there is no loop or if the jump address is within the loop.

- (CI) —clear interrupt—lets programs reuse an interrupt while it is being serviced

Normally, the processor ignores and does not latch an interrupt that reoccurs while its service routine is already executing. Jump (CI) clears the status of the current interrupt without leaving the interrupt service routine. This feature reduces the interrupt routine to a normal subroutine and allows the interrupt to occur again, as a result of a different event or task in the system. The jump (CI) instruction should be located within the interrupt service routine.

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

When returning from a reduced subroutine, programs must use the (LR) modifier of the RTS instruction if the interrupt occurs during the last two instructions of a loop.

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

SIMD Mode

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

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

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

Note that for the compute, the X element uses the specified registers and the Y element uses the complementary registers.

The following 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) (options), (if PEx COND) compute ;
IF (PEx AND PEy COND) JUMP (PC, <reladdr6>) (options), (if PEx COND) compute ;
IF (PEx AND PEy COND) JUMP (Md, Ic) (options), ELSE (if NOT PEx) compute ;
IF (PEx AND PEy COND) JUMP (PC, <reladdr6>) (options), ELSE (if NOT PEx)
compute ;

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

```
IF (PEx AND PEy COND) CALL (PC, <reladdr6>) (options), ELSE (if NOT PEx)
compute;
```

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

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

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

Example

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

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

When the processors are in SIMD mode, the indirect jump in the first instruction occurs in parallel with both processing elements executing computations. In PEx, `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 ANDing 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.

Type9b Instruction Syntax

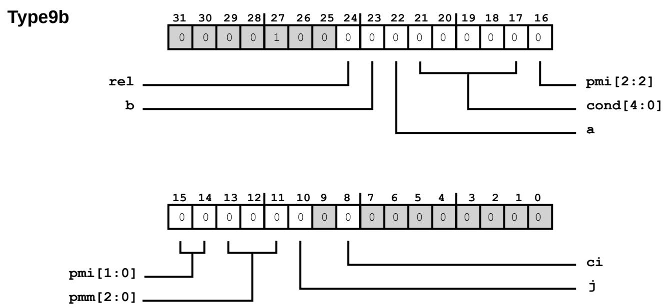

Figure 15-3: Type9b Instruction

ADDRCLAUSE (Type 9b)

ADDRCLAUSE Encode Table

rel	Syntax
0	(M2REG Register Class,I2REG Register Class)
1	(pc,imm6visapc Register Type)

JUMPCLAUSE (Type 9b)

JUMPCLAUSE Encode Table

b	a	j	ci	Syntax
0	0	0	0	jump ADDRCLAUSE (Type 9b)
0	0	0	1	jump ADDRCLAUSE (Type 9b) (ci)
0	0	1	0	jump ADDRCLAUSE (Type 9b) (db)
0	0	1	1	jump ADDRCLAUSE (Type 9b) (db,ci)
0	1	0	0	jump ADDRCLAUSE (Type 9b) (la)
0	1	1	0	jump ADDRCLAUSE (Type 9b) (db,la)
1	0	0	0	call ADDRCLAUSE (Type 9b)
1	0	1	0	call ADDRCLAUSE (Type 9b) (db)

Type 10a ISA (cond + branch + else comp + mem data move

Syntax Summary

Type	Addr	Option 1	Operation	Option 2	Option 3
10a	ISA	IF *cond*	JUMP (*Md*,*Ic*) JUMP (PC,<*reladdr6*>)	,ELSE *compute*; ,ELSE DM(*Ia*,*Mb*) = *Dreg*; ,ELSE *Dreg* =DM(*Ia*,Mb); ,ELSE *compute*, DM(*Ia*,*Mb*) = *Dreg*; ,ELSE *compute*, *Dreg* =DM(*Ia*,Mb);	Yes

The following table provides the opcode field values (cond, compute) and the instruction syntax overview (Syntax)

cond	compute	Syntax
11111	00000000000000000000000	jump ADDRCLAUSE (Type 10a) , else ACCESS (Type 10a) ;
11111	-----------------------	jump ADDRCLAUSE (Type 10a) , else COMPUTE , ACCESS (Type 10a) ;
-----	00000000000000000000000	IFCOND jump ADDRCLAUSE (Type 10a) , else ACCESS (Type 10a) ;
-----	-----------------------	IFCOND jump ADDRCLAUSE (Type 10a) , else COMPUTE , ACCESS (Type 10a) ;

Abstract

Indirect (or PC-relative) jump or optional compute operation with transfer between data memory and register file. This instruction is not supported in VISA address space.

Description

SISD Mode

In SISD mode, the Type 10a 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, programs may not use the delay branch (DB), loop abort (LA), or clear interrupt (CI) modifiers.

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

SIMD Mode

In SIMD mode, the Type 10a 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.

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 10a instruction's explicit and implicit operations in SIMD mode.

Broadcast Mode

If the broadcast read bits—`BDCST1` (for `I1`) or `BDCST9` (for `I9`)—are set, the Y element uses the specified I register without adding one.

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

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

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


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

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

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

```
IF (PEx AND PEy COND) Jump (Md, Ic) , Else (if NOT PEy) compute, DM(Ia + k,
Mb) = dreg ;
```

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

```
IF (PEx AND PEy COND) Jump (Md, Ic) , Else (if NOT PEy) compute, dreg = DM(Ia
+ k, Mb) ;
```

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

If Broadcast Load Mode memory read k=0. If SIMD mode NW access k=1, SW access k=2, BW access k=4.

Example

```
IF TF JUMP(M8, I8),  ELSE R6=DM(I6, M1);
```

```
IF NE JUMP(PC, 0x20),  ELSE F12=FLOAT R10 BY R3, R6=DM(I5, M0);
```

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

When the processors are in SIMD mode, the indirect jump in the first instruction executes depending on the outcome of the conditional in both processing element. The condition is evaluated independently on each processing element, PEx and PEy. The indirect jump executes based on the logical ANDing of the PEx and PEy conditional tests. So, the indirect jump executes if the condition tests true in both PEx and PEy. The data memory read is performed independently on either PEx or PEy based on the negative evaluation of the condition code seen by that PE.

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

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

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

Type10a Instruction Syntax

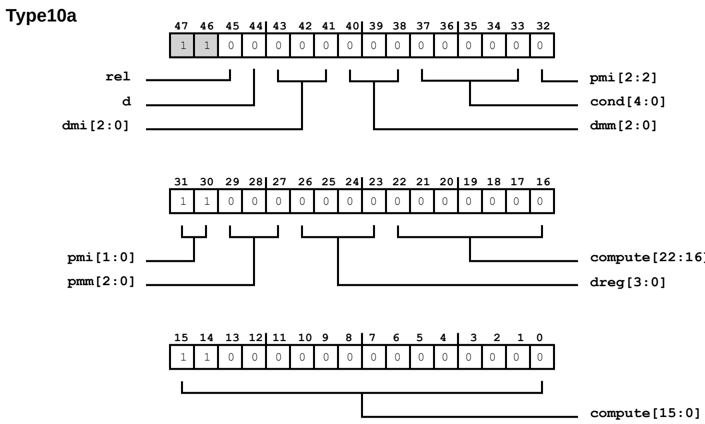

Figure 15-4: Type10a Instruction

ACCESS (Type 10a)

ACCESS Encode Table

d	Syntax
0	RFREG Register Class = dm(I1REG Register Class, M1REG Register Class)
1	dm(I1REG Register Class, M1REG Register Class) = RFREG Register Class

ADDRCLAUSE (Type 10a)

ADDRCLAUSE Encode Table

rel	Syntax
0	(M2REG Register Class,I2REG Register Class)
1	(pc,imm6pc Register Type)

Type 11a ISA/VISA (cond + branch return + comp/else comp)

Syntax Summary

Type	Addr	Option 1	Operation	Modifier 2	Option 3	
11a	ISA VISA	`IF` *`cond`*	`RTS`	`(DB) (LR) (DB,LR)`	`,ELSE`*`compute;`* `,`*`compute;`*	Yes
			`RTI`	`(DB)`		Yes

The following table provides the opcode field values (cond) and the instruction syntax overview (Syntax)

cond	Syntax
11111	RETURN (Type 11a) COMPUTECLAUSE (Type 11a) ;
-----	IFCOND RETURN (Type 11a) COMPUTECLAUSE (Type 11a) ;

Abstract

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

Description

SISD Mode

In SISD mode, the Type 11 instruction provides a return from a subroutine (RTS) or return from an interrupt service routine (RTI). A return causes the processor to branch to the address stored at the top of the PC 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 or the timer interrupt
- Clears the appropriate bit in the interrupt latch register (IRPTL) and the interrupt mask pointer (IMASKP)

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

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

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

Programs must also use the (LR) modifier for RTS when returning from a subroutine that has been reduced from an interrupt service routine with a jump (CI) instruction. This case occurs when the interrupt occurs during the last two instructions of a loop.

SIMD Mode

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

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

Note that for the `compute`, the X element uses the specified registers, and the Y element uses the complementary registers.

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 (options), (if PEx COND) compute;
IF (PEx AND PEy COND) RTS (options), ELSE (if NOT PEx) compute;

IF (PEx AND PEy COND) RTI (options), (if PEx COND) compute;
IF (PEx AND PEy COND) RTI (options), 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;
IF (PEx AND PEy COND) RTS (DB), ELSE (if NOT PEy) compute;

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

Example

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

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

When the processors are in SIMD mode, the first instruction performs a return from interrupt and both processing elements execute the computation in parallel. The result from PEx is placed in `R6`, and the result from PEy is placed in `S6`. The second instruction performs a return from subroutine (RTS) if the condition tests true in both PEx or PEy. In the third instruction, the condition is evaluated independently on each processing element, PEx and PEy.

The RTS executes based on the logical ANDing of the PEx and PEy conditional tests. So, the RTS executes if the condition tests true in both PEx and PEy. Because the ELSE inverts the conditional test, the computation is performed independently on either PEx or PEy based on the negative evaluation of the condition code seen by that processing element. The R0 register stores the result in PEx, and S0 stores the result in PEy if the computations are executed.

Type11a Instruction Syntax

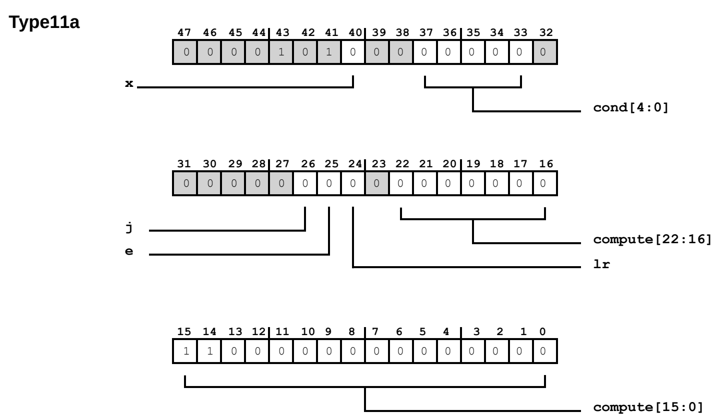

Figure 15-5: Type11a Instruction

COMPUTECLAUSE (Type 11a)

COMPUTECLAUSE Encode Table

e	compute	Syntax
0	00000000000000000000000	
0	-----------------------	, COMPUTE
1	-----------------------	, else COMPUTE

RETURN (Type 11a)

RETURN Encode Table

x	j	lr	Syntax
0	0	0	rts
0	0	1	rts (lr)
0	1	0	rts (db)
0	1	1	rts (db,lr)
1	0	0	rti

x	j	lr	Syntax
1	1	0	rti (db)

Type 11c VISA (cond + branch return)

Syntax Summary

Type	Addr	Option 1	Operation	Option 2	Option 3	
11c	VISA	`IF cond`	`RTS`	`(DB)(LR)(DB,LR);`		Yes
			`RTI`	`(DB);`		Yes

The following table provides the opcode field values (cond) and the instruction syntax overview (Syntax)

cond	Syntax
11111	RETURN (Type 11c) ;
-----	IFCOND RETURN (Type 11c) ;

Abstract

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

Description

SISD Mode

In SISD mode, the Type 11 instruction provides a return from a subroutine (RTS) or return from an interrupt service routine (RTI). A return causes the processor to branch to the address stored at the top of the PC 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` or the timer interrupt
- Clears the appropriate bit in the interrupt latch register (`IRPTL`) and the interrupt mask pointer (`IMASKP`)

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

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

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

Programs must also use the (LR) modifier for RTS when returning from a subroutine that has been reduced from an interrupt service routine with a jump (CI) instruction. This case occurs when the interrupt occurs during the last two instructions of a loop.

SIMD Mode

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

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

Note that for the `compute`, the X element uses the specified registers, and the Y element uses the complementary registers.

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 (options), (if PEx COND) compute;
IF (PEx AND PEy COND) RTS (options), ELSE (if NOT PEx) compute;

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

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

```
IF (PEx AND PEy COND) RTS (options), (if PEy COND) compute;
IF (PEx AND PEy COND) RTS (options), ELSE (if NOT PEy) compute;

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

Example

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

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

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

Type11c Instruction Syntax

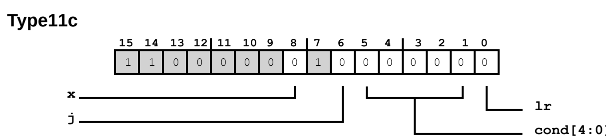

Figure 15-6: Type11c Instruction

RETURN (Type 11c)

RETURN Encode Table

x	j	lr	Syntax
0	0	0	rts
0	0	1	rts (lr)
0	1	0	rts (db)
0	1	1	rts (db,lr)
1	0	0	rti
1	1	0	rti (db)

Type 12a ISA/VISA (do until imm loop counter expired)

Syntax Summary

Table 15-1: Group I Instructions by Instruction Type

Type	Addr	Option1	Operation	Option2	Option3
12a (imm)	ISA VISA		`LCNTR = <data16>, DO <addr24> UNTIL LCE;` `LCNTR =<data16>, DO (PC,<reladdr24>) UNTIL LCE`		

The following table provides the opcode field values (mode) and the instruction syntax overview (Syntax)

mode	Syntax
0	lcntr = uimm16 Register Type, do (pc,imm23pc Register Type) until lce ;
1	lcntr = uimm16 Register Type, do (pc,imm23pc Register Type) until lce (f) ;

Abstract

Load loop counter, do loop until loop counter expired

Description

SISD and SIMD Modes

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 two's-complement address. The LCNTR is pushed on the loop counter stack and becomes the CURLCNTR value. The loop executes until the CURLCNTR reaches zero.

The Mode bit (bit 23) configures if this is an E2 active loop (=0) or a F1 active loop (=1)

Example

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

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

Type12a_imm Instruction Syntax

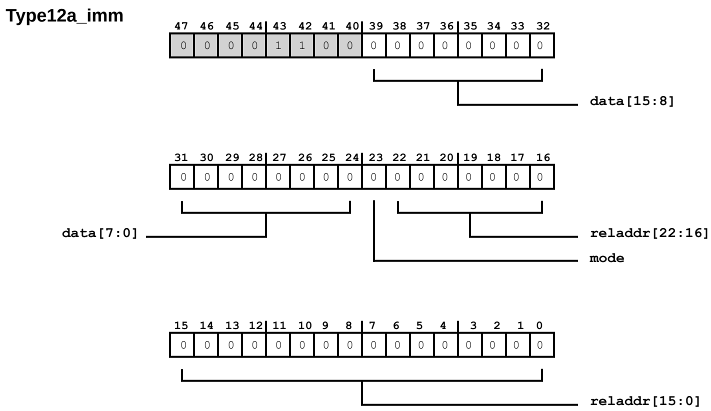

Figure 15-7: Type12a_imm Instruction

Type 12a ISA/VISA (do until ureg loop counter expired)

Syntax Summary

Table 15-2: Group I Instructions by Instruction Type

Type	Addr	Option1	Operation	Option2	Option3
12a (imm)	ISA VISA		`LCNTR = Ureg, DO <addr24> UNTIL LCE;` `LCNTR = Ureg, DO(PC,<reladdr24>) UNTIL LCE;`		

The following table provides the opcode field values (mode) and the instruction syntax overview (Syntax)

mode	Syntax
0	lcntr = UREG Registers Class, do (pc,imm23pc Register Type) until lce ;
1	lcntr = UREG Registers Class, do (pc,imm23pc Register Type) until lce (f) ;

Abstract

Load loop counter, do loop until loop counter expired

Description

SISD and SIMD Modes

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 two's-complement address. The LCNTR is pushed on the loop counter stack and becomes the CURLCNTR value. The loop executes until the CURLCNTR reaches zero.

The Mode bit (bit 23) configures if this is an E2 active loop (=0) or a F1 active loop (=1)

Example

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

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

Type12a_ureg Instruction Syntax

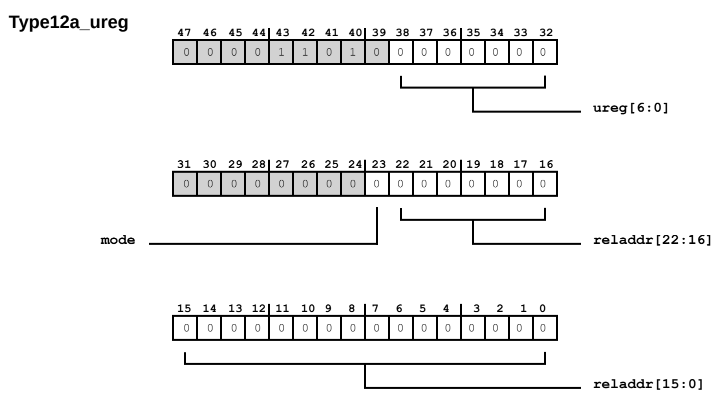

Figure 15-8: Type12a_ureg Instruction

Type 13a ISA/VISA (do until termination)

Syntax Summary

Type	Addr	Option 1	Operation	Option 2	Option 3	
13a	ISA VISA		`DO <addr24> UNTIL termination;` `DO (PC,<reladdr24>) UNTIL termination;`			Yes

The following table provides the opcode field values (mode) and the instruction syntax overview (Syntax)

mode	Syntax
0	do (pc,imm23pc Register Type) until TERM (Type 13a) ;
1	do (pc,imm23pc Register Type) until TERM (Type 13a) (f) ;

Abstract

Do until termination

Description

SISD Mode

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

SIMD Mode

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

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

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

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

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

```
/* No implicit PEy operation */
```

Example

placeholder

Type13a Instruction Syntax

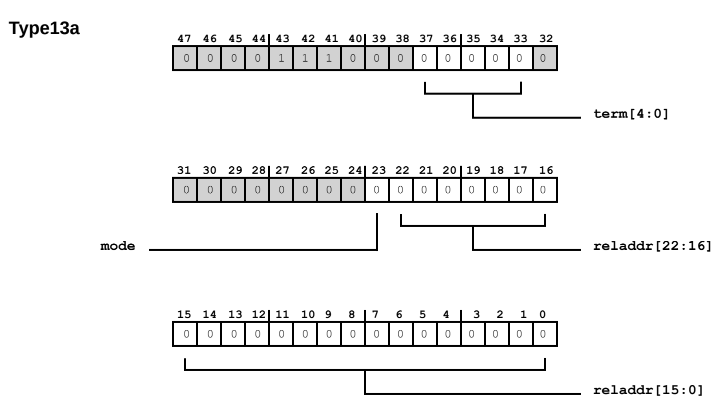

Figure 15-9: Type13a Instruction

TERM (Type 13a)

TERM Encode Table

term	Syntax
00000	eq
00001	lt
00010	le
00011	ac
00100	av
00101	mv
00110	ms
00111	sv
01000	sz
01001	flag0_in
01010	flag1_in
01011	flag2_in
01100	flag3_in
01101	tf
01110	bm
01110	sf
01111	lce
10000	ne
10001	ge
10010	gt
10011	not ac
10100	not av
10101	not mv
10110	not ms
10111	not sv
11000	not sz
11001	not flag0_in
11010	not flag1_in
11011	not flag2_in
11100	not flag3_in

term	Syntax
11101	not tf
11110	not bm
11110	not sf
11111	forever

16 Group III Immediate Data Move Instructions

The group III instructions contain data move operation with immediate data or indirect addressing.

Type	Addr	Operation	Modifier	SHARC 5 Stage Core
14a	ISA VISA	DM(<addr32>) = Ureg PM(<addr32>) = Ureg Ureg = DM(<addr32>) Ureg = PM(<addr32>)	(lw);	Yes
14d	ISA	DM(<addr32>) = Dreg	(lw/sw/bw/ex);	No
		Dreg = DM(<addr32>)	(lw/sw/bw/ex); (swse/bwse/ex);	No
15a	ISA VISA	DM(<data32>,Ia) = Ureg PM(<data32>,Ic) = Ureg Ureg = DM(<data32>,Ia) Ureg = PM(<data32>,Ic)	(lw);	Yes
15b	VISA	DM(<data7>,Ia) = Ureg PM(<data7>,Ic) = Ureg Ureg = DM(<data7>,Ia) Ureg = PM(<data7>,Ic)	(lw);	Yes
16a	ISA VISA	DM(Ia,Mb) = <data32>; PM(Ic,Md) = <data32>;		Yes
16b	VISA	DM(Ia,Mb) = <data16>; PM(Ic,Md) = <data16>;		Yes
17a	ISA VISA	Ureg = <data32>;		Yes
17b	VISA	Ureg = <data16>;		Yes

Type 14a ISA/VISA (mem data move)

Syntax Summary

Type	Addr	Operation	Modifier	SHARC 5 Stage Core

14a	ISA
	VISA
		`DM(<addr32>) = Ureg`
		`PM(<addr32>) = Ureg`
		`Ureg = DM(<addr32>)`
		`Ureg = PM(<addr32>)`
			`(lw);`	Yes

The following table provides the opcode field values (g, d, l) and the instruction syntax overview (Syntax)

g	d	l	Syntax
0	0	0	UREG Registers Class = dm(imm32 Register Type);
0	1	0	dm(imm32 Register Type) = UREG Registers Class;
1	0	0	UREG Registers Class = pm(imm32 Register Type);
1	1	0	pm(imm32 Register Type) = UREG Registers Class;
0	0	1	UREG Registers Class = dm(imm32 Register Type) (lw);
0	1	1	dm(imm32 Register Type) = UREG Registers Class (lw);
1	0	1	UREG Registers Class = pm(imm32 Register Type) (lw);
1	1	1	pm(imm32 Register Type) = UREG Registers Class (lw);

Abstract

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

Description

SISD Mode

In SISD mode, the Type 14 instruction sets up an access between data or program memory and a universal register, with direct addressing. The entire data or program memory address is specified in the instruction. The optional (LW) in this syntax lets programs specify long word addressing, overriding default addressing from the memory map.

SIMD Mode

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

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

For the universal register, the X element uses the specified *Ureg*, and the Y element uses the complementary register (*Cureg*) that corresponds to the *Ureg* register specified in the instruction. Note that only the *Cureg* subset registers which have complementary registers are effected by SIMD mode.

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

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

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

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

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

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

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

Note that if the instruction type uses optional forced long word modifier (LW) SIMD mode is overwritten and register pair access is performed.

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

```
DM(<addr32>) = ureg0/1 (LW);

PM(<addr32>) = ureg0/1 (LW);

ureg0/1 = DM(<addr32>) (LW);

ureg0/1 = PM(<addr32>) (LW);
```

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

no operations

Example

```
DM(temp)=MODE1; /* temp is a program label */
LCNTR=PM(0x90500);
```

When the processors are in SISD mode, the first instruction performs a direct memory write of the value in the `MODE1` register into data memory with the data memory destination address specified by the program label, `temp`. The second instruction initializes the `LCNTR` register with the value found in the specified address in program memory.

Because of the register selections in this example, these two instructions operate the same in SIMD and SISD mode. The `MODE1` (`SREG`) and `LCNTR` (`UREG`) registers have no complements, so they do not operate differently in SIMD mode.

Type14a Instruction Opcode

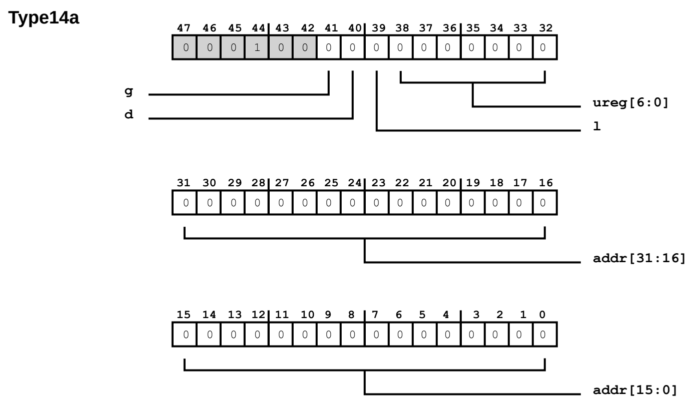

Figure 16-1: Type14a Instruction

Type 14d ISA/VISA (exclusive mem data move)

Syntax Summary

NOTE: The 48-bit 14d instruction type is an extension (exclusive access) to 14a instruction.

Type	Addr	Operation	Option
14d	ISA VISA	`DM(<addr32>) = Dreg`	`(lw/sw/bw/ex);`
		`Dreg = DM(<addr32>)`	`(lw/sw/bw/ex);` `(swse/bwse/ex);`

The following table provides the opcode field values (w, ex, d, l) and the instruction syntax overview (Syntax)

w	ex	d	l	Syntax
1	1	0	0	RFREG Register Class = dm(imm32 Register Type) EX (Type 14d);
1	1	1	0	dm(imm32 Register Type) = RFREG Register Class EX (Type 14d);
0	1	0	-	RFREG Register Class = dm(imm32 Register Type) BHSEEX (Type 14d);
0	1	1	-	dm(imm32 Register Type) = RFREG Register Class BHEX (Type 14d);
0	0	0	-	RFREG Register Class = dm(imm32 Register Type) BHSE (Type 14d);
0	0	1	-	dm(imm32 Register Type) = RFREG Register Class BH (Type 14d);
1	1	0	1	RFREG Register Class = dm(imm32 Register Type) LWEX (Type 14d);

w	ex	d	l	Syntax
1	1	1	1	dm(imm32 Register Type) = RFREG Register Class LWEX (Type 14d);

Abstract

Transfer between data or program memory and register file, direct addressing, immediate address with options for byte address sub-word access and exclusive access

Description

The type 14d instruction provides additional options for the Type 14 direct address instruction. The access is however restricted to the R register file, or in SIMD mode R and S registers, rather than the entire universal register set.

The optional `(BW)`, `(BWSE)`, `(SW)`, and `(SWSE)`, may only be used when the I-register addresses byte space. (BW) specifies a byte access; the 8-bit value loaded into a register is zero extended to 32-bits and the value stored is the low order 8-bits of the 32-bit value in the register. `(SW)` specifies a short word access; the 16-bit value loaded into a register is zero extended to 32-bits and the value stored is the low order 16-bits of the 32-bit value in the register. (BWSE) and `(SWSE)` may only be used on loads and specify the 8-bit value is sign extended or 16-bit value is sign extended respectively. These options may be used in SISD and SIMD mode.

The `(EX)` specifies an exclusive access. This option may be combined with `(LW)`, `(BW)`, `(BWSE)`, `(SW)`, `(SWSE)` which is written `(LW, EX)` etc., or used alone to specify a normal word exclusive access.

Example

```
R2 = DM(0x12456) (BW);

PM(symbolic_addr) = R12 (EX);
```

Type14d Instruction Opcode

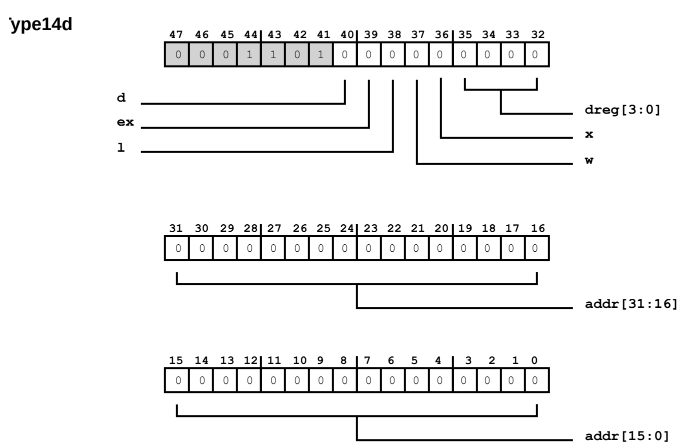

Figure 16-2: Type14d Instruction

BH (Type 14d)

BH Encode Table

l	Syntax
0	(bw)
1	(sw)

BHEX (Type 14d)

BHEX Encode Table

l	Syntax
0	(bw,ex)
1	(sw,ex)

BHSE (Type 14d)

BHSE Encode Table

l	x	Syntax
0	0	(bw)
1	0	(sw)
0	1	(bwse)
1	1	(swse)

BHSEEX (Type 14d)

BHSEEX Encode Table

l	x	Syntax
0	0	(bw,ex)
1	0	(sw,ex)
0	1	(bwse,ex)
1	1	(swse,ex)

EX (Type 14d)

EX Encode Table

Syntax
(ex)

LWEX (Type 14d)

LWEX Encode Table

Syntax
(lw,ex)

Type 15a ISA/VISA (<data32> move)

Syntax Summary

Type	Addr	Operation	Option
15a	ISA VISA	DM(<data32>,Ia) = Ureg PM(<data32>,Ic) = Ureg Ureg = DM(<data32>,Ia) Ureg = PM(<data32>,Ic)	(lw);

The following table provides the opcode field values (g, d, l) and the instruction syntax overview (Syntax)

g	d	l	Syntax
0	0	0	UREG Registers Class = dm(imm32 Register Type, I1REG Register Class);
0	1	0	dm(imm32 Register Type, I1REG Register Class) = UREGXDAG1 Register Class;
1	0	0	UREG Registers Class = pm(imm32 Register Type, I2REG Register Class);
1	1	0	pm(imm32 Register Type, I2REG Register Class) = UREGXDAG2 Register Class;
0	0	1	UREG Registers Class = dm(imm32 Register Type, I1REG Register Class) (lw);
0	1	1	dm(imm32 Register Type, I1REG Register Class) = UREGXDAG1 Register Class (lw);
1	0	1	UREG Registers Class = pm(imm32 Register Type, I2REG Register Class) (lw);
1	1	1	pm(imm32 Register Type, I2REG Register Class) = UREGXDAG2 Register Class (lw);

Abstract

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

Description

SISD Mode

In SISD mode, the Type 15 instruction sets up an access between data or program memory and a universal register, with indirect addressing using I registers. The I register is pre-modified with an immediate value specified in the instruction. The I register is not updated. The *Ureg* may not be from the same DAG (that is, DAG1 or DAG2) as `Ia/Mb` or `Ic/Md`. The optional (LW) in this syntax lets programs specify long word addressing, overriding default addressing from the memory map.

SIMD Mode

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

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

The *Ureg* specified in the instruction is used for the X processing element transfer and may not be from the same DAG (that is, DAG1 or DAG2) as `Ia/Mb` or `Ic/Md`. The Y element uses the complementary register (*Cureg*) that correspond to the *Ureg* register specified in the instruction. Note that only the *Cureg* subset registers which have complimentary registers are effected by SIMD mode.

The following 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 ;
PM(<data32>, Ic) = ureg ;

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

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

```
DM(<data32>+k, Ia) = cureg ;
PM(<data32>+k, Ic) = cureg ;

cureg = DM(<data32>+k, Ia) ;
cureg = PM(<data32>+k, Ic) ;
```

Note that if the instruction type uses optional forced long word modifier (LW) SIMD mode is overwritten and register pair access is done

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

```
DM(<data32>, Ia) = ureg0/1 (LW);
PM(<data32>, Ic) = ureg0/1 (LW);
ureg0/1 = DM(<data32>, Ia) (LW);
ureg0/1 = PM(<data32>, Ic) (LW);
```

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

no instructions

If Broadcast Load Mode memory read k=0. If SIMD mode NW access k=1, SW access k=2, BW access k=4.

Example

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

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

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

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

Type15a Instruction Opcode

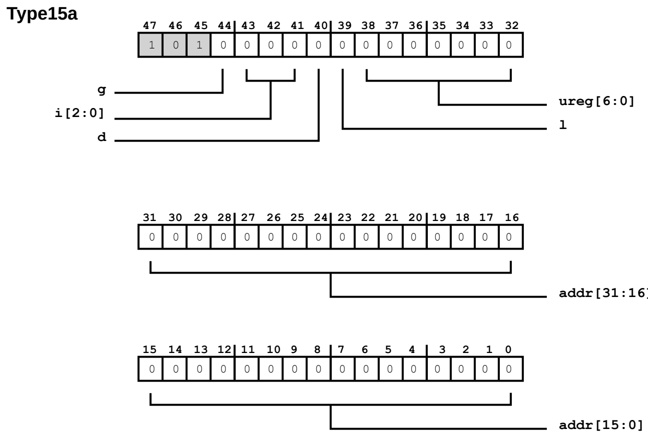

Figure 16-3: Type15a Instruction

Type 15b VISA (<data7> move)

Syntax Summary

Type	Addr	Operation	Option	
15b	VISA	DM(*<data7>*,*Ia*) = *Ureg* PM(*<data7>*,*Ic*) = *Ureg* *Ureg* = DM(*<data7>*,*Ia*) *Ureg* = PM(*<data7>*,*Ic*)	(lw);	Yes

The following table provides the opcode field values (g, d, l) and the instruction syntax overview (Syntax)

g	d	l	Syntax
0	0	0	UREG Registers Class = dm(imm7visa Register Type, I1REG Register Class);
0	1	0	dm(imm7visa Register Type, I1REG Register Class) = UREGXDAG1 Register Class;
1	0	0	UREG Registers Class = pm(imm7visa Register Type, I2REG Register Class);
1	1	0	pm(imm7visa Register Type, I2REG Register Class) = UREGXDAG2 Register Class;
0	0	1	UREG Registers Class = dm(imm7visa Register Type, I1REG Register Class) (lw);
0	1	1	dm(imm7visa Register Type, I1REG Register Class) = UREGXDAG1 Register Class (lw);
1	0	1	UREG Registers Class = pm(imm7visa Register Type, I2REG Register Class) (lw);
1	1	1	pm(imm7visa Register Type, I2REG Register Class) = UREGXDAG2 Register Class (lw);

Abstract

Transfer (7-bit data) between data or program memory and universal register, indirect addressing, immediate modifier

Description

SISD Mode

In SISD mode, the Type 15 instruction sets up an access between data or program memory and a universal register, with indirect addressing using I registers. The I register is pre-modified with an immediate value specified in the instruction. The I register is not updated. The *Ureg* may not be from the same DAG (that is, DAG1 or DAG2) as `Ia/Mb` or `Ic/Md`. The optional (LW) in this syntax lets programs specify long word addressing, overriding default addressing from the memory map.

SIMD Mode

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

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

The *Ureg* specified in the instruction is used for the X processing element transfer and may not be from the same DAG (that is, DAG1 or DAG2) as `Ia/Mb` or `Ic/Md`. The Y element uses the complementary register (*Cureg*) that correspond to the *Ureg* register specified in the instruction. Note that only the *Cureg* subset registers which have complimentary registers are effected by SIMD mode.

The following 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(<data7>, Ia) = ureg (LW);
PM(<data7>, Ic) = ureg (LW);

ureg = DM(<data7>, Ia) (LW);
ureg = PM(<data7>, Ic) (LW);
```

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

```
DM(<data7>+k, Ia) = cureg (LW);
PM(<data7>+k, Ic) = cureg (LW);

cureg = DM(<data7>+k, Ia) (LW);
cureg = PM(<data7>+k, Ic) (LW);
```

Note that if the instruction type uses optional forced long word modifier (LW) SIMD mode is overwritten and register pair access is done

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

```
DM(<data7>, Ia) = ureg0/1 (LW);
PM(<data7>, Ic) = ureg0/1 (LW);
ureg0/1 = DM(<data7>, Ia) (LW);
ureg0/1 = PM(<data7>, Ic) (LW);
```

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

no instructions

If Broadcast Load Mode memory read k=0. If SIMD mode NW access k=1, SW access k=2, BW access k=4.

Example

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

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

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

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

Type15b Instruction Opcode

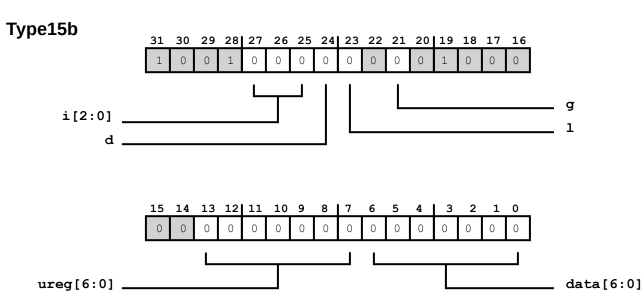

Figure 16-4: Type15b Instruction

Type 16a ISA/VISA (<data32> move)

Syntax Summary

Type	Addr	Operation	Option	
16a	ISA VISA	`DM(Ia,Mb) = <data32>;` `PM(Ic,Md) = <data32>;`		Yes

The following table provides the opcode field values (g) and the instruction syntax overview (Syntax)

g	Syntax
0	dm(I1REG Register Class, M1REG Register Class) = imm32f Register Type;
1	pm(I2REG Register Class, M2REG Register Class) = imm32f Register Type;

Abstract

Immediate 32-bit data write to data or program memory

Description

SISD Mode

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

SIMD Mode

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

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

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

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

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

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

```
DM(Ia+k, 0) = <data32> ;
```

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

If Broadcast Load Mode memory read k=0. If SIMD mode NW access k=1, SW access k=2, BW access k=4.

Example

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

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

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

Type16a Instruction Opcode

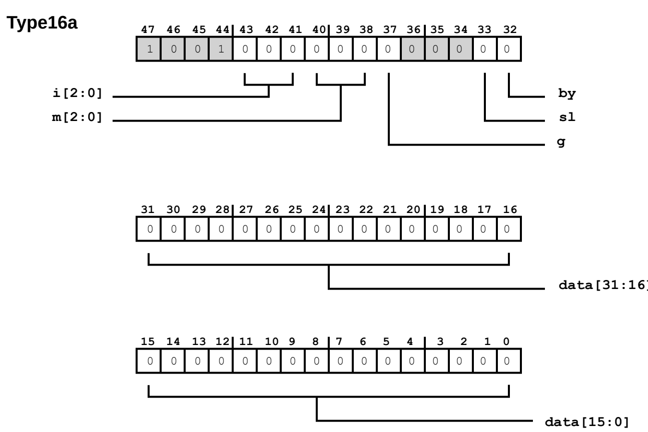

Figure 16-5: Type16a Instruction

Type 16b VISA (<data16> move)

Syntax Summary

Type	Addr	Operation	Option	
16b	VISA	`DM(Ia,Mb) = <data16>;` `PM(Ic,Md) = <data16>;`		Yes

The following table provides the opcode field values (g) and the instruction syntax overview (Syntax)

g	Syntax
0	dm(I1REG Register Class, M1REG Register Class) = imm16visa Register Type;
1	pm(I2REG Register Class, M2REG Register Class) = imm16visa Register Type;

Abstract

Immediate 16-bit data write to data or program memory

Description

SISD Mode

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

SIMD Mode

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

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

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

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

```
DM(Ia, Mb) = <data16> ;
PM(Ic, Md) = <data16> ;
```

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

```
DM(Ia+k, 0) = <data16> ;
PM(Ic+k, 0) = <data16> ;
```

If Broadcast Load Mode memory read k=0. If SIMD mode NW access k=1, SW access k=2, BW access k=4.

Type16b Instruction Opcode

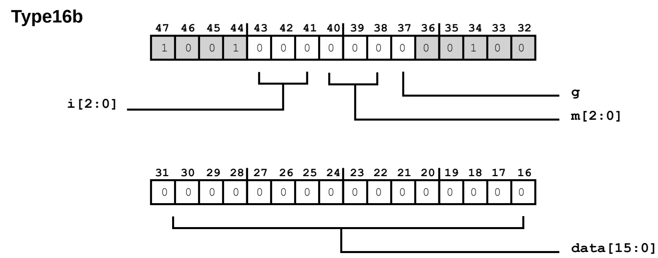

Figure 16-6: Type16b Instruction

Type 17a ISA/VISA (<data32> move)

Syntax Summary

Type	Addr	Operation	Option	
17a	ISA VISA	*Ureg = <data32>;*		Yes

The following table provides the instruction syntax overview (Syntax)

Syntax
UREG Registers Class = imm32f Register Type ;

Abstract

Immediate 32-bit data write to universal register

Description

SISD Mode

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.

SIMD Mode

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

The X element uses the specified *Ureg*, and the Y element uses the complementary *Cureg*. Note that only the *Cureg* subset registers which have complimentary registers are effected by SIMD mode.

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

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

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

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

```
cureg = <data32> ;
```

Example

```
ASTATx=0x0;
M15=mod1;   /* mod1 is user-defined constant */
```

When the processors are in SISD mode, the two instructions load immediate values into the specified registers.

Because of the register selections in this example, the second instruction in this example operates the same in SIMD and SISD mode. The `ASTATx` (system) register is included in the *`Cureg`* subset. In the first instruction, the immediate data write to the system register `ASTATx` and its complimentary register `ASTATy` are performed in parallel on PEx and PEy respectively. In the second instruction, the `M15` register is not included in the *`Cureg`* subset. So, the second instruction operates the same in SIMD and SISD mode.

Type17a Instruction Opcode

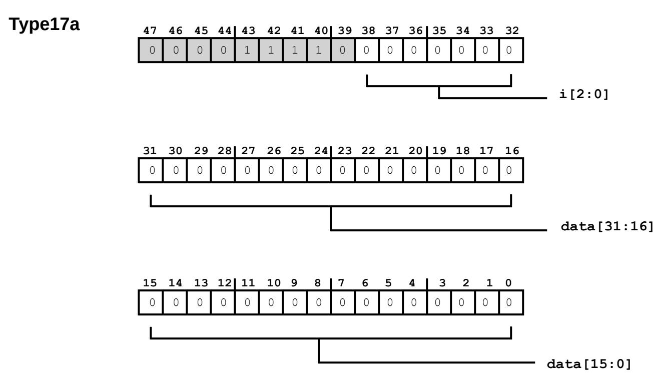

Figure 16-7: Type17a Instruction

Type 17b VISA (<data16> move)

Syntax Summary

Type	Addr	Operation	Option	
17b	VISA	*`Ureg = <data16>;`*		Yes

The following table provides the instruction syntax overview (Syntax)

Syntax
UREG Registers Class = imm16visa Register Type ;

Abstract

Immediate 16-bit data write to universal register

Description

SISD Mode

In SISD mode, the Type 17 instruction writes 16-bit immediate data to a universal register. If the register is 40 bits wide, the data is placed in the most significant 32 bits, and the least significant 8 bits are loaded with 0s.

SIMD Mode

In SIMD mode, the Type 17 instruction provides the same write of 16-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.

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

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

```
ureg = <data16> ;
```

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

```
cureg = <data16> ;
```

Type17b Instruction Syntax

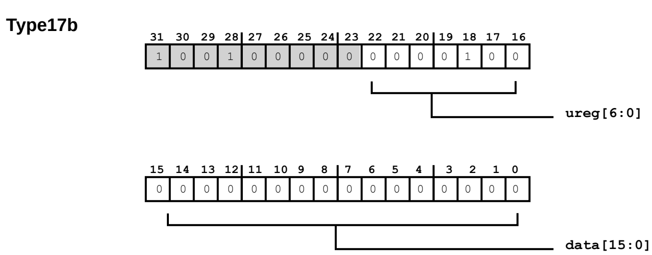

Figure 16-8: Type17b Instruction

17 Group IV Miscellaneous Instructions

The group IV instructions contains miscellaneous operations.

Type	Addr	Operation	SHARC 5 Stage Core
18a	ISA VISA	BIT SET *Sreg <data32>*; BIT CLR *Sreg <data32>*; BIT TGL *Sreg <data32>*; BIT TST *Sreg <data32>*; BIT XOR *Sreg <data32>*;	Yes
19a (modify)	ISA VISA	MODIFY (*Ia*,*<data32>*); MODIFY (*Ic*,*<data32>*); *Ia* = MODIFY (*Ia*,*<data32>*); *Ic* = MODIFY (*Ic*,*<data32>*);	Yes
19a (bit rev)	ISA VISA	BITREV (*Ia*, *<data32>*); BITREV (*Ic*, *<data32>*); *Ia* = BITREV (*Ia*,*<data32>*); *Ic* = BITREV (*Ic*,*<data32>*);	Yes
20a	ISA VISA	PUSH LOOP, PUSH STS, PUSH PCSTK, FLUSH CACHE; POP LOOP, POP STS, POP PCSTK, FLUSH CACHE;	Yes
		INVALIDATE I_CACHE; INVALIDATE DM_CACHE, INVALIDATE PM_CACHE; WRITEBACK DM_CACHE, WRITEBACK PM_CACHE; FLUSH DM_CACHE, FLUSH PM_CACHE;	No
21a	ISA VISA	NOP;	Yes
21c	VISA	NOP;	Yes
22a	ISA VISA	IDLE; EMUIDLE;	Yes
22c	VISA	IDLE; EMUIDLE;	Yes
2324	Reserved		

Type	Addr	Operation	SHARC 5 Stage Core
25a	ISA VISA	CJUMP *<addr24>* (db); CJUMP (PC, *<reladdr24>*) (db); RFRAME;	Yes
25a (direct)	ISA VISA	CJUMP *<addr24>* (db);	Yes
25a (PC relative)	ISA VISA	CJUMP (PC, *<reladdr24>*) (db);	Yes
25a (rframe)	ISA	RFRAME;	Yes
25c	VISA	RFRAME;	Yes
26a	ISA VISA	SYNC;	No

Type 18a ISA/VISA (register bit manipulation)

Syntax Summary

Type	Addr	Operation	
18a	ISA VISA	BIT SET *Sreg <data32>*; BIT CLR *Sreg <data32>*; BIT TGL *Sreg <data32>*; BIT TST *Sreg <data32>*; BIT XOR *Sreg <data32>*;	Yes

The following table provides the instruction syntax overview (Syntax)

Syntax
bit BOP (Type 18a) SYSREG Register Class imm32c Register Type ;

Abstract

System register bit manipulation

Description

SISD Mode

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

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

SIMD Mode

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

The X element operation uses the specified Sreg, and the Y element operations uses the complementary Csreg.

The following 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> ;
BIT CLR sreg <data32> ;
BIT TGL sreg <data32> ;
BIT TST sreg <data32> ;
BIT XOR sreg <data32> ;
```

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

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

Example

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

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

Because of the register selections in this example, the first instruction operates the same in SISD and SIMD, but the second instruction operates differently in SIMD. Only the *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 PEs (ASTATx and ASTATy), either one PE (ASTATx or ASTATy), or neither PE dependent on the outcome of the bit test operation.

Type18a Instruction Syntax

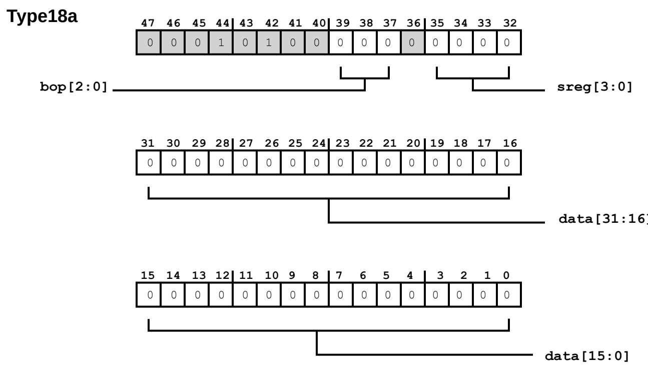

Figure 17-1: Type18a Instruction

BOP (Type 18a)

BOP Encode Table

bop	Syntax
000	set
001	clr
010	tgl
100	tst
101	xor

Type 19a ISA/VISA (index modify)

Syntax Summary

Type	Addr	Operation	

19a (modify)	ISA
	VISA
		`MODIFY (Ia,<data32>);`
		`MODIFY (Ic,<data32>);`
		`Ia = MODIFY (Ia,<data32>);`
		`Ic = MODIFY (Ic,<data32>);`
			Yes

The following table provides the opcode field values (sc, g) and the instruction syntax overview (Syntax)

sc	g	Syntax
10	0	modify(I1REG Register Class,imm32 Register Type);
--	0	I1REG Register Class = modify(I1REG Register Class,imm32 Register Type) BH (Type 19a - modify);
10	1	modify(I2REG Register Class,imm32 Register Type);

sc	g	Syntax
--	1	I2REG Register Class = modify(I2REG Register Class,imm32 Register Type) BH (Type 19a - modify);

Abstract

Immediate I register modify

Description

SISD and SIMD Modes

In SISD and SIMD modes, the Type 19 instruction modifies and adds the specified source Ia/Ic register with an immediate 32-bit data value and stores the result to the specified destination Ia/Ic register. If no destination register is specified then the source I register is updated. No address is output.

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

Example

```
MODIFY (I4, 304);
   /* operation is the same as I4=MODIFY(I4,304) */
I3 = MODIFY (I2,0x123);
I9 = MODIFY (I9,0x1);
```

Type19a Instruction Opcode

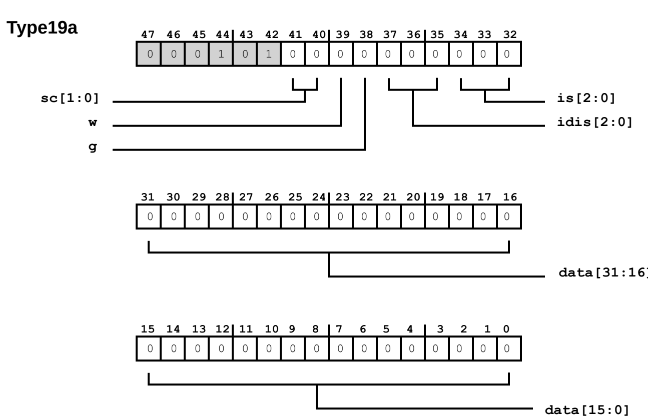

Figure 17-2: Type19a Instruction

BH (Type 19a - modify)

BH Encode Table

sc	w	Syntax
10	0	
01	0	(sw)
01	1	(nw)

Type 19a ISA/VISA (index bitrev)

Syntax Summary

Type	Addr	Operation	

19a (bit rev)	ISA
	VISA
		`BITREV (Ia, <data32>);`
		`BITREV (Ic, <data32>);`
		`Ia = BITREV (Ia,<data32>);`
		`Ic = BITREV (Ic,<data32>);`
			Yes

The following table provides the opcode field values (g) and the instruction syntax overview (Syntax)

g	Syntax
0	bitrev(I1REG Register Class,imm32 Register Type);
0	I1REG Register Class = bitrev(I1REG Register Class,imm32 Register Type);
1	bitrev(I2REG Register Class,imm32 Register Type);
1	I2REG Register Class = bitrev(I2REG Register Class,imm32 Register Type);

Abstract

Immediate I register bit-reverse

Description

SISD and SIMD Modes

In SISD and SIMD modes, if the address is to be bit-reversed (as specified by mnemonic), the modified value is bit-reversed before being written back to the destination I register. No address is output.

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

Example

```
4) */
BITREV (I7, space);
   /* "space" is a user-defined constant,
      operation is the same as
      I7=BITREV(I7,space) */
I2 = BITREV (I1,122); I15 =BITREV(I12,0x10);
```

Type19a_bitrev Instruction Opcode

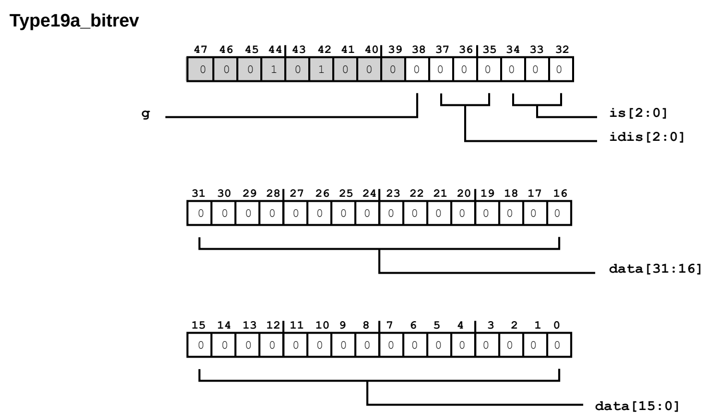

Figure 17-3: Type19a_bitrev Instruction

Type 20a ISA/VISA (push/pop stack/manipulate cache)

Syntax Summary

Type	Addr	Operation

20a	ISA
	VISA
		`PUSH LOOP, PUSH STS, PUSH PCSTK, FLUSH CACHE;POP LOOP, POP STS, POP PCSTK,`
		`FLUSH CACHE;`
		`INVALIDATE I_CACHE;`
		`INVALIDATE DM_CACHE, INVALIDATE PM_CACHE;`
		`WRITEBACK DM_CACHE, WRITEBACK PM_CACHE;`
		`FLUSH DM_CACHE, FLUSH PM_CACHE;`

The following table lists the opcode fields and provides links to their values.

Bit Fields	Control
fc	flush cache (see CACHE (Type 20a))
ppu, ppo	push/pop pcstk (see PCSTK (Type 20a))
spu, spo	push/pop sts (see STS (Type 20a))
lpu, lpo	push/pop loop (see LOOP (Type 20a))

Bit Fields	Control
l1ii	invalidate L1 I-cache (see ICACHE (Type 20a))
l1di, l1dwb	writeback, flush, or invalidate L1 DM-cache (see DMCACHE (Type 20a))
l1pi, l1pwb	writeback, flush, or invalidate L1 PM-cache (see PMCACHE (Type 20a))

Abstract

Push or Pop of loop and/or status stacks, or cache maintenance instruction.

Description

SISD and SIMD Modes

In SISD and SIMD modes, the Type 20 instruction pushes or pops the loop address and loop counter stacks, the status stack, and/or the PC stack, and/or clear the instruction-conflict 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-conflict cache invalidates all entries in the cache, and has an effect latency of one instruction when executing from internal memory, and two instructions when executing from external memory.

The Type 20 instruction also invalidates, flushes or writes back the L1 cache. These operations may not be combined with any other.

Example

```
PUSH LOOP;            // push loop stack
POP LOOP:          // pop loop stack
PUSH STS;         // push status stack
POP STS;        // pop status stack
PUSH PCSTK;        // push PC stack
POP PCSTK;        // pop PC stack
FLUSH CACHE;    // flush instruction-conflict cache
INVALIDATE I_CACHE;    // invalidate one line in L1 instruction cache
INVALIDATE DM_CACHE, INVALIDATE PM_CACHE; // invalidate one line in L1 data cache
WRITEBACK DM_CACHE, WRITEBACK PM_CACHE;  // write back line in L1 data cache
FLUSH DM_CACHE, FLUSH PM_CACHE ;    // flush one line in L1 data cache
PUSH LOOP, PUSH STS;               // push loop and status stacks
POP PCSTK, FLUSH CACHE;    // pop PC stack and flush instruction-conflict 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.

Type20a Instruction Opcode

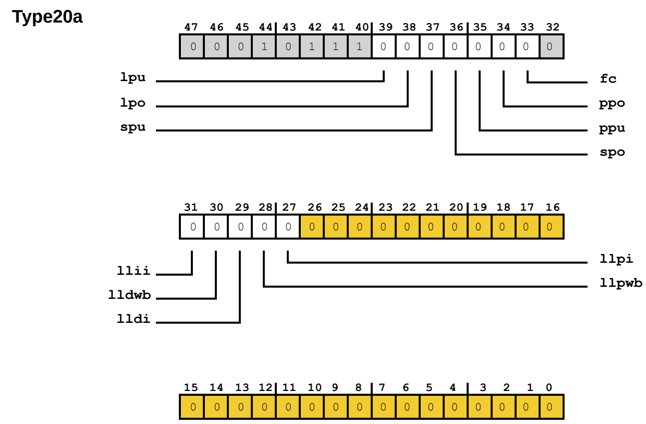

Figure 17-4: Type20a Instruction

CACHE (Type 20a)

CACHE Encode Table

The following table provides the opcode field values (spu, spo) and the instruction syntax overview (Syntax)

spu	spo	Syntax
0	0	
1	0	push sts
0	1	pop sts

DMCACHE (Type 20a)

DMCACHE Encode Table

The following table provides the opcode field values (l1di, l1dwb) and the instruction syntax overview (Syntax)

l1di	l1dwb	Syntax
0	0	
1	0	invalidate dm_cache
0	1	writeback dm_cache
1	1	writeback dm_cache

ICACHE (Type 20a)

ICACHE Encode Table

The following table provides the opcode field values (l1ii) and the instruction syntax overview (Syntax)

l1ii	Syntax
0	
1	invalidate i_cache

LOOP (Type 20a)

LOOP Encode Table

The following table provides the opcode field values (lpu, lpo) and the instruction syntax overview (Syntax)

lpu	lpo	Syntax
0	0	
1	0	push loop
0	1	pop loop

PCSTK (Type 20a)

PCSTK Encode Table

The following table provides the opcode field values (ppu, ppo) and the instruction syntax overview (Syntax)

ppu	ppo	Syntax
0	0	
1	0	push pcstk
0	1	pop pcstk

PMCACHE (Type 20a)

PMCACHE Encode Table

The following table provides the opcode field values (l1pi, l1pwb) and the instruction syntax overview (Syntax)

l1pi	l1pwb	Syntax
0	0	
1	0	invalidate pm_cache
0	1	writeback pm_cache
1	1	flush pm_cache

STS (Type 20a)

STS Encode Table

The following table provides the opcode field values (spu, spo) and the instruction syntax overview (Syntax)

spu	spo	Syntax
0	0	
1	0	push sts
0	1	pop sts

Type 21a ISA/VISA (nop)

Syntax Summary

Type	Addr	Operation	
21a	ISA VISA	`NOP;`	Yes

The following table provides the instruction syntax overview (Syntax)

Syntax
nop ;

Abstract

No Operation (NOP)

Description

SISD and SIMD Modes

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

Example

```
nop;
```

Type21a Instruction Opcode

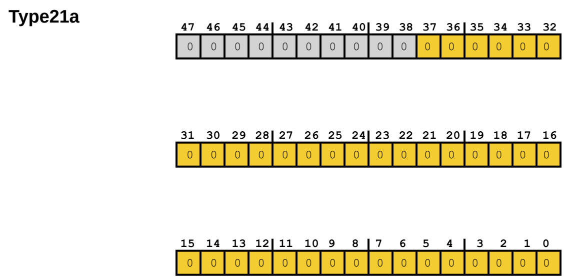

Figure 17-5: Type21a Instruction

Type 21c VISA (nop)

Syntax Summary

Type	Addr	Operation	
21c	VISA	`NOP;`	Yes

The following table provides the instruction syntax overview (Syntax)

Syntax
nop ;

Abstract

No Operation (NOP)

Description

SISD and SIMD Modes

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

Example

```
nop;
```

Type21c Instruction Opcode

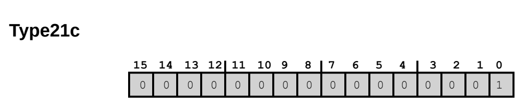

Figure 17-6: Type21c Instruction

Type 22a ISA/VISA (idle/emuidle)

Syntax Summary

Type	Addr	Operation	
22a	ISA VISA	`IDLE;` `EMUIDLE;`	Yes

The following table provides the opcode field values (emu) and the instruction syntax overview (Syntax)

emu	Syntax
0	idle ;
1	emuidle ;

Abstract

Low power/emulation halt instruction

Description

SISD and SIMD Modes

In SISD and SIMD modes, the Type 22 `idle` instruction puts the processor in a low power state. The processor remains in the low power state until an interrupt occurs. On return from the interrupt, execution continues at the instruction following the Idle instruction. The `emuidle` instruction halts the core caused by a software breakpoint hit and places the core in emulation space. An RTI instruction releases the core back to user space.

Example

```
IDLE;
EMUIDLE;
```

Type22a Instruction Opcode

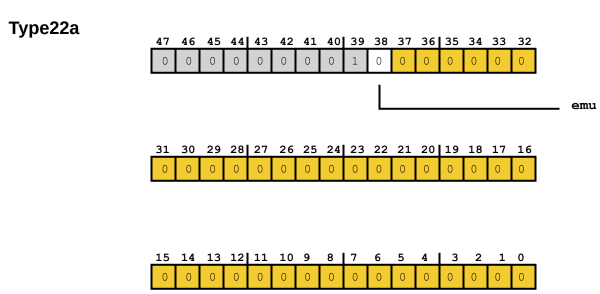

Figure 17-7: Type22a Instruction

Type 22c VISA (idle/emuidle)

Syntax Summary

Type	Addr	Operation	
22c	VISA	`IDLE;` `EMUIDLE;`	Yes

The following table provides the opcode field values (emu) and the instruction syntax overview (Syntax)

emu	Syntax
0	idle ;
1	emuidle ;

Abstract

Low power/emulation halt instruction

Description

SISD and SIMD Modes

In SISD and SIMD modes, the Type 22 `idle` instruction puts the processor in a low power state. The processor remains in the low power state until an interrupt occurs. On return from the interrupt, execution continues at the instruction following the Idle instruction. The `emuidle` instruction halts the core caused by a software breakpoint hit and places the core in emulation space. An RTI instruction releases the core back to user space.

Example

```
IDLE;
EMUIDLE;
```

Type22c Instruction Syntax

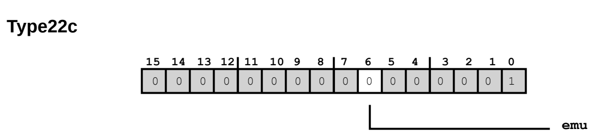

Figure 17-8: Type22c Instruction

Type 25a ISA/VISA (cjump direct)

Syntax Summary

Type	Addr	Operation	
25a (direct)	ISA VISA	`CJUMP <addr24> (db);`	Yes

The following table provides the instruction syntax overview (Syntax)

Syntax
cjump imm24 Register Type (db) ;

Abstract

Cjump (Compiler-generated instruction)

Description

Function (SISD and SIMD)

In SISD mode, the Type 25 instruction (cjump) combines a direct jump with register transfer operations that save 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 in assembly programs. The cjump instruction should always use the DB modifier.

Example

Table 17-1: Operations Done by Forms of the Type 25 Instruction

Compiler-Generated Instruction	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;`

Type25a_direct Instruction Opcode

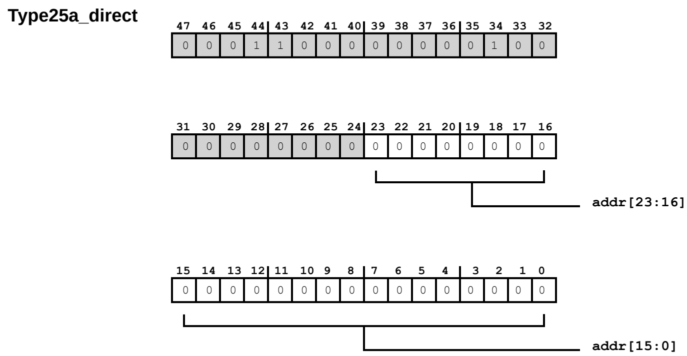

Figure 17-9: Type25a_direct Instruction

Type 25a ISA/VISA (cjump PC relative)

Syntax Summary

Type	Addr	Operation	
25a (PC relative)	ISA VISA	`CJUMP (PC, <reladdr24>) (db);`	Yes

The following table provides the instruction syntax overview (Syntax)

Syntax
cjump (pc,imm24pc Register Type) (db) ;

Abstract

Cjump (Compiler-generated instruction)

Description

Function (SISD and SIMD)

In SISD mode, the Type 25 instruction (cjump) combines a PC-relative jump with register transfer operations that save 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 in assembly programs. The cjump instruction should always use the DB modifier.

The different forms of this instruction perform the operations where `raddr` indicates a relative 24-bit address.

Type25a_pcrel Instruction Opcode

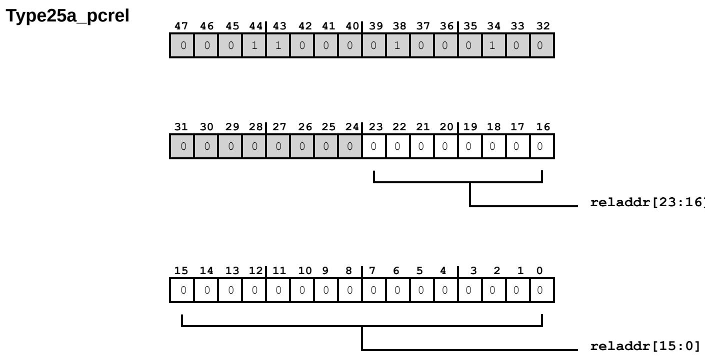

Figure 17-10: Type25a_pcrel Instruction

Type 25a ISA/VISA (rframe)

Syntax Summary

Type	Addr	Operation
25a (rframe)	ISA VISA	RFRAME;

The following table provides the instruction syntax overview (Syntax)

Syntax
rframe ;

Abstract

Rframe (Compiler-generated instruction)

Description

Function (SISD and SIMD)

In SISD mode, 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 rframe in assembly programs.

Example

Table 17-2: Operations Done by Forms of the Type 25 Instruction

Compiler-Generated Instruction	Operations Performed in SISD Mode	Operations Performed in SIMD Mode
`RFRAME;`	`I7=I6, I6=DM(0,I6);`	`I7=I6, I6=DM(0,I6);`

Type25a rframe Instruction Opcode

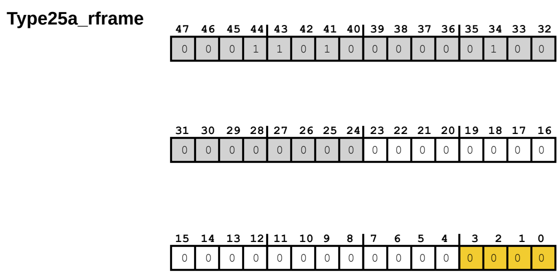

Figure 17-11: Type25a_rframe Opcode

Type 25c VISA (rframe)

Abstract

Rframe (Compiler-generated instruction)

Syntax Summary

Type	Addr	Operation
25c (rframe)	VISA	`RFRAME;`

The following table provides the instruction syntax overview (Syntax)

Syntax
(rframe)

Description

Function (SISD and SIMD)

In SISD mode, 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 rframe in assembly programs.

Example

Table 17-3: Operations Done by Forms of the Type 25 Instruction

Compiler-Generated Instruction	Operations Performed in SISD Mode	Operations Performed in SIMD Mode
`RFRAME;`	`I7=I6, I6=DM(0,I6);`	`I7=I6, I6=DM(0,I6);`

Description

Function (SISD and SIMD)

In SISD mode, 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 rframe in assembly programs.

Example

Table 17-4: Operations Done by Forms of the Type 25 Instruction

Compiler-Generated Instruction	Operations Performed in SISD Mode	Operations Performed in SIMD Mode
`RFRAME;`	`I7=I6, I6=DM(0,I6);`	`I7=I6, I6=DM(0,I6);`

Type25c_rframe Instruction Opcode

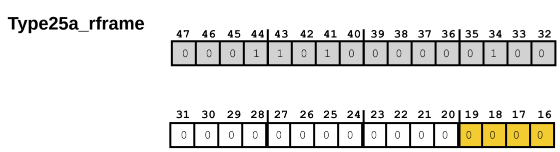

Figure 17-12: Type25c_rframe Instruction

Type 26a ISA/VISA (sync)

Syntax Summary

Type	Addr	Operation	
26a	ISA VISA	`SYNC;`	No

The following table provides the instruction syntax overview (Syntax)

Syntax

Syntax
`sync ;`

Abstract

Synchronization insruction

Description

The SYNC instruction ensures completion of all pending writes on the system interface as well as the internal memory (L1) interface. The core pipeline is stalled until SYNC completes

Example

```
SYNC;
```

Type26a Sync Instruction Opcode

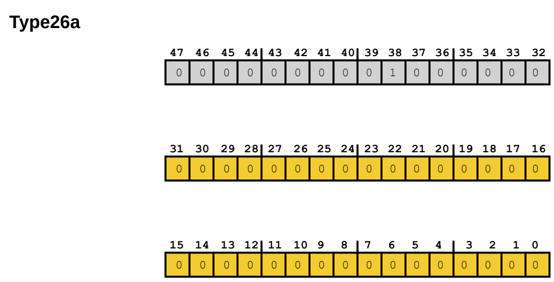

Figure 17-13: Type26a Instruction

18 Computation Opcode Reference

This chapter describes the fields from the instruction set types (COMPUTE, SHORT COMPUTE and SHIFT IMMEDIATE). The 23-bit compute field is a mini instruction within the ADSP-21xxx instruction. The following compute operations can be specified:

- Single-function operations involve a single computation unit
- Shift immediate functions (type 6a only)
- Short compute functions (type 2c only)
- Multifunction operations specify parallel operation of the multiplier and 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 instruction, the assembly language syntax, including options, and its related functionality are described. All related status flags are listed.

The following tables show the different compute field coding depending on single computations versus multi-computation and its supported data format.

Table 18-1: Single Computation Instruction Coding SINGLEFN

MF bit [22]	CU bit [21:20]	Opcode bit [19:12]	Computation Type	Data Format
0	00	0xxx xxxx	ALU	32-bit Fixed
0	00	1xxx xxxx		32/40-bit Float
0	00	0xx1 xxxx		64-bit Float
0	01	xxxx xxxx	Multiply	32-bit Fixed
0	01	0011 0000		32/40-bit Float
0	01	0011 0011		64-bit Float
0	10	xxxx xxxx	Shifter	32-bit Fixed

Table 18-2: Short Compute Instruction Coding SINGLEFN (type 2C only)

SC bit [15:12]	Opcode bit [11:8]	Computation Type	Data Format
1100	xxxx	Short Compute	32/40-bit Fixed/Float

Table 18-3: Single Compute Parallel Add/Subtract Instruction Coding SINGLEFN

MF bit [22]	CU bit [21:20]	Opcode bit [19:16]	Computation Type	Data Format
0	00	0111	Dual Add/Subtract	32-bit Fixed
0	00	1111		32/40-bit Float

Table 18-4: Multi Compute Instruction Coding MULTIFN

MF bit [22]	Opcode bit [21:16]	Computation Type	Data Format
1	0xxxxx	MUL/ALU	32-bit Fixed
1	011xxx		32/40-bit Float
1	00xx11		64-bit Float
1	10xxxx	MUL Dual Add/Subtract	32-bit Fixed
1	11xxxx		32/40-bit Float

Compute (Compute) Opcode

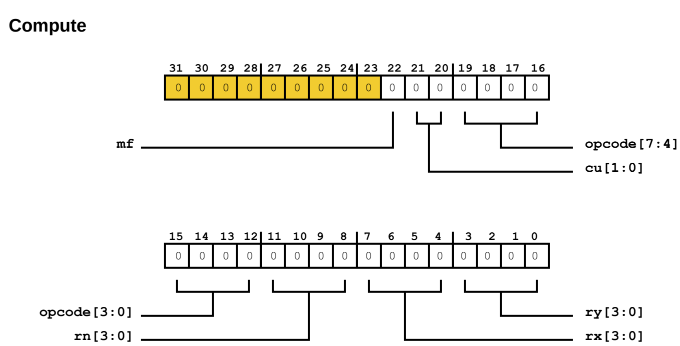

Figure 18-1: Compute Instruction

Short Compute (ShortCompute) Opcode

The following compute instructions are supported as type 2c instructions in VISA space under the condition that one source register and one destination register must be identical.

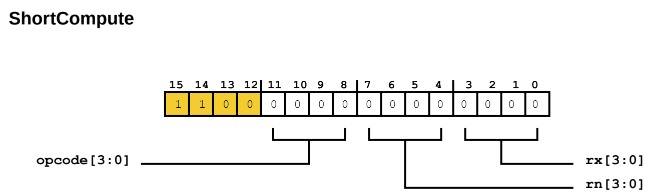

Figure 18-2: ShortCompute Instruction

The following table provides the opcode field values (opcode), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

opcode (bits 11–8)	Syntax	Instruction
0000	RREG Register Class = RREG Register Class + RREG Register Class	RN = RN + RX
0001	RREG Register Class = RREG Register Class - RREG Register Class	RN = RN - RX
0010	RREG Register Class = pass RREG Register Class	RN = pass RX;
0011	comp (RREG Register Class, RREG Register Class)	comp (RN, RX)
0100	RREG Register Class = not RREG Register Class	RN = not RX;
0101	RREG Register Class = RREG Register Class + 1	RN = RX + 1;
0110	RREG Register Class = RREG Register Class - 1	RN = RX – 1;
0111	RREG Register Class = RREG Register Class * RREG Register Class (ssi)	RN = RN * RX (ssi)
1000	FREG Register Class = FREG Register Class + FREG Register Class	FN = FN + FX
1001	FREG Register Class = FREG Register Class - FREG Register Class	FN = FN - FX
1010	FREG Register Class = float RREG Register Class	FN = float RX
1011	comp (FREG Register Class, FREG Register Class)	comp (FN, FX)
1100	RREG Register Class = RREG Register Class and RREG Register Class	RN = RN and RX
1101	RREG Register Class = RREG Register Class or RREG Register Class	RN = RN or RX

opcode (bits 11–8)	Syntax	Instruction
1110	RREG Register Class = RREG Register Class xor RREG Register Class	RN = RN xor RX
1111	FREG Register Class = FREG Register Class * FREG Register Class	FN = FN * FX

Shift Immediate (ShiftImm) Opcode (Type 6 Instruction only)

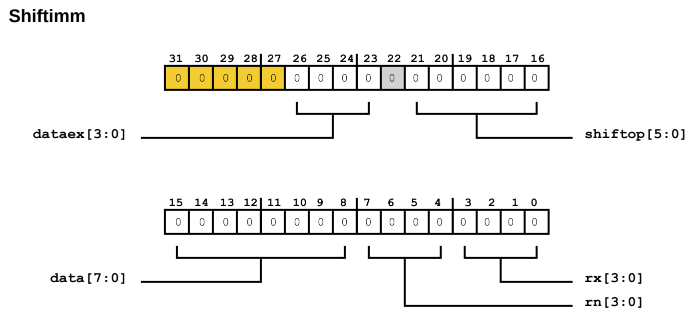

Figure 18-3: ShiftImm Computation Opcode

Single Function Instruction (SINGLEFN)

ALUOP

Table 18-5: ALUOP Encoding (32-bit/40-bit fixed-point/floating-point operations)

opcode (bits 19–12)	Syntax	Instruction
00000001	RREG Register Class = RREG Register Class + RREG Register Class	RN = RX + RY;
00000010	RREG Register Class = RREG Register Class - RREG Register Class	RN = RX – RY;
00000101	RREG Register Class = RREG Register Class + RREG Register Class + ci	RN = RX + RY + ci;
00000110	RREG Register Class = RREG Register Class - RREG Register Class + ci - 1	RN = RX – RY + ci – 1;
00001001	RREG Register Class = (RREG Register Class + RREG Register Class) / 2	RN = (RX + RY) / 2;
00001010	comp(RREG Register Class, RREG Register Class)	comp (RX, RY);
00001011	compu(RREG Register Class, RREG Register Class)	compu (RX, RY);

Table 18-5: ALUOP Encoding (32-bit/40-bit fixed-point/floating-point operations) (Continued)

opcode (bits 19–12)	Syntax	Instruction
00100001	RREG Register Class = pass RREG Register Class	RN = pass RX;
00100010	RREG Register Class = - RREG Register Class	RN = –RX;
00100101	RREG Register Class = RREG Register Class + ci	RN = RX + ci;
00100110	RREG Register Class = RREG Register Class + ci – 1	RN = RX + ci – 1;
00101001	RREG Register Class = RREG Register Class + 1	RN = RX + 1;
00101010	RREG Register Class = RREG Register Class – 1	RN = RX – 1;
00110000	RREG Register Class = abs RREG Register Class	RN = abs RX;
01000000	RREG Register Class = RREG Register Class and RREG Register Class	RN = RX and RY;
01000001	RREG Register Class = RREG Register Class or RREG Register Class	RN = RX or RY;
01000010	RREG Register Class = RREG Register Class xor RREG Register Class	RN = RX xor RY;
01000011	RREG Register Class = not RREG Register Class	RN = not RX;
01100001	RREG Register Class = min(RREG Register Class, RREG Register Class)	RN = min (RX, RY);
01100010	RREG Register Class = max(RREG Register Class, RREG Register Class)	RN = max (RX, RY);
01100011	RREG Register Class = clip RREG Register Class by RREG Register Class	RN = clip RX by RY;
10000001	FREG Register Class = FREG Register Class + FREG Register Class	FN = FX + FY;
10000010	FREG Register Class = FREG Register Class - FREG Register Class	FN = FX – FY;
10001001	FREG Register Class = (FREG Register Class + FREG Register Class) / 2	FN = (FX + FY) / 2;
10001010	comp(FREG Register Class, FREG Register Class)	comp (FX, FY);
10010001	FREG Register Class = abs (FREG Register Class + FREG Register Class)	FN = abs (FX + FY);
10010010	FREG Register Class = abs (FREG Register Class – FREG Register Class)	FN = abs (FX – FY);
10100001	FREG Register Class = pass FREG Register Class	FN = pass FX;
10100010	FREG Register Class = - FREG Register Class	FN = –FX;
10100101	FREG Register Class = rnd FREG Register Class	FN = rnd FX;
10101101	RREG Register Class = mant FREG Register Class	RN = mant FX;
10110000	FREG Register Class = abs FREG Register Class	FN = abs FX;

Table 18-5: ALUOP Encoding (32-bit/40-bit fixed-point/floating-point operations) (Continued)

opcode (bits 19–12)	Syntax	Instruction
10111101	FREG Register Class = scalb FREG Register Class by RREG Register Class	FN = scalb FX by RY;
11000001	RREG Register Class = logb FREG Register Class	RN = logb FX;
11000100	FREG Register Class = recips FREG Register Class	FN = recips FX;
11000101	FREG Register Class = rsqrts FREG Register Class	FN = rsqrts FX;
11001001	RREG Register Class = fix FREG Register Class	RN = fix FX;
11001010	FREG Register Class = float RREG Register Class	FN = float RX;
11001101	RREG Register Class = trunc FREG Register Class	RN = trunc FX;
11011001	RREG Register Class = fix FREG Register Class by RREG Register Class	RN = fix FX by RY;
11011010	FREG Register Class = float RREG Register Class by RREG Register Class	FN = float RX by RY;
11011101	RREG Register Class = trunc FREG Register Class by RREG Register Class	RN = trunc FX by RY;
11100000	FREG Register Class = FREG Register Class copysign FREG Register Class	FN = FX copysign FY;
11100001	FREG Register Class = min(FREG Register Class, FREG Register Class)	FN = min (FX, FY);
11100010	FREG Register Class = max(FREG Register Class, FREG Register Class)	FN = max (FX, FY);
11100011	FREG Register Class = clip FREG Register Class by FREG Register Class	FN = clip FX by FY;

Table 18-6: ALUOP Encoding (64-bit floating-point operations)

opcode (bits 19–12)	Syntax	Instruction
00010001	DBLREG Register Type = DBLREG Register Type + DBLREG Register Type	FM:N = FX:Y + FZ:W;
00010010	DBLREG Register Type = DBLREG Register Type - DBLREG Register Type	FM:N = FX:Y - FZ:W;
00010011	comp(DBLREG Register Type, DBLREG Register Type)	comp (FX:Y, FZ:W);
00010100	DBLREG Register Type = - DBLREG Register Type	FM:N = - FX:Y;
00010101	DBLREG Register Type = abs DBLREG Register Type	FM:N = abs FX:Y;
00010110	DBLREG Register Type = pass DBLREG Register Type	FM:N = pass FX:Y;
00010111	RREG Register Class = fix DBLREG Register Type	

Table 18-6: ALUOP Encoding (64-bit floating-point operations) (Continued)

opcode (bits 19–12)	Syntax	Instruction
		RN=fix FX:Y;
00011000	RREG Register Class = fix DBLREG Register Type by RREG Register Class	RN = fix FX:Y by RY;
00011001	RREG Register Class = trunc DBLREG Register Type	RN = trunc FX:Y;
00011010	RREG Register Class = trunc DBLREG Register Type by RREG Register Class	RN = trunc FX:Y by RY;
00011011	DBLREG Register Type = float RREG Register Class	FM:N = float RX;
00011100	DBLREG Register Type = float RREG Register Class by RREG Register Class	FM:N = float RX by RY;
00011101	DBLREG Register Type = cvt FREG Register Class	FM:N = cvt FX;
00011110	FREG Register Class = cvt DBLREG Register Type	FN = cvt FX:Y;
00011111	DBLREG Register Type = scalb DBLREG Register Type by RREG Register Class	FM:N = scalb FX:Y by RY;

MULOP

This section describes the multiplier operations. These tables use the following symbols to indicate the location of operands and other features:

- y = y-input (1 = signed, 0 = unsigned)
- x = x-input (1 = signed, 0 = unsigned)
- f = format (1 = fractional, 0 = integer)
- r = rounding (1 = yes, 0 = no)

Table 18-7: MULOP Encode Table (32-bit/40-bit fixed-point/floating-point operations)

opcode (bits 19–12)	Syntax	Instruction
0000 F00x	RREG Register Class = sat mrf MOD2	(RN | mrf | mrb) = sat (mrf | mrb) MOD2;
0000 F01x	RREG Register Class = sat mrb MOD2	(RN | mrf | mrb) = sat (mrf | mrb) MOD2;
0000 F10x	mrf = sat mrf MOD2	(RN | mrf | mrb) = sat (mrf | mrb) MOD2;
0000 F11x	mrb = sat mrb MOD2	(RN | mrf | mrb) = sat (mrf | mrb) MOD2;
0001 0100	mrf = 0	(mrf | mrb) = 0;
0001 0110	mrb = 0	(mrf | mrb) = 0;
0001 100x	RREG Register Class = rnd mrf MOD3	(RN | mrf | mrb) = rnd (mrf | mrb) MOD3;

Table 18-7: MULOP Encode Table (32-bit/40-bit fixed-point/floating-point operations) (Continued)

opcode (bits 19–12)	Syntax	Instruction
0001 101x	RREG Register Class = rnd mrb MOD3	(RN | mrf | mrb) = rnd (mrf | mrb) MOD3;
0001 110x	mrf = rnd mrf MOD3	(RN | mrf | mrb) = rnd (mrf | mrb) MOD3;
0001 111x	mrb = rnd mrb MOD3	(RN | mrf | mrb) = rnd (mrf | mrb) MOD3;
01yx f00r	RREG Register Class = RREG Register Class * RREG Register Class MOD1	(RN | mrf | mrb) = RX * RY MOD1;
01yx F10r	mrf = RREG Register Class * RREG Register Class MOD1	(RN | mrf | mrb) = RX * RY MOD1;
01yx F11r	mrb = RREG Register Class * RREG Register Class MOD1	(RN | mrf | mrb) = RX * RY MOD1;
10yx F00r	RREG Register Class = mrf + RREG Register Class * RREG Register Class MOD1	RN = (mrf | mrb) + RX * RY MOD1;
10yx F01r	RREG Register Class = mrb + RREG Register Class * RREG Register Class MOD1	RN = (mrf | mrb) + RX * RY MOD1;
10yx F10r	mrf = mrf + RREG Register Class * RREG Register Class MOD1	(mrf | mrb) = MRF + RX * RY MOD1;
10yx F11r	mrb = mrb + RREG Register Class * RREG Register Class MOD1	(mrf | mrb) = MRF + RX * RY MOD1;
11yx F00r	RREG Register Class = mrf - RREG Register Class * RREG Register Class MOD1	RN = (mrf | mrb) – RX * RY MOD1;
11yx F01r	RREG Register Class = mrb - RREG Register Class * RREG Register Class MOD1	RN = (mrf | mrb) – RX * RY MOD1;
11yx F10r	mrf = mrf - RREG Register Class * RREG Register Class MOD1	(mrf | mrb) = (mrf | mrb) – RX * RY MOD1;
11yx F11r	mrb = mrb - RREG Register Class * RREG Register Class MOD1	(mrf | mrb) = (mrf | mrb) – RX * RY MOD1;
0011 0000	FN = FX * FY	FLP_Mult

Table 18-8: MULOP Encode Table (64-bit floating-point operations)

opcode (bits 19–12)	Syntax	Instruction
0011 0001	DBLREG Register Type = DBLREG Register Type * DBLREG Register Type	FM:N = FX:Y * FZ:W;
0011 0010	DBLREG Register Type = DBLREG Register Type * FREG Register Class	FM:N = FX:Y * FY;
0011 0011	DBLREG Register Type = FREG Register Class * FREG Register Class	FM:N = FX * FY;

MOD1

The Mod1 modifiers in the following table are optional modifiers. It is enclosed in parentheses and consists of three or four letters that indicate whether:

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

MOD1 Encode Table

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

MOD2

The Mod2 modifiers in the following table are optional modifiers, enclosed in parentheses, consisting of two letters that indicate whether the input is signed (S) or unsigned (U) and whether the input is in integer (I) or fractional (F) format.

MOD2 Encode Table

Option	Opcode
(SI)	_ _ _ _ 0 _ _ 1
(UI)	_ _ _ _ 0 _ _ 0
(SF)	_ _ _ _ 1 _ _ 1

Option	Opcode
(UF)	_ _ _ _ 1 _ _ 0

MOD3

MOD3 Encode Table

Option	Opcode
(SF)	_ _ _ _ 1 _ _ 1
(UF)	_ _ _ _ 1 _ _ 0

SHIFTOP/SHIFTIMM

The following table provides opcode field values for the shiftimm instruction (see Shift Immediate (ShiftImm) Opcode (Type 6 Instruction only)) and the Compute instruction (see Compute (Compute) Opcode).

Table 18-9: SHIFTOP/SHIFTIMM Encode Table (32-bit Fixed-Point Operations)

shiftimm (bits 21-16)	Shiftimm Syntax, Type 6 Instruction	shiftop (bits 19-12)	Shiftop Syntax	Instruction
000000	RFREG Register Class = lshift RREG Register Class by DATA8	00000000	RREG Register Class = lshift RREG Register Class by RREG Register Class	RN = lshift RX by (RY | DATA8); DATA8
000001	RREG Register Class = ashift RREG Register Class by DATA8	00000100	RREG Register Class = ashift RREG Register Class by RREG Register Class	FXP_ASHIFT_BY DATA8
000010	RREG Register Class = rot RREG Register Class by DATA8	00001000	RREG Register Class = rot RREG Register Class by RREG Register Class	RN = rot RX by (RY | DATA);
001000	RREG Register Class = RREG Register Class or lshift RREG Register Class by DATA8	00100000	RREG Register Class = RREG Register Class or lshift RREG Register Class by RREG Register Class	FXP_OR_LSHIFT_BY
001001	RREG Register Class = RREG Register Class or ashift RREG Register Class by DATA8	00100100	RREG Register Class = RREG Register Class or ashift RREG Register Class by RREG Register Class	RN = RN or ashift RX by (RY | DATA8);
010000	RREG Register Class = fext RREG Register Class by BIT6:LEN6	01000000	RREG Register Class = fext RREG Register Class by RREG Register Class	RN = fext RX by (RY | BIT6:LEN6);
010001	RREG Register Class = fdep RREG Register Class by BIT6:LEN6	01000100	RREG Register Class = fdep RREG Register Class by RREG Register Class	RN = fdep RX by (RY | BIT6:LEN6);BIT6:LEN6

Table 18-9: SHIFTOP/SHIFTIMM Encode Table (32-bit Fixed-Point Operations) (Continued)

shiftimm (bits 21-16)	Shiftimm Syntax, Type 6 Instruction	shiftop (bits 19-12)	Shiftop Syntax	Instruction
010010	RREG Register Class = fext RREG Register Class by BIT6:LEN6 (se)	01001000	RREG Register Class = fext RREG Register Class by RREG Register Class (se)	RN = fext RX by (RY | BIT6:LEN6) (se);BIT6:LEN6
010011	RREG Register Class = fdep RREG Register Class by BIT6:LEN6 (se)	01001100	RREG Register Class = fdep RREG Register Class by RREG Register Class (se)	FXP_BITEXT_bitlen12_NU BIT6:LEN6
010100	RREG Register Class = bitext BITLEN12 (nu)	01010000	RREG Register Class = bitext RREG Register Class	FXP_BITEXT_bitlen12_NU
011001	RREG Register Class = bitext BITLEN12 (nu)	01011000	RREG Register Class = bitext RREG Register Class (nu)	RN = bitext (RX | BITLEN12) (nu);
011011	RREG Register Class = RREG Register Class or fdep RREG Register Class by BIT6:LEN6	01100100	RREG Register Class = RREG Register Class or fdep RREG Register Class by RREG Register Class	RN = RN or fdep RX by (RY | BIT6:LEN6);
011101	RREG Register Class = RREG Register Class or fdep RREG Register Class by BIT6:LEN6 (se)	01101100	RREG Register Class = RREG Register Class or fdep RREG Register Class by RREG Register Class (se)	RN = RN or fdep RX by (RY | BIT6:LEN6) (se);
	N/A	01110000	RREG Register Class = bffwrp	FXP_BFFWRP
011111	bffwrp = DATA7	01111100	bffwrp = RREG Register Class	bffwrp = (RN | DATA7);
110000	RREG Register Class = bset RREG Register Class by BITLEN12	11000000	RREG Register Class = bset RREG Register Class by RREG Register Class	RN = bset RX by (RY | DATA8);
110001	RREG Register Class = bclr RREG Register Class by DATA7	11000100	RREG Register Class = bclr RREG Register Class by RREG Register Class	RN = bclr RX by (RY | DATA8);
110010	RREG Register Class = btgl RREG Register Class by RREG Register Class	11001000	RREG Register Class = btgl RREG Register Class by RREG Register Class	RN = btgl RX by (RY | DATA8);
110011	btst RREG Register Class by RREG Register Class	11001100	btst RREG Register Class by RREG Register Class	btst RX by (RY | DATA8);
		10000000	RREG Register Class = exp RREG Register Class	RN = exp RX;
		10000100	bffwrp = DATA7	RN = exp RX; (ex)
		10001000	RREG Register Class = leftz RREG Register Class	RN = leftz RX;
		10001100	RREG Register Class = lefto RREG Register Class	RN = lefto RX;

Table 18-9: SHIFTOP/SHIFTIMM Encode Table (32-bit Fixed-Point Operations) (Continued)

shiftimm (bits 21-16)	Shiftimm Syntax, Type 6 Instruction	shiftop (bits 19-12)	Shiftop Syntax	Instruction
		10010000	RREG Register Class = fpack FREG Register Class	RN = fpack FX;
		10010100	FREG Register Class = funpack RREG Register Class	FN = funpack RX;

Dual Add/Subtract

Table 18-10: Dual add/subtract Encode Table (32-bit/40-bit fixed-point/floating-point operations)

Opcode (bits 19–16)	Syntax	Instruction
0111	RREG Register Class = RREG Register Class + RREG Register Class , RREG Register Class = RREG Register Class – RREG Register Class	Ra = Rx + Ry , Rs = Rx – Ry
1111	FREG Register Class = FXAREG Register Class + FYAREG Register Class , FREG Register Class = FXAREG Register Class – FYAREG Register Class	Fa = Fx + Fy , Fs = Fx – Fy

Register File

This section covers all source and result register file encodings depending on compute instruction types.

Single Computation Encoding 32/40-bit

Table 18-11: Compute Field (Fixed-Point)

11	10	9	8	7	6	5	4	3	2	1	0
RN/FN				Rx/Fx				Ry/Fy			

Table 18-12: Compute Field Bit Descriptions

Bit	Description
RN	Specifies fixed-point result register
Rx	Specifies fixed-point X input register
Ry	Specifies fixed-point Y input register
FN	Specifies floating-point ALU addition result
Fx	Specifies floating-point X input register

Table 18-12: Compute Field Bit Descriptions (Continued)

Bit	Description
Fy	Specifies floating-point Y input register

Dual Add/Subtract Encoding 32/40-bit

Table 18-13: Compute Field (Fixed-Point)

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="4">Rx/Fs</td><td colspan="4">Ra/Fa</td><td colspan="4">Rx/Fx</td><td colspan="4">Ry/Fy</td></tr>
</table>

Table 18-14: Compute Field Bit (Fixed-Point) Bit Descriptions

Bit	Description
Rx	Specifies fixed-point X input ALU register
Ry	Specifies fixed-point Y input ALU register
Rs	Specifies fixed-point ALU subtraction result
Ra	Specifies fixed-point ALU addition result
Fx	Specifies floating-point X input ALU register
Fy	Specifies floating-point Y input ALU register
Fs	Specifies floating-point ALU subtraction result
Fa	Specifies floating-point ALU addition result

Mul/ALU Encoding 32/40-bit

Table 18-15: Compute Field (Fixed-Point)

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="4">Rm</td><td colspan="4">Ra</td><td colspan="2">Rxm</td><td colspan="2">Rym</td><td colspan="2">Rxa</td><td colspan="2">Rya</td></tr>
</table>

Table 18-16: Compute Field (Floating-Point)

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="4">Fm</td><td colspan="4">Fa</td><td colspan="2">Fxm</td><td colspan="2">Fym</td><td colspan="2">Fxa</td><td colspan="2">Fya</td></tr>
</table>

Table 18-17: Mul/ALU Encoding 32/40-bit Bit Descriptions

Bit	Description
Rxa	Specifies fixed-point X input ALU register (R11–8)
Rya	Specifies fixed-point Y input ALU register (R15–12)
Ra	Specifies fixed-point ALU result

Table 18-17: Mul/ALU Encoding 32/40-bit Bit Descriptions (Continued)

Bit	Description
Fxa	Specifies floating-point X input ALU register (F11–8)
Fya	Specifies floating-point Y input ALU register (F15–12)
Fa	Specifies floating-point ALU result
Rxm	Specifies fixed-point X input multiply register (R3–0)
Rym	Specifies fixed-point Y input multiply register (R7–4)
Rm	Specifies fixed-point multiply result register
Fxm	Specifies floating-point X input multiply register (F3–0)
Fym	Specifies floating-point Y input multiply register (F7–4)
Fm	Specifies floating-point multiply result register

Mul Dual Add/Subtract Encoding 32/40-bit

Table 18-18: Compute Field (Fixed-Point)

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rs				Rm				Ra				Rxm		Rym		Rxa		Rya	

Table 18-19: Compute Field (Floating-Point)

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Fs				Fm				Fa				Fxm		Fym		Fxa		Fya	

Table 18-20: Mul dual Add/Subtract Encoding 32/40-bit Bit Descriptions

Bit	Description
Rxa	Specifies fixed-point X input ALU register (R11–8)
Rya	Specifies fixed-point Y input ALU register (R15–12)
Rs	Specifies fixed-point ALU subtraction result
Ra	Specifies fixed-point ALU addition result
Fxa	Specifies floating-point X input ALU register (F11–8)
Fya	Specifies floating-point Y input ALU register (F15–12)
Fs	Specifies floating-point ALU subtraction result
Fa	Specifies floating-point ALU addition result
Rxm	Specifies fixed-point X input multiply register (R3–0)
Rym	Specifies fixed-point Y input multiply register (R7–4)

Table 18-20: Mul dual Add/Subtract Encoding 32/40-bit Bit Descriptions (Continued)

Bit	Description
Rm	Specifies fixed-point multiply result register
Fxm	Specifies floating-point X input multiply register (F3–0)
Fym	Specifies floating-point Y input multiply register (F7–4)
Fm	Specifies floating-point multiply result register

Short Compute 32/40-bit

Table 18-21: Compute Field (Fixed-Point)

7	6	5	4	3	2	1	0
RN/FN				RX/FX			

Table 18-22: Compute Field (Fixed-Point) Bit Descriptions

Bit	Description
RX	Specifies fixed-point X input register
RN	Specifies fixed-point Y input and result register
FX	Specifies floating-point X input register
FN	Specifies floating-point Y input and result register

Single Function Floating-Point 64-bit

Table 18-23: Single Function Floating-Point 64-bit

11	10	9	8	7	6	5	4	3	2	1	0
Fm:n				Fx:y				Fz:w			

Table 18-24: Single Function Floating-Point 64-bit Bit Descriptions

Bit	Description
Fm:n	Specifies the result register
Fx:y	Specifies the source1 operand
Fz:w	Specifies the source2 operand

Multi-function Floating-Point 64-bit

Table 18-25: Multi-function Floating-point 64-bit

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Fm:n				Fa:b				Fx:y		Fz:w		Fp:q		Fr:s	

Table 18-26: Multi-function Floating-point 64-bit Bit Descriptions

Bit	Description
Fm:n	Specifies the result register for the multiplier result
Fa:b	Specifies the result register for the dual Add/Subtract result
Fx:y	Specifies the source1 operand for the multiplier
Fz:w	Specifies the source2 operand for the multiplier
Fp:q	Specifies the source1 operand for the add/subtract operation
Fr:s	Specifies the source2 operand for the add/subtract operation

Table 18-27: Source Register Encoding (64-bit float)

Opcode	Multi-Function Multiplier Source 1	Multi-Function Multiplier Source 2	Multi-Function Dual Add/Subtract Source 1	Multi-Function Dual Add/Subtract Source 2
	Fx:y	Fz:w	Fp:q	Fr:s
00	F1:0	F5:4	F9:8	F13:12
01	–	–	–	–
10	F3:2	F7:6	F11:10	F15:14
11	–	–	–	–

Table 18-28: Result Register Encoding (64-bit float)

Opcode	Single Computation Source/Destination		Multi–Computation Destination	
	Fm:n/Fx:y/Fz:w	Fn/Fx/Fy/Fz	Rn/Rx/Ry/Rz	Fm:n/Fa:b
0000	F1:0	F0	R0	F1:0
0001	–	F1	R1	–
0010	F3:2	F2	R2	F3:2
0011	–	F3	R3	–
0100	F5:4	F4	R4	F5:4
0101	–	F5	R5	–
0110	F7:6	F6	R6	F7:6
0111	–	F7	R7	–
1000	F9:8	F8	R8	F9:8

Table 18-28: Result Register Encoding (64-bit float) (Continued)

Opcode	Single Computation Source/Destination		Multi–Computation Destination	
	Fm:n/Fx:y/Fz:w	Fn/Fx/Fy/Fz	Rn/Rx/Ry/Rz	Fm:n/Fa:b
1001	–	F9	R9	–
1010	F11:10	F10	R10	F11:10
1011	–	F11	R11	–
1100	F13:12	F12	R12	F13:12
1101	–	F13	R13	–
1110	F15:14	F14	R14	F15:14
1111	–	F15	R15	–

MR Register Data Move (MRDATAMOVE)

The following table indicates how the opcode specifies the MR register, and RN specifies the data register. D-bit determines the direction of the transfer (0 = to register file, 1 = to MR register).

Table 18-29: MRDATAMOVE Encode Table

D-bit[16]	opcode[15:12]	Syntax	Instruction
x	0000	RN = MR0F/MR0F= RN	FXP_MR_ST
x	0001	RN = MR1F/MR1F= RN	
x	0010	RN = MR2F/MR2F= RN	
x	0100	RN = MR0B/MR0B= RN	
x	0101	RN = MR1B/MR1B= RN	
x	0110	RN = MR2B/MR2B= RN	

19 ALU Fixed-Point Computations

This section describes the ALU Fixed-point operations (FXP_). For all of the instructions in this section, the status flag AF bit is cleared (=0) indicating fixed-point operation. Note that the CACC flag bits are only set for the compare instructions, otherwise they have no effect.

For information on syntax and opcodes, see Compute (Compute) Opcode.

For information on arithmetic status, see the "Register Descriptions".

RN = RX + RY;

General Form

Compute (Compute) Opcode
RREG Register Class = RREG Register Class + RREG Register Class

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 `REGF_MODE1.ALUSAT` bit is set) positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	Sticky indicator for AV bit set
AIS	No effect
AVS	No effect

RN = RX – RY;

General Form

Compute (Compute) Opcode
RREG Register Class = RREG Register Class - RREG Register Class

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 `REGF_MODE1.ALUSAT` bit is set) positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	Sticky indicator for AV bit set
AIS	No effect
AVS	No effect

RN = RX + RY + ci;

General Form

Compute (Compute) Opcode
RREG Register Class = RREG Register Class + RREG Register Class + 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 `REGF_MODE1.ALUSAT` bit is set) positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	Sticky indicator for AV bit set
AIS	No effect
AVS	No effect

RN = RX – RY + ci – 1;

General Form

Compute (Compute) Opcode
RREG Register Class = RREG Register Class - RREG Register Class + 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 REGF_MODE1.ALUSAT bit is set) positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	Sticky indicator for AV bit set
AIS	No effect
AVS	No effect

RN = (RX + RY) / 2;

General Form

Compute (Compute) Opcode
RREG Register Class = (RREG Register Class + RREG Register Class) / 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 REGF_MODE1.RND32 (rounding mode) bit.

ASTATx/y Flags

AC	Set if the carry from the most significant adder stage is 1, otherwise cleared
AI	Cleared
AN	Set if the most significant output bit is 1, otherwise cleared
AS	Cleared
AV	Cleared

AZ	Set if the fixed-point output is all 0s, otherwise cleared

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	No effect
AVS	No effect

comp (RX, RY);

General Form

Compute (Compute) Opcode
comp(RREG Register Class, RREG Register Class)

Function

Compares the signed fixed-point field in register Rx with the fixed-point field in register Ry. Sets the AZ flag if the two operands are equal, and the AN flag if the operand in register Rx is smaller than the operand in register Ry. The ASTAT register stores the results of the previous eight ALU compare operations in CACC bits 3124. These bits are shifted right (bit 24 is overwritten) whenever a fixed-point or floating-point compare instruction is executed.

ASTATx/y Flags

AC	Cleared
AI	Cleared
CACC	The MSB bit of CACC is set if the X operand is greater than the Y operand (its value is the AND of AZ and AN); otherwise cleared
AN	Set if the signed operand in the Rx register is smaller than the operand in the Ry register, otherwise cleared
AS	Cleared
AV	Cleared
AZ	Set if the signed operands in registers Rx and Ry are equal, otherwise cleared

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	No effect

AVS	No effect

compu (RX, RY);

General Form

Compute (Compute) Opcode
compu(RREG Register Class, RREG Register Class)

Function

Compares the unsigned fixed-point field in register Rx with the fixed-point field in register Ry, Sets the AZ flag if the two operands are equal, and the AN flag if the operand in register Rx is smaller than the operand in register Ry. This operation performs a magnitude comparison of the fixed-point contents of Rx and Ry. The ASTAT register stores the results of the previous eight ALU compare operations in CACC bits 3124. These bits are shifted right (bit 24 is overwritten) whenever a fixed-point or floating-point compare instruction is executed.

ASTATx/y Flags

AC	Cleared
AI	Cleared
CACC	The MSB bit of CACC is set if the X operand is greater than the Y operand (its value is the AND of AZ and AN); otherwise cleared
AN	Set if the unsigned operand in the Rx register is smaller than the operand in the Ry register, otherwise cleared
AS	Cleared
AV	Cleared
AZ	Set if the unsigned operands in registers Rx and Ry are equal, otherwise cleared

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	No effect
AVS	No effect

RN = RX + ci;

General Form

Compute (Compute) Opcode

RREG Register Class = RREG Register Class + 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 REGF_MODE1.ALUSAT bit is set) positive overflows return the maximum positive number (0x7FFF FFFF).

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	Sticky indicator for AV bit set
AIS	No effect
AVS	No effect

RN = RX + ci – 1;

General Form

Compute (Compute) Opcode
RREG Register Class = RREG Register Class + 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 REGF_MODE1.ALUSAT bit is set) positive overflows return the maximum positive number (0x7FFF FFFF).

ASTATx/y Flags

AC	Set if the carry from the most significant adder stage is 1, otherwise cleared

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

STKYx/y Flags

AUS	No effect
AOS	Sticky indicator for AV bit set
AIS	No effect
AVS	No effect

RN = RX + 1;

General Form

Compute (Compute) Opcode
RREG Register Class = RREG Register Class + 1
Short Compute (ShortCompute) Opcode
RREG Register Class = RREG Register Class + 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 `REGF_MODE1.ALUSAT` bit is set), overflow causes the maximum positive number (0x7FFF FFFF) to be returned.

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	Sticky indicator for AV bit set
AIS	No effect
AVS	No effect

RN = RX – 1;

General Form

Compute (Compute) Opcode
RREG Register Class = RREG Register Class - 1
Short Compute (ShortCompute) Opcode
RREG Register Class = RREG Register Class - 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 `REGF_MODE1.ALUSAT` bit is set), underflow causes the minimum negative number (0x8000 0000) to be returned.

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	Sticky indicator for AV bit set
AIS	No effect
AVS	No effect

RN = –RX;

General Form

Compute (Compute) Opcode
RREG Register Class = - RREG Register Class

Function

Negates the fixed-point operand in Rx by two's-complement. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. Negation of the minimum negative number (0x8000 0000) causes an overflow. In saturation mode (the `REGF_MODE1.ALUSAT` bit is set), overflow causes the maximum positive number (0x7FFF FFFF) to be returned.

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	Sticky indicator for AV bit set
AIS	No effect
AVS	No effect

RN = abs RX;

General Form

Compute (Compute) Opcode
RREG Register Class = abs RREG Register Class

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 REGF_MODE1.ALUSAT bit is set), overflow causes the maximum positive number (0x7FFF FFFF) to be returned.

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	Sticky indicator for AV bit set
AIS	No effect
AVS	No effect

RN = pass RX;

General Form

Compute (Compute) Opcode
RREG Register Class = pass RREG Register Class
Short Compute (ShortCompute) Opcode
RREG Register Class = pass RREG Register Class

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.

ASTATx/y Flags

AC	Cleared
AI	Cleared
AN	Set if the most significant output bit is 1, otherwise cleared
AS	Cleared

AV	Cleared
AZ	Set if the fixed-point output is all 0s, otherwise cleared

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	No effect
AVS	No effect

RN = RX and RY;

General Form

Compute (Compute) Opcode
RREG Register Class = RREG Register Class and RREG Register Class

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.

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	No effect
AVS	No effect

RN = RX or RY;

General Form

Compute (Compute) Opcode
RREG Register Class = RREG Register Class or RREG Register Class

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.

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	No effect
AVS	No effect

RN = RX xor RY;

General Form

Compute (Compute) Opcode
RREG Register Class = RREG Register Class xor RREG Register Class

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.

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	No effect
AVS	No effect

RN = not RX;

General Form

Compute (Compute) Opcode
RREG Register Class = not RREG Register Class
Short Compute (ShortCompute) Opcode
RREG Register Class = not RREG Register Class

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.

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	No effect
AVS	No effect

RN = min (RX, RY);

General Form

Compute (Compute) Opcode
RREG Register Class = min(RREG Register Class, RREG Register Class)

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.

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	No effect
AVS	No effect

RN = max (RX, RY);

General Form

Compute (Compute) Opcode

RREG Register Class = max(RREG Register Class, RREG Register Class)

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.

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	No effect
AVS	No effect

RN = clip RX by RY;

General Form

Compute (Compute) Opcode
RREG Register Class = clip RREG Register Class by RREG Register Class

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.

ASTATx/y Flags

AC	Cleared
AI	Cleared

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

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	No effect
AVS	No effect

20 ALU Floating-Point Computations

This section describes the 32/40/64-bit ALU floating-point operations. For all of the instructions is this section, the status flag AF bit is set (=1) indicating floating-point operation. Note that the CACC flag bits are only set for the compare instructions, otherwise they have no effect.

For information on syntax and opcodes, see Compute (Compute) Opcode.

For information on arithmetic status, see the "Register Descriptions" chapters

32-bit and 40-bit Operations

The following sections provide descriptions for the 32-bit and 40-bit operations.

FN = FX + FY;

General Form

Compute (Compute) Opcode
FREG Register Class = FREG Register Class + FREG Register Class

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.

REGF_ASTATX/REGF_ASTATY Flags

AC	Cleared
AI	Set if either of the input operands is a NAN, or if they are opposite-signed infinities, otherwise cleared
AN	Set if the floating-point result is negative, otherwise cleared

AS	Cleared
AV	Set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared
AZ	Set if the post-rounded result is a denormal (unbiased exponent < -126) or zero, otherwise cleared

STKYx/y Flags

AUS	Sticky indicator for AZ bit set
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	Sticky indicator for AV bit set

FN = FX – FY;

General Form

Compute (Compute) Opcode
FREG Register Class = FREG Register Class - FREG Register Class

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 (`REGF_MODE1.TRUNCATE`) and rounding boundary (`REGF_MODE1.RND32`) bits. 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.

ASTATx/y Flags

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

STKYx/y Flags

AUS	Sticky indicator for AZ bit set
AOS	No effect
AIS	Sticky indicator for AI bit set

AVS	Sticky indicator for AV bit set

FN = abs (FX + FY);

General Form

Compute (Compute) Opcode
FREG Register Class = abs (FREG Register Class + FREG Register Class)

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 rounding mode (`REGF_MODE1.TRUNCATE`) and rounding boundary (`REGF_MODE1.RND32`) bits. 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.

ASTATx/y Flags

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

STKYx/y Flags

AUS	Sticky indicator for AZ bit set
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	Sticky indicator for AV bit set

FN = abs (FX – FY);

General Form

Compute (Compute) Opcode
FREG Register Class = abs (FREG Register Class - FREG Register Class)

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 rounding mode (`REGF_MODE1.TRUNCATE`) and rounding boundary (`REGF_MODE1.RND32`) bits. 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.

ASTATx/y Flags

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

STKYx/y Flags

AUS	Sticky indicator for AZ bit set
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	Sticky indicator for AV bit set

FN = (FX + FY) / 2;

General Form

Compute (Compute) Opcode
FREG Register Class = (FREG Register Class + FREG Register Class) / 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 (`REGF_MODE1.TRUNCATE`) and rounding boundary (`REGF_MODE1.RND32`) bits. 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.

ASTATx/y Flags

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

STKYx/y Flags

AUS	Sticky indicator for AZ bit set
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	No effect

comp (FX, FY);

General Form

Compute (Compute) Opcode
comp(FREG Register Class, FREG Register Class)

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 `REGF_ASTATX`/`REGF_ASTATY` register stores the results of the previous eight ALU compare operations in CACC bits 31 through 24. These bits are shifted right (bit 24 is overwritten) whenever a fixed-point or floating-point compare instruction is executed.

ASTATx/y Flags

AC	Cleared
AI	Set if either of the input operands is a NAN, otherwise cleared
CACC	The MSB of CACC is set if the X operand is greater than the Y operand (its value is the AND of AZ and AN); otherwise cleared
AN	Set if the operand in the Fx register is smaller than the operand in the Fy register, otherwise cleared
AS	Cleared
AV	Cleared

AZ	Set if the operands in registers Fx and Fy are equal, otherwise cleared

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	No effect

FN = –FX;

General Form

Compute (Compute) Opcode
FREG Register Class = - FREG Register Class

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.

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	No effect

FN = abs FX;

General Form

Compute (Compute) Opcode
FREG Register Class = abs FREG Register Class

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.

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	No effect

FN = pass FX;

General Form

Compute (Compute) Opcode
FREG Register Class = pass FREG Register Class

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.

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	No effect

FN = rnd FX;

General Form

Compute (Compute) Opcode
FREG Register Class = rnd FREG Register Class

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 `REGF_MODE1.RND32` bit. 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.

ASTATx/y Flags

AC	Cleared
AI	Set if the input operand is a NAN, otherwise cleared
AN	Set if the floating-point result is negative, otherwise cleared
AS	Cleared
AV	Set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared
AZ	Set if the result operand is a ±zero, otherwise cleared

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	Sticky indicator for AV bit set

FN = scalb FX by RY;

General Form

Compute (Compute) Opcode
FREG Register Class = scalb FREG Register Class by RREG Register Class

Function

Scales the exponent of the floating-point operand in Fx by adding to it the fixed-point two's-complement integer in Ry. The scaled floating-point result is placed in register Fn. Overflow returns ±infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). Denormal returns ±zero. Denormal inputs are flushed to ±zero. A NAN input returns an all 1s result.

ASTATx/y Flags

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

STKYx/y Flags

AUS	Sticky indicator for AZ bit set
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	Sticky indicator for AV bit set

RN = mant FX;

General Form

Compute (Compute) Opcode
RREG Register Class = mant FREG Register Class

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).

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	Sticky indicator for AV bit set

RN = logb FX;

General Form

Compute (Compute) Opcode
RREG Register Class = logb FREG Register Class

Function

Converts the exponent of the floating-point operand in register Fx to an unbiased two's-complement fixed-point integer. The result is placed in the fixed-point field in register Rn. Unbiasing is done by subtracting 127 from the floating-point exponent in Fx. If saturation mode (`REGF_MODE1.ALUSAT`) 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.

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	Sticky indicator for AV bit set

RN = fix FX;

General Form

Compute (Compute) Opcode
RREG Register Class = fix FREG Register Class

Function

Converts the floating-point operand in Fx to a two's-complement 32-bit fixed-point integer result. If the REGF_MODE1.TRUNCATE bit =1, the Fix operation truncates the mantissa towards infinity. If the REGF_MODE1.TRUNCATE bit =0, the Fix operation rounds the mantissa towards the nearest integer. The trunc operation always truncates toward 0. The REGF_MODE1.TRUNCATE bit does not influence operation of the trunc instruction. 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 REGF_MODE1.ALUSAT bit is 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 REGF_MODE1.RND32 bit. 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).

ASTATx/y Flags

AC	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
AN	Set if the fixed-point result is negative, otherwise cleared
AS	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
AZ	Set if the fixed-point result is zero, otherwise cleared

STKYx/y Flags

AUS	Sticky indicator Set if the pre-rounded result is between -1.0 and 1.0 (except -1, 1, 0), otherwise not effected

RN = fix FX by RY;

General Form

Compute (Compute) Opcode
RREG Register Class = fix FREG Register Class by RREG Register Class

Function

Converts the floating-point operand in Fx to a two's-complement 32-bit fixed-point integer result. If the REGF_MODE1.TRUNCATE bit =1, the Fix operation truncates the mantissa towards infinity. If the REGF_MODE1.TRUNCATE bit =0, the Fix operation rounds the mantissa towards the nearest integer. The trunc operation always truncates toward 0. The REGF_MODE1.TRUNCATE bit does not influence operation of the trunc instruction. A scaling factor (Ry) is specified, the fixed-point two's-complement integer in Ry is added to the exponent of the floating-point operand in Fx before the conversion. The result of the conversion is right-justified (32.0 format) in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s.

In saturation mode (the REGF_MODE1.ALUSAT bit is 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 REGF_MODE1.RND32 bit. 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).

ASTATx/y Flags

	Cleared

AC	
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
AN	Set if the fixed-point result is negative, otherwise cleared
AS	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
AZ	Set if the fixed-point result is zero, otherwise cleared

STKYx/y Flags

AUS	Sticky indicator Set if the pre-rounded result is between -1.0 and 1.0 (except -1, 1, 0), otherwise not effected

RN = trunc FX;

General Form

Compute (Compute) Opcode
RREG Register Class = trunc FREG Register Class

Function

Converts the floating-point operand in Fx to a two's-complement 32-bit fixed-point integer result. If the `REGF_MODE1.TRUNCATE` bit =1, the Fix operation truncates the mantissa towards -infinity. If the `REGF_MODE1.TRUNCATE` bit =0, the Fix operation rounds the mantissa towards the nearest integer. The trunc operation always truncates toward 0. The `REGF_MODE1.TRUNCATE` bit does not influence operation of the trunc instruction. 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 `REGF_MODE1.ALUSAT` bit is 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 `REGF_MODE1.RND32` bit. 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).

ASTATx/y Flags

AC	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

AN	Set if the fixed-point result is negative, otherwise cleared
AS	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
AZ	Set if the fixed-point result is zero, otherwise cleared

STKYx/y Flags

AUS	Sticky indicator Set if the pre-rounded result is between -1.0 and 1.0 (except -1, 1, 0), otherwise not effected

RN = trunc FX by RY;

General Form

Compute (Compute) Opcode
RREG Register Class = trunc FREG Register Class by RREG Register Class

Function

Converts the floating-point operand in Fx to a two's-complement 32-bit fixed-point integer result. If the REGF_MODE1.TRUNCATE bit =1, the Fix operation truncates the mantissa towards infinity. If the REGF_MODE1.TRUNCATE bit =0, the Fix operation rounds the mantissa towards the nearest integer. The trunc operation always truncates toward 0. The REGF_MODE1.TRUNCATE bit does not influence operation of the trunc instruction. A scaling factor (Ry) is specified, the fixed-point two's-complement integer in Ry is added to the exponent of the floating-point operand in Fx before the conversion. The result of the conversion is right-justified (32.0 format) in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s.

In saturation mode (the REGF_MODE1.ALUSAT bit is 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).

ASTATx/y Flags

AC	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
AN	Set if the fixed-point result is negative, otherwise cleared
AS	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
AZ	Set if the fixed-point result is zero, otherwise cleared

STKYx/y Flags

AUS	Sticky indicator Set if the pre-rounded result is between -1.0 and 1.0 (except -1, 1, 0), otherwise not effected

FN = float RX;

General Form

Compute (Compute) Opcode
FREG Register Class = float RREG Register Class

Function

Converts the fixed-point operand in Rx to a 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.

ASTATx/y Flags

AC	Cleared
AI	Cleared
AN	Set if the floating-point result is negative, otherwise cleared
AS	Cleared
AV	Cleared
AZ	Set if the result is an unbiased exponent < -126, or zero, otherwise cleared

STKYx/y Flags

AUS	Sticky indicator for AZ bit set
AOS	No effect
AIS	No effect
AVS	Sticky indicator for AV bit set

FN = float RX by RY;

General Form

Compute (Compute) Opcode
FREG Register Class = float RREG Register Class by RREG Register Class

Function

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

ASTATx/y Flags

AC	Cleared
AI	Cleared
AN	Set if the floating-point result is negative, otherwise cleared
AS	Cleared
AV	Cleared
AZ	Set if the result is an unbiased exponent < -126, or zero, otherwise cleared

STKYx/y Flags

AUS	Sticky indicator for AZ bit set
AOS	No effect
AIS	No effect
AVS	Sticky indicator for AV bit set

FN = recips FX;

General Form

Compute (Compute) Opcode
FREG Register Class = recips FREG Register Class

Function

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

The following code performs floating-point division using an iterative convergence algorithm.[i] The result is accurate to one LSB in whichever format mode, 32-bit or 40-bit, is set. The following inputs are required: F0=numerator and F12=denominator, F11=2.0. The quotient is returned in F0. (The two indented instructions can be removed if only a ±1 LSB accurate single-precision result is necessary.) Note that, in the algorithm example's comments, references to R0, R1, R2, and R3 do not refer to data registers. Rather, they refer to variables in the algorithm.

```
F0=RECIPS F12, F7=F0;      /* Get 8 bit seed R0=1/D */
F12=F0*F12;                /* D' = D*R0 */
F7=F0*F7, F0=F11-F12;      /* F0=R1=2-D', F7=N*R0 */
F12=F0*F12;                /* F12=D'-D'*R1 */
F7=F0*F7, F0=F11-F12;      /* F7=N*R0*R1, F0=R2=2-D' */
  F12=F0*F12;              /* F12=D'=D'*R2 */
  F7=F0*F7, F0=F11-F12;    /* F7=N*R0*R1*R2, F0=R3=2-D' */
F0=F0*F7;                  /* F7=N*R0*R1*R2*R3 */
```

To make this code segment a subroutine, add an RTS(DB) clause to the third-to-last instruction.

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

ASTATx/y Flags

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

STKYx/y Flags

AUS	Sticky indicator for AZ bit set
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	Sticky indicator for AV bit set

FN = rsqrts FX;

General Form

Compute (Compute) Opcode
FREG Register Class = rsqrts FREG Register Class

Function

Creates a 4-bit accurate seed for 1√(Fx)$^{1/2}$, the reciprocal square root of Fx. The mantissa of the seed is determined from a ROM table, using the LSB of the biased exponent of Fx concatenated with the six MSBs (excluding the hidden bit of the mantissa) of Fx's index. The unbiased exponent of the seed is calculated as the two's-complement of the unbiased Fx exponent, shifted right by one bit and decremented by one; that is, if e is the unbiased exponent of Fx, then the unbiased exponent of 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)$^{1/2}$) using a Newton-Raphson iteration algorithm.1 The result is accurate to one LSB in whichever format mode, 32-bit or 40-bit, is set. 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 indented instructions can be removed if only a ±1 LSB accurate single-precision result is necessary.)

```
F4=RSQRTS F0;              /* Fetch 4-bit seed */
F12=F4*F4;                 /* F12=X0^2 */
F12=F12*F0;                /* F12=C*X0^2 */
F4=F1*F4, F12=F8-F12;      /* F4=.5*X0, F12=3-C*X0^2 */
F4=F4*F12;                 /* F4=X1=.5*X0(3-C*X0^2) */
F12=F4*F4;                 /* F12=X1^2 */
```

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.

ASTATx/y Flags

AC	Cleared
AI	Set if the input operand is negative and nonzero, or a NAN, otherwise cleared
AN	Set if the input operand is -zero, otherwise cleared

AS	Cleared
AV	Set if the input operand is ±zero, otherwise cleared
AZ	Set if the floating-point result is +zero (Fx = +infinity), otherwise cleared

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	Sticky indicator for AV bit set

FN = FX copysign FY;

General Form

Compute (Compute) Opcode
FREG Register Class = FREG Register Class copysign FREG Register Class

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.

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	No effect

FN = min (FX, FY);

General Form

Compute (Compute) Opcode
FREG Register Class = min(FREG Register Class, FREG Register Class)

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.

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	No effect

FN = max (FX, FY);

General Form

Compute (Compute) Opcode
FREG Register Class = max(FREG Register Class, FREG Register Class)

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.

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	No effect

FN = clip FX by FY;

General Form

Compute (Compute) Opcode
FREG Register Class = clip FREG Register Class by FREG Register Class

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.

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	No effect

64-bit Floating-Point Computations

This section describes the 64-bit floating-point operations.

FM:N = FX:Y + FZ:W;

General Form

Compute (Compute) Opcode
DBLREG Register Type = DBLREG Register Type + DBLREG Register Type

Function

Adds the floating-point operands in register pairs Fx:y and Fz:w. The normalized result is placed in register Fm:n.

Rounding is to nearest (IEEE) or by truncation, as defined by the TRUNC bit 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.

This operation requires seven execution cycles. The destination register Fm:n and status registers (ASTATx/y or STKYx/y) get updated at the end of 7th execution cycle.

ASTATx/y Flags

AC	Cleared
AI	Set if either of the input operands is a NAN, or if they are opposite-signed Infinities, otherwise cleared
AN	Set if the floating-point result is negative, otherwise cleared
AS	Cleared
AV	Set if the post-rounded result overflows (unbiased exponent > +1023), otherwise cleared
AZ	Set if the post-rounded result is a denormal (unbiased exponent < –1022) or Zero, otherwise cleared

STKYx/y Flags

AUS	Set if the post-rounded result is a denormal (unbiased exponent < –1022)
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	Sticky indicator for AV bit set

FM:N = FX:Y - FZ:W;

General Form

Compute (Compute) Opcode
DBLREG Register Type = DBLREG Register Type - DBLREG Register Type

Function

Subtracts the floating-point operand in register pair Fz:w from the floating-point operand in register pair Fx:y. The normalized result is placed in register pair Fm:n.

Rounding is to nearest (IEEE) or by truncation, 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.

This operation requires seven execution cycles. The destination register Fm:n and status registers (ASTATx/y or STKYx/y) get updated at the end of 7th execution cycle.

ASTATx/y Flags

AC	Cleared
AI	Set if either of the input operands is a NAN, or if they are opposite-signed Infinities, otherwise cleared
AN	Set if the floating-point result is negative, otherwise cleared
AS	Cleared
AV	Set if the post-rounded result overflows (unbiased exponent > +1023), otherwise cleared
AZ	Set if the post-rounded result is a denormal (unbiased exponent < –1022) or Zero, otherwise cleared

STKYx/y Flags

AUS	Sticky indicator sets if the post-rounded result is a denormal (unbiased exponent < –1022)
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	Sticky indicator for AV bit set

comp (FX:Y, FZ:W);

General Form

Compute (Compute) Opcode
comp(DBLREG Register Type, DBLREG Register Type)

Function

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

The ASTAT register stores the results of the previous eight ALU compare operations in the CACC bits 31–24. These bits are shifted right (bit 24 is overwritten) whenever a fixed-point or floating-point compare instruction is executed.

This operation requires seven execution cycles. The status registers (ASTATx/y or STKYx/y) get updated at the end of 7th execution cycle.

ASTATx/y Flags

AC	Cleared
AI	Set if either of the input operands is a NAN, otherwise cleared
AN	Set if the operand in the Fx:y register is smaller than the operand in the Fz:w register, otherwise cleared
AS	Cleared
AV	Cleared
AZ	Set if the operands in registers Fx:y and Fz:w are equal, otherwise cleared
CACC	The MSB of CACC is set if the X operand is greater than the Y operand; otherwise cleared

STKYx/y Flags

AUS	No effect
AOS	No effect

AIS	Sticky indicator for AI bit set
AVS	No effect

FM:N = - FX:Y;

General Form

Compute (Compute) Opcode
DBLREG Register Type = - DBLREG Register Type

Function

Complements the sign bit of the floating-point operand in Fx:y. The complemented result is placed in register Fm:n.

A denormal input is flushed to ±zero.

A NAN input returns an all 1s result.

This operation requires 2 execution cycles. The destination register Fm:n and status registers (ASTATx/y or STKYx/y) get updated at the end of 2nd execution cycle.

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	No effect

FM:N = abs FX:Y;

General Form

Compute (Compute) Opcode
DBLREG Register Type = abs DBLREG Register Type

Function

Returns the absolute value of the floating-point operand in register Fx:y by setting the sign bit of the operand to 0.

A denormal input is flushed to ±zero.

A NAN input returns an all 1s result.

This operation requires 2 execution cycles. The destination register Fm:n and status registers (ASTATx/y or STKYx/y) get updated at the end of 2nd execution cycle.

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	No effect

FM:N = pass FX:Y;

General Form

Compute (Compute) Opcode
DBLREG Register Type = pass DBLREG Register Type

Function

Passes the floating-point operand in Fx:y through the ALU to the floating point register Fm:n.

A denormal input is flushed to ±zero.

A NAN input returns an all 1s result.

This operation requires 2 execution cycles. The destination register Fm:n and status registers (ASTATx/y or STKYx/y) get updated at the end of 2nd execution cycle.

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	No effect

FM:N = scalb FX:Y by RY;

General Form

Compute (Compute) Opcode
DBLREG Register Type = scalb DBLREG Register Type by RREG Register Class

Function

Scales the exponent of the floating-point operand in Fx:y by adding to it the fixed-point two's complement integer in Ry. The scaled floating point result is placed in register Fm:n.

A denormal input is flushed to ±zero.

A NAN input returns an all 1s result.

An infinite input results into infinite output of same sign.

Rounding is to nearest (IEEE) or by truncation, as defined by the rounding mode.

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 (denormal) returns ±zero.

This operation requires 2 execution cycles. The destination register Fm:n and status registers (ASTATx/y or STKYx/y) get updated at the end of 2nd execution cycle.

ASTATx/y Flags

AC	Cleared
AI	Set if the input is a NAN, otherwise cleared
AN	Set if the floating-point result is negative, otherwise cleared
AS	Cleared
AV	Set if the post rounded result overflows (unbiased exponent > 1023), otherwise cleared
AZ	Set if the post rounded result is a denormal (unbiased exponent < –1022) or zero, otherwise cleared

STKYx/y Flags

AUS	Sticky indicator sets if the post-rounded result is a denormal, otherwise cleared
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	Sticky indicator for AV bit set

RN=fix FX:Y;

DPFLP_FIX

General Form

Compute (Compute) Opcode
RREG Register Class = fix DBLREG Register Type
Rn = Fix Fx:y

Function

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

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.

If the `REGF_MODE1.TRUNCATE` bit =0, the Fix operation rounds the mantissa towards the nearest integer.

In saturation mode (the `REGF_MODE1.ALUSAT` bit =1), positive overflows and +infinity return the maximum positive number (0x7FFF FFFF), and negative overflows and –infinity return the minimum negative number (0x8000 0000). 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 or rounded-to-zero, return zero; and negative underflows that are rounded by truncation return –1 (0xFFFF FFFF).

A denormal input is flushed to ±zero.

A NAN input returns a floating point all 1s result, (0xFFFFFFFF or 0xFFFFFFFFFF) depending on the `REGF_MODE1.RND32` bit value..

This operation requires four execution cycles. The destination register Fm:n and status registers (ASTATx/y or STKYx/y) get updated at the end of 4th execution cycle.

ASTATx/y Flags

AC	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
AN	Set if the fixed-point result is negative, otherwise cleared
AS	Cleared
AV	Set if the pre-rounded result is outside the 2's complement signed 32-bit integer range, or if the input is ±infinity, otherwise cleared
AZ	Set if the fixed-point result is zero, otherwise cleared

STKYx/y Flags

AUS	Sticky indicator sets if the pre-rounded result is between -1.0 and 1.0 (except -1, 1, 0), otherwise not effected
AOS	No effect
AIS	Sticky indicator for AI bit set
AVS	Sticky indicator for AV bit set

RN = fix FX:Y by RY;

General Form

Compute (Compute) Opcode
RREG Register Class = fix DBLREG Register Type by RREG Register Class

Function

Converts the floating-point operand in Fx:y to a two's-complement 32-bit fixed-point integer result. If a scaling factor (Ry) is specified, the fixed-point two's-complement integer in Ry is added to the exponent of the floating-point operand in Fx:y 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.

If the REGF_MODE1.TRUNCATE bit =1, the "Fix" operation truncates the mantissa towards –infinity. If this bit =0, the "Fix" operation rounds the mantissa towards the nearest integer. The truncate operation always truncates toward 0. The REGF_MODE1.TRUNCATE bit does not influence operation of the trunc instruction.

In saturation mode (the REGF_MODE1.ALUSAT bit =1), positive overflows and +infinity return the maximum positive number (0x7FFF FFFF), and negative overflows and –infinity return the minimum negative number (0x8000 0000). 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 or rounded-to-zero, return zero; and negative underflows that are rounded by truncation return –1 (0xFFFF FFFF).

A denormal input is flushed to ±zero.

A NAN input returns a floating point all 1s result, (0xFFFFFFFF or 0xFFFFFFFFFF) depending on the REGF_MODE1.RND32 bit value.

This operation requires four execution cycles. The destination register Fm:n and status registers (ASTATx/y or STKYx/y) get updated at the end of 4th execution cycle.

ASTATx/y Flags

AC	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
AN	Set if the fixed-point result is negative, otherwise cleared
AS	Cleared
AV	Set if the pre-rounded result is outside the 2's complement signed 32-bit integer range, or if the input is ±infinity, otherwise cleared
AZ	Set if the fixed-point result is zero, otherwise cleared

STKYx/y Flags

AUS	Sticky indicator sets if the pre-rounded result is between -1.0 and 1.0 (except -1, 1, 0), otherwise not effected
AOS	No effect
AIS	Sticky indicator for AI bit set This operation requires four execution cycles
AVS	Sticky indicator for AV bit set

RN = trunc FX:Y;

General Form

Compute (Compute) Opcode

RREG Register Class = trunc DBLREG Register Type

Function

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

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.

When the `REGF_MODE1.TRUNCATE` bit =1, the Fix operation truncates the mantissa towards –infinity. The truncate operation always truncates toward 0. The `REGF_MODE1.TRUNCATE` bit does not influence operation of the trunc instruction.

In saturation mode (the `REGF_MODE1.ALUSAT` bit =1), positive overflows and +infinity return the maximum positive number (0x7FFF FFFF), and negative overflows and –infinity return the minimum negative number (0x8000 0000). 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 or rounded-to-zero, return zero; and negative underflows that are rounded by truncation return –1 (0xFFFF FFFF).

A denormal input is flushed to ±zero.

A NAN input returns a floating point all 1s result, (0xFFFFFFFF or 0xFFFFFFFFFF) depending on the `REGF_MODE1.RND32` bit value.

This operation requires four execution cycles. The destination register Fm:n and status registers (ASTATx/y or STKYx/y) get updated at the end of 4th execution cycle.

ASTATx/y Flags

AC	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
AN	Set if the fixed-point result is negative, otherwise cleared
AS	Cleared
AV	Set if the pre-rounded result is outside the 2's complement signed 32-bit integer range, or if the input is ±infinity, otherwise cleared
AZ	Set if the fixed-point result is zero, otherwise cleared

STKYx/y Flags

AUS	Sticky indicator sets if the pre-rounded result is between -1.0 and 1.0 (except -1, 1, 0), otherwise not effected
AOS	No effect

AIS	Sticky indicator for AI bit set
AVS	Sticky indicator for AV bit set

RN = trunc FX:Y by RY;

DPFLP_TRUNC_BY

General Form

Compute (Compute) Opcode
RREG Register Class = trunc DBLREG Register Type by RREG Register Class

Function

Converts the floating-point operand in Fx:y to a two's-complement 32-bit fixed-point integer result. If a scaling factor (Ry) is specified, the fixed-point two's-complement integer in Ry is added to the exponent of the floating-point operand in Fx:y 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.

If the MODE1 register TRUNC bit is 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.

In saturation mode (the ALUSAT 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). 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 or rounded-to-zero, return zero; and negative underflows that are rounded by truncation return -1 (0xFFFF FFFF).

A denormal input is flushed to ±zero.

A NAN input returns a floating point all 1s result, i.e. 0xFFFFFFFF or 0xFFFFFFFFFF depending on RND32 bit value in MODE1 register.

The destination register Fm:n and status registers (ASTATx/y or STKYx/y) get updated at the end of 4th execution cycle.

ASTATx/y Flags

AC	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
AN	Set if the fixed-point result is negative, otherwise cleared
AS	Cleared
AV	Set if the pre-rounded result is outside the 2's complement signed 32-bit integer range, or if the input is ±infinity, otherwise cleared
AZ	Set if the fixed-point result is zero, otherwise cleared

STKYx/y Flags

AUS	Sticky indicator sets if the pre-rounded result is between -1.0 and 1.0 (except -1, 1, 0), otherwise not effected
AOS	No effect
AIS	Sticky indicator for AI bit set This operation requires four execution cycles
AVS	Sticky indicator for AV bit set

FM:N = float RX;

General Form

Compute (Compute) Opcode
DBLREG Register Type = float RREG Register Class

Function

Converts the fixed-point operand in Rx to a 64-bit floating-point result.

The final result is placed in register Fm:n. Rounding is to nearest (IEEE) or by truncation, as defined by the rounding mode bit (REGF_MODE1.TRUNCATE). The exponent scale bias may cause a floating-point overflow or a floating-point underflow. Overflow produces ±infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). Underflow generates a return of ±zero.

This operation requires 2 execution cycles. The destination register Fm:n and status registers (ASTATx/y or STKYx/y) get updated at the end of 2nd execution cycle.

ASTATx/y Flags

AC	Cleared
AI	Cleared
AN	Set if the floating-point result is negative, otherwise cleared
AS	Cleared
AV	Cleared

AZ	Set if the result is zero, otherwise cleared

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	No effect
AVS	No effect

FM:N = float RX by RY;

General Form

Compute (Compute) Opcode
DBLREG Register Type = float RREG Register Class by RREG Register Class

Function

Converts the fixed-point operand in Rx to a Double Precision floating-point result, and then the fixed-point two's-complement integer in Ry is added to the exponent of the floating-point result. The final result is placed in register Fm:n.

Rounding is to nearest (IEEE) or by truncation, as defined by the rounding mode bit (`REGF_MODE1.TRUNCATE`).

The exponent scale bias may cause a floating-point overflow or a floating-point underflow.

Overflow produces ±infinity (round-to-nearest) or ±NORM.MAX (round-to-zero).

Underflow generates a return of ±zero.

This operation requires 4 execution cycles. The destination register Fm:n and status registers (ASTATx/y or STKYx/y) get updated at the end of 4th execution cycle.

ASTATx/y Flags

AC	Cleared
AI	Cleared
AN	Set if the floating-point result is negative, otherwise cleared
AS	Cleared
AV	Set if the result overflows (unbiased exponent > 1023), otherwise cleared
AZ	Set if the result is a denormal (unbiased exponent < –1022) or zero, otherwise cleared

STKYx/y Flags

AUS	Sticky indicator sets if the post-rounded result is a denormal, otherwise cleared
AOS	No effect
AIS	No effect
AVS	Sticky indicator for AV bit set

FM:N = cvt FX;

General Form

Compute (Compute) Opcode

Function

Converts the single precision floating point operand in Fx register to double precision floating point format. The converted result is placed in register Fm:n.

A denormal input is flushed to ±zero.

A NAN input returns an all 1s result.

An infinite input results into infinite output of same sign.

The input is treated as either 32-bit or 40-bit format, depending on the `REGF_MODE1.RND32` bit.

This operation takes 2 cycles. The destination register Fm:n and status registers (ASTATx/y or STKYx/y) get updated at the end of 2nd execution cycle.

ASTATx/y Flags

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

STKYx/y Flags

AUS	No effect
AOS	No effect
AIS	Sticky indicator for AI bit set

AVS	No effect

FN = cvt FX:Y;

General Form

Compute (Compute) Opcode

Function

Converts the double precision floating point operand in Fx:y register to single precision floating point format. The converted result is placed in register FN.

A denormal input is flushed to ±zero.

A NAN input returns an all 1s result.

An infinite input results into infinite output of same sign.

The output can be either 32-bit or 40-bit format, depending on the `REGF_MODE1.RND32` bit.

The `REGF_MODE1.TRUNCATE` specifies the rounding mode. If `REGF_MODE1.TRUNCATE` = 0, round to nearest and if `REGF_MODE1.TRUNCATE` = 1, round by truncation. Round to nearest, can result in incrementing exponent and hence may overflow single/extended precision range.

Post-rounded overflow returns ±Infinite, if `REGF_MODE1.TRUNCATE` = 0.

Post-rounded overflow returns ±NORM.MAX, if `REGF_MODE1.TRUNCATE` = 1.

Post-rounded underflows are denormal for single-precision format. They are rounded to zero and underflow flag should be set.

This operation requires 4 execution cycles. The destination register Fm:n and status registers (ASTATx/y or STKYx/y) get updated at the end of 4th execution cycle.

ASTATx/y Flags

AC	Cleared
AI	Set if the input operand is a NAN, otherwise cleared
AN	Set if the floating-point result is negative, otherwise cleared
AS	Cleared
AV	Set if the post-rounded result overflows (unbiased exponent > 127), otherwise cleared
AZ	Set if the result operand is a ±zero, otherwise cleared

STKYx/y Flags

AVS	Sticky indicator for AV bit set
AOS	No effect
AIS	Sticky indicator for AI bit set
AUS	Sticky indicator sets if the post-rounded result underflows (unbiased exponent <-126), otherwise cleared

21 MR Register Data Move Operations

This section describes the multiplier result (MR) register data move operations.

For information on syntax and opcodes, see Compute (Compute) Opcode.

For information on arithmetic status, see REGF_ASTATX and REGF_ASTATYregisters.

(mrf | mrb) = RN;

General Form

MRXFBREG Register Class = RREG Register Class

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.

ASTATx/y Flags

MU	Cleared
MN	Cleared
MI	Cleared
MV	Cleared

STKYx/y Flags

MOS	No effect
MIS	No effect
MVS	No effect
MUS	No effect

RN = (mrf | mrb);

General Form

RREG Register Class = MRXFBREG Register Class

Function

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.

ASTATx/y Flags

MU	Cleared
MN	Cleared
MI	Cleared
MV	Cleared

STKYx/y Flags

MOS	No effect
MIS	No effect
MVS	No effect
MUS	No effect

22 Multiplier Fixed-Point Computations

This section describes the multiplier operations. Data moves between MR registers and data registers are considered multiplier operations and are also covered.

Some of the instructions accept the following Mod1, Mod2, and Mod3 modifiers enclosed in parentheses and that consist of three or four letters that indicate whether:

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

For information on syntax and opcodes, see Compute (Compute) Opcode.

For information on arithmetic status, see REGF_ASTATX and REGF_ASTATY registers.

(mrf | mrb) = MRF + RX * RY MOD1;

General Form

Compute (Compute) Opcode
mrf = mrf + RREG Register Class * RREG Register Class MOD1
mrb = mrb + RREG Register Class * RREG Register Class MOD1

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.

ASTATx/y Flags

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
MN	Set if the result is negative, otherwise cleared
MI	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

STKYx/y Flags

MOS	Sticky indicator for MV bit set
MIS	No effect
MVS	No effect
MUS	MUS No effect

RN = (mrf | mrb) + RX * RY MOD1;

General Form

Compute (Compute) Opcode
RREG Register Class = mrf + RREG Register Class * RREG Register Class MOD1
RREG Register Class = mrb + RREG Register Class * RREG Register Class MOD1

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.

ASTATx/y Flags

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
MN	Set if the result is negative, otherwise cleared
MI	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

STKYx/y Flags

MOS	Sticky indicator for MV bit set
MIS	No effect
MVS	No effect
MUS	No effect

(mrf | mrb) = (mrf | mrb) – RX * RY MOD1;

General Form

Compute (Compute) Opcode
mrf = mrf - RREG Register Class * RREG Register Class MOD1
mrb = mrb - RREG Register Class * RREG Register Class MOD1

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.

ASTATx/y Flags

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
MN	Set if the result is negative, otherwise cleared
MI	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

STKYx/y Flags

MOS	Sticky indicator for MV bit set

MIS	No effect
MVS	No effect
MUS	No effect

RN = (mrf | mrb) – RX * RY MOD1;

General Form

Compute (Compute) Opcode
RREG Register Class = mrf - RREG Register Class * RREG Register Class MOD1
RREG Register Class = mrb - RREG Register Class * RREG Register Class MOD1

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.

ASTATx/y Flags

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
MN	Set if the result is negative, otherwise cleared
MI	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

STKYx/y Flags

MOS	Sticky indicator for MV bit set
MIS	No effect
MVS	No effect
MUS	No effect

(RN | mrf | mrb) = RX * RY MOD1;

General Form

Compute (Compute) Opcode
RREG Register Class = RREG Register Class * RREG Register Class MOD1
mrf = RREG Register Class * RREG Register Class MOD1
mrb = RREG Register Class * RREG Register Class MOD1

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.

ASTATx/y Flags

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
MN	Set if the result is negative, otherwise cleared
MI	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

STKYx/y Flags

MOS	Sticky indicator for MV bit set
MIS	No effect
MVS	No effect
MUS	No effect

(RN | mrf | mrb) = rnd (mrf | mrb) MOD3;

General Form

Compute (Compute) Opcode
RREG Register Class = rnd mrf MOD3
RREG Register Class = rnd mrb MOD3

mrf = rnd mrf MOD3
mrb = rnd mrb MOD3

Function

Rounds the specified MR value to nearest at bit 32 (the MR1-MR0 boundary). The result is placed either in the fixed-point field in register Rn or one of the MR accumulation registers, which must be the same MR register that provided the input. If Rn is specified, only the portion of the result that has the same format as the inputs is transferred (bits 31-0 for integers, bits 63-32 for fractional). The floating-point extension field in Rn is set to all 0s. If MRF or MRB is specified, the entire 80-bit result is placed in MRF or MRB.

ASTATx/y Flags

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
MN	Set if the result is negative, otherwise cleared
MI	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

STKYx/y Flags

MOS	Sticky indicator for MV bit set
MIS	No effect
MVS	No effect
MUS	No effect

(RN | mrf | mrb) = sat (mrf | mrb) MOD2;

General Form

Compute (Compute) Opcode
RREG Register Class = sat mrf MOD2
RREG Register Class = sat mrb MOD2
mrf = sat mrf MOD2
mrb = sat mrb MOD2

Function

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

ASTATx/y Flags

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
MN	Set if the result is negative, otherwise cleared
MI	Cleared
MV	Cleared

STKYx/y Flags

MOS	No effect
MIS	No effect
MVS	No effect
MUS	No effect

(mrf | mrb) = 0;

General Form

Compute (Compute) Opcode
mrf = 0
mrb = 0

Function

Sets the value of the specified MR register to zero. All 80 bits (MR2, MR1, MR0, MS2, MS1, MS0) are cleared.

NOTE: Only only MRF/MRB=0 instructions are valid. MSF/MSB=0 instruction does not exist.

ASTATx/y Flags

MU	Cleared

(mrf | mrb) = 0;

MN	Cleared
MI	Cleared
MV	Cleared

STKYx/y Flags

MOS	No effect
MIS	No effect
MVS	No effect
MUS	No effect

23 Multiplier Floating-Point Computations

32/40/64-bit Multiplier floating-point operations are described in this section.

For information on syntax and opcodes, see Compute (Compute) Opcode.

For information on arithmetic status, see REGF_ASTATX and REGF_ASTATY registers.

32-bit/40-bit Floating-Point Operations

The following sections provide descriptions for the Multiplier 32-bit and 40-bit operations.

FN = FX * FY;

General Form

Compute (Compute) Opcode
FREG Register Class = FREG Register Class * FREG Register Class

Function

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

ASTATx/y Flags

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

STKYx/y Flags

MOS	No effect
MIS	Sticky indicator for MI bit set
MVS	Sticky indicator for MV bit set

MUS	Sticky indicator for MU bit set

64-bit Floating-Point Operations

The following sections provide descriptions for the Multiplier 64-bit Floating-Point operations.

FM:N = FX:Y * FZ:W;

General Form

Compute (Compute) Opcode
DBLREG Register Type = DBLREG Register Type * DBLREG Register Type

Function

Multiplies the floating-point operands in register Fx:y and Fz:w and places the result in the register Fm:n.

This instruction uses MR register for intermediate computations. The MR register could be MRF or MRB depending on the `REGF_MODE1.SRCU` bit.

WARNING: The data in MR register at the end of execution of this instruction is not valid.

Hence, if the MR register contains a valid data which may be required later, the user must save the data in MR before executing this instruction.

This operation requires seven execution cycles. The lower half of the destination register Fm:n (for example Rn) gets updated at the end of 6th execution cycle. The upper half of the destination register Fm:n (for example Rm) and status registers (ASTATx/y or STKYx/y) get updated at the end of 7th execution cycle.

NOTE: For all 64-bit multiply operations, note the following:

- If only one of the operand of 64-bit Multiply operation is a NAN, the sign of the result will be the sign of the input operand, which is a NAN.
- If both the operands of a 64-bit Multiply operation are NANs, the sign of the result will be OR of the signs of the input operands.

ASTATx/y Flags

MI	Set if either input is a NAN or if the inputs are ±infinity and ±zero, otherwise cleared
MN	Set if the result is negative, otherwise cleared
MU	Set if the unbiased exponent of the result is less than –1022, otherwise cleared
MV	Set if the unbiased exponent of the result is greater than 1023, otherwise cleared

STKYx/y Flags

MUS	Sticky indicator for MU bit set
MOS	No effect
MIS	Sticky indicator for MI bit set
MVS	Sticky indicator for MV bit set

FM:N = FX:Y * FY;

General Form

Compute (Compute) Opcode
DBLREG Register Type = DBLREG Register Type * FREG Register Class

Function

Multiplies the Double Precision floating-point operands in register Fx:y with Single precision floating-point operand in register FY and places the double precision floating point result in the register Fm:n.

This instruction uses MR register for intermediate computations. The MR register could be MRF or MRB depending on the `REGF_MODE1.SRCU` bit.

WARNING: The data in MR register at the end of execution of this instruction is not valid.

Hence, if the MR register contains a valid data which may be required later, the user must save the data in MR before executing this instruction.

This operation requires seven execution cycles.

The lower half of the destination register Fm:n (i.e. Rn) gets updated at the end of 6th execution cycle. The upper half of the destination register Fm:n (i.e. Rm) and status registers (ASTATx/y or STKYx/y) get updated at the end of 7th execution cycle.

ASTATx/y Flags

MI	Set if either input is a NAN or if the inputs are ±infinity and ±zero, otherwise cleared
MN	Set if the result is negative, otherwise cleared
MU	Set if the unbiased exponent of the result is less than –1022, otherwise cleared
MV	Set if the unbiased exponent of the result is greater than 1023, otherwise cleared

STKYx/y Flags

MUS	Sticky indicator for MU bit set

MOS	No effect
MIS	Sticky indicator for MI bit set
MVS	Sticky indicator for MV bit set

FM:N = FX * FY;

General Form

Compute (Compute) Opcode
DBLREG Register Type = FREG Register Class * FREG Register Class

Function

Multiplies the Single Precision floating-point operands in register FX with Single precision floating-point operand in register FY and places the double precision floating-point result in the register Fm:n.

Caution

This instruction uses MR register for intermediate computations. The MR register could be MRF or MRB depending on the `REGF_MODE1.SRCU` bit.

WARNING: The data in MR register at the end of execution of this instruction is not valid.

Hence, if the MR register contains a valid data which may be required later, the user must save the data in MR before executing this instruction.

This operation requires seven execution cycles.

The lower half of the destination register Fm:n (i.e. Rn) gets updated at the end of 6th execution

cycle. The upper half of the destination register Fm:n (i.e. Rm) and status registers (ASTATx/y or STKYx/y) get updated at the end of 7th execution cycle.

ASTATx/y Flags

MI	Set if either input is a NAN or if the inputs are ±infinity and ±zero, otherwise cleared
MN	Set if the result is negative, otherwise cleared
MU	Set if the unbiased exponent of the result is less than –1022, otherwise cleared
MV	Set if the unbiased exponent of the result is greater than 1023, otherwise cleared

STKYx/y Flags

MUS	Sticky indicator for MU bit set
MOS	No effect

MIS	Sticky indicator for MI bit set
MVS	Sticky indicator for MV bit set

24 Shifter Immediate Computations

Shifter and shift immediate operations are described in this section. The succeeding pages provide detailed descriptions of each operation. Some of the instructions accept the following modifiers.

Some of the instructions in this group accept the following modifiers enclosed in parentheses.

- (SE) = Sign extension of deposited or extracted field
- (EX) = Extended exponent extract
- (NU) = No update (bit FIFO)

For information on syntax and opcodes, see Compute (Compute) Opcode.

For information on arithmetic status, see `REGF_ASTATX` and `REGF_ASTATY` registers.

RN = lshift RX by (RY | DATA8);

General Form

Compute (Compute) Opcode
RREG Register Class = lshift RREG Register Class by RREG Register Class
Shift Immediate (ShiftImm) Opcode (Type 6 Instruction only)
RFREG Register Class = lshift RFREG Register Class by imm8c12 Register Type

Function

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

ASTATx/y Flags

SS	Cleared

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

RN = RN or lshift RX by (RY | DATA8);

General Form

Compute (Compute) Opcode
RREG Register Class = RREG Register Class or lshift RREG Register Class by RREG Register Class
Shift Immediate (ShiftImm) Opcode (Type 6 Instruction only)
RFREG Register Class = RFREG Register Class or lshift RFREG Register Class by imm8c12 Register Type

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 two's-complement numbers. Positive values select a left shift, negative values select a right shift. The 8-bit immediate data can take values between -128 and 127 inclusive, allowing for a shift of a 32-bit field from off-scale right to off-scale left.

ASTATx/y Flags

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

RN = ashift RX by (RY | DATA8);

General Form

Compute (Compute) Opcode
RREG Register Class = ashift RREG Register Class by RREG Register Class
Shift Immediate (ShiftImm) Opcode (Type 6 Instruction only)
RFREG Register Class = ashift RFREG Register Class by imm8c12 Register Type

Function

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

ASTATx/y Flags

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

RN = RN or ashift RX by (RY | DATA8);

General Form

Compute (Compute) Opcode
RREG Register Class = RREG Register Class or ashift RREG Register Class by RREG Register Class
Shift Immediate (ShiftImm) Opcode (Type 6 Instruction only)
RFREG Register Class = RFREG Register Class or ashift RFREG Register Class by imm8c12 Register Type

Function

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

ASTATx/y 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

RN = rot RX by (RY | DATA);

General Form

Compute (Compute) Opcode
RREG Register Class = rot RREG Register Class by RREG Register Class
Shift Immediate (ShiftImm) Opcode (Type 6 Instruction only)
RFREG Register Class = rot RFREG Register Class by imm8c12 Register Type

Function

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

ASTATx/y Flags

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

RN = bclr RX by (RY | DATA8);

General Form

Compute (Compute) Opcode
RREG Register Class = bclr RREG Register Class by RREG Register Class
Shift Immediate (ShiftImm) Opcode (Type 6 Instruction only)
RFREG Register Class = bclr RFREG Register Class by uimm5c12 Register Type

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.

ASTATx/y Flags

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

There is also a bit manipulation instruction (type 18 a) that affects one or more bits in a system register. The BIT CLR `Sysreg` instruction should not be confused with the BCLR `Dreg` instruction. This shifter operation affects only one bit in a data register file location. For more information, see System Register Bit Manipulation.

RN = bset RX by (RY | DATA8);

General Form

Compute (Compute) Opcode
RREG Register Class = bset RREG Register Class by RREG Register Class
Shift Immediate (ShiftImm) Opcode (Type 6 Instruction only)
RFREG Register Class = bset RFREG Register Class by uimm5c12 Register Type

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.

ASTATx/y Flags

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

There is also a bit manipulation instruction (type 18 a) that affects one or more bits in a system register. The BIT SET `Sysreg` instruction should not be confused with the BSET `Dreg` instruction. This shifter operation affects only one bit in a data register file location. For more information, see System Register Bit Manipulation.

RN = btgl RX by (RY | DATA8);

General Form

Compute (Compute) Opcode
RREG Register Class = btgl RREG Register Class by RREG Register Class
Shift Immediate (ShiftImm) Opcode (Type 6 Instruction only)
RFREG Register Class = btgl RFREG Register Class by uimm5c12 Register Type

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.

ASTATx/y Flags

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

There is also a bit manipulation instruction (type 18 a) that affects one or more bits in a system register. The BIT TGL *Sysreg* instruction should not be confused with the BTGL *Dreg* instruction. This shifter operation affects only one bit in a data register file location. For more information, see System Register Bit Manipulation.

btst RX by (RY | DATA8);

General Form

Compute (Compute) Opcode
btst RREG Register Class by RREG Register Class
Shift Immediate (ShiftImm) Opcode (Type 6 Instruction only)
btst RFREG Register Class by uimm5c12 Register Type

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.

ASTATx/y Flags

SS	Cleared
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

There is also a bit manipulation instruction (type 18 a) that affects one or more bits in a system register. The BIT TST *Sysreg* instruction should not be confused with the BTST *Dreg* instruction. This shifter operation affects only one bit in a data register file location. For more information, see System Register Bit Manipulation.

RN = fdep RX by (RY | BIT6:LEN6);

General Form

Compute (Compute) Opcode

RREG Register Class = fdep RREG Register Class by RREG Register Class
Shift Immediate (ShiftImm) Opcode (Type 6 Instruction only)
RFREG Register Class = fdep RFREG Register Class by uimm6bit Register Type : uimm6len Register Type

Function

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

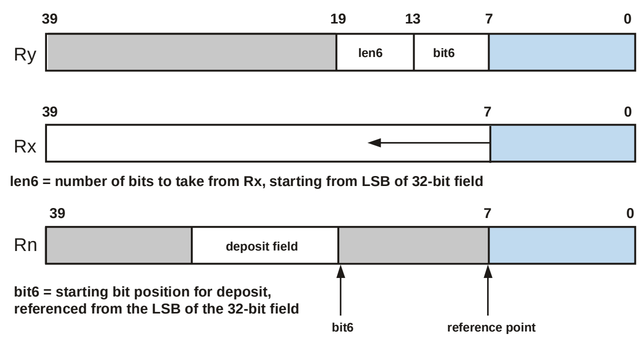

Figure 24-1: Field Alignment

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)
```

ASTATx/y Flags

SS	Cleared
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

RN = RN or fdep RX by (RY | BIT6:LEN6);

General Form

Compute (Compute) Opcode
RREG Register Class = RREG Register Class or fdep RREG Register Class by RREG Register Class
Shift Immediate (ShiftImm) Opcode (Type 6 Instruction only)
RFREG Register Class = RFREG Register Class or fdep RFREG Register Class by uimm6bit Register Type : uimm6len Register Type

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.

```
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-------/
```

ASTATx/y Flags

SS	Cleared
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

RN = fdep RX by (RY | BIT6:LEN6) (se);

General Form

Compute (Compute) Opcode
RREG Register Class = fdep RREG Register Class by RREG Register Class (se)
Shift Immediate (ShiftImm) Opcode (Type 6 Instruction only)
RFREG Register Class = fdep RFREG Register Class by uimm6bit Register Type : uimm6len Register Type (se)

Function

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

.

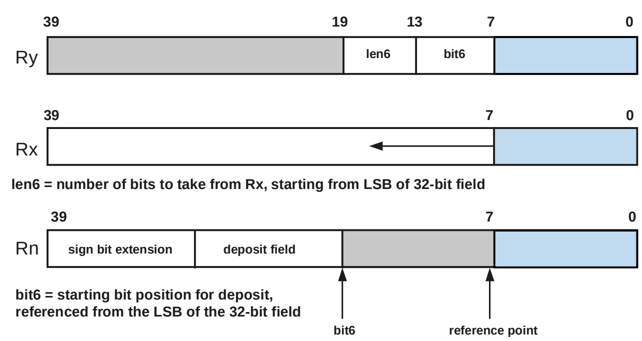

Figure 24-2: Field Alignment

39 19137 0 len6 bit6 Ry 39 70 Rx len6 = number of bits to take from Rx, starting from LSB of32-bitfield 39 70 bit6 = startingbit position for deposit, referenced from the LSB of the 32-bit field Rn signbit extension deposit field bit6 reference point

```
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)
```

ASTATx/y Flags

SS	Cleared
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

RN = RN or fdep RX by (RY | BIT6:LEN6) (se);

General Form

Compute (Compute) Opcode
RREG Register Class = RREG Register Class or fdep RREG Register Class by RREG Register Class (se)
Shift Immediate (ShiftImm) Opcode (Type 6 Instruction only)
RFREG Register Class = RFREG Register Class or fdep RFREG Register Class by uimm6bit Register Type : uimm6len Register Type (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.

```
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-----/
```

ASTATx/y Flags

SS	Cleared
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

RN = fext RX by (RY | BIT6:LEN6);

General Form

Compute (Compute) Opcode
RREG Register Class = fext RREG Register Class by RREG Register Class
Shift Immediate (ShiftImm) Opcode (Type 6 Instruction only)
RFREG Register Class = fext RFREG Register Class by uimm6bit Register Type : uimm6len Register Type

Function

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

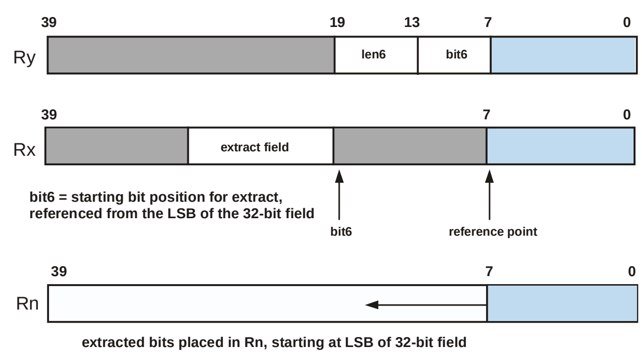

Figure 24-3: Field Alignment

```
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
```

ASTATx/y Flags

SS	Cleared
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

RN = fext RX by (RY | BIT6:LEN6) (se);

General Form

Compute (Compute) Opcode
RREG Register Class = fext RREG Register Class by RREG Register Class (se)
Shift Immediate (ShiftImm) Opcode (Type 6 Instruction only)
RFREG Register Class = fext RFREG Register Class by uimm6bit Register Type : uimm6len Register Type (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.

```
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
```

ASTATx/y Flags

SS	Cleared

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

RN = exp RX;

General Form

Compute (Compute) Opcode
RREG Register Class = exp RREG Register Class

Function

Extracts the exponent of the fixed-point operand in Rx. The exponent is placed in the shf8 field in register Rn. The exponent is calculated as the two's-complement of: # leading sign bits in Rx - 1

ASTATx/y Flags

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

RN = exp RX (ex);

General Form

Compute (Compute) Opcode
RREG Register Class = exp RREG Register Class (ex)

Function

Extracts the exponent of the fixed-point operand in Rx, assuming that the operand is the result of an ALU operation. The exponent is placed in the shf8 field in register Rn. If the AV status bit is set, a value of +1 is placed in the shf8 field to indicate an extra bit (the ALU overflow bit). If the AV status bit is not set, the exponent is calculated as the two's-complement of: # leading sign bits in Rx - 1

ASTATx/y Flags

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

RN = leftz RX;

General Form

Compute (Compute) Opcode
RREG Register Class = leftz RREG Register Class

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.

ASTATx/y Flags

SS	Cleared
SZ	Set if the MSB of Rx is 1, otherwise cleared
SV	Set if the result is 32, otherwise cleared

RN = lefto RX;

General Form

Compute (Compute) Opcode
RREG Register Class = lefto RREG Register Class

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.

ASTATx/y Flags

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

RN = fpack FX;

General Form

Compute (Compute) Opcode
RREG Register Class = fpack FREG Register Class

Function

Converts the IEEE 32-bit floating-point value in Fx to a 16-bit floating- 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 <= 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 <= 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 exp = source exponent sign bit remains the same in all cases

*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.

ASTATx/y Flags

SS	Cleared
SZ	Cleared

FN = funpack RX;

General Form

Compute (Compute) Opcode
FREG Register Class = funpack RREG Register Class

Function

Converts the 16-bit floating-point value in Rx to an IEEE 32-bit floating- point value stored in Fx. The result consists of:

0 < exp[*1] ≤ 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.

ASTATx/y Flags

SS	Cleared
SZ	Cleared
SV	Cleared

bitdep RX by (RY | BITLEN12);

General Form

Compute (Compute) Opcode
bitdep RREG Register Class by RREG Register Class
Shift Immediate (ShiftImm) Opcode (Type 6 Instruction only)
bitdep RFREG Register Class by uimm12 Register Type

Function

Deposits the bitlen number of bits (specified by Ry or bitlen) in the bit FIFO from Rx. The bits read from Rx are right justified. Write pointer incremented by the number of bit appended. To understand the BITDEP instruction, it is easiest to observe how the data register and bit FIFO behave during instruction execution. If the data register, Rx (40 Bits), contains:

```
                              39       32
                              |--------|
31       23       15          7        0
|--------|----abcd|efghijkl|--------|
               \-----------/
                bitlen bits
```

And, the bit FIFO (64 Bits), before instruction execution contains:

```
63       55       47       39       32
|qwertyui|opasdfgh|lmn-----|--------|
                       ^- BFFWRP - Write Pointer
```

```
31        23        15        7         0
|--------|--------|--------|--------|
```

Then, after instruction execution, the bit FIFO (64 Bits) contains:

```
63        55        47        39       32
|qwertyui|opasdfgh|lmnabcde|fghijkl-|
                                   ^- BFFWRP - Write Pointer
31        23        15        7         0
|--------|--------|--------|--------|
```

This operation on the bit FIFO is equivalent to:

1. BFF = BFF OR FDEP Rx BY <64-(BFFWRP+bitlen)> : <bitlen>
2. BFFWRP = BFFWRP + <bitlen>

Note: Do not use the pseudo code above as instruction syntax.

The first operation is similar to the FDEP instruction, but the right and left shifters are modified to be 64-bit shifters. The second operation provides write pointer update and flag update, which differs from the FDEP instruction.

SF is set or reset according to the value of write pointer. A data of more than 32 in the lower 6 bits of Ry or immediate field (bitlen12) is prohibited, and use of such data sets SV. Attempts to append more bits than the bit FIFO has room for results in an undefined bit FIFO and write pointer. SV is set in that case, otherwise SV is cleared. SZ and SS are cleared.

ASTATx/y Flags

SS	Cleared
SZ	Cleared
SF	Set if updated BFFWRP>= 32, otherwise cleared **NOTE:** SF has up-to one cycle of effect latency on conditional non-L1 accesses
SV	Set if any bits are deposited to the left of the 32-bit fixed-point output field (that is, if Ry or bitlen12 > 32), otherwise cleared

RN = bitext (RX | BITLEN12) (nu);

General Form

Compute (Compute) Opcode
RREG Register Class = bitext RREG Register Class
RREG Register Class = bitext RREG Register Class (nu)
Shift Immediate (ShiftImm) Opcode (Type 6 Instruction only)
RFREG Register Class = bitext uimm12 Register Type

RFREG Register Class = bitext uimm12 Register Type (nu)

Function

Extracts bitlen number of bits (specified by Rx or bitlen) from the bit FIFO and places the data in Rn. The bits in Rn are right justified. Decrements write pointer by same number as read bits. Remaining content of the bit FIFO is left-shifted so that it is MSB aligned. The optional modifier NU (no update) or query only, returns the requested number of bits as usual but does not modify the bit FIFO or Write pointer. To understand the BITEXT instruction, it is easiest to observe how the data register and bit FIFO behave during instruction execution. If the bit FIFO (64 bits) contains:

```
63       55       47       39      32
|abcdefgh|ijklmn--|--------|--------|
 \-----------/  ^ - BFFWRP Pointer
bitlen bits
31       23       15       7        0
|--------|--------|--------|--------|
```

After instruction execution, the Rn register (40 bits) contains:

```
                           39      32
                           |00000000|
31       23       15       7        0
|00000000|0000abcd|efghijkl|00000000|
```

And the bit FIFO (64 Bits) contains:

```
63       55       47       39      32
|mn------|--------|--------|--------|
  ^- BFFWRP Pointer
31       23       15       7        0
|--------|--------|--------|--------|
```

This operation on the Bit FIFO is equivalent to:

1. Rn = FEXT BFF[63:32] BY <(32-bitlen)>:<bitlen>
2. BFF = BFF << bitlen 3. BFFWRP = BFFWRP - bitlen

Note: Do not use the pseudo code above as instruction syntax.

The first operation is the same as an FEXT instruction operation.

The second operation (bit FIFO 64-bit register with a left shift) and third operation (write pointer update and flag update) are unique to the bit FIFO operation.

ASTATx/y Flags

A value of more than 32 in the lower 6 bits of Rx or the bitlen immediate field is prohibited and use of such a value sets SV. Attempts to get more bits than those in the bit FIFO results in undefined pointer and bit FIFO. SV is set in that case. SF is set if write pointer is greater than or equal to 32. SZ is set if output is zero, otherwise cleared. SS is cleared. Usage of the NU modifier affects SV, SZ, and SS as described above and the SF flag is not updated.

SS	Cleared
SZ	Set if output is zero, otherwise cleared
SF	Set if updated BFFWRP >= 32, otherwise cleared. If NU modifier is used SF reflects the un-updated Write pointer status **NOTE:** SF has up-to one cycle of effect latency on conditional non-L1 accesses
SV	Set if an attempt is made to extract more bits than those in bit FIFO, otherwise cleared

bffwrp = (RN | DATA7);

General Form

Compute (Compute) Opcode
bffwrp = RREG Register Class
Shift Immediate (ShiftImm) Opcode (Type 6 Instruction only)
bffwrp = uimm7c12 Register Type

Function

Updates write pointer from Rn or the immediate 7 bit data specified. Only 7 least significant bits of Rn are written. The maximum permissible data to be written into BFFWRP is 64. Examples For bit FIFO examples, see bitdep RX by (RY | BITLEN12);.

ASTATx/y Flags

SS	Cleared
SZ	Cleared
SF	Set if updated BFFWRP >= 32, otherwise cleared **NOTE:** SF has up-to one cycle of effect latency on conditional non-L1 accesses
SV	Set if written <data7> is > 64, otherwise cleared

RN = bffwrp;

General Form

Compute (Compute) Opcode
RREG Register Class = bffwrp

Function

Transfers write pointer value to Rn. Examples For bit FIFO examples, see the BITDEP instruction bitdep RX by (RY | BITLEN12);.

ASTATx/y Flags

SS	Cleared
SZ	Cleared
SV	Cleared
SF	Not affected

25 Multi-Function Instruction Computations

Multifunction instructions are parallelized single ALU and Multiplier instructions. For functional description and status flags and for parallel Multiplier and ALU instructions input operand constraints see the ALU Fixed-Point Computations section and the Multiplier Fixed-Point Computations section. This section lists all possible instruction syntax options.

Note that the MRB register is not supported in multifunction instructions.

32-Bit, 40-Bit Instructions

Fixed-Point ALU (dual Add and Subtract)

Ra = Rx + Ry , Rs = Rx – Ry

Floating-Point ALU (dual Add and Subtract)

Fa = Fx + Fy , Fs = Fx – Fy

Fixed-Point Multiplier and ALU

Rm = R3-0 * R7-4 (SSFR), Ra = R11-8 + R15-12

Rm = R3-0 * R7-4 (SSFR), Ra = R11-8 – R15-12

Rm = R3-0 * R7-4 (SSFR), Ra = (R11-8 + R15-12)/2

MRF = MRF + R3-0 * R7-4 (SSF), Ra = R11-8 + R15-12

MRF = MRF + R3-0 * R7-4 (SSF), Ra = R11-8 – R15-12

MRF = MRF + R3-0 * R7-4 (SSF), Ra = (R11-8 + R15-12)/2

Rm = MRF + R3-0 * R7-4 (SSFR), Ra = R11-8 + R15-12

Rm = MRF + R3-0 * R7-4 (SSFR), Ra = R11-8 – R15-12

Rm = MRF + R3-0 * R7-4 (SSFR), Ra =(R11-8 + R15-12)/2

MRF = MRF – R3-0 * R7-4 (SSF), Ra = R11-8 + R15-12

MRF = MRF – R3-0 * R7-4 (SSF), Ra = R11-8 – R15-12

MRF = MRF – R3-0 * R7-4 (SSF), Ra = (R11-8 + R15-12)/2

Rm = MRF – R3-0 * R7-4 (SSFR), Ra = R11-8 + R15-12

Rm = MRF – R3-0 * R7-4 (SSFR), Ra = R11-8 – R15-12

Rm = MRF – R3-0 * R7-4 (SSFR), Ra =(R11-8 + R15-12)/2

Floating-Point Multiplier and ALU

Fm = F3-0 * F7-4, Fa = F11-8 + F15-12

Fm = F3-0 * F7-4, Fa = F11-8 – F15-12

Fm = F3-0 * F7-4, Fa = FLOAT R11-8 by R15-12

Fm = F3-0 * F7-4, Ra = FIX F11-8 by R15-12

Fm = F3-0 * F7-4, Fa = (F11-8 + F15-12)/2

Fm = F3-0 * F7-4, Fa = ABS F11-8

Fm = F3-0 * F7-4, Fa = MAX (F11-8, F15-12)

Fm = F3-0 * F7-4, Fa = MIN (F11-8, F15-12)

Fixed-Point Multiplier and ALU (dual Add and Subtract)

Rm=R3-0 * R7-4 (SSFR), Ra=R11-8 + R15-12, Rs=R11-8 – R15-12

Floating-Point Multiplier and ALU (dual Add and Subtract)

Fm=F3-0 * F7-4, Fa=F11-8 + F15-12, Fs=F11-8 – F15-12

Note that both instructions above are typically used for fixed- or floating- point FFT butterfly calculations.

64-Bit Instructions

All 64-bit float instruction require a valid register pairing for source or destination register (for example F15:14, F1:0)

Floating-Point Multiplier and ALU/add

Fm:n = F3/1:2/0 * F7/5:6/4, Fa:b = F11/9:10/8 + F15/13:14/12;

Fm:n = any valid register pair

Fa:b = any valid register pair

Floating-Point Multiplier and ALU/subtract

Fm:n = F3/1:2/0 * F7/5:6/4, Fa:b = F11/9:10/8 - F15/13:14/12;

Fm:n = any valid register pair

Fa:b = any valid register pair

26 Immediate (imm) and Constant (const) Opcodes

This section provides opcodes for the immediate data and constant types.

imm16visa Register Type

imm16visa Attributes

range	allow_label
-0x8000:0x7fff	true

imm23pc Register Type

imm23pc Attributes

range	allow_label
-0x400000:0x3fffff	true

imm24 Register Type

imm24 Attributes

range	allow_label
-0x800000:0x7fffff	true

imm24pc Register Type

imm24pc Attributes

range	allow_label
-0x800000:0x7fffff	true

imm32 Register Type

imm32 Attributes

range	allow_label
-0x80000000:0x7fffffff	true

imm32c Register Type

imm32c Attributes

range	allow_label
-0x80000000:0x7fffffff	false

imm32f Register Type

imm32f Attributes

range	allow_label
-0x80000000:0x7fffffff	true

imm6 Register Type

imm6 Attributes

range	allow_label
-0x20:0x1f	true

imm6pc Register Type

imm6pc Attributes

range	allow_label
-0x20:0x1f	true

imm6visa Register Type

imm6visa Attributes

range	allow_label
-0x20:0x1f	true

imm6visapc Register Type

imm6visapc Attributes

range	allow_label
-0x20:0x1f	true

imm7visa Register Type

imm7visa Attributes

range	allow_label
-0x40:0x3f	true

imm8c12 Register Type

imm8c12 Attributes

range
-0x80:0x7f

uimm12 Register Type

uimm12 Attributes

range
0x0:0xfff

uimm16 Register Type

uimm16 Attributes

range	allow_label
0x0:0xffff	true

uimm5c12 Register Type

uimm5c12 Attributes

range
0x0:0x1f

uimm6bit Register Type

uimm6bit Attributes

range	allow_label
0x0:0x3f	true

uimm6len Register Type

uimm6len Attributes

range	allow_label
0x0:0x3f	true

uimm7c12 Register Type

uimm7c12 Attributes

range
0x0:0x7f

27 Register (reg) Opcodes

This section provides opcodes for the register types. These instructions are multi-issuable with compute.

B1REG Register Class

The B1REG (base registers, DAG1) class includes the base address registers from data address generator 1.

B1REG Syntax

Code	Syntax
000	b0
001	b1
010	b2
011	b3
100	b4
101	b5
110	b6
111	b7

B2REG Register Class

The B2REG (base registers, DAG2) class includes the base address registers from data address generator 2.

B2REG Syntax

Code	Syntax
000	b8
001	b9
010	b10
011	b11

Code	Syntax
100	b12
101	b13
110	b14
111	b15

DBLREG Register Type

The DBLREG (64-bit floating-point data registers, PEx) class includes the 64-bit floating-point data registers from processing element x (PEx).

DBLREG Syntax

Code	Syntax
0000	f1:0
0010	f3:2
0100	f5:4
0110	f7:6
1000	f9:8
1010	f11:10
1100	f13:12
1110	f15:14

DBLREG3 Register Class

The DBLREG3 (64-bit floating-point data registers, PEx) class includes the 64-bit floating-point data registers from processing element x (PEx).

DBLREG3 Syntax

Code	Syntax
000	f1:0
001	f3:2
010	f5:4
011	f7:6
100	f9:8
101	f11:10

Code	Syntax
110	f13:12
111	f15:14

DBLXAREG Register Class

The DBLXAREG (register file data register, 64-bit floating-point, input x, range "A") class includes register file locations, 64-bit floating-point, input x, range "A".

DBLXAREG Syntax

Code	Syntax
0	f9:8
1	f11:10

DBLXMREG Register Class

The DBLXMREG (register file data register, 64-bit floating-point, input x, range "M") class includes register file locations, 64-bit floating-point, input x, range "M".

DBLXMREG Syntax

Code	Syntax
0	f1:0
1	f3:2

DBLYAREG Register Class

The DBLYAREG (register file data register, 64-bit floating-point, input y, range "A") class includes register file locations, 64-bit floating-point, input Y, range "A".

DBLYAREG Syntax

Code	Syntax
0	f13:12
1	f15:14

DBLYMREG Register Class

The DBLYMREG (register file data register, 64-bit floating-point, input y, range "M") class includes register file locations, 64-bit floating-point, input y, range "M".

DBLYMREG Syntax

Code	Syntax
0	f5:4
1	f7:6

FREG Register Class

The FREG (floating-point data registers, PEx) class includes the floating-point data registers from processing element x (PEx).

FREG Syntax

Code	Syntax
0000	f0
0001	f1
0010	f2
0011	f3
0100	f4
0101	f5
0110	f6
0111	f7
1000	f8
1001	f9
1010	f10
1011	f11
1100	f12
1101	f13
1110	f14
1111	f15

FXAREG Register Class

The FXAREG (floating-point data registers, input x, range "A") class includes data register file locations, floating-point, input x, range "A".

FXAREG Syntax

Code	Syntax
00	f8
01	f9
10	f10
11	f11

FXMREG Register Class

The FXMREG (floating-point data registers, input x, range "M") class includes data register file locations, floating-point, input x, range "M".

FXMREG Syntax

Code	Syntax
00	f0
01	f1
10	f2
11	f3

FYAREG Register Class

The FYMREG (floating-point data registers, input y, range "A") class includes data register file locations, floating-point, input y, range "A".

FYAREG Syntax

Code	Syntax
00	f12
01	f13
10	f14
11	f15

FYMREG Register Class

The FYMREG (floating-point data registers, input y, range "M") class includes data register file locations, floating-point, input y, range "M".

FYMREG Syntax

Code	Syntax
00	f4
01	f5
10	f6
11	f7

I1REG Register Class

The I1REG (index registers, DAG1) class includes the index address registers from data address generator 1.

I1REG Syntax

Code	Syntax
000	i0
001	i1
010	i2
011	i3
100	i4
101	i5
110	i6
111	i7

I2REG Register Class

The I2REG (index registers, DAG2) class includes the index address registers from data address generator 2.

I2REG Syntax

Code	Syntax
000	i8
001	i9
010	i10

Code	Syntax
011	i11
100	i12
101	i13
110	i14
111	i15

M1REG Register Class

The M1REG (modifier registers, DAG1) class includes the address modifier registers from data address generator 1.

M1REG Syntax

Code	Syntax
000	m0
001	m1
010	m2
011	m3
100	m4
101	m5
110	m6
111	m7

M2REG Register Class

The M2REG (modifier registers, DAG2) class includes the address modifier registers from data address generator 2.

M2REG Syntax

Code	Syntax
000	m8
001	m9
010	m10
011	m11
100	m12
101	m13

Code	Syntax
110	m14
111	m15

MRXFBREG Register Class

The MRXFBREG (multipler results registers) class includes the foreground and background multiplier results registers. The register syntax provides access to each portion of the result.

MRXFBREG Syntax

Code	Syntax
0000	mr0f
0001	mr1f
0010	mr2f
0100	mr0b
0101	mr1b
0110	mr2b

RFREG Register Class

The RFREG (register file data register) class includes:

- The `r0` through `r15` tokens indicate processing element X register file locations, fixed-point.
- The `f0` through `f15` tokens indicate processing element X register file locations, floating-point.

RFREG Syntax

Code	Syntax	Syntax Alias
0000	r0	f0
0001	r1	f1
0010	r2	f2
0011	r3	f3
0100	r4	f4
0101	r5	f5
0110	r6	f6
0111	r7	f7

Code	Syntax	Syntax Alias
1000	r8	f8
1001	r9	f9
1010	r10	f10
1011	r11	f11
1100	r12	f12
1101	r13	f13
1110	r14	14
1111	r15	f15

RREG Register Class

The RREG (register file data register, PEx, fixed-point) class includes the processing element x register file locations, fixed-point.

RREG Syntax

Code	Syntax
0000	r0
0001	r1
0010	r2
0011	r3
0100	r4
0101	r5
0110	r6
0111	r7
1000	r8
1001	r9
1010	r10
1011	r11
1100	r12
1101	r13
1110	r14
1111	r15

RXAREG Register Class

The RXAREG (register file data register, fixed-point, input x, range "A") class includes register file locations, fixed-point, input x, range "A".

RXAREG Syntax

Code	Syntax
00	r8
01	r9
10	r10
11	r11

RXMREG Register Class

The RXMREG (register file data register, fixed-point, input x, range "B") class includes register file locations, fixed-point, input x, range "B".

RXMREG Syntax

Code	Syntax
00	r0
01	r1
10	r2
11	r3

RYAREG Register Class

The RYAREG (register file data register, fixed-point, input y, range "A") class includes register file locations, fixed-point, input y, range "A".

RYAREG Syntax

Code	Syntax
00	r12
01	r13
10	r14
11	r15

RYMREG Register Class

The RYMREG (register file data register, fixed-point, input y, range "M") class includes register file locations, fixed-point, input y, range "M".

RYMREG Syntax

Code	Syntax
00	r4
01	r5
10	r6
11	r7

SREG Register Class

The SREG (register file data register, PEy) class includes:

- The S0 through S15 tokens indicate processing element y register file locations, fixed-point.
- The SF0 through SF15 tokens indicate processing element y register file locations, floating-point.

When used in complementary data register operations, the SREG class registers are used as CDREG class registers.

SREG Syntax

Code	Syntax	Syntax Alias
0000	s0	sf0
0001	s1	sf1
0010	s2	sf2
0011	s3	sf3
0100	s4	sf4
0101	s5	sf5
0110	s6	sf6
0111	s7	sf7
1000	s8	sf8
1001	s9	sf9
1010	s10	sf10
1011	s11	sf11
1100	s12	sf12
1101	s13	sf13

Code	Syntax	Syntax Alias
1110	s14	sf14
1111	s15	sf15

SYSREG Register Class

The SYSREG (system registers) class includes mode control registers, status registers, status stack register, flag register, and interrupt control registers.

SYSREG Syntax

Code	Syntax
0000	ustat1
0001	ustat2
0010	mode1
0011	mmask
0100	mode2
0101	flags
0110	astatx
0111	astaty
1000	stkyx
1001	stkyy
1010	irptl
1011	imask
1100	imaskp
1101	mode1stk
1110	ustat3
1111	ustat4
1100	astat
1110	stky

UREG Registers Class

The UREG (universal registers) class includes the registers from register classes: RFEG, SREG, I1REG, I2REG, M1REG, M2REG, B1REG, B2REG, and SYSREG.

UREG Syntax

Code	Syntax	Syntax Alias
0000000	r0	f0
0000001	r1	f1
0000010	r2	f2
0000011	r3	f3
0000100	r4	f4
0000101	r5	f5
0000110	r6	f6
0000111	r7	f7
0001000	r8	f8
0001001	r9	f9
0001010	r10	f10
0001011	r11	f11
0001100	r12	f12
0001101	r13	f13
0001110	r14	f14
0001111	r15	f15
0010000	i0	-
0010001	i1	-
0010010	i2	-
0010011	i3	-
0010100	i4	-
0010101	i5	-
0010110	i6	-
0010111	i7	-
0011000	i8	-
0011001	i9	-
0011010	i10	-
0011011	i11	-
0011100	i12	-
0011101	i13	-
0011110	i14	-

Code	Syntax	Syntax Alias
0011111	i15	-
0100000	m0	-
0100001	m1	-
0100010	m2	-
0100011	m3	-
0100100	m4	-
0100101	m5	-
0100110	m6	-
0100111	m7	-
0101000	m8	-
0101001	m9	-
0101010	m10	-
0101011	m11	-
0101100	m12	-
0101101	m13	-
0101110	m14	-
0101111	m15	-
0110000	l0	-
0110001	l1	-
0110010	l2	-
0110011	l3	-
0110100	l4	-
0110101	l5	-
0110110	l6	-
0110111	l7	-
0111000	l8	-
0111001	l9	-
0111010	l10	-
0111011	l11	-
0111100	l12	-
0111101	l13	-
0111110	l14	-

Code	Syntax	Syntax Alias
0111111	l15	-
1000000	b0	-
1000001	b1	-
1000010	b2	-
1000011	b3	-
1000100	b4	-
1000101	b5	-
1000110	b6	-
1000111	b7	-
1001000	b8	-
1001001	b9	-
1001010	b10	-
1001011	b11	-
1001100	b12	-
1001101	b13	-
1001110	b14	-
1001111	b15	-
1010000	s0	sf0
1010001	s1	sf1
1010010	s2	sf2
1010011	s3	sf3
1010100	s4	sf4
1010101	s5	sf5
1010110	s6	sf6
1010111	s7	sf7
1011000	s8	sf8
1011001	s9	sf9
1011010	s10	sf10
1011011	s11	sf11
1011100	s12	sf12
1011101	s13	sf13
1011110	s14	sf14

Code	Syntax	Syntax Alias
1011111	s15	sf15
1100000	faddr	-
1100001	daddr	-
1100011	pc	-
1100100	pcstk	-
1100101	pcstkp	-
1100110	laddr	-
1100111	curlcntr	-
1101000	lcntr	-
1101001	emuclk	-
1101010	emuclk2	-
1101011	px	-
1101100	px1	-
1101101	px2	-
1101110	tperiod	-
1101111	tcount	-
1110000	ustat1	-
1110010	mode1	-
1110011	mmask	-
1110100	mode2	-
1110101	flags	-
1110110	astatx	-
1110111	astaty	-
1111000	stkyx	-
1111001	stkyy	-
1111010	irptl	-
1111011	imask	-
1111100	imaskp	-
1111101	lirptl	-
1111101	mode1stk	-
1111110	ustat3	-
1111111	ustat4	-

Code	Syntax	Syntax Alias
1111100	astat	-
1111110	stky	-

UREGDBL Register Class

The UREGDBL (universal registers,floating-point data registers, PEx and PEy) class includes 64-bit fixed- and floating-point data register file locations for both processing elements.

UREGDBL Syntax

Code	Syntax	Syntax Alias
0000000	r1:0	f1:0
0000001	r0:1	f0:1
0000010	r3:2	f3:2
0000011	r2:3	f2:3
0000100	r5:4	f5:4
0000101	r4:5	f4:5
0000110	r7:6	f7:6
0000111	r6:7	f6:7
0001000	r9:8	f9:8
0001001	r8:9	f8:9
0001010	r11:10	f11:10
0001011	r10:11	f10:11
0001100	r13:12	F13:12
0001101	r12:13	f12:13
0001110	r15:14	f15:14
0001111	r14:15	f14:15
1010000	s1:0	sf1:0
1010001	s0:1	sf0:1
1010010	s3:2	sf3:2
1010011	s2:3	sf2:3
1010100	s5:4	sf5:4
1010101	s4:5	sf4:5
1010110	s7:6	sf7:6
1010111	s6:7	sf6:7

Code	Syntax	Syntax Alias
1011000	s9:8	sf9:8
1011001	s8:9	sf8:9
1011010	s11:10	sf11:10
1011011	s10:11	sf10:11
1011100	s13:12	sf13:12
1011101	s12:13	sf12:13
1011110	s15:14	sf15:14
1011111	s14:15	sf14:15

UREGXDAG1 Register Class

The UREGXDAG1 (universal registers, excluding DAG1) class includes the same registers as the UREG class, but omits DAG1 specific index, modify, base, and length registers.

UREGXDAG1 Syntax

Code	Syntax	Syntax Alias
0000000	r0	f0
0000001	r1	f1
0000010	r2	f2
0000011	r3	f3
0000100	r4	f4
0000101	r5	f5
0000110	r6	f6
0000111	r7	f7
0001000	r8	f8
0001001	r9	f9
0001010	r10	f10
0001011	r11	f11
0001100	r12	f12
0001101	r13	f13
0001110	r14	f14
0001111	r15	f15
0011000	i8	-

Code	Syntax	Syntax Alias
0011001	i9	-
0011010	i10	-
0011011	i11	-
0011100	i12	-
0011101	i13	-
0011110	i14	-
0011111	i15	-
0101000	m8	-
0101001	m9	-
0101010	m10	-
0101011	m11	-
0101100	m12	-
0101101	m13	-
0101110	m14	-
0101111	m15	-
0111000	l8	-
0111001	l9	-
0111010	l10	-
0111011	l11	-
0111100	l12	-
0111101	l13	-
0111110	l14	-
0111111	l15	-
1001000	b8	-
1001001	b9	-
1001010	b10	-
1001011	b11	-
1001100	b12	-
1001101	b13	-
1001110	b14	-
1001111	b15	-
1010000	s0	sf0

Code	Syntax	Syntax Alias
1010001	s1	sf1
1010010	s2	sf2
1010011	s3	sf3
1010100	s4	sf4
1010101	s5	sf5
1010110	s6	sf6
1010111	s7	sf7
1011000	s8	sf8
1011001	s9	sf9
1011010	s10	sf10
1011011	s11	sf11
1011100	s12	sf12
1011101	s13	sf13
1011110	s14	sf14
1011111	s15	sf15
1100000	faddr	-
1100001	daddr	-
1100011	pc	-
1100100	pcstk	-
1100101	pcstkp	-
1100110	laddr	-
1100111	curlcntr	-
1101000	lcntr	-
1101001	emuclk	-
1011001	emuclk	-
1101010	emuclk2	-
1011000	emuclk2	-
1101011	px	-
1011011	px	-
1101100	px1	-
1011100	px1	-
1101101	px2	-

Code	Syntax	Syntax Alias
1011101	px2	-
1101110	tperiod	-
1011110	tperiod	-
1101111	tcount	-
1011111	tcount	-
1110000	ustat1	-
1110001	ustat2	-
1110010	mode1	-
1111011	mode1	-
1110011	mmask	-
1110100	mode2	-
1111010	mode2	-
1110101	flags	-
1110110	astatx	-
1110111	astaty	-
1111000	stkyx	-
1111001	stkyy	-
1111010	irptl	-
1111001	irptl	-
1111011	imask	-
1111101	imask	-
1111100	imaskp	-
1111111	imaskp	-
1111101	lirptl	-
1111101	mode1stk	-
1111110	ustat3	-
1111111	ustat4	-
1111100	astat	-
1111110	stky	-

UREGXDAG1DBL Register Class

The UREGXDAG1DBL (universal registers,floating-point data registers, PEx and PEy) class includes 64-bit fixed- and floating-point data register file locations for both processing elements.

UREGXDAG1DBL Syntax

Code	Syntax	Syntax Alias
0000000	r1:0	f1:0
0000001	r0:1	f0:1
0000010	r3:2	f3:2
0000011	r2:3	f2:3
0000100	r5:4	f5:4
0000101	r4:5	f4:5
0000110	r7:6	f7:6
0000111	r6:7	f6:7
0001000	r9:8	f9:8
0001001	r8:9	f8:9
0001010	r11:10	f11:10
0001011	r10:11	f10:11
0001100	r13:12	f13:12
0001101	r12:13	f12:13
0001110	r15:14	f15:14
0001111	r14:15	f14:15
1010000	s1:0	sf1:0
1010001	s0:1	sf0:1
1010010	s3:2	sf3:2
1010011	s2:3	sf2:3
1010100	s5:4	sf5:4
1010101	s4:5	sf4:5
1010110	s7:6	sf7:6
1010111	s6:7	sf6:7
1011000	s9:8	sf9:8
1011001	s8:9	sf8:9
1011010	s11:10	sf11:10
1011011	s10:11	sf10:11

Code	Syntax	Syntax Alias
1011100	s13:12	sf13:12
1011101	s12:13	sf12:13
1011110	s15:14	sf15:14
1011111	s14:15	sf14:15

UREGXDAG2 Register Class

The UREGXDAG2 (universal registers, excluding DAG2) class includes the same registers as the UREG class, but omits DAG2 specific index, modify, base, and length registers.

UREGXDAG2 Syntax

Code	Syntax	Syntax Alias
0000000	r0	f0
0000001	r1	f1
0000010	r2	f2
0000011	r3	f3
0000100	r4	f4
0000101	r5	f5
0000110	r6	f6
0000111	r7	f7
0001000	r8	f8
0001001	r9	f9
0001010	r10	f10
0001011	r11	f11
0001100	r12	f12
0001101	r13	f13
0001110	r14	f14
0001111	r15	f15
0010000	i0	-
0010001	i1	-
0010010	i2	-
0010011	i3	-
0010100	i4	-

Code	Syntax	Syntax Alias
0010101	i5	-
0010110	i6	-
0010111	i7	-
0100000	m0	-
0100001	m1	-
0100010	m2	-
0100011	m3	-
0100100	m4	-
0100101	m5	-
0100110	m6	-
0100111	m7	-
0110000	l0	-
0110001	l1	-
0110010	l2	-
0110011	l3	-
0110100	l4	-
0110101	l5	-
0110110	l6	-
0110111	l7	-
1000000	b0	-
1000001	b1	-
1000010	b2	-
1000011	b3	-
1000100	b4	-
1000101	b5	-
1000110	b6	-
1000111	b7	-
1010000	s0	sf0
1010001	s1	sf1
1010010	s2	sf2
1010011	s3	sf3
1010100	s4	sf4

Code	Syntax	Syntax Alias
1010101	s5	sf5
1010110	s6	sf6
1010111	s7	sf7
1011000	s8	sf8
1011001	s9	sf9
1011010	s10	sf10
1011011	s11	sf11
1011100	s12	sf12
1011101	s13	sf13
1011110	s14	sf14
1011111	s15	sf15
1100000	faddr	-
1100001	daddr	-
1100011	pc	-
1100100	pcstk	-
1100101	pcstkp	-
1100110	laddr	-
1100111	curlcntr	-
1101000	lcntr	-
1101001	emuclk	-
1011001	emuclk	-
1101010	emuclk2	-
1011000	emuclk2	-
1101011	px	-
1011011	px	-
1101100	px1	-
1011100	px1	-
1101101	px2	-
1011101	px2	-
1101110	tperiod	-
1011110	tperiod	-
1101111	tcount	-

Code	Syntax	Syntax Alias
1011111	tcount	-
1110000	ustat1	-
1110001	ustat2	-
1110010	mode1	-
1111011	mode1	-
1110011	mmask	-
1110100	mode2	-
1111010	mode2	-
1110101	flags	-
1110110	astatx	-
1110111	astaty	-
1111000	stkyx	-
1111001	stkyy	-
1111010	irptl	-
1111001	irptl	-
1111011	imask	-
1111101	imask	-
1111100	imaskp	-
1111111	imaskp	-
1111101	lirptl	-
1111101	mode1stk	-
1111110	ustat3	-
1111111	ustat4	-
1111100	astat	-
1111110	stky	-

UREGXDAG2DBL Register Class

The UREGXDAG2DBL (universal registers,floating-point data registers, PEx and PEy) class includes 64-bit fixed- and floating-point data register file locations for both processing elements.

UREGXDAG2DBL Syntax

Code	Syntax	Syntax Alias
0000000	r1:0	f1:0
0000001	r0:1	f0:1
0000010	r3:2	f3:2
0000011	r2:3	f2:3
0000100	r5:4	f5:4
0000101	r4:5	f4:5
0000110	r7:6	f7:6
0000111	r6:7	f6:7
0001000	r9:8	f9:8
0001001	r8:9	f8:9
0001010	r11:10	f11:10
0001011	r10:11	f10:11
0001100	r13:12	f13:12
0001101	r12:13	f12:13
0001110	r15:14	f15:14
0001111	r14:15	f14:15
1010000	s1:0	sf1:0
1010001	s0:1	sf0:1
1010010	s3:2	sf3:2
1010011	s2:3	sf2:3
1010100	s5:4	sf5:4
1010101	s4:5	sf4:5
1010110	s7:6	sf7:6
1010111	s6:7	sf6:7
1011000	s9:8	sf9:8
1011001	s8:9	sf8:9
1011010	s11:10	sf11:10
1011011	s10:11	sf10:11
1011100	s13:12	sf13:12
1011101	s12:13	sf12:13
1011110	s15:14	sf15:14

Code	Syntax	Syntax Alias
1011111	s14:15	sf14:15

28 Numeric Formats

The processor supports the 32-bit single-precision floating-point and 64-bit double-precision floating-point data format defined in the IEEE Standard 754/854. In addition, the processor supports an extended-precision version of the same format with eight additional bits in the mantissa (40 bits total). The processor also supports 32-bit fixed-point formats-fractional and integer-which can be signed (two's-complement) or unsigned.

IEEE Single-Precision Floating-Point Data Format

The IEEE Standard 754/854 specifies a 32-bit single-precision floating-point format, shown in the ***IEEE 32-Bit Single-Precision Floating-Point Format*** figure. A number in this format consists of a sign bit(s), a 24-bit significand, and an 8-bit unsigned-magnitude exponent (e).

NOTE: In this manual the Single-Precision Floating-Point standard is referred to as 32-bit or 32-bit floating-point.

For normalized numbers, the significand consists of a 23-bit fraction, *f* and a "hidden" bit of 1 that is implicitly presumed to precede f_{22} in the significand. The binary point is presumed to lie between this hidden bit and f_{22}. The least significant bit (LSB) of the fraction is f_0; the LSB of the exponent is e_0.

The hidden bit effectively increases the precision of the floating-point significand to 24 bits from the 23 bits actually stored in the data format. It also ensures that the significand of any number in the IEEE normalized number format is always greater than or equal to one and less than two.

The unsigned exponent, e, can range between $1 \leq e \leq 254$ for normal numbers in single-precision format. This exponent is biased by +127. To calculate the true unbiased exponent, subtract 127 from e.

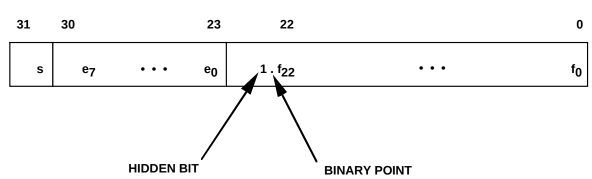

Figure 28-1: IEEE 32-Bit Single-Precision Floating-Point Format

The IEEE Standard also provides several special data types in the single-precision floating-point format:

- An exponent value of 255 (all ones) with a non-zero fraction is a not-a-number (NAN). NANs are usually used as flags for data flow control, for the values of uninitialized variables, and for the results of invalid operations such as $0 * \infty$.

- Infinity is represented as an exponent of 255 and a zero fraction. Note that because the fraction is signed, both positive and negative infinity can be represented.
- Zero is represented by a zero exponent and a zero fraction. As with infinity, both positive zero and negative zero can be represented.

The IEEE single-precision floating-point data types supported by the processor and their interpretations are summarized in the ***IEEE Single-Precision Floating-Point Data Types*** table.

Table 28-1: IEEE Single-Precision Floating-Point Data Types

Type	Exponent	Fraction	Value
NAN	255	Non-zero	Undefined
Infinity	255	0	$(-1)^s$ Infinity
Normal	1 e 254	Any	$(-1)^s (1.f_{22-0}) 2^{e-127}$
Zero	0	0	0 $(-1)^s$ Zero

IEEE Double-Precision Floating-Point (64-bit) Support

This section describes the Double-Precision Floating-Point instructions supported in SHARC+ core, their assembly language syntax, encoding of instructions and usage details.

NOTE: In this manual the Double-Precision Floating-Point standard is referred to as 64-bit or 64-bit floating-point.

IEEE Standard 754-2008 specifies a binary64 floating-point (Also known as double-precision floating-point in IEEE Standard 754-1985) format as shown in the following figure. A number represented in this format consists of a sign bit s, an 11-bit Exponent e and a 53-bit mantissa. For normalized numbers, the mantissa consists of a 52-bit fraction f and a “hidden” bit 1 that is implicitly presumed to precede bit-51. The binary point is presumed to reside between this hidden bit and bit-51.

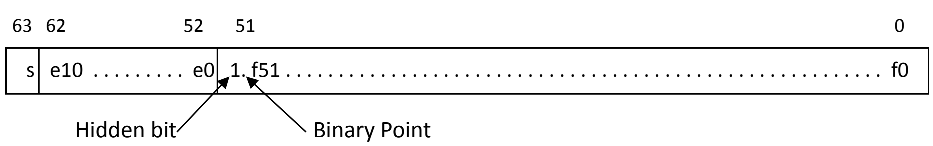

Figure 28-2: IEEE 64-bit Double Precision floating point format

The unsigned exponent e is within the set [1, 2046] for normal numbers. The true exponent value is biased by +1023 to produce e. To calculate the true unbiased exponent, 1023 must be subtracted from e. The range of numbers representable by double-precision format is listed in the table below:

Table 28-2: 64-bit Floating-Point Numbers

Sign	Biased Exponent	Mantissa	Number
x	0	0	±Zero

Table 28-2: 64-bit Floating-Point Numbers (Continued)

Sign	Biased Exponent	Mantissa	Number
x	0	≠0	Denormal
x	2047	0	±Infinity
x	2047	≠0	±NAN
x	0 < biased exp < 2047	X	Normal

Extended-Precision Floating-Point Format

The extended-precision floating-point format is 40 bits wide, with the same 8-bit exponent as in the IEEE standard format but with a 32-bit significand. This format is shown in the ***40-Bit Extended-Precision Floating-Point Format*** figure. In all other respects, the extended-precision floating-point format is the same as the IEEE standard format.

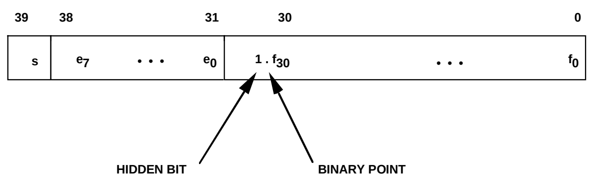

Figure 28-3: 40-Bit Extended-Precision Floating-Point Format

Short Word Floating-Point Format

The processor supports a 16-bit floating-point data type and provides conversion instructions for it. The short float data format has an 11-bit mantissa with a 4-bit exponent plus sign bit, as shown in the ***16-Bit Floating-Point Format*** figure. The 16-bit floating-point numbers reside in the lower 16 bits of the 32-bit floating-point field.

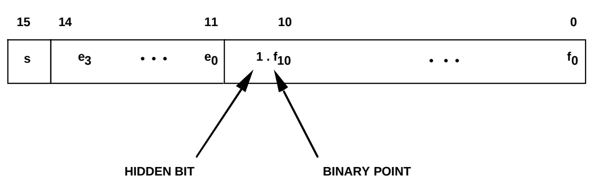

Figure 28-4: 16-Bit Floating-Point Format

Packing for Floating-Point Data

Two shifter instructions, `FPACK` and `FUNPACK`, perform the packing and unpacking conversions between 32-bit floating-point words and 16-bit floating-point words. The `FPACK` instruction converts a 32-bit IEEE floating-point number to a 16-bit floating-point number. The `FUNPACK` instruction converts 16-bit floating-point numbers back to 32-bit IEEE floating-point. Each instruction executes in a single cycle. The results of the `FPACK` and `FUNPACK` operations appear in the ***FPACK Operations*** and ***FUNPACK Operations*** tables.

Fixed-Point Formats

The processor supports two 32-bit fixed-point formats-fractional and integer. In both formats, numbers can be signed (two's-complement) or unsigned. The four possible combinations are shown in the ***32-Bit Fixed-Point Formats*** figure. In the fractional format, there is an implied binary point to the left of the most significant magnitude bit. In integer format, the binary point is understood to be to the right of the LSB. Note that the sign bit is negatively weighted in a two's-complement format.

If one operand is signed and the other unsigned, the result is signed. If both inputs are signed, the result is signed and automatically shifted left one bit. The LSB becomes zero and bit 62 moves into the sign bit position. Normally bit 63 and bit 62 are identical when both operands are signed. (The only exception is full-scale negative multiplied by itself.) Thus, the left-shift normally removes a redundant sign bit, increasing the precision of the most significant product. Also, if the data format is fractional, a single bit left-shift renormalizes the MSP to a fractional format. The signed formats with and without left-shifting are shown in the ***64-Bit Unsigned and Signed Fixed-Point Product*** figure.

ALU outputs have the same width and data format as the inputs. The multiplier, however, produces a 64-bit product from two 32-bit inputs. If both operands are unsigned integers, the result is a 64-bit unsigned integer. If both operands are unsigned fractions, the result is a 64-bit unsigned fraction. These formats are shown in the ***64-Bit Unsigned and Signed Fixed-Point Product*** figure.

The multiplier has an 80-bit accumulator to allow the accumulation of 64-bit products. For more information on the multiplier and accumulator, see ***Multiplier*** in the Processing Elements chapter.

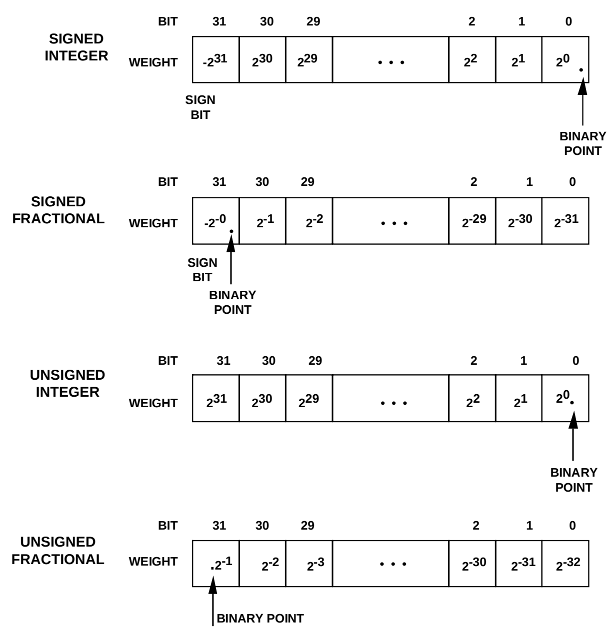

Figure 28-5: 32-Bit Fixed-Point Formats

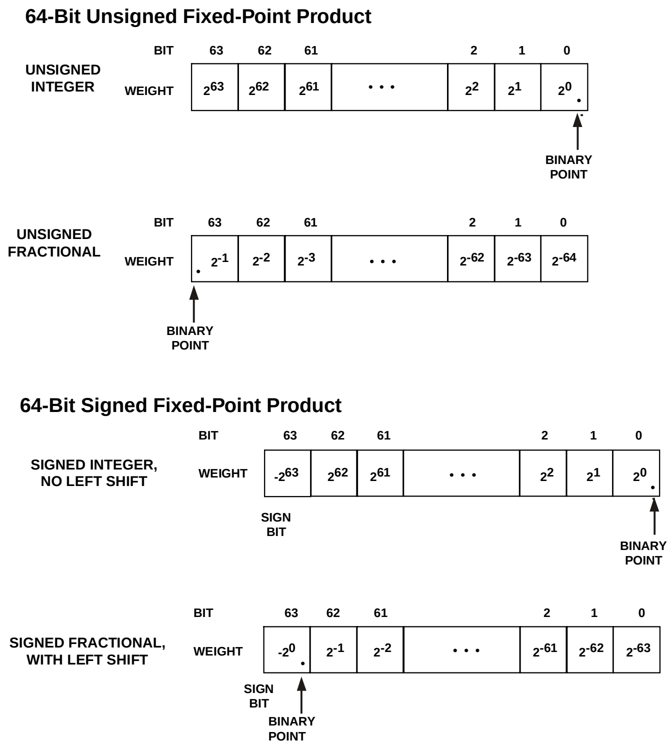

Figure 28-6: 64-Bit Unsigned and Signed Fixed-Point Product

29 SHARC-PLUS REGF Register Descriptions

SHARC+ Core (REGF) contains the following registers.

Table 29-1: SHARC-PLUS REGF Register List

Name	Description
REGF_ASTATX	Arithmetic Status (PEx) Register
REGF_ASTATY	Arithmetic Status (PEy) Register
REGF_B[n]	Base (Circular Buffer) Registers
REGF_CURLCNTR	Current Loop Counter Register
REGF_DADDR	Decode Address Register
REGF_EMUCLK	Emulation Counter Register
REGF_EMUCLK2	Emulation Counter Register 2
REGF_FADDR	Instruction Pipeline Stage Address Register
REGF_FLAGS	Flag I/O Register
REGF_IMASK	Interrupt Mask Register
REGF_IMASKP	Interrupt Mask Pointer Register
REGF_IRPTL	Interrupt Latch Register
REGF_I[n]	Index Registers
REGF_LADDR	Loop Address Stack Register
REGF_LCNTR	Loop Counter Register
REGF_L[n]	Length (Circular Buffer) Registers
REGF_MMASK	Mode Mask Register
REGF_MODE1	Mode Control 1 Register
REGF_MODE1STK	Mode 1 Stack (Top Entry) Register
REGF_MODE2	Mode Control 2 Register
REGF_MR0B	Multiplier Results 0 (PEx) Background Register
REGF_MR0F	Multiplier Results 0 (PEx) Foreground Register

Table 29-1: SHARC-PLUS REGF Register List (Continued)

Name	Description
REGF_MR1B	Multiplier Results 1 (PEx) Background Register
REGF_MR1F	Multiplier Results 1 (PEx) Foreground Register
REGF_MR2B	Multiplier Results 2 (PEx) Background Register
REGF_MR2F	Multiplier Results 2 (PEx) Foreground Register
REGF_MRB	Multiplier Results (PEx) Background Register
REGF_MRF	Multiplier Results (PEx) Foreground Register
REGF_MS0B	Multiplier Results 0 (PEy) Background Register
REGF_MS0F	Multiplier Results 0 (PEy) Foreground Register
REGF_MS1B	Multiplier Results 1 (PEy) Background Register
REGF_MS1F	Multiplier Results 1 (PEy) Foreground Register
REGF_MS2B	Multiplier Results 2 (PEy) Background Register
REGF_MS2F	Multiplier Results 2 (PEy) Foreground Register
REGF_MSB	Multiplier Results (PEy) Background Register
REGF_MSF	Multiplier Results (PEy) Foreground Register
REGF_M[n]	Modify Registers
REGF_PC	Program Counter Register
REGF_PCSTK	Program Counter Stack Register
REGF_PCSTKP	Program Counter Stack Pointer Register
REGF_PX	PMD-DMD Bus Exchange Register
REGF_PX1	PMD-DMD Bus Exchange 1 Register
REGF_PX2	PMD-DMD Bus Exchange 2 Register
REGF_R[n]	Register File (PEx) Data Registers (Rx, Fx)
REGF_STKYX	Sticky Status (PEx) Register
REGF_STKYY	Sticky Status (PEy) Register
REGF_S[n]	Register File (PEy) Data Registers (Sx, SFx)
REGF_TCOUNT	Timer Count Register
REGF_TPERIOD	Timer Period Register
REGF_USTAT1	User-Defined Status 1 Register
REGF_USTAT2	User-Defined Status 2 Register
REGF_USTAT3	User-Defined Status 3 Register
REGF_USTAT4	User-Defined Status 4 Register

Arithmetic Status (PEx) Register

The REGF_ASTATX register indicates status for processing element x (PEx) operations. If this register is loaded manually, there is a one cycle effect latency before the new value in the REGF_ASTATX register can be used in a conditional instruction.

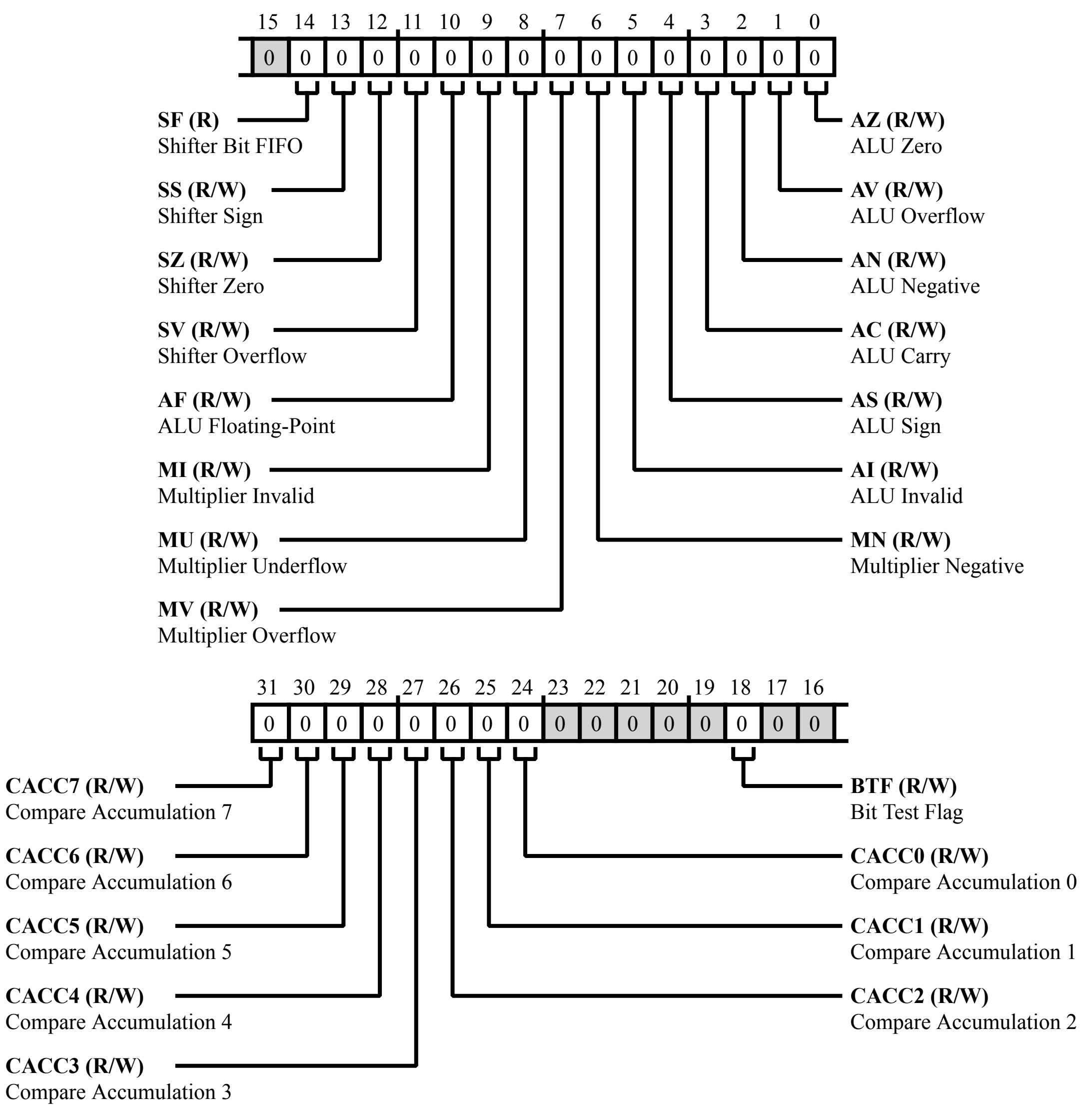

Figure 29-1: REGF_ASTATX Register Diagram

Table 29-2: REGF_ASTATX Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31 (R/W)	CACC7	Compare Accumulation 7. The REGF_ASTATX.CACC7 bit indicates which operand was greater during the most recent ALU compare operation: X input (if set, = 1) or Y input (if cleared, = 0). The CACC bits form a right-shift register, each storing a previous compare accumulation result. With each new compare, the processor right shifts the values of CACC, storing the newest value in the REGF_ASTATX.CACC7 bit and storing the oldest value in the REGF_ASTATX.CACC0 bit.
30 (R/W)	CACC6	Compare Accumulation 6. The REGF_ASTATX.CACC6 bit indicates which operand was greater during a previous ALU compare operation. For more information, see the REGF_ASTATX.CACC7 bit description.
29 (R/W)	CACC5	Compare Accumulation 5. The REGF_ASTATX.CACC5 bit indicates which operand was greater during a previous ALU compare operation. For more information, see the REGF_ASTATX.CACC7 bit description.
28 (R/W)	CACC4	Compare Accumulation 4. The REGF_ASTATX.CACC4 bit indicates which operand was greater during a previous ALU compare operation. For more information, see the REGF_ASTATX.CACC7 bit description.
27 (R/W)	CACC3	Compare Accumulation 3. The REGF_ASTATX.CACC3 bit indicates which operand was greater during a previous ALU compare operation. For more information, see the REGF_ASTATX.CACC7 bit description.
26 (R/W)	CACC2	Compare Accumulation 2. The REGF_ASTATX.CACC2 bit indicates which operand was greater during a previous ALU compare operation. For more information, see the REGF_ASTATX.CACC7 bit description.
25 (R/W)	CACC1	Compare Accumulation 1. The REGF_ASTATX.CACC1 bit indicates which operand was greater during a previous ALU compare operation. For more information, see the REGF_ASTATX.CACC7 bit description.
24 (R/W)	CACC0	Compare Accumulation 0. The REGF_ASTATX.CACC0 bit indicates which operand was greater during a previous ALU compare operation. For more information, see the REGF_ASTATX.CACC7 bit description.

Table 29-2: REGF_ASTATX Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
18 (R/W)	BTF	Bit Test Flag. The REGF_ASTATX.BTF bit indicates whether the system register bit is true (if set, = 1) or false (if cleared, = 0). The processor sets REGF_ASTATX.BTF when the bit(s) in a system register and value in the Bit Tst instruction match. The processor also sets REGF_ASTATX.BTF when the bit(s) in a system register and value in the Bit Xor instruction match.
14 (R/NW)	SF	Shifter Bit FIFO. The REGF_ASTATX.SF bit indicates the current value of bit FIFO write pointer. This bit is set (=1) when the write pointer is greater than or equal to 32 (FIFO is half full). Otherwise, the bit is cleared.
13 (R/W)	SS	Shifter Sign. The REGF_ASTATX.SS bit indicates whether the most recent shifter operation's input was negative (if set, = 1) or positive (if cleared, = 0). The shifter updates this bit for all shifter operations.
12 (R/W)	SZ	Shifter Zero. The REGF_ASTATX.SZ bit indicates whether the most recent shifter operation's result was zero (if set, = 1) or non-zero (if cleared, = 0). The shifter updates this bit for all shifter operations. The processor also sets REGF_ASTATX.SZ if the shifter operation performs a bit test on a bit outside of the 32-bit fixed-point field.
11 (R/W)	SV	Shifter Overflow. The REGF_ASTATX.SV bit indicates whether the most recent shifter operation's result overflowed (if set, = 1) or did not overflow (if cleared, = 0). The shifter updates this bit for all shifter operations. The processor sets REGF_ASTATX.SV if the shifter operation: • Shifts the significant bits to the left of the 32-bit fixed-point field • Tests, sets, or clears a bit outside of the 32-bit fixed-point field • Extracts a field that is past or crosses the left edge of the 32-bit fixed-point field • Performs a LEFTZ or LEFTO operation that returns a result of 32
10 (R/W)	AF	ALU Floating-Point. The REGF_ASTATX.AF bit indicates whether the most recent ALU operation was floating-point (if set, = 1) or fixed-point (if cleared, = 0). The ALU updates REGF_ASTATX.AF for all fixed-point and floating-point ALU operations.

Table 29-2: REGF_ASTATX Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
9 (R/W)	MI	Multiplier Invalid. The REGF_ASTATX.MI bit indicates whether the most recent multiplier operation's input was invalid (if set, = 1) or valid (if cleared, = 0). The multiplier updates this bit for floating-point multiplier operations. The processor sets the MI bit and the REGF_STKYX.MIS bit if the ALU operation: • Receives a NAN input operand • Receives an Infinity and zero as input operands
8 (R/W)	MU	Multiplier Underflow. The REGF_ASTATX.MU bit indicates whether the most recent multiplier operation's result underflowed (if set, = 1) or did not underflow (if cleared, = 0). The multiplier updates this bit for all fixed- and floating-point multiplier operations. For floating-point results, the processor sets MU and the REGF_STKYX.MUS bit if the floating-point result underflows (unbiased exponent < -126). Denormal operands are treated as zeros. So, they never cause underflows. For fixed-point results, the processor sets MU and the REGF_STKYX.MUS bit if the result of the multiplier operation is: • Two's-complement, fractional: with upper 48 bits all zeros or all ones, lower 32 bits not all zeros • Unsigned, fractional: with upper 48 bits all zeros, lower 32 bits not all zeros If the multiplier operation directs a fixed-point, fractional result to an MR register, the processor places the underflowed portion of the result in MR0.
7 (R/W)	MV	Multiplier Overflow. The REGF_ASTATX.MV bit indicates whether the most recent multiplier operation's result overflowed (if set, = 1) or did not overflow (if cleared, = 0). The multiplier updates this bit for all fixed-point and floating-point multiplier operations. For floating-point results, the processor sets MV and the REGF_STKYX.MVS bit if the rounded result overflows (unbiased exponent > 127). For fixed-point results, the processor sets the MV bit and the REGF_STKYX.MOS bit register if the result of the multiplier operation is: • Two's-complement, fractional with the upper 17 bits of MR not all zeros or all ones • Two's-complement, integer with the upper 49 bits of MR not all zeros or all ones • Unsigned, fractional with the upper 16 bits of MR not all zeros • Unsigned, integer with the upper 48 bits of MR not all zeros If the multiplier operation directs a fixed-point result to an MR register, the processor places the overflowed portion of the result in MR1 and MR2 for an integer result or places it in MR2 only for a fractional result.

Table 29-2: REGF_ASTATX Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
6 (R/W)	MN	Multiplier Negative. The REGF_ASTATX.MN bit indicates whether the most recent multiplier operation's result was negative (if set, = 1) or positive (if cleared, = 0). The multiplier updates this bit for all fixed- and floating-point multiplier operations.
5 (R/W)	AI	ALU Invalid. The REGF_ASTATX.AI indicates whether the most recent ALU operation's input was invalid (if set, = 1) or valid (if cleared, = 0). The ALU updates REGF_ASTATX.AI for all fixed- and floating-point ALU operations. The processor sets REGF_ASTATX.AI, REGF_STKYX.AIS, and REGF_STKYY.AIS if the ALU operation: • Receives a NAN input operand • Adds opposite-signed infinities • Subtracts like-signed infinities • Overflows during a floating-point to fixed-point conversion when saturation mode is not set • Operates on an infinity during a floating-point to fixed-point operation when the saturation mode is not set
4 (R/W)	AS	ALU Sign. The REGF_ASTATX.AS bit indicates whether the most recent ALU ABS or MANT operation's input was negative (if set, = 1) or positive (if cleared, = 0). The ALU updates REGF_ASTATX.AS only for fixed- and floating-point ABS and MANT operations. The ALU clears REGF_ASTATX.AS for all operations other than ABS and MANT.
3 (R/W)	AC	ALU Carry. The REGF_ASTATX.AC bit indicates whether the the most recent fixed-point ALU operation had a carry out of the MSB of the result (if set, = 1) or had no carry (if cleared, = 0). The ALU updates REGF_ASTATX.AC for all fixed-point operations. The processor clears REGF_ASTATX.AC during the fixed-point logic operations: PASS, MIN, MAX, COMP, ABS, and CLIP. The ALU reads REGF_ASTATX.AC for the fixed-point accumulate operations: Addition with Carry and Fixed-point Subtraction with Carry.
2 (R/W)	AN	ALU Negative. The REGF_ASTATX.AN bit indicates whether the most recent ALU operation's result was negative (if set, = 1) or positive (if cleared, = 0). The ALU updates REGF_ASTATX.AN for all fixed-point and floating-point ALU operations.

Table 29-2: REGF_ASTATX Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
1 (R/W)	AV	ALU Overflow. The REGF_ASTATX.AV bit indicates whether the most recent ALU operation's result overflowed (if set, = 1) or did not overflow (if cleared, = 0). The ALU updates REGF_ASTATX.AV for all fixed-point and floating-point ALU operations. For fixed-point results, the processor sets REGF_ASTATX.AV, REGF_STKYX.AOS, and REGF_STKYY.AOS when the XOR of the two most significant bits (MSBs) is a 1. For floating-point results, the processor sets REGF_ASTATX.AV, REGF_STKYX.AVS, and REGF_STKYY.AVS when the rounded result overflows (unbiased exponent > 127).
0 (R/W)	AZ	ALU Zero. The REGF_ASTATX.AZ bit indicates whether the most recent ALU operation's result was zero (if set, = 1) or non-zero (if cleared, = 0). The ALU updates REGF_ASTATX.AZ for all fixed-point and floating-point ALU operations. This bit also can indicate a floating-point underflow. During an ALU underflow (indicated by a set (= 1) REGF_STKYY.AUS bit), the processor sets REGF_ASTATX.AZ if the floating-point result is smaller than can be represented in the output format.

Arithmetic Status (PEy) Register

The REGF_ASTATY register indicates status for processing element y (PEy) operations. If this register is loaded manually, there is a one cycle effect latency before the new value in the REGF_ASTATY register can be used in a conditional instruction.

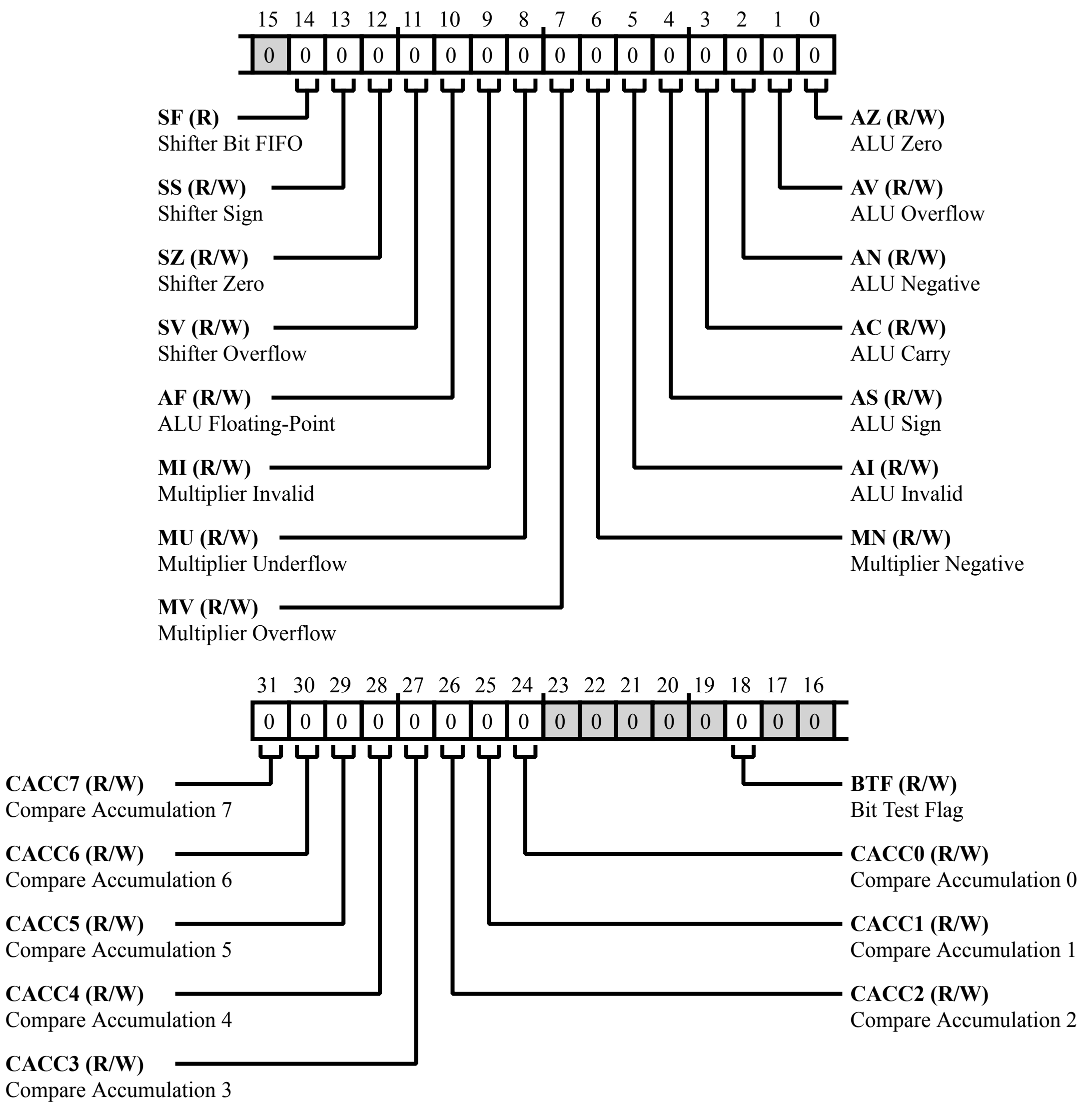

Figure 29-2: REGF_ASTATY Register Diagram

Table 29-3: REGF_ASTATY Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31 (R/W)	CACC7	Compare Accumulation 7. The REGF_ASTATY.CACC7 bit indicates which operand was greater during the most recent ALU compare operation: X input (if set, = 1) or Y input (if cleared, = 0). The CACC bits form a right-shift register, each storing a previous compare accumulation result. With each new compare, the processor right shifts the values of CACC, storing the newest value in the REGF_ASTATY.CACC7 bit and storing the oldest value in the REGF_ASTATY.CACC0 bit.
30 (R/W)	CACC6	Compare Accumulation 6. The REGF_ASTATY.CACC6 bit indicates which operand was greater during a previous ALU compare operation. For more information, see the REGF_ASTATY.CACC7 bit description.
29 (R/W)	CACC5	Compare Accumulation 5. The REGF_ASTATY.CACC5 bit indicates which operand was greater during a previous ALU compare operation. For more information, see the REGF_ASTATY.CACC7 bit description.
28 (R/W)	CACC4	Compare Accumulation 4. The REGF_ASTATY.CACC4 bit indicates which operand was greater during a previous ALU compare operation. For more information, see the REGF_ASTATY.CACC7 bit description.
27 (R/W)	CACC3	Compare Accumulation 3. The REGF_ASTATY.CACC3 bit indicates which operand was greater during a previous ALU compare operation. For more information, see the REGF_ASTATY.CACC7 bit description.
26 (R/W)	CACC2	Compare Accumulation 2. The REGF_ASTATY.CACC2 bit indicates which operand was greater during a previous ALU compare operation. For more information, see the REGF_ASTATY.CACC7 bit description.
25 (R/W)	CACC1	Compare Accumulation 1. The REGF_ASTATY.CACC1 bit indicates which operand was greater during a previous ALU compare operation. For more information, see the REGF_ASTATY.CACC7 bit description.
24 (R/W)	CACC0	Compare Accumulation 0. The REGF_ASTATY.CACC0 bit indicates which operand was greater during a previous ALU compare operation. For more information, see the REGF_ASTATY.CACC7 bit description.

Table 29-3: REGF_ASTATY Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
18 (R/W)	BTF	Bit Test Flag. The REGF_ASTATY.BTF bit indicates whether the system register bit is true (if set, = 1) or false (if cleared, = 0). The processor sets REGF_ASTATY.BTF when the bit(s) in a system register and value in the Bit Tst instruction match. The processor also sets REGF_ASTATY.BTF when the bit(s) in a system register and value in the Bit Xor instruction match.
14 (R/NW)	SF	Shifter Bit FIFO. The REGF_ASTATY.SF bit indicates the current value of bit FIFO write pointer. This bit is set (=1) when the write pointer is greater than or equal to 32 (FIFO is half full). Otherwise, the bit is cleared.
13 (R/W)	SS	Shifter Sign. The REGF_ASTATY.SS bit indicates whether the most recent shifter operation's input was negative (if set, = 1) or positive (if cleared, = 0). The shifter updates this bit for all shifter operations.
12 (R/W)	SZ	Shifter Zero. The REGF_ASTATY.SZ bit indicates whether the most recent shifter operation's result was zero (if set, = 1) or non-zero (if cleared, = 0). The shifter updates this bit for all shifter operations. The processor also sets REGF_ASTATY.SZ if the shifter operation performs a bit test on a bit outside of the 32-bit fixed-point field.
11 (R/W)	SV	Shifter Overflow. The REGF_ASTATY.SV bit indicates whether the most recent shifter operation's result overflowed (if set, = 1) or did not overflow (if cleared, = 0). The shifter updates this bit for all shifter operations. The processor sets REGF_ASTATY.SV if the shifter operation: • Shifts the significant bits to the left of the 32-bit fixed-point field • Tests, sets, or clears a bit outside of the 32-bit fixed-point field • Extracts a field that is past or crosses the left edge of the 32-bit fixed-point field • Performs a LEFTZ or LEFTO operation that returns a result of 32
10 (R/W)	AF	ALU Floating-Point. The REGF_ASTATY.AF bit indicates whether the most recent ALU operation was floating-point (if set, = 1) or fixed-point (if cleared, = 0). The ALU updates REGF_ASTATY.AF for all fixed-point and floating-point ALU operations.

Table 29-3: REGF_ASTATY Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
9 (R/W)	MI	Multiplier Invalid. The REGF_ASTATY.MI bit indicates whether the most recent multiplier operation's input was invalid (if set, = 1) or valid (if cleared, = 0). The multiplier updates this bit for floating-point multiplier operations. The processor sets the MI bit and the REGF_STKYY.MIS bit if the ALU operation: • Receives a NAN input operand • Receives an Infinity and zero as input operands
8 (R/W)	MU	Multiplier Underflow. The REGF_ASTATY.MU bit indicates whether the most recent multiplier operation's result underflowed (if set, = 1) or did not underflow (if cleared, = 0). The multiplier updates this bit for all fixed- and floating-point multiplier operations. For floating-point results, the processor sets MU and the REGF_STKYY.MUS bit if the floating-point result underflows (unbiased exponent < -126). Denormal operands are treated as zeros. So, they never cause underflows. For fixed-point results, the processor sets MU and the REGF_STKYY.MUS bit if the result of the multiplier operation is: • Two's-complement, fractional: with upper 48 bits all zeros or all ones, lower 32 bits not all zeros • Unsigned, fractional: with upper 48 bits all zeros, lower 32 bits not all zeros If the multiplier operation directs a fixed-point, fractional result to an MR register, the processor places the underflowed portion of the result in MR0.
7 (R/W)	MV	Multiplier Overflow. The REGF_ASTATY.MV bit indicates whether the most recent multiplier operation's result overflowed (if set, = 1) or did not overflow (if cleared, = 0). The multiplier updates this bit for all fixed-point and floating-point multiplier operations. For floating-point results, the processor sets MV and the REGF_STKYY.MVS bit if the rounded result overflows (unbiased exponent > 127). For fixed-point results, the processor sets the MV bit and the REGF_STKYY.MOS bit register if the result of the multiplier operation is: • Two's-complement, fractional with the upper 17 bits of MR not all zeros or all ones • Two's-complement, integer with the upper 49 bits of MR not all zeros or all ones • Unsigned, fractional with the upper 16 bits of MR not all zeros • Unsigned, integer with the upper 48 bits of MR not all zeros If the multiplier operation directs a fixed-point result to an MR register, the processor places the overflowed portion of the result in MR1 and MR2 for an integer result or places it in MR2 only for a fractional result.

Table 29-3: REGF_ASTATY Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
6 (R/W)	MN	Multiplier Negative. The REGF_ASTATY.MN bit indicates whether the most recent multiplier operation's result was negative (if set, = 1) or positive (if cleared, = 0). The multiplier updates this bit for all fixed- and floating-point multiplier operations.
5 (R/W)	AI	ALU Invalid. The REGF_ASTATY.AI indicates whether the most recent ALU operation's input was invalid (if set, = 1) or valid (if cleared, = 0). The ALU updates REGF_ASTATY.AI for all fixed- and floating-point ALU operations. The processor sets REGF_ASTATY.AI, REGF_STKYX.AIS, and REGF_STKYY.AIS if the ALU operation: • Receives a NAN input operand • Adds opposite-signed infinities • Subtracts like-signed infinities • Overflows during a floating-point to fixed-point conversion when saturation mode is not set • Operates on an infinity during a floating-point to fixed-point operation when the saturation mode is not set
4 (R/W)	AS	ALU Sign. The REGF_ASTATY.AS bit indicates whether the most recent ALU ABS or MANT operation's input was negative (if set, = 1) or positive (if cleared, = 0). The ALU updates REGF_ASTATY.AS only for fixed- and floating-point ABS and MANT operations. The ALU clears REGF_ASTATY.AS for all operations other than ABS and MANT.
3 (R/W)	AC	ALU Carry. The REGF_ASTATY.AC bit indicates whether the the most recent fixed-point ALU operation had a carry out of the MSB of the result (if set, = 1) or had no carry (if cleared, = 0). The ALU updates REGF_ASTATY.AC for all fixed-point operations. The processor clears REGF_ASTATY.AC during the fixed-point logic operations: PASS, MIN, MAX, COMP, ABS, and CLIP. The ALU reads REGF_ASTATY.AC for the fixed-point accumulate operations: Addition with Carry and Fixed-point Subtraction with Carry.
2 (R/W)	AN	ALU Negative. The REGF_ASTATY.AN bit indicates whether the most recent ALU operation's result was negative (if set, = 1) or positive (if cleared, = 0). The ALU updates REGF_ASTATY.AN for all fixed-point and floating-point ALU operations.

Table 29-3: REGF_ASTATY Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
1 (R/W)	AV	ALU Overflow. The REGF_ASTATY.AV bit indicates whether the most recent ALU operation's result overflowed (if set, = 1) or did not overflow (if cleared, = 0). The ALU updates REGF_ASTATY.AV for all fixed-point and floating-point ALU operations. For fixed-point results, the processor sets REGF_ASTATY.AV, REGF_STKYX.AOS, and REGF_STKYY.AOS when the XOR of the two most significant bits (MSBs) is a 1. For floating-point results, the processor sets REGF_ASTATY.AV, REGF_STKYX.AVS, and REGF_STKYY.AVS when the rounded result overflows (unbiased exponent > 127).
0 (R/W)	AZ	ALU Zero. The REGF_ASTATY.AZ bit indicates whether the most recent ALU operation's result was zero (if set, = 1) or non-zero (if cleared, = 0). The ALU updates REGF_ASTATY.AZ for all fixed-point and floating-point ALU operations. This bit also can indicate a floating-point underflow. During an ALU underflow (indicated by a set (= 1) REGF_STKYX.AUS bit), the processor sets REGF_ASTATY.AZ if the floating-point result is smaller than can be represented in the output format.

Base (Circular Buffer) Registers

The data address generators (DAGs) control circular buffering operations with length (REGF_L[n]) registers and base (REGF_B[n]) registers. Registers L0 through L7 and B0 through B7 are for DAG1, and registers L8 through L15 and B8 through B15 are for DAG2. Length and base registers set up the range of addresses and the starting address for a circular buffer.

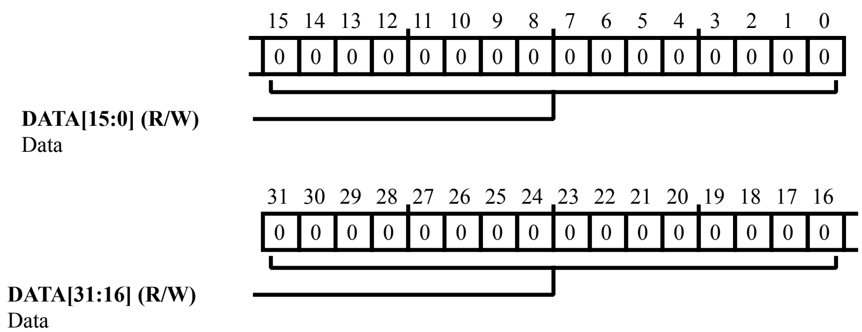

Figure 29-3: REGF_B[n] Register Diagram

Table 29-4: REGF_B[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_B[n].DATA bit field contains circular buffer base address data.

Current Loop Counter Register

The current loop counter (REGF_CURLCNTR) register provides access to the loop counter stack and tracks iterations for the DO UNTIL LCE loop being executed. For more information about using the REGF_CURLCNTR register, see the Loop Counter Stack Access section.

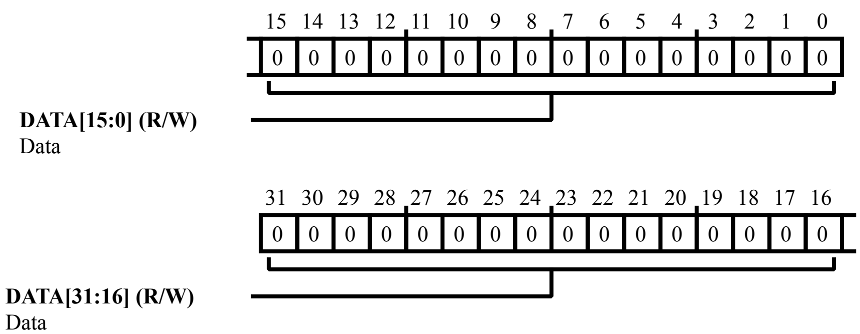

Figure 29-4: REGF_CURLCNTR Register Diagram

Table 29-5: REGF_CURLCNTR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_CURLCNTR.DATA bit field contains data.

Decode Address Register

The decode address (REGF_DADDR) register reads the third stage (D) in the instruction pipeline and contains the 24-bit address of the instruction that the processor decodes on the next cycle.

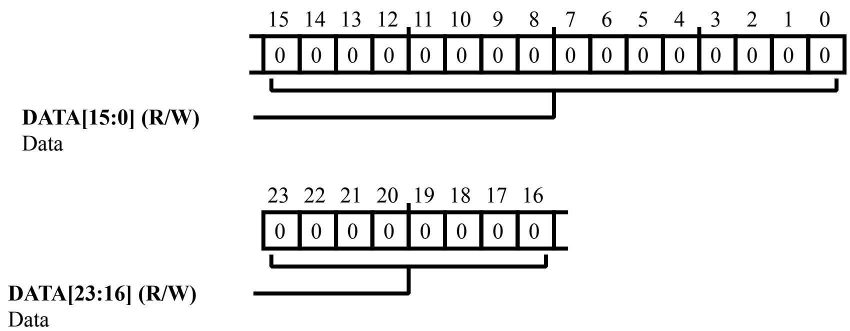

Figure 29-5: REGF_DADDR Register Diagram

Table 29-6: REGF_DADDR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
23:0 (R/W)	DATA	Data. The REGF_DADDR.DATA bit field contains decode address data.

Emulation Counter Register

The REGF_EMUCLK register is read-only from user-space and can be written only when the processor is in emulation space.

The emulation clock counter consists of a 32-bit count register (REGF_EMUCLK) and a 32-bit scaling register (REGF_EMUCLK2). The REGF_EMUCLK counts core clock cycles while the user has control of the processor and stops counting when the emulator gains control. These registers let you gauge the amount of time spent executing a particular section of code. The REGF_EMUCLK2 register extends the time REGF_EMUCLK can count by incrementing each time the REGF_EMUCLK value rolls over to zero. The combined emulation clock counter can count accurately for thousands of hours. Note that the counters increment during an idle instruction.

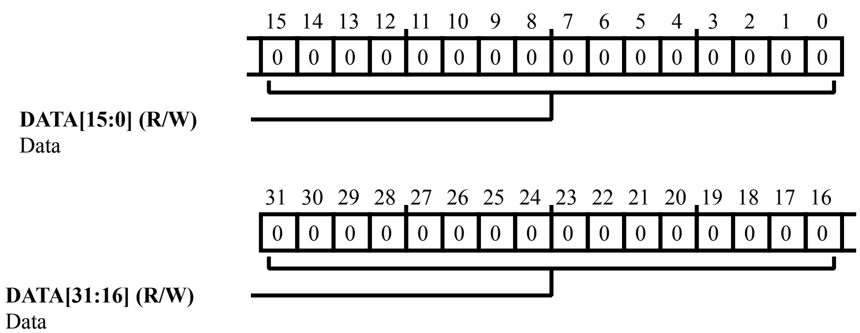

Figure 29-6: REGF_EMUCLK Register Diagram

Table 29-7: REGF_EMUCLK Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_EMUCLK.DATA bit field contains data.

Emulation Counter Register 2

The REGF_EMUCLK2 register is read-only from user-space and can be written only when the processor is in emulation space.

The emulation clock counter consists of a 32-bit count register (REGF_EMUCLK) and a 32-bit scaling register (REGF_EMUCLK2). The REGF_EMUCLK counts core clock cycles while the user has control of the processor and stops counting when the emulator gains control. These registers let you determine the amount of time spent executing a particular section of code. The REGF_EMUCLK2 register extends the time REGF_EMUCLK can count by incrementing each time the REGF_EMUCLK value rolls over to zero. The combined emulation clock counter can count accurately for thousands of hours. Note that the counters increment during an idle instruction.

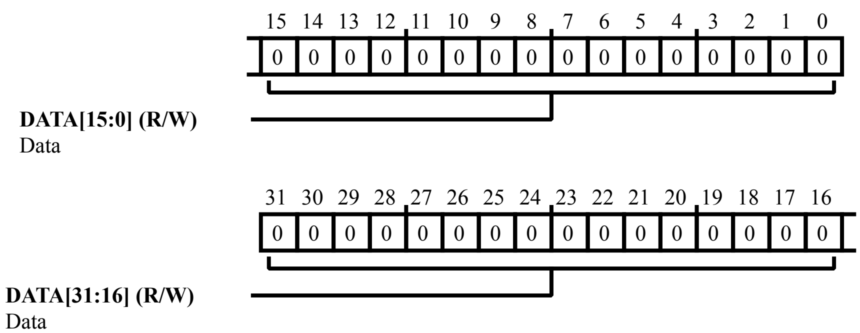

Figure 29-7: REGF_EMUCLK2 Register Diagram

Table 29-8: REGF_EMUCLK2 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_EMUCLK2.DATA bit field contains data.

Instruction Pipeline Stage Address Register

The instruction pipeline stage address REGF_FADDR register holds the addresses based on the pipeline stages. There are 11 registers:

#define F1ADDR 0x300E0

#define F2ADDR 0x300E1

#define F3ADDR 0x300E2

#define F4ADDR 0x300E3

#define M1ADDR 0x300E6

#define M2ADDR 0x300E7

#define M3ADDR 0x300E8

#define M4ADDR 0x300E9

#define D1ADDR 0x300E4

#define D2ADDR 0x300E5

#define E2ADDR 0x300EA

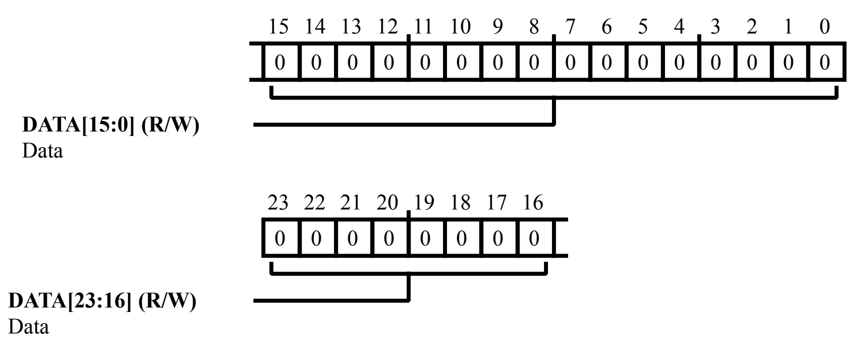

Figure 29-8: REGF_FADDR Register Diagram

Table 29-9: REGF_FADDR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
23:0 (R/W)	DATA	Data. The REGF_FADDR.DATA bit field contains fetch address data.

Flag I/O Register

The SHARC+ core provides direct instruction support for setting/resetting/reading the four FLAGs. The REGF_FLAGS register indicates the state of the FLAGx pins. When a FLAGx pin is an output, the processor outputs a high in response to a program setting the bit in the REGF_FLAGS register. The I/O direction (input or output) selection of each bit is controlled by its corresponding REGF_FLAGS.FLG0O, REGF_FLAGS.FLG1O, REGF_FLAGS.FLG2O, or REGF_FLAGS.FLG3O bit.

Programs can not change the output selects of the REGF_FLAGS register and provide a new value in the same instruction. Instead, programs must use two write instructions-the first to change the output select of a particular FLAG pin, and the second to provide the new value as shown in the example:

```
bit set FLAGS FLG1O; /* set Flag1 IO output */
bit set FLAGS FLG1; /* set Flag1 level 1 */
```

In the REGF_FLAGS register bit definitions, note that:

- For all FLGx bits, FLAGx values are as follows: 0 = low, 1 = high.
- For all FLGxO bits, FLAGx output selects are as follows: 0 = FLAGx Input, 1 = FLAGx Output.
- The REGF_FLAGS.FLG0, REGF_FLAGS.FLG1, REGF_FLAGS.FLG2, and REGF_FLAGS.FLG3 bits can be immediately used for conditional instruction.

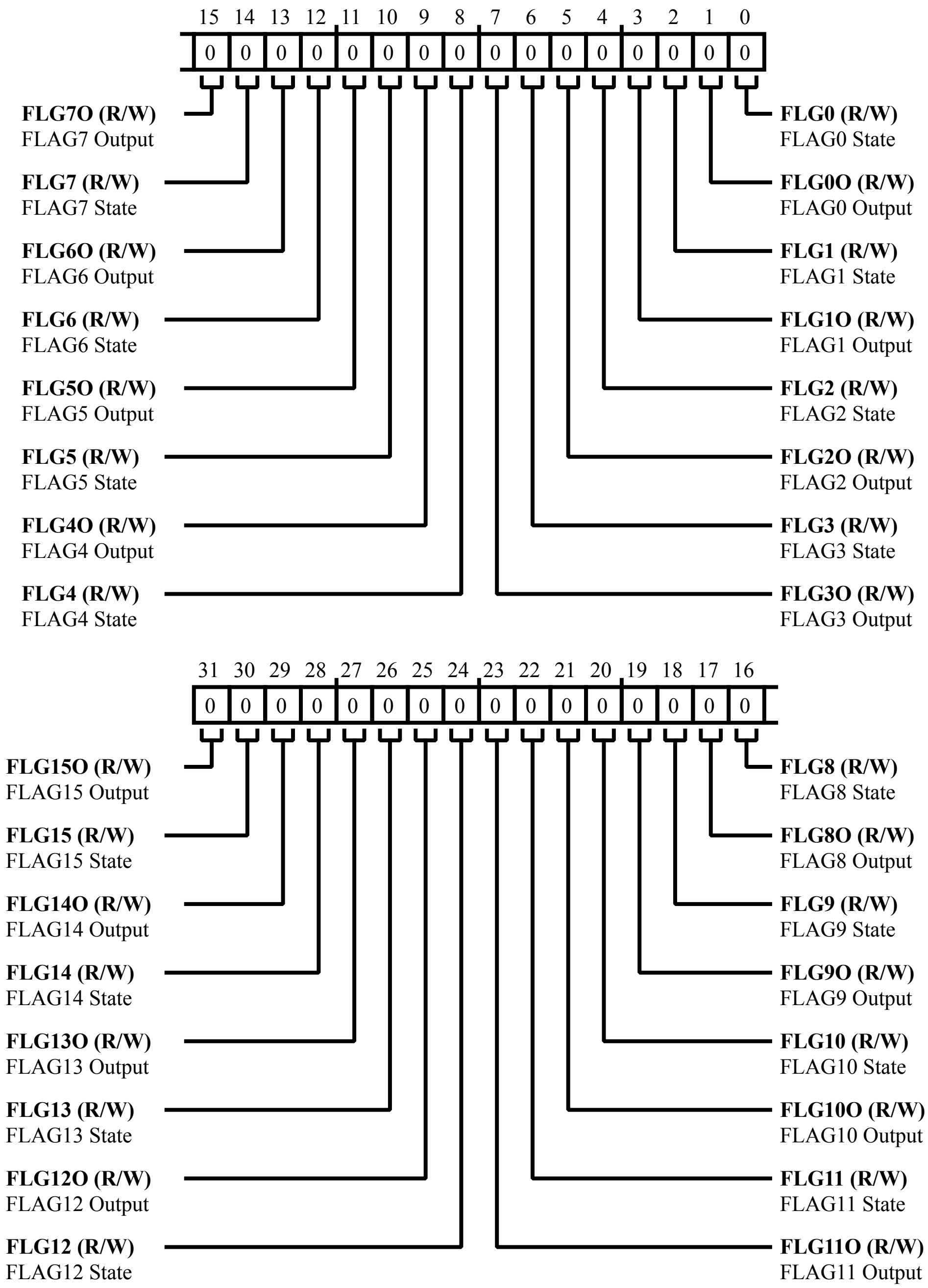

Figure 29-9: REGF_FLAGS Register Diagram

Table 29-10: REGF_FLAGS Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
31 (R/W)	FLG15O	FLAG15 Output. The REGF_FLAGS.FLG15O bit selects the I/O direction for the FLAG15 pin. The I/O direction of the pin is programmed as an output (if bit set, = 1) or input (if bit cleared, = 0). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.	
		0	Input
		1	Output
30 (R/W)	FLG15	FLAG15 State. The REGF_FLAGS.FLG15 bit indicates the state of the FLAG15 pin as high (if set, = 1) or low (if cleared, = 0). This bit is only available in the ADSP-SC59x/ ADSP-2159x processor.	
29 (R/W)	FLG14O	FLAG14 Output. The REGF_FLAGS.FLG14O bit selects the I/O direction for the FLAG14 pin. The I/O direction of the pin is programmed as an output (if bit set, = 1) or input (if bit cleared, = 0). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.	
		0	Input
		1	Output
28 (R/W)	FLG14	FLAG14 State. The REGF_FLAGS.FLG14 bit indicates the state of the FLAG14 pin as high (if set, = 1) or low (if cleared, = 0). This bit is only available in the ADSP-SC59x/ ADSP-2159x processor.	
27 (R/W)	FLG13O	FLAG13 Output. The REGF_FLAGS.FLG13O bit selects the I/O direction for the FLAG13 pin. The I/O direction of the pin is programmed as an output (if bit set, = 1) or input (if bit cleared, = 0). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.	
		0	Input
		1	Output
26 (R/W)	FLG13	FLAG13 State. The REGF_FLAGS.FLG13 bit indicates the state of the FLAG13 pin as high (if set, = 1) or low (if cleared, = 0). This bit is only available in the ADSP-SC59x/ ADSP-2159x processor.	
25 (R/W)	FLG12O	FLAG12 Output. The REGF_FLAGS.FLG12O bit selects the I/O direction for the FLAG12 pin. The I/O direction of the pin is programmed as an output (if bit set, = 1) or input (if bit cleared, = 0). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.	
		0	Input
		1	Output

Table 29-10: REGF_FLAGS Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
24 (R/W)	FLG12	FLAG12 State. The REGF_FLAGS.FLG12 bit indicates the state of the FLAG12 pin as high (if set, = 1) or low (if cleared, = 0). This bit is only available in the ADSP-SC59x/ ADSP-2159x processor.	
23 (R/W)	FLG11O	FLAG11 Output. The REGF_FLAGS.FLG11O bit selects the I/O direction for the FLAG11 pin. The I/O direction of the pin is programmed as an output (if bit set, = 1) or input (if bit cleared, = 0). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.	
		0	Input
		1	Output
22 (R/W)	FLG11	FLAG11 State. The REGF_FLAGS.FLG11 bit indicates the state of the FLAG11 pin as high (if set, = 1) or low (if cleared, = 0). This bit is only available in the ADSP-SC59x/ ADSP-2159x processor.	
21 (R/W)	FLG10O	FLAG10 Output. The REGF_FLAGS.FLG10O bit selects the I/O direction for the FLAG10 pin. The I/O direction of the pin is programmed as an output (if bit set, = 1) or input (if bit cleared, = 0). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.	
		0	Input
		1	Output
20 (R/W)	FLG10	FLAG10 State. The REGF_FLAGS.FLG10 bit indicates the state of the FLAG10 pin as high (if set, = 1) or low (if cleared, = 0). This bit is only available in the ADSP-SC59x/ ADSP-2159x processor.	
19 (R/W)	FLG9O	FLAG9 Output. The REGF_FLAGS.FLG9O bit selects the I/O direction for the FLAG9 pin. The I/O direction of the pin is programmed as an output (if bit set, = 1) or input (if bit cleared, = 0). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.	
		0	Input
		1	Output
18 (R/W)	FLG9	FLAG9 State. The REGF_FLAGS.FLG9 bit indicates the state of the FLAG9 pin as high (if set, = 1) or low (if cleared, = 0). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.	

Table 29-10: REGF_FLAGS Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
17 (R/W)	FLG8O	FLAG8 Output. The REGF_FLAGS.FLG8O bit selects the I/O direction for the FLAG8 pin. The I/O direction of the pin is programmed as an output (if bit set, = 1) or input (if bit cleared, = 0). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.	
		0	Input
		1	Output
16 (R/W)	FLG8	FLAG8 State. The REGF_FLAGS.FLG8 bit indicates the state of the FLAG8 pin as high (if set, = 1) or low (if cleared, = 0). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.	
15 (R/W)	FLG7O	FLAG7 Output. The REGF_FLAGS.FLG7O bit selects the I/O direction for the FLAG7 pin. The I/O direction of the pin is programmed as an output (if bit set, = 1) or input (if bit cleared, = 0). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.	
		0	Input
		1	Output
14 (R/W)	FLG7	FLAG7 State. The REGF_FLAGS.FLG7 bit indicates the state of the FLAG7 pin as high (if set, = 1) or low (if cleared, = 0). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.	
13 (R/W)	FLG6O	FLAG6 Output. The REGF_FLAGS.FLG6O bit selects the I/O direction for the FLAG6 pin. The I/O direction of the pin is programmed as an output (if bit set, = 1) or input (if bit cleared, = 0). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.	
		0	Input
		1	Output
12 (R/W)	FLG6	FLAG6 State. The REGF_FLAGS.FLG6 bit indicates the state of the FLAG6 pin as high (if set, = 1) or low (if cleared, = 0). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.	
11 (R/W)	FLG5O	FLAG5 Output. The REGF_FLAGS.FLG5O bit selects the I/O direction for the FLAG5 pin. The I/O direction of the pin is programmed as an output (if bit set, = 1) or input (if bit cleared, = 0). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.	
		0	Input
		1	Output

Table 29-10: REGF_FLAGS Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
10 (R/W)	FLG5	FLAG5 State. The REGF_FLAGS.FLG5 bit indicates the state of the FLAG5 pin as high (if set, = 1) or low (if cleared, = 0). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.	
9 (R/W)	FLG4O	FLAG4 Output. The REGF_FLAGS.FLG4O bit selects the I/O direction for the FLAG4 pin. The I/O direction of the pin is programmed as an output (if bit set, = 1) or input (if bit cleared, = 0). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.	
		0	Input
		1	Output
8 (R/W)	FLG4	FLAG4 State. The REGF_FLAGS.FLG4 bit indicates the state of the FLAG4 pin as high (if set, = 1) or low (if cleared, = 0). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.	
7 (R/W)	FLG3O	FLAG3 Output. The REGF_FLAGS.FLG3O bit selects the I/O direction for the FLAG3 pin. The I/O direction of the pin is programmed as an output (if bit set, = 1) or input (if bit cleared, = 0).	
		0	Input
		1	Output
6 (R/W)	FLG3	FLAG3 State. The REGF_FLAGS.FLG3 bit indicates the state of the FLAG3 pin as high (if set, = 1) or low (if cleared, = 0).	
5 (R/W)	FLG2O	FLAG2 Output. The REGF_FLAGS.FLG2O bit selects the I/O direction for the FLAG2 pin. The I/O direction of the pin is programmed as an output (if bit set, = 1) or input (if bit cleared, = 0).	
		0	Input
		1	Output
4 (R/W)	FLG2	FLAG2 State. The REGF_FLAGS.FLG2 bit indicates the state of the FLAG2 pin as high (if set, = 1) or low (if cleared, = 0).	

Table 29-10: REGF_FLAGS Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
3 (R/W)	FLG1O	FLAG1 Output. The REGF_FLAGS.FLG1O bit selects the I/O direction for the FLAG1 pin. The I/O direction of the pin is programmed as an output (if bit set, = 1) or input (if bit cleared, = 0).	
		0	Input
		1	Output
2 (R/W)	FLG1	FLAG1 State. The REGF_FLAGS.FLG1 bit indicates the state of the FLAG1 pin as high (if set, = 1) or low (if cleared, = 0).	
1 (R/W)	FLG0O	FLAG0 Output. The REGF_FLAGS.FLG0O bit selects the I/O direction for the FLAG0 pin. The I/O direction of the pin is programmed as an output (if bit set, = 1) or input (if bit cleared, = 0).	
		0	Input
		1	Output
0 (R/W)	FLG0	FLAG0 State. The REGF_FLAGS.FLG0 bit indicates the state of the FLAG0 pin as high (if set, = 1) or low (if cleared, = 0).	

Interrupt Mask Register

Each bit in the REGF_IMASK register corresponds to a bit with the same name in the REGF_IRPTL register. The bits in REGF_IMASK unmask (enable if set, =1), or mask (disable if cleared, = 0) the interrupts that are latched in the REGF_IRPTL register. Except for the RSTI and EMUI bits, all interrupts are maskable.

When the REGF_IMASK register masks an interrupt, the masking disables the processor's response to the interrupt. The IRPTL register still latches an interrupt even when masked, and the processor responds to that latched interrupt if it is later unmasked.

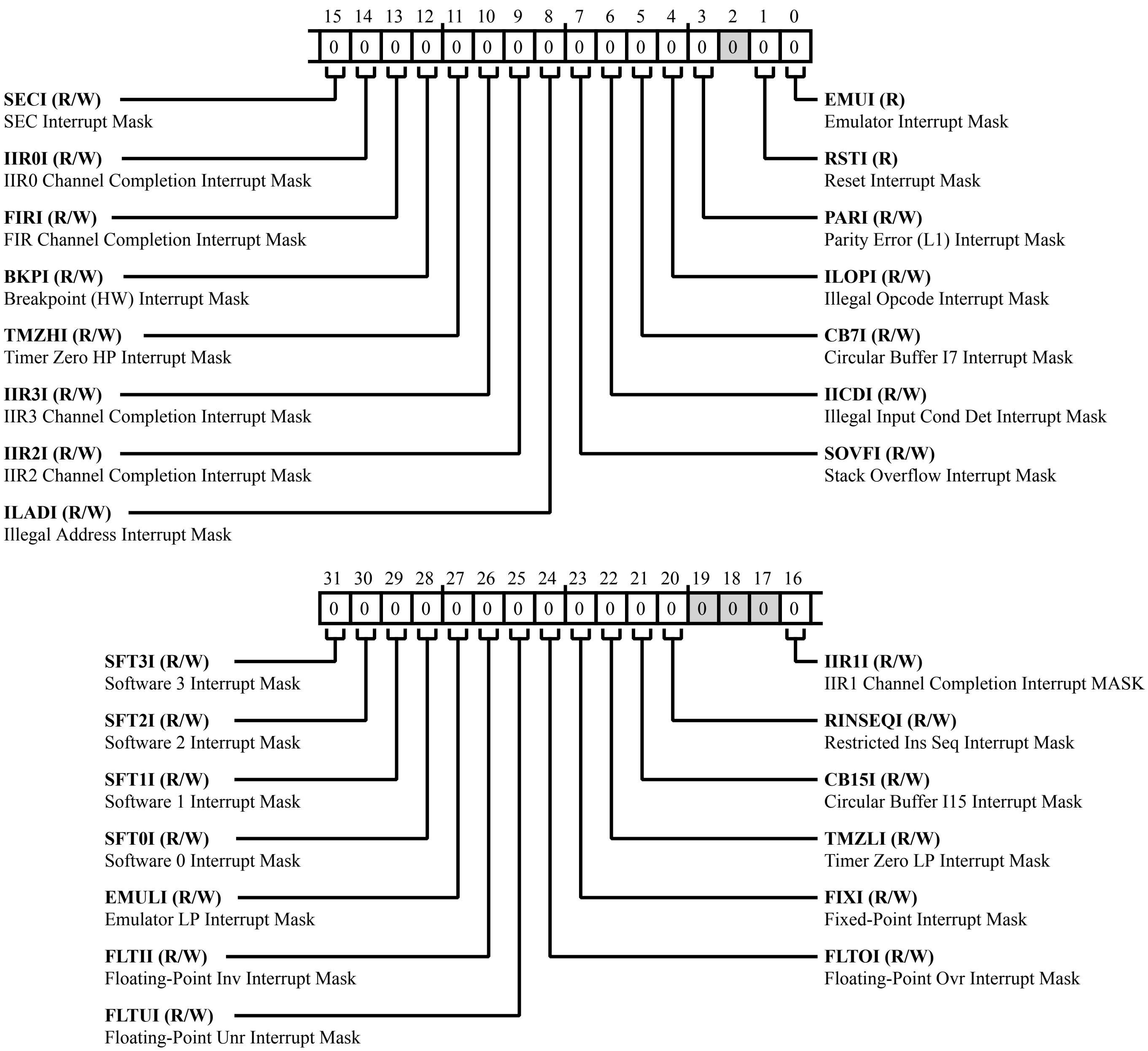

Figure 29-10: REGF_IMASK Register Diagram

Table 29-11: REGF_IMASK Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31 (R/W)	SFT3I	Software 3 Interrupt Mask. The REGF_IMASK.SFT3I bit masks (if cleared, = 0) or unmasks (if set, = 1) the software (user) 3 interrupt (SFT3I).
30 (R/W)	SFT2I	Software 2 Interrupt Mask. The REGF_IMASK.SFT2I bit masks (if cleared, = 0) or unmasks (if set, = 1) the software (user) 2 interrupt (SFT2I).
29 (R/W)	SFT1I	Software 1 Interrupt Mask. The REGF_IMASK.SFT1I bit masks (if cleared, = 0) or unmasks (if set, = 1) the software (user) 1 interrupt (SFT1I).
28 (R/W)	SFT0I	Software 0 Interrupt Mask. The REGF_IMASK.SFT0I bit masks (if cleared, = 0) or unmasks (if set, = 1) the software (user) 0 interrupt (SFT0I).
27 (R/W)	EMULI	Emulator LP Interrupt Mask. The REGF_IMASK.EMULI bit masks (if cleared, = 0) or unmasks (if set, = 1) the emulator low-priority interrupt (EMULI).
26 (R/W)	FLTII	Floating-Point Inv Interrupt Mask. The REGF_IMASK.FLTII bit masks (if cleared, = 0) or unmasks (if set, = 1) the floating-point invalid operation interrupt (FLTII).
25 (R/W)	FLTUI	Floating-Point Unr Interrupt Mask. The REGF_IMASK.FLTUI bit masks (if cleared, = 0) or unmasks (if set, = 1) the floating-point underflow interrupt (FLTUI).
24 (R/W)	FLTOI	Floating-Point Ovr Interrupt Mask. The REGF_IMASK.FLTOI bit masks (if cleared, = 0) or unmasks (if set, = 1) the floating-point overflow interrupt (FLTOI).
23 (R/W)	FIXI	Fixed-Point Interrupt Mask. The REGF_IMASK.FIXI bit masks (if cleared, = 0) or unmasks (if set, = 1) the fixed-point overflow interrupt (FIXI).
22 (R/W)	TMZLI	Timer Zero LP Interrupt Mask. The REGF_IMASK.TMZLI bit masks (if cleared, = 0) or unmasks (if set, = 1) the core timer zero (expired) low priority interrupt (TMZLI). A TMZLI occurs when the timer decrements to zero. Note that this event also triggers a TMZHI. Because the timer expired event (TCOUNT decrements to zero) generates two interrupts (TMZHI and TMZLI), programs should unmask the timer interrupt with the desired priority and leave the other one masked.

Table 29-11: REGF_IMASK Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
21 (R/W)	CB15I	Circular Buffer I15 Interrupt Mask. The `REGF_IMASK.CB15I` bit masks (if cleared, = 0) or unmasks (if set, = 1) the circular buffer overflow interrupt (CB15I).
20 (R/W)	RINSEQI	Restricted Ins Seq Interrupt Mask. The `REGF_IMASK.RINSEQI` bit masks (if cleared, = 0) or unmasks (if set, = 1) the restricted instruction sequence interrupt (RINSEQI).
16 (R/W)	IIR1I	IIR1 Channel Completion Interrupt MASK. The `REGF_IMASK.IIR1I` bit masks (if cleared, = 0) or unmasks (if set, = 1) the second IIR channel completion interrupt (IIR1I). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.
15 (R/W)	SECI	SEC Interrupt Mask. The `REGF_IMASK.SECI` bit masks (if cleared, = 0) or unmasks (if set, = 1) the system event controller (SEC) interrupt (SECI). This masking functionality is augmented with the operation of the `REGF_MODE1.NESTM` bit and `REGF_MODE2.SNEN` bit. If these bits are set (enabling nest multiple interrupts and enabling SEC self nesting), a new SECI interrupt may latch in `REGF_IRPTL.SECI` while an SECI interrupt is being serviced. SECI is not masked but lower priority interrupts are. If a higher priority interrupt interrupts SECI, SECI becomes masked.
14 (R/W)	IIR0I	IIR0 Channel Completion Interrupt Mask. The `REGF_IMASK.IIR0I` bit masks (if cleared, = 0) or unmasks (if set, = 1) the first IIR channel completion interrupt (IIR0I). This bit is available in ADSP-2156x and the ADSP-SC59x/ADSP-2159x processor. It is called IIRI in the ADSP-2156x processor.
13 (R/W)	FIRI	FIR Channel Completion Interrupt Mask. The `REGF_IMASK.FIRI` bit masks (if cleared, = 0) or unmasks (if set, = 1) the FIR channel completion interrupt (FIRI). This bit is only available in the ADSP-2156x and the ADSP-SC59x/ADSP-2159x processors.
12 (R/W)	BKPI	Breakpoint (HW) Interrupt Mask. The `REGF_IMASK.BKPI` bit masks (if cleared, = 0) or unmasks (if set, = 1) the hardware breakpoint interrupt (BKPI).

Table 29-11: REGF_IMASK Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
11 (R/W)	TMZHI	Timer Zero HP Interrupt Mask. The REGF_IMASK.TMZHI bit masks (if cleared, = 0) or unmasks (if set, = 1) the core timer zero (expired) high priority interrupt (TMZHI). A TMZHI occurs when the timer decrements to zero. Note that this event also triggers a TMZLI. Because the timer expired event (TCOUNT decrements to zero) generates two interrupts (TMZHI and TMZLI), programs should unmask the timer interrupt with the desired priority and leave the other one masked.
10 (R/W)	IIR3I	IIR3 Channel Completion Interrupt Mask. The REGF_IMASK.IIR3I bit masks (if cleared, = 0) or unmasks (if set, = 1) the fourth IIR channel completion interrupt (IIR3I). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.
9 (R/W)	IIR2I	IIR2 Channel Completion Interrupt Mask. The REGF_IMASK.IIR2I bit masks (if cleared, = 0) or unmasks (if set, = 1) the third IIR channel completion interrupt (IIR2I). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.
8 (R/W)	ILADI	Illegal Address Interrupt Mask. The REGF_IMASK.ILADI bit masks (if cleared, = 0) or unmasks (if set, = 1) the illegal address space detected interrupt (ILADI).
7 (R/W)	SOVFI	Stack Overflow Interrupt Mask. The REGF_IMASK.SOVFI bit masks (if cleared, = 0) or unmasks (if set, = 1) the stack overflow interrupt (SOVFI). A SOVFI occurs when a stack in the program sequencer overflows or is full.
6 (R/W)	IICDI	Illegal Input Cond Det Interrupt Mask. The REGF_IMASK.IICDI bit masks (if cleared, = 0) or unmasks (if set, = 1) the illegal input condition detected interrupt (IICDI).
5 (R/W)	CB7I	Circular Buffer I7 Interrupt Mask. The REGF_IMASK.CB7I bit masks (if cleared, = 0) or unmasks (if set, = 1) the circular buffer overflow interrupt (CB7I).
4 (R/W)	ILOPI	Illegal Opcode Interrupt Mask. The REGF_IMASK.ILOPI bit masks (if cleared, = 0) or unmasks (if set, = 1) the illegal opcode detected interrupt (ILOPI).
3 (R/W)	PARI	Parity Error (L1) Interrupt Mask. The REGF_IMASK.PARI bit masks (if cleared, = 0) or unmasks (if set, = 1) the parity error on L1 access interrupt (PARI).

Table 29-11: REGF_IMASK Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
1 (R/NW)	RSTI	Reset Interrupt Mask. The `REGF_IMASK.RSTI` bit is read-only. The reset interrupt (RSTI) is non-maskable.
0 (R/NW)	EMUI	Emulator Interrupt Mask. The `REGF_IMASK.EMUI` bit is read-only. The emulator interrupt (EMUI) is non-maskable.

Interrupt Mask Pointer Register

Each bit in the REGF_IMASKP register corresponds to a bit with the same name in the REGF_IRPTL register. The REGF_IMASKP register supports an interrupt nesting scheme that lets higher priority events interrupt an ISR and keeps lower priority events from interrupting.

When interrupt nesting is enabled, the bits in the REGF_IMASKP register mask interrupts that have a lower priority than the interrupt that is currently being serviced. Other bits in this register unmask interrupts having higher priority than the interrupt that is currently being serviced. Interrupt nesting is enabled using REGF_MODE1.NESTM bit. The REGF_IRPTL register latches a lower priority interrupt even when masked, and the processor responds to that latched interrupt if it is later unmasked.

When interrupt nesting is disabled (REGF_MODE1.NESTM =0), the bits in the REGF_IMASKP register mask all interrupts while an interrupt is currently being serviced. The REGF_IRPTL register still latches these interrupts even when masked, and the processor responds to the highest priority latched interrupt after servicing the current interrupt.

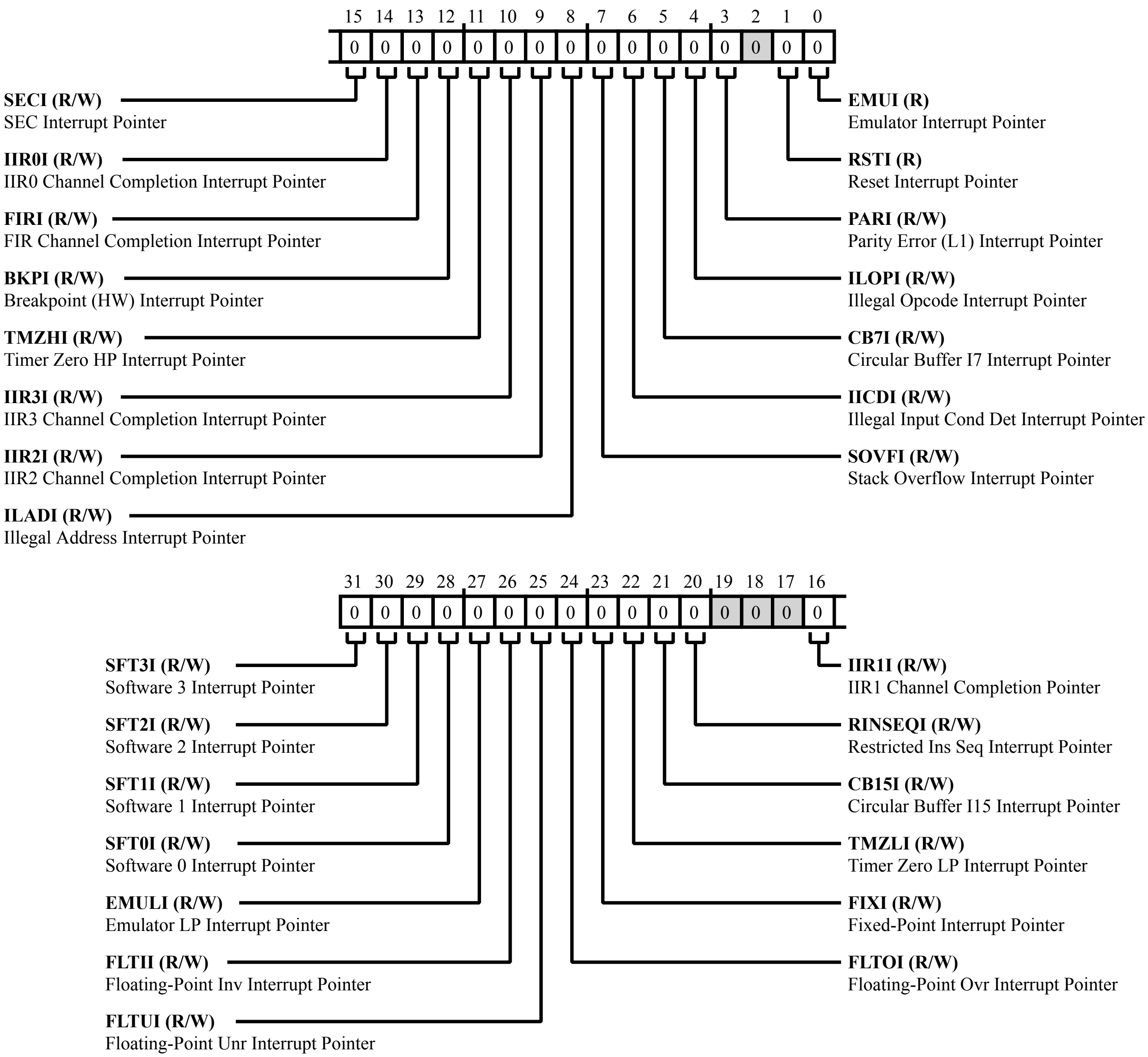

Figure 29-11: REGF_IMASKP Register Diagram

Table 29-12: REGF_IMASKP Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31 (R/W)	SFT3I	Software 3 Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.SFT3I bit masks (if cleared, = 0) or unmasks (if set, = 1) the software (user) 3 interrupt (SFT3I) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description.
30 (R/W)	SFT2I	Software 2 Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.SFT2I bit masks (if cleared, = 0) or unmasks (if set, = 1) the software (user) 2 interrupt (SFT2I) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description.
29 (R/W)	SFT1I	Software 1 Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.SFT1I bit masks (if cleared, = 0) or unmasks (if set, = 1) the software (user) 1 interrupt (SFT1I) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description.
28 (R/W)	SFT0I	Software 0 Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.SFT0I bit masks (if cleared, = 0) or unmasks (if set, = 1) the software (user) 0 interrupt (SFT0I) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description.
27 (R/W)	EMULI	Emulator LP Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.EMULI bit masks (if cleared, = 0) or unmasks (if set, = 1) the emulator low-priority interrupt (EMULI) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description.
26 (R/W)	FLTII	Floating-Point Inv Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.FLTII bit masks (if cleared, = 0) or unmasks (if set, = 1) the floating-point invalid operation interrupt (FLTII) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description.
25 (R/W)	FLTUI	Floating-Point Unr Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.FLTUI bit masks (if cleared, = 0) or unmasks (if set, = 1) the floating-point underflow interrupt (FLTUI) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description.

Table 29-12: REGF_IMASKP Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
24 (R/W)	FLTOI	Floating-Point Ovr Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.FLTOI bit masks (if cleared, = 0) or unmasks (if set, = 1) the floating-point overflow interrupt (FLTOI) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description.
23 (R/W)	FIXI	Fixed-Point Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.FIXI bit masks (if cleared, = 0) or unmasks (if set, = 1) the fixed-point overflow interrupt (FIXI) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description.
22 (R/W)	TMZLI	Timer Zero LP Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.TMZLI bit masks (if cleared, = 0) or unmasks (if set, = 1) the core timer zero (expired) low priority interrupt (TMZLI) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description.
21 (R/W)	CB15I	Circular Buffer I15 Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.CB15I bit masks (if cleared, = 0) or unmasks (if set, = 1) the circular buffer overflow interrupt (CB15I) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description.
20 (R/W)	RINSEQI	Restricted Ins Seq Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.RINSEQI bit masks (if cleared, = 0) or unmasks (if set, = 1) the restricted instruction sequence interrupt (RINSEQI) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description.
16 (R/W)	IIR1I	IIR1 Channel Completion Pointer. When interrupt nesting is enabled, the REGF_IMASKP.IIR1I bit masks (if cleared, = 0) or unmasks (if set, = 1) the second IIR channel completion interrupt (IIR1I) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description. This bit is only available in the ADSP-SC59x/ADSP-2159x processor.
15 (R/W)	SECI	SEC Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.SECI bit masks (if cleared, = 0) or unmasks (if set, = 1) the system event controller (SEC) interrupt (SECI) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description. Note that this interrupt supports additional nesting (self nesting) when enabled with the REGF_MODE2.SNEN bit.

Table 29-12: REGF_IMASKP Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
14 (R/W)	IIR0I	IIR0 Channel Completion Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.IIR0I bit masks (if cleared, = 0) or unmasks (if set, = 1) the first IIR channel completion interrupt (IIR0I) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description. This bit is only available in the ADSP-2156x and the ADSP-SC59x/ADSP-2159x processors. It is called IIRI in the ADSP-2156x processor.
13 (R/W)	FIRI	FIR Channel Completion Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.FIRI bit masks (if cleared, = 0) or unmasks (if set, = 1) the FIR channel completion interrupt (FIRI) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description. This bit is available in the ADSP-2156x and the ADSP-SC59x/ADSP-2159x processors.
12 (R/W)	BKPI	Breakpoint (HW) Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.BKPI bit masks (if cleared, = 0) or unmasks (if set, = 1) the hardware breakpoint interrupt (BKPI) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description.
11 (R/W)	TMZHI	Timer Zero HP Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.TMZHI bit masks (if cleared, = 0) or unmasks (if set, = 1) the core timer zero (expired) high priority interrupt (TMZHI) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description.
10 (R/W)	IIR3I	IIR3 Channel Completion Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.IIR3I bit masks (if cleared, = 0) or unmasks (if set, = 1) the fourth IIR channel completion interrupt (IIR3I) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description. This bit is only available in the ADSP-SC59x/ADSP-2159x processor.
9 (R/W)	IIR2I	IIR2 Channel Completion Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.IIR2I bit masks (if cleared, = 0) or unmasks (if set, = 1) the third IIR channel completion interrupt (IIR2I) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description. This bit is only available in the ADSP-SC59x/ADSP-2159x processor.
8 (R/W)	ILADI	Illegal Address Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.ILADI bit masks (if cleared, = 0) or unmasks (if set, = 1) the illegal address space detected interrupt (ILADI) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description.

Table 29-12: REGF_IMASKP Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
7 (R/W)	SOVFI	Stack Overflow Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.SOVFI bit masks (if cleared, = 0) or unmasks (if set, = 1) the stack overflow or full interrupt (SOVFI) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description.
6 (R/W)	IICDI	Illegal Input Cond Det Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.IICDI bit masks (if cleared, = 0) or unmasks (if set, = 1) the illegal input condition detected interrupt (IICDI) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description.
5 (R/W)	CB7I	Circular Buffer I7 Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.CB7I bit masks (if cleared, = 0) or unmasks (if set, = 1) the circular buffer overflow interrupt (CB7I) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description.
4 (R/W)	ILOPI	Illegal Opcode Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.ILOPI bit masks (if cleared, = 0) or unmasks (if set, = 1) the illegal opcode detected interrupt (ILOPI) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description.
3 (R/W)	PARI	Parity Error (L1) Interrupt Pointer. When interrupt nesting is enabled, the REGF_IMASKP.PARI bit masks (if cleared, = 0) or unmasks (if set, = 1) the parity error on L1 access interrupt (PARI) while the processor is servicing a higher priority interrupt. For more information, see the REGF_IMASKP register description.
1 (R/NW)	RSTI	Reset Interrupt Pointer. The REGF_IMASKP.RSTI bit is read-only. The reset interrupt (RSTI) is non-maskable.
0 (R/NW)	EMUI	Emulator Interrupt Pointer. The REGF_IMASKP.EMUI bit is read-only. The emulator interrupt (EMUI) is non-maskable.

Interrupt Latch Register

The REGF_IRPTL register indicates latch status for core interrupts. The system event controller (SEC) of the processor manages the configuration of all system event sources (such as interrupts). The SEC also manages the propagation of system events to all connected SHARC cores and the system fault interface. For more information about interrupt control, see the processor hardware reference.

The REGF_IRPTL register provides a number of software interrupts. When a program sets the latch bit for one of these interrupts, the sequencer services the interrupt, and the processor branches to the corresponding interrupt routine. Software interrupts have the same behavior as all other maskable interrupts. For more information about interrupt sequencing, see the Variations in Program Flow section of the Program Sequencer chapter.

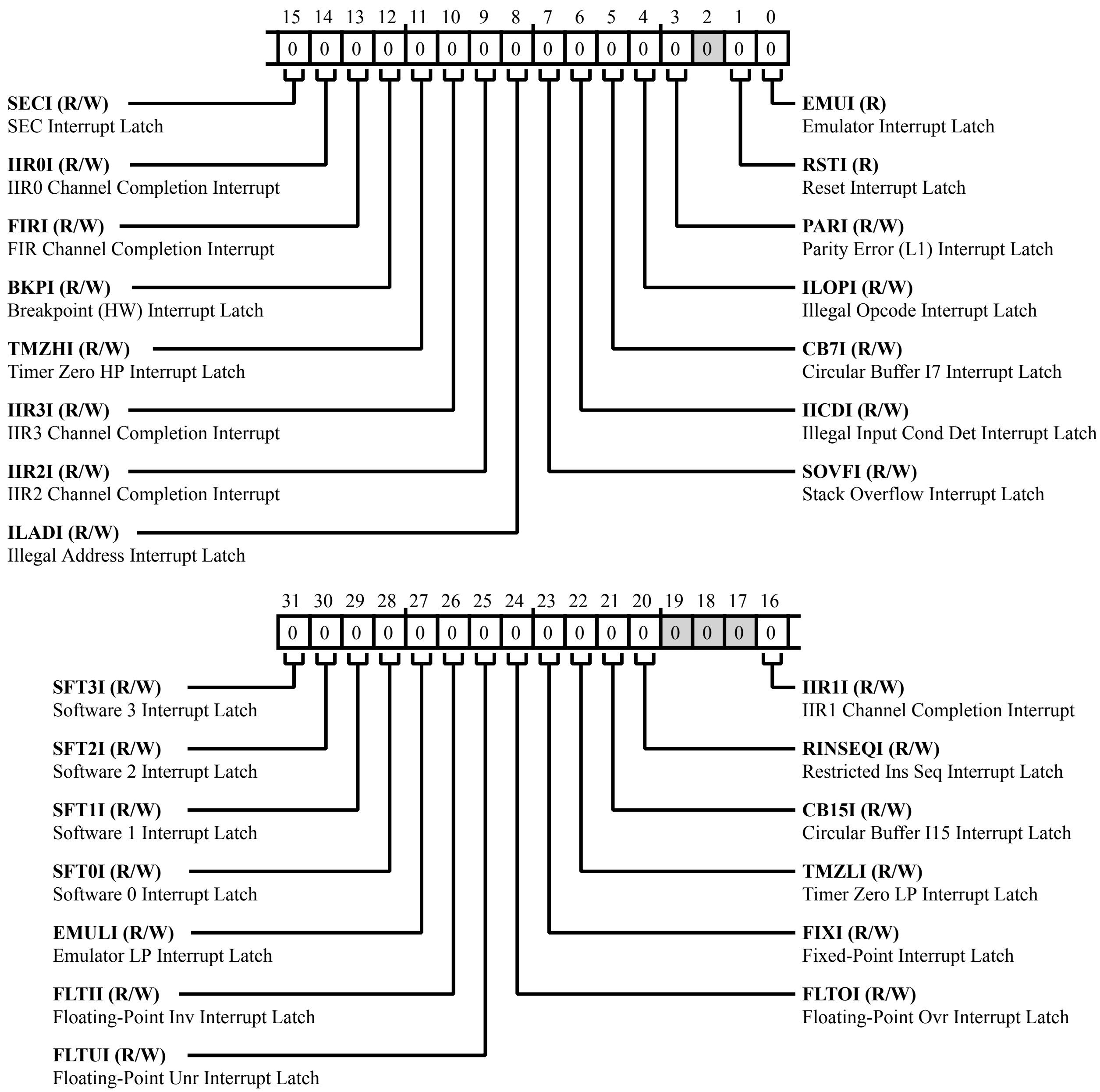

Figure 29-12: REGF_IRPTL Register Diagram

Table 29-13: REGF_IRPTL Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31 (R/W)	SFT3I	Software 3 Interrupt Latch. The `REGF_IRPTL.SFT3I` bit indicates whether the processor detected (latched) a software (user) 3 interrupt (SFT3I). An SFT3I occurs when a program sets (= 1) this bit.

Table 29-13: REGF_IRPTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
30 (R/W)	SFT2I	Software 2 Interrupt Latch. The REGF_IRPTL.SFT2I bit indicates whether the processor detected (latched) a software (user) 2 interrupt (SFT2I). An SFT2I occurs when a program sets (= 1) this bit.
29 (R/W)	SFT1I	Software 1 Interrupt Latch. The REGF_IRPTL.SFT1I bit indicates whether the processor detected (latched) a software (user) 1 interrupt (SFT1I). An SFT1I occurs when a program sets (= 1) this bit.
28 (R/W)	SFT0I	Software 0 Interrupt Latch. The REGF_IRPTL.SFT0I bit indicates whether the processor detected (latched) a software (user) 0 interrupt (SFT0I). An SFT0I occurs when a program sets (= 1) this bit.
27 (R/W)	EMULI	Emulator LP Interrupt Latch. The REGF_IRPTL.EMULI bit indicates whether the processor detected (latched) an emulator low-priority interrupt (EMULI). An EMULI occurs during background telemetry channel (BTC) operations. This interrupt has a lower priority than EMUI, but higher priority than software interrupts.
26 (R/W)	FLTII	Floating-Point Inv Interrupt Latch. The REGF_IRPTL.FLTII bit indicates whether the processor detected (latched) a floating-point operation invalid interrupt (FLTII). For more information about floating-point invalid operations, see the descriptions of the status registers: REGF_ASTATX and REGF_ASTATY.
25 (R/W)	FLTUI	Floating-Point Unr Interrupt Latch. The REGF_IRPTL.FLTUI bit indicates whether the processor detected (latched) a floating-point underflow interrupt (FLTUI). For more information about floating-point underflow, see the descriptions of the status registers: REGF_ASTATX and REGF_ASTATY.
24 (R/W)	FLTOI	Floating-Point Ovr Interrupt Latch. The REGF_IRPTL.FLTOI bit indicates whether the processor detected (latched) a floating-point overflow interrupt (FLTOI). For more information about floating-point overflow, see the descriptions of the status registers: REGF_ASTATX and REGF_ASTATY.
23 (R/W)	FIXI	Fixed-Point Interrupt Latch. The REGF_IRPTL.FIXI bit indicates whether the processor detected (latched) a fixed-point overflow interrupt (FIXI). For more information about fixed-point overflow, see the descriptions of the status registers: REGF_ASTATX and REGF_ASTATY.

Table 29-13: REGF_IRPTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
22 (R/W)	TMZLI	Timer Zero LP Interrupt Latch. The REGF_IRPTL.TMZLI bit indicates whether the processor detected (latched) a core timer zero (expired) low priority interrupt (TMZLI). A TMZLI occurs when the timer decrements to zero. Note that this event also triggers a TMZHI. Because the timer expired event (TCOUNT decrements to zero) generates two interrupts, TMZHI and TMZLI, programs should unmask the timer interrupt with the desired priority and leave the other one masked.
21 (R/W)	CB15I	Circular Buffer I15 Interrupt Latch. The REGF_IRPTL.CB15I bit indicates whether the processor detected (latched) a circular buffer overflow interrupt for a circular buffer indexed with data address generator (DAG) 2 index 15 (I15). A circular buffer overflow occurs when the DAG circular buffering operation increments the index register past the end of the buffer.
20 (R/W)	RINSEQI	Restricted Ins Seq Interrupt Latch. The REGF_IRPTL.RINSEQI bit indicates whether the processor detected (latched) a restricted instruction sequence interrupt (RINSEQI). A RINSEQI occurs when the processor executes as follows: • In the case of nested loop where the inner loop is an e2-active Counter-based Loop, the outer loop is an Arithmetic Loop, and the last instruction of both the loops are separated by one instruction. The L-2 of inner loop cannot have any branches. The second last instruction of the inner loop should not have a branch instruction. • If the Last 5 instructions of an Arithmetic-loop have a delayed branch instruction.
16 (R/W)	IIR1I	IIR1 Channel Completion Interrupt. The REGF_IRPTL.IIR1I bit indicates whether the processor detected (latched) the second IIR channel completion interrupt (IIR1I). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.
15 (R/W)	SECI	SEC Interrupt Latch. The REGF_IRPTL.SECI bit indicates whether the processor detected (latched) an system event controller (SEC) interrupt (SECI). When SECI interrupt self nesting is enabled with the REGF_MODE2.SNEN bit, an SECI can latch even when the interrupt is currently being serviced (REGF_IMASKP.SECI bit is set). For a list of SEC interrupts, see the SEC chapter in the processor hardware reference.
14 (R/W)	IIR0I	IIR0 Channel Completion Interrupt. The REGF_IRPTL.IIR0I bit indicates whether the processor detected (latched) the first IIR channel completion interrupt (IIR0I). This bit is available only in the ADSP-2156x and ADSP-SC59x/ADSP-2159x processors. It is called IIRI in the ADSP-2156x processor.

Table 29-13: REGF_IRPTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
13 (R/W)	FIRI	FIR Channel Completion Interrupt. The `REGF_IRPTL.FIRI` bit indicates whether the processor detected (latched) FIR channel completion interrupt (FIRI). This bit is available in the ADSP-2156x and ADSP-SC59x/ADSP-2159x processors.
12 (R/W)	BKPI	Breakpoint (HW) Interrupt Latch. The `REGF_IRPTL.BKPI` bit indicates whether the processor detected (latched) a hardware breakpoint interrupt (BKPI).
11 (R/W)	TMZHI	Timer Zero HP Interrupt Latch. The `REGF_IRPTL.TMZHI` bit indicates whether the processor detected (latched) a core timer zero (expired) high priority interrupt (TMZHI). A TMZHI occurs when the timer decrements to zero. Note that this event also triggers a TMZLI. Because the timer expired event (TCOUNT decrements to zero) generates two interrupts, TMZHI and TMZLI, programs should unmask the timer interrupt with the desired priority and leave the other one masked.
10 (R/W)	IIR3I	IIR3 Channel Completion Interrupt. The `REGF_IRPTL.IIR3I` bit indicates whether the processor detected (latched) the fourth IIR channel completion interrupt (IIR3I). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.
9 (R/W)	IIR2I	IIR2 Channel Completion Interrupt. The `REGF_IRPTL.IIR2I` bit indicates whether the processor detected (latched) the third IIR channel completion interrupt (IIR2I). This bit is only available in the ADSP-SC59x/ADSP-2159x processor.
8 (R/W)	ILADI	Illegal Address Interrupt Latch. The `REGF_IRPTL.ILADI` bit indicates whether the processor detected (latched) an illegal address space detected interrupt (ILADI). An ILADI occurs when the processor makes a byte or short word access to any space other than byte address space. An ILADI also occurs when the processor executes a modify instruction with the (sw) flag using an index register in non-byte address space or with the (nw) flag using an index register in any long word or short word address space. For more information about illegal address conditions, see the descriptions of the sticky status registers: `REGF_STKYX` and `REGF_STKYX`.
7 (R/W)	SOVFI	Stack Overflow Interrupt Latch. The `REGF_IRPTL.SOVFI` bit indicates whether the processor detected (latched) an stack overflow or full interrupt (SOVFI). A SOVFI occurs when a stack in the program sequencer overflows or is full.

Table 29-13: REGF_IRPTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
6 (R/W)	IICDI	Illegal Input Cond Det Interrupt Latch. The REGF_IRPTL.IICDI bit indicates whether the processor detected (latched) an illegal input condition interrupt (IICDI). An IICDI occurs when a TRUE results from the logical OR'ing of the illegal I/O processor register access status bit (IIRA) bit and unaligned 64-bit memory access status bit (U64MA). For more information about illegal input conditions, see the descriptions of the sticky status registers: REGF_STKYX and REGF_STKYX.
5 (R/W)	CB7I	Circular Buffer I7 Interrupt Latch. The REGF_IRPTL.CB7I bit indicates whether the processor detected (latched) a circular buffer overflow interrupt for a circular buffer indexed with data address generator (DAG) 1 index 7 (I7). A circular buffer overflow occurs when the DAG circular buffering operation increments the index register past the end of the buffer.
4 (R/W)	ILOPI	Illegal Opcode Interrupt Latch. The REGF_IRPTL.ILOPI bit indicates whether the processor detected (latched) an illegal opcode detected interrupt (ILOPI). An ILOPI occurs when the processor performs an instruction fetch and encounters an instruction that does not match with existing opcodes. For more information about illegal opcode conditions, see the descriptions of the sticky status registers: REGF_STKYX and REGF_STKYX.
3 (R/W)	PARI	Parity Error (L1) Interrupt Latch. The REGF_IRPTL.PARI bit indicates whether the processor detected (latched) a parity error on L1 access interrupt (PARI). A PARI occurs when the processor performs an L1 memory access with parity check enabled and detects a parity error. Parity checking is enabled with the IPERREN, DPERREN, and SPERREN bits in the REGF_MODE1 register and (when enabled) checking occurs on writes to L1 and reads of L1. The PARI for core accesses is generated only for valid accesses. Accesses that are aborted do not generate a parity error even if an error is detected. For more information about parity check conditions, see the descriptions of the IPERREN, DPERREN, and SPERREN bits in the REGF_MODE1 register.
1 (R/NW)	RSTI	Reset Interrupt Latch. The REGF_IRPTL.RSTI bit indicates whether the processor detected (latched) a reset interrupt (RSTI). An RSTI occurs as an external device asserts the SYS_HWRST pin or after a software reset through the reset control unit (RCU). Note that this bit is read-only and the RSTI interrupt is non-maskable.
0 (R/NW)	EMUI	Emulator Interrupt Latch. The REGF_IRPTL.EMUI bit indicates whether the processor detected (latched) an emulator interrupt (EMUI). An EMUI occurs when the external emulator triggers an interrupt or the core hits a emulator breakpoint. Note that this interrupt has highest priority, is read-only, and is non-maskable.

Index Registers

The data address generators (DAGs) store addresses in index (REGF_I[n]) registers. Registers I0 through I7 for are for DAG1, and registers I8 through I15 are for DAG2. An index register holds an address and acts as a pointer to a memory location.

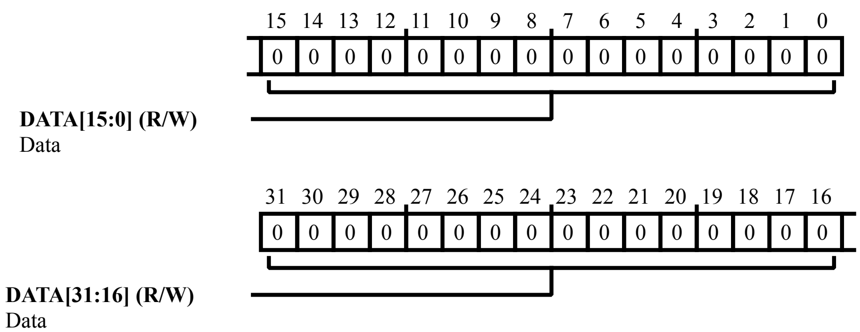

Figure 29-13: REGF_I[n] Register Diagram

Table 29-14: REGF_I[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_I[n].DATA bit field contains index address data.

Loop Address Stack Register

The REGF_LADDR register contains the top entry in the loop address stack. The loop address stack described is six levels deep by 32 bits wide. The 32-bit word of each level consists of a 24-bit loop termination address, a 5-bit termination code, and a 3-bit loop type code.

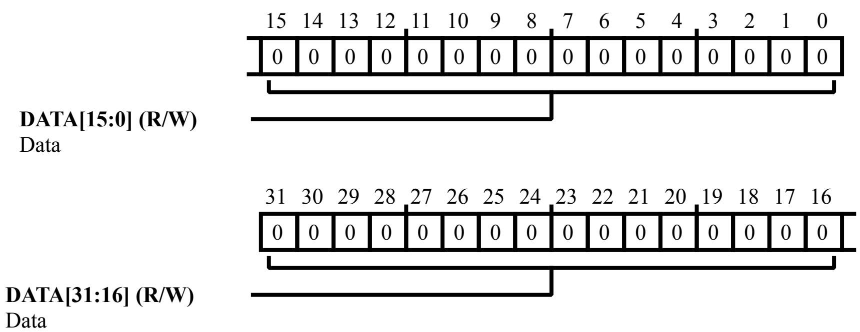

Figure 29-14: REGF_LADDR Register Diagram

Table 29-15: REGF_LADDR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_LADDR.DATA bit field contains address data.

Loop Counter Register

The loop counter (REGF_LCNTR) register provides access to the loop counter stack and holds the count value before the DO UNTIL termination loop is executed. For more information about using the REGF_LCNTR register, see the Loop Counter Stack Access section.

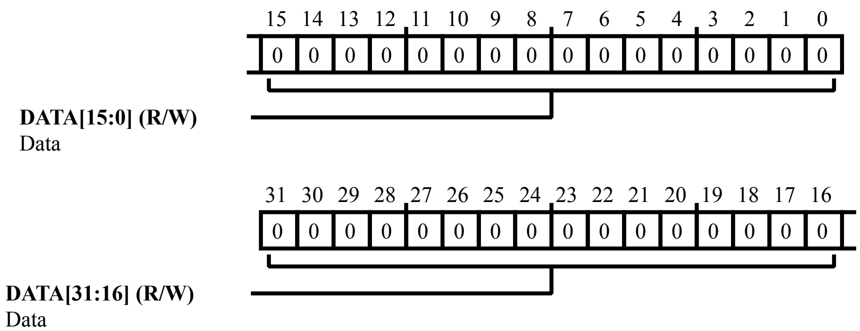

Figure 29-15: REGF_LCNTR Register Diagram

Table 29-16: REGF_LCNTR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_LCNTR.DATA bit field contains data.

Length (Circular Buffer) Registers

The data address generators (DAGs) control circular buffering operations with length (REGF_L[n]) registers and base (REGF_B[n]) registers. Registers L0 through L7 and B0 through B7 are for DAG1, and registers L8 through L15 and B8 through B15 are for DAG2. Length and base registers set up the range of addresses and the starting address for a circular buffer.

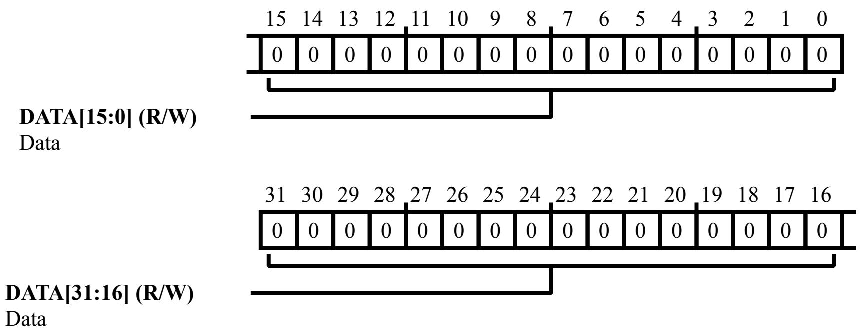

Figure 29-16: REGF_L[n] Register Diagram

Table 29-17: REGF_L[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_L[n].DATA bit field contains circular buffer length data.

Mode Mask Register

Bits that are set in the REGF_MMASK register are used to clear bits in the REGF_MODE1 register when the processor's status stack is pushed. This effectively disables different modes when servicing an interrupt, or when executing a push sts instruction. The processor's status stack is pushed:

- When executing a push sts instruction explicitly in code
- When any interrupt occurs

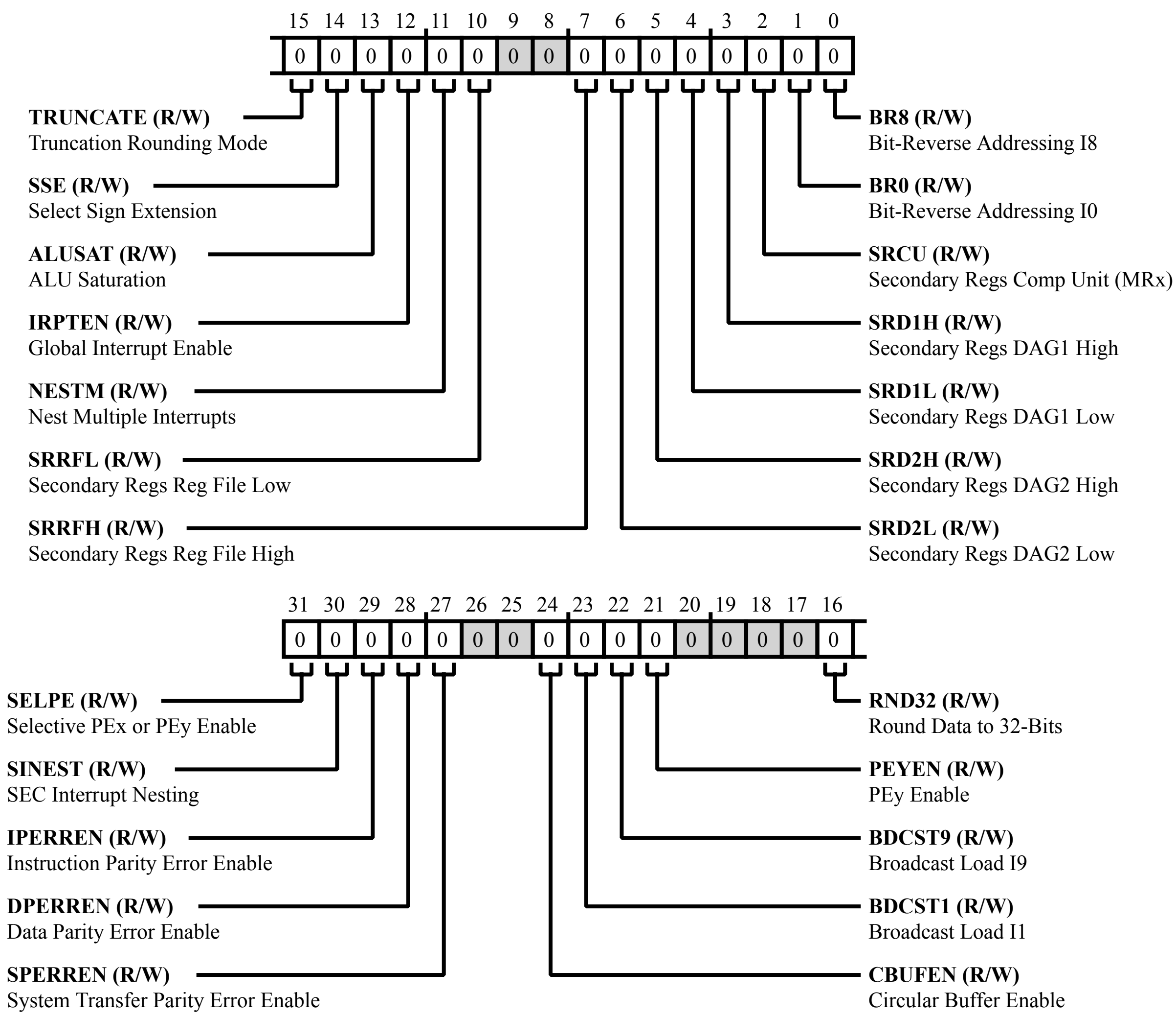

Figure 29-17: REGF_MMASK Register Diagram

Table 29-18: REGF_MMASK Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31 (R/W)	SELPE	Selective PEx or PEy Enable. Setting the REGF_MMASK.SELPE bit clears the REGF_MODE1.SELPE bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.
30 (R/W)	SINEST	SEC Interrupt Nesting. Setting the REGF_MMASK.SINEST bit clears the REGF_MODE1.SINEST bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.
29 (R/W)	IPERREN	Instruction Parity Error Enable. Setting the REGF_MMASK.IPERREN bit clears the REGF_MODE1.IPERREN bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.
28 (R/W)	DPERREN	Data Parity Error Enable. Setting the REGF_MMASK.DPERREN bit clears the REGF_MODE1.DPERREN bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.
27 (R/W)	SPERREN	System Transfer Parity Error Enable. Setting the REGF_MMASK.SPERREN bit clears the REGF_MODE1.SPERREN bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.
24 (R/W)	CBUFEN	Circular Buffer Enable. Setting the REGF_MMASK.CBUFEN bit clears the REGF_MODE1.CBUFEN bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.
23 (R/W)	BDCST1	Broadcast Load I1. Setting the REGF_MMASK.BDCST1 bit clears the REGF_MODE1.BDCST1 bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.
22 (R/W)	BDCST9	Broadcast Load I9. Setting the REGF_MMASK.BDCST9 bit clears the REGF_MODE1.BDCST9 bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.
21 (R/W)	PEYEN	PEy Enable. Setting the REGF_MMASK.PEYEN bit clears the REGF_MODE1.PEYEN bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.
16 (R/W)	RND32	Round Data to 32-Bits. Setting the REGF_MMASK.RND32 bit clears the REGF_MODE1.RND32 bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.
15 (R/W)	TRUNCATE	Truncation Rounding Mode. Setting the REGF_MMASK.TRUNCATE bit clears the REGF_MODE1.TRUNCATE bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.

Table 29-18: REGF_MMASK Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
14 (R/W)	SSE	Select Sign Extension. Setting the REGF_MMASK.SSE bit clears the REGF_MODE1.SSE bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.
13 (R/W)	ALUSAT	ALU Saturation. Setting the REGF_MMASK.ALUSAT bit clears the REGF_MODE1.ALUSAT bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.
12 (R/W)	IRPTEN	Global Interrupt Enable. Setting the REGF_MMASK.IRPTEN bit clears the REGF_MODE1.IRPTEN bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.
11 (R/W)	NESTM	Nest Multiple Interrupts. Setting the REGF_MMASK.NESTM bit clears the REGF_MODE1.NESTM bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.
10 (R/W)	SRRFL	Secondary Regs Reg File Low. Setting the REGF_MMASK.SRRFL bit clears the REGF_MODE1.SRRFL bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.
7 (R/W)	SRRFH	Secondary Regs Reg File High. Setting the REGF_MMASK.SRRFH bit clears the REGF_MODE1.SRRFH bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.
6 (R/W)	SRD2L	Secondary Regs DAG2 Low. Setting the REGF_MMASK.SRD2L bit clears the REGF_MODE1.SRD2L bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.
5 (R/W)	SRD2H	Secondary Regs DAG2 High. Setting the REGF_MMASK.SRD2H bit clears the REGF_MODE1.SRD2H bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.
4 (R/W)	SRD1L	Secondary Regs DAG1 Low. Setting the REGF_MMASK.SRD1L bit clears the REGF_MODE1.SRD1L bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.
3 (R/W)	SRD1H	Secondary Regs DAG1 High. Setting the REGF_MMASK.SRD1H bit clears the REGF_MODE1.SRD1H bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.
2 (R/W)	SRCU	Secondary Regs Comp Unit (MRx). Setting the REGF_MMASK.SRCU bit clears the REGF_MODE1.SRCU bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.

Table 29-18: REGF_MMASK Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
1 (R/W)	BR0	Bit-Reverse Addressing I0. Setting the REGF_MMASK.BR0 bit clears the REGF_MODE1.BR0 bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.
0 (R/W)	BR8	Bit-Reverse Addressing I8. Setting the REGF_MMASK.BR8 bit clears the REGF_MODE1.BR8 bit when the status stack is pushed. All other bits in REGF_MODE1 are unaffected.

Mode Control 1 Register

The REGF_MODE1 register controls operating modes for the computation units and other processor core resources, see the bit descriptions for detailed information. The contents of REGF_MODE1, REGF_ASTATX, and REGF_ASTATY may be manually pushed onto the status stack or popped off of the status stack. The REGF_MODE1STK register provides access to the most recently pushed REGF_MODE1 value. The REGF_MMASK register permits selecting REGF_MODE1 bits are cleared when the processor status stack is pushed. This effectively disables different modes when servicing an interrupt, or when executing a `push sts` instruction. For more information, see operating modes description in the Processing Elements chapter and see the functional description in the Program Sequencer chapter.

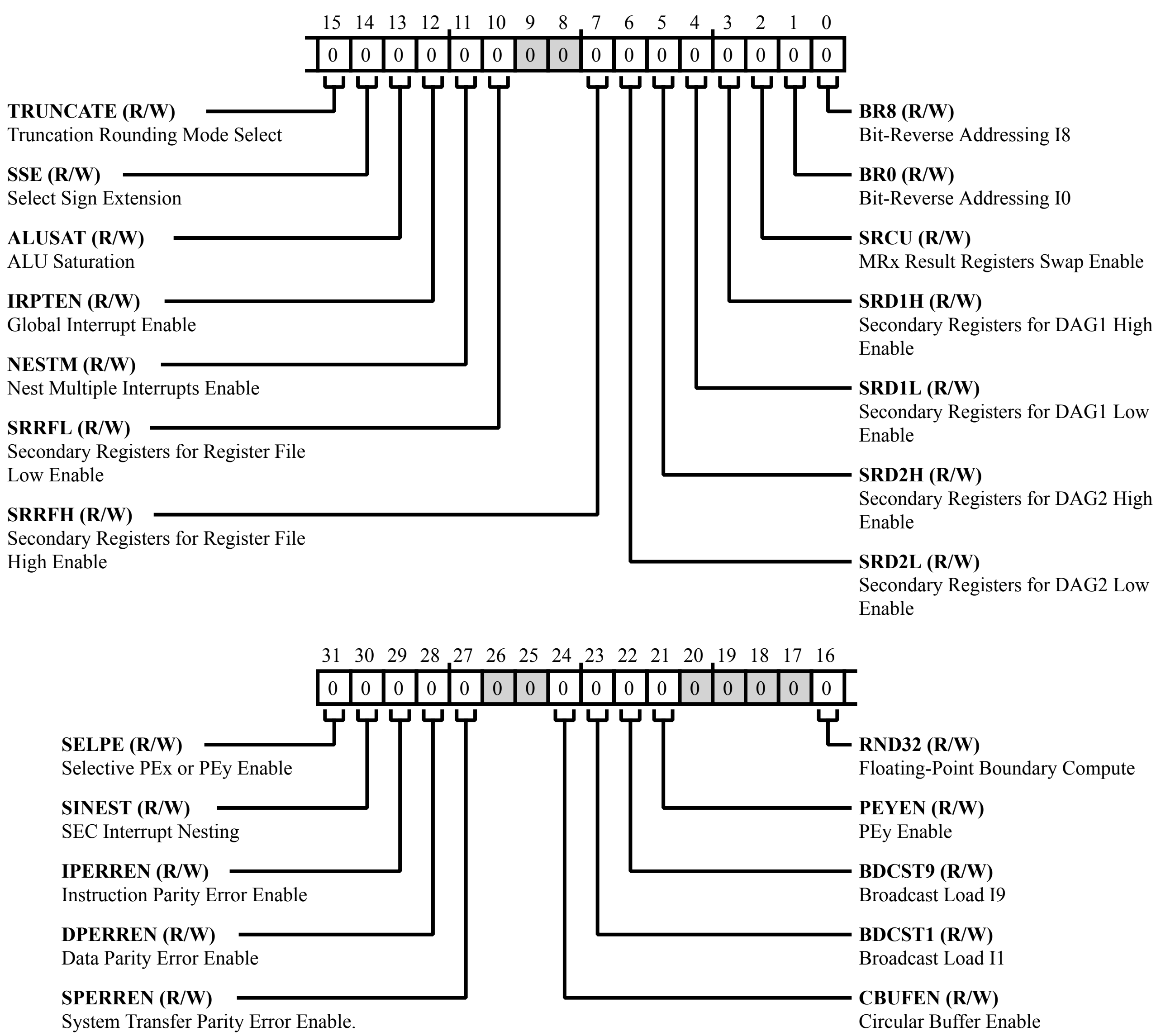

Figure 29-18: REGF_MODE1 Register Diagram

Table 29-19: REGF_MODE1 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31 (R/W)	SELPE	Selective PEx or PEy Enable. The REGF_MODE1.SELPE bit enables (if set, =1) or disables (if cleared, =0) selective, conditional execution in processing elements (PEx and PEy). If a condition evaluates as true in one processing element and false in the other (for example, true in PEx and false in PEy, or false in PEx and true in PEy), a special condition for selective execution in SIMD mode is available. To use these conditions, processor must be in SIMD mode (REGF_MODE1.PEYEN bit is set) with the REGF_MODE1.SELPE bit is set. If the REGF_MODE1.SELPE bit is not set, these conditions behave as FLAG2_IN (true PEx, false PEy) and NOT FLAG2_IN (false PEx, true PEy) both in SIMD and SISD mode. In SISD mode, if the MODE1.SELPE bit is set, these conditions behave as TRUE (true PEx, false PEy) and FALSE (false PEx, true PEy). For more information about selective execution, see the Conditional Execution on One PE in SIMD Mode section of the programming reference.
30 (R/W)	SINEST	SEC Interrupt Nesting. The REGF_MODE1.SINEST bit selects whether the SEC clears the REGF_IMASKP.SECI bit and uses implicitly masking in nest-multiple interrupts mode (REGF_MODE1.NESTM is set). The implicit masking occurs when REGF_MODE1.SINEST is set and the REGF_MODE2.SNEN bit is set. In this mode, the following occur: • On vectoring to the SECI ISR, after automatically pushing the previous value of the MODE1 resister, the NESTED bit in MODE1 is automatically set. • On executing RTI, when the current interrupt is SECI and NESTED is set in the REGF_MODE1STK register, the IMASKP register and interrupt mask are not changed. Otherwise, IMASKP and the masked interrupts are modified as normal. After the REGF_MODE1STK register is tested the RTI instruction pops the mode stack as normal.
29 (R/W)	IPERREN	Instruction Parity Error Enable. The REGF_MODE1.IPERREN bit enables (if set, = 1) or disables (if cleared, = 0) instruction parity error detection for L1 instruction accesses. When this bit is set, the L1 memory interface generate a PARI interrupt when it detects a parity error during an L1 instruction read by the processor core.
28 (R/W)	DPERREN	Data Parity Error Enable. The REGF_MODE1.DPERREN bit enables (if set, = 1) or disables (if cleared, = 0) data parity error detection for L1 data memory accesses. When this bit is set, the L1 memory interface generate a PARI interrupt when it detects a parity error during an L1 data memory read by the processor core.

Table 29-19: REGF_MODE1 Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
27 (R/W)	SPERREN	System Transfer Parity Error Enable.. The REGF_MODE1.SPERREN bit enables (if set, = 1) or disables (if cleared, = 0) system parity error detection for DMA access. When this bit is set, the L1 memory interface generate a PARI interrupt when it detects a parity error during an L1 system transfer (over the S1 or S2 port) by the processor core.
24 (R/W)	CBUFEN	Circular Buffer Enable. The REGF_MODE1.CBUFEN bit enables (circular if set, = 1) or disables (linear if cleared, = 0) circular buffer addressing for buffers with loaded I, M, B, and L DAG registers.
23 (R/W)	BDCST1	Broadcast Load I1. The REGF_MODE1.BDCST1 bit enables (if set, = 1) or disables (if cleared, = 0) broadcast register loads for loads that use the data address generator (DAG) index 1 (I1) register. When this bit is set, data register loads from the DM data bus that use the I1 register are "broadcast" to a register or register pair in each processing element.
22 (R/W)	BDCST9	Broadcast Load I9. The REGF_MODE1.BDCST9 bit enables (if set, = 1) or disables (if cleared, = 0) broadcast register loads for loads that use the data address generator (DAG) index 9 (I9) register. When this bit is set, data register loads from the DM data bus that use the I9 register are "broadcast" to a register or register pair in each processing element.
21 (R/W)	PEYEN	PEy Enable. The REGF_MODE1.PEYEN bit enables PEy (SIMD mode, if =1) or disables PEy (SISD mode, if =0). When this bit is set, the processing element Y (PEy) accepts instruction dispatches. When cleared, PEy goes into a low power mode. If SIMD mode is disabled, programs can load data to the secondary registers (for example, `s0=dm(i0,m0);`), but the PEy computations do not execute.
16 (R/W)	RND32	Floating-Point Boundary Compute. The REGF_MODE1.RND32 bit selects whether the computational units round floating-point data to 32 bits (if =1) or round data to 40 bits (if =0).
15 (R/W)	TRUNCATE	Truncation Rounding Mode Select. The REGF_MODE1.TRUNCATE bit selects whether the ALU or multiplier units round results with round-to-zero (if 1) or round-to-nearest (if 0).
14 (R/W)	SSE	Select Sign Extension. The REGF_MODE1.SSE bit selects whether the core unit sign-extend short-word, 16-bit data (if 1) or zero-fill the upper 16 bits (if 0).
13 (R/W)	ALUSAT	ALU Saturation. The REGF_MODE1.ALUSAT bit selects whether the computational units saturate results on positive or negative fixed-point overflows (if 1) or return unsaturated results (if 0).

Table 29-19: REGF_MODE1 Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
12 (R/W)	IRPTEN	Global Interrupt Enable. The `REGF_MODE1.IRPTEN` bit enables (if set, = 1) or disables (if cleared, = 0) all maskable interrupts. This bit provides a global control for interrupt masking, but it does not replace individual interrupt mask or unmask settings.
11 (R/W)	NESTM	Nest Multiple Interrupts Enable. The `REGF_MODE1.NESTM` bit enables (nest if set, = 1) or disables (no nesting if cleared, = 0) interrupt nesting in the interrupt controller. When interrupt nesting is disabled, a higher priority interrupt can not interrupt a lower priority interrupt's service routine. Other interrupts are latched as they occur, but the processor processes them after the active routine finishes. When interrupt nesting is enabled, a higher priority interrupt can interrupt a lower priority interrupt's service routine. Lower interrupts are latched as they occur, but the processor processes them after the nested routines finish.
10 (R/W)	SRRFL	Secondary Registers for Register File Low Enable. The `REGF_MODE1.SRRFL` bit enables (use secondary if set, = 1) or disables (use primary if cleared, = 0) secondary data registers for the lower half (R7-R0/S7-S0) of the computational units.
7 (R/W)	SRRFH	Secondary Registers for Register File High Enable. The `REGF_MODE1.SRRFH` bit enables (use secondary if set, = 1) or disables (use primary if cleared, = 0) secondary data registers for the upper half (R15-R8/S15-S8) of the computational units.
6 (R/W)	SRD2L	Secondary Registers for DAG2 Low Enable. The `REGF_MODE1.SRD2L` bit enables (use secondary if set, = 1) or disables (use primary if cleared, = 0) secondary DAG2 registers for the lower half (I, M, L, B11-8) of the address generator.
5 (R/W)	SRD2H	Secondary Registers for DAG2 High Enable. The `REGF_MODE1.SRD2H` bit enables (use secondary if set, = 1) or disables (use primary if cleared, = 0) secondary DAG2 registers for the upper half (I, M, L, B15-12) of the address generator.
4 (R/W)	SRD1L	Secondary Registers for DAG1 Low Enable. The `REGF_MODE1.SRD1L` bit enables (use secondary if set, = 1) or disables (use primary if cleared, = 0) secondary DAG1 registers for the lower half (I, M, L, B3-0) of the address generator.
3 (R/W)	SRD1H	Secondary Registers for DAG1 High Enable. The `REGF_MODE1.SRD1H` bit enables (use secondary if set, = 1) or disables (use primary if cleared, = 0) secondary DAG1 registers for the upper half (I, M, L, B7-4) of the address generator.

Table 29-19: REGF_MODE1 Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
2 (R/W)	SRCU	MRx Result Registers Swap Enable. The `REGF_MODE1.SRCU` bit ... Enables the swapping of the MRF and MRB registers contents if set (= 1). This can be used as foreground and background registers. In SIMD Mode the swapping also performed between MSF and MSB registers. This works similar to the data register swapping instructions Rx<->Sx.
1 (R/W)	BR0	Bit-Reverse Addressing I0. The `REGF_MODE1.BR0` bit enables (if set, = 1) or disables (if cleared, = 0) bit-reversed addressing for accesses that are indexed with data address generator (DAG) index 0 (`I1`) register.
0 (R/W)	BR8	Bit-Reverse Addressing I8. The `REGF_MODE1.BR8` bit enables (if set, = 1) or disables (if cleared, = 0) bit-reversed addressing for accesses that are indexed with data address generator (DAG) index 8 (`I8`) register.

Mode 1 Stack (Top Entry) Register

It is possible to read and write the REGF_MODE1 register value in the top entry of the status stack through REGF_MODE1STK register.

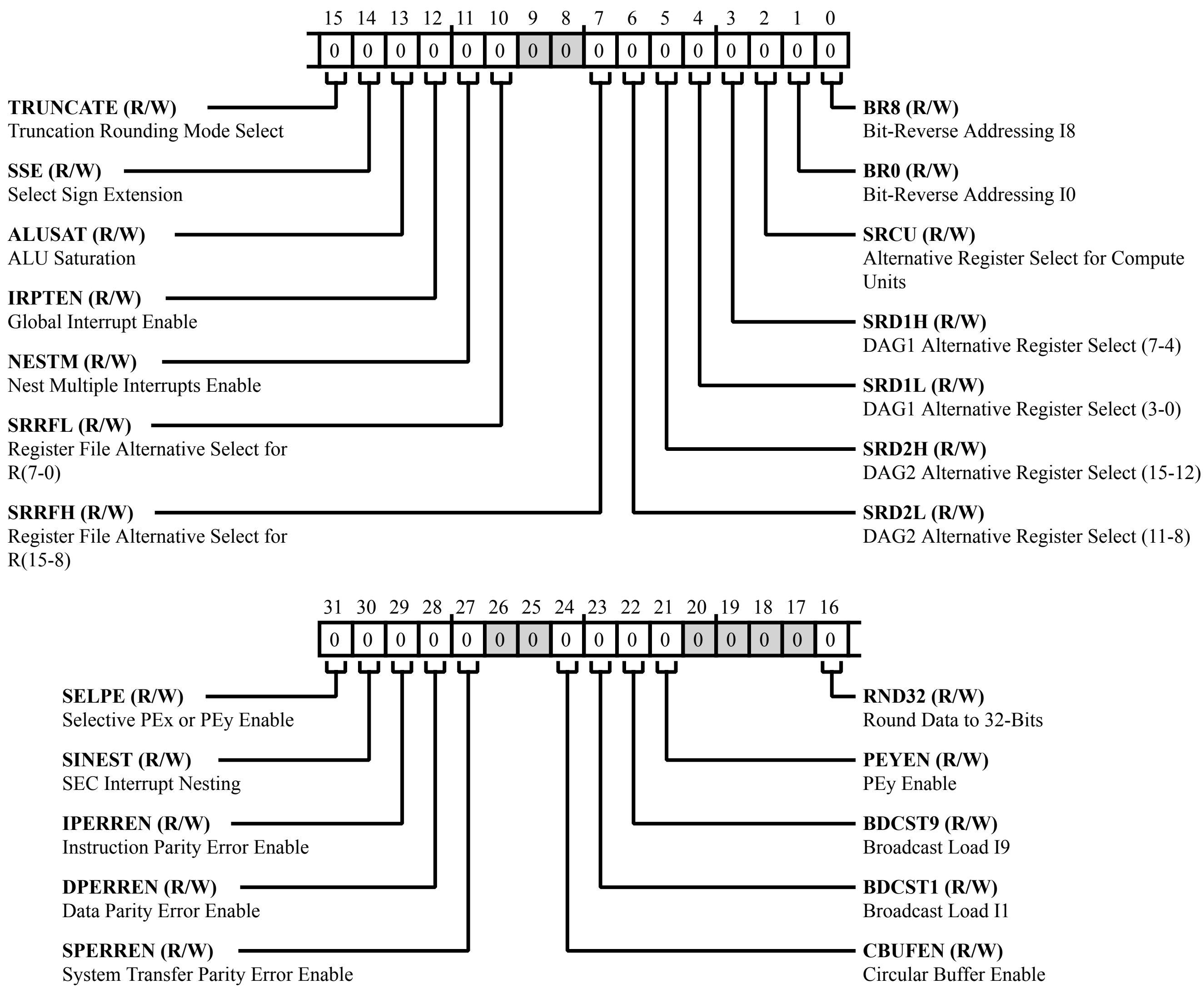

Figure 29-19: REGF_MODE1STK Register Diagram

Table 29-20: REGF_MODE1STK Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31 (R/W)	SELPE	Selective PEx or PEy Enable. The REGF_MODE1STK.SELPE bit provides access to the most recently pushed REGF_MODE1.SELPE bit on the status stack. For more information, see the REGF_MODE1.SELPE bit description.

Table 29-20: REGF_MODE1STK Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
30 (R/W)	SINEST	SEC Interrupt Nesting. The REGF_MODE1STK.SINEST bit provides access to the most recently pushed REGF_MODE1.SINEST bit on the status stack. For more information, see the REGF_MODE1.SINEST bit description.
29 (R/W)	IPERREN	Instruction Parity Error Enable. The REGF_MODE1STK.IPERREN bit provides access to the most recently pushed REGF_MODE1.IPERREN bit on the status stack. For more information, see the REGF_MODE1.IPERREN bit description.
28 (R/W)	DPERREN	Data Parity Error Enable. The REGF_MODE1STK.DPERREN bit provides access to the most recently pushed REGF_MODE1.DPERREN bit on the status stack. For more information, see the REGF_MODE1.DPERREN bit description.
27 (R/W)	SPERREN	System Transfer Parity Error Enable. The REGF_MODE1STK.SPERREN bit provides access to the most recently pushed REGF_MODE1.SPERREN bit on the status stack. For more information, see the REGF_MODE1.SPERREN bit description.
24 (R/W)	CBUFEN	Circular Buffer Enable. The REGF_MODE1STK.CBUFEN bit provides access to the most recently pushed REGF_MODE1.CBUFEN bit on the status stack. For more information, see the REGF_MODE1.CBUFEN bit description.
23 (R/W)	BDCST1	Broadcast Load I1. The REGF_MODE1STK.BDCST1 bit provides access to the most recently pushed REGF_MODE1.BDCST1 bit on the status stack. For more information, see the REGF_MODE1.BDCST1 bit description.
22 (R/W)	BDCST9	Broadcast Load I9. The REGF_MODE1STK.BDCST9 bit provides access to the most recently pushed REGF_MODE1.BDCST9 bit on the status stack. For more information, see the REGF_MODE1.BDCST9 bit description.
21 (R/W)	PEYEN	PEy Enable. The REGF_MODE1STK.PEYEN bit provides access to the most recently pushed REGF_MODE1.PEYEN bit on the status stack. For more information, see the REGF_MODE1.PEYEN bit description.
16 (R/W)	RND32	Round Data to 32-Bits. The REGF_MODE1STK.RND32 bit provides access to the most recently pushed REGF_MODE1.RND32 bit on the status stack. For more information, see the REGF_MODE1.RND32 bit description.

Table 29-20: REGF_MODE1STK Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
15 (R/W)	TRUNCATE	Truncation Rounding Mode Select. The REGF_MODE1STK.TRUNCATE bit provides access to the most recently pushed REGF_MODE1.TRUNCATE bit on the status stack. For more information, see the REGF_MODE1.TRUNCATE bit description.
14 (R/W)	SSE	Select Sign Extension. The REGF_MODE1STK.SSE bit provides access to the most recently pushed REGF_MODE1.SSE bit on the status stack. For more information, see the REGF_MODE1.SSE bit description.
13 (R/W)	ALUSAT	ALU Saturation. The REGF_MODE1STK.ALUSAT bit provides access to the most recently pushed REGF_MODE1.ALUSAT bit on the status stack. For more information, see the REGF_MODE1.ALUSAT bit description.
12 (R/W)	IRPTEN	Global Interrupt Enable. The REGF_MODE1STK.IRPTEN bit provides access to the most recently pushed REGF_MODE1.IRPTEN bit on the status stack. For more information, see the REGF_MODE1.IRPTEN bit description.
11 (R/W)	NESTM	Nest Multiple Interrupts Enable. The REGF_MODE1STK.NESTM bit provides access to the most recently pushed REGF_MODE1.NESTM bit on the status stack. For more information, see the REGF_MODE1.NESTM bit description.
10 (R/W)	SRRFL	Register File Alternative Select for R(7-0). The REGF_MODE1STK.SRRFL bit provides access to the most recently pushed REGF_MODE1.SRRFL bit on the status stack. For more information, see the REGF_MODE1.SRRFL bit description.
7 (R/W)	SRRFH	Register File Alternative Select for R(15-8). The REGF_MODE1STK.SRRFH bit provides access to the most recently pushed REGF_MODE1.SRRFH bit on the status stack. For more information, see the REGF_MODE1.SRRFH bit description.
6 (R/W)	SRD2L	DAG2 Alternative Register Select (11-8). The REGF_MODE1STK.SRD2L bit provides access to the most recently pushed REGF_MODE1.SRD2L bit on the status stack. For more information, see the REGF_MODE1.SRD2L bit description.
5 (R/W)	SRD2H	DAG2 Alternative Register Select (15-12). The REGF_MODE1STK.SRD2H bit provides access to the most recently pushed REGF_MODE1.SRD2H bit on the status stack. For more information, see the REGF_MODE1.SRD2H bit description.

Table 29-20: REGF_MODE1STK Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
4 (R/W)	SRD1L	DAG1 Alternative Register Select (3-0). The REGF_MODE1STK.SRD1L bit provides access to the most recently pushed REGF_MODE1.SRD1L bit on the status stack. For more information, see the REGF_MODE1.SRD1L bit description.
3 (R/W)	SRD1H	DAG1 Alternative Register Select (7-4). The REGF_MODE1STK.SRD1H bit provides access to the most recently pushed REGF_MODE1.SRD1H bit on the status stack. For more information, see the REGF_MODE1.SRD1H bit description.
2 (R/W)	SRCU	Alternative Register Select for Compute Units. The REGF_MODE1STK.SRCU bit provides access to the most recently pushed REGF_MODE1.SRCU bit on the status stack. For more information, see the REGF_MODE1.SRCU bit description.
1 (R/W)	BR0	Bit-Reverse Addressing I0. The REGF_MODE1STK.BR0 bit provides access to the most recently pushed REGF_MODE1.BR0 bit on the status stack. For more information, see the REGF_MODE1.BR0 bit description.
0 (R/W)	BR8	Bit-Reverse Addressing I8. The REGF_MODE1STK.BR8 bit provides access to the most recently pushed REGF_MODE1.BR8 bit on the status stack. For more information, see the REGF_MODE1.BR8 bit description.

Mode Control 2 Register

The REGF_MODE2 register controls operating modes for the computation units and other processor core resources, see the bit descriptions for detailed information.

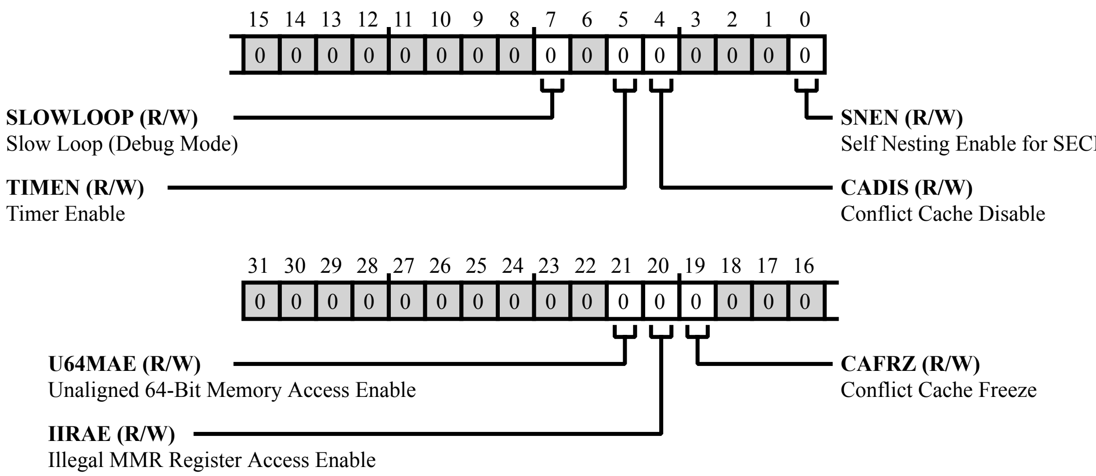

Figure 29-20: REGF_MODE2 Register Diagram

Table 29-21: REGF_MODE2 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
21 (R/W)	U64MAE	Unaligned 64-Bit Memory Access Enable. The REGF_MODE2.U64MAE bit enables (if set, = 1) or disables (if cleared, = 0) detection of unaligned long word accesses. If this bit is set, the processor flags an unaligned long word access by setting the U64MA bit in the REGF_STKYX register or the REGF_STKYY register.
20 (R/W)	IIRAE	Illegal MMR Register Access Enable. The REGF_MODE2.IIRAE bit enables (if set, = 1) or disables (if cleared, = 0) illegal CMMR/SMMR register accesses. When this bit is set, the illegal MMR register accesses set the IIRA sticky status bit. For more information about illegal MMR register access, see the descriptions of the sticky status registers: REGF_STKYX and REGF_STKYY.
19 (R/W)	CAFRZ	Conflict Cache Freeze. The REGF_MODE2.CAFRZ bit freezes the conflict cache (retain contents if set, = 1) or thaws the cache (allow new input if cleared, = 0).
7 (R/W)	SLOWLOOP	Slow Loop (Debug Mode). The REGF_MODE2.SLOWLOOP bit enables slow loop operation for user mode debug operations. This bit can be set to override the opcode of a F1-active loop. When the REGF_MODE2.SLOWLOOP bit is set, all counter-based loops execute in E2-active mode. Used primarily for the debugger.

Table 29-21: REGF_MODE2 Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
5 (R/W)	TIMEN	Timer Enable. The REGF_MODE2.TIMEN bit enables the core timer (starts, if set, = 1) or disables the core timer (stops, if cleared, = 0).
4 (R/W)	CADIS	Conflict Cache Disable. The REGF_MODE2.CADIS bit disables the conflict cache (if set, = 1) or enables the conflict cache (if cleared, = 0).
0 (R/W)	SNEN	Self Nesting Enable for SECI. The REGF_MODE2.SNEN bit enables self-nesting for the SEC interrupt (SECI). When this bit is set, the SECI interrupt can latch even when SECI interrupt is currently being serviced (bit is set in the REGF_IMASKP register). If the REGF_MODE1.IRPTEN and REGF_MODE1.NESTM bits also are set and the SECI interrupt is currently being serviced, the SECI interrupt is not masked, but lower priority interrupts are masked. If a higher priority interrupt interrupts SECI, the SECI interrupt becomes masked.

Multiplier Results 0 (PEx) Background Register

The REGF_MR0B contains the least significant 32 bits of the REGF_MRB register. For more information, see the REGF_MRB register description.

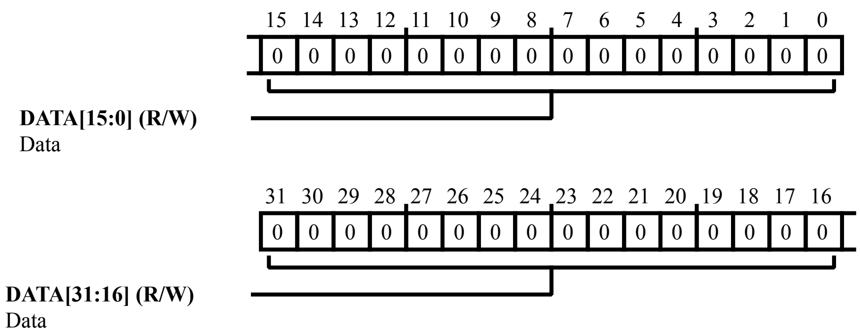

Figure 29-21: REGF_MR0B Register Diagram

Table 29-22: REGF_MR0B Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_MR0B.DATA bit field contains the least significant 32-bits of results data.

Multiplier Results 0 (PEx) Foreground Register

The REGF_MR0F contains the least significant 32 bits of the REGF_MRF register. For more information, see the REGF_MRF register description.

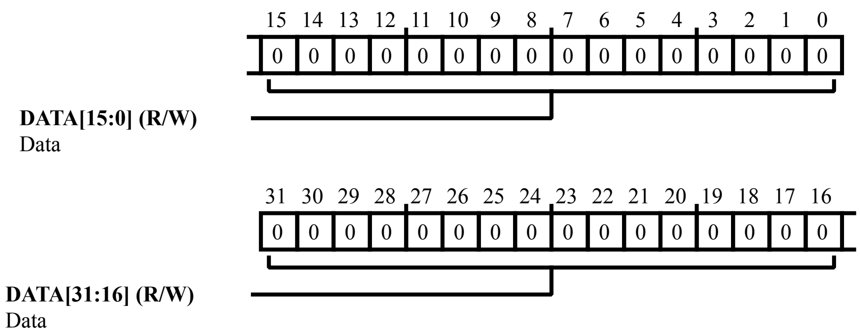

Figure 29-22: REGF_MR0F Register Diagram

Table 29-23: REGF_MR0F Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_MR0F.DATA bit field contains the least significant 32-bits of results data.

Multiplier Results 1 (PEx) Background Register

The REGF_MR1B contains 32 bits (bits 63-32) of the REGF_MRB register. For more information, see the REGF_MRB register description.

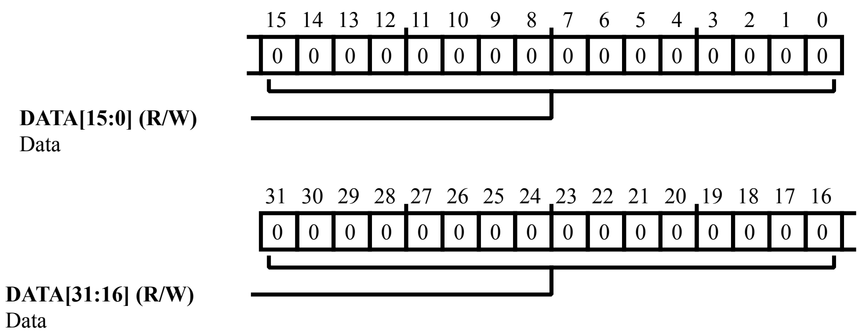

Figure 29-23: REGF_MR1B Register Diagram

Table 29-24: REGF_MR1B Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_MR1B.DATA bit field contains 32-bits of results data (bits 63-32).

Multiplier Results 1 (PEx) Foreground Register

The REGF_MR1F contains 32 bits (bits 63-32) of the REGF_MRF register. For more information, see the REGF_MRF register description.

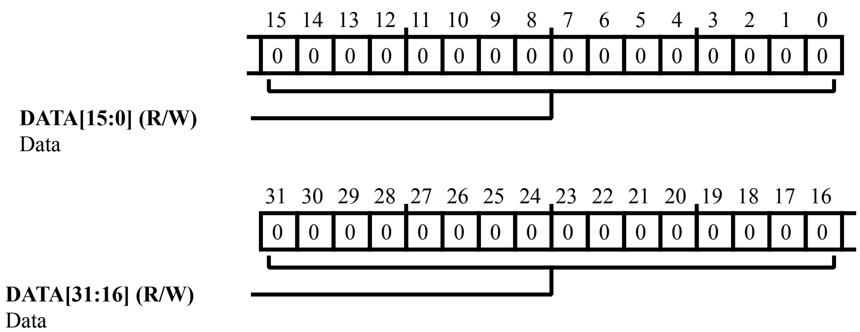

Figure 29-24: REGF_MR1F Register Diagram

Table 29-25: REGF_MR1F Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_MR1F.DATA bit field contains 32-bits of results data (bits 63-32).

Multiplier Results 2 (PEx) Background Register

The REGF_MR2B contains the most significant 16 bits of the REGF_MRB register. For more information, see the REGF_MRB register description.

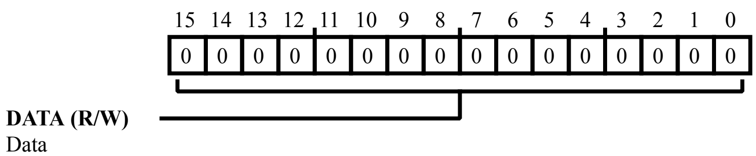

Figure 29-25: REGF_MR2B Register Diagram

Table 29-26: REGF_MR2B Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
15:0 (R/W)	DATA	Data. The REGF_MR2B.DATA bit field contains the most significant 16-bits of results data.

Multiplier Results 2 (PEx) Foreground Register

The REGF_MR2F contains the most significant 16 bits of the REGF_MRF register. For more information, see the REGF_MRF register description.

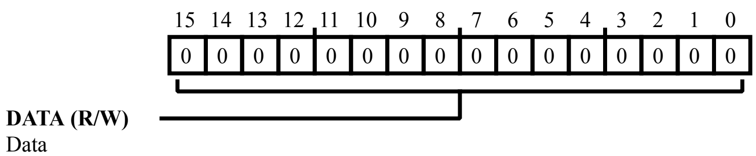

Figure 29-26: REGF_MR2F Register Diagram

Table 29-27: REGF_MR2F Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
15:0 (R/W)	DATA	Data. The REGF_MR2F.DATA bit field contains the most significant 16-bits of results data.

Multiplier Results (PEx) Background Register

Processing element x (PEx) has a foreground (primary) multiply result (MRF) register and background (alternate) results (MRB) register. The multiply accumulator (MAC) places 80-bit results of fixed-point operations in the MRF or MRB register, depending on which register has been selected (made active) through the REGF_MODE1 register.

Table 29-28: REGF_MRB Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
79:0 (R/W)	DATA	Data. The REGF_MRB.DATA bit field contains the full 80-bits of results data.

Multiplier Results (PEx) Foreground Register

Processing element x (PEx) has a foreground (primary) multiply result (MRF) register and background (alternate) results (MRB) register. The multiply accumulator (MAC) places 80-bit results of fixed-point operations in the MRF or MRB register, depending on which register has been selected (made active) through the REGF_MODE1 register.

Table 29-29: REGF_MRF Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
79:0 (R/W)	DATA	Data. The REGF_MRF.DATA bit field contains the full 80-bits of results data.

Multiplier Results 0 (PEy) Background Register

The REGF_MS0B contains the least significant 32 bits of the REGF_MSB register. For more information, see the REGF_MSB register description.

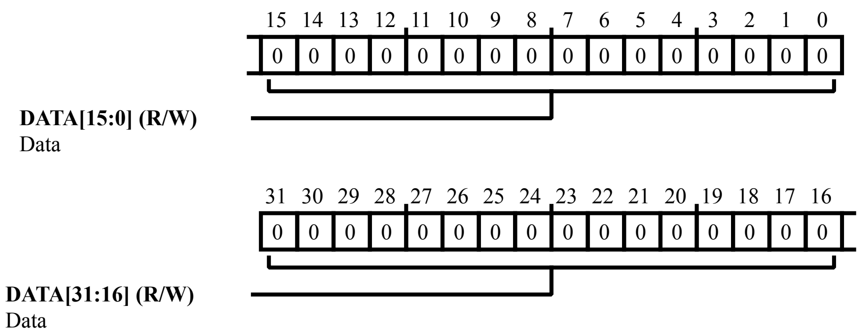

Figure 29-27: REGF_MS0B Register Diagram

Table 29-30: REGF_MS0B Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_MS0B.DATA bit field contains the least significant 32-bits of results data.

Multiplier Results 0 (PEy) Foreground Register

The REGF_MS0F contains the least significant 32 bits of the REGF_MSF register. For more information, see the REGF_MSF register description.

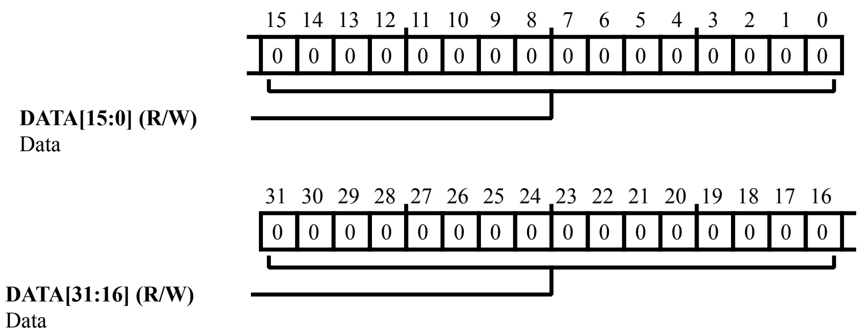

Figure 29-28: REGF_MS0F Register Diagram

Table 29-31: REGF_MS0F Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_MS0F.DATA bit field contains the least significant 32-bits of results data.

Multiplier Results 1 (PEy) Background Register

The REGF_MS1B contains 32 bits (bits 63-32) of the REGF_MSB register. For more information, see the REGF_MSB register description.

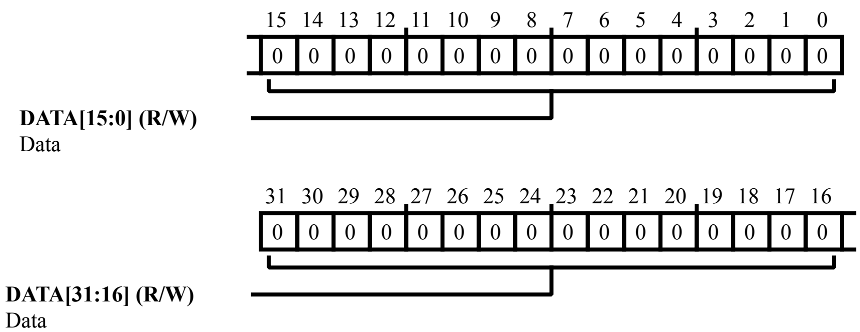

Figure 29-29: REGF_MS1B Register Diagram

Table 29-32: REGF_MS1B Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_MS1B.DATA bit field contains 32-bits of results data (bits 63-32).

Multiplier Results 1 (PEy) Foreground Register

The REGF_MS1F contains 32 bits (bits 63-32) of the REGF_MSF register. For more information, see the REGF_MSF register description.

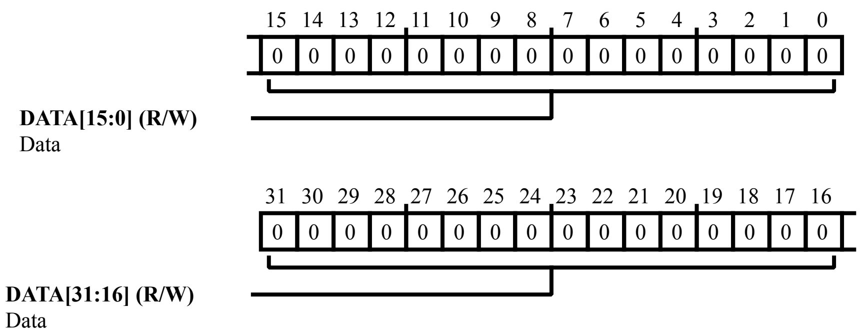

Figure 29-30: REGF_MS1F Register Diagram

Table 29-33: REGF_MS1F Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_MS1F.DATA bit field contains 32-bits of results data (bits 63-32).

Multiplier Results 2 (PEy) Background Register

The REGF_MS2B contains the most significant 16 bits of the REGF_MSB register. For more information, see the REGF_MSB register description.

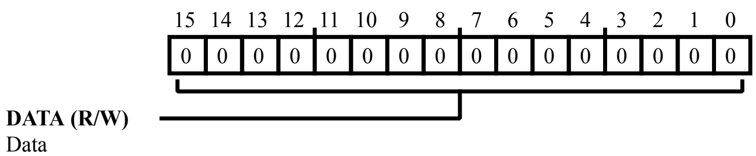

Figure 29-31: REGF_MS2B Register Diagram

Table 29-34: REGF_MS2B Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
15:0 (R/W)	DATA	Data. The REGF_MS2B.DATA bit field contains the most significant 16-bits of results data.

Multiplier Results 2 (PEy) Foreground Register

The REGF_MS2F contains the most significant 16 bits of the REGF_MSF register. For more information, see the REGF_MSF register description.

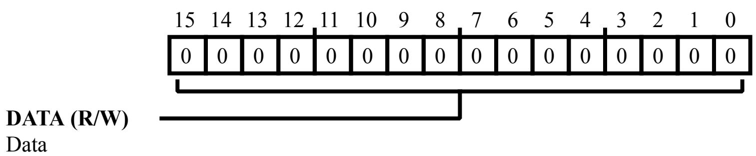

Figure 29-32: REGF_MS2F Register Diagram

Table 29-35: REGF_MS2F Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
15:0 (R/W)	DATA	Data. The REGF_MS2F.DATA bit field contains the most significant 16-bits of results data.

Multiplier Results (PEy) Background Register

Processing element y (PEy) has a foreground (primary) multiply result (MSF) register and background (alternate) results (MSB) register. The multiply accumulator (MAC) places 80-bit results of fixed-point operations in the MSF or MSB register, depending on which register has been selected (made active) through the REGF_MODE1 register.

Table 29-36: REGF_MSB Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
79:0 (R/W)	DATA	Data. The REGF_MSB.DATA bit field contains the full 80-bits of results data.

Multiplier Results (PEy) Foreground Register

Processing element y (PEy) has a foreground (primary) multiply result (MSF) register and background (alternate) results (MSB) register. The multiply accumulator (MAC) places 80-bit results of fixed-point operations in the MSF or MSB register, depending on which register has been selected (made active) through the REGF_MODE1 register.

Table 29-37: REGF_MSF Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
79:0 (R/W)	DATA	Data. The REGF_MSF.DATA bit field contains the full 80-bits of results data.

Modify Registers

The data address generators (DAGs) update stored addresses using modify (REGF_M[n]) registers. Registers M0 through M7 are for DAG1, and registers M8 through M15 are for DAG2. A modify register provides the increment or step size by which an index register is pre-modified or post-modified during a register move.

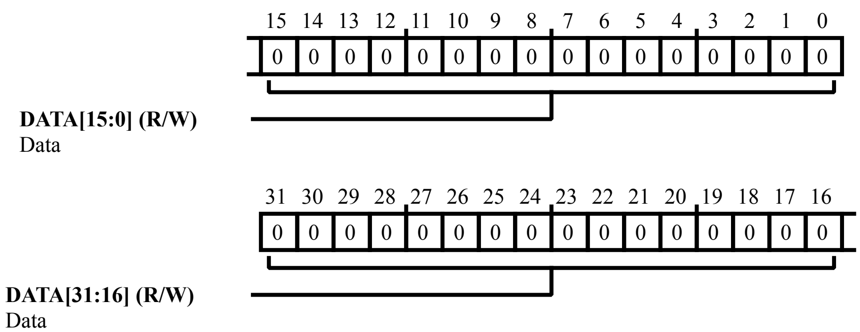

Figure 29-33: REGF_M[n] Register Diagram

Table 29-38: REGF_M[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_M[n].DATA bit field contains address modifier data.

Program Counter Register

The program counter (REGF_PC) register reads the last stage (E) in the instruction pipeline and contains the 24-bit address of the instruction that the processor executes on the next cycle. The PC register works with the program counter stack, REGF_PCSTK register, which stores return addresses and top-of-loop addresses. All PC relative branch instruction require access to the register.

```
n: R0=PC; /* Execution address in PC */
n+1: instruction1;
n+2: instruction2;
n+3: instruction3;
n+4: instruction4;
n+5: instruction5;
```

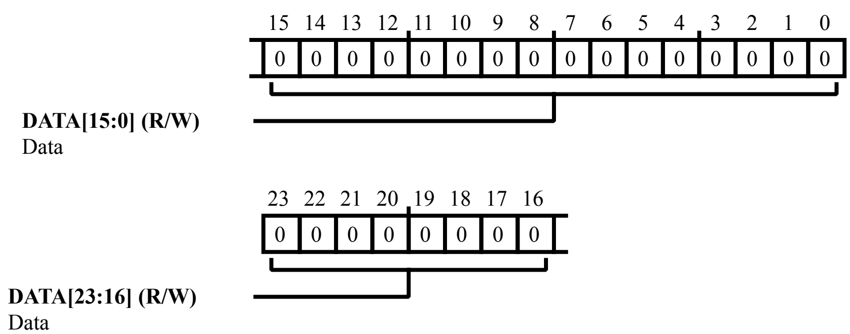

Figure 29-34: REGF_PC Register Diagram

Table 29-39: REGF_PC Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
23:0 (R/W)	DATA	Data. The REGF_PC.DATA bit field contains address data.

Program Counter Stack Register

The program counter stack (REGF_PCSTK) register contains the address of the top of the PC stack.

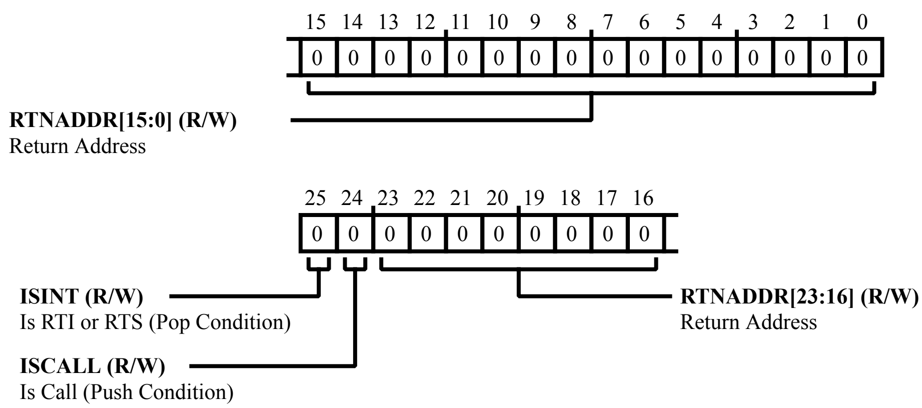

Figure 29-35: REGF_PCSTK Register Diagram

Table 29-40: REGF_PCSTK Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
25 (R/W)	ISINT	Is RTI or RTS (Pop Condition). The REGF_PCSTK.ISINT bit indicates whether the REGF_PCSTK.RTNADDR bit field contains an address for an RTS or RTI operation that is popped off of the stack.
24 (R/W)	ISCALL	Is Call (Push Condition). The REGF_PCSTK.ISCALL bit indicates whether the REGF_PCSTK.RTNADDR bit field contains an address for a Call, IVT branch, or Do Until operation that is pushed on to the stack.
23:0 (R/W)	RTNADDR	Return Address. The REGF_PCSTK.RTNADDR bit field contains the return address.

Program Counter Stack Pointer Register

The program counter stack pointer (REGF_PCSTKP) register contains the value of PCSTKP. This value is:

- 0 - when the PC stack is empty
- 1 through 30 - when the stack contains data
- 31 - when the stack overflows

A write to PCSTKP takes effect after a one-cycle delay. If the PC stack is overflowed, a write to PCSTKP has no effect.

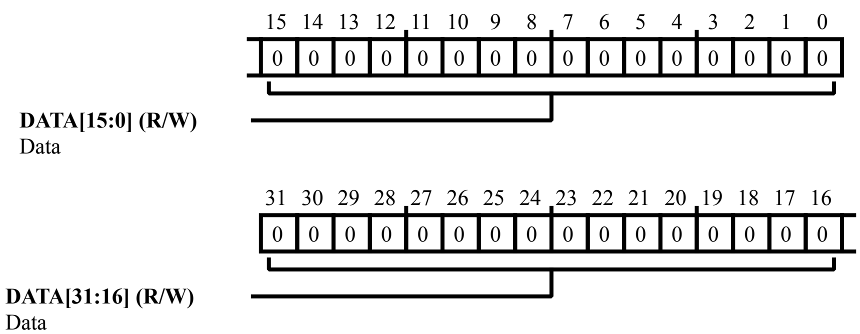

Figure 29-36: REGF_PCSTKP Register Diagram

Table 29-41: REGF_PCSTKP Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_PCSTKP.DATA bit field contains stack pointer data.

PMD-DMD Bus Exchange Register

The PM bus exchange (REGF_PX) register permits data to flow between the PM and DM data buses. This register can work as one 64-bit register or as two 32-bit registers (REGF_PX1 and REGF_PX2). The REGF_PX1 register is the lower 32 bits of the REGF_PX register, and the REGF_PX2 register is the upper 32 bits of the REGF_PX register.

The REGF_PX register lets programs transfer data between the data buses, but cannot be an input or output in a calculation. For more information about using the REGF_PX register, see the Combined Data Bus Exchange Register section.

Table 29-42: REGF_PX Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
63:0 (R/W)	DATA	Data. The REGF_PX.DATA bit field contains 64-bits of PMD-DMD bus-exchange data.

PMD-DMD Bus Exchange 1 Register

The REGF_PX1 register is the lower 32 bits of the REGF_PX register. For more information, see the REGF_PX register description.

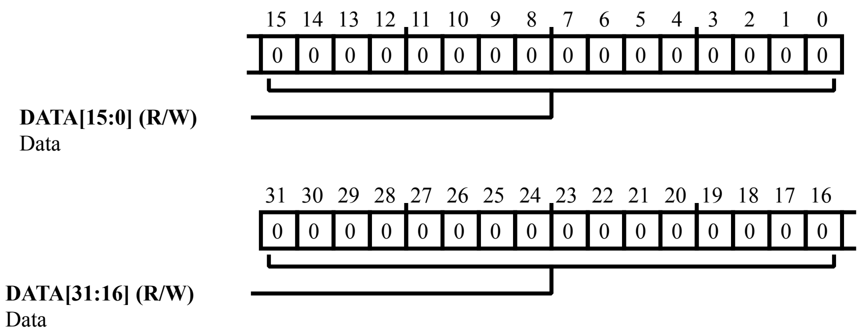

Figure 29-37: REGF_PX1 Register Diagram

Table 29-43: REGF_PX1 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_PX1.DATA bit field contains the least significant 32-bits of PMD-DMD bus-exchange data.

PMD-DMD Bus Exchange 2 Register

The REGF_PX2 register is the upper 32 bits of the REGF_PX register. For more information, see the REGF_PX register description.

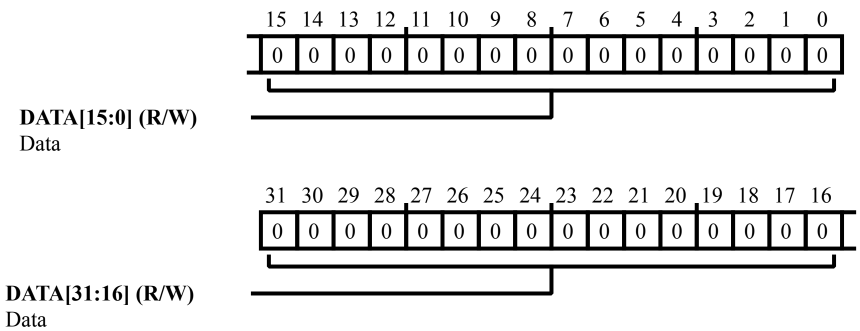

Figure 29-38: REGF_PX2 Register Diagram

Table 29-44: REGF_PX2 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_PX2.DATA bit field contains the most significant 32-bits of PMD-DMD bus-exchange data.

Register File (PEx) Data Registers (Rx, Fx)

Each of the processing elements (PEx and PEy) has a data register file comprising 16 40-bit registers. The processing elements use these 40-bit data registers to transfer data between the data buses and the computation units. These registers also provide local storage for operands and results.

Each data register can be accessed using either an R or F prefixed name. For example R0 is the same register as F0. The R or F prefixes on register names do not effect the 32-bit or 40-bit data transfer. The naming convention determines how the ALU, multiplier, and shifter treat the data and determines which processing element's data registers are being used. For more information about using these registers, see the Register Files chapter.

Table 29-45: REGF_R[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
39:0 (R/W)	DATA	Data. The `REGF_R[n].DATA` bit field contains data, which is treated as fixed-point data (Rn syntax) or floating-point data (Fn syntax).

Sticky Status (PEx) Register

The REGF_STKYX register indicates sticky status for processing element x (PEx) operations and some program sequencer stacks. This register only indicates status for PEx operations.

Note that sticky bits do not clear themselves after the condition is no longer true. They remain "sticky" until cleared by the program.

The processor sets a sticky bit in response to a condition. For example, the processor sets the REGF_STKYX.AIS bit when an invalid ALU floating-point operation sets the REGF_ASTATX.AI bit. The processor clears AI if the next ALU operation is valid. However the AIS bit remains set until a program clears it. Interrupt service routines (ISRs) must clear their interrupt's corresponding sticky bit so the processor can detect a re-occurrence of the condition. For example, an ISR for a floating-point underflow exception interrupt (FLTUI) clears the REGF_STKYX.AUS bit near the beginning of the routine.

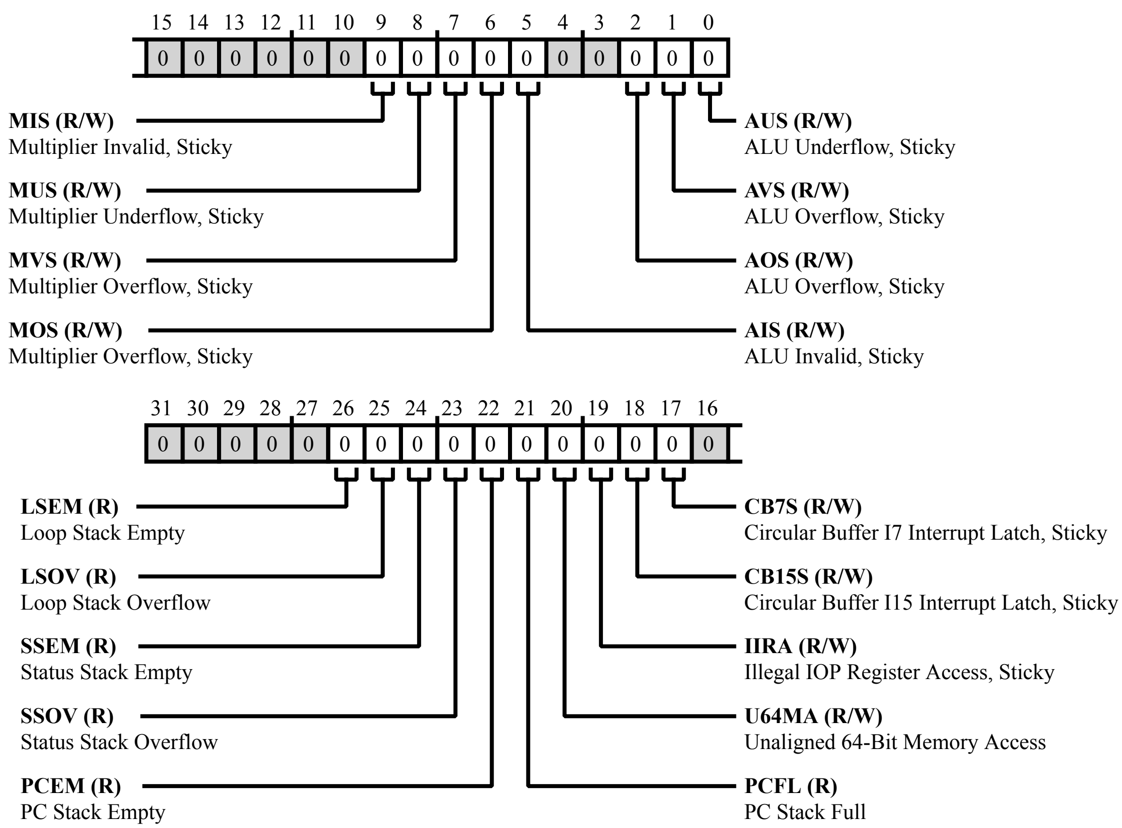

Figure 29-39: REGF_STKYX Register Diagram

Table 29-46: REGF_STKYX Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
26 (R/NW)	LSEM	Loop Stack Empty. The REGF_STKYX.LSEM bit indicates whether the loop counter stack and loop stack are empty (if 1) or not empty (if 0). This bit is not sticky, and it is cleared by a push operation.
25 (R/NW)	LSOV	Loop Stack Overflow. The REGF_STKYX.LSOV bit provides sticky status, indicating whether the loop counter stack and loop stack are overflowed (if 1) or are not overflowed (if 0).
24 (R/NW)	SSEM	Status Stack Empty. The REGF_STKYX.SSEM bit indicates whether the status stack is empty (if 1) or not empty (if 0)-not sticky. This bit is cleared by a push.
23 (R/NW)	SSOV	Status Stack Overflow. The REGF_STKYX.SSOV bit indicates whether the status stack is overflowed (if 1) or not overflowed (if 0). This bit is a sticky bit.
22 (R/NW)	PCEM	PC Stack Empty. The REGF_STKYX.PCEM bit indicates whether the PC stack is empty (if 1) or not empty (if 0). This bit is not sticky and is cleared by a push operation.
21 (R/NW)	PCFL	PC Stack Full. The REGF_STKYX.PCFL bit indicates whether the PC stack is full (if 1) or not full (if 0). This bit is not a sticky bit and is cleared by a pop operation.
20 (R/W)	U64MA	Unaligned 64-Bit Memory Access. The REGF_STKYX.U64MA bit indicates whether (if set, = 1) a forced Normal word access (LW mnemonic) addressing an uneven memory address has occurred or (if cleared, =0) has not occurred.
19 (R/W)	IIRA	Illegal IOP Register Access, Sticky. The REGF_STKYX.IIRA bit provides a sticky indicator for the illegal I/O processor (IOP) register accesses, which are detected when the REGF_MODE2.IIRAE bit is set. For more information, see the REGF_MODE2.IIRAE bit description.
18 (R/W)	CB15S	Circular Buffer I15 Interrupt Latch, Sticky. The REGF_STKYX.CB15S bit provides a sticky indicator for the REGF_IRPTL.CB15I bit. For more information, see the REGF_IRPTL.CB15I bit description.
17 (R/W)	CB7S	Circular Buffer I7 Interrupt Latch, Sticky. The REGF_STKYX.CB7S bit provides a sticky indicator for the REGF_IRPTL.CB7I bit. For more information, see the REGF_IRPTL.CB7I bit description.

Table 29-46: REGF_STKYX Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
9 (R/W)	MIS	Multiplier Invalid, Sticky. The REGF_STKYX.MIS bit provides a sticky status indicator for the multiplier REGF_ASTATX.MI bit. For more information, see the REGF_ASTATX.MI bit description.
8 (R/W)	MUS	Multiplier Underflow, Sticky. The REGF_STKYX.MUS bit provides a sticky status indicator for the multiplier REGF_ASTATX.MU bit. For more information, see the REGF_ASTATX.MU bit description.
7 (R/W)	MVS	Multiplier Overflow, Sticky. The REGF_STKYX.MVS bit provides a sticky status indicator for the multiplier REGF_ASTATX.MV bit. For more information, see the REGF_ASTATX.MV bit description.
6 (R/W)	MOS	Multiplier Overflow, Sticky. The REGF_STKYX.MOS bit provides a sticky status indicator for the multiplier fixed-point overflow (REGF_ASTATX.MV bit). For more information, see see the REGF_ASTATX.MV bit description.
5 (R/W)	AIS	ALU Invalid, Sticky. The REGF_STKYX.AIS bit provides a sticky indicator for the REGF_ASTATX.AI bit. For more information, see the REGF_ASTATX.AI bit description.
2 (R/W)	AOS	ALU Overflow, Sticky. The REGF_STKYX.AOS bit provides a sticky indicator for the REGF_ASTATX.AV bit. For more information, see the REGF_ASTATX.AV bit description.
1 (R/W)	AVS	ALU Overflow, Sticky. The REGF_STKYX.AVS bit provides a sticky indicator for the REGF_ASTATX.AV bit. For more information, see the REGF_ASTATX.AV bit description.
0 (R/W)	AUS	ALU Underflow, Sticky. The REGF_STKYX.AUS bit provides a sticky indicator for the REGF_ASTATX.AZ bit. For more information, see the REGF_ASTATX.AZ bit description.

Sticky Status (PEy) Register

The REGF_STKYY register indicates sticky status for processing element y (PEy) operations and some program sequencer stacks. This register only indicates status for PEy operations.

Note that sticky bits do not clear themselves after the condition is no longer true. They remain "sticky" until cleared by the program.

The processor sets a sticky bit in response to a condition. For example, the processor sets the REGF_STKYY.AIS bit when an invalid ALU floating-point operation sets the REGF_ASTATY.AI bit. The processor clears AI if the next ALU operation is valid. However the AIS bit remains set until a program clears it. Interrupt service routines (ISRs) must clear their interrupt's corresponding sticky bit so the processor can detect a re-occurrence of the condition. For example, an ISR for a floating-point underflow exception interrupt (FLTUI) clears the REGF_STKYY.AUS bit near the beginning of the routine.

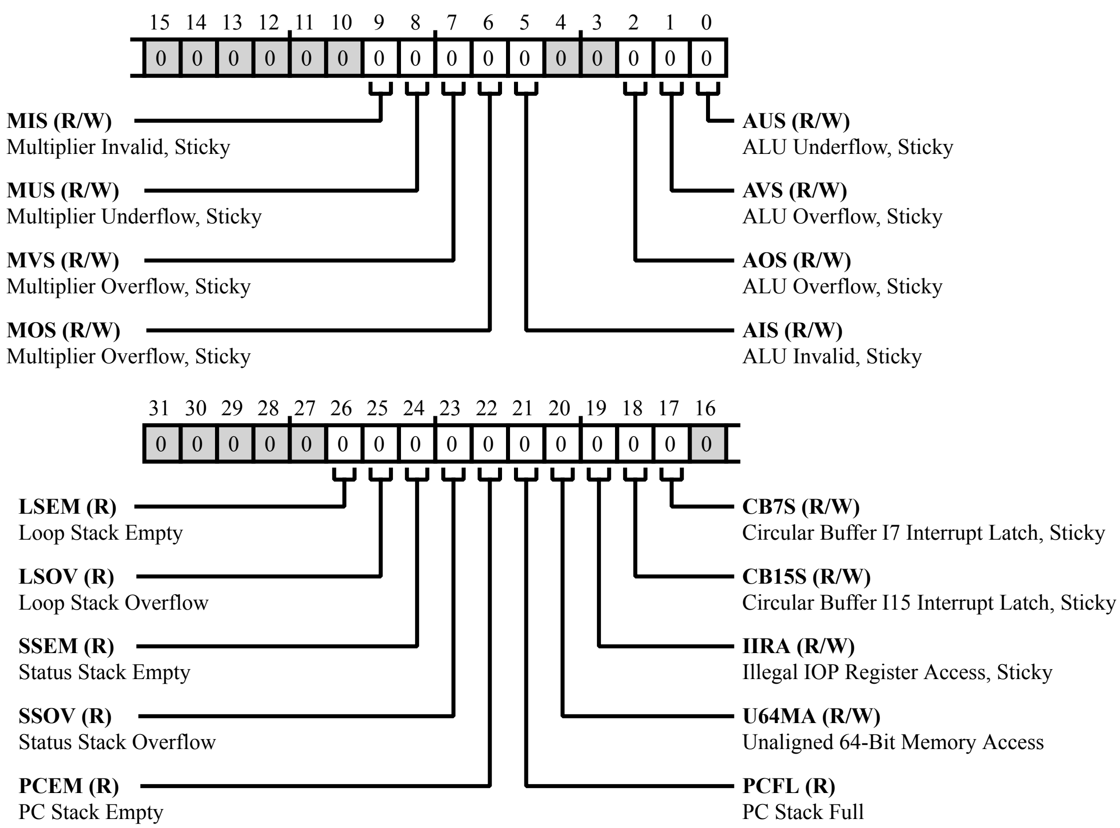

Figure 29-40: REGF_STKYY Register Diagram

Table 29-47: REGF_STKYY Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
26 (R/NW)	LSEM	Loop Stack Empty. The REGF_STKYY.LSEM bit indicates whether the loop counter stack and loop stack are empty (if 1) or not empty (if 0). This bit is not sticky, and it is cleared by a push operation.
25 (R/NW)	LSOV	Loop Stack Overflow. The REGF_STKYY.LSOV bit provides sticky status, indicating whether the loop counter stack and loop stack are overflowed (if 1) or are not overflowed (if 0).
24 (R/NW)	SSEM	Status Stack Empty. The REGF_STKYY.SSEM bit indicates whether the status stack is empty (if 1) or not empty (if 0)-not sticky. This bit is cleared by a push.
23 (R/NW)	SSOV	Status Stack Overflow. The REGF_STKYY.SSOV bit indicates whether the status stack is overflowed (if 1) or not overflowed (if 0). This bit is a sticky bit.
22 (R/NW)	PCEM	PC Stack Empty. The REGF_STKYY.PCEM bit indicates whether the PC stack is empty (if 1) or not empty (if 0). This bit is not sticky and is cleared by a push operation.
21 (R/NW)	PCFL	PC Stack Full. The REGF_STKYY.PCFL bit indicates whether the PC stack is full (if 1) or not full (if 0). This bit is not a sticky bit and is cleared by a pop operation.
20 (R/W)	U64MA	Unaligned 64-Bit Memory Access. The REGF_STKYY.U64MA bit indicates whether (if set, = 1) a forced Normal word access (LW mnemonic) addressing an uneven memory address has occurred or (if cleared, =0) has not occurred.
19 (R/W)	IIRA	Illegal IOP Register Access, Sticky. The REGF_STKYY.IIRA bit provides a sticky indicator for the illegal I/O processor (IOP) register accesses, which are detected when the REGF_MODE2.IIRAE bit is set. For more information, see the REGF_MODE2.IIRAE bit description.
18 (R/W)	CB15S	Circular Buffer I15 Interrupt Latch, Sticky. The REGF_STKYY.CB15S bit provides a sticky indicator for the REGF_IRPTL.CB15I bit. For more information, see the REGF_IRPTL.CB15I bit description.
17 (R/W)	CB7S	Circular Buffer I7 Interrupt Latch, Sticky. The REGF_STKYY.CB7S bit provides a sticky indicator for the REGF_IRPTL.CB7I bit. For more information, see the REGF_IRPTL.CB7I bit description.

Table 29-47: REGF_STKYY Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
9 (R/W)	MIS	Multiplier Invalid, Sticky. The REGF_STKYY.MIS bit provides a sticky status indicator for the multiplier REGF_ASTATY.MI bit. For more information, see the REGF_ASTATY.MI bit description.
8 (R/W)	MUS	Multiplier Underflow, Sticky. The REGF_STKYY.MUS bit provides a sticky status indicator for the multiplier REGF_ASTATY.MU bit. For more information, see the REGF_ASTATY.MU bit description.
7 (R/W)	MVS	Multiplier Overflow, Sticky. The REGF_STKYY.MVS bit provides a sticky status indicator for the multiplier REGF_ASTATY.MV bit. For more information, see the REGF_ASTATY.MV bit description.
6 (R/W)	MOS	Multiplier Overflow, Sticky. The REGF_STKYY.MOS bit provides a sticky status indicator for the multiplier fixed-point overflow (REGF_ASTATY.MV bit). For more information, see see the REGF_ASTATY.MV bit description.
5 (R/W)	AIS	ALU Invalid, Sticky. The REGF_STKYY.AIS bit provides a sticky indicator for the REGF_ASTATY.AI bit. For more information, see the REGF_ASTATY.AI bit description.
2 (R/W)	AOS	ALU Overflow, Sticky. The REGF_STKYY.AOS bit provides a sticky indicator for the REGF_ASTATY.AV bit. For more information, see the REGF_ASTATY.AV bit description.
1 (R/W)	AVS	ALU Overflow, Sticky. The REGF_STKYY.AVS bit provides a sticky indicator for the REGF_ASTATY.AV bit. For more information, see the REGF_ASTATY.AV bit description.
0 (R/W)	AUS	ALU Underflow, Sticky. The REGF_STKYY.AUS bit provides a sticky indicator for the REGF_ASTATX.AZ bit. For more information, see the REGF_ASTATX.AZ bit description.

Register File (PEy) Data Registers (Sx, SFx)

Each of the processing elements (PEx and PEy) has a data register file comprising 16 40-bit registers. The processing elements use these 40-bit data registers to transfer data between the data buses and the computation units. These registers also provide local storage for operands and results.

Each data register can be accessed using either an S or SF prefixed name. For example S0 is the same register as SF0. The S or SF prefixes on register names do not effect the 32-bit or 40-bit data transfer. The naming convention determines how the ALU, multiplier, and shifter treat the data and determines which processing element's data registers are being used. For more information about using these registers, see the Register Files chapter.

Table 29-48: REGF_S[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
39:0 (R/W)	DATA	Data. The `REGF_S[n].DATA` bit field contains data, which is treated as fixed-point data (Sn syntax) or floating-point data (SFn syntax).

Timer Count Register

The timer count REGF_TCOUNT register contains the decrementing timer count value, counting down the cycles between timer interrupts. For more information about using the REGF_TCOUNT register, see the Timer chapter.

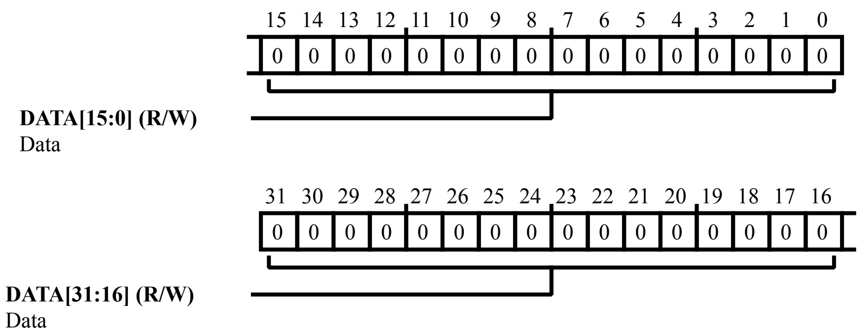

Figure 29-41: REGF_TCOUNT Register Diagram

Table 29-49: REGF_TCOUNT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_TCOUNT.DATA bit field contains timer-count data.

Timer Period Register

The timer period REGF_TPERIOD register contains the timer period, indicating the number of cycles between timer interrupts. For more information about using this register, see the Timer chapter.

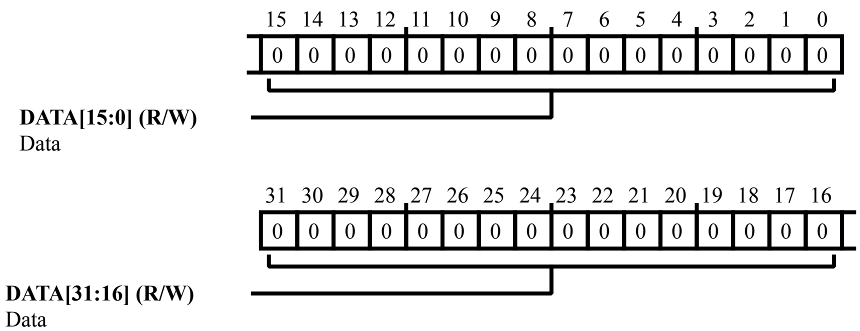

Figure 29-42: REGF_TPERIOD Register Diagram

Table 29-50: REGF_TPERIOD Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_TPERIOD.DATA bit field contains timer-period data.

User-Defined Status 1 Register

The REGF_USTAT1 register is a user-defined, general-purpose status register. Programs can use this register with bit-wise instructions (SET, CLEAR, TEST, and others). Often, programs use this register for low overhead, general-purpose flags or for temporary 32-bit storage of data.

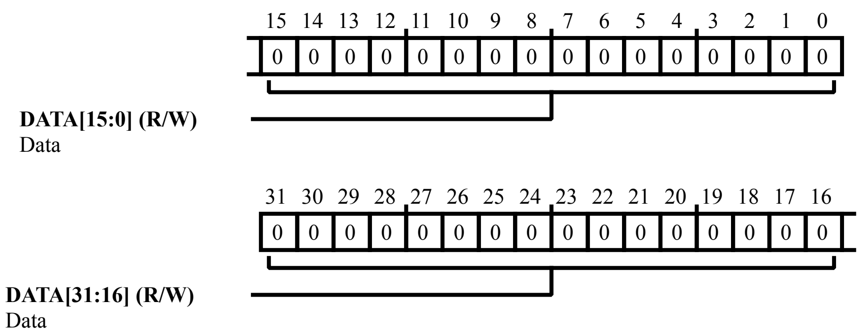

Figure 29-43: REGF_USTAT1 Register Diagram

Table 29-51: REGF_USTAT1 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_USTAT1.DATA bit field contains user-status data.

User-Defined Status 2 Register

The REGF_USTAT2 register is a user-defined, general-purpose status register. Programs can use this register with bit-wise instructions (SET, CLEAR, TEST, and others). Often, programs use this register for low overhead, general-purpose flags or for temporary 32-bit storage of data.

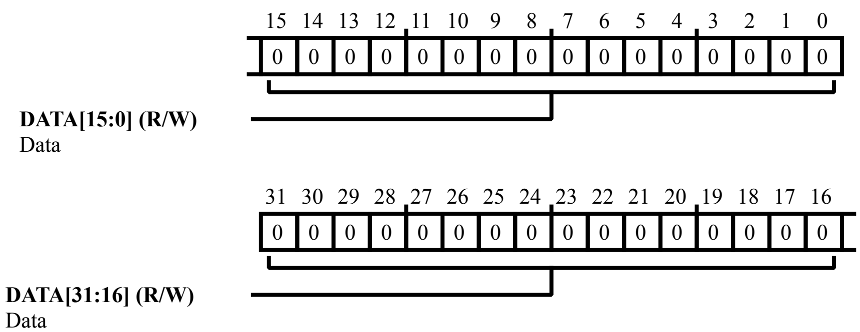

Figure 29-44: REGF_USTAT2 Register Diagram

Table 29-52: REGF_USTAT2 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The REGF_USTAT2.DATA bit field contains user-status data.

User-Defined Status 3 Register

The REGF_USTAT3 register is a user-defined, general-purpose status register. Programs can use this register with bit-wise instructions (SET, CLEAR, TEST, and others). Often, programs use this register for low overhead, general-purpose flags or for temporary 32-bit storage of data.

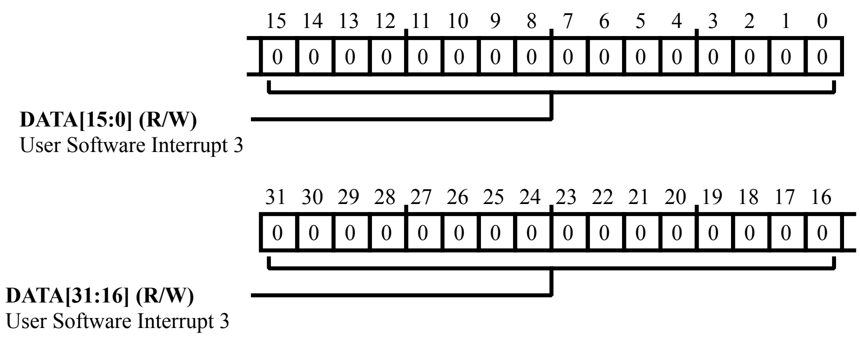

Figure 29-45: REGF_USTAT3 Register Diagram

Table 29-53: REGF_USTAT3 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	User Software Interrupt 3. The REGF_USTAT3.DATA bit field contains user-status data.

User-Defined Status 4 Register

The REGF_USTAT4 register is a user-defined, general-purpose status register. Programs can use this register with bit-wise instructions (SET, CLEAR, TEST, and others). Often, programs use this register for low overhead, general-purpose flags or for temporary 32-bit storage of data.

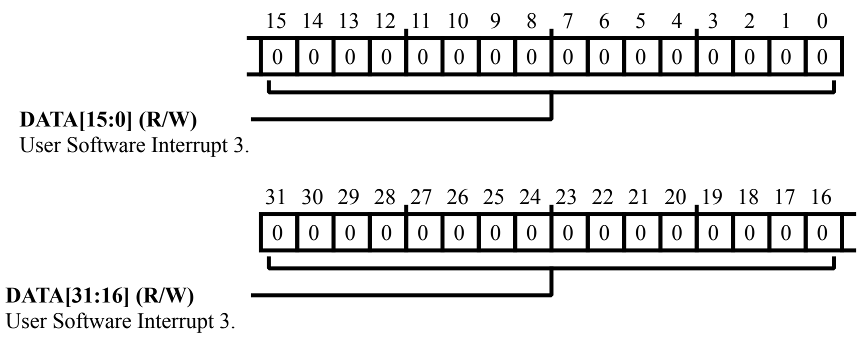

Figure 29-46: REGF_USTAT4 Register Diagram

Table 29-54: REGF_USTAT4 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	User Software Interrupt 3.. The REGF_USTAT4.DATA bit field contains user-status data.

30 SHARC-PLUS CMMR Register Descriptions

Miscellaneous core MMRs (CMMR) contains the following registers.

Table 30-1: SHARC-PLUS CMMR Register List

Name	Description
CMMR_GPERR_STAT	General-Purpose Parity Error Status Register
CMMR_PFB_NOCHRT0_END	PFB No Caching Return 0 End Address Register
CMMR_PFB_NOCHRT0_ST	PFB No Caching Return 0 Start Address Register
CMMR_PWR_GLB_CTL	Core Global Power Control Register
CMMR_PWR_L1_LS_CTL	L1 BANK SLEEP CONTROL
CMMR_PWR_L1_SD_CTL	L1 BANK SHUT DOWN CONTROL
CMMR_SYSCTL	System Control Register

General-Purpose Parity Error Status Register

The CMMR_GPERR_STAT register indicates parity error and interrupt status for L1 memory accesses. This register is considered "general-purpose" because it indicates status for all types of L1 memory accesses. These include program memory accesses, data memory accesses, accesses for system transfers, and all types of cache accesses.

After an error condition is registered, the condition is locked and remains locked until all of the status bits are cleared (clearing the error status manually) in the CMMR_GPERR_STAT register. This operation requires writing 0x0 to this register. To avoid continuous generation of this interrupt, reset the core or write 0x0 to this register in the Parity ISR.

For simultaneous errors between DM (DAG1) and PM (DAG2) data reads, only the DM error is indicated.

For simultaneous errors between slave port 1 (S1) and slave port 2 (S2), only the S1 error is indicated.

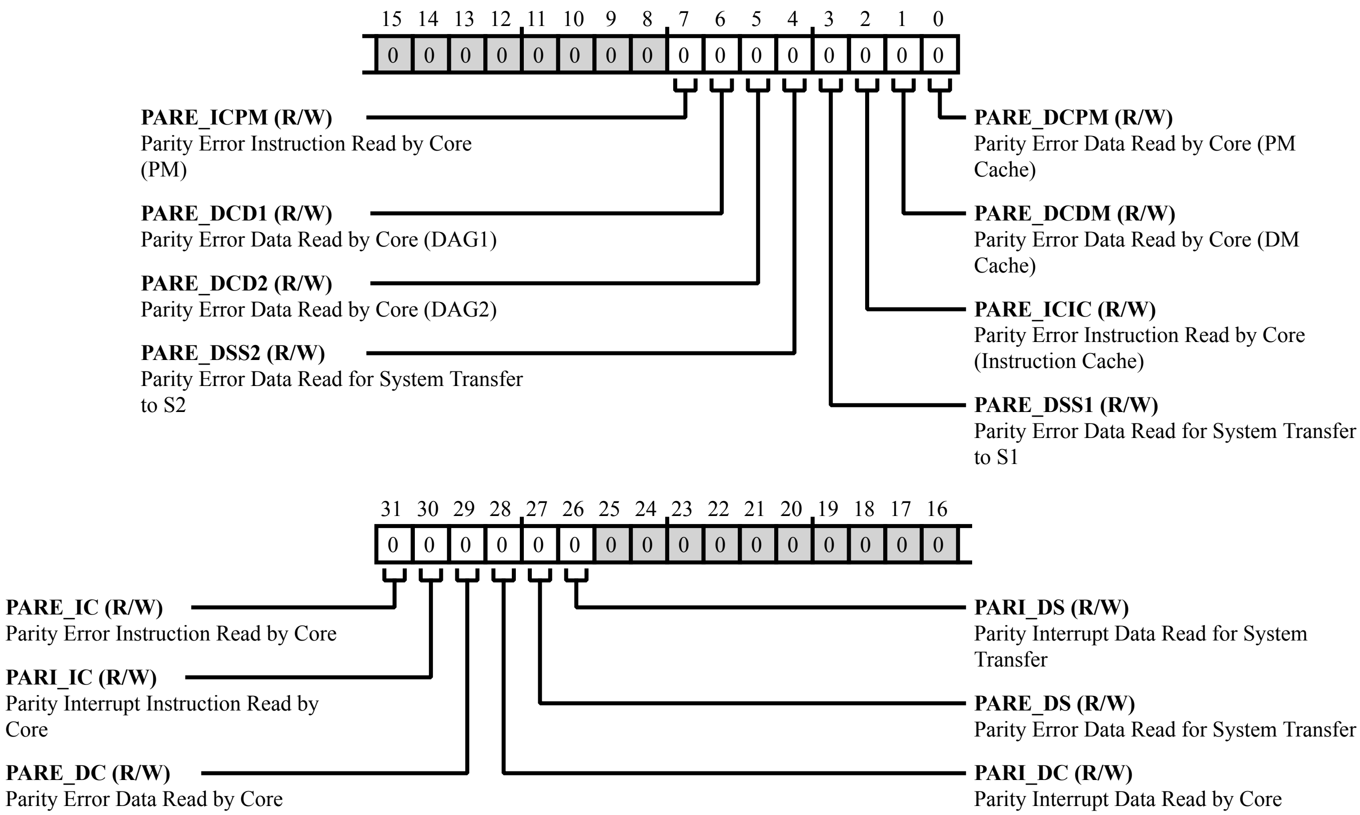

Figure 30-1: CMMR_GPERR_STAT Register Diagram

Table 30-2: CMMR_GPERR_STAT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31 (R/W)	PARE_IC	Parity Error Instruction Read by Core. Write 0x0 to the `CMMR_GPERR_STAT.PARE_IC` bit in the Parity ISR to avoid continuous generation of the Parity Interrupt.
30 (R/W)	PARI_IC	Parity Interrupt Instruction Read by Core. The `CMMR_GPERR_STAT.PARI_IC` bit indicates whether the processor has detected (and latched in the REGF_IRPTL register) a parity interrupt (PARI) on an instruction read of L1 by the core.
29 (R/W)	PARE_DC	Parity Error Data Read by Core. The `CMMR_GPERR_STAT.PARE_DC` bit indicates whether the processor detected a parity error on a data read of L1 by the core.
28 (R/W)	PARI_DC	Parity Interrupt Data Read by Core. The `CMMR_GPERR_STAT.PARI_DC` bit indicates whether the processor has detected (and latched in the REGF_IRPTL register) a parity interrupt (PARI) on a data read of L1 by the core.
27 (R/W)	PARE_DS	Parity Error Data Read for System Transfer. The `CMMR_GPERR_STAT.PARE_DS` bit indicates whether the processor detected a parity error on a data read of L1 for a system transfer.
26 (R/W)	PARI_DS	Parity Interrupt Data Read for System Transfer. The `CMMR_GPERR_STAT.PARI_DS` bit indicates whether the processor has detected (and latched in the REGF_IRPTL register) a parity interrupt (PARI) on a data read of L1 by for a system transfer.
7 (R/W)	PARE_ICPM	Parity Error Instruction Read by Core (PM). The `CMMR_GPERR_STAT.PARE_ICPM` bit indicates whether the processor detected a parity error on an instruction read (fetch) of L1 by the core from program memory.
6 (R/W)	PARE_DCD1	Parity Error Data Read by Core (DAG1). The `CMMR_GPERR_STAT.PARE_DCD1` bit indicates whether the processor detected a parity error on a data read of L1 by the core for a data address generator 1 (DAG1) access.
5 (R/W)	PARE_DCD2	Parity Error Data Read by Core (DAG2). The `CMMR_GPERR_STAT.PARE_DCD2` bit indicates whether the processor detected a parity error on a data read of L1 by the core for a data address generator 2 (DAG2) access.
4 (R/W)	PARE_DSS2	Parity Error Data Read for System Transfer to S2. The `CMMR_GPERR_STAT.PARE_DSS2` bit indicates whether the processor detected a parity error on a data read of L1 for a system transfer through the S2 memory port.

Table 30-2: CMMR_GPERR_STAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
3 (R/W)	PARE_DSS1	Parity Error Data Read for System Transfer to S1. The `CMMR_GPERR_STAT.PARE_DSS1` bit indicates whether the processor detected a parity error on a data read of L1 for a system transfer through the S1 memory port.
2 (R/W)	PARE_ICIC	Parity Error Instruction Read by Core (Instruction Cache). The `CMMR_GPERR_STAT.PARE_ICIC` bit indicates whether the processor detected a parity error on an instruction read of L1 by the core for an instruction cache access.
1 (R/W)	PARE_DCDM	Parity Error Data Read by Core (DM Cache). The `CMMR_GPERR_STAT.PARE_DCDM` bit indicates whether the processor detected a parity error on a data read of L1 by the core for a data memory cache access.
0 (R/W)	PARE_DCPM	Parity Error Data Read by Core (PM Cache). The `CMMR_GPERR_STAT.PARE_DCPM` bit indicates whether the processor detected a parity error on a data read of L1 by the core for a program memory cache access.

PFB No Caching Return 0 End Address Register

The CMMR_PFB_NOCHRT0_END register holds the PFB No caching return0 end address value register

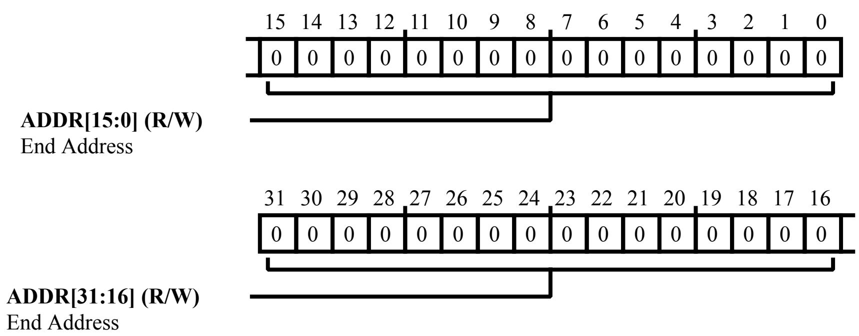

Figure 30-2: CMMR_PFB_NOCHRT0_END Register Diagram

Table 30-3: CMMR_PFB_NOCHRT0_END Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	End Address. The CMMR_PFB_NOCHRT0_END.ADDR bit field contains end address.

PFB No Caching Return 0 Start Address Register

The CMMR_PFB_NOCHRT0_ST register holds the PFB No caching return0 start address value.

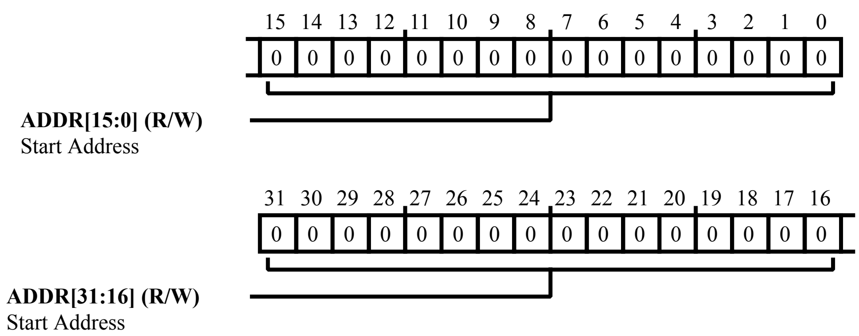

Figure 30-3: CMMR_PFB_NOCHRT0_ST Register Diagram

Table 30-4: CMMR_PFB_NOCHRT0_ST Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Start Address. The CMMR_PFB_NOCHRT0_ST.ADDR bit field contains start address.

Core Global Power Control Register

This register controls various Core and L1-Memory power modes.

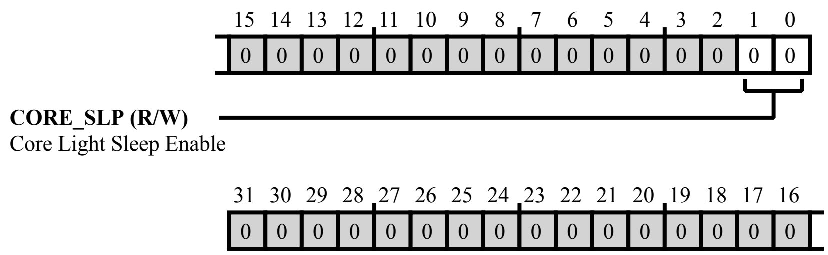

Figure 30-4: CMMR_PWR_GLB_CTL Register Diagram

Table 30-5: CMMR_PWR_GLB_CTL Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
1:0 (R/W)	CORE_SLP	Core Light Sleep Enable.	
		0	No Sleep Mode
		1	Light Sleep
		2	Reserved
		3	Reserved

L1 BANK SLEEP CONTROL

This register controls the Sleep of 4 L1 -Banks

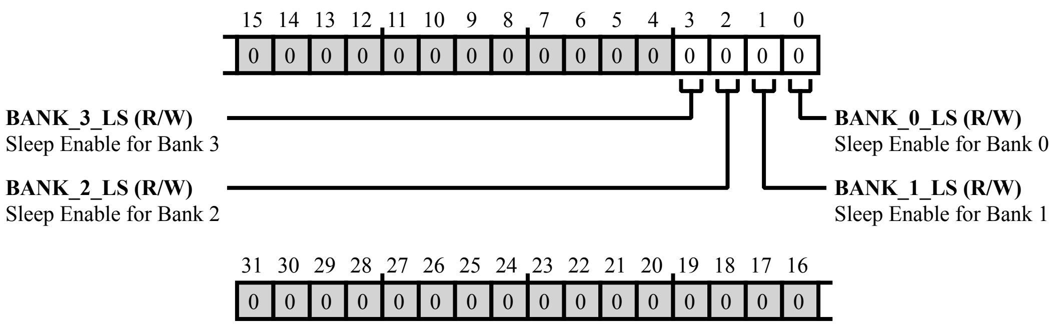

Figure 30-5: CMMR_PWR_L1_LS_CTL Register Diagram

Table 30-6: CMMR_PWR_L1_LS_CTL Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
3 (R/W)	BANK_3_LS	Sleep Enable for Bank 3.
2 (R/W)	BANK_2_LS	Sleep Enable for Bank 2.
1 (R/W)	BANK_1_LS	Sleep Enable for Bank 1.
0 (R/W)	BANK_0_LS	Sleep Enable for Bank 0.

L1 BANK SHUT DOWN CONTROL

This register controls the Shut Down of 4 L1 -Banks

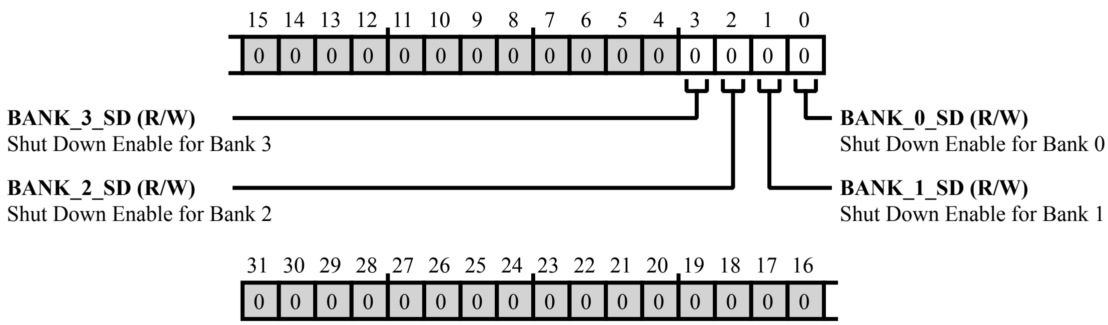

Figure 30-6: CMMR_PWR_L1_SD_CTL Register Diagram

Table 30-7: CMMR_PWR_L1_SD_CTL Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
3 (R/W)	BANK_3_SD	Shut Down Enable for Bank 3.
2 (R/W)	BANK_2_SD	Shut Down Enable for Bank 2.
1 (R/W)	BANK_1_SD	Shut Down Enable for Bank 1.
0 (R/W)	BANK_0_SD	Shut Down Enable for Bank 0.

System Control Register

The CMMR_SYSCTL register as it relates to the processor core configures data memory width memory use and interrupts.

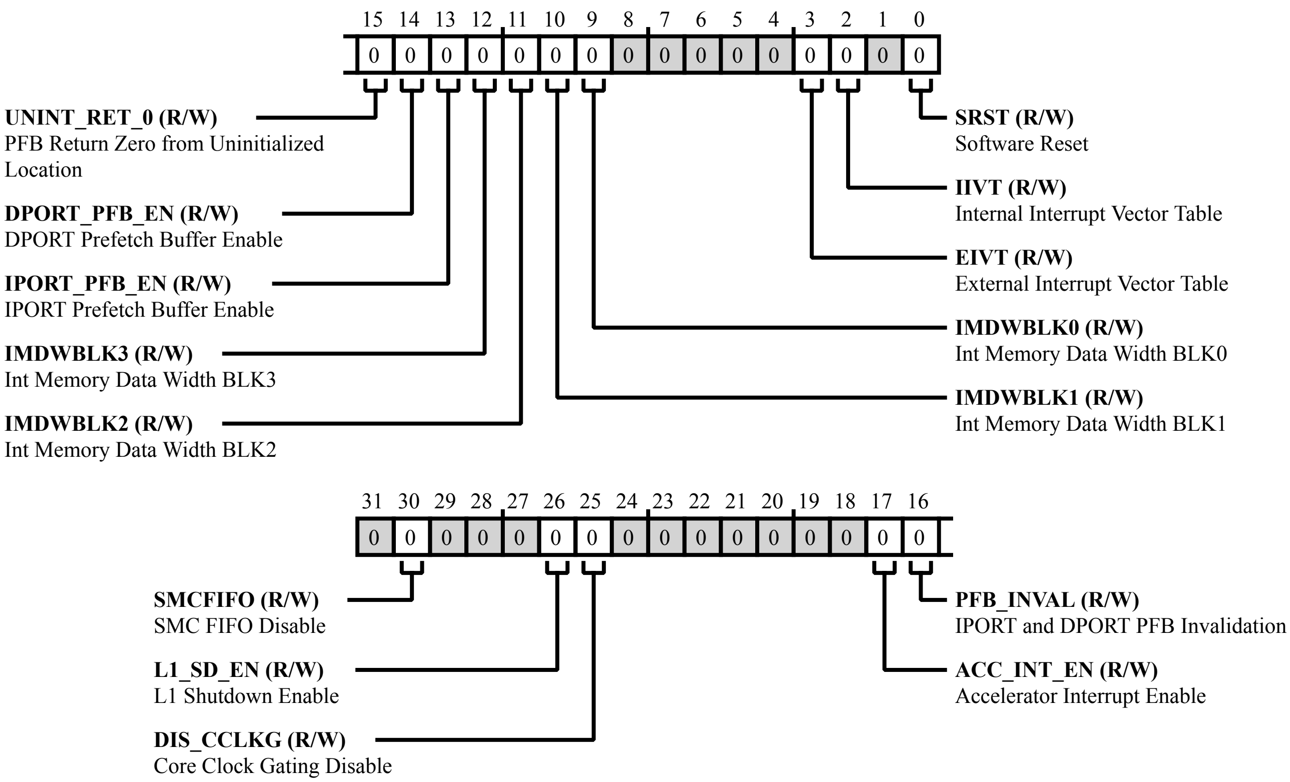

Figure 30-7: CMMR_SYSCTL Register Diagram

Table 30-8: CMMR_SYSCTL Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
30 (R/W)	SMCFIFO	SMC FIFO Disable. The CMMR_SYSCTL.SMCFIFO bit disables the SMC FIFO operation, disabling speculative access on the SMC.
26 (R/W)	L1_SD_EN	L1 Shutdown Enable.
25 (R/W)	DIS_CCLKG	Core Clock Gating Disable.
17 (R/W)	ACC_INT_EN	Accelerator Interrupt Enable.

Table 30-8: CMMR_SYSCTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
16 (R/W)	PFB_INVAL	IPORT and DPORT PFB Invalidation.	
15 (R/W)	UNINT_RET_0	PFB Return Zero from Uninitialized Location.	
14 (R/W)	DPORT_PFB_EN	DPORT Prefetch Buffer Enable.	
13 (R/W)	IPORT_PFB_EN	IPORT Prefetch Buffer Enable.	
12 (R/W)	IMDWBLK3	Int Memory Data Width BLK3. The `CMMR_SYSCTL.IMDWBLK3` bits select the internal memory data width for for block 3.	
		0	32-bit wide access
		1	48-bit wide access
11 (R/W)	IMDWBLK2	Int Memory Data Width BLK2. The `CMMR_SYSCTL.IMDWBLK2` bits select the internal memory data width for for block 2.	
		0	32-bit wide access
		1	48-bit wide access
10 (R/W)	IMDWBLK1	Int Memory Data Width BLK1. The `CMMR_SYSCTL.IMDWBLK1` bits select the internal memory data width for for block 1.	
		0	32-bit wide access
		1	48-bit wide access
9 (R/W)	IMDWBLK0	Int Memory Data Width BLK0. The `CMMR_SYSCTL.IMDWBLK0` bits select the internal memory data width for for block 0.	
		0	32-bit wide access
		1	48-bit wide access
3 (R/W)	EIVT	External Interrupt Vector Table. The `CMMR_SYSCTL.EIVT` bit when set, maps the IVT to the external DDR_Address 0x400000.	

Table 30-8: CMMR_SYSCTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
2 (R/W)	IIVT	Internal Interrupt Vector Table. The CMMR_SYSCTL.IIVT bit when set maps the IVT to the internal memory Address 0x900000. On reset, this bit is cleared and the IVT is mapped to L2CTL ROM1 boot memory address 0x500000.
0 (R/W)	SRST	Software Reset. The CMMR_SYSCTL.SRST bit initiates a software reset. This bit has an effect latency of 1 cycle, so the next instruction after a SYSCTL write for a soft reset will also be executed.

31 SHARC-PLUS SHBTB Register Descriptions

Branch Target Buffer (SHBTB) contains the following registers.

Table 31-1: SHARC-PLUS SHBTB Register List

Name	Description
SHBTB_CFG	Configuration Register
SHBTB_LOCK_END	Lock Range End Register
SHBTB_LOCK_START	Lock Range Start Register

Configuration Register

The SHBTB_CFG register enables the BTB and configures BTB features, such as range-based locking and return optimization.

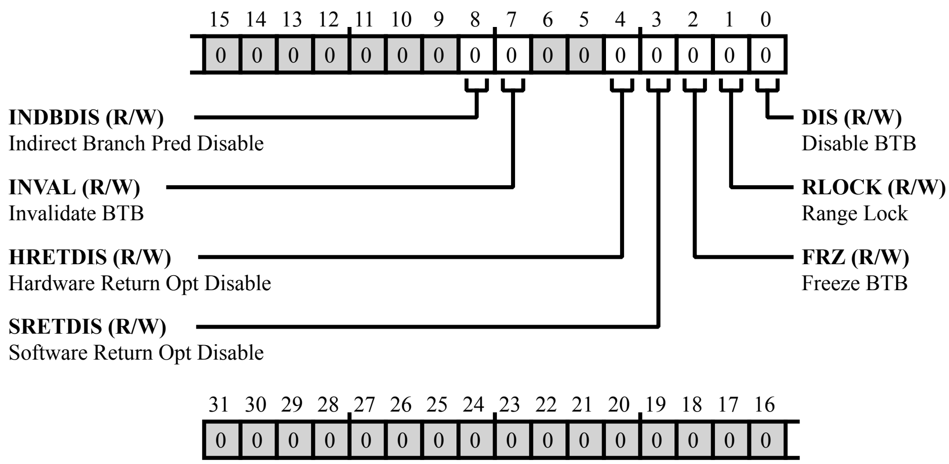

Figure 31-1: SHBTB_CFG Register Diagram

Table 31-2: SHBTB_CFG Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
8 (R/W)	INDBDIS	Indirect Branch Pred Disable. The SHBTB_CFG.INDBDIS bit disables indirect branch prediction for the BTB. Setting this bit disables the predictions of any indirect branch.
7 (R/W)	INVAL	Invalidate BTB. The SHBTB_CFG.INVAL bit invalidates BTB contents. Setting this bit invalidates the BTB memory.
4 (R/W)	HRETDIS	Hardware Return Opt Disable. The SHBTB_CFG.HRETDIS bit disables hardware return-from-subroutine (RTS) optimization for the BTB. If this bit is cleared (=0), the target address for an RTS is brought from the top of the PC stack. If this bit is set (=1), the target address is brought from BTB memory.
3 (R/W)	SRETDIS	Software Return Opt Disable. The SHBTB_CFG.SRETDIS bit disables software return-from-subroutine (m14,i12) optimization for the BTB. If this bit is cleared (=0), the target address value is brought from I12 register and is added with 1 (instead of M14). If this bit is set (=1), the target address is the address stored in the BTB memory during the last update of the branch.
2 (R/W)	FRZ	Freeze BTB. The SHBTB_CFG.FRZ bit freezes BTB contents. All of the valid and invalid locations remain unchanged.

Table 31-2: SHBTB_CFG Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
1 (R/W)	RLOCK	Range Lock. The SHBTB_CFG.RLOCK bit enables range-based lock operations of the BTB. Setting this bit validates the values in the SHBTB_LOCK_START register and the SHBTB_LOCK_END register. According to this range, the address locations in BTB memory are locked. Program the SHBTB_LOCK_START register and the SHBTB_LOCK_END register before enabling ranged-based locking.
0 (R/W)	DIS	Disable BTB. The SHBTB_CFG.DIS bit disables BTB operation.

Lock Range End Register

The SHBTB_LOCK_END register indicates the last address to lock a range of memory address in BTB memory. This address is valid only after the SHBTB_CFG.RLOCK bit (range-based locking) is enabled. The SHBTB_LOCK_END value should not be less than the value in the SHBTB_LOCK_START register.

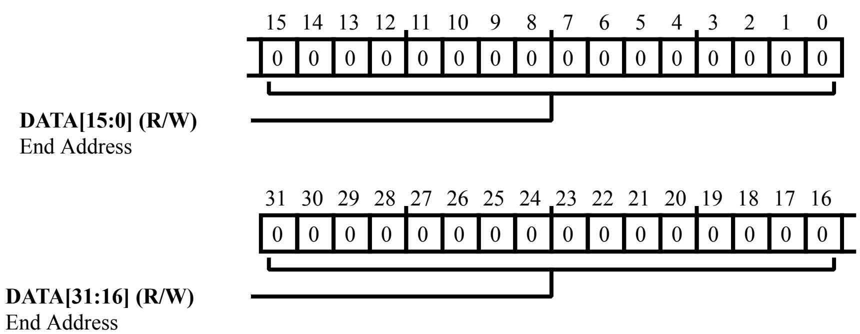

Figure 31-2: SHBTB_LOCK_END Register Diagram

Table 31-3: SHBTB_LOCK_END Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	End Address. The SHBTB_LOCK_END.DATA bit field contains address data.

Lock Range Start Register

The SHBTB_LOCK_START register indicates the initial address to lock a range of address in BTB memory. This address is valid only after the SHBTB_CFG.RLOCK bit (range-based locking mode) is enabled.

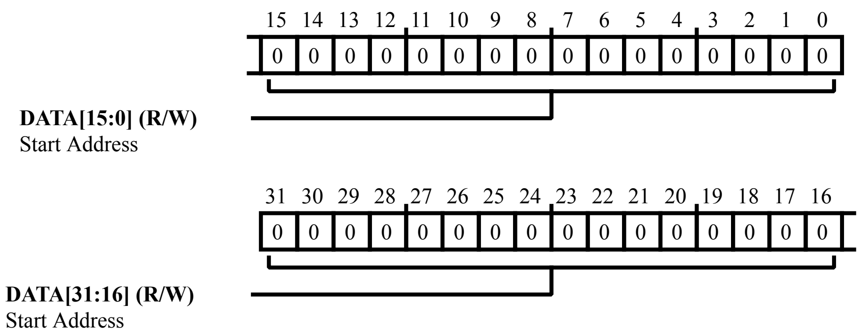

Figure 31-3: SHBTB_LOCK_START Register Diagram

Table 31-4: SHBTB_LOCK_START Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Start Address. The SHBTB_LOCK_START.DATA bit field contains address data.

32 SHARC-PLUS SHDBG Register Descriptions

Debug Core (SHDBG) contains the following registers.

Table 32-1: SHARC-PLUS SHDBG Register List

Name	Description
SHDBG_BRKCTL	Break Control Register
SHDBG_BRKSTAT	Break Status Register
SHDBG_CORE_ID	Core ID Register
SHDBG_D1ADDR	Decode 1 Stage Address Register
SHDBG_D2ADDR	Decode 2 Stage Address Register
SHDBG_DBGREG_ILLOP	Illegal Opcode Detected Register
SHDBG_DMA1E	DM Data Address 1 End Register
SHDBG_DMA1S	DM Data Address 1 Start Register
SHDBG_DMA2E	DM Data Address 2 End Register
SHDBG_DMA2S	DM Data Address 2 Start Register
SHDBG_E2ADDR	Execute 2 Stage Address Register
SHDBG_EMUN	Emulator Number (BP Hits) Register
SHDBG_F1ADDR	Fetch 1 Stage Address Register
SHDBG_F2ADDR	Fetch 2 Stage Address Register
SHDBG_F3ADDR	Fetch 3 Stage Address Register
SHDBG_F4ADDR	Fetch 4 Stage Address Register
SHDBG_M1ADDR	Memory 1 Stage Address Register
SHDBG_M2ADDR	Memory 2 Stage Address Register
SHDBG_M3ADDR	Memory 3 Stage Address Register
SHDBG_M4ADDR	Memory 4 Stage Address Register
SHDBG_OSPID	O/S Processor ID Register
SHDBG_PMDAE	PM Data Address 1 End Register

Table 32-1: SHARC-PLUS SHDBG Register List (Continued)

Name	Description
SHDBG_PMDAS	PM Data Address 1 Start Register
SHDBG_PSA1E	Program Sequence Address 1 End Register
SHDBG_PSA1S	Program Sequence Address 1 Start Register
SHDBG_PSA2E	Program Sequence Address 2 End Register
SHDBG_PSA2S	Program Sequence Address 2 Start Register
SHDBG_PSA3E	Program Sequence Address 3 End Register
SHDBG_PSA3S	Program Sequence Address 3 Start Register
SHDBG_PSA4E	Program Sequence Address 4 End Register
SHDBG_PSA4S	Program Sequence Address 4 Start Register
SHDBG_REVID	ID Code Register
SHDBG_SECI_ID	SEC Interrupt ID Register

Break Control Register

The SHDBG_BRKCTL register enables (=1) or disables (=0 default) breakpoint mode.

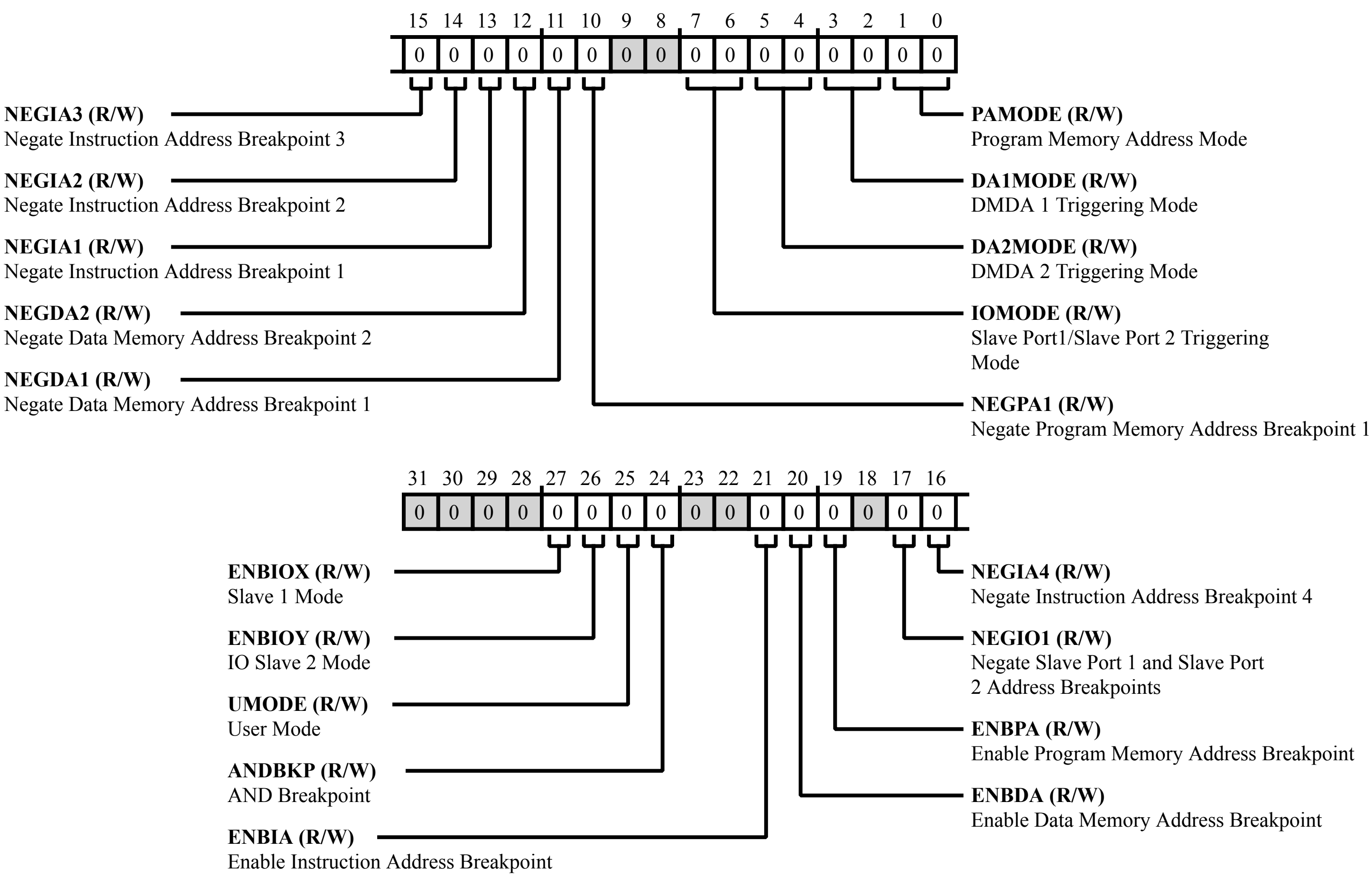

Figure 32-1: SHDBG_BRKCTL Register Diagram

Table 32-2: SHDBG_BRKCTL Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
27 (R/W)	ENBIOX	Slave 1 Mode. The SHDBG_BRKCTL.ENBIOX bit configures the slave 1 address breakpoint.
26 (R/W)	ENBIOY	IO Slave 2 Mode. The SHDBG_BRKCTL.ENBIOY bit configures slave 2 address breakpoint.
25 (R/W)	UMODE	User Mode. The SHDBG_BRKCTL.UMODE bit configures user mode.

Table 32-2: SHDBG_BRKCTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
24 (R/W)	ANDBKP	AND Breakpoint. The SHDBG_BRKCTL.ANDBKP bit ANDs the composite breakpoints.
21 (R/W)	ENBIA	Enable Instruction Address Breakpoint. The SHDBG_BRKCTL.ENBIA bit configures
20 (R/W)	ENBDA	Enable Data Memory Address Breakpoint. The SHDBG_BRKCTL.ENBDA bit enables a data memory address breakpoint.
19 (R/W)	ENBPA	Enable Program Memory Address Breakpoint. The SHDBG_BRKCTL.ENBPA bit enables a program memory address breakpoint.
17 (R/W)	NEGIO1	Negate Slave Port 1 and Slave Port 2 Address Breakpoints. The SHDBG_BRKCTL.NEGIO1 bit negates the slave port 1 and slave port 2 address breakpoints.
16 (R/W)	NEGIA4	Negate Instruction Address Breakpoint 4. The SHDBG_BRKCTL.NEGIA4 bit negates instruction breakpoint 4.
15 (R/W)	NEGIA3	Negate Instruction Address Breakpoint 3. The SHDBG_BRKCTL.NEGIA3 bit negates instruction breakpoint 3.
14 (R/W)	NEGIA2	Negate Instruction Address Breakpoint 2. The SHDBG_BRKCTL.NEGIA2 bit negates instruction breakpoint 2.
13 (R/W)	NEGIA1	Negate Instruction Address Breakpoint 1. The SHDBG_BRKCTL.NEGIA1 bit negates instruction breakpoint 1.
12 (R/W)	NEGDA2	Negate Data Memory Address Breakpoint 2. The SHDBG_BRKCTL.NEGDA2 bit negates data memory address breakpoint 2.
11 (R/W)	NEGDA1	Negate Data Memory Address Breakpoint 1. The SHDBG_BRKCTL.NEGDA1 bit negates data memory address breakpoint 1.
10 (R/W)	NEGPA1	Negate Program Memory Address Breakpoint 1. The SHDBG_BRKCTL.NEGPA1 bit negates program memory address breakpoint 1.
7:6 (R/W)	IOMODE	Slave Port1/Slave Port 2 Triggering Mode. The SHDBG_BRKCTL.IOMODE bit field configures slave port 1 and slave port 2 triggering mode.
5:4 (R/W)	DA2MODE	DMDA 2 Triggering Mode. The SHDBG_BRKCTL.DA2MODE bit field configures DMDA 2 breakpoint triggering mode.

Table 32-2: SHDBG_BRKCTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
3:2 (R/W)	DA1MODE	DMDA 1 Triggering Mode. The SHDBG_BRKCTL.DA1MODE bit field configures DMDA 1 breakpoint triggering mode.
1:0 (R/W)	PAMODE	Program Memory Address Mode. The SHDBG_BRKCTL.PAMODE bit field configures PMDA breakpoint triggering mode.

Break Status Register

The SHDBG_BRKSTAT register provides information about breakpoint hits.

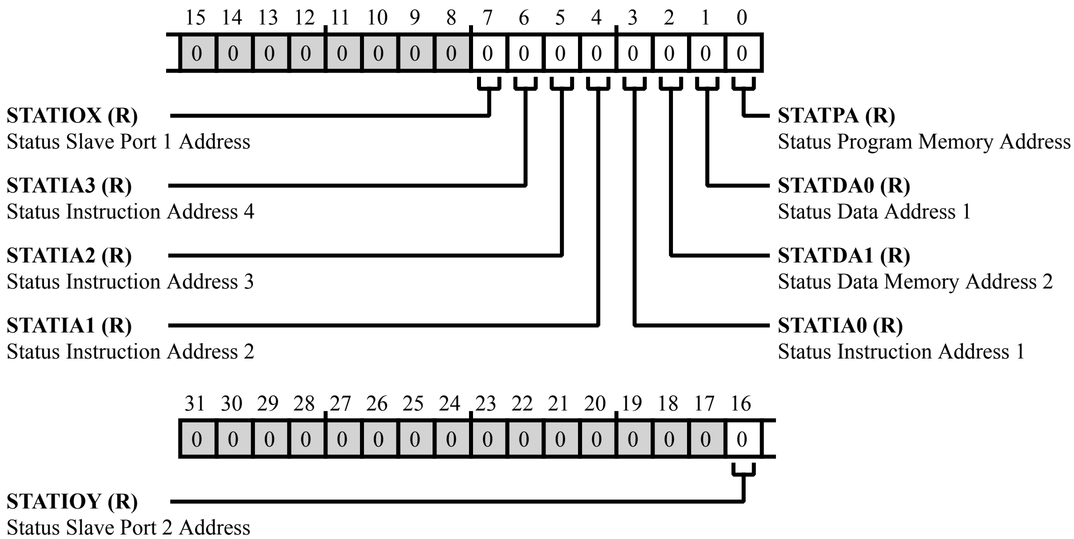

Figure 32-2: SHDBG_BRKSTAT Register Diagram

Table 32-3: SHDBG_BRKSTAT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
16 (R/NW)	STATIOY	Status Slave Port 2 Address. The SHDBG_BRKSTAT.STATIOY bit indicates a slave port 2 address breakpoint hit.
7 (R/NW)	STATIOX	Status Slave Port 1 Address. The SHDBG_BRKSTAT.STATIOX bit indicates a slave port 1 address breakpoint hit.
6 (R/NW)	STATIA3	Status Instruction Address 4. The SHDBG_BRKSTAT.STATIA3 bit indicates instruction address breakpoint hit #4.
5 (R/NW)	STATIA2	Status Instruction Address 3. The SHDBG_BRKSTAT.STATIA2 bit indicates instruction address breakpoint hit #3.
4 (R/NW)	STATIA1	Status Instruction Address 2. The SHDBG_BRKSTAT.STATIA1 bit indicates instruction address breakpoint hit #2.

Table 32-3: SHDBG_BRKSTAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
3 (R/NW)	STATIA0	Status Instruction Address 1. The SHDBG_BRKSTAT.STATIA0 bit indicates instruction address breakpoint hit #1.
2 (R/NW)	STATDA1	Status Data Memory Address 2. The SHDBG_BRKSTAT.STATDA1 bit indicates a data memory address breakpoint hit #2.
1 (R/NW)	STATDA0	Status Data Address 1. The SHDBG_BRKSTAT.STATDA0 bit indicates the data memory address breakpoint hit #1.
0 (R/NW)	STATPA	Status Program Memory Address. The SHDBG_BRKSTAT.STATPA bit indicates a program memory data address breakpoint hit.

Core ID Register

The value in the SHDBG_CORE_ID register indicates the SHARC Core ID. In the ADSP-SC5xx/ADSP-215xx processors:

0 = SHARC+ core 0; applicable to ADSP-2156x processors

1 = SHARC+ core 1

2 = SHARC+ core 2

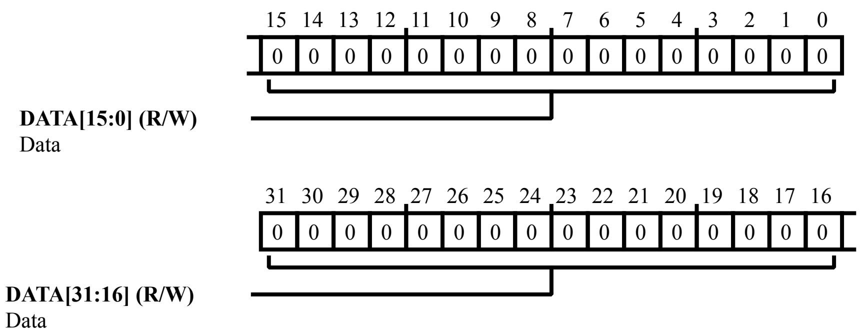

Figure 32-3: SHDBG_CORE_ID Register Diagram

Table 32-4: SHDBG_CORE_ID Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The SHDBG_CORE_ID.DATA bit fields contains the available core IDs.

Decode 1 Stage Address Register

The SHDBG_D1ADDR register holds the address of the pipeline Decode1 stage.

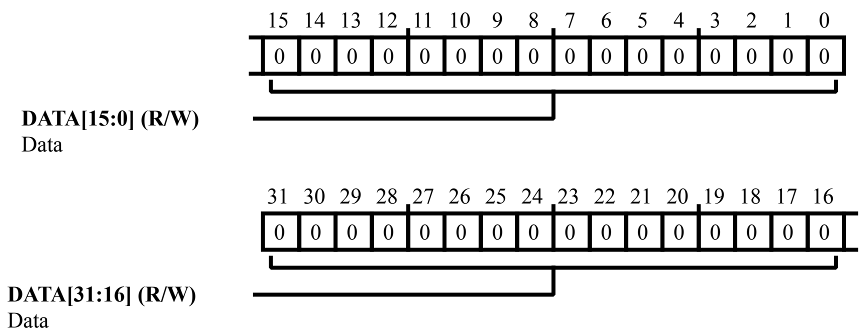

Figure 32-4: SHDBG_D1ADDR Register Diagram

Table 32-5: SHDBG_D1ADDR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data.

Decode 2 Stage Address Register

The SHDBG_D2ADDR register holds the address of the pipeline Decode2 stage.

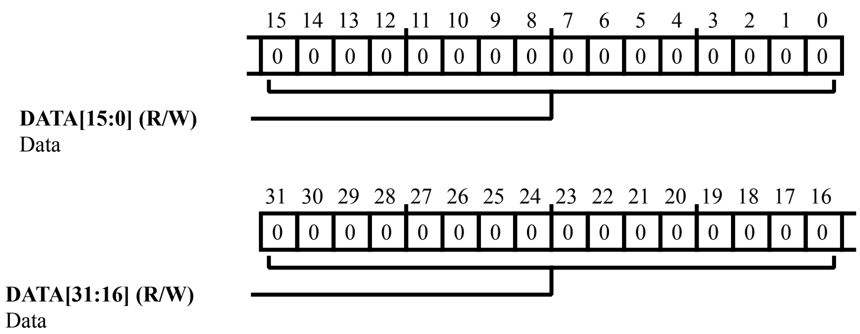

Figure 32-5: SHDBG_D2ADDR Register Diagram

Table 32-6: SHDBG_D2ADDR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data.

Illegal Opcode Detected Register

The SHDBG_DBGREG_ILLOP register holds the address for which an illegal opcode interrupt is generated. The SHDBG_DBGREG_ILLOP register also contains a bit to indicate whether the ILLOPI interrupt was generated when there was an un-interruptible cycle active or inactive.

A dummy write to this register only clears its content.

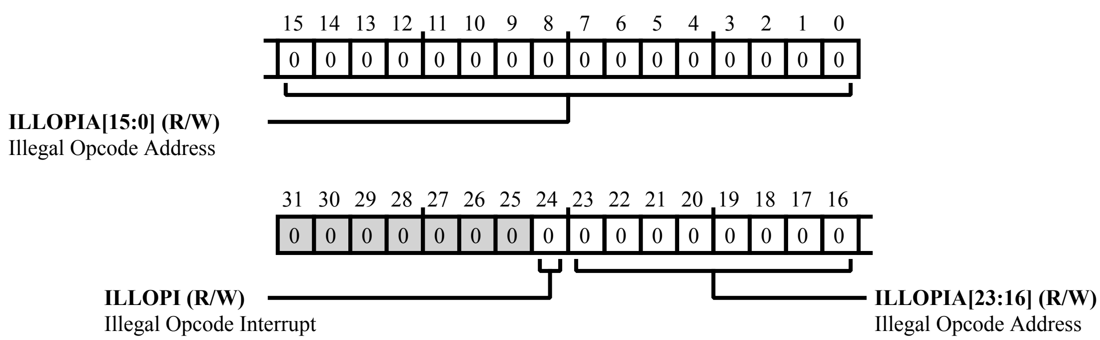

Figure 32-6: SHDBG_DBGREG_ILLOP Register Diagram

Table 32-7: SHDBG_DBGREG_ILLOP Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
24 (R/W)	ILLOPI	Illegal Opcode Interrupt. The SHDBG_DBGREG_ILLOP.ILLOPI bit is set if the ILLOPI fires when un-interruptible cycle is active. In this case simple RTI from the ISR does not ensure the correct functioning of the program.
23:0 (R/W)	ILLOPIA	Illegal Opcode Address. The SHDBG_DBGREG_ILLOP.ILLOPIA bit field stores the address for which the ILLOPI interrupt was generated.

DM Data Address 1 End Register

The SHDBG_DMA1E register holds the Data memory address end #1.

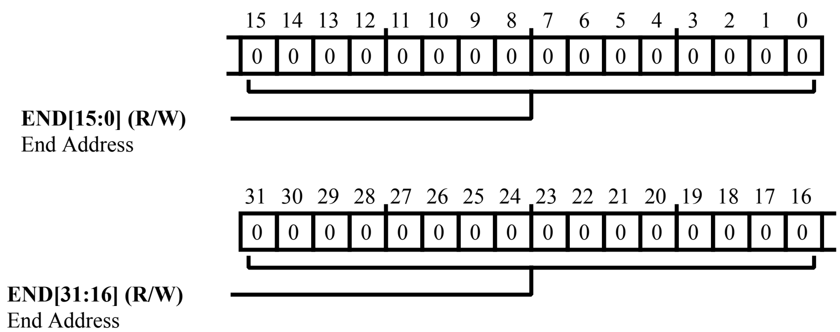

Figure 32-7: SHDBG_DMA1E Register Diagram

Table 32-8: SHDBG_DMA1E Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	END	End Address. The SHDBG_DMA1E.END bit field holds the Data memory end address #1.

DM Data Address 1 Start Register

The SHDBG_DMA1S register holds the Data memory address start #1.

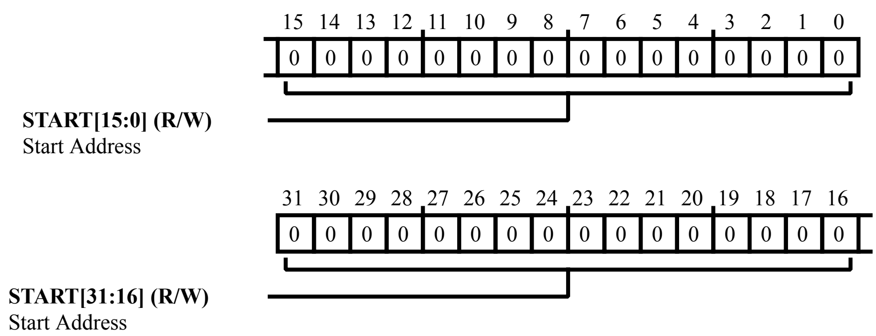

Figure 32-8: SHDBG_DMA1S Register Diagram

Table 32-9: SHDBG_DMA1S Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	START	Start Address. The SHDBG_DMA1S.START bit field holds the Data memory start address #1.

DM Data Address 2 End Register

The SHDBG_DMA2E register holds the Data memory address end #2.

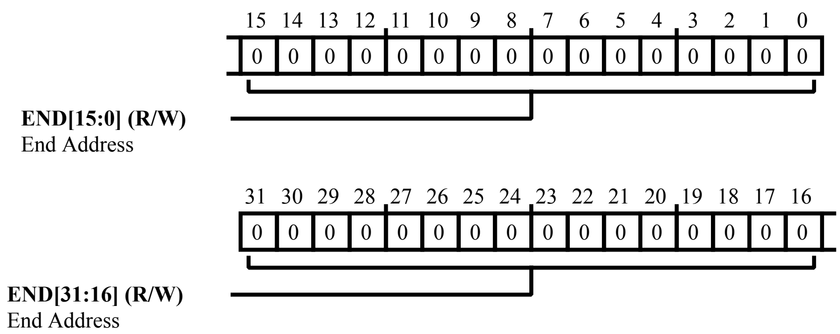

Figure 32-9: SHDBG_DMA2E Register Diagram

Table 32-10: SHDBG_DMA2E Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	END	End Address. The SHDBG_DMA2E.END bit field holds the data memory data end address breakpoint #1.

DM Data Address 2 Start Register

The SHDBG_DMA2S register holds the Data memory address start #2.

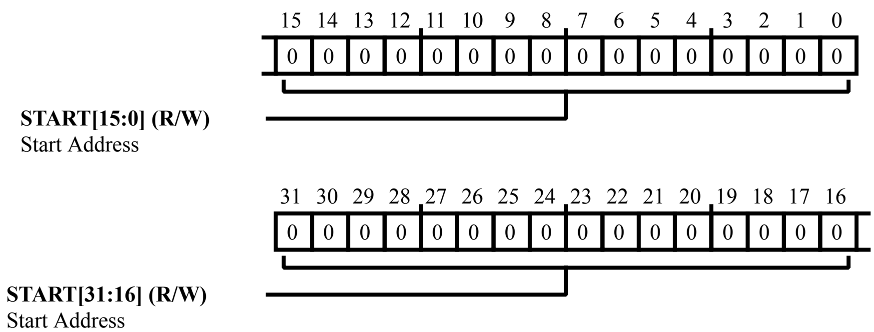

Figure 32-10: SHDBG_DMA2S Register Diagram

Table 32-11: SHDBG_DMA2S Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	START	Start Address. The SHDBG_DMA2S.START bit field holds the Data memory start address #1.

Execute 2 Stage Address Register

The SHDBG_E2ADDR register holds the address of the pipeline Execute2 stage.

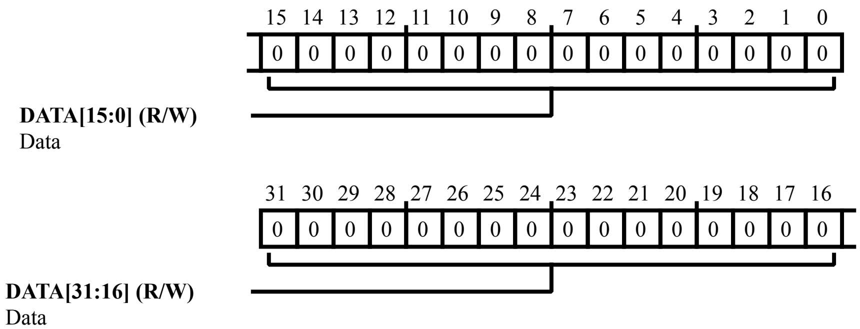

Figure 32-11: SHDBG_E2ADDR Register Diagram

Table 32-12: SHDBG_E2ADDR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data.

Emulator Number (BP Hits) Register

The SHDBG_EMUN register provides the number of emulator breakpoint hits.

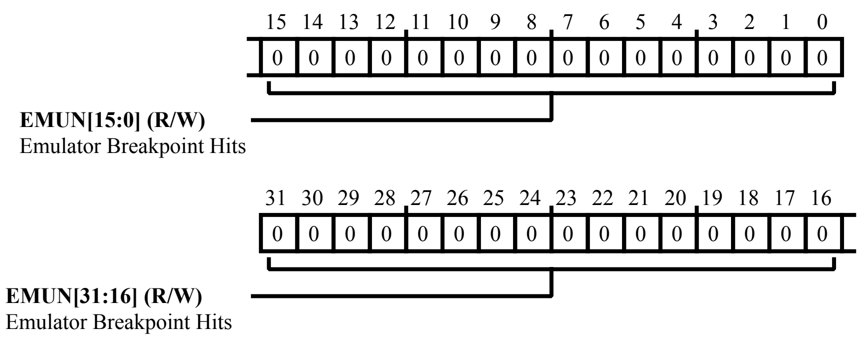

Figure 32-12: SHDBG_EMUN Register Diagram

Table 32-13: SHDBG_EMUN Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	EMUN	Emulator Breakpoint Hits. The SHDBG_EMUN.EMUN bit field provides the number of emulator breakpoint hits.

Fetch 1 Stage Address Register

The SHDBG_F1ADDR register holds the address of the pipeline Fetch1 stage.

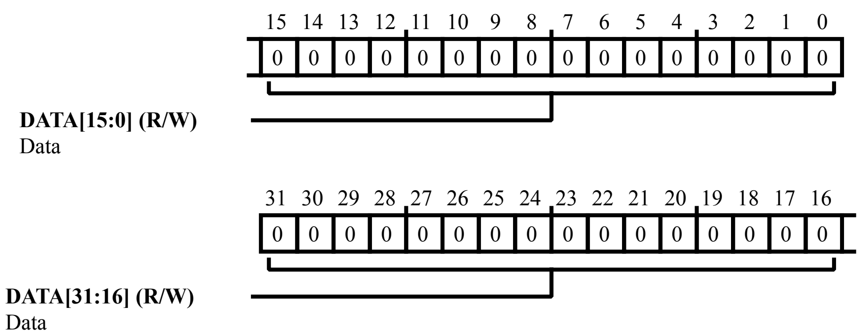

Figure 32-13: SHDBG_F1ADDR Register Diagram

Table 32-14: SHDBG_F1ADDR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data.

Fetch 2 Stage Address Register

The SHDBG_F2ADDR register holds the address of the pipeline Fetch2 stage.

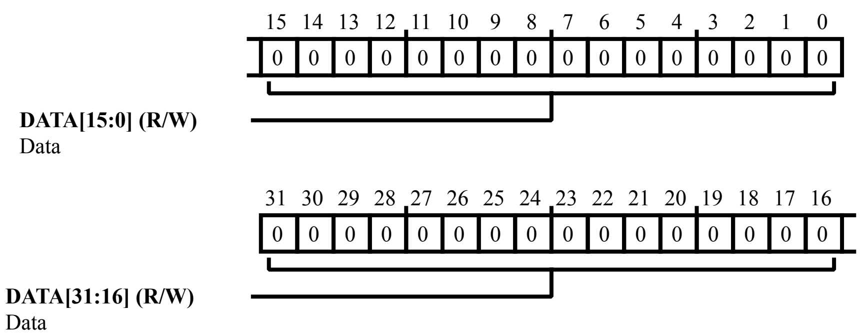

Figure 32-14: SHDBG_F2ADDR Register Diagram

Table 32-15: SHDBG_F2ADDR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data.

Fetch 3 Stage Address Register

The SHDBG_F3ADDR register holds the address of the pipeline Fetch3 stage.

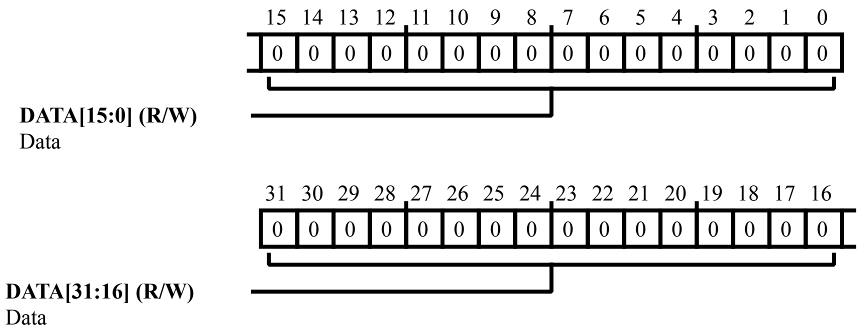

Figure 32-15: SHDBG_F3ADDR Register Diagram

Table 32-16: SHDBG_F3ADDR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data.

Fetch 4 Stage Address Register

The SHDBG_F4ADDR register holds the address of the pipeline Fetch4 stage.

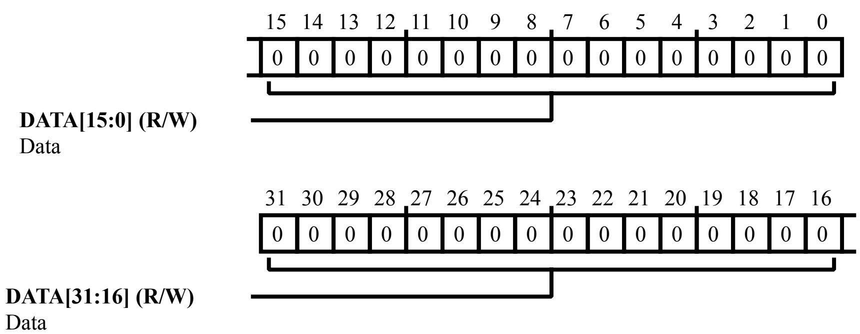

Figure 32-16: SHDBG_F4ADDR Register Diagram

Table 32-17: SHDBG_F4ADDR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data.

Memory 1 Stage Address Register

The SHDBG_M1ADDR register holds the address of the pipeline Memory1 stage.

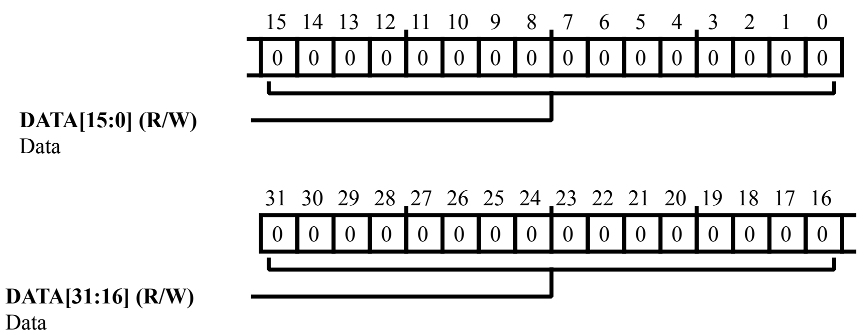

Figure 32-17: SHDBG_M1ADDR Register Diagram

Table 32-18: SHDBG_M1ADDR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data.

Memory 2 Stage Address Register

The SHDBG_M2ADDR register holds the address of the pipeline Memory2 stage.

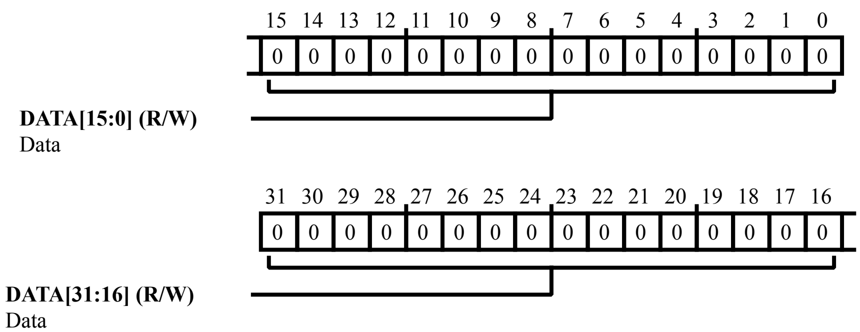

Figure 32-18: SHDBG_M2ADDR Register Diagram

Table 32-19: SHDBG_M2ADDR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data.

Memory 3 Stage Address Register

The SHDBG_M3ADDR register holds the address of the pipeline Memory3 stage.

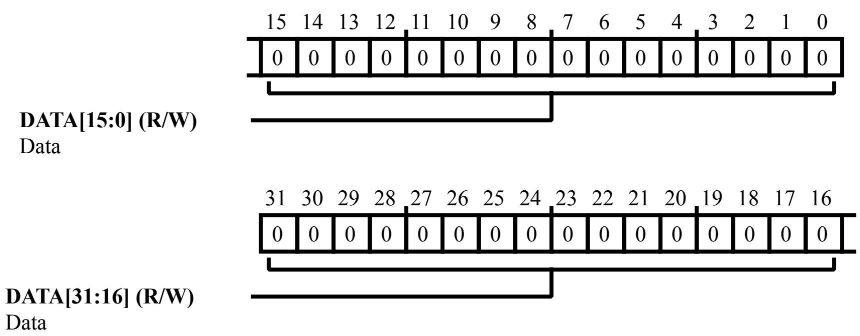

Figure 32-19: SHDBG_M3ADDR Register Diagram

Table 32-20: SHDBG_M3ADDR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data.

Memory 4 Stage Address Register

The SHDBG_M4ADDR register holds the address of the pipeline Memory4 stage.

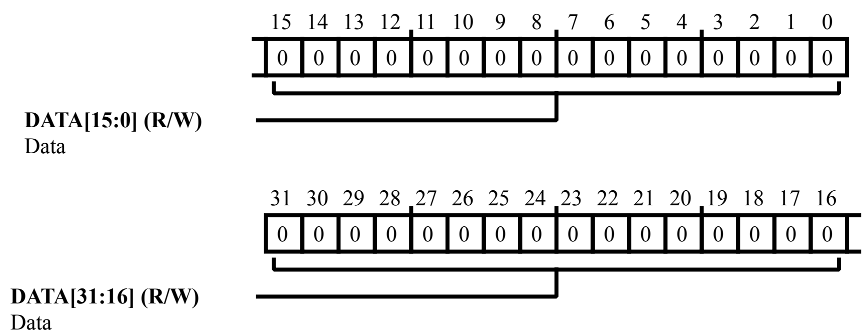

Figure 32-20: SHDBG_M4ADDR Register Diagram

Table 32-21: SHDBG_M4ADDR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data.

O/S Processor ID Register

In a multi-tasking operating system, the operating system assigns a thread number for each thread or process. The OS must write the process ID (thread ID) of the task into the SHDBG_OSPID register at the beginning of the task's slot.

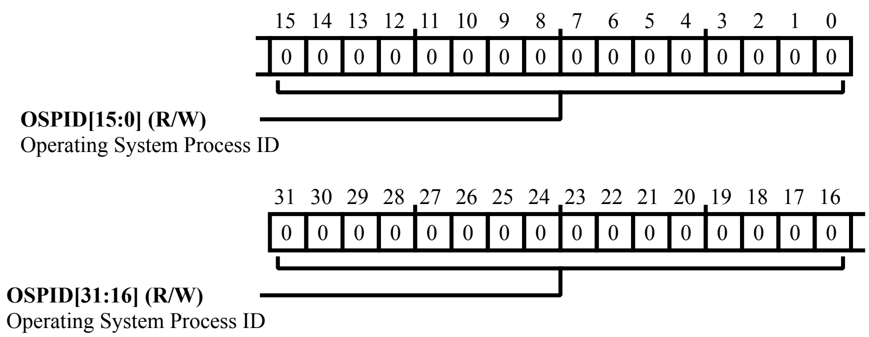

Figure 32-21: SHDBG_OSPID Register Diagram

Table 32-22: SHDBG_OSPID Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	OSPID	Operating System Process ID. The SHDBG_OSPID.OSPID bit field is the operating system process/thread ID value.

PM Data Address 1 End Register

The SHDBG_PMDAE register holds the Program memory data address end #1.

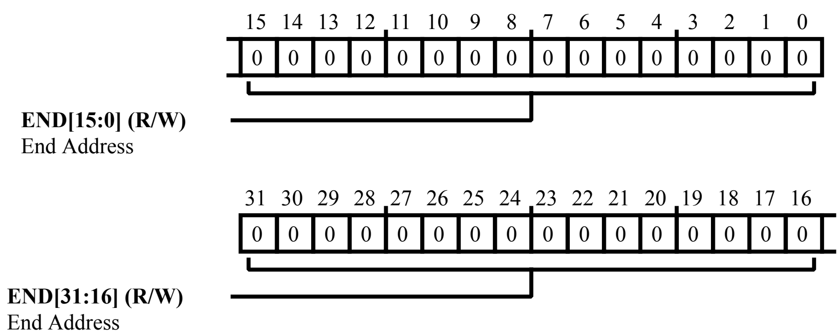

Figure 32-22: SHDBG_PMDAE Register Diagram

Table 32-23: SHDBG_PMDAE Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	END	End Address. The SHDBG_PMDAE.END bit field holds the program memory data end address 1 breakpoint.

PM Data Address 1 Start Register

The SHDBG_PMDAS register holds the Program memory data address start #1.

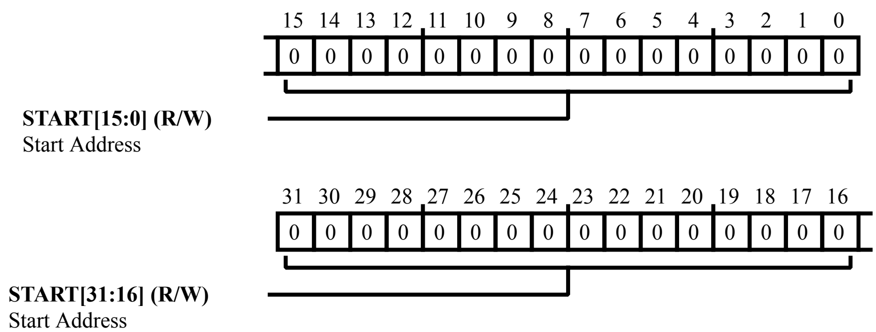

Figure 32-23: SHDBG_PMDAS Register Diagram

Table 32-24: SHDBG_PMDAS Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	START	Start Address. The SHDBG_PMDAS.START bit field holds the program memory data start address 1 breakpoint.

Program Sequence Address 1 End Register

The SHDBG_PSA1E register holds the Instruction address end #1.

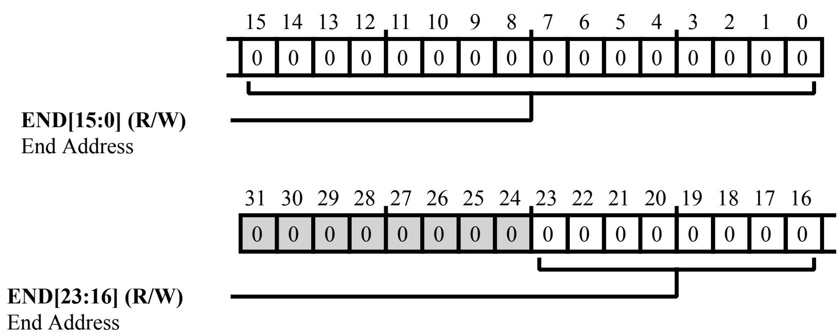

Figure 32-24: SHDBG_PSA1E Register Diagram

Table 32-25: SHDBG_PSA1E Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
23:0 (R/W)	END	End Address. The SHDBG_PSA1E.END bit field provides the instruction breakpoint #1 end address.

Program Sequence Address 1 Start Register

The SHDBG_PSA1S register holds the Instruction address start #1.

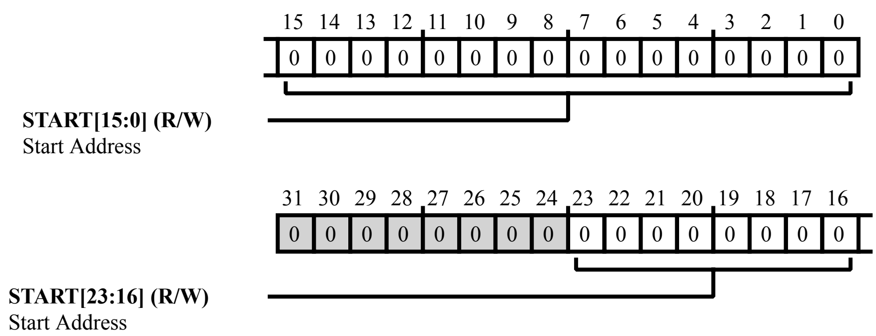

Figure 32-25: SHDBG_PSA1S Register Diagram

Table 32-26: SHDBG_PSA1S Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
23:0 (R/W)	START	Start Address. The SHDBG_PSA1S.START bit field provides the instruction breakpoint #1 start address.

Program Sequence Address 2 End Register

The SHDBG_PSA2E register holds the Instruction address end #2.

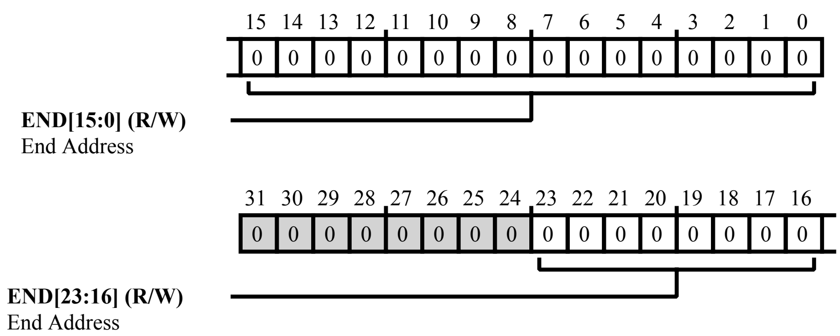

Figure 32-26: SHDBG_PSA2E Register Diagram

Table 32-27: SHDBG_PSA2E Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
23:0 (R/W)	END	End Address. The SHDBG_PSA2E.END bit field provides the instruction breakpoint #1 end address.

Program Sequence Address 2 Start Register

The SHDBG_PSA2S register holds the Instruction address start #2.

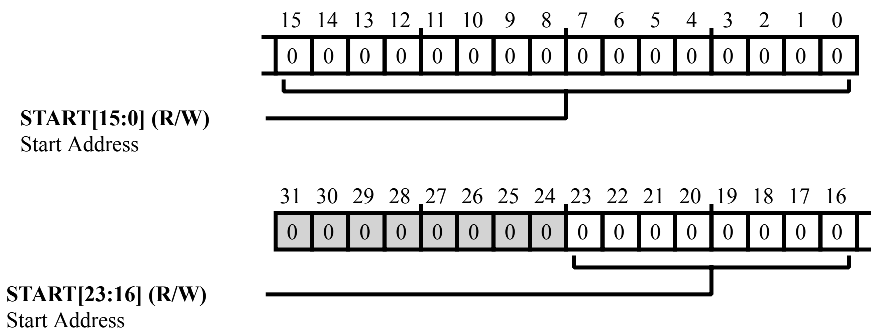

Figure 32-27: SHDBG_PSA2S Register Diagram

Table 32-28: SHDBG_PSA2S Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
23:0 (R/W)	START	Start Address. The SHDBG_PSA2S.START bit field provides the instruction breakpoint #1 start address.

Program Sequence Address 3 End Register

The SHDBG_PSA3E register holds the Instruction address end #3.

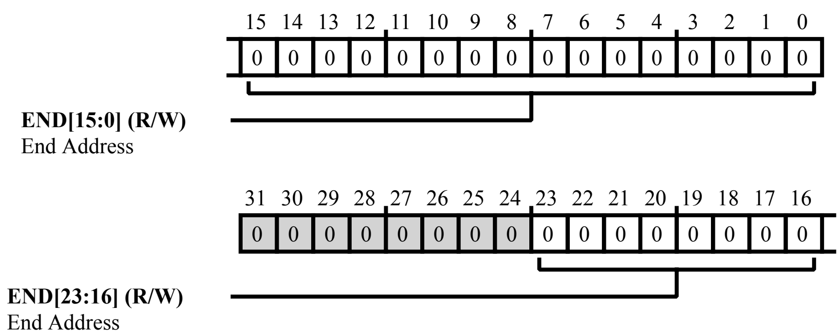

Figure 32-28: SHDBG_PSA3E Register Diagram

Table 32-29: SHDBG_PSA3E Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
23:0 (R/W)	END	End Address. The SHDBG_PSA3E.END bit field provides the instruction breakpoint #1 end address.

Program Sequence Address 3 Start Register

The SHDBG_PSA3S register holds the Instruction address start #3.

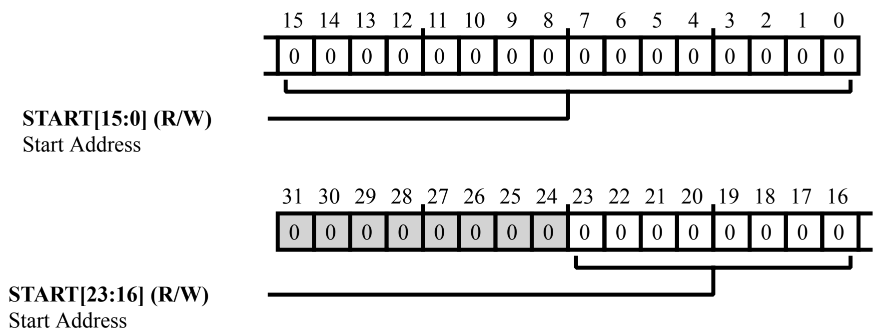

Figure 32-29: SHDBG_PSA3S Register Diagram

Table 32-30: SHDBG_PSA3S Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
23:0 (R/W)	START	Start Address. The SHDBG_PSA3S.START bit field provides the instruction breakpoint #1 start address.

Program Sequence Address 4 End Register

The SHDBG_PSA4E register holds the Instruction address end #4.

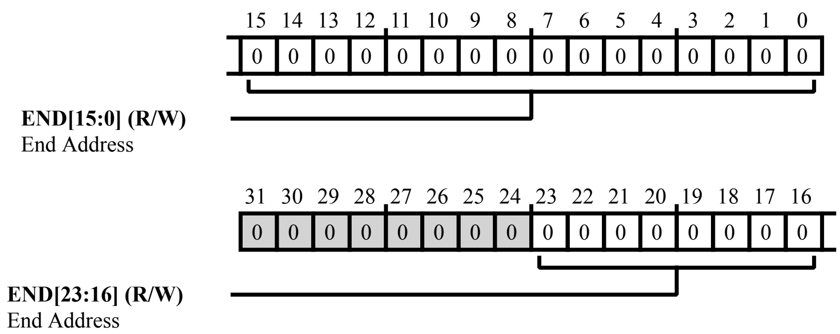

Figure 32-30: SHDBG_PSA4E Register Diagram

Table 32-31: SHDBG_PSA4E Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
23:0 (R/W)	END	End Address. The SHDBG_PSA4E.END bit field provides the instruction breakpoint #1 end address.

Program Sequence Address 4 Start Register

The SHDBG_PSA4S register holds the Instruction address start #4.

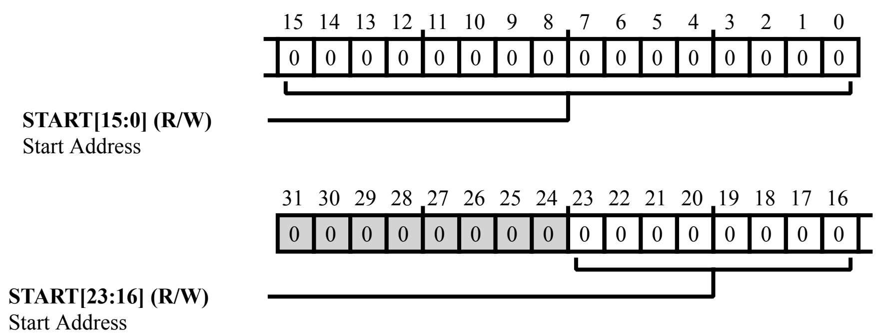

Figure 32-31: SHDBG_PSA4S Register Diagram

Table 32-32: SHDBG_PSA4S Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
23:0 (R/W)	START	Start Address. The SHDBG_PSA4S.START bit field provides the instruction breakpoint #1 start address.

ID Code Register

The SHDBG_REVID register provides the SHARC+ core revision ID value.

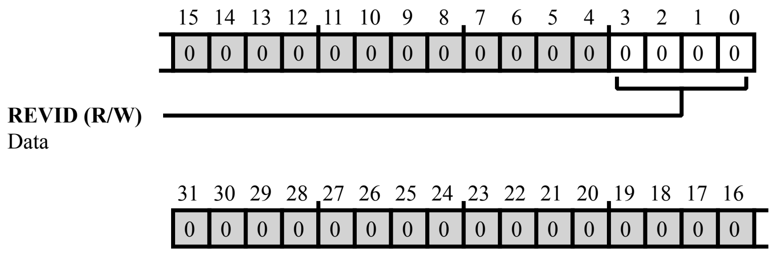

Figure 32-32: SHDBG_REVID Register Diagram

Table 32-33: SHDBG_REVID Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
3:0 (R/W)	REVID	Data.

SEC Interrupt ID Register

The SHDBG_SECI_ID registers holds the SID of the current SEC interrupt. This SID value is the same as the SEC_SID register of the SEC module (Refer to the hardware reference manual). The ACK signal going to the SEC from the core is asserted whenever a write operation is performed on the SECI_ID register.

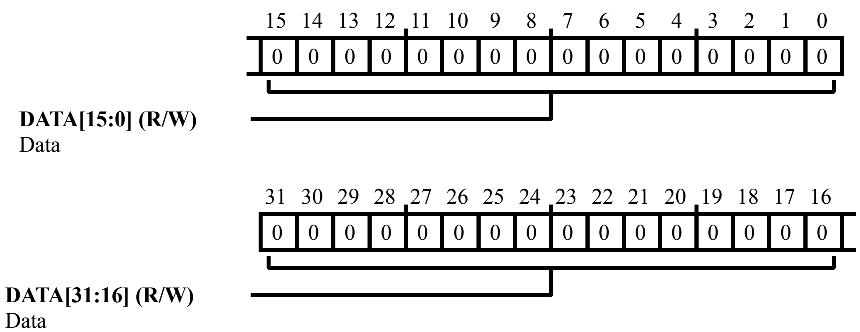

Figure 32-33: SHDBG_SECI_ID Register Diagram

Table 32-34: SHDBG_SECI_ID Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The SHDBG_SECI_ID.DATA bit field holds the SID value which is the same as the SEC_SID register of the SEC module.

33 SHARC-PLUS SHL1C Register Descriptions

SHARC+ L1-Cache Controller (SHL1C) contains the following registers.

Table 33-1: SHARC-PLUS SHL1C Register List

Name	Description
SHL1C_CFG	L1 Cache Configuration 1 Register
SHL1C_CFG2	Range Register Functionality Selection Register
SHL1C_INV_CNT0	Invalidation/Write Back Count 0 Register
SHL1C_INV_IXSTART0	Invalidation/Write Back Index Start 0 Register
SHL1C_RANGE_END0	Range End 0 (Inv, WB, WBI, and Lock) Register
SHL1C_RANGE_END1	Range End 1 (Inv, WB, WBI, and Lock) Register
SHL1C_RANGE_END2	Range End 2 (Non-cacheable and Lock) Register
SHL1C_RANGE_END3	Range End 3 (Non-cacheable and Lock) Register
SHL1C_RANGE_END4	Range End 4 (Non-cacheable and Write Through) Register
SHL1C_RANGE_END5	Range End 5 (Non-cacheable and Write Through) Register
SHL1C_RANGE_END6	Range End 6 (Non-cacheable and Write Through) Register
SHL1C_RANGE_END7	Range End 7 (Non-cacheable and Write Through) Register
SHL1C_RANGE_START0	Range Start 0 (Inv, WB, WBI, and Lock) Register
SHL1C_RANGE_START1	Range Start 1 (Inv, WB, WBI, and Lock) Register
SHL1C_RANGE_START2	Range Start 2 (Non-cacheable and Lock) Register
SHL1C_RANGE_START3	Range Start 3 (Non-cacheable and Lock) Register
SHL1C_RANGE_START4	Range Start 4 (Non-cacheable and Write Through) Register
SHL1C_RANGE_START5	Range Start 5 (Non-cacheable and Write Through) Register
SHL1C_RANGE_START6	Range Start 6 (Non-cacheable and Write Through) Register
SHL1C_RANGE_START7	Range Start 7 (Non-cacheable and Write Through) Register

L1 Cache Configuration 1 Register

The SHL1C_CFG register enables the instruction cache, data memory cache, and program memory cache. This register also selects the size of the caches and other features.

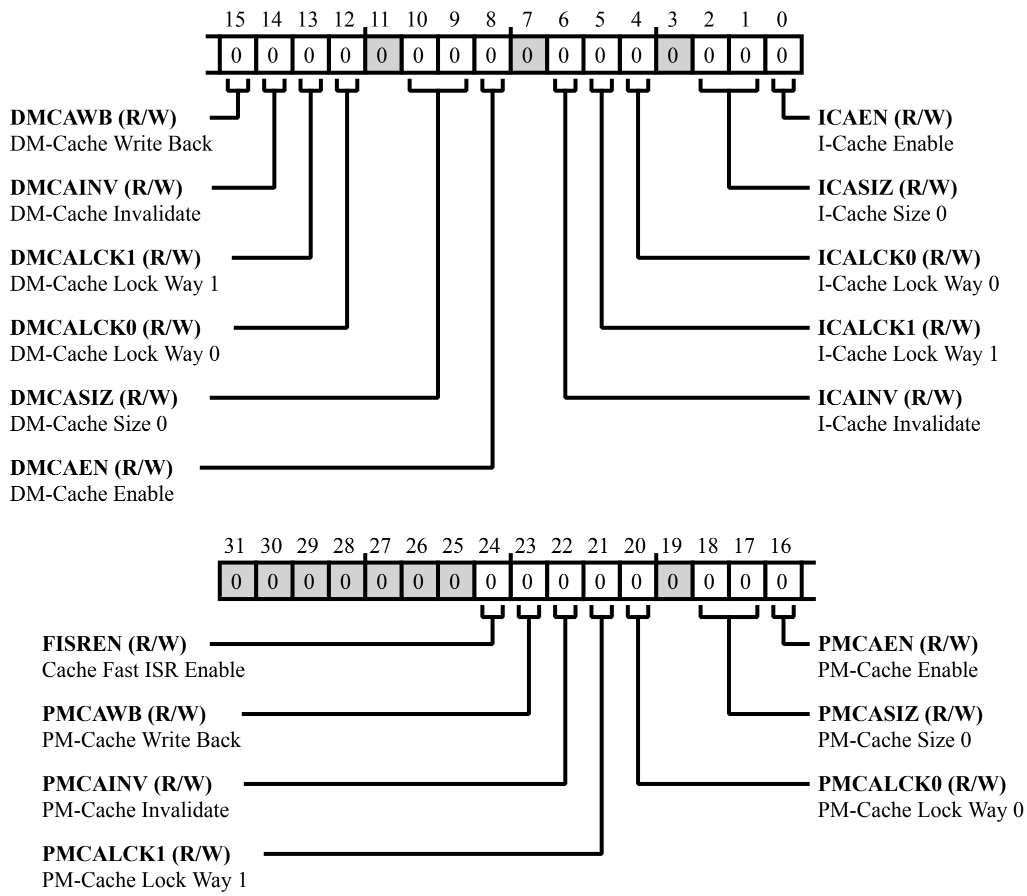

Figure 33-1: SHL1C_CFG Register Diagram

Table 33-2: SHL1C_CFG Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
24 (R/W)	FISREN	Cache Fast ISR Enable. The SHL1C_CFG.FISREN bit enables fast interrupt service for both the instruction and the data cache (I-Cache and D-Cache). Once enabled, the cache controller converts cache line fill decisions to through access if an interrupt is pending.
23 (R/W)	PMCAWB	PM-Cache Write Back. The SHL1C_CFG.PMCAWB bit enables the program memory cache write-back operations.

Table 33-2: SHL1C_CFG Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
22 (R/W)	PMCAINV	PM-Cache Invalidate. The SHL1C_CFG.PMCAINV bit invalidates the program memory cache entries.	
21 (R/W)	PMCALCK1	PM-Cache Lock Way 1. The SHL1C_CFG.PMCALCK1 bit locks program memory cache way 1.	
20 (R/W)	PMCALCK0	PM-Cache Lock Way 0. The SHL1C_CFG.PMCALCK0 bit locks program memory cache way 0.	
18:17 (R/W)	PMCASIZ	PM-Cache Size 0. The SHL1C_CFG.PMCASIZ bit field selects the program memory cache size.	
		0	128K bits
		1	256K bits
		2	512K bits
		3	1M bits
16 (R/W)	PMCAEN	PM-Cache Enable. The SHL1C_CFG.PMCAEN bit enables the program memory cache. This cache can only be enabled in combination with the I-cache and the DM cache (all three caches enabled together). The PM and DM caches cannot be configured independently.	
15 (R/W)	DMCAWB	DM-Cache Write Back. The SHL1C_CFG.DMCAWB bit enables the data memory cache write-back operations.	
14 (R/W)	DMCAINV	DM-Cache Invalidate. The SHL1C_CFG.DMCAINV bit invalidates the data memory cache entries.	
13 (R/W)	DMCALCK1	DM-Cache Lock Way 1. The SHL1C_CFG.DMCALCK1 bit locks data memory cache way 1.	
12 (R/W)	DMCALCK0	DM-Cache Lock Way 0. The SHL1C_CFG.DMCALCK0 bit locks data memory cache way 0.	
10:9 (R/W)	DMCASIZ	DM-Cache Size 0. The SHL1C_CFG.DMCASIZ bit field selects the data memory cache size.	
		0	128K bits
		1	256K bits
		2	512K bits
		3	1M bits

Table 33-2: SHL1C_CFG Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
8 (R/W)	DMCAEN	DM-Cache Enable. The SHL1C_CFG.DMCAEN bit enables the data memory cache. This cache can only be enabled in combination with both the I-cache and the PM cache (all three caches enabled together). The PM and DM caches cannot be configured independently.	
6 (R/W)	ICAINV	I-Cache Invalidate. The SHL1C_CFG.ICAINV bit invalidates the instruction cache entries.	
5 (R/W)	ICALCK1	I-Cache Lock Way 1. The SHL1C_CFG.ICALCK1 bit locks instruction cache way 1.	
4 (R/W)	ICALCK0	I-Cache Lock Way 0. The SHL1C_CFG.ICALCK0 bit locks instruction cache way 0.	
2:1 (R/W)	ICASIZ	I-Cache Size 0. The SHL1C_CFG.ICASIZ bit field selects the instruction cache size.	
		0	128K bits
		1	256K bits
		2	512K bits
		3	1M bits
0 (R/W)	ICAEN	I-Cache Enable. The SHL1C_CFG.ICAEN bit enables the instruction cache. Note that the I-Cache can be enabled by itself or in combination with both the PM and DM caches. The PM and DM caches cannot be configured independently.	

Range Register Functionality Selection Register

The SHL1C_CFG2 register selects the functionality of range register pairs, supporting a variety of cache operations.

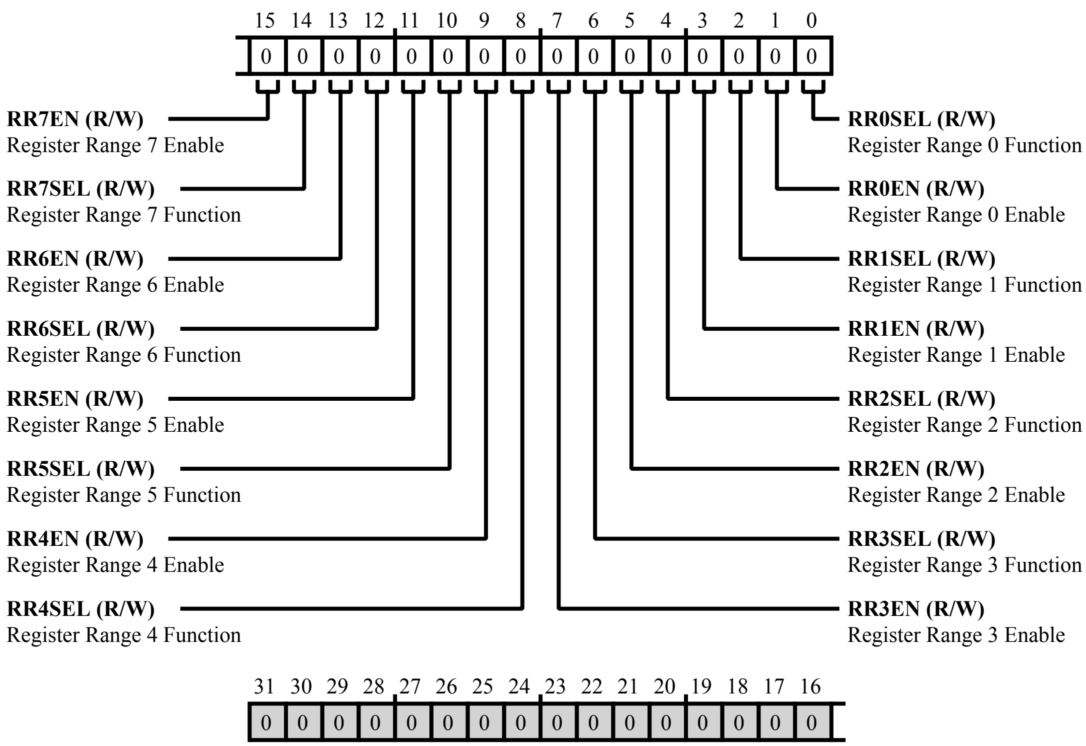

Figure 33-2: SHL1C_CFG2 Register Diagram

Table 33-3: SHL1C_CFG2 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
15 (R/W)	RR7EN	Register Range 7 Enable. The SHL1C_CFG2.RR7EN bit enables the function selected with the SHL1C_CFG2.RR7SEL bit for this range.	
14 (R/W)	RR7SEL	Register Range 7 Function. The SHL1C_CFG2.RR7SEL bit selects whether the register range is a cache write-through range or a non-cacheable range.	
		0	Write-Through Range
		1	Non-cacheable Range
13 (R/W)	RR6EN	Register Range 6 Enable. The SHL1C_CFG2.RR6EN bit enables the function selected with the SHL1C_CFG2.RR6SEL bit for this range.	

Table 33-3: SHL1C_CFG2 Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
12 (R/W)	RR6SEL	Register Range 6 Function. The SHL1C_CFG2.RR6SEL bit selects whether the register range is a cache write-through range or a non-cacheable range.	
		0	Write-Through Range
		1	Non-cacheable Range
11 (R/W)	RR5EN	Register Range 5 Enable. The SHL1C_CFG2.RR5EN bit enables the function selected with the SHL1C_CFG2.RR5SEL bit for this range.	
10 (R/W)	RR5SEL	Register Range 5 Function. The SHL1C_CFG2.RR5SEL bit selects whether the register range is a cache write-through range or a non-cacheable range.	
		0	Write-Through Range
		1	Non-cacheable Range
9 (R/W)	RR4EN	Register Range 4 Enable. The SHL1C_CFG2.RR4EN bit enables the function selected with the SHL1C_CFG2.RR4SEL bit for this range.	
8 (R/W)	RR4SEL	Register Range 4 Function. The SHL1C_CFG2.RR4SEL bit selects whether the register range is a cache write-through range or a non-cacheable range.	
		0	Write-Through Range
		1	Non-cacheable Range
7 (R/W)	RR3EN	Register Range 3 Enable. The SHL1C_CFG2.RR3EN bit enables the function selected with the SHL1C_CFG2.RR3SEL bit for this range.	
6 (R/W)	RR3SEL	Register Range 3 Function. The SHL1C_CFG2.RR3SEL bit selects whether the register range is a cache lock range or a non-cacheable range.	
		0	Cache Lock Range
		1	Non-cacheable Range
5 (R/W)	RR2EN	Register Range 2 Enable. The SHL1C_CFG2.RR2EN bit enables the function selected with the SHL1C_CFG2.RR2SEL bit for this range.	

Table 33-3: SHL1C_CFG2 Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
4 (R/W)	RR2SEL	Register Range 2 Function. The `SHL1C_CFG2.RR2SEL` bit selects whether the register range is a cache lock range or a non-cacheable range. 0 Cache Lock Range 1 Non-cacheable Range
3 (R/W)	RR1EN	Register Range 1 Enable. The `SHL1C_CFG2.RR1EN` bit enables the function selected with the `SHL1C_CFG2.RR1SEL` bit for this range.
2 (R/W)	RR1SEL	Register Range 1 Function. The `SHL1C_CFG2.RR1SEL` bit selects whether the register range is a cache lock range or a cache write-back invalidate range. 0 Cache Lock Range 1 Write-Back Invalidate Range
1 (R/W)	RR0EN	Register Range 0 Enable. The `SHL1C_CFG2.RR0EN` bit enables the function selected with the `SHL1C_CFG2.RR0SEL` bit for this range.
0 (R/W)	RR0SEL	Register Range 0 Function. The `SHL1C_CFG2.RR0SEL` bit selects whether the register range is a cache lock range or a cache write-back invalidate range. 0 Cache Lock Range 1 Write-Back Invalidate Range

Invalidation/Write Back Count 0 Register

The SHL1C_INV_CNT0 register selects a count value of the number of indexes to be Invalidated or WBI. These registers are running registers, which means that after clearing one index, the value of the count register decrements. When invalidation or flushing is in progress these registers should not be accessed.

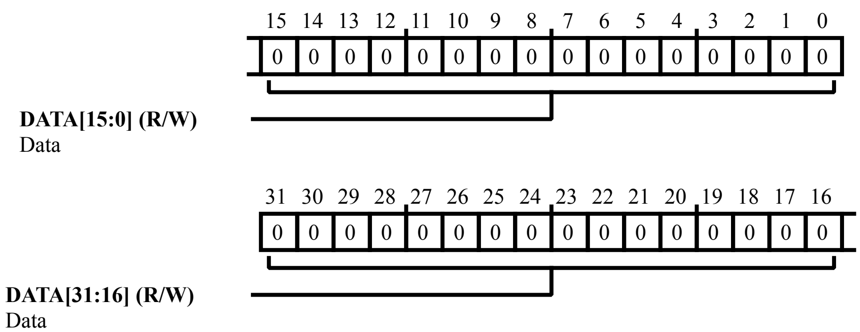

Figure 33-3: SHL1C_INV_CNT0 Register Diagram

Table 33-4: SHL1C_INV_CNT0 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The SHL1C_INV_CNT0.DATA bits hold the count value of the number of indexes to be Invalidated or WBI. These registers are running registers, which means that after clearing one index, value of count register decrements. When invalidation or flushing is in progress these registers should not be accessed.

Invalidation/Write Back Index Start 0 Register

The SHL1C_INV_IXSTART0 register contains the index value for address range-based cache write-back and cache write-back invalidation operations.

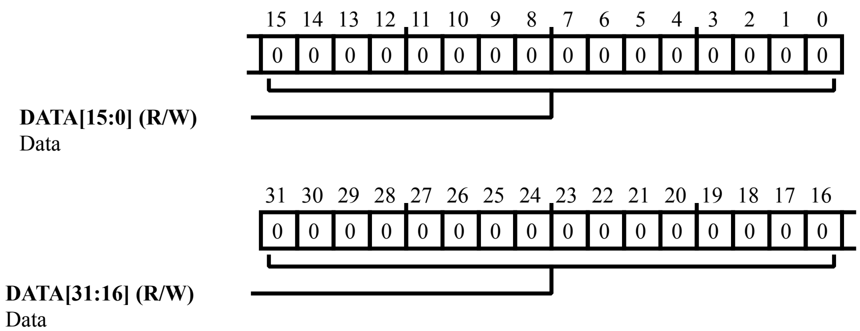

Figure 33-4: SHL1C_INV_IXSTART0 Register Diagram

Table 33-5: SHL1C_INV_IXSTART0 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Data. The SHL1C_INV_IXSTART0.DATA bit field holds the index value corresponding to the start address of the configured Range register. Once the Range registers are filled and properties selected, the cache controller internally computes the starting index that corresponds to the start address and stores it in this bit field. This is a running register, which means that after clearing one index, the value of the index register increments. When invalidation or flushing is in progress this register should not be accessed.

Range End 0 (Inv, WB, WBI, and Lock) Register

The SHL1C_RANGE_END0 register selects a end range address for cache invalidation, cache write-back replacement, and cache write-back invalidation operations.

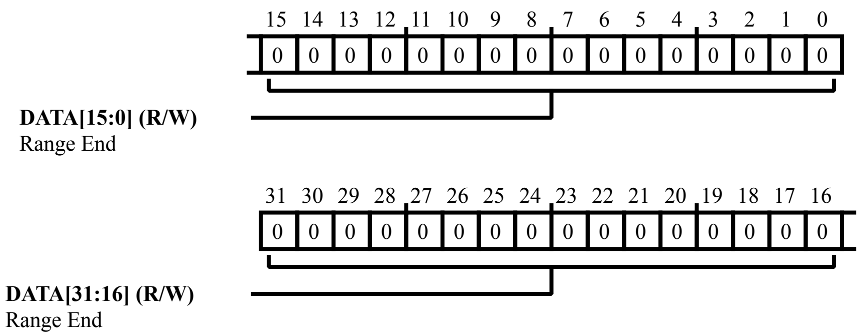

Figure 33-5: SHL1C_RANGE_END0 Register Diagram

Table 33-6: SHL1C_RANGE_END0 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Range End. The SHL1C_RANGE_END0.DATA bits hold the range end address.

Range End 1 (Inv, WB, WBI, and Lock) Register

The SHL1C_RANGE_END1 register selects a end range address for cache invalidation, cache write-back replacement, and cache write-back invalidation operations.

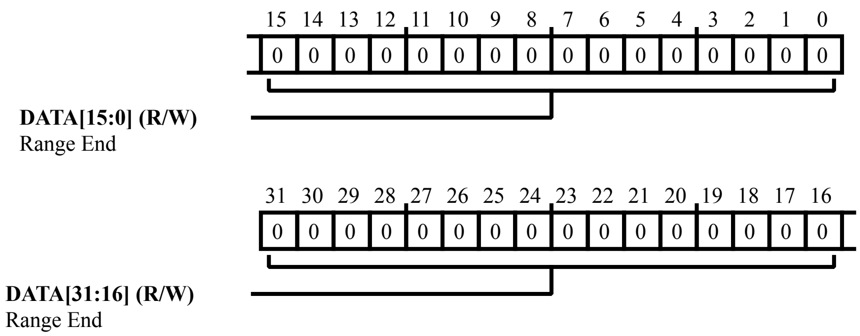

Figure 33-6: SHL1C_RANGE_END1 Register Diagram

Table 33-7: SHL1C_RANGE_END1 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Range End. The SHL1C_RANGE_END1.DATA bits hold the range end address.

Range End 2 (Non-cacheable and Lock) Register

The SHL1C_RANGE_END2 register selects a end range address for non-cacheable and cache locking operations.

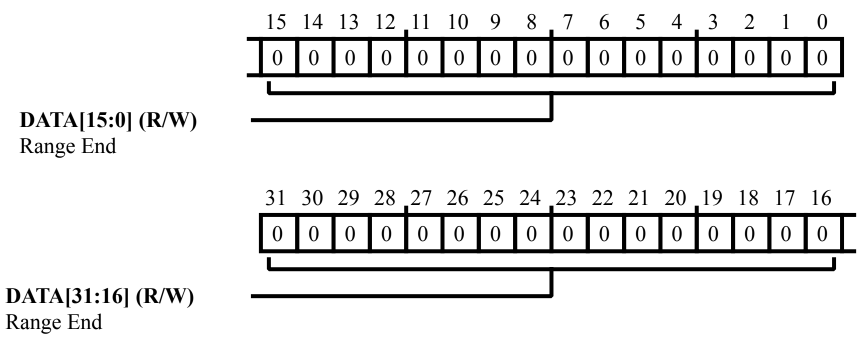

Figure 33-7: SHL1C_RANGE_END2 Register Diagram

Table 33-8: SHL1C_RANGE_END2 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Range End. The SHL1C_RANGE_END2.DATA bits hold the range end address.

Range End 3 (Non-cacheable and Lock) Register

The SHL1C_RANGE_END3 register selects a end range address for non-cacheable and cache locking operations.

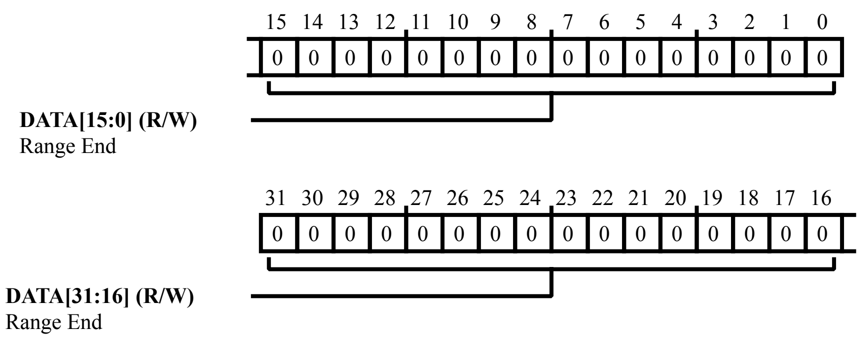

Figure 33-8: SHL1C_RANGE_END3 Register Diagram

Table 33-9: SHL1C_RANGE_END3 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Range End. The SHL1C_RANGE_END3.DATA bits hold the range end address.

Range End 4 (Non-cacheable and Write Through) Register

The SHL1C_RANGE_END4 register selects a end range address for non-cacheable and cache write through operations.

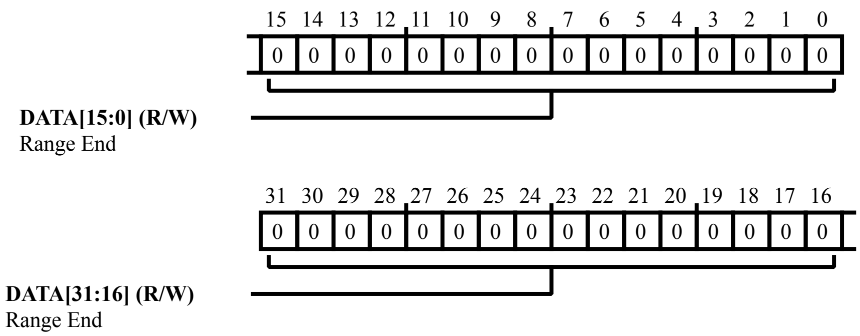

Figure 33-9: SHL1C_RANGE_END4 Register Diagram

Table 33-10: SHL1C_RANGE_END4 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Range End. The SHL1C_RANGE_END4.DATA bits hold the range end address.

Range End 5 (Non-cacheable and Write Through) Register

The SHL1C_RANGE_END5 register selects a end range address for non-cacheable and cache write through operations.

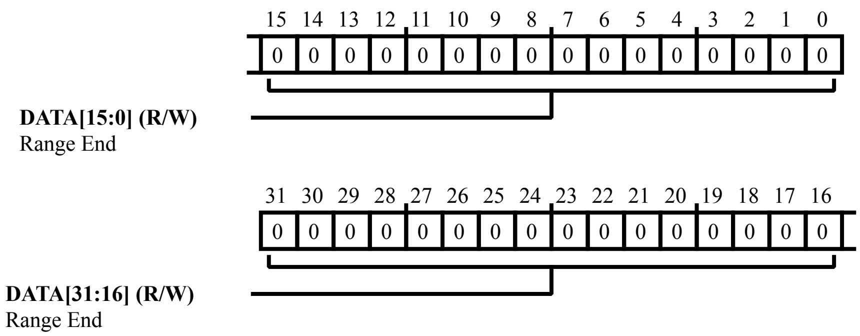

Figure 33-10: SHL1C_RANGE_END5 Register Diagram

Table 33-11: SHL1C_RANGE_END5 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Range End. The SHL1C_RANGE_END5.DATA bits hold the range end address.

Range End 6 (Non-cacheable and Write Through) Register

The SHL1C_RANGE_END6 register selects a end range address for non-cacheable and cache write through operations.

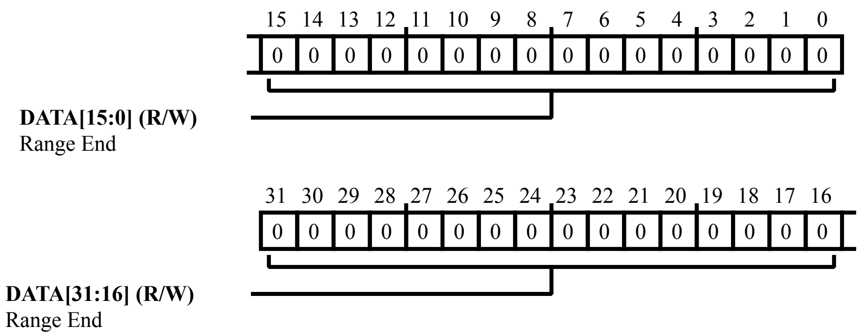

Figure 33-11: SHL1C_RANGE_END6 Register Diagram

Table 33-12: SHL1C_RANGE_END6 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Range End. The SHL1C_RANGE_END6.DATA bits hold the range end address.

Range End 7 (Non-cacheable and Write Through) Register

The SHL1C_RANGE_END7 register selects a end range address for non-cacheable and cache write through operations.

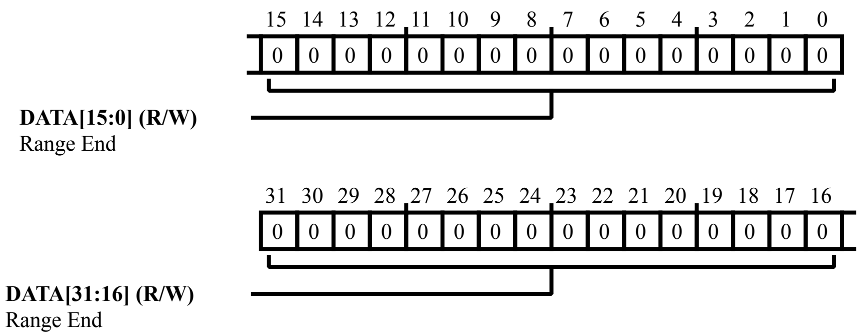

Figure 33-12: SHL1C_RANGE_END7 Register Diagram

Table 33-13: SHL1C_RANGE_END7 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Range End. The SHL1C_RANGE_END7.DATA bits hold the range end address.

Range Start 0 (Inv, WB, WBI, and Lock) Register

The SHL1C_RANGE_START0 register selects a start range address for cache invalidation, cache write-back replacement, and cache write-back invalidation operations.

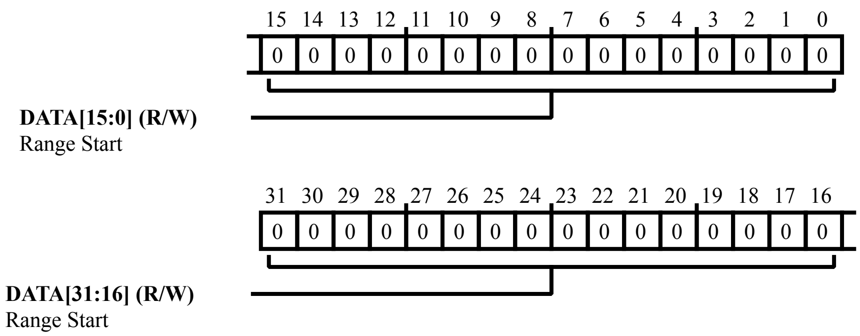

Figure 33-13: SHL1C_RANGE_START0 Register Diagram

Table 33-14: SHL1C_RANGE_START0 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Range Start. The SHL1C_RANGE_START0.DATA bits hold the range start address.

Range Start 1 (Inv, WB, WBI, and Lock) Register

The SHL1C_RANGE_START1 register selects a start range address for cache invalidation, cache write-back replacement, and cache write-back invalidation operations.

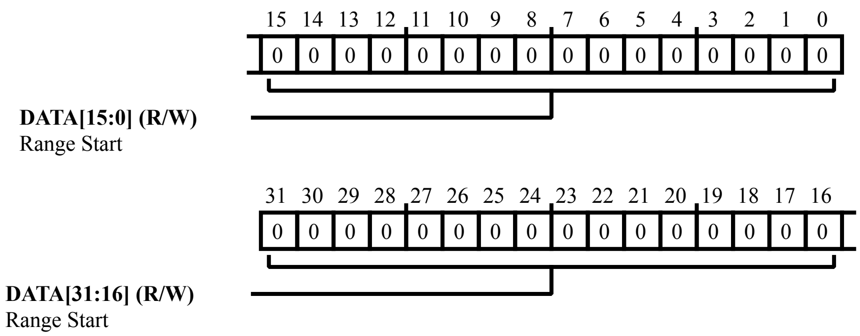

Figure 33-14: SHL1C_RANGE_START1 Register Diagram

Table 33-15: SHL1C_RANGE_START1 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Range Start. The SHL1C_RANGE_START1.DATA bits hold the range start address.

Range Start 2 (Non-cacheable and Lock) Register

The SHL1C_RANGE_START2 register selects a start range address for non cacheable and cache locking operations.

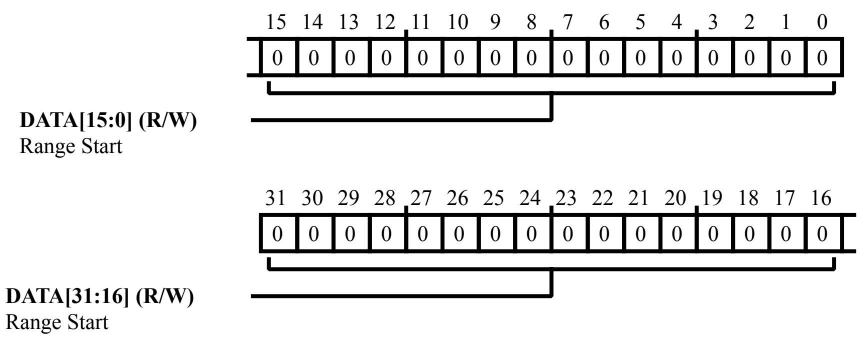

Figure 33-15: SHL1C_RANGE_START2 Register Diagram

Table 33-16: SHL1C_RANGE_START2 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Range Start. The SHL1C_RANGE_START2.DATA bits hold the range start address.

Range Start 3 (Non-cacheable and Lock) Register

The SHL1C_RANGE_START3 register selects a start range address for non-cacheable and cache locking operations.

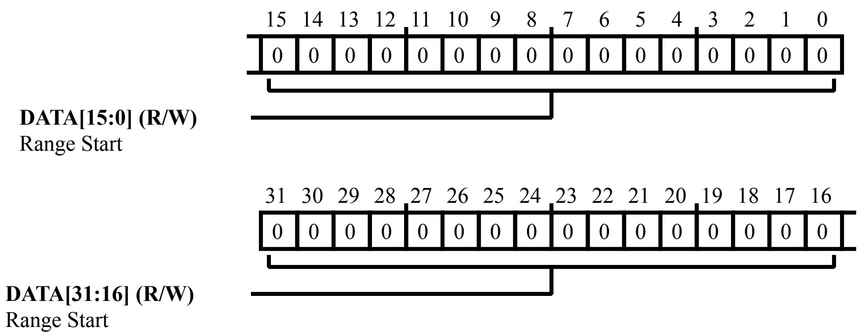

Figure 33-16: SHL1C_RANGE_START3 Register Diagram

Table 33-17: SHL1C_RANGE_START3 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Range Start. The SHL1C_RANGE_START3.DATA bits hold the range start address.

Range Start 4 (Non-cacheable and Write Through) Register

The SHL1C_RANGE_START4 register selects a start range address for non-cacheable and cache write through operations.

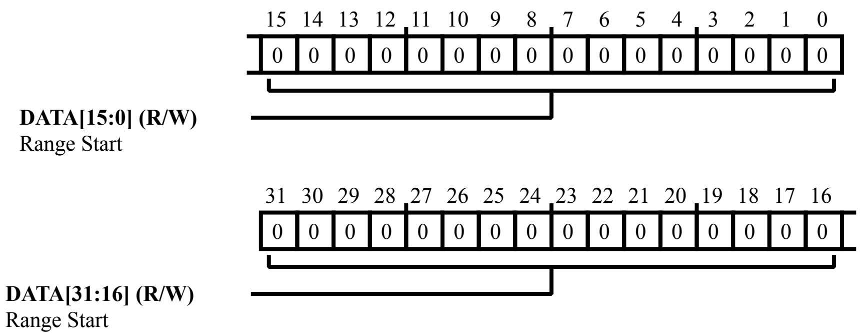

Figure 33-17: SHL1C_RANGE_START4 Register Diagram

Table 33-18: SHL1C_RANGE_START4 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Range Start. The SHL1C_RANGE_START4.DATA bits hold the range start address.

Range Start 5 (Non-cacheable and Write Through) Register

The SHL1C_RANGE_START5 register selects a start range address for non-cacheable and cache write through operations.

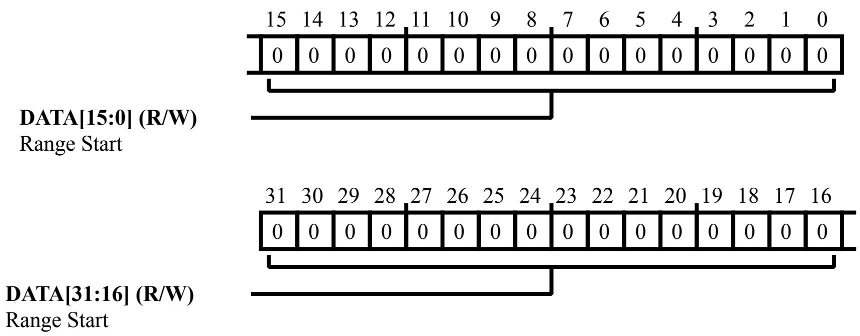

Figure 33-18: SHL1C_RANGE_START5 Register Diagram

Table 33-19: SHL1C_RANGE_START5 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Range Start. The SHL1C_RANGE_START5.DATA bits hold the range start address.

Range Start 6 (Non-cacheable and Write Through) Register

The SHL1C_RANGE_START6 register selects a start range address for non-cacheable and cache write through operations.

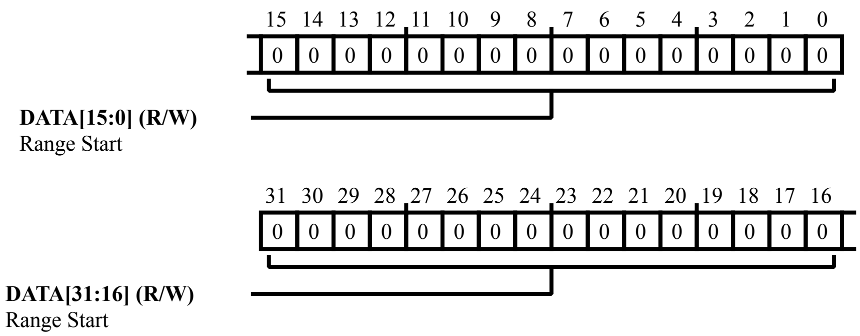

Figure 33-19: SHL1C_RANGE_START6 Register Diagram

Table 33-20: SHL1C_RANGE_START6 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Range Start. The SHL1C_RANGE_START6.DATA bits hold the range start address.

Range Start 7 (Non-cacheable and Write Through) Register

The SHL1C_RANGE_START7 register selects a start range address for non-cacheable and cache write through operations.

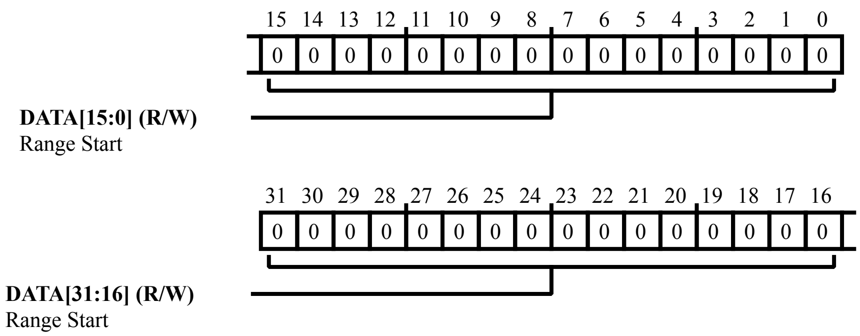

Figure 33-20: SHL1C_RANGE_START7 Register Diagram

Table 33-21: SHL1C_RANGE_START7 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Range Start. The SHL1C_RANGE_START7.DATA bits hold the range start address.

34 SHARC-PLUS PFM Register Descriptions

Performance Monitor (PFM) contains the following registers.

Table 34-1: SHARC-PLUS PFM Register List

Name	Description
PFM_CFG	Configuration Register
PFM_CNTR3	Counter 3 Register
PFM_CNTR3CLR	Counter 3 Clear Register
PFM_CNTR3PAUSE	Counter 3 Pause Register
PFM_CNTR3START	Counter 3 Start Register
PFM_CNTR4	Counter 4 Register
PFM_CNTR4CLR	Counter 4 Clear Register
PFM_CNTR4PAUSE	Counter 4 Pause Register
PFM_CNTR4START	Counter 4 Start Register
PFM_CNTR5	Counter 5 Register
PFM_CNTR5CLR	Counter 5 Clear Register
PFM_CNTR5PAUSE	Counter 5 Pause Register
PFM_CNTR5START	Counter 5 Start Register
PFM_CNTR6	Counter 6 Register
PFM_CNTR6CLR	Counter 6 Clear Register
PFM_CNTR6PAUSE	Counter 6 Pause Register
PFM_CNTR6START	Counter 6 Start Register

Configuration Register

The PFM_CFG register is for enabling performance cache parameters. Enable only one cache memory mode at a time.

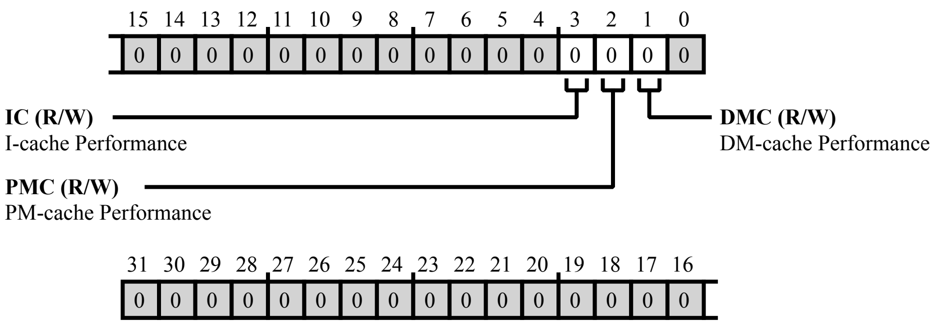

Figure 34-1: PFM_CFG Register Diagram

Table 34-2: PFM_CFG Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
3 (R/W)	IC	I-cache Performance. Set this bit to record I-cache performance parameters in counters.
2 (R/W)	PMC	PM-cache Performance. Set this bit to record PM-cache performance parameters in counters.
1 (R/W)	DMC	DM-cache Performance. Set this bit to record DM-cache performance parameters in counters.

Counter 3 Register

The PFM_CNTR3 register records the total number of occurrences of cache hits. It will record DM, PM or I cache hits based on the configuration settings in the PFM_CFG register.

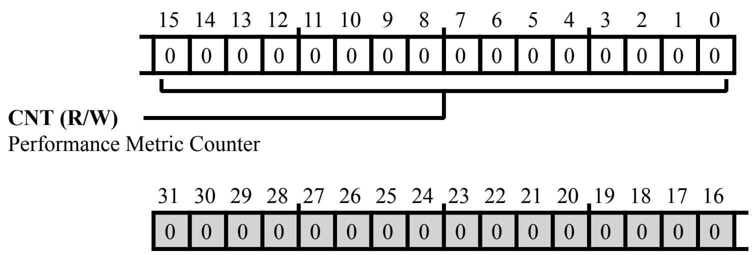

Figure 34-2: PFM_CNTR3 Register Diagram

Table 34-3: PFM_CNTR3 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
15:0 (R/W)	CNT	Performance Metric Counter. Count performance metric.

Counter 3 Clear Register

The PFM_CNTR3CLR register is the control register for clearing a count in the PFM_CNTR3 register.

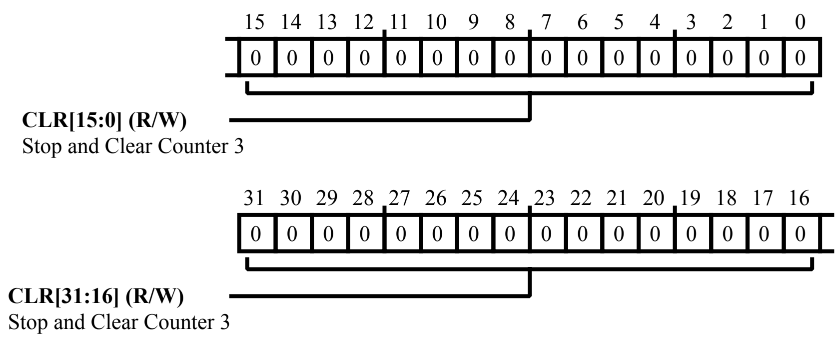

Figure 34-3: PFM_CNTR3CLR Register Diagram

Table 34-4: PFM_CNTR3CLR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	CLR	Stop and Clear Counter 3. Write any value to stop and clear the counter

Counter 3 Pause Register

The PFM_CNTR3PAUSE register is the control register for pausing a count in the PFM_CNTR3 register.

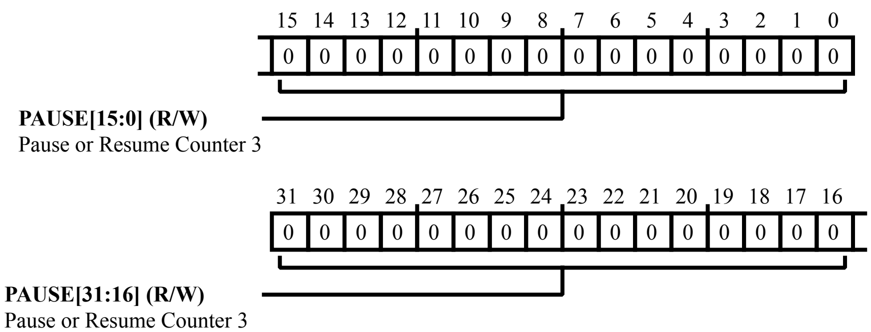

Figure 34-4: PFM_CNTR3PAUSE Register Diagram

Table 34-5: PFM_CNTR3PAUSE Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	PAUSE	Pause or Resume Counter 3. Write any value to pause or resume the counter

Counter 3 Start Register

The PFM_CNTR3START register is the control register for initiating a count in the PFM_CNTR3 register.

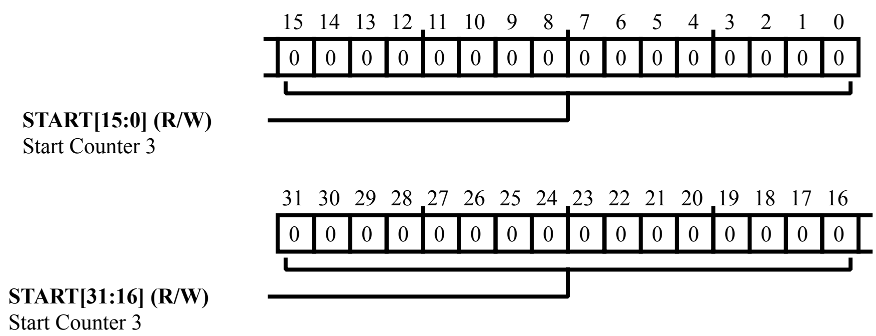

Figure 34-5: PFM_CNTR3START Register Diagram

Table 34-6: PFM_CNTR3START Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	START	Start Counter 3. Write any value to start the counter

Counter 4 Register

The PFM_CNTR4 register records the number of crosscheck hits when configured for DM-cache or PM-cache in the PFM_CFG register. If the PFM_CFG register is configured for I-cache, the PFM_CNTR4 register records the number of cache misses.

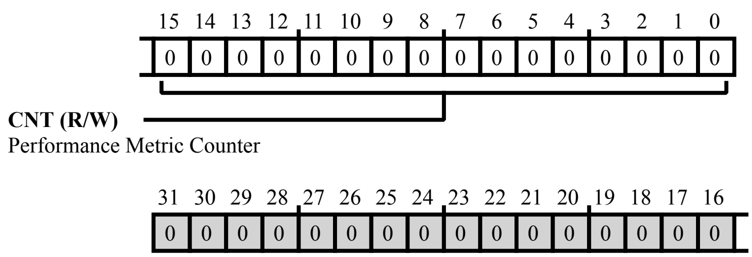

Figure 34-6: PFM_CNTR4 Register Diagram

Table 34-7: PFM_CNTR4 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
15:0 (R/W)	CNT	Performance Metric Counter. Count performance metric.

Counter 4 Clear Register

The PFM_CNTR4CLR register is the control register for clearing a count in the PFM_CNTR4 register.

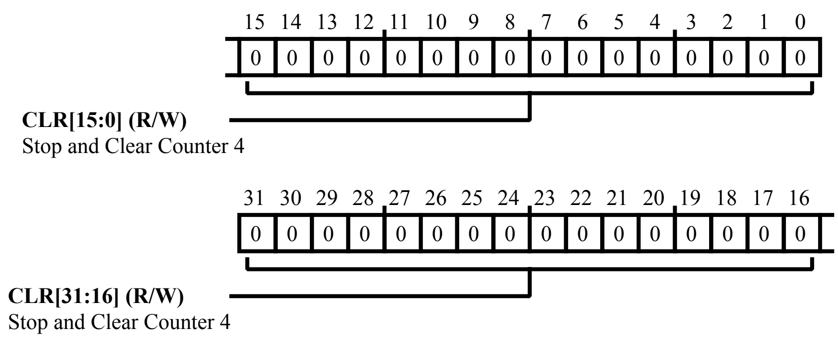

Figure 34-7: PFM_CNTR4CLR Register Diagram

Table 34-8: PFM_CNTR4CLR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	CLR	Stop and Clear Counter 4. Write any value to stop and clear the counter

Counter 4 Pause Register

The PFM_CNTR4PAUSE register is the control register for pausing a count in the PFM_CNTR4 register.

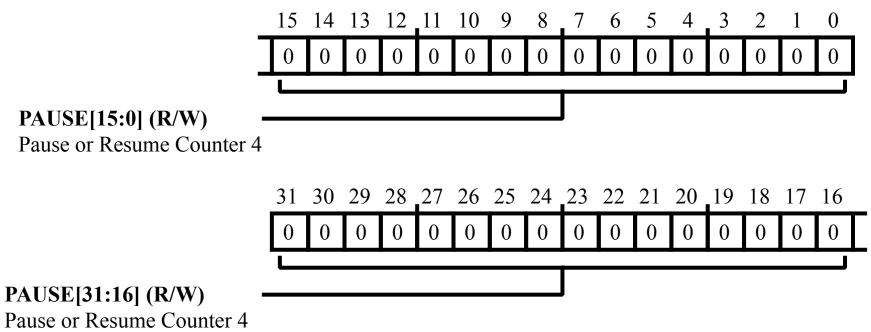

Figure 34-8: PFM_CNTR4PAUSE Register Diagram

Table 34-9: PFM_CNTR4PAUSE Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	PAUSE	Pause or Resume Counter 4. Write any value to pause or resume the counter

Counter 4 Start Register

The PFM_CNTR4START register is the control register for initiating a count in the PFM_CNTR4 register.

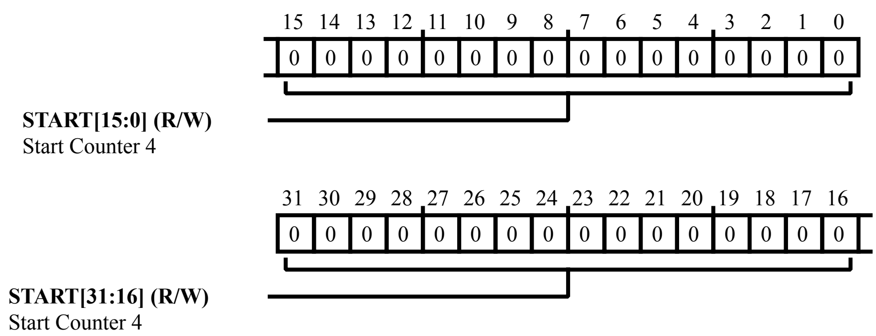

Figure 34-9: PFM_CNTR4START Register Diagram

Table 34-10: PFM_CNTR4START Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	START	Start Counter 4. Write any value to start the counter

Counter 5 Register

The PFM_CNTR5 register records the number of cache misses without writeback for DM-cache or PM-cache depending on the settings in the PFM_CFG register.

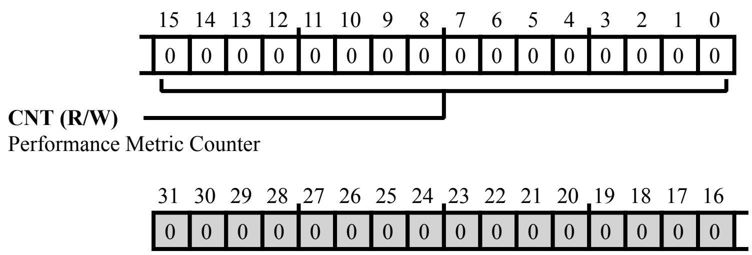

Figure 34-10: PFM_CNTR5 Register Diagram

Table 34-11: PFM_CNTR5 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
15:0 (R/W)	CNT	Performance Metric Counter. Count performance metric.

Counter 5 Clear Register

The PFM_CNTR5CLR register is the control register for clearing a count in the PFM_CNTR5 register.

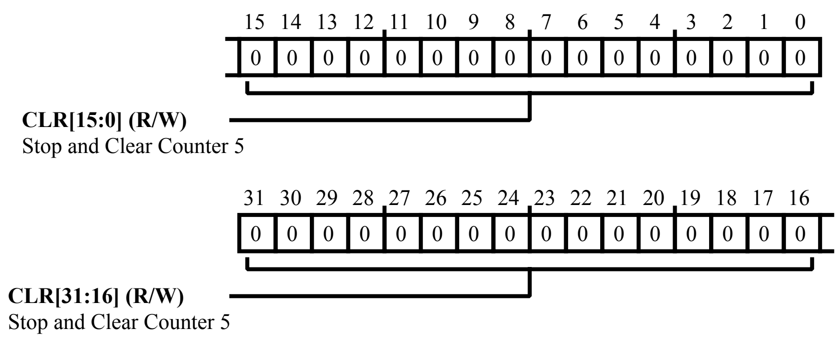

Figure 34-11: PFM_CNTR5CLR Register Diagram

Table 34-12: PFM_CNTR5CLR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	CLR	Stop and Clear Counter 5. Write any value to stop and clear the counter

Counter 5 Pause Register

The PFM_CNTR5PAUSE register is the control register for pausing a count in the PFM_CNTR5 register.

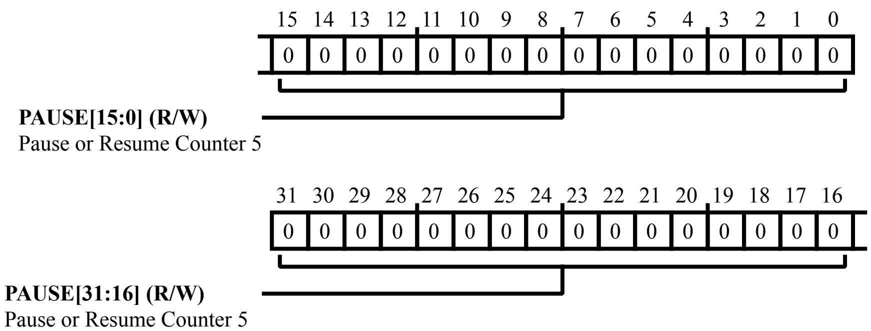

Figure 34-12: PFM_CNTR5PAUSE Register Diagram

Table 34-13: PFM_CNTR5PAUSE Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	PAUSE	Pause or Resume Counter 5. Write any value to pause or resume the counter

Counter 5 Start Register

The PFM_CNTR5START register is the control register for initiating a count in the PFM_CNTR5 register.

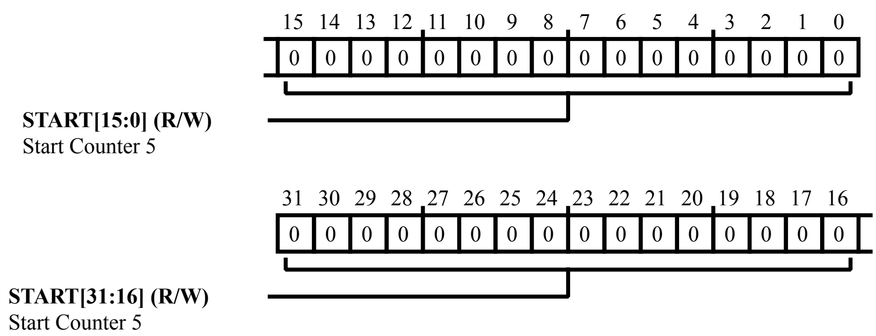

Figure 34-13: PFM_CNTR5START Register Diagram

Table 34-14: PFM_CNTR5START Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	START	Start Counter 5. Write any value to start the counter

Counter 6 Register

The PFM_CNTR6 register records the number of cache misses with writeback for DM-cache or PM-cache depending on the settings in the PFM_CFG register.

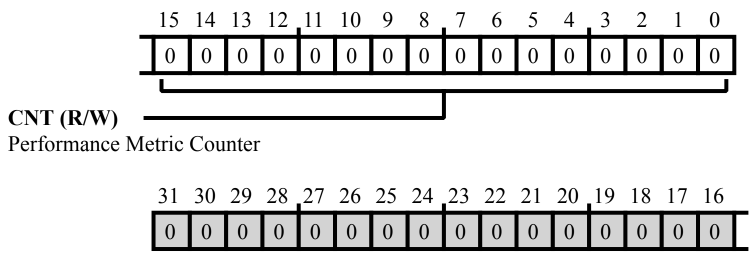

Figure 34-14: PFM_CNTR6 Register Diagram

Table 34-15: PFM_CNTR6 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
15:0 (R/W)	CNT	Performance Metric Counter. Count performance metric.

Counter 6 Clear Register

The PFM_CNTR6CLR register is the control register for clearing a count in the PFM_CNTR6 register.

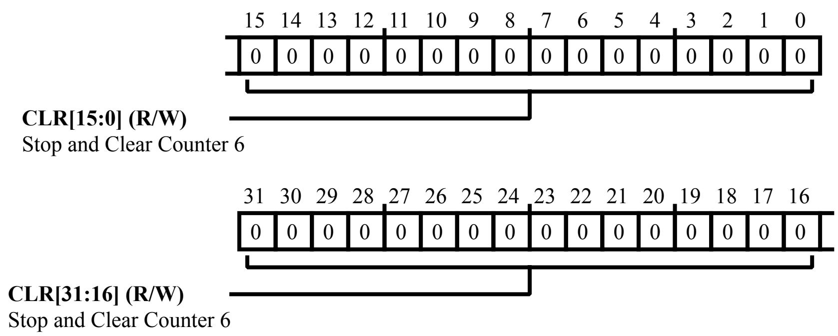

Figure 34-15: PFM_CNTR6CLR Register Diagram

Table 34-16: PFM_CNTR6CLR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	CLR	Stop and Clear Counter 6. Write any value to stop and clear the counter

Counter 6 Pause Register

The PFM_CNTR6PAUSE register is the control register for pausing a count in the PFM_CNTR6 register.

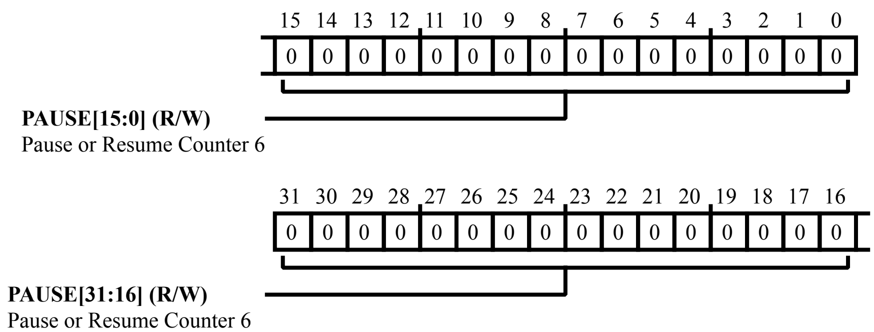

Figure 34-16: PFM_CNTR6PAUSE Register Diagram

Table 34-17: PFM_CNTR6PAUSE Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	PAUSE	Pause or Resume Counter 6. Write any value to pause or resume the counter

Counter 6 Start Register

The PFM_CNTR6START register is the control register for initiating a count in the PFM_CNTR6 register.

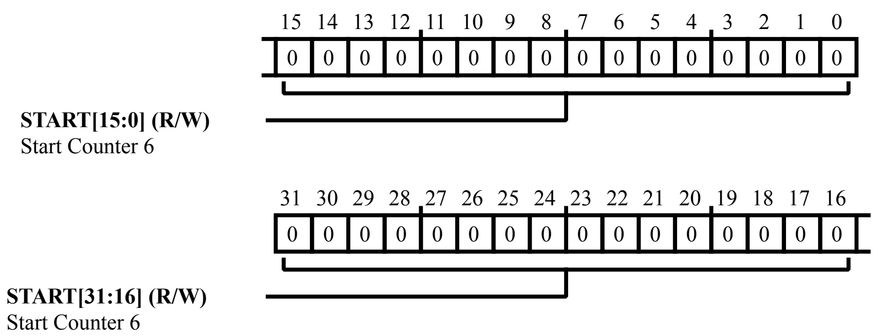

Figure 34-17: PFM_CNTR6START Register Diagram

Table 34-18: PFM_CNTR6START Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	START	Start Counter 6. Write any value to start the counter

35 SHARC-PLUS Register List

This appendix lists Memory-Mapped Register address and register names. The modules are presented in alphabetical order.

Table 35-1: SHARC-PLUS CMMR MMR Register Addresses

Memory Mapped Address	Register Name	Description	Reset Value
0x30024	CMMR_SYSCTL	CMMR System Control Register	0x00000000
0x3B000	CMMR_GPERR_STAT	CMMR General-Purpose Parity Error Status Register	0x00000000
0x3B100	CMMR_PFB_NOCHRT0_ST	CMMR PFB No Caching Return 0 Start Address Register	0x00000000
0x3B101	CMMR_PFB_NOCHRT0_END	CMMR PFB No Caching Return 0 End Address Register	0x00000000
0x3E030	CMMR_PWR_L1_SD_CTL	CMMR L1 BANK SHUT DOWN CONTROL	0x00000000
0x3E031	CMMR_PWR_GLB_CTL	CMMR Core Global Power Control Register	0x00000000
0x3E032	CMMR_PWR_L1_LS_CTL	CMMR L1 BANK SLEEP CONTROL	0x00000000

Table 35-2: SHARC-PLUS REGF MMR Register Addresses

Memory Mapped Address	Register Name	Description	Reset Value
0x1010	REGF_MODE1	REGF Mode Control 1 Register	0x00000000
0x1020	REGF_MMASK	REGF Mode Mask Register	0x00000000
0x1030	REGF_MODE1STK	REGF Mode 1 Stack (Top Entry) Register	0x00000000
0x1040	REGF_MODE2	REGF Mode Control 2 Register	0x00000000
0x1050	REGF_FADDR	REGF Instruction Pipeline Stage Address Register	0x00000000
0x1060	REGF_DADDR	REGF Decode Address Register	0x00000000
0x1070	REGF_PC	REGF Program Counter Register	0x00000000
0x1080	REGF_PCSTK	REGF Program Counter Stack Register	0x00000000
0x1090	REGF_PCSTKP	REGF Program Counter Stack Pointer Register	0x00000000

Table 35-2: SHARC-PLUS REGF MMR Register Addresses (Continued)

Memory Mapped Address	Register Name	Description	Reset Value
0x10A0	REGF_LADDR	REGF Loop Address Stack Register	0x00000000
0x10B0	REGF_LCNTR	REGF Loop Counter Register	0x00000000
0x10C0	REGF_CURLCNTR	REGF Current Loop Counter Register	0x00000000
0x10D0	REGF_TPERIOD	REGF Timer Period Register	0x00000000
0x10E0	REGF_TCOUNT	REGF Timer Count Register	0x00000000
0x10F0	REGF_R[n]	REGF Register File (PEx) Data Registers (Rx, Fx)	0x00000000
0x10F1	REGF_R[n]	REGF Register File (PEx) Data Registers (Rx, Fx)	0x00000000
0x10F2	REGF_R[n]	REGF Register File (PEx) Data Registers (Rx, Fx)	0x00000000
0x10F3	REGF_R[n]	REGF Register File (PEx) Data Registers (Rx, Fx)	0x00000000
0x10F4	REGF_R[n]	REGF Register File (PEx) Data Registers (Rx, Fx)	0x00000000
0x10F5	REGF_R[n]	REGF Register File (PEx) Data Registers (Rx, Fx)	0x00000000
0x10F6	REGF_R[n]	REGF Register File (PEx) Data Registers (Rx, Fx)	0x00000000
0x10F7	REGF_R[n]	REGF Register File (PEx) Data Registers (Rx, Fx)	0x00000000
0x10F8	REGF_R[n]	REGF Register File (PEx) Data Registers (Rx, Fx)	0x00000000
0x10F9	REGF_R[n]	REGF Register File (PEx) Data Registers (Rx, Fx)	0x00000000
0x10FA	REGF_R[n]	REGF Register File (PEx) Data Registers (Rx, Fx)	0x00000000
0x10FB	REGF_R[n]	REGF Register File (PEx) Data Registers (Rx, Fx)	0x00000000
0x10FC	REGF_R[n]	REGF Register File (PEx) Data Registers (Rx, Fx)	0x00000000
0x10FD	REGF_R[n]	REGF Register File (PEx) Data Registers (Rx, Fx)	0x00000000
0x10FE	REGF_R[n]	REGF Register File (PEx) Data Registers (Rx, Fx)	0x00000000
0x10FF	REGF_R[n]	REGF Register File (PEx) Data Registers (Rx, Fx)	0x00000000
0x1100	REGF_S[n]	REGF Register File (PEy) Data Registers (Sx, SFx)	0x00000000
0x1101	REGF_S[n]	REGF Register File (PEy) Data Registers (Sx, SFx)	0x00000000
0x1102	REGF_S[n]	REGF Register File (PEy) Data Registers (Sx, SFx)	0x00000000
0x1103	REGF_S[n]	REGF Register File (PEy) Data Registers (Sx, SFx)	0x00000000
0x1104	REGF_S[n]	REGF Register File (PEy) Data Registers (Sx, SFx)	0x00000000
0x1105	REGF_S[n]	REGF Register File (PEy) Data Registers (Sx, SFx)	0x00000000
0x1106	REGF_S[n]	REGF Register File (PEy) Data Registers (Sx, SFx)	0x00000000
0x1107	REGF_S[n]	REGF Register File (PEy) Data Registers (Sx, SFx)	0x00000000
0x1108	REGF_S[n]	REGF Register File (PEy) Data Registers (Sx, SFx)	0x00000000
0x1109	REGF_S[n]	REGF Register File (PEy) Data Registers (Sx, SFx)	0x00000000

Table 35-2: SHARC-PLUS REGF MMR Register Addresses (Continued)

Memory Mapped Address	Register Name	Description	Reset Value
0x110A	REGF_S[n]	REGF Register File (PEy) Data Registers (Sx, SFx)	0x00000000
0x110B	REGF_S[n]	REGF Register File (PEy) Data Registers (Sx, SFx)	0x00000000
0x110C	REGF_S[n]	REGF Register File (PEy) Data Registers (Sx, SFx)	0x00000000
0x110D	REGF_S[n]	REGF Register File (PEy) Data Registers (Sx, SFx)	0x00000000
0x110E	REGF_S[n]	REGF Register File (PEy) Data Registers (Sx, SFx)	0x00000000
0x110F	REGF_S[n]	REGF Register File (PEy) Data Registers (Sx, SFx)	0x00000000
0x1110	REGF_MRF	REGF Multiplier Results (PEx) Foreground Register	0xDEADD0D0
0x1120	REGF_MR0F	REGF Multiplier Results 0 (PEx) Foreground Register	0x00000000
0x1130	REGF_MR1F	REGF Multiplier Results 1 (PEx) Foreground Register	0x00000000
0x1140	REGF_MR2F	REGF Multiplier Results 2 (PEx) Foreground Register	0x00000000
0x1150	REGF_MSF	REGF Multiplier Results (PEy) Foreground Register	0xDEADD0D0
0x1160	REGF_MS0F	REGF Multiplier Results 0 (PEy) Foreground Register	0x00000000
0x1170	REGF_MS1F	REGF Multiplier Results 1 (PEy) Foreground Register	0x00000000
0x1180	REGF_MS2F	REGF Multiplier Results 2 (PEy) Foreground Register	0x00000000
0x1190	REGF_MRB	REGF Multiplier Results (PEx) Background Register	0xDEADD0D0
0x11A0	REGF_MR0B	REGF Multiplier Results 0 (PEx) Background Register	0x00000000
0x11B0	REGF_MR1B	REGF Multiplier Results 1 (PEx) Background Register	0x00000000
0x11C0	REGF_MR2B	REGF Multiplier Results 2 (PEx) Background Register	0x00000000
0x11D0	REGF_MSB	REGF Multiplier Results (PEy) Background Register	0xDEADD0D0
0x11E0	REGF_MS0B	REGF Multiplier Results 0 (PEy) Background Register	0x00000000
0x11F0	REGF_MS1B	REGF Multiplier Results 1 (PEy) Background Register	0x00000000
0x1200	REGF_MS2B	REGF Multiplier Results 2 (PEy) Background Register	0x00000000
0x1210	REGF_PX	REGF PMD-DMD Bus Exchange Register	0x00000000
0x1220	REGF_PX1	REGF PMD-DMD Bus Exchange 1 Register	0x00000000
0x1230	REGF_PX2	REGF PMD-DMD Bus Exchange 2 Register	0x00000000
0x1240	REGF_ASTATX	REGF Arithmetic Status (PEx) Register	0x00000000
0x1250	REGF_ASTATY	REGF Arithmetic Status (PEy) Register	0x00000000
0x1260	REGF_STKYX	REGF Sticky Status (PEx) Register	0x00000000
0x1270	REGF_STKYY	REGF Sticky Status (PEy) Register	0x00000000
0x1280	REGF_USTAT1	REGF User-Defined Status 1 Register	0x00000000
0x1290	REGF_USTAT2	REGF User-Defined Status 2 Register	0x00000000

Table 35-2: SHARC-PLUS REGF MMR Register Addresses (Continued)

Memory Mapped Address	Register Name	Description	Reset Value
0x12A0	REGF_USTAT3	REGF User-Defined Status 3 Register	0x00000000
0x12B0	REGF_USTAT4	REGF User-Defined Status 4 Register	0x00000000
0x12C0	REGF_I[n]	REGF Index Registers	0x00000000
0x12C1	REGF_I[n]	REGF Index Registers	0x00000000
0x12C2	REGF_I[n]	REGF Index Registers	0x00000000
0x12C3	REGF_I[n]	REGF Index Registers	0x00000000
0x12C4	REGF_I[n]	REGF Index Registers	0x00000000
0x12C5	REGF_I[n]	REGF Index Registers	0x00000000
0x12C6	REGF_I[n]	REGF Index Registers	0x00000000
0x12C7	REGF_I[n]	REGF Index Registers	0x00000000
0x12C8	REGF_I[n]	REGF Index Registers	0x00000000
0x12C9	REGF_I[n]	REGF Index Registers	0x00000000
0x12CA	REGF_I[n]	REGF Index Registers	0x00000000
0x12CB	REGF_I[n]	REGF Index Registers	0x00000000
0x12CC	REGF_I[n]	REGF Index Registers	0x00000000
0x12CD	REGF_I[n]	REGF Index Registers	0x00000000
0x12CE	REGF_I[n]	REGF Index Registers	0x00000000
0x12CF	REGF_I[n]	REGF Index Registers	0x00000000
0x12D0	REGF_M[n]	REGF Modify Registers	0x00000000
0x12D1	REGF_M[n]	REGF Modify Registers	0x00000000
0x12D2	REGF_M[n]	REGF Modify Registers	0x00000000
0x12D3	REGF_M[n]	REGF Modify Registers	0x00000000
0x12D4	REGF_M[n]	REGF Modify Registers	0x00000000
0x12D5	REGF_M[n]	REGF Modify Registers	0x00000000
0x12D6	REGF_M[n]	REGF Modify Registers	0x00000000
0x12D7	REGF_M[n]	REGF Modify Registers	0x00000000
0x12D8	REGF_M[n]	REGF Modify Registers	0x00000000
0x12D9	REGF_M[n]	REGF Modify Registers	0x00000000
0x12DA	REGF_M[n]	REGF Modify Registers	0x00000000
0x12DB	REGF_M[n]	REGF Modify Registers	0x00000000
0x12DC	REGF_M[n]	REGF Modify Registers	0x00000000

Table 35-2: SHARC-PLUS REGF MMR Register Addresses (Continued)

Memory Mapped Address	Register Name	Description	Reset Value
0x12DD	REGF_M[n]	REGF Modify Registers	0x00000000
0x12DE	REGF_M[n]	REGF Modify Registers	0x00000000
0x12DF	REGF_M[n]	REGF Modify Registers	0x00000000
0x12E0	REGF_L[n]	REGF Length (Circular Buffer) Registers	0x00000000
0x12E1	REGF_L[n]	REGF Length (Circular Buffer) Registers	0x00000000
0x12E2	REGF_L[n]	REGF Length (Circular Buffer) Registers	0x00000000
0x12E3	REGF_L[n]	REGF Length (Circular Buffer) Registers	0x00000000
0x12E4	REGF_L[n]	REGF Length (Circular Buffer) Registers	0x00000000
0x12E5	REGF_L[n]	REGF Length (Circular Buffer) Registers	0x00000000
0x12E6	REGF_L[n]	REGF Length (Circular Buffer) Registers	0x00000000
0x12E7	REGF_L[n]	REGF Length (Circular Buffer) Registers	0x00000000
0x12E8	REGF_L[n]	REGF Length (Circular Buffer) Registers	0x00000000
0x12E9	REGF_L[n]	REGF Length (Circular Buffer) Registers	0x00000000
0x12EA	REGF_L[n]	REGF Length (Circular Buffer) Registers	0x00000000
0x12EB	REGF_L[n]	REGF Length (Circular Buffer) Registers	0x00000000
0x12EC	REGF_L[n]	REGF Length (Circular Buffer) Registers	0x00000000
0x12ED	REGF_L[n]	REGF Length (Circular Buffer) Registers	0x00000000
0x12EE	REGF_L[n]	REGF Length (Circular Buffer) Registers	0x00000000
0x12EF	REGF_L[n]	REGF Length (Circular Buffer) Registers	0x00000000
0x12F0	REGF_B[n]	REGF Base (Circular Buffer) Registers	0x00000000
0x12F1	REGF_B[n]	REGF Base (Circular Buffer) Registers	0x00000000
0x12F2	REGF_B[n]	REGF Base (Circular Buffer) Registers	0x00000000
0x12F3	REGF_B[n]	REGF Base (Circular Buffer) Registers	0x00000000
0x12F4	REGF_B[n]	REGF Base (Circular Buffer) Registers	0x00000000
0x12F5	REGF_B[n]	REGF Base (Circular Buffer) Registers	0x00000000
0x12F6	REGF_B[n]	REGF Base (Circular Buffer) Registers	0x00000000
0x12F7	REGF_B[n]	REGF Base (Circular Buffer) Registers	0x00000000
0x12F8	REGF_B[n]	REGF Base (Circular Buffer) Registers	0x00000000
0x12F9	REGF_B[n]	REGF Base (Circular Buffer) Registers	0x00000000
0x12FA	REGF_B[n]	REGF Base (Circular Buffer) Registers	0x00000000
0x12FB	REGF_B[n]	REGF Base (Circular Buffer) Registers	0x00000000

Table 35-2: SHARC-PLUS REGF MMR Register Addresses (Continued)

Memory Mapped Address	Register Name	Description	Reset Value
0x12FC	REGF_B[n]	REGF Base (Circular Buffer) Registers	0x00000000
0x12FD	REGF_B[n]	REGF Base (Circular Buffer) Registers	0x00000000
0x12FE	REGF_B[n]	REGF Base (Circular Buffer) Registers	0x00000000
0x12FF	REGF_B[n]	REGF Base (Circular Buffer) Registers	0x00000000
0x1300	REGF_FLAGS	REGF Flag I/O Register	0x00000000
0x1310	REGF_IRPTL	REGF Interrupt Latch Register	0x00000000
0x1320	REGF_IMASK	REGF Interrupt Mask Register	0x00000000
0x1330	REGF_IMASKP	REGF Interrupt Mask Pointer Register	0x00000000
0x1340	REGF_EMUCLK	REGF Emulation Counter Register	0x00000000
0x1350	REGF_EMUCLK2	REGF Emulation Counter Register 2	0x00000000

Table 35-3: SHARC-PLUS SHBTB MMR Register Addresses

Memory Mapped Address	Register Name	Description	Reset Value
0x31400	SHBTB_CFG	SHBTB Configuration Register	0x00000000
0x31401	SHBTB_LOCK_START	SHBTB Lock Range Start Register	0x00000000
0x31402	SHBTB_LOCK_END	SHBTB Lock Range End Register	0x00000000

Table 35-4: SHARC-PLUS SHDBG MMR Register Addresses

Memory Mapped Address	Register Name	Description	Reset Value
0x30021	SHDBG_BRKSTAT	SHDBG Break Status Register	0x00000000
0x30023	SHDBG_OSPID	SHDBG O/S Processor ID Register	0x00000000
0x30025	SHDBG_BRKCTL	SHDBG Break Control Register	0x00000000
0x30026	SHDBG_REVID	SHDBG ID Code Register	0x00000000
0x300A0	SHDBG_PSA1S	SHDBG Program Sequence Address 1 Start Register	0x00000000
0x300A1	SHDBG_PSA1E	SHDBG Program Sequence Address 1 End Register	0x00000000
0x300A2	SHDBG_PSA2S	SHDBG Program Sequence Address 2 Start Register	0x00000000
0x300A3	SHDBG_PSA2E	SHDBG Program Sequence Address 2 End Register	0x00000000
0x300A4	SHDBG_PSA3S	SHDBG Program Sequence Address 3 Start Register	0x00000000
0x300A5	SHDBG_PSA3E	SHDBG Program Sequence Address 3 End Register	0x00000000
0x300A6	SHDBG_PSA4S	SHDBG Program Sequence Address 4 Start Register	0x00000000

Table 35-4: SHARC-PLUS SHDBG MMR Register Addresses (Continued)

Memory Mapped Address	Register Name	Description	Reset Value
0x300A7	SHDBG_PSA4E	SHDBG Program Sequence Address 4 End Register	0x00000000
0x300AE	SHDBG_EMUN	SHDBG Emulator Number (BP Hits) Register	0x00000000
0x300B2	SHDBG_DMA1S	SHDBG DM Data Address 1 Start Register	0x00000000
0x300B3	SHDBG_DMA1E	SHDBG DM Data Address 1 End Register	0x00000000
0x300B4	SHDBG_DMA2S	SHDBG DM Data Address 2 Start Register	0x00000000
0x300B5	SHDBG_DMA2E	SHDBG DM Data Address 2 End Register	0x00000000
0x300B8	SHDBG_PMDAS	SHDBG PM Data Address 1 Start Register	0x00000000
0x300B9	SHDBG_PMDAE	SHDBG PM Data Address 1 End Register	0x00000000
0x300E0	SHDBG_F1ADDR	SHDBG Fetch 1 Stage Address Register	0x00000000
0x300E1	SHDBG_F2ADDR	SHDBG Fetch 2 Stage Address Register	0x00000000
0x300E2	SHDBG_F3ADDR	SHDBG Fetch 3 Stage Address Register	0x00000000
0x300E3	SHDBG_F4ADDR	SHDBG Fetch 4 Stage Address Register	0x00000000
0x300E4	SHDBG_D1ADDR	SHDBG Decode 1 Stage Address Register	0x00000000
0x300E5	SHDBG_D2ADDR	SHDBG Decode 2 Stage Address Register	0x00000000
0x300E6	SHDBG_M1ADDR	SHDBG Memory 1 Stage Address Register	0x00000000
0x300E7	SHDBG_M2ADDR	SHDBG Memory 2 Stage Address Register	0x00000000
0x300E8	SHDBG_M3ADDR	SHDBG Memory 3 Stage Address Register	0x00000000
0x300E9	SHDBG_M4ADDR	SHDBG Memory 4 Stage Address Register	0x00000000
0x300EA	SHDBG_E2ADDR	SHDBG Execute 2 Stage Address Register	0x00000000
0x300EB	SHDBG_SECI_ID	SHDBG SEC Interrupt ID Register	0x00000000
0x300EC	SHDBG_DBGREG_ILLOP	SHDBG Illegal Opcode Detected Register	0x00000000
0x300ED	SHDBG_CORE_ID	SHDBG Core ID Register	0x00000000

Table 35-5: SHARC-PLUS SHL1C MMR Register Addresses

Memory Mapped Address	Register Name	Description	Reset Value
0x3E000	SHL1C_CFG	SHL1C L1 Cache Configuration 1 Register	0x00000000
0x3E002	SHL1C_CFG2	SHL1C Range Register Functionality Selection Register	0x00000000
0x3E010	SHL1C_RANGE_START0	SHL1C Range Start 0 (Inv, WB, WBI, and Lock) Register	0x00000000
0x3E011	SHL1C_RANGE_END0	SHL1C Range End 0 (Inv, WB, WBI, and Lock) Register	0x00000000
0x3E012	SHL1C_RANGE_START1	SHL1C Range Start 1 (Inv, WB, WBI, and Lock) Register	0x00000000

Table 35-5: SHARC-PLUS SHL1C MMR Register Addresses (Continued)

Memory Mapped Address	Register Name	Description	Reset Value
0x3E013	SHL1C_RANGE_END1	SHL1C Range End 1 (Inv, WB, WBI, and Lock) Register	0x00000000
0x3E014	SHL1C_RANGE_START2	SHL1C Range Start 2 (Non-cacheable and Lock) Register	0x00000000
0x3E015	SHL1C_RANGE_END2	SHL1C Range End 2 (Non-cacheable and Lock) Register	0x00000000
0x3E016	SHL1C_RANGE_START3	SHL1C Range Start 3 (Non-cacheable and Lock) Register	0x00000000
0x3E017	SHL1C_RANGE_END3	SHL1C Range End 3 (Non-cacheable and Lock) Register	0x00000000
0x3E018	SHL1C_RANGE_START4	SHL1C Range Start 4 (Non-cacheable and Write Through) Register	0x00000000
0x3E019	SHL1C_RANGE_END4	SHL1C Range End 4 (Non-cacheable and Write Through) Register	0x00000000
0x3E01A	SHL1C_RANGE_START5	SHL1C Range Start 5 (Non-cacheable and Write Through) Register	0x00000000
0x3E01B	SHL1C_RANGE_END5	SHL1C Range End 5 (Non-cacheable and Write Through) Register	0x00000000
0x3E01C	SHL1C_RANGE_START6	SHL1C Range Start 6 (Non-cacheable and Write Through) Register	0x00000000
0x3E01D	SHL1C_RANGE_END6	SHL1C Range End 6 (Non-cacheable and Write Through) Register	0x00000000
0x3E01E	SHL1C_RANGE_START7	SHL1C Range Start 7 (Non-cacheable and Write Through) Register	0x00000000
0x3E01F	SHL1C_RANGE_END7	SHL1C Range End 7 (Non-cacheable and Write Through) Register	0x00000000
0x3E020	SHL1C_INV_IXSTART0	SHL1C Invalidation/Write Back Index Start 0 Register	0x00000000
0x3E021	SHL1C_INV_CNT0	SHL1C Invalidation/Write Back Count 0 Register	0x00000000

Table 35-6: SHARC-PLUS PFM MMR Register Addresses

Memory Mapped Address	Register Name	Description	Reset Value
0x30200	PFM_CFG	PFM Configuration Register	0x00000000
0x30203	PFM_CNTR3	PFM Counter 3 Register	0x00000000
0x30204	PFM_CNTR4	PFM Counter 4 Register	0x00000000
0x30205	PFM_CNTR5	PFM Counter 5 Register	0x00000000
0x30206	PFM_CNTR6	PFM Counter 6 Register	0x00000000
0x3020D	PFM_CNTR3START	PFM Counter 3 Start Register	0x00000000
0x3020E	PFM_CNTR3PAUSE	PFM Counter 3 Pause Register	0x00000000

Table 35-6: SHARC-PLUS PFM MMR Register Addresses (Continued)

Memory Mapped Address	Register Name	Description	Reset Value
0x3020F	PFM_CNTR3CLR	PFM Counter 3 Clear Register	0x00000000
0x30210	PFM_CNTR4START	PFM Counter 4 Start Register	0x00000000
0x30211	PFM_CNTR4PAUSE	PFM Counter 4 Pause Register	0x00000000
0x30212	PFM_CNTR4CLR	PFM Counter 4 Clear Register	0x00000000
0x30213	PFM_CNTR5START	PFM Counter 5 Start Register	0x00000000
0x30214	PFM_CNTR5PAUSE	PFM Counter 5 Pause Register	0x00000000
0x30215	PFM_CNTR5CLR	PFM Counter 5 Clear Register	0x00000000
0x30216	PFM_CNTR6START	PFM Counter 6 Start Register	0x00000000
0x30217	PFM_CNTR6PAUSE	PFM Counter 6 Pause Register	0x00000000
0x30218	PFM_CNTR6CLR	PFM Counter 6 Clear Register	0x00000000

36 Glossary

To make the best use of the FFTA, it is useful to understand the following terms.

Alternate Registers

See index registers in I/O Processor Register.

Arithmetic Logic Unit (ALU)

This part of a processing element performs arithmetic and logic operations on fixed-point and floatingpoint data.

Arm

The ADSP-ADSP-SC58x Processor includes an Arm® Cortex-A5®core. The Arm Cortex-A5 processor is the smallest, lowest cost and lowest power Armv7 application processor.

Asynchronous Transfers

Communications in which data can be transmitted intermittently rather than in a steady stream.

Barrel Shifter

This part of a processing element completes logical shifts, arithmetic shifts, bit manipulation, field deposit, and field extraction operations on 32-bit operands. Also, the shifter can derive exponents.

Base Address

The starting address of a circular buffer to which the DAG wraps around. This address is stored in a DAG Bx register.

Base Register

A base (Bx) register is a data address generator (DAG) register that sets up the starting address for a circular buffer.

Bit-Reverse Addressing

The data address generator (DAG) provides a bit-reversed address during a data move without reversing the stored address.

Boot Modes

The boot mode determines how the processor starts up (loads its initial code). The processors can boot from various sources based on the BMODE pins.

Branch Predictor

The branch predictor unit examines each fetch address to determine whether it is a branch instruction. If the unit detects a branch instruction, the unit provides an address of the likely next instruction. If no conditions require otherwise, the processor fetches and executes instructions from memory in sequential order.

Branch Target Buffer

Implementation of a hardware-based branch predictor (BP) and branch target buffer (BTB) reduce branch delay. The program sequencer supports efficient branching using this branch target buffer (BTB) for conditional and unconditional instructions.

Broadcast Data Moves

The data address generator (DAG) performs dual data moves to complementary registers in each processing element to support SIMD mode.

Cache Entry

The smallest unit of memory that is transferred to/from the next level of memory from/to a cache as a result of a cache miss.

Cache Hit

A memory access that is satisfied by a valid, present entry in the cache.

Cache Miss

A memory access that does not match any valid entry in the cache.

Circular Buffer Addressing

The DAG uses the Ix, Mx and Lx register settings to constrain addressing to a range of addresses. This range contains data that the DAG steps through repeatedly, "wrapping around" to repeat stepping through the range of addresses in a circular pattern.

CCES

CrossCore® Embedded Studio (CCES) integrated development environment is the preferred programming tool set for SHARC processors.

Companding (Compressing/Expanding)

This is the process of logarithmically encoding and decoding data to minimize the number of bits that must be sent by the SPORTs.

Conditional Branches

These are JUMP or CALL/return instructions whose execution is based on testing an IF condition.

SHARC+ Core

The SHARC+ core is an SoC in the SHARC processor and consists of these functional blocks: SIMD Processing units, dual DAGs, instruction sequencer, interrupt controller, loop controller, core timer, conflict cache and debug/emulation interface.

Core memory-mapped registers (CMMR) are located in the core clock domain and accessed via an address. These registers control sytem, BTB, I/D cache, debug and monitor

Register File Complementary Data (CDreg).

These are registers in the PEy processing element. These registers are hold operands for multiplier, ALU, or shifter operations and are denoted as Sx when used for fixed point operations or SFx when used for floating-point operations.

Complementary Universal Registers (CUreg)

These are any core registers (data registers), any data address generator (DAG) registers, used in SIMD mode.

Data Address Generator (DAG)

The data address generators (DAGs) provide memory addresses when data is transferred between memory and registers.

Register File Data (Dreg)

These are registers in the PEx processing element. These registers are hold operands for multiplier, ALU, or shifter operations and are denoted as Rx when used for fixed point operations or Fx when used for floating-point operations.

Delayed Branches

In JUMP and CALL instructions that use the delayed branch (DB) modifier, one instruction cycle is lost in the instruction pipeline. This is because the processor executes the two instructions after the branch and the third is aborted while the instruction pipeline fills with instructions from the new location.

Denormal Operands

When the biased exponent is zero, smaller floating-point numbers can only be represented by making the integer bit (and perhaps other leading bits) of the significant zero. The numbers in this range are called denormalized (or tiny) numbers. The use of leading zeros with denormalized numbers allows smaller numbers to be represented.

Direct Branches

These are JUMP or CALL instructions that use an absolute-not changing at runtime-address (such as a program label) or use a PC-relative address.

DMA (Direct Memory Accessing)

The processor supports DMA of data between processor memory and external memory, or peripherals. Each DMA operation transfers an entire block of data.

DMA Chaining

The processor supports chaining together multiple DMA sequences. In chained DMA, the DMA loads the next transfer control block (DMA parameters) into the DMA parameter registers when the current DMA finishes and auto-initializes the next DMA sequence.

DMA Parameter Registers

These registers function similarly to data address generator registers, setting up a memory access process. These registers include internal index registers, internal modify registers, count registers, chain pointer registers, external index registers, external modify registers, and external count registers.

DMA TCB Chain Loading

This is the process that the DMA uses for loading the TCB of the next DMA sequence into the parameter registers during chained DMA. This term is also know as a DMA descriptor.

Double-Precision Floating-Point (64-bit)

IEEE Standard 754-2008 specifies a binary64 floating-point (Also known as double-precision floating-point in IEEE Standard 754-1985) format. A number represented in this format consists of a sign bit s, an 11-bit Exponent e and a 53-bit mantissa. For normalized numbers, the mantissa consists of a 52-bit fraction f and a “hidden” bit 1 that is implicitly presumed to precede bit-51. The binary point is presumed to reside between this hidden bit and bit-51.

E-2 Active loops

Zero-overhead loop where loop counter decrement and check of termination condition occurs in the E2 pipeline stage.

Edge-Sensitive Interrupt

The processor detects this type of interrupt if the input signal is high (inactive) on one cycle and low (active) on the next cycle when sampled on the rising edge of clock.

Endian Format, Little Versus Big

The processor uses big-endian format-moves data starting with most-significant-bit and finishing with least-significant-bit-in almost all instances. There are some exceptions (such as serial port operations) which provide both little-endian and big-endian format support to ensure their compatibility with different devices.

With byte-addressing, a normal-word load from a byte-address loads in little-endian format. A LW load to an even register in normal-word address space also loads little-endian. When the compiler is used with the -char-size-32 command line option, it uses little endian. Note that the Arm core also uses little endian.

Explicit Versus Implicit Operations.

In SIMD mode, identical instructions execute on the PEx and PEy computational units; the difference is the data. The data registers for PEy operations are identified (implicitly) from the PEx registers in the instruction. This implicit relation between PEx and PEy data registers corresponds to complementary register pairs.

F-1 Active Loop

Zero-overhead loop where loop counter decrement and check of termination condition occurs in the F1 pipeline stage.

Field Deposit (Fdep) Instructions

These shifter instructions take a group of bits from the input register (starting at the LSB of the 32-bit integer field) and deposit the bits as directed anywhere within the result register.

Field Extract (Fext) Instructions

These shifter extract a group of bits as directed from anywhere within the input register and place them in the result register (aligned with the LSB of the 32-bit integer field).

FIFO (First In, First Out)

A hardware buffer or data structure from which items are taken out in the same order they were put in.

Flag Pins (Programmable)

Flag pins can be programmed as input or output pins using bit settings in the FLAGS register. The status of the flag pins is also given in the GPIO PORT register.

Flag Update

The processor's update to status flags occurs at the end of the cycle in which the status is generated and is available on the next cycle.

General-Purpose Input/Output Pins

See programmable flag pins.

Harvard Architecture

Processor's use memory architectures that have separate buses for program and data storage. The two buses let the processor get a data word and an instruction simultaneously.

IDLE

An instruction that causes the processor to cease operations, holding its current state until an interrupt occurs. Then, the processor services the interrupt and continues normal execution.

Index Registers

An index register is a data address generator (DAG) register that holds an address and acts as a pointer to memory.

Indirect Branches

These are JUMP or CALL instructions that use a dynamic-changes at runtime-address that comes from the PM data address generator.

Inexact Flags

An exception flag whose bit position is inexact.

Input Clock

Device that generates a steady stream of timing signals to provide the frequency, duty cycle, and stability to allow accurate internal clock multiplication via the phase locked loop (PLL) module.

Interleaved Data

SIMD mode requires a special memory layout since the implicit modifier is 1 or 2 based on NW or SW addresses. This then requires data to be in an interleaved organization in the memory layout.

Internal Memory Address Space

Internal memory space refers to the processor's on-chip SRAM L1 blocks.

Internal Memory Interface (IMIF)

The SoC has a central logic which controls all busses (crossbar) to the internal memory blocks from the different sources (SHARC core vs DMA vs co-processor).

Instruction Set Architecture (ISA)

48-bit Instruction Set Architecture, supported by all SHARC processors.

Interrupts

Subroutines in which a runtime event (not an instruction) triggers the execution of the routine.

IVT

The SHARC+ core has an Interrupt Vector Table with 256x48 SRAM locations to serve/control core and SEC based interrupts. The IVT is located after reset in L2 memory. It may be allocated to L2/L3 mem based on the SYSCTL registers.

JTAG Port

This port supports the IEEE standard 1149.1 Joint Test Action Group (JTAG) standard for system test. This standard defines a method for serially scanning the I/O status of each component in a system. This interface is also used for processor debug.

Jumps

Program flow transfers permanently to another part of program memory.

Latency

Latency of memory access is the time between when an address is posted on the address bus and the core receives data on the corresponding data bus.

Length Registers

A length register is a data address generator (DAG) register that sets up the range of addresses a circular buffer.

Level-Sensitive Interrupts

The processor detects this type of interrupt if the signal input is low (active) when sampled on the rising edge of clock.

Loops (zero overhead)

One sequence of instructions executes several times with zero overhead.

Memory Blocks and Banks

The processor's internal memory is divided into blocks that are each associated with different data address generators. The processor's external memory spaces is divided into banks, which may be addressed by either data address generator.

Modified Addressing

The DAG generates an address that is incremented by a value or a register.

Modify Instruction

The data address generator (DAG) increments the stored address without performing a data move.

Modify Registers

A modify register is a data address generator (DAG) register that provides the increment or step size by which an index register is pre- or post-modified during a register move.

Multifunction Computations

Using the many parallel data paths within its computational units, the processor supports parallel execution of multiple computational instructions. These instructions complete in a single cycle, and they combine parallel operation of the multiplier and the ALU or dual ALU functions. The multiple operations perform the same as if they were in corresponding single-function computations.

Multiplier

This part of a processing element does floating-point and fixed-point multiplication and executes fixedpoint multiply/add and multiply/subtract operations.

Neighbor Data Registers

In long word addressed accesses, the processor moves data to or from two neighboring data registers. The least-significant-32 bits moves to or from the explicit (named) register in the neighbor register pair. In forced long word accesses (normal word address with LW mnemonic), the processor converts the normal word address to long word, placing the even normal word location in the explicit register and the odd normal word location in the other register in the neighbor pair.

Nonzero numbers

Nonzero, finite numbers are divided into two classes: normalized and denormalized.

Normal Word 2colunm/3column

The internal memory supports 4x16-bit maximum width which represents a long word (4columns). It can also control 2x16-bit data which is a normal word (2columns). Another option is 3x16 data control which is also a normal word (3columns). The IMDW bit (SYSCTL) bit controls about 2/3column normal word DAG access per block.

Peripherals

This refers to everything outside the processor core. The peripherals include internal memory, parallel port, I/O processor, JTAG port, and any external devices that connect to the processor. Detailed information about the peripherals is found in the product-specific hardware reference.

Phase Locked Loop (PLL)

An on-chip frequency synthesizer that produces a full speed clock from a lower frequency input clock signal.

Post-Modify Addressing

The data address generator (DAG) provides an address during a data move and auto-increments the stored address for the next move.

Precision.

The precision of a floating-point number depends on the number of bits after the binary point in the storage format for the number. The processor supports two high precision floating-point formats: 32-bit IEEE single-precision floating-point (which uses 8 bits for the exponent and 24 bits for the mantissa) and a 40-bit extended precision version of the IEEE format plus an IEEE double-precision format.

Pre-Modify Addressing

The data address generator (DAG) provides a modified address during a data move without incrementing the stored address.

Register File Registers

This is the set of all core registers accessed directly by an instruction.

Register Swaps

This special type of register-to-register move instruction uses the special swap operator, <->. A registerto- register swap occurs when registers in different processing elements exchange values.

Saturation (ALU Saturation Mode)

In this mode, all positive fixed-point overflows return the maximum positive fixed-point number (0x7FFF FFFF), and all negative overflows return the maximum negative number (0x8000 0000).

SHARC processor

The SHARC processor is SoC based on the SHARC+ core + internal memory I/F, the L1 memory blocks, the Instruction/data caches + two requester and two completer ports for communication to the system fabric.

SIMD (Single-Instruction, Multiple-Data)

SIMD mode of SHARC+ core provides mechanism to perform dual identical compute and/or data moves. This can result in upto 2x performance improvement on any operation. This mode is very effective if intended operation can be split into two perfectly identical sequences. However in many applications, finding perfectly identical sequence is not possible but still major part can be parallelized. In such cases, effectiveness of SIMD mode reduces as it requires switching off and on of SIMD mode and moving data to-and-fro from the second processing element (PEy) to the primary processing element (PEx) to perform non-identical part of the application. This is because, non-identical part of code requires SISD processing and that is possible only in PEx. This change of mode and data movement reduces the effectiveness of SIMD. This affects both compilers as well assembly level programmers.

SISD (Single-Instruction, Single-Data)

A computer architecture or processor mode in which an instruction processes single data elements at a time. Contrast with SIMD.

Completer Ports

The SHARC SoC processors can be a bus completer to other processors. The current SoC post the address to the completer ports (core or DMA) to access the local memory of the completer.

Stack, hardware

A data structure for storing items that are to be accessed in last in, first out (LIFO) order. When a data item is added to the stack, it is "pushed"; when a data item is removed from the stack, it is "popped."

Subroutines

The processor temporarily interrupts sequential flow to execute instructions from another part of program memory.

Stalls

The time spent waiting for an operation to take place. It may refer to a variable length of time a program has to wait before it can be processed, or to a fixed duration of time, such as a machine cycle. When memory is too slow to respond to the CPU's request for it, wait states are introduced until the memory can catch up.

System Clock (SYSCLK)

The system clock (SYSCLK) controls the processor's system fabric and is defined as (system clock) Clock Period = 2 tCCLK.

System Clock (SCLK0/1)

System Clock (SCLK0/1) are output clocks provided to the peripheral modules.

Three-State Versus Tri-state

Analog Devices documentation uses the term "three-state" instead of "tri-state" because Tri-state is a trademarked term, which is owned by National Semiconductor.

Universal Registers (Ureg).

These are any processing element registers (data registers), any data address generator (DAG) registers, any program sequencer registers.

Variable Instruction Set Architecture

Variable 48/32/16-bit Instruction Set Architecture (supported upon 214xx SHARC products) Also called non-VISA or compressed instruction set.

Von Neumann Architecture

This is the architecture used by most (non-processor) microprocessors. This architecture uses a single address and data bus for memory access.

Wait States

See Stalls

Index

Symbols

A

B

C

D

J

L

M

N

O

P

R

T

U

V

W

Z