SC3900FP Flexible Vector Processor (FVP) Core Reference Manual

SC3900FPCRM
Rev. C
July 2014

freescale™

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number SC3900FPCRM
Revision C, 7/2014

freescale™

Contents

Paragraph Number | Title | Page Number

Chapter 1
Introduction

Chapter 2
Programmer’s Model and Semantics

Contents

Paragraph Number | **Title** | **Page Number**

Contents

Paragraph Number	Title	Page Number

Chapter 3
Data Arithmetic Logic Unit (DALU)

Contents

Paragraph Number | Title | Page Number

Chapter 4
Address Generation and Memory Interface

Contents

Paragraph Number | **Title** | **Page Number**

Contents

Paragraph Number | Title | Page Number

Chapter 5
Program Control

Contents

Paragraph Number | Title | Page Number

Chapter 6 Encoding

Contents

Paragraph Number — Title — Page Number

Chapter 7
The Interlocked Pipeline

Contents

Chapter 8
Programming Rules and Guidelines

Contents

Paragraph Number | Title | Page Number

Chapter 9
Debug and Trace Support

Appendix A
Revision History

Appendix B Changes between SC3900 and SC3900FP

Contents

Appendix C
Instruction Set

Contents

Paragraph Number	Title	Page Number

Figures

Figures

Figure Number **Title** **Page Number**

Figures

Figures

Figure Number	Title	Page Number

Tables

Tables

Table Number	Title	Page Number

Tables

Tables

Table Number	Title	Page Number

About This Book

This document describes the SC3900FP StarCore architecture. Although the architecture is mostly finalized, the document is still considered preliminary and may change without notice as errors are corrected and architectural adjustments take place.

Audience

WARNING

> The document can be shared outside only with explicit approval from NMG marketing, and under NDA.

This document is intended for technical teams from selected customers and for an internal Freescale audience, which includes system software developers, hardware designers, software tool developers and application developers.

Conventions

This document uses certain conventions to assist in identifying, locating, and understanding information.

Table i list suffixes used to identify different numbering systems:

Table i. Suffix identification

Suffix	Meaning
b	Binary number. For example, the binary equivalent of the number 5 is written 101b.
h	Hexadecimal number. For example, the hexadecimal equivalent of the number 60 is written 3Ch.

Table ii lists the notational conventions used throughout this document:

Table ii. Notational conventions

Example	Description
placeholder	Items in italics are placeholders for information that you provide. Italicized text is also used for the titles of publications and for emphasis.
code	Fixed-width type indicates text that must be typed exactly as shown. It is used for instruction mnemonics, symbols, subcommands, parameters, and operators. Fixed-width type is also used for example code.
[*n*]	A number enclosed in brackets represents a single bit in a register or in memory.
[*n:m*]	Numbers enclosed in brackets and separated by a colon represent the endpoints of a continuous range of bits in a register or in memory.
SR.SCM register.field	The way to represent a field name of a particular register. For instance, SR.SCM is the Scaling Mode (SCM) field of the Status Register (SR).

Table iii list terms that have special meanings.

Table iii. Special terms

Term	Meaning
asserted	Refers to the state of a signal as follows: • An active-high signal is asserted when high (1). • An active-low signal is asserted when low (0).
byte	An 8-bit data object
deasserted	Refers to the state of a signal as follows: • An active-high signal is deasserted when low (0). • An active-low signal is deasserted when high (1).
quad-long word	A 128-bit data object
double-long word	A 64-bit data object
long word	A 32-bit data object
word	A 16-bit data object

Chapter 1
Introduction

The StarCore SC3900FP flexible vector processor (FVP) is the next generation StarCore DSP core, following the SC140 through SC3850 cores. As a flexible, programmable DSP core that handles compute-intensive communications applications, providing high performance but requiring low power, the SC3900FP addresses the key market needs of mainstream DSP applications. While it focuses on baseband wireless processing, it also targets other DSP markets such as video, voice, industrial, and military applications.

The SC3900FP core is an evolutionary enhancement of the SC3900 core, mainly adding floating point support. It is binary compatible with the SC3900 core. The list of architectural changes in the SC3900FP relative to the SC3000 is described in Appendix B, “Changes between SC3900 and SC3900FP”.

The SC3900 core family offers major improvements relative to previous StarCore generations in numerical capabilities, compiler capability for both DSP code and control code, and improves the energy consumption per task.

1.1 Target markets

The SC3900FP core fulfills the constantly increasing computational requirements of DSP applications, which are driven by the following concerns:

- An ever-increasing demand for higher performance and more product features
- Evolving communication standards and services
- Increasing demand for wideband channels and high data rates
- New user interfaces and media requirements

Target markets for the SC3900FP architecture include:

- Wireless base stations
- Wireless and wireline infrastructure
- Broadband wireless: GSM, 3G, LTE and LTE-Advanced
- Speech coding, synthesis, and voice recognition
- Wideband (music) and synthetic waveform coding
- Multimedia engines (video transcoding)
- Wireless internet and multimedia
- Network and data communication
- Large-scale multichannel access systems:
  - VoIP gateways
  - Video and voice transcoding

- Residential gateways

1.2 Core architecture features

Table 1-1 summarizes the key features of the StarCore SC3900FP flexible vector processor.

Table 1-1. Key features

Feature	Details
Main core resources	• A data arithmetic and logic unit (DALU) that includes four data multiplication units (DMU). Each unit can compute up to eight 16 × 16 multiplications, two complex multiplications, two 32 × 32 multiplications, or two IEEE single precision floating point multiplications. It can generate two 40-bit results every cycle. • An address generation and general computation unit (AGU) that includes two load-store units (LSU) and one integer processing unit (IPU) that includes a multiplier and a barrel shifter. • One program control unit (PCU), with sophisticated compiler aids—multipredicate bits model, a branch target buffer (BTB), smart prefetching, branch prediction and speculation. • Sixty-four 40-bit data registers with 8 guard bits, freely accessible by DALU instructions. • Thirty-two 32-bit general purpose registers, which can be used as address registers, freely accessible by address generation instructions. These registers are also used for integer calculation.
Data type support	• Byte (8-bit), word (16-bit), and long (32-bit) data widths, 20-bit and 40-bit accumulation are supported by instructions and memory moves. • Both integer (signed and unsigned) and fractional data types. • Single-precision floating point (32-bit) data type support. • Packed fractional and integer complex data type of both 20-bit and 40-bit. • Several packed data types (2 to 4 objects on the same register) for SIMD operations. • Explicit saturation and rounding support.
Instruction set	• 32-bit instruction set, expandable to 48-bit and 64-bit instructions, enable highly orthogonal, compiler friendly code. • Native, single-instruction support for complex and 32 × 32 bit multiplication. • High orthogonality of operands. Most instructions are defined as non-destructive (that is, destination is not one of the sources). • Rich instruction set for DSP code as well as general control operations. • A very good compiler target (substantially improved compared to previous generations) for DSP code as well as for control code.
Application-specific instructions for acceleration of the following algorithms	• Baseband operations • FFT • Viterbi • Video processing
Very high execution parallelism	• Up to eight instructions executed in a single clock cycle, statically scheduled, including —Up to four DALU operations, performing up to thirty-two multiply-accumulate (MAC) operations and generating eight 40-bit results in a single cycle. —Up to three integer and/or address calculations in a single cycle, including up to two 512 bit wide load/store operations and instructions for 32 × 32 multiplication and 32-bit shift. —One flow control instruction (change-of-flow, hardware looping, etc.), not taking an AGU execution slot. • Variable length execution set (VLES) execution model—any length from 16-bit up to 256-bit VLES is supported.

Table 1-1. Key features (continued)

Feature	Details
Very high numerical throughput for DSP operations	• Each of the four DMUs can perform eight 16 × 16 multiplications per cycle (total of thirty-two multiplications for all DMUs), which can be used to perform in a single instruction/cycle: —Two 32 × 32 multiplication —Two complex multiplication of (16 real, 16 image) × (16 real, 16 image) into (16 real, 16 image) results. —One complex dot product of ((16 real, 16 image) × (16 real, 16 image) + (16 real, 16 image) × (16 real, 16 image)) into (40 real, 40 image) result. —SIMD2 dot product acceleration—two (40 + (16 × 16) + (16 × 16) + (16 × 16) + (16 × 16)) into 40 calculations. —SIMD4 multiplication and accumulation into two 20-bit register portions. —Acceleration of extended precision multiplication (including 64-bit support). —Two single precision floating point multiplications. • Table 1-2 shows key performance summary metrics.
IEEE single-precision floating point support	• Each DMU can perform two single-precision floating point instruction per cycle (total of eight single precision floating point instruction for all DMUs). • The instructions supported include: —Floating point fused multiply-add instructions —Floating point arithmetic instructions: add, sub, absolute value —Floating point compare instructions —Floating point conversion instructions: including integer-to-float and float-to-integer —Floating point estimate instructions: reciprocal, square root, and log
High throughput memory interface	• Unified, 32-bit byte addressable memory space. • Dual Harvard architecture, with which it is possible to issue per cycle one 256-bit program access and one of the following: —Two 256-bit read data accesses —Two 256-bit write data accesses —One 512-bit read and one 512 write data accesses • Core to data caches throughput of up to 1.2 Tbit per second, at 1.2 GHz core frequency.
Powerful address generation model	• Zero overhead modulo arithmetic support for address pointers • Several addressing modes: absolute, register indirect, indexed, post-update, and more • Zero overhead linear address pre-calculation • Strong implicit unalignment support: Unaligned data accesses are supported for all types of load and store transactions. If an unaligned access crosses a 4 Kbyte-aligned boundary, a single hold cycle is generated; otherwise, the access has no penalty.
Advanced pipeline	• 13 stage, fully interlocked pipeline. • Zero load to use overhead and zero use to store overhead for the main DSP path of memory load to register, DALU operation such as a MAC, and result storage to memory (including scaling, rounding and limiting). • Speculation of conditionally executed instructions and change of flow execution paths. • Non blocking program and data memory stalls: —Instruction can continue executing including load/store while program memory is stalled. —Program memory fetch ahead can continue when load/store transactions are stalled or instructions are interlocked due to pipe hazards.

Table 1-1. Key features (continued)

Feature	Details
Control features	• Zero-overhead hardware loops with up to four levels of nesting. • A branch target buffer (BTB) for accelerating execution of change of flow instructions. • A powerful predication model: —Six predicate bits —Efficient predicate spilling mechanism virtually enabling up to 32 predicate bits —Rich set of instruction generating and manipulating predicate values —Fully predicated ISA with orthogonal predicated execution • Deep pre-fetch buffer • Enhanced function call mechanism •
OS support	• Precise memory exception support, for advanced OS. • Task oriented execution model. • Full support for memory protection and address translation in the off-core MMU. • Task, Exception, and Debug stack pointers for software stack support. • Low task switch overhead using wide stack save and restore instructions.
Rich set of real-time debug capabilities through the Debug and Trace Unit (DTU)	• Dedicated core instructions • One user-defined debug exception vector • A breakpoint unit, which includes: —Four general purpose event detection units, which can support up to 8 PC-only breakpoints/events, or up to 4 PC ranges, or up to 4 data addresses ranges (or their combinations) —One exception detector —A task ID comparators • A profiling unit, which contains: —Six event counters, organized in 2 triads, for profiling —A reloadable counters, for debug control and validating real-time intervals • Flexible event generation, which includes: —User-defined cross triggering logic, which connects all debug events to outcomes —Four state bits, with which the user can configure event state machines or sequence detectors • A trace unit, based on Nexus (IEEE ISTO 5001) standard, with an independent trace output bus, which supports the following: —Program flow trace —Hardware loop exact timing measurement —User defined trace messages (data acquisition), based on writing to dedicated core registers —Tracing of profiling counters
Low power design	• Clock gating to unused modules—such as, gate clocks to DALU while executing control code using the AGU • Low-power idle mode • Fully static logic • Support for hardware loop internal buffering, reducing the program bus access • Improved fetch ahead mechanism, reducing the program data footprint

Table 1-2 shows key performance summary metrics.

Table 1-2. Multiplication throughput summary figures for the SC3900FP

Operation	Precision	Instructions per operation	Result throughput (Four DMUs)
Real multiply	16 × 16	0.125	32
	16 × 32	0.25	16
	32 × 32	0.5	8
Complex multiply	16 × 16	0.5	8
	16 × 32	1	4
Floating point multiply	Single Precision	0.5	8

1.3 Typical SC3900FP DSP subsystem configuration

The SC3900FP is a high-performance, general-purpose, fixed-point and floating-point DSP core, allowing it to support many system-on-chip (SoC) configurations. The DSP core is embedded in a tightly coupled unit that is termed the “SC3900FP DSP subsystem.” The subsystem includes cache/memory modules, associated buffers, a memory management unit, an interrupt controller, debug modules, and more. It has standard interfaces that hide the high-frequency domain of the core from the SoC. A library of modules containing memories, peripherals, debug, interface and other units, as well as other core flavors makes it possible to build a DSP subsystem that addresses many applications and SoCs with different needs. It allows for a variety of modular and cost-effective SoC devices to be built around the core in a relatively short amount of time.

Figure 1-1 shows a typical DSP subsystem implementation incorporating a StarCore SC3900FP core. Although not indicated in this configuration, several such subsystems may be integrated together in a

SC3900FP cluster sharing an L2 cache. A typical SoC may contains more than one cluster, with shared resources such as M3 memories, L3 caches, accelerators (for example, MAPLE), and memory interfaces.

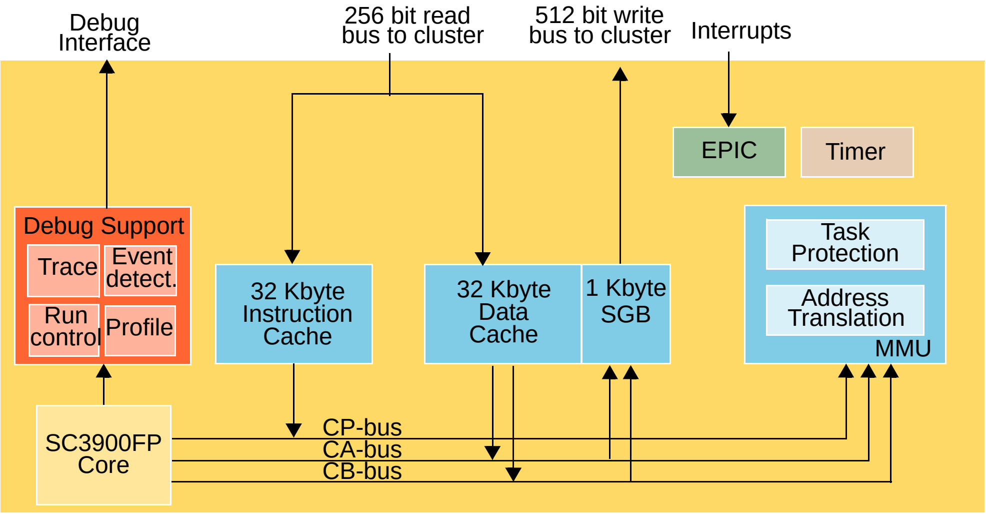


Figure 1-1. Block diagram of a typical SC3900FP DSP subsystem implementation

The SC3900FP supports the incorporation of the following components in an SC3900FP-based DSP subsystem:

Table 1-3. Components supported by SC3900FP

Component	Use
L1 program cache	The program-cache memory for the processor core is a relatively small memory, located close to the processor core, used to store VLESes from memory to reduce the access time to this memory and to increase the performance. The instruction channel is the unit group that includes the L1 program cache; the instruction channel is responsible for fetching, buffering, and interfacing the system.
L1 data cache	This cache holds the most currently required data in a relatively small memory near the processor core. When the processor core references data in the cache, the data is delivered without stalling the core. The cache memory is updated according to the processor core's data usage. If the data is not inside the cache, the processor core stalls so that the cache can be updated from memory. The data channel is the unit group that includes the L1 data cache; the data channel is responsible for fetching, buffering, and interfacing the system. The data cache may support hardware coherency with the higher level memory, if supported by these memories.
Memory management unit	This unit manages memory resources both inside and outside of the SC3900FP core-based subsystem. It also provides protection against unauthorized access of system devices from the core and translates a core virtual address to the actual physical address.
L2 cache interface	The L1 caches include an interface that enables seamless connection with a CoreNet compliant L2 cache. The L2 cache itself is not part of the subsystem, and is usually shared between a few SC3900FP subsystem in a SC3900FP cluster. This interface allows the memory subsystem to support full memory coherency, per the CoreNet definition.
Debug and profile support blocks	The core has a powerful internal support for debug and profiling, which enables a nonintrusive means of interacting with the processor core and its peripherals, such as various breakpoints, examine/update registers, and memory values. Additional low intrusive means enables real time snooping of memory locations. The debug and trace unit (DTU) is the subsystem block that supports these functions. This unit supports, among other things, tracing compliant with the Nexus IEEE-ISTO 5001 standard.

Table 1-3. Components supported by SC3900FP

Component	Use
Programmable interrupt controller	This controller monitors and prioritizes maskable interrupt requests. It also controls the interrupt vector that the core uses for both maskable and nonmaskable interrupts.
General purpose timers	These are the timers that can be used for real-time operating system (RTOS) support and for other needs.
System bus interfaces	These consist of a variety of interfaces—synchronous and asynchronous—in various protocols that interface the SC3900FP-based DSP subsystem with the SoC. The selection of the interface and the supported throughput are selected in a way that suits the needs of the SoC.

1.4 Core functional units and components

Figure 1-2 shows a block diagram of the SC3900FP core.

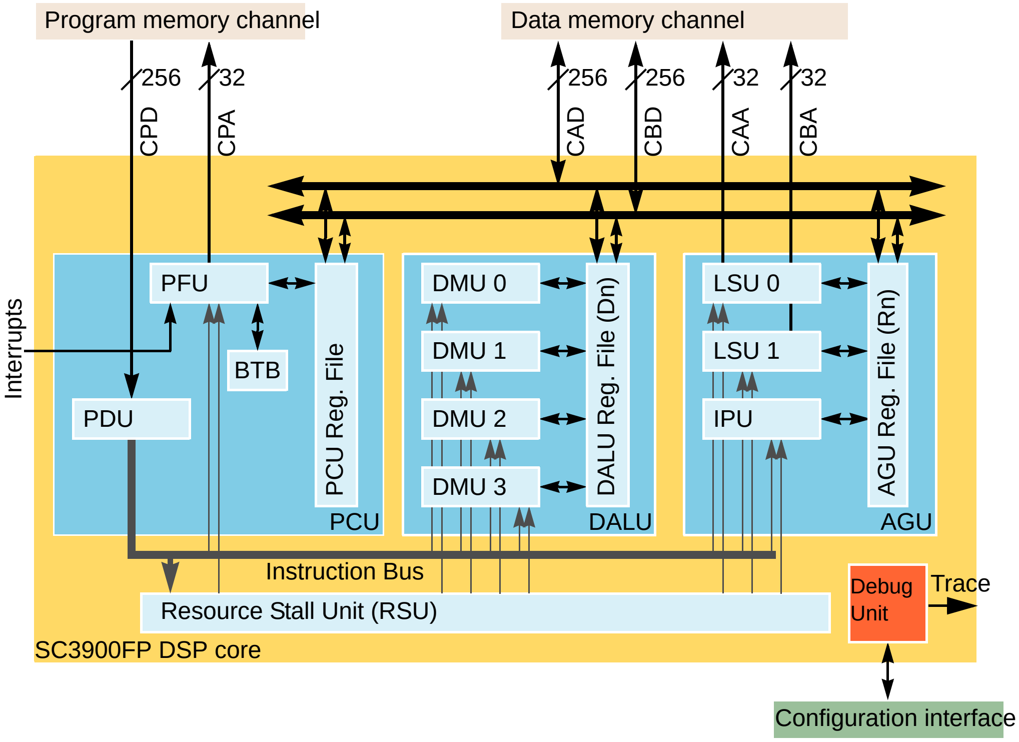


Figure 1-2. SC3900FP core block diagram

A StarCore SC3900FP flexible vector processor consists of the following functional units and components:

- A data arithmetic logic unit (DALU) that contains four instances of a data multiplication unit (DMU) and a DALU register file containing 64 40-bit data registers (Dn)
- An address generation and general execution unit (AGU) that contains the following:
  - — Two load/store units (LSU)
  - — One integer processing unit (IPU)

  - An AGU register file mainly containing 32 general purpose 32-bit registers (Rn) and the core control registers
- A program control unit (PCU) that contains the following:
  - A program fetch unit (PFU)
  - A branch target buffer (BTB)
  - Program dispatch unit (PDU)
- A resource stall unit (RSU), which enforces all pipe interlocks.
- A debug and trace unit (DTU), which supports the core and subsystem debug, profiling and trace features.

1.4.1 Data arithmetic logic unit (DALU)

The data arithmetic logic unit (DALU) performs DSP arithmetic and logical operations on data operands in the processor core. The components of the DALU are as follows:

- Four parallel data multiply units (DMU 0, DMU 1, DMU 2 and DMU 3), each containing eight 16 × 16 multiply-accumulate (MAC) units, adders, shifters, and so on, as well as floating point support and DSP application-specific instructions implementation. Each DMU can execute one DALU instruction per cycle, read up to three dual-register operands from the DALU register file, and write back single or dual registers to the DALU register file.
- A register file of 64 40-bit registers (Dn)

1.4.2 Address generation and general execution unit (AGU)

The address generation and general execution unit (AGU) calculates the address of the two load/store buses and performs other general-purpose integer operations needed for address calculation and general integer arithmetic (for example, add, multiply, shift, or, and so on). Address generation mathematics implements four types of arithmetic: linear, modulo, multiple wrap-around modulo, and reverse-carry.

The components of the address generation and general execution units block are as follows:

- Two load/store units (LSU 0 and LSU 1) capable of handling two issues of the following:
  - Load/store instruction, with optional zero overhead pre or post calculation of the address memory pointer. All above mentioned four arithmetic types supported[1].
  - General integer operations (add, sub, shift, logic operations, etc.).
  - Multibit shifts
  - Condition calculation
- One Integer processing unit (IPU), capable of linear integer operations. This unit is capable of handling one issue out of the following:
  - General linear integer operations (add, sub, shift, logic operations, etc.)
  - 32 × 32 multiplication
  - Register-based multibit shift

[1] Zero overhead pre calculation is supported only for linear addressing.

  - Condition calculation

In general LSU and IPU instructions operands (sources and targets) are taken from the AGU register file, although load/store and move instructions can write/read all of the SC3900FP registers (excluding the PC).

The AGU has a general purpose register file that includes the following:

- 32 general purpose 32-bit registers that can be used as pointers to memory
- Three 32-bit stack pointers (TSP, ESP and DSP)
- One 32-bit Modulo control register (MCTL)
- A register for arithmetic mode and status:
  - SR—Status Register, which holds the predicates and arithmetic mode and status bits
  - GCR—General Configuration Register, with additional arithmetic mode and status bits
  - BTR0–BTR1—registers dedicated to supporting the application-specific instructions for the Viterbi algorithm
- Program control registers
  - PC—Program Counter Register
  - SR2—Status Register 2, holding mode and status information related to privilege, SP selection and exception support
  - TID—Task Identification register
  - LC0–LC3—Four loop counter registers
  - CESRA0–CESRA1—Core Exception Service Routine Address registers
  - MOCR—Miscellaneous Option Configuration Register
  - LR0–LR2—Three link registers
  - CORREV—Core Revision register
- Registers for debug support
  - PROCID—A register that holds the full process iD, configured by the OS for debug and trace functions
  - TMTAG and TMDAT—registers for supporting user-defiled trace (data acquisition) through the Nexus trace

The AGU also provides access services to all SC3900FP control registers. Any instruction that is used to initialize these registers is an AGU instruction.

1.4.3 Program control unit (PCU)

The program control unit (PCU) performs instruction fetch (including change of flow and hardware loop control), instruction dispatch, and exception processing. The PCU controls the different processing states of the processor core. The PCU consists of three main units:

- Program fetch unit (PFU)
- Branch target buffer (BTB)
- Program dispatch unit (PDU)

The program fetch unit is responsible for controlling the sequence of the program flow, which includes linear code execution, all change of flow instructions, and hardware loops and exception processing. It also generates the program counter (PC) value for instruction fetch operations.

The branch target buffer unit enables dynamic prediction of previous taken change of flow, eliminating the change of flow penalty in most cases.

The program dispatch unit is responsible for detecting the VLES out of one or two fetch sets and dispatching the VLES's various instructions to their appropriate execution units—where they are decoded—and to the RSU (in order to calculate interlock hazards).

1.4.4 Resource stall unit (RSU)

The RSU controls the hardware interlocks. It collects information from the instruction bus, holds the status for all resources in the core, and inserts enough stalls to resolve the pipeline hazards.

1.4.5 Debug and trace unit (DTU)

The debug and trace unit provides a rich set of features that aid in debugging a real-time application; it also collects information about its operation non-intrusively. The debug unit interfaces with the SoC through a dedicated bus through which the external host could control the debug functionality.

1.4.6 Core interfaces

The main functional core interfaces include the following buses and signals:

- Program memory data (CPD) and program memory address (CPA) buses for carrying program words from memory to the core.
- Data memory A address (CAA) and data memory A data (CAD) buses for data transfers between the core and memory.
- Data memory B address (CBA) and data memory B data (CBD) buses for data transfers between the core and memory.

The processor core uses a single address space. Each location in the address space can contain either program information or data. The exact memory configuration is customizable for each chip containing a processor core. The memory system supporting the core typically consists of a memory management unit (MMU), hierarchy of caches and SoC level memory that can be expanded off-chip. The core-coupled memory system must support two parallel data accesses. However, it may hold the core in case it needs more cycles to complete so accesses.

1.5 Performance evaluation of the SC3900FP on typical DSP kernels

This section describes the current performance of the SC3900FP on typical DSP kernels. This performance evaluation is based on detailed analysis of the existing kernels written for the SC3850.

Table 1-4 shows the SC3900FP cycle count improvements of the typical DSP kernels relative to the previous generation, SC3850.

Table 1-4. Basic DSP kernels cycle count

Kernel	SC3850	SC3900FP	SC3900FP/SC3850 performance ratio
Complex FFT 256 (16 bits precision)	820	235	3.5x
Complex FFT 2048 (16 bits precision)	8770	1970	4.4x
Complex correlation	64	16	4x
Complex vector multiply (16b precision, vector length = 128)	96	16	6x
Real block FIR (40 data samples, 16 filter taps)	80	20	4×
Complex FIR (40 data samples, 16 filter taps)	320	80	4x
4x4 Complex matrix multiplication	32	8	4x
2x2 Complex matrix multiplication	4	1	4×
4x4 complex matrix inversion	165	52	3.2x
2x2 complex matrix inversion	18	4	4.5x
4x4 complex hermitian matrix inversion	90	31	2.9x
2x4 MMSE equalizer	44	14	3.1x
4x4 MMSE equalizer	240	65	3.7x
Dot product	32	8	4x
Division (1/x, 16bits)	4.5	1	4.5x
Maximum value and Index search (N = 512)	155	35	4.4×
LTE Descrambler (Cycle/bit)	1.1	0.09	12×

Chapter 2
Programmer's Model and Semantics

This chapter describes the programmer's model of the SC3900FP, including the register set, execution capacity and assembly ordering semantics.

2.1 Description of the SC3900FP registers

The register set of the SC3900FP is described in Figure 2-1 below. The sections that follow describe the various registers.

NOTE

Figure 2-1 shows the allocation of control registers (non R or D) to execution units. This allocation relates to functional affinity only, and does not imply the pipe behavior and stall scheme when they are accessed or initialized. For example MCTL is listed in the AGU because it is involved in setting the AGU modulo mode. However, transferring a value to MCTL may cause stalls that are unlike a similar transfer to an R register.

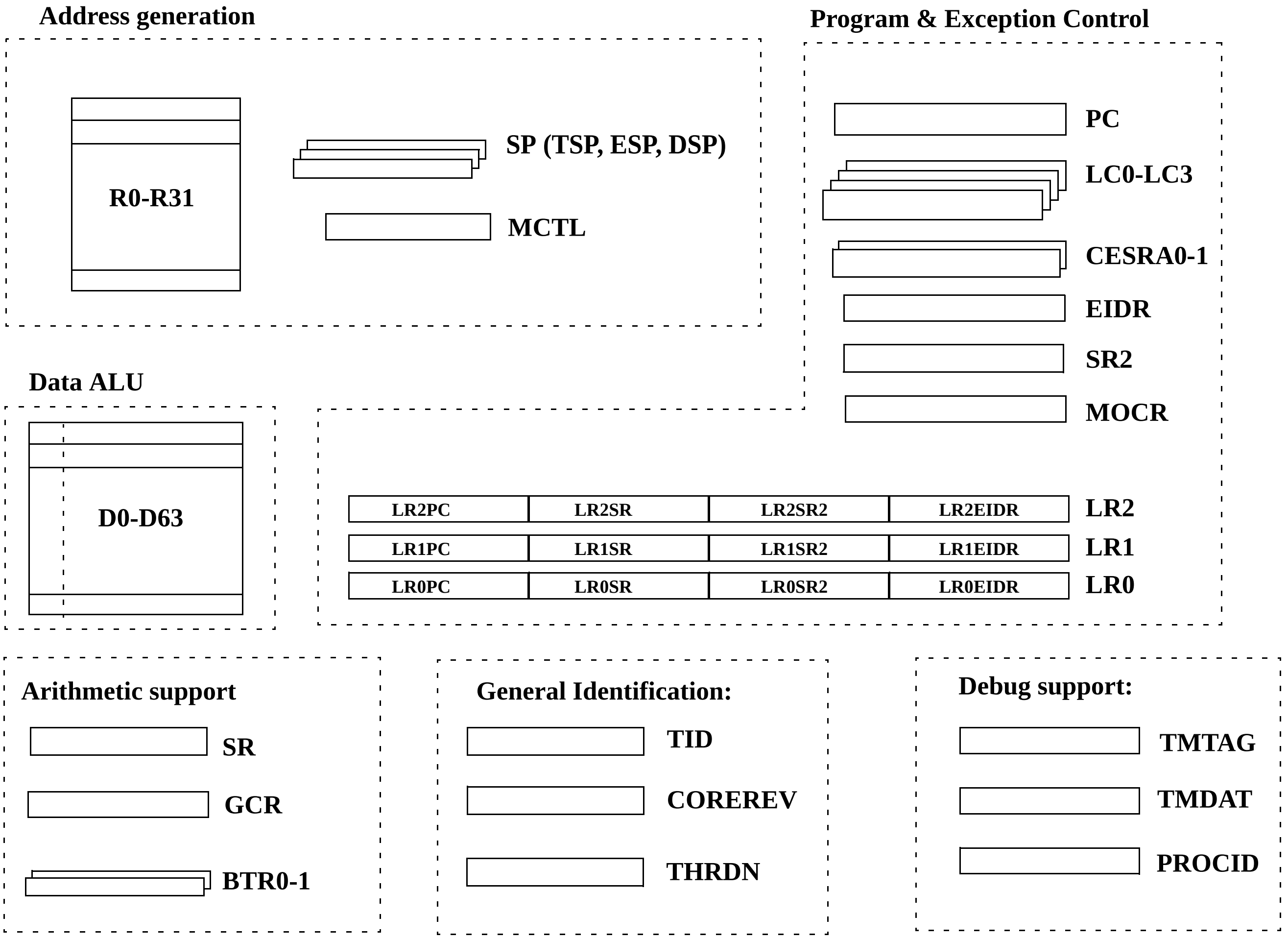


Figure 2-1. SC3900FP registers

2.1.1 Data Registers (D0–D63)

The data registers (D0–D63) are each 40-bit wide. These registers are the only ones that can be used as source or destination register operands of DALU instructions. Certain portions of each register can be specified as operands for various core instructions. Figure 2-2 shows the naming conventions for portions of the data registers.

In addition, there are efficient instructions to load or store D registers from/to memory, some of which access multiple registers.

Instructions either use various width portions as their operands, according to the data type (8, 16, 32, 40-bit value or 64-bit value in two coupled data registers), or use several portions independently in a SIMD fashion (Single Instruction, Multiple Data).

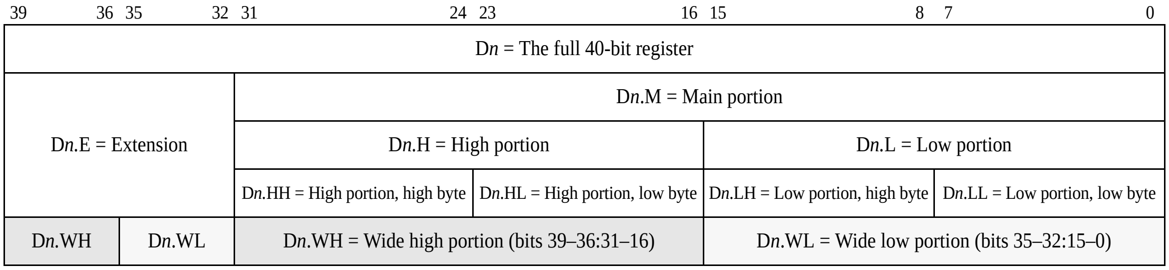

39
36 35
32 31
24 23
16 15
8 7
0
Dn = The full 40-bit register
Dn.M = Main portion
Dn.E = Extension
Dn.H = High portion
Dn.L = Low portion
Dn.HH = High portion, high byte
Dn.HL = High portion, low byte
Dn.LH = Low portion, high byte
Dn.LL = Low portion, low byte
Dn.WH
Dn.WL
Dn.WH = Wide high portion (bits 39–36:31–16)
Dn.WL = Wide low portion (bits 35–32:15–0)

Figure 2-2. Naming Conventions for Portions of the Data Registers

Register portions may be treated as holding variables having independent bit numbering. These bit numbers have different physical locations in the D register. Table 2-1 describes the explicit bit assignments of the register portions. Note that the WL and WH register portions occupy non-contiguous bit ranges in D*n*.

Table 2-1. Bit assignments of D Register portions

Register Portion	Number of bits of the variable	Bits in the D*n* register
D*n*	40	[39:0]
D*n*.M	32	[31:0]
D*n*.H	16	[31:16]
D*n*.L	16	[15:0]
D*n*.E	8	[39:32]
D*n*.LL	8	[7:0]
D*n*.LH	8	[15:8]
D*n*.HL	8	[23:16]
D*n*.HH	8	[31:24]
D*n*.WL	20	{ [35:32],[15:0] }
D*n*.WH	20	{ [39:36],[31:16] }

2.1.2 Address registers (R0–R31)

The address registers (R0–R31) are each 32-bit wide. Address registers are the only ones that can be used as source or destination register operands of arithmetic AGU instructions (such as, ADDA). In SC3900FP, some of the address registers also function as address pointers, address offsets of memory accesses, and as modulo modifier registers.

Nearly all AGU arithmetic instructions use the whole register as an operand, which is then named R*n*. A few instructions use a 16-bit portion of a register, which is then named R*n*.H or R*n*.L. Store instructions may read the full 32-bits of an R register, the lower 16 bits, or the lower 8 bits. Load instructions always update the whole R register, sign or zero extends the value loaded from memory if it is shorter.

This figure describes the allocation of address registers for SC3900FP.

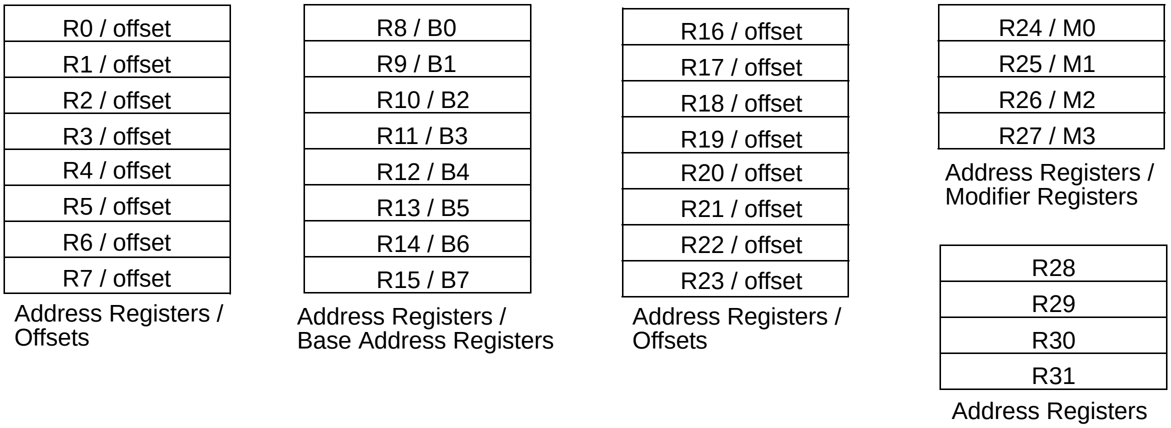


Figure 2-3. Address register allocation

The four modifier registers (M0–M3) are physically mapped to R registers (unlike the SC3850, which has dedicated registers for the M registers).

Sixteen of the R registers (R0–R7 and R16–R23) can function as address offsets in the indexed addressing mode. See Section 4.3.1, "Indexed addressing modes."

2.1.3 Stack pointers

The SC3900FP has three stack pointer registers, one of which is active. The active SP selection is set upon jump to an interrupt or return from it:

- TSP is used for Task SP (TSP is named NSP in previous architectures).
- ESP is used for Exception SP (except for debug).
- DSP is used for debug exception[1].

Instructions use the single register name "SP." The physical register that is actually used (TSP, ESP and DSP) is dynamically selected as "active," according to the SPSEL field in SR2.

The SP is explicitly used as an operand in some load and store instructions, and in some address arithmetic instructions. SP is also implicitly modified by PUSH/POP instructions, or by change of flow (COF) operations that perform an implicit push or pop (for example, JSR). The addresses in the stack pointer registers must be aligned to 8, hence the three LS bits in these registers are tied to 0 by the hardware.

The SC3900FP enables use of an additional SP selection in parallel with the selection pointed to by SPSEL. This is achieved with the ASPSEL field (the Alternative SP Select, which is located in SR2). The following dedicated instructions use the physical SP register specified in ASPSEL, instead of in SPSEL: PUSHC, POPC and TFRCA. This feature allows the OS to access and modify another stack (for example, when saving the old context) while maintaining the ability to work with its own stack. Upon an exception, the previous SP selection is copied from SPSEL to ASPSEL, as it is the most useful selection of an

[1] Debug exceptions are not supported in all subsystem implementations

alternative SP. However, ASPSEL can be modified to a different value by the software. For more information on SP selection, see Section 4.9.1, “Stack pointer selection.”

2.1.4 Registers for non-linear arithmetic support

The functionality of non-linear arithmetic remains unchanged from previous StarCore generations; the only modification is the aliasing of M0–M3 registers with R24–27.

The following registers are used for non-linear arithmetic support:

- R0–R7: these pointers are used to access the cyclic memory buffer
- B0–B7 (R8–R15): the base address of the cyclic modulo buffer
- M0–M3 (R24–R27): registers that hold the size of the modulo buffer. These registers can be dynamically allocated (with MCTL configuration) to an R pointer (R0–R7).
- MCTL (Modulo Control register): A 32-bit configuration register that is selected for each of R0 to R7 when used with a special addressing mode, and M register.

The special address arithmetic modes supported by MCTL configuration are as follows:

- Linear addressing: normal addition (default reset value)
- Modulo addressing
- Multiple wrap modulo
- Reverse carry

The structure of the MCTL register is illustrated in this figure.

Access: U: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	AM_R7				AM_R6				AM_R5				AM_R4			
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	AM_R3				AM_R2				AM_R1				AM_R0			
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 2-4. MCTL register format

The AM_R*n* field associated with each address register (R0–R7) reflects the address arithmetic mode for the address register as listed above. Each of the R*n* registers can use M0, M1, M2, or M3 as their associated modulo register in either modulo address arithmetic mode or multiple wrap-around modulo address arithmetic mode. When activating the modulo address arithmetic mode, the corresponding B*n* register defines the lower boundary value (B0 with R0, and so on). When the linear or reverse-carry address calculation modes are used, R8–R15 and R24–R27could be used as general purpose registers.

Table 2-2 lists the AM_R*n* field values and the corresponding address arithmetic modes.

Table 2-2. Address arithmetic mode values

AM_R*n*[1]	Address arithmetic modes
0000b	Linear addressing[2]
0001b	Reverse-carry addressing
1000b	M0 used—Modulo arithmetic
1001b	M1 used—Modulo arithmetic
1010b	M2 used—Modulo arithmetic
1011b	M3 used—Modulo arithmetic
1100b	M0 used—Multiple wrap-around modulo arithmetic
1101b	M1 used—Multiple wrap-around modulo arithmetic
1110b	M2 used—Multiple wrap-around modulo arithmetic
1111b	M3 used—Multiple wrap-around modulo arithmetic

[1] Unlisted combinations are reserved and should not be used.

[2] The MCTL register is reset to 0x00000000; therefore, linear addressing is the default address arithmetic mode.

For a full description of non-linear arithmetic, see Section 4.4, "Address modifier modes."

2.1.5 Registers for arithmetic status and control

2.1.5.1 Status Register (SR)

The SR is a 32-bit register that holds a mix of status flags and arithmetic mode bits.

The SR is automatically saved during an exception (to a link register) or subroutine call (directly to the stack). When jumping to an exception, SR is cleared. Return From Exception (RTE) instructions restore the SR from the stack to the value it had upon jumping to the exception (unless modified in memory).

This figure describes the SR field structure.

Access: U: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R											P5	P4	P3	P2	P1	
W	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨						▨
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R			RF	W20	SAT			SCM[2]	SM2	S	SCM[1:0]		RM	SM	P0 (T)	C
W	▨	▨				▨	▨									
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 2-5. Status Register (SR)

This table describes the functionality of the bits and fields of the SR.

Table 2-3. Status Register (SR) field descriptions

Field	Description
0 C	Carry 0: No carry or borrow generated 1: Carry generated from last addition, or borrow generated from last subtraction, or by another instruction that updates it.
1 P0 (T)	Predicate P0 (legacy T bit) 0: false 1: true
2 SM	32-bit Arithmetic Saturation Mode, affecting some legacy instructions. New SC3900FP instructions are not affected by this bit. 0: Disabled. 1: Enabled.
3 RM	Rounding Mode 0: Convergent rounding 1: Twos-complement rounding
8,5,4 SCM	Scaling Mode—Specifies the scaling to be performed during ST.SRS instructions as well as the rounding position for DALU instructions that perform rounding. The SCM field is split over non-contiguous range in SR. 000: No scaling 001: scale down by 1 bit 010: Scale up by 1 bit 011: Scale down by 2 bits 100: Scale down by 3 bits
6 S	Scaling Status flag. It is a sticky bit that once set remains so until explicitly cleared by software 0: The absolute value of the data of an ST.SRS instruction in the VLES (after scale, round and saturation) is less than 0.25 or greater than or equal to 0.75. 1: The absolute value of the data is greater than or equal to 0.25 and less than 0.75.
7, 12 SM2, W20	The SM2 (16-bit arithmetic saturation) and the W20 (wide 20-bit) mode bits affect how legacy SIMD2 instructions, or instructions that operate on a SIMD2 portion of a register, operate. New SC3900FP instructions are not sensitive to these mode bits. [W20,SM2] 00: Dual 16-bit, no saturation 01: Dual 16-bit, with saturation 10: Dual 20-bit, no saturation 11: Reserved
11 SAT	Saturation occurred. It is a sticky bit that once set remains so until explicitly cleared by software 0: Saturation did not occur since last clearing 1: Saturation occurred since last clearing
13 RF	Reservation Failed 0: No Reservation Failed occurred 1: Reservation Failed occurred
17 P1	Predicate P1 0: false 1: true

Table 2-3. Status Register (SR) field descriptions (continued)

Field	Description
18 P2	Predicate P2 0 - false 1 - true
19 P3	Predicate P3 0: false 1: true
20 P4	Predicate P4 0: false 1: true
21 P5	Predicate P5 0: false 1: true

2.1.5.2 General Configuration Register (GCR)

The General Configuration Register (GCR) holds additional status and configuration that relates to DALU instructions.

Access: U: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R			FDENI	FUNDER	FOVER	FDIVZ	FINVAL	FINEX								
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	AS3	AS2	AS1	AS0										BAM		
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 2-6. General Configuration Register (GCR)

Table 2-4. General Configuration Register (GCR) field descriptions

Field	Description
2-0 BAM	Byte Alignment Mode for the DOALIGN instruction 000: No shift 001: Shift 1 byte right 010: Shift 2 bytes right 011: Shift 3 bytes right 100: No shift 101: Shift 1 byte left 110: Shift 2 bytes left 111: Shift 3 bytes left
12 AS0	Arithmetic Status 0 — captures the carry bit of the DIVP.0

Table 2-4. General Configuration Register (GCR) field descriptions (continued)

Field	Description
13 AS1	Arithmetic Status 1 — captures the carry bit of the DIVP.1
14 AS2	Arithmetic Status 2 — captures the carry bit of the DIVP.2
15 AS3	Arithmetic Status 3 — captures the carry bit of the DIVP.2
24 FINEX	Floating point Inexact flag. A sticky flag, set if an Inexact condition occurred in any of the floating point instructions in the VLES. See Section 3.4.3.4.1, "Float Inexact Flag (FINEX)".
25 FINVAL	Floating point Invalid flag. A sticky flag, set if an Invalid Operation condition occurred in any of the floating point instructions in the VLES. See Section 3.4.3.4.2, "Float Invalid Flag (FINVAL)".
26 FDIVZ	Floating point Divide by Zero flag. A sticky flag, set if a Divide by Zero condition occurred in any of the floating point instructions in the VLES.See Section 3.4.3.4.3, "Float Division by Zero Flag (FDIVZ)".
27 FOVER	Floating point Overflow flag. A sticky flag, set if an Overflow condition occurred in any of the floating point instructions in the VLES. See Section 3.4.3.4.4, "Float Overflow Flag (FOVER)".
28 FUNDER	Floating point Underflow flag. A sticky flag, set if an Underflow condition occurred in any of the floating point instructions in the VLES. See Section 3.4.3.4.5, "Float Underflow Flag (FUNDR)".
29 FDENI	Floating point Denormalized Input flag. A sticky flag, set if a Denormalized Input condition occurred in any of the floating point instructions in the VLES. See Section 3.4.3.4.6, "Float Denormalized input Flag (FDENI)".

2.1.5.3 Back Trace Registers (BTR0, BTR1)

These two 32-bit registers help in the processing of the Viterbi algorithm by the ACS.H.2W and ACS.L.2W instructions. For more details, see the description of these instructions in Appendix C, "Instruction Set"

2.1.6 Registers for program and exception control

2.1.6.1 Using the Program Counter register (PC)

The PC is a 32-bit register that holds the program address of the currently executing VLES. An address of a VLES is that of the first program word in the VLES. The PC can be transferred to an R register with the instruction TFRA PC,Rn. In this case, the value of the PC is that of the VLES to which the TFRA instruction belongs. The PC register cannot be written to directly as a destination of a general instruction such as TFRA. It can only be changed by an explicit Change Of Flow (COF) instruction, or automatically by the core Program Control Unit (PCU) for example when servicing an interrupt.

Instructions are at least 16-bits long, and must be aligned in memory. Therefore, the values in the PC always have the LS bit as 0.

2.1.6.2 Using the Loop Count registers (LC*n*)

The four Loop Count registers (LC0–LC3) are used for specifying and controlling the iteration count of the hardware loops. These are 32-bit registers, each of which holds the loop count of the respective hardware loop. The LC*n* registers are updated implicitly when executing the loop control instructions such as DOEN.*n* and LPEND. The LC registers have limited accessibility by general instructions (TFRA to/from R registers)

See Section 5.3.1, "Optimizing loop execution with the hardware loop model," for a description of the hardware loop mechanism.

2.1.6.3 Using the Core Exception Service Routine Address registers (CESRA0/CESRA1)

The Core Exception Service Routine Address registers hold the target address of exceptions generated by internal core conditions. Each holds only the 26 MS bits of the address. Bits [5:0] of the registers are tied to zero; thus, the target address of internal exceptions must be aligned to 64 bytes.

The exceptions that use the CESRA0–1 registers are as follows:

- CESRA0—TRAP.0 (intended for use as a low-latency exception)
- CESRA1—Illegal + TRAP.1 (intended for use for all other exceptions)

Service routine addresses of other exceptions are programmed in their respective units: in the MMU (for memory exceptions), in the DTU (for debug exceptions, if supported) and in the EPIC (the subsystem interrupt controller, which is used for normal and critical interrupts).

2.1.6.4 Using Status Register 2 (SR2)

The SR2 is a 32-bit register used to specify the following:

- SP select/Alternate SP select
- Interrupt priority level
- Disable exception/interrupts nesting
- Exception masking

The SR2 is automatically updated during exception processing, and its value before the exception is copied to the appropriate link register. During execution, it is possible to update a subset of the fields of this register. Return From Exception (RTE) instructions restore the value of SR2 from the stack.

See Section 5.4.4, "How the core handles exceptions," for a detailed description on exception processing.

Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	IRR	ILL				DDN	DMN	DIN	DCI	DI		IPM				
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R										EXP		IDE	ASPSEL		SPSEL	
W																
Reset	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

Figure 2-7. Status Register 2 (SR2)

The functionality of each field in SR2 is described in Table 2-5. Fields that are described as “not writable by software” can be updated only when automatically jumping or returning from interrupts. However, core commands in debug mode can update these fields. A write to these fields during execution (for example with a TFRA instruction), is ignored without generating an error.

Table 2-5. Status Register 2 (SR2) field descriptions

Field	Description
1-0 SPSEL	Selects the stack pointer that is used in implicit operations (JSR/BSR/RTE), push/pop instructions and explicit SP reference. The value is modified automatically when jumping to an exception, and can be modified only during RTE. Not writable by software. 00: Task 01: Exception 10: (Reserved) 11: Debug
3-2 ASPSEL	Alternate SP selection 00: Task 01: Exception 10: (Reserved) 11: Debug
4 IDE	In Debug Exception. Set upon entry to debug exception (if supported), can be modified only during RTE. Not writable by software. 0: Not in debug exception 1: In debug exception
6 EXP	Exception. Set upon exception. Modified only during RTE. Not writable by software. Writing TID.[DP]ID + PROCID enabled only when EXP=1. 0: Not in exception 1: Exception
20-16 IPM	Interrupt Priority Mask 00000: Mask IPL 0. ... 11110: Mask IPLs 0–30. 11111: Mask IPLs 0–31.

Table 2-5. Status Register 2 (SR2) field descriptions (continued)

Field	Description
22 DI	Disable Maskable Interrupts. Set automatically when servicing any exception. 0: Interrupts are enabled. 1: Interrupts are disabled
23 DCI	Disable Critical Interrupts (Critical Interrupts previously called NMI). Set automatically when servicing a critical interrupt. 0: Critical interrupts are enabled 1: Critical interrupts are disabled
24 DIN	Disables all exceptions & interrupts except MMU and Debug exceptions. DIN is set automatically when servicing any exception. 0: Interrupts and exceptions are enabled 1: Interrupts and exceptions are disabled
25 DMN	Disables MMU exception nesting. DMN is set automatically when accepting MMU exceptions. 0: Exceptions are enabled 1: Exceptions are disabled
26 DDN	Disables Debug exception nesting. DDN is set automatically when accepting Debug exceptions. 0: Exceptions are enabled 1: Exceptions are disabled
30 ILL	Illegal detection indication Set when jumping to an Illegal exception. 0: No Illegal exception condition occurred since this bit was last cleared. 1: An Illegal exception condition occurred.
31 IRR	Irrecoverable exception indication Set when jumping to an Irrecoverable exception. 0: No Irrecoverable exception condition occurred since this bit was last cleared. 1: An Irrecoverable exception condition occurred.

2.1.6.5 Using the Exception ID Register (EIDR)

The Exception Identification Register (EIDR) holds a qualifier value, updated upon jumping to an exception. The value of EIDR prior to the exception is copied to the respective link register. The value that is updated in the EIDR is either sampled from the interrupt interface (driven by the source interrupting device or interrupt controller), or supplied by the core. Instructions that cause exceptions, such as TRAP, usually have an immediate field that is sampled into EIDR, allowing the user to qualify the type of service or exception situation for the ISR software. The maximal length of the ID value is 10 bits, as shown in Figure 2-8. Some interrupts may be limited to a shorter value. For a detailed description on exception processing, see Section 5.4.4, “How the core handles exceptions.”

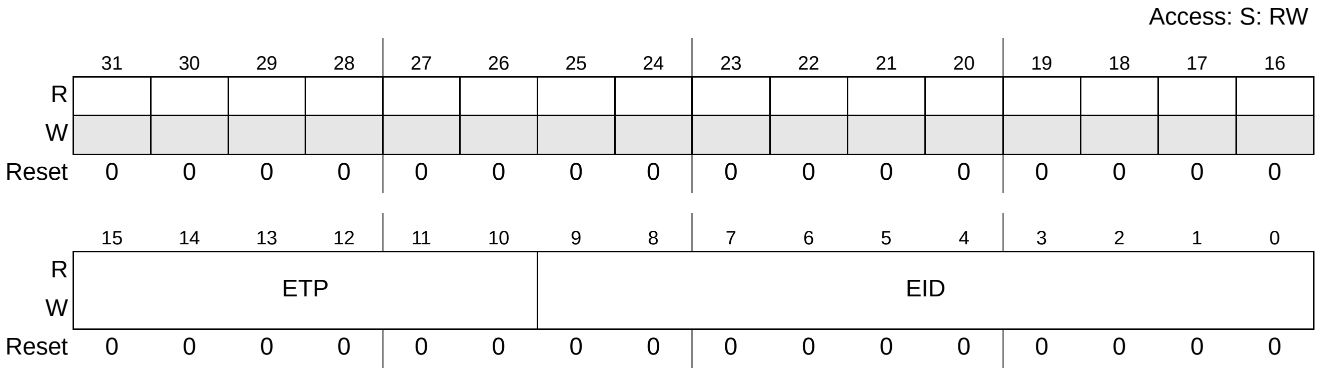


Figure 2-8. Exception ID Register (EIDR) diagram

Table 2-6. Fields of Exception ID Register (EIDR)

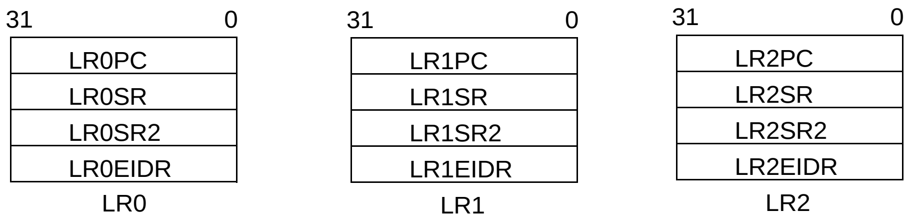
Field	Description
9-0 EID	Exception ID. Holds the immediate field or detailed data.
15-10 ETP	Exception Type. Marks the unit generating the exception + type.

2.1.6.6 Using the Link Registers (LR0, LR1, LR2)

The core programming model has three link registers, as follows:

LR0	Link register used for all exceptions and interrupts except for MMU and debug exceptions
LR1	Link register used for MMU exceptions only
LR2	Link register used for Debug exceptions only[1]

The Link Registers (128-bits each) consist of the following four registers (32-bits each):



- Upon exception or interrupt:
  — PC:SR:SR2:EIDR are copied to the appropriate link register
  — PC:SR:SR2:EIDR registers are updated by the interrupt/exception
  — The user (in the beginning of the ISR) is responsible for preserving the link register, for example, by pushing it into the stack before enabling any other interrupt/exception that uses the same link register. Section 5.4.4, "How the core handles exceptions," provides example code sequences for this procedure.

[1] Debug exceptions are not supported in all implementations of the subsystem

- Direct access to the link registers:
  - — The link registers can be pushed and popped from stack (a single 128 bit access per link register):
    PUSH/POP.4L LR*n* (*n*=[0,1,2])
  - — The link registers can be transferred from/to R registers: 32-bits access with no stalls:
    TFRA LR*n*.xx,Rn (where: xx=[PC/SR/SR2/EIDR], *n*=[0,1,2])

2.1.6.7 Using the Miscellaneous Option Configuration Register (MOCR)

This register includes mode bits that control miscellaneous core options. Some of these options affect only the cycle (that is, micro-architectural) core behavior, without changing the numerical results; others may be visible to the application.

This register is not normally modified by the application. It has a hardwired reset value, which in products using the SC3900FP is configured to 0x310. If needed, it should be written once during boot and normally should not be modified afterwards.

A write to this register (TFRA, BMSETA, BMCLRA, BMCHGA) should be done with a parallel CLRIC instruction that clears the BTB. See Rule J.6.

This figure describes the bits of the MOCR.

Access: S: RW

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R							DLSB	DMHFS			DSBRS	DCASC	DSIAO	DHLB	DBTB	DSO
W																
Reset	0	0	0	0	0	0	U	U	0	0	U	U	U	U	U	U

Figure 2-9. Miscellaneous Option Configuration Register (MOCR)

Table 2-7. Miscellaneous Option Configuration Register (MOCR) field descriptions

Field	Description
0 DSO	Disable Speculative COF operations 0: Enabled 1: Disabled
1 DBTB	Disable Updating the Branch Target Buffer 0: Enabled 1: Disabled
2 DHLB	Disable hardware loop BTB 0: Enabled 1: Disabled - every LPEND performs a full COF

Table 2-7. Miscellaneous Option Configuration Register (MOCR) field descriptions (continued)

Field	Description
3 DSIAO	Disable Speculative IF.P*n* AGU Operations 0: Enabled 1: Disabled
4 DCASC	Disable COF speculation after a Speculative COF 0: Enabled 1: Disabled
5 DSBRS	Disable Subroutine instruction Speculation - do not activate BSR, JSR and RTS speculation 0: Enabled 1: Disabled
8 DMHFS	Disable Multi-hit in a Fetch Set - When disabled, only one COF per fetch set is updated in the BTB. The BTB is updated with the last taken COF it encountered, replacing any other COF from that fetch set 0: Enabled 1: Disabled
9 DLSB	Disable Loop Start in the hardware loop BTB. When this disables, it blocks the effect of the LPST* instruction on both non-sequential loops and loops that are longer than 8 fetch sets, making the first iteration a loop-BTB miss. This also disables the dynamic update of the LC predictor in the BTB when the loop is initialized by DOEN* R*n* and when R*n* is changed between occurrences of the loop. 0: Enabled 1: Disabled

2.1.7 Registers for identification and revision control

2.1.7.1 Using the Task ID register (TID)

The TID is a 32-bit register that holds identification information on the current task that is relevant for MMU interface and cache micro-architecture.

PID is the task ID used by program accesses, and DID is the task ID used by data accesses. The separation allows users better cache performance when the same program is instanced several times, using different data for each instance. See Section 4.2, “Memory addressing,” for details on the task ID and its usage in memory addressing.

The Alternate ID (AID) is a field used by some memory accesses to override the ID value that is issued to the MMU (either DID or PID, depending on the case), using the Instruction Attribute Override mechanism (see Section 4.11, “Instruction attribute override”).

The DID and PID fields can be written to only when SR2.EXP = 1. When EXP = 0 the write does not affect the contents of these fields. However, a write during a core command in debug mode updates these fields.

This figure describes the fields of the TID register.

Access: S: RW

<table>
<tr><td></td><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td></tr>
<tr><td>R</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="8" rowspan="2">AID</td></tr>
<tr><td>W</td><td>▨</td><td>▨</td><td>▨</td><td>▨</td><td>▨</td><td>▨</td><td>▨</td><td>▨</td></tr>
<tr><td>Reset</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

<table>
<tr><td></td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>R</td><td colspan="8" rowspan="2">DID</td><td colspan="8" rowspan="2">PID</td></tr>
<tr><td>W</td></tr>
<tr><td>Reset</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Figure 2-10. Task ID register (TID) diagram

Table 2-8. Task ID register (TID) field descriptions

Field	Description
7-0 PID	Program ID Can be written to only when SR2.EXP is set
15-8 DID	Data ID Can be written to only when SR2.EXP is set
23-16 AID	Alternate ID

2.1.7.2 Using the Core Revision register (COREREV)

The Core Revision register (COREREV) is a 32-bit read-only register, that holds hard-wired information on the revision number of the core as well as information on the values used for configurable architectural parameters in this core. This register can only be transferred to an R register using a TFRA instruction.

Access: U: R

<table>
<tr><td></td><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td></tr>
<tr><td>R</td><td colspan="4">RUNIQ</td><td colspan="4">REVMN</td><td colspan="4">REVMJ</td><td colspan="4">ARCHV</td></tr>
<tr><td>W</td><td colspan="4">R</td><td colspan="4">R</td><td colspan="4">R</td><td colspan="4">R</td></tr>
<tr><td>Reset</td><td>U</td><td>U</td><td>U</td><td>U</td><td>U</td><td>U</td><td>U</td><td>U</td><td>U</td><td>U</td><td>U</td><td>U</td><td>U</td><td>U</td><td>U</td><td>U</td></tr>
</table>

<table>
<tr><td></td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>R</td><td></td><td></td><td></td><td></td><td colspan="2">CICN</td><td colspan="4">CLUSTN</td><td colspan="6">COREN</td></tr>
<tr><td>W</td><td>▨</td><td>▨</td><td>▨</td><td>▨</td><td colspan="2">R</td><td colspan="4">R</td><td colspan="6">R</td></tr>
<tr><td>Reset</td><td>0</td><td>0</td><td>0</td><td>0</td><td>U</td><td>U</td><td>U</td><td>U</td><td>U</td><td>U</td><td>U</td><td>U</td><td>U</td><td>U</td><td>U</td><td>U</td></tr>
</table>

Figure 2-11. Core Revision Register (COREREV) diagram

Table 2-9. Core Revision Register (COREREV) field descriptions

Field	Description
5-0 COREN	Core Number in the SoC. Accumulative across all clusters. Can be used as a bit index in SoC registers that allocate a bit per core in the SoC. Some of the SoC registers allocate a bit per thread which may be a different number - see Section 2.1.7.3, “Using the Thread Number Register (THRDN). Possible values are 0–63
9-6 CLUSTN	Cluster Number in the SoC Possible values are 0–15
11-10 CICN	Core Number in the Cluster Possible values 0–3
19-16 ARCHV	Architecture Version. Versions of the instruction set. 0000: V7 All others reserved.
23 -20 REVMJ	Major Revision Number. Core families based on the same ISA architecture (minor ISA additions still included) 0000: SC3900 0001: SC3900FP All others reserved.
27-24 REVMN	Minor Revision Number. Revisions that include differences in user features. 0000: Rev1 All others reserved.
31-28 RUNIQ	Revision-Unique information. Used for differentiating between dashes, process migrations, and so on. 0000: First release All others reserved.

2.1.7.3 Using the Thread Number Register (THRDN)

The Thread Number register (THRDN) is a 32-bit read-only register, that holds hard-wired SoC thread identification number running on this core. A “thread” in this context is the hardware term for an independent programming flow that has its own programming model, not a dynamic software entity allocated by the OS. The SoC thread number is used as a bit index when accessing SoC registers that hold a bit per thread in the SoC, for example to control entry and exit from low power modes. Although the SC3900FP core supports running only a single thread, other cores in the SoC may support multiple threads, so the SoC thread index of this SC3900FP core may have an offset relative to the SoC index of the core in the SoC (the core index is configured in COREREV.COREN).

This register can only be transferred to an R register using a TFRA instruction.

Access: U: R

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	THREDN															
W	R															
Reset	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	THREDN															
W	R															
Reset	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U

Figure 2-12. Thread Number Register (THRDN) diagram

Table 2-10. Thread Number (THRDN) field descriptions

Field	Description
31-0 THREDN	The SoC identification number of the thread

2.1.8 Registers for debug support

The core programming model has three registers for trace and debug functionality support. This is in addition to the main programming model of debug support, which is configured in the DTU (see Chapter 9, "Debug and Trace Support"). Two of the registers are defined to support data acquisition trace messages. These two registers can always be written to or read from by the program, with no other effect except for the possible trace. For more information on tracing and data acquisition messages, see the "Debug and Trace Support" chapter in the *SC3900FP FVP Cluster Reference Manual.*"

2.1.8.1 Using the Trace Message Tag register (TMTAG)

Trace Message Tag: a 32-bit register, bits [7:0] hold a tag that will be issued as part of the Data Acquisition trace Message (DQM). This tag can be used by the profiling software to qualify the data of the message that is written to TMDAT. Writing to this register does not trigger a DQM.

2.1.8.2 Using the Trace Message Data register (TMDAT)

The TMDAT is a 32-bit, write-only register. On writing to this register, if trace is enabled and properly configured, a data acquisition trace message (DQM) is generated and the tag is taken from TMTAG[7:0]. Writing to this register during debug mode is reflected in a memory-mapped DTU register, to aid the debugger when downloading the core registers during debug mode. For details, see the "Debug and Trace Support" chapter in the *SC3900FP FVP Cluster Reference Manual.*

2.1.8.3 Using the Process ID register (PROCID)

The Process ID register (PROCID) is a 32-bit read-write register that can be written to only when the SR2.EXP bit is set, however core commands in debug mode can write to it regardless of the state of SR2.EXP. When a task switch occurs, the register should be updated by the OS with the process ID number that is about to run. This value can be configured in the debug and trace unit to be compared with a reference value when SR2.EXP is cleared, and the comparison result used for event or trace filtering.

The fields in the TID register are shorter (see Section 2.1.7.1, “Using the Task ID register (TID)”) because they are used as part of the virtual address in the MMU. Thus, an OS that uses more than 256 processes typically manages the PID/DID as a reusable resource between several processes, which makes uniquely identifying them during debug more difficult. For best support of the debug features, the PROCID register should be programmed even if the OS does not uses more than 256 tasks and thus has no problem with PID/DID aliasing. In this case, it is recommended that the OS program the value DID:PID into PROCID.

In general, the value programmed in PROCID should enable differentiation between the tasks in the finest granularity defined by the OS.

2.2 Memory access rights

The OS can assign access rights to memory segments, allowing, per task, that this segment is accessible to data accesses, program fetches, or none. Access permission could also be differentiated for loads and for stores. The MMU checks the accesses issued by the core without adding latency. However, when the MMU detects a violation of the access rights, it generates a precise memory exception. The exception aborts the VLES that generated the access so that the architectural state of the core and memory is kept intact as it was before this VLES executed. The exception handler can decide whether or not to abort the task before it has done any damage to the memory of the system or of other tasks. Alternatively, the OS can decide to grant the task access rights by modifying the configuration of the MMU. Upon return from the exception, the VLES that caused the exception is re-executed. This way a software-based virtual memory scheme can be implemented. For more information on the MMU and memory protection scheme, see the *SC3900FP FVP subsystem Reference Manual*.

The SC3900FP core supports an “Access Attribute Override” mechanism, which enables the OS to issue memory accesses and pre-fetch commands to the virtual memory space of other tasks, by using the tasks access attributes, such as the privilege level and task ID, instead of those used by the task currently running. For a detailed explanation on access attribute override, see Section 4.11, “Instruction attribute override.”

2.3 Execution capacity

2.3.1 Instruction types

This section describes what types of instructions exist in the SC3900FP, based on the instruction’s ability to activate the different execution units. Most SC3900FP instructions activate one execution unit. Some instructions may activate implicitly two execution units.

2.3.1.1 DALU instructions

A 32-bit DALU instruction activates one of the four DMUs. The DMUs are identical units. Therefore, in principle, a DALU instruction can be executed in any DMU, assuming there is no semantic ordering limitation (see Section 2.3.2, “Sequence semantics inside a VLES”). The 64-bit DALU instructions activate two DMUs (DMU0,1 or DMU2,3), or one odd-numbered DMU while its even counterpart is left idle (although not usable by another instruction).

A DALU instruction uses only D registers as source and destination register operands. A set of DALU instructions can explicitly update predicates (for example, CMP.EQ Da,Db,Pn) and manipulate them (for example, PCALC #u8,Pa,Pb,Pc,Pn). Implicit status bits can be updated in SR, GCR and BTR0,1. The operation of the instruction may be affected by mode bits in SR and GCR.

2.3.1.2 AGU instructions

An AGU instruction activates one of the three execution units in the AGU. Unlike the DALU, the AGU execution units are not symmetrical, but are of the following two types:

- LSU (Load-Store Unit). Two units that can perform the following operations:
  - A memory data load or store operation
  - Data cache control instructions
  - Integer arithmetic operation based on R registers (excluding special operations)
  - Non-linear arithmetic, such as modulo and reverse carry calculations
- IPU (Integer Processing Unit). A single execution unit that can perform a set of integer arithmetic instructions. Some instructions unique for this unit are as follows:
  - Integer multiplication
  - Register-based multi-bit shifting
  - Several integer arithmetic instructions supported also by the LSU (such as TFRA)
  - Transfer of control register to or from an R register
  - Initializing loop counter registers
  - Operations on control registers

Load and Store instructions can transfer data between the memory and nearly all core registers. These transfers may be done in conjunction with an integer address calculation that is based on SP or R registers.

AGU instructions that do not involve a memory transaction may use R, SP, or control registers as sources and destinations. An exception are the register MOVE instructions, that can exchange data with D registers.

2.3.1.3 PCU instructions

The PCU instructions activate the program control unit. In previous StarCore architectures these instructions were considered AGU instructions. In SC3900FP they are split into an independent group. These instructions include the following:

- Change Of Flow (COF) instructions
- Hardware loop control instructions (except for DOEN.n)
- Interrupt triggering (TRAP, and so on)
- Debug support (such as DEBUGM.*n*).
  Note: Data writes to memory-mapped DTU registers are not part of this group.

PCU instructions that use an R register (such as JMP Rn) or perform an implicit data memory access (such as JSR, BSR, RTS, RTE) occupy an AGU execution slot in addition to the PCU execution slot.

2.3.1.4 Suffix instruction words

Some VLES may be encoded with an additional suffix instruction word. The suffix is decoded in the dispatcher unit, and not by a specific execution unit. A suffix may be generated by the assembler in response to a mnemonic, or when specific operands are used.

- Extend the encoding of instructions in the VLES for some operands: predicates P3–P5, and DALU registers D32–D64 in DALU instructions
- A NOP instruction. Used as a VLES place holder, or for VLES length alignment.

Suffix instruction words do not consume an execution slot.

2.3.2 VLES model and grouping capacity

In StarCore terminology, an “Execution Set” is a group of instructions dispatched together in the same cycle for parallel execution. The number of execution units that could be activated in a single execution set is variable, hence the term “Variable Length Execution Set” (VLES) to denote the group of instructions that execute in parallel. No NOP instructions are required to pad the execution set for unused execution units.

The grouping information is statically specified in the source assembly code (manually written or automatically generated by the compiler). Grouping of instructions is in general very flexible, but is restricted by several obvious factors, as follows:

- The execution capacity that is set by the number of execution units, which is:
  - Up to four DALU instructions
  - Up to two LSU instructions
  - Up to one IPU instruction
  - Up to one PCU instructions
- The total encoded length of the VLES: 256 bits, or 16 instruction words
- The total encoded length of the DALU instructions is 128 bits
- Resource conflicts between instructions such as writing to the same destination

In some cases not all instruction combinations that conform to the rules above are supported. The full list of grouping limitations is described in Chapter 8, “Programming Rules and Guidelines.”

2.3.3 Meta-instructions

The term “Meta instructions” in the StarCore architecture relates to mnemonics that have that appearance of a regular instruction, performing a distinct function, however are encoded by the assembler as several instructions that are grouped together, belonging to different execution units. The reason for having meta instructions is to improve the readability in case a certain operation mandatorily requires several grouped instructions. For example, the following cache control instruction, that programs a CME channel to pre-fetch a block of data into the L1 cache:

```
        dfetchm r0,r1,r2
```

is encoded as the following three instructions (2 LSU instructions and a PCU instruction):

```
        dfetchb r1,r2    bccas r0         sync.b
```

Usually, the base portion of meta-instructions ends with an M, as can be seen in the example above. For more detail on meta-instructions, see Section 4.10, “Cache control instructions” and Section 4.13, “Memory barriers”.

Whenever possible, the user should use a meta-instruction instead of an explicit specification of the low-level instructions. However, the low level instructions used by the meta-instructions are described in Appendix C, “Instruction Set”, and the assembler supports them.

2.4 Data type support

Algorithm developers typically assign a data type to all data variables used in a program. The data type defines the mathematical aspects of the variable, for example:

Width — The number of bits the variable is occupying. This number affects the numerical range and/or the resolution of the values the variable may hold.

Mathematical representation
Fixed or floating point.
For fixed point data types, the position of the fraction point, for example in integer and fractional data types.

Modulo arithmetic and other specialized arithmetic types

Saturating arithmetic — The results of arithmetic operations are saturated to a maximum and minimum value, and do not wrap around in case of an overflow of a result.

Sign — Whether the variable is signed or unsigned

High-level programming languages, such as C, support natively some data types like Integer, Float, and so on. Some languages also allow the programmer to extend the set of supported data types to additional types, such as C++, embedded C, and so on. Once defined, the compiler can map the operations specified in the program to core assembly instructions. The efficiency of this mapping depends on the level of support the core offers for the various data types and operations. In general, processors give optimized support only for a subset of the data types the user may use in the code written in a high-level programming language. Data types outside of this subset can be supported but are usually translated to more core instructions per high level operation, giving less efficient code.

The SC3900FP supports an exceptionally large set of data types. This allows a compiler and code developer to choose the most efficient data representation of DSP and control algorithm, which results in efficient code.

This section provides a high-level description of the supported data types. The detailed mathematical definition of these data types is described in Chapter 3, “Data Arithmetic Logic Unit (DALU),” and in Chapter 4, “Address Generation and Memory Interface.”

2.4.1 Aspects of data type support

There are several hardware aspects in which the SC3900FP core supports a data type, as follows:

- Width and position in memory: in most cases, a variable is stored in memory occupying the native width of the data type, for example a signed integer byte occupies one byte in memory. However, data types with non-standard widths such as 40 bits occupy memory that is aligned to the next power of 2 bits (64 bits in this case).
- Register support: D registers support all data types of the set supported by the SC3900FP core. The R registers can support only non-saturating integer types of bytes, words and longs. For these types, the compiler or user may choose in what register to place them (R or D), and also change their position afterwards. All other data types are supported in D registers only.
- Width and position in the register: Data types that have a bit length that is shorter than the register they occupy require a convention regarding the portion of the register they are placed into, and whether they require specific data to be placed in the unused portion (such as sign or zero extension)
- Dedicated load and store instructions that perform the required alignment and transfer between the register and memory
- Dedicated arithmetic instructions that perform the special flavor of arithmetic operations required by the data type
- SIMD support: load, store and arithmetic instructions that can perform operations on several variables of the data type in one instruction. This kind of support can be seen as either an implicit way for the compiler to improve performance of a single-variable algorithm, or the user may explicitly define a vector of simpler data types that should be processed together.
- Complex arithmetic support: two variables of the same data type may be defined as holding the real and imaginary parts of a complex number. The SC3900FP core has instructions that support complex variables built of several basic data types.

Ideally, a data type supported by the SC3900FP uniquely uses a subset of the full instruction set architecture (ISA). This ISA subset ensures that a variable of this data type can be loaded from memory and placed in the proper position in the register, and that a set of arithmetic operations can use input operands of this data type, and generate an output of the same data type, which can be used by subsequent instructions. Also, proper support includes a set of instructions that allow casting from other (usually standard) data types into this data type, and vice versa. Optimized support allows the mentioned functions for SIMD operations and Complex arithmetic as well.

Not all data types are supported in a way that allows for every high-level language operation to be translated into a single assembly instruction. For some data types, some operations require more than one assembly instruction to support them. For example, a 64-bit addition (long-long C data type) is supported by a sequence of two instructions in SC3900FP. At the extreme case, data types that do not have special support in the SC3900FP core are supported by the compiler using library functions, for example, the double-precision floating point data type.

Note that, in many cases, there is no one-to-one mapping between an SC3900FP instruction and a data type; in cases where the arithmetic definition allows it, some SC3900FP instructions are used for supporting several data types. For example, a 40-bit ADD.X is also used for 32-bit addition (both integer and fractional data types). In this case, the value in the 8-bit extension (D*n*.E) is not affecting the arithmetic result of the operation, so the 40-bit operation can be used for shorter data types. However, a right arithmetic shift is mathematically different for a 40-bit variable and a 32-bit variable. Thus, the SC3900FP has different instructions for a 40-bit arithmetic shift (ASH.RGT.X) and a 32-bit arithmetic shift (ASH.RGT.L).

Some data types (namely, 8-bit wide data types) do not have many dedicated arithmetic instructions to support them. Byte arithmetic is performed by casting the variable into a wider data type (20 or 40 bits wide), and performing the arithmetic operations using the instructions that support the wider data type. Full support is given to load and store instructions, which function as "load and cast" or "cast and store" operations. For example, when loading a signed integer byte from memory, the LD.B instruction positions the byte in D*n*.LL, and then the sign extends it to the left. The integer byte is thus casted to be a 40 or 16-bit signed integer word. There is no dedicated SC3900FP instruction for multiplying a signed integer byte. Rather, a signed integer word multiplication should be used (MPY.I.X), which takes the 16-bit D*n*.L as its source operand. However, since this 16-bit operand has its upper 8 bits sign-extended, the resulting number in Dn has the correct byte multiplication result in D*n*.LL. A subsequent byte store (ST.B) takes the correct result into memory.

This scheme of byte support may cost the compiler extra instructions to make sure that the "unused" portion of the register does not affect the arithmetic result. This is particularly true if the byte variable has undergone processing that affected the unused part, from the time it was loaded from memory. For example, the sequence for performing a comparison of two integer bytes, one loaded from memory:

```
[ ld.b (r0),d0            ; load the byte from memory, sign extending it in d0
sxt.b.x d1,d1 ]           ; sign-extend the other variable
cmp.eq.x d0,d1,p0         ; perform a 40-bit comparison
```

This scheme is extended to SIMD support of two byte operations per instruction, because two 16-bit operations can be supported based on a single D*n* register. Wider SIMD is also possible with instructions that are processing pairs of source registers (with two bytes in each). The byte processing throughput for general arithmetic is thus equal to that of 16 bits operands. However, there are specific instructions that allow SIMD4 processing of bytes in the same register, for optimizing video kernels (such as SAD.4B).

Another case of implicit casting occurs for nearly all multiplication instructions, which inherently double the width of the data they return. The SC3900FP multiplication-based instructions also include instructions that mix different data widths as their source operands, for example:

```
mac.x    Da.H,Db.H,Dn    ; multiplies 16-bit operands, and accumulates in 40-bits
macm.x   Da.H,Db,Dn      ; multiplies 16x32, and accumulates the MS 32-bits to 40 bits
```

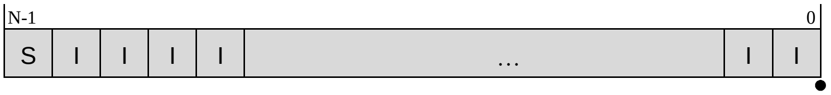
Note that, in cases like this, because the numerical results may be different relative to an implementation in which the accuracy within the data type is maintained after every instruction, the casting of an operation into a wider data type is exposed in the algorithm.

2.4.2 Overview of fixed point data types

The SC3900FP core supports few types of fixed point variables, which differ from one another by their width and attributes. Supported fixed point variables may either be unsigned or signed (two's complement). These attributes are defined in the following sections.

2.4.2.1 Signed integer

A signed integer number has the decimal point right of the LSBit, and the sign is the MSBit. Signed integers of 20, 40 and 64 bits are supported in D registers only. Signed integers of 8, 16 and 32 bits are supported either in R or D registers.



The numerical range of a signed integer number X of width N is: $-2^{[N-1]} \le X \le [2^{[N-1]} - 1]$ and the granularity (value of 1 LSBit) is 1.

The following table shows the decimal and hexadecimal values of signed integer numbers, according to the width.

Table 2-11. Signed integer values

Width	Name	Minimum value		One LSBit		Maximum value	
		Dec	Hex	Dec	Hex	Dec	Hex
8-bit	byte / char	–128	0x80	1	0x01	127	0x7F
16-bit	word / short	–32768	0x8000	1	0x0001	32767	0x7FFF
20-bit	—	–524288	0x8_0000	1	0x0_0001	524287	0x7_FFFF
32-bit	long (integer)	-2^{31}	0x8000_0000	1	0x0000_0001	$2^{31}-1$	0x7FFF_FFFF
40-bit	—	-2^{39}	0x80_0000_0000	1	0x00_0000_0001	$2^{39}-1$	0x7F_FFFF_FFFF
64-bit	longlong	-2^{63}	0x8000_0000_0000_0000	1	0x0000_0000_0000_0001	$2^{63}-1$	0x7FFF_FFFF_FFFF_FFFF

When signed integers are multiplied, the SC3900FP supports three types of results, as follows:

- Full accuracy, in which the output width is at least twice of the multiplicands,
- the high part of the result with the same width,
- or the low part of the result with the same width.

The type of result is defined by the instruction. Note that when only the high part is taken, it is always rounded.

The following example shows 16x16 signed integer multiplication in “C like” definition:

```
short a,b,res_high,res_low;
long40 res_full;// long40 is a 40-bit wide integer
res_full = (long40)a * (long40)b;// 32-bit full result signed extended to 40-bit
res_high = (short)(((long40)a * (long40)b + 0x8000) >>16);
         // Upper 16 bits of the rounded result
res_low  = a * b;// Lower 16 bits of the result
```

The following is the assembly code implementing the example above (note that, in some cases, SIMD instructions are used with only one part of the result used):

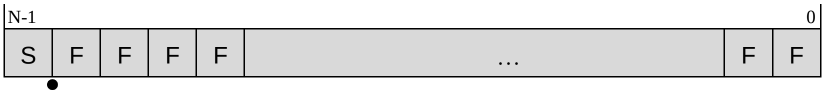
```
; a - D0.H, b - D2.H, res_full - D4, res_high - D6.H, res_low - D8.H
        mpy.i.x    d0.h,d2.h,d4; Calculate res_full into D4
        mpy.ir.4t  d0:d1,d2:d3,d6:d7; Calculate res_high into D6.H
        mpy.l.i.4t d0:d1,d2:d3,d8:d9; Calculate res_low  into D8.H
```

NOTE

While the 8-bit signed integer is supported by very few DALU instructions, the casting of it during load and store is supported.

2.4.2.2 Signed fraction

A signed fractional number has the decimal point right to the MSBit, and the sign is the MSBit. Signed fractions are supported only in D registers.



The numerical range of a signed fractional number X of width N is: $-1.0 \le X \le [1-2^{-[N-1]}]$ with granularity (the value of 1 LSBit) of $2^{-[N-1]}$.

This table shows the decimal and hexadecimal values of signed fractional numbers according to the width.

Table 2-12. Signed fraction values

Width	Name	Minimum value		One LSBit		Maximum value	
		Dec	Hex	Dec	Hex	Dec	Hex
8-bit	byte / char	–1.0	0x80	2^{-7}	0x01	$1.0–2^{-7}$	0x7F
16-bit	word / short	–1.0	0x8000	2^{-15}	0x0001	$1.0–2^{-15}$	0x7FFF
32-bit	long (integer)	–1.0	0x8000_0000	2^{-31}	0x0000_0001	$1.0–2^{-31}$	0x7FFF_FFFF

When signed fractions are multiplied, the SC3900FP supports three types of results, as follows:

- Full accuracy, in which the output width is at least twice of the multiplicands width,
- the high part of the result with the same width,
- or the rounded high part of the result with the same width.

The type of the result is defined by the instruction.

In order to calculate fraction multiplication correctly, the result should be shifted left by one bit relative to integer multiplication. This implicit shift left by fraction multiplication is the only difference between instructions that operate on signed integer variables and signed fraction variables.

The following example shows 16x16 signed fraction multiplication in “C” like definition:

```
short a,b,res_high,res_high_round;
long40 res_full;// long40 is a 40-bit wide integer
res_full = ((long40)a * (long40)b) << 1;// Getting 32-bit full result
res_high = (short)(((long40)a * (long40)b)>>15);
        // Getting the upper 16 bits of the result
```

```
res_high_round = (short)((((long40)a * (long40)b)+0x4000)>>15);
        // Getting the upper 16 bits of the rounded result
```

Following is the assembly code implementing the above (note that in some cases SIMD instructions are used with only one part of the result used):

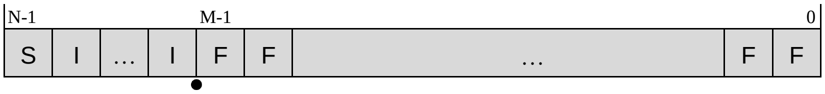
```
; a - D0.H, b - D2.H, res_full - D4, res_high - D6.H, res_high_round - D8.H
        mpy.x    d0.h,d2.h,d4; Calculate res_full into D4
        mpy.2t   d0,d2,d6; Calculate res_high into D6.H
        mpy.r.2t d0,d2,d; Calculate res_high_round into D8.H
```

NOTE

While the 8-bit signed fraction is not supported by DALU instructions, the casting of it during load and store is supported.

2.4.2.3 Signed accumulate (fraction)

A signed accumulate number has both an integer portion and a fractional portion, and thus has the decimal point in the middle of the number. The sign is the MSBit. Signed accumulate data types are supported only in D registers.



The numerical range of a signed fraction number X of width N and fractional width M is: $–2^{[N – M]} \le X \le [2^{[N – M]}-2^{-[M – 1]}]$ and granularity (value of 1 LSBit) of $2^{-[M – 1]}$.

This table shows the decimal and hexadecimal values of signed accumulate numbers, according to the width.

Table 2-13. Signed accumulate values

Width	Fraction width	Minimum value		One LSBit		Maximum value	
		Dec	Hex	Dec	Hex	Dec	Hex
20-bit	16-bit	–16.0	0x8_0000	2^{-15}	0x0_0001	$16.0–2^{-15}$	0x7_FFFF
40-bit	32-bit	–256.0	0x80_0000_0000	2^{-31}	0x00_0000_0001	$256.0–2^{-31}$	0x7F_FFFF_FFFF

The accumulate type is usually referred to as a type of fraction. In this case, the fraction portion is the actual “value” and the integer portion is referred to as the “extension” or the “guard bits.” The accumulate type is used for results of fraction multiplication by many fraction multiplication and MAC instructions.

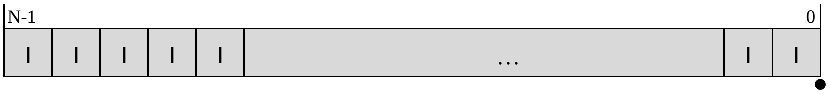
When the value of an accumulate type value is between –1.0 and $1.0–2^{[M-1]}$ (a legal fraction number value), all the extension bits have the same value as the fraction MSBit (the sign value). This situation is referred to as “clean guard bits.”

Signed accumulate types cannot be an input to a multiplication instruction.

A signed accumulate variable may be casted with saturation into a shorter data type when stored to memory. This kind of saturation, which is done on the way to the memory and does not change the value of the core register, is sometimes termed "limiting" in the document. It is performed with the ST.SRS instruction, see Section 3.2.2.3, "Scale, round and saturate store instructions."

2.4.2.4 Unsigned integer

An unsigned integer number has the decimal point right to the LSBit, and no sign. Unsigned integers of 20 and 40 bits are supported in D registers only. Unsigned integers of 8, 16 and 32 bits are supported either in R or D registers.



The numerical range of a signed integer number X of width N is: $0 \leq X \leq [2^N - 1]$ and granularity (value of 1 LSBit) of 1.

Table 2-14 shows the decimal and hexadecimal value of unsigned integer numbers according to the width.

Table 2-14. Signed Integer Values

Width	Name	Minimum value		One LSBit		Maximum value	
		Dec	Hex	Dec	Hex	Dec	Hex
8-bit	byte / char	0	0x00	1	0x01	255	0xFF
16-bit	word / short	0	0x0000	1	0x0001	65535	0xFFFF
32-bit	long (integer)	0	0x0000_0000	1	0x0000_0001	$2^{32}–1$	0xFFFF_FFFF
40-bit		0	0x00_0000_0000	1	0x00_0000_0001	$2^{40}–1$	0xFF_FFFF_FFFF

Unsigned integers are very similar to signed integers, except for multiplication (in which one of the multiplicands can be unsigned integer while another can be a signed or unsigned integer) and compare unsigned instructions.

2.4.2.5 Unsigned fractional

Unsigned fractions are similar to signed fractions, except for multiplication and compare instructions. SC3900FP provides support for unsigned comparisons. Unsigned fractions are supported only for D registers.

There are few multiply instructions that treat the input as unsigned fractions. These instructions support wide multiplication, and are not intended to support an unsigned fraction numerical type per se.

2.4.2.6 Non-linear address arithmetic

As follows, some LSU instructions support special non-linear arithmetic modes that can aid in address calculation in DSP kernels:

Modulo arithmetic — Where the result of a calculation is ensured to remain within the specified buffer (base address + buffer size M). The buffer size is flexible, however, the calculation has to be bounded so it does not deviate from the buffer by more than the buffer size (at most +M or -M from the buffer upper bound or lower bound, respectively).

Multiple-wrap modulo arithmetic
Where the result is ensured to remain always in the buffer; however, the buffer size is limited to sizes of powers of 2.

Reverse carry
A special arithmetic mode useful in addressing in FFT kernels.

The instructions supporting these modes are a subset of the load-store instructions that perform post-modify pointer updates such as (Rn)+, and of a small set of arithmetic instructions that are not related to loads or stores. The non-linear address arithmetic modes are enabled for these instructions by configuring the MCTL register. Some high-level languages (such as Embedded C) have support for pointers of modulo buffers as a data type, but in the general case using this type of arithmetic requires explicit programmer control. See Section 4.4, "Address modifier modes," for a detailed description of these non-linear arithmetic modes.

2.4.3 Saturation of fixed point numbers

In most of the DALU instructions (and in all the AGU mathematical instructions), when the calculated result cannot be represented in the destination (overflow condition), it is wrapped around, meaning that high bits are discarded.

For example, the sum of two C "short" numbers –27354 + –7382 = –34736 is 0x9526 + 0xE32A = 0x1_7850. This number cannot be represented as a short number, since lowest possible number is 0x8000 or –32768, and thus the result is taken without the overflow bit as 0x7850, which is 30800. This is the expected result of "C" variables (by definition for unsigned numbers and de-facto for signed numbers).

In many DSP algorithms, the desired behavior is saturation, meaning that when the result overflows and cannot be represented in the output format, it is replaced by the largest positive number that could be represented in the given number of bits (if the unsaturated result was positive), or by the smallest negative number that could be represented in the given number of bits (if the unsaturated result was negative).

The SC3900FP can optionally saturate the result in two cases, as follows:

- 16-bit signed integer or fraction saturated to a 16-bit value.
  In this case, all non-multiplicand inputs of the instruction are 16-bit wide and the output is 16-bit wide; if the output cannot be represented in 16-bit, it is saturated to 0x7FFF when the result is positive, and to 0x8000 when the result is negative. Note that, when a pair of 16-bit values is packed in a 40-bit register, the upper 8-bit of the register does not affect the 16-bit result, and is written as zero for 16-bit saturating instructions.
- 40-bit signed integer or fraction saturated to a 32-bit value.
  In this case, all non-multiplicand inputs of the instruction are 40-bit wide and the output is 40-bit wide; if the output cannot be represented in 32-bits (not 40-bits), it is saturated to 0x00_7FFF_FFFF when the result is positive, and to 0xFF_8000_0000 when the result is negative. Note that, although all variables are 40-bit wide, the number is saturated to a 32-bit value so that the result is a fractional number ($–1.0 \leq X < 1.0$).

Saturation that affects the results in the destination registers is supported only for D registers. Note that on-the-fly saturation of data that is stored on the way to the memory, without changing it in the register, could be seen as saturated casting that is performed during the specialized ST.SRS instructions. This is supported only for fractional data types. For more information, see Section 3.2.2.3, "Scale, round and saturate store instructions."

Saturation is marked by a flag in the mnemonic of the relevant instructions. In addition, saturation can be activated for some legacy instructions by setting the SM2 bit or SM bit in SR register for 16-bit saturation or 32-bit saturation, respectively.

Some specific instructions support saturating 20-bit results to a 20-bit signed number, and saturating 40-bit results to a 40-bit signed number (for example, arithmetic shift, absolute value).

2.4.4 Data types supported in the SC3900FP

2.4.4.1 Overview of supported data types

This section gives a general overview of the data types that are supported in SC3900FP. As explained in previous sections, the SC3900FP may have a different level of support for different data types. The following descriptive support levels are used for the different data types:

Key instructions — Special ISA support is limited to a few instructions that accelerate some key arithmetic operations.

Basic — There is support that is enough for a high level compiler to map the important arithmetic operations (addition, subtraction, multiplication, shifting, comparison and logic operations). SIMD support is partial or does not exist.

Full — Support for basic arithmetic, additional operations and in addition SIMD variants.

Extensive — There is ISA support for many arithmetic variants, usually including Complex arithmetic; some of these instructions are supported with the aid of compiler intrinsic functions.

A high-level summary of the data type support in SC3900FP is described in Table 2-15. "D:" specifies the level of support for D registers, and "R:" specifies the level of support for R registers. An empty cell means that this data type may be supported through emulation.

Table 2-15. Overview of Data Type Support in SC3900FP

Data type		8 bits	16 bits	20 bits	32 bits	40 bits	64 bits
Integer	Signed, no saturation	D: full, via casting R,D: basic, via casting	D: extensive R: basic, via casting	D: extensive (–)[1]	D: extensive R,D: basic	D: extensive (–)[1]	D: key instructions
	Signed, saturated	—	D: full	—	D: full	—	—
	Unsigned, no saturation	R,D: basic, via casting	D: basic R: basic, via casting	—	D: basic R: basic	D: basic	—
	Unsigned, saturated	D: key instructions	—	—	—	—	—

Table 2-15. Overview of Data Type Support in SC3900FP (continued)

Data type		8 bits	16 bits	20 bits	32 bits	40 bits	64 bits
Fraction	Signed, no saturation	D: full, via casting	D: extensive	D: extensive (–)[1]	D: full	D: extensive (–)[1]	—
	Signed, saturated	D: full, via casting	D: extensive	—	D: extensive	—	—
	Unsigned, no saturation	—	D: basic (–)[1]	—	D: basic (–)[1]	D: basic (–)[1]	—
Floating point (single precision)		—	—	—	D: full	—	—

Note:

[1] Not supported as multiplication sources

Arithmetic support for the different data types is described in detail, in Section 3.2, "DALU data types."

2.4.4.2 8-bit data types

The byte-wide data types have full load-store support (including SIMD); however, arithmetic support is usually performed by way of casting to a wider data type, using the native instructions of that wider data type. This casting is illustrated in Figure 2-14. The following data types are supported:

Table 2-13. 8-bit data types supported in SC3900FP

Data type	Support
Signed integer, no saturation	Supported in both R and D registers. A single byte in a register is casted to 32 bits (in an R register), or 40 bits (in a D register). Two bytes could be packed in a D register, in which cases each byte is casted to 20 bits.
Unsigned integer, no saturation	Supported in the same way as the signed integer byte described above, with dedicated unsigned load instruction. The arithmetic operations on the casted instructions may be performed as signed because the numeric range of the unsigned byte is included in the numeric range of the wider signed type it is casted to.
Unsigned integer, saturated	Supported with very few instructions using D registers, such as CLIP, primarily used in video kernels.
Signed fraction, no saturation	Supported only in D registers. Dedicated loads cast fractional bytes into wider fraction accumulate data types (for a single variable, casting is into 40 bit fraction; for two packed variables, casting is to 20 bit fractions). Processing then continues with native accumulate non-saturated data type instructions.
Signed fraction, saturated	Supported only in D registers. Dedicated loads cast fractional bytes into wider fraction data types (for a single variable, casting is into 40 bit fraction; for two packed variables, casting is to 20 bit fractions). Processing then continues with native 16-bit saturated data type instructions.

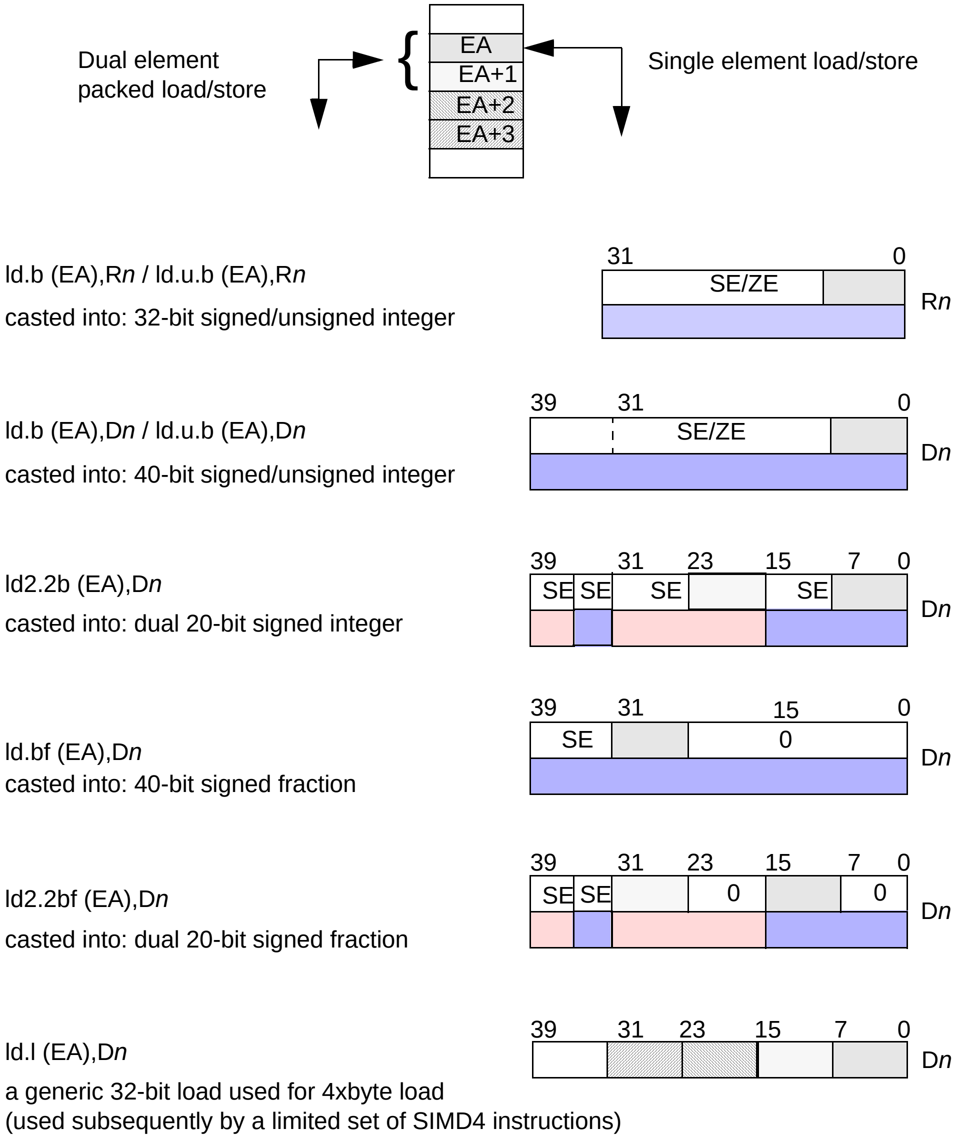

Dual element
packed load/store
EA
EA+1
EA+2
EA+3
Single element load/store
ld.b (EA),Rn / ld.u.b (EA),Rn
casted into: 32-bit signed/unsigned integer
31
0
SE/ZE
Rn
ld.b (EA),Dn / ld.u.b (EA),Dn
casted into: 40-bit signed/unsigned integer
39
31
0
SE/ZE
Dn
ld2.2b (EA),Dn
casted into: dual 20-bit signed integer
39
31
23
15
7
0
SE
SE
SE
SE
Dn
ld.bf (EA),Dn
casted into: 40-bit signed fraction
39
31
15
0
SE
0
Dn
ld2.2bf (EA),Dn
casted into: dual 20-bit signed fraction
39
31
23
15
7
0
SE
SE
0
0
Dn
ld.l (EA),Dn
a generic 32-bit load used for 4xbyte load
(used subsequently by a limited set of SIMD4 instructions)
39
31
23
15
7
0
Dn

Figure 2-14. Casting of byte data types during memory transfers

Since byte-wide SIMD instructions are casted to 20 bits, they have the same bandwidth as their 16-bit or 20-bit counterparts, supporting two operations per register. There are some dedicated key instructions (for video processing kernels), which can perform 4-way SIMD byte operations. Note that SIMD4 instructions are also supported, using more than one data register, for example NEG.S.4W Da:Db,Dn:Dm.

2.4.4.3 16-bit data types

The 16-bit-wide data types usually have extensive support in the ISA using D registers. Support in R registers exists but is more limited. The 16-bit data types share some of their instructions with the 20-bit accumulate data types because, for instructions such as ADD, the lower 16-bits of the result are the same for 20 bits and 16 bits. In most cases, where there is a difference, the ISA has independent versions for 16 and 20 bits. In the DALU, the most efficient use of 16-bit arithmetic is as part of SIMD operations, which pack two 16-bit variables in one D registers.

The following table lists the 16-bit data types that are supported in SC3900FP.

Table 2-15. 16-bit data types supported in SC3900FP

Data type	Support
Signed integer, no saturation	Supported extensively in D registers. Data loaded into R registers is casted into 32-bits, and is processed with 32-bit instructions. 16x16 bits multiplication is natively supported in this data type, and have many variants.
Signed integer, saturated	Supported only for D registers, either with dedicated instructions, or, for legacy SC3850 instructions, by activating the SM2 mode bit in SR.
Unsigned integer, no saturation	Supported at the basic level in both R and D registers. Data loaded into R registers is casted to unsigned 32 bits, and is processed with 32-bit unsigned instructions.
Signed fraction, no saturation	Supported only in D registers. This data type has extensive support, due to the many variants of 16x16 multiplication instructions.
Signed fraction, saturated	Supported only in D registers. Nearly always a non-saturating instruction has a saturating counter part.
Unsigned fraction, saturated	Limited support, only in D registers.

NOTE

The numerical conditions for performing 16-bit saturation are different from 32-bit saturation. The 32-saturation studies the full extension before saturation, while 16-bit saturation studies 2 bits in the extension.

2.4.4.4 20-bit accumulate data types

In general, 20-bit accumulation is used for representing intermediate results in registers, and not as a format used to store data in the memory. Usually, data is loaded from memory as shorter data types (byte or words), and casted into 20-bits in the D registers, or generated as the result of the calculation of shorter operands (accumulation). After the calculation is done, the 20-bit operand is normally saturated into a shorter data type when stored into memory. Thus, there are no dedicated load or store instructions that save 20-bit operands into memory. If required, the whole D register is saved, consuming 64 bits in memory. However, SIMD2 operations of 20-bits uses the available width of the D register most efficiently, hence this data type is very useful and has extensive support in the ISA. Normally, 20-bit variables are not saturated to 20 bits (although, a few specific instructions do).

The following 20-bit data types are supported in SC3900FP, more or less equivalently:

- Signed integer, no saturation
- Signed fraction, no saturation

2.4.4.5 32-bit data types

32-bit data types are important for supporting native C integer data types, and as multiplication sources for 32-bit multiplication. SIMD is supported for D registers by using register pairs as operands. R registers support only integer, non-SIMD and not-saturating types.

This table lists the 32-bit data types supported in SC3900FP.

Table 2-16. 32-bit data types supported in SC3900FP

Data type	Support
Signed integer, no saturation	Supported extensively for D registers and with basic support for R registers. Usually used for algorithms using the standard C “int” data type, including DSP algorithms. Can be used as a native input to multiplication instructions. SIMD is supported by using instruction pairs as operands.
Signed integer, saturated	Supported for D registers only.
Unsigned integer, no saturation	This data type is given basic support both for R and D registers, it is used mainly for supporting the C “unsigned int” data type.
Signed fraction, no saturation	Supported in full for D registers only. Can be used as a native input to multiplication instructions. SIMD is supported by using instruction pairs as operands.
Signed fraction, saturated	Supported extensively, for D registers only.
Unsigned fraction, no saturation	Basic support for D registers only.

NOTE

The numerical conditions for performing 16-bit saturation are different from 32-bit saturation. The 32-saturation studies the full extension before saturation, while 16-bit saturation studies 2 bits in the extension.

2.4.4.6 40-bit accumulate data types

In general, 40-bit accumulation is used for representing intermediate results in registers. Data is usually loaded from memory as shorter data types (such as 32 bits), and extended into 40-bits in the D registers, or generated as the result of calculation of shorter operands (accumulation). After the calculation is done, the 40-bit operand is normally saturated into a shorter data type when stored into memory. If required, the whole D register is saved, consuming 64 bits in memory (24 bits remain unused). 40-bit accumulation gives the most accuracy, using the full D register. 40-bit operands cannot be multiplied by single instructions, a portions of them (32 or 16 bit) has to be used. SIMD2 operations of 40-bits are supported by using register pairs as operands. 40-bit variables are not saturated to 40 bits.

This table lists the 40-bit data types supported in SC3900FP.

Table 2-17. 40-bit data types supported in SC3900FP

Data type	Support
Signed integer, no saturation	Supported only in D registers
Unsigned integer, no saturation	Mostly uses some of the signed versions of the instructions, with an addition of a few dedicated instructions.
Signed fraction, no saturation	Supported only in D registers
Unsigned fraction, no saturation	Mostly uses some of the signed versions of the instructions, with an addition of a few dedicated instructions.

2.4.4.7 64-bit data types

64-bit data is located in two adjacent D registers. Some key instructions enable efficient processing of 64-bit variables, such as addition/subtraction (ADD.LL), and a single instruction for 64-bit shift. Other operations such as multiplication should be emulated by an instruction sequence.

2.4.4.8 Floating point data types

The SC3900FP supports single precision IEEE compliant floating point arithmetic. Floating point ISA operate on D registers, using the 32 lower bits. SIMD2 instructions operating on two D registers in parallel are also supported. For more information on floating point support see Section 3.3.5, “Floating point”.

2.5 Assembly semantics

2.5.1 Serial semantics between VLES

From a functional point of view, a sequence of VLES has serial semantics. This means that all destinations from the previous VLES are assumed updated before they are used by the current VLES. The following is an example:

```
[ add.x d0,d1,d2  ld.l (r0)+,d4 ]   ; VLES 1
[ mpy.x d2,d4,d5  adda r0,r5,r6 ]   ; VLES 2
```

The values of d2, d4 and r0 that are used as source operands in VLES 2 should be assumed as holding the updated values that the instructions in VLES 1 updated as destinations. The core attempts to forward the destination data without interlock cycles. If it cannot, the core inserts interlock cycles until the destinations are available.

2.5.2 Parallel and serial semantics inside a VLES

In most cases, the SC3900FP uses parallel semantics for the instructions in the same VLES. This means that all sources of all the instructions are assumed to be read at the beginning of the VLES, and all destinations are updated at the end; in other words, a destination that is updated by the VLES does not affect other instructions through their source operands. The following is an example:

```
[ add.x d0,d1,d2  st.w d2,(r0) ]
```

In this example, d2 is updated by the ADD instruction, however the value of d2 that is stored to memory is that of d2 before it was updated by the ADD. The following is another example:

```
[ tfr.x d0,d1     tfr.x d1,d0 ]
```

This VLES switches the values of d0 and d1, because each TFR uses the source operand before it was updated by the other instruction. The same semantics hold when mixing memory loads and stores to the same location in the same VLES, as follows:

```
[ st.l d0,(r0) ld.b (r1),d1 ] ; R0 holds 0x100, R1 holds 0x101.
```

In this example, address $101 is accessed by both the load and the store. Parallel semantics dictate that the memory is read with the old value, before it was updated by the store.

An outcome of this definition is that, in most cases, the same resource from two different instructions cannot be updated in the same VLES. SC3900FP semantics allow some instructions to update the same resource in the same VLES. There are two types of this case, as follows:

- Some Status bits that are updated by more than one instruction in the VLES are updated as the logical OR of the value generated by each instruction. For example:
  - Two compare instructions updating the same predicate
  - The Scale (S) flag that is set by the ST.SRS instruction
- Some instructions employ serial semantics when updating certain destinations. The instructions in the VLES are assumed to be ordered from left to right. The “last” (right most) instruction, as written in the assembly code, will be the one that updates the resource. The update of that resource by earlier instructions is overridden. For these instructions, with respect to the special resource in question, it could be assumed that they execute one after the other, from left to right.

NOTE

The cases of serial semantics are limited to specific instructions and specific operands of those instructions. This means that the general parallel semantics mostly hold, and they are overridden for some specific cases.

2.5.2.1 Cases of serial semantics within a VLES

The cases where serial semantics are employed are as follows:

1. Two memory writes to overlapping bytes. For example:

   ```
   [ st.l d0,(r0) st.b d1,(r1) ] R0 holds 0x100, R1 holds 0x101.
   ```

   In this example, bytes 0x100 to 0x103 are written by the ST.L. The store by the ST.B instruction executes “after it” so after the VLES, byte 0x101 holds the value originating from d1.

   The memory bytes are stored with the same priority as before (meaning that byte 0x101 originates from the ST.B). However, note that r0 was incremented by the ST.L, and r0 is also used as the source of the ST.B. For r0, the semantics remain parallel, meaning, that the value of r0 that is used by the ST.B instruction is that or r0 before it was updated by the ST.L. This example shows that the serial semantics apply not to the whole instruction, but only to the specific resource (in this case, the memory).

2. Two push or two pop instructions (explicit and implicit) can be used in a VLES. With some exceptions, two such instructions are considered to execute serially, meaning that the second instruction uses the SP after it is incremented or decremented by the first instruction.
3. The ACS.H.2W and ACS.L.2W instructions write bits to the BTR registers and shift them, in the order in which the instructions appear in the VLES. These instructions must be grouped either as two or four in a VLES.

2.6 Assembly re-ordering requirements

The core expects the assembler to re-order the instructions in the VLES, according to their type:

- A PCU instruction
- LSU instructions
- An IPU instruction
- The DALU instructions

This list means, for example, that if there is a PCU instruction in the VLES, it should be encoded first (closest to the start of the VLES). Within each instruction type, the assembler should in general keep the order of instructions as they appear in the source assembly, to preserve the semantic order for the cases where it is required. VLES ordering is described in more detail in Section 6.1.2, “VLES ordering and limitations.”

Chapter 3
Data Arithmetic Logic Unit (DALU)

The Data Arithmetic Logic Unit (DALU) is the main computation unit responsible for performing intensive arithmetic operations required by advanced digital processing algorithms. The DALU contains a register file of 64 registers, each of which is 40-bit wide. Mathematical operations are performed by the DALU on data read from this register file, and results are stored back into that same file. In addition, the DALU supports load and store operations from memory to the register file and back, using LSU load and store instructions. The DALU executes up to four instructions in parallel in each cycle, reading up to 24 source registers from the register file and writing up to eight registers as results back to it. Up to 16 registers can be loaded from memory and up to 16 registers can be stored to memory utilizing two LSU instructions.

The DALU consists of the following main components:

- A register file of 64 data registers, each of which is 40-bit wide
- Four symmetric Data Multiply Units (DMUs), each comprised of multiply-accumulate (MAC) logic, adders, shifters, bit manipulation logic as well as logic for supporting DSP application specific instructions
- Connectivity to the memory system over two 256-bit read buses and two 256-bit write busses, controlled by LSU load and store instructions.
  In addition to connectivity with the memory system, the DALU also has connectivity with the AGU and PCU for control-oriented operations.

Although DALU can execute up to four instructions every cycle, execution of DALU instructions takes one to three pipe stages according to the instructions, and thus interlocks of up to two cycles are generated when a register is used as a source for a three pipe stages instruction, one cycle after it is updated by a previous DALU instruction.

The following sections provide a deeper understanding of the DALU structure and capabilities.

3.1 DALU description

3.1.1 Block diagram

Figure 3-1 is a high level block diagram of the DALU.

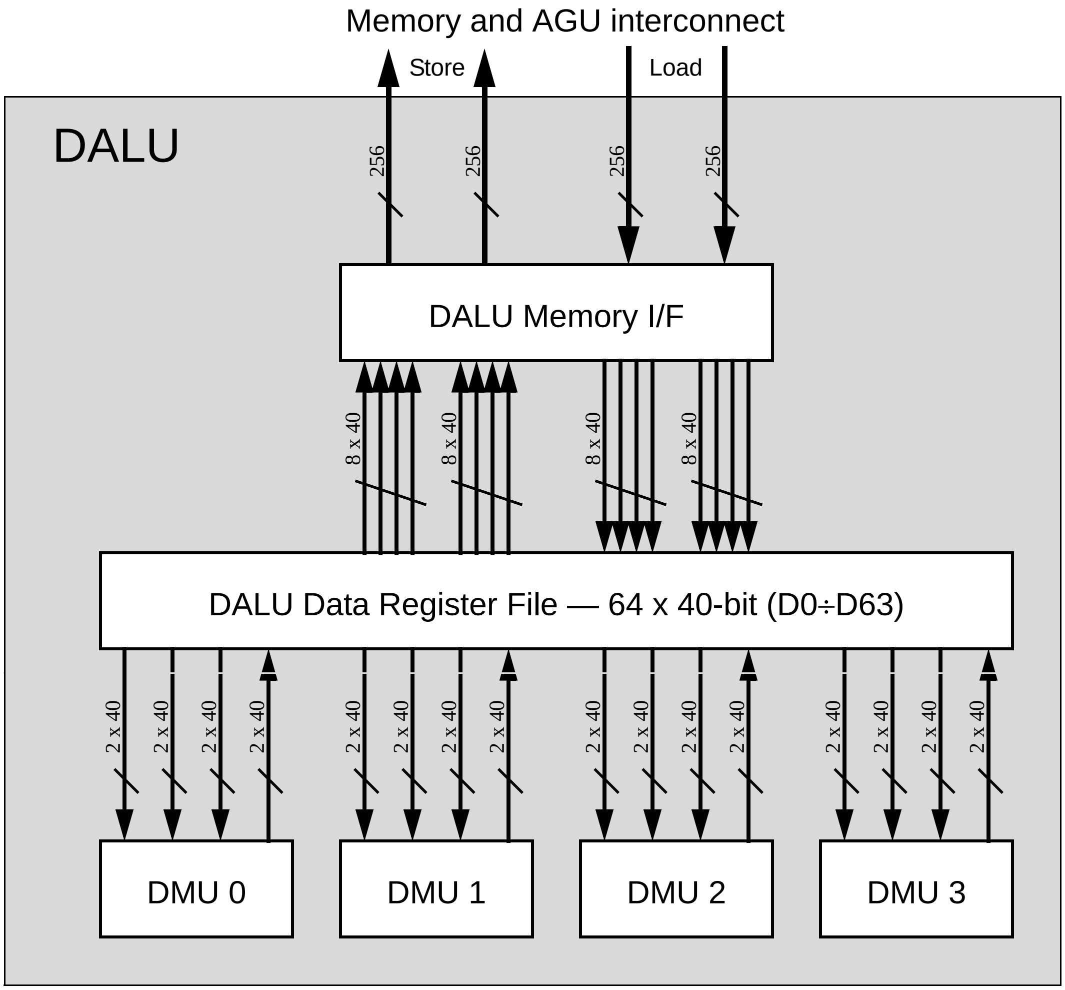


Figure 3-1. DALU high-level block and connectivity diagram

Each of the following DMU units can execute a single DALU instruction every cycle, reading up to three register pairs from the register file, calculating the result and writing up to one register pair back to the register file. A register pair is a composite of two registers, one even (D0, D2, D4, …, D62) and one odd (D1, D3, D5, …, D63). In most of the instructions that access a register pair, the pair is encoded consecutively (second register number equals the first register number plus one). The assembler notation for a register pair is Dn:Dm (for example, D5:D6).

There are few DALU instructions that activate two DMU units in one 64-bit instructions (either DMU0 and DMU1 or DMU2 and DMU3). For such instructions, there are up to six register pair inputs and two register pair outputs.

Note that for some of the instructions some of the input operands are read from the register file as a 32-bit value (without the extension), for example, multiplicands input for all multiply and MAC operations.

The DALU memory I/F block is controlled by the AGU load, store and move instructions. As shown in Figure 3-1, the SC3900FP core can update two sets of up to eight registers servicing a load or a move instruction and can write to memory or to another register value derived from two sets of up to eight registers due to a store or move operation. When all eight registers are accessed, each of the eight registers must have a different modulo 8 value (the modulo 8 value for Dn is n%8, for example, for D35, the value is 35%8=3). In most of the AGU instructions, all of the accessed registers are sequential; however, a few other groups are available, such as four even registers (for example, D2:D4:D6:D8). Other special orders such as modulo 8 wrap around ordering are defined for support of specific applications such as FFT and DFT. The assembler notation for a sequential register series is a list of the registers delimited by ":", or the first and last registers of the list separated by "::" in short hand notation. For example, a sequential register octet is Da:Db:Dc:Dd:De:Df:Dg:Dh (for example, D3:D4:D5:D6:D7:D8:D9:D10), or Da::Dh (for example, D3::D10). Special load instructions can write to 16 registers in a single instruction, utilizing both DALU load ports. No other load or move instruction is allowed to be grouped with such 16 registers load instruction (since all load resources are already in use). In a similar way, special store instructions can read from 16 registers in a single instruction, utilizing both DALU store ports. No other store or move instruction is allowed to be grouped with such 16 registers store instruction (since all store resources are already in use). See Section 4.6, "Load and store instructions," for more details on load store and move operations.

3.1.2 DALU pipeline

The DALU execution units work in a three stage pipeline: stage M, stage Y and stage E, where M is the earlier stage and E is the latest of the three stages). All DALU instructions (as well as all SC3900FP core instructions) are single cycle instructions, and thus four DALU instructions can be processed at each cycle. The three state pipeline implementation can generate interlock (stall) cycles due to resource conflicts, according to the following:

All DALU instructions write the results back to the DALU register files at stage E.

DALU instructions read register values from the register files at different pipe stages, according to instruction type. The following is a rough description of major DALU instruction groups implementation:

- All operands of floating point instructions are read at stage M (two/three cycle interlock if the value was updated during the previous cycle).
- Multiplicand operands of multiply and MAC operations are read at stage M. Accumulative input for MAC operations is read at stage E (no interlocks).
- Offset of multi-bit shift operations is read at stage Y (one cycle interlock if the value was updated during the previous cycle). Input value of multi-bit shift operations is read at stage E (no interlocks).
- Operands of basic arithmetic and logic operations (add, sub, and, or, and so on) are read at stage E (no interlocks).
- Compare instructions and other instructions that update the predicate bits read the input operands at stage E (no interlocks).

- Special instructions, such as insert, extract, SAD4, and max find, read part or all of the input operands at stage Y (one cycle interlock if the value was updated during the previous cycle).
- All types of basic arithmetic (add, sub, cmp), SOD, logic operations (all SC3400/SC3850 legacy 1 stage operations except shifts) are executed during stage E.

Loaded data is written to the DALU register file at stage M; therefore, there is no interlock between load data into the DALU register file and usage of it by a DALU instruction (zero load to use latency). Stored data is read from the DALU register file at stage E; therefore, there is no interlock between update of a DALU register by a DALU instruction and store its value to memory (zero use to store latency).

Moved data between registers (via LSU or IPU instruction) is read at stage M; therefore, there is a two cycle interlock between the update of a DALU register by a DALU instruction and the MOVE of it's value to another register (either DALU or AGU register).

3.2 DALU data types

The DALU execution units supports fixed point and floating point arithmetic on Dn registers, in a few data types, as described in the following sections. Other data types are supported in memory, and converted (casting) during load and store to the DALU register file supported data types.

For an introduction on data types and the implementation of mathematical types by the SC3900FP architecture, please see Section 2.4, "Data type support." This chapter describes in more detail the representation of the various data types inside the DALU registers as well as the behavior of load and store different data types from memory to the DALU register file and vice versa.

3.2.1 Fixed point representation in DALU registers

The DALU register file can hold several fixed point numerical data types that can be manipulated by different DALU instructions (each of the relevant instructions supports one or more representations).

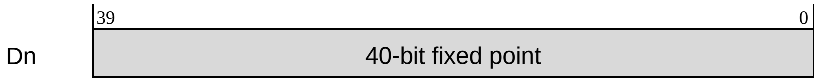
The SC3900FP has an embedded support of both real fixed point numerical data types and complex fixed point numerical data types.

Complex fixed point numerical data types contain two values representing the real and imaginary parts of the complex number. The instruction support of complex numerical data types include complex multiplications, with optional conjugate multiplication and optional multiplication by J, as well as other complex mathematics instructions.

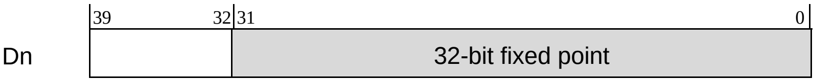
3.2.1.1 Real fixed point representation in DALU registers

The following are the real fixed point numerical data format in Dn registers that are supported by DALU instructions.

- 40-bit fixed point number.
  — Reside in a single DALU register.



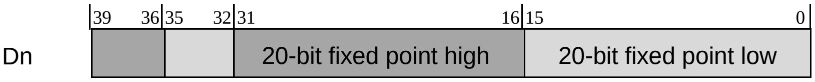
  - Either integer or fractional number.
  - Either signed or unsigned number. If signed then bit 39 is the sign.
  - Optionally saturated if cannot be represented as a 32-bit signed number.
- 32-bit fixed point number:
  - Reside in low 32-bits of a single DALU register.



The upper 8-bit (extension) is not used as an input and is undefined. Zero extended, sign extended or 40-bit value can be written to it according to the used instruction.

Dn.M = Dn[31:0].

  - Either an integer or a fractional number
  - Either a signed or an unsigned number
- 20-bit fixed point number.
  - Two numbers reside in a single DALU register.

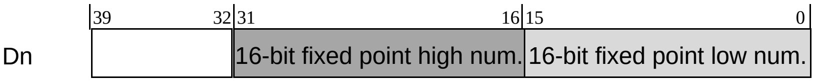


The high number resides in bits 39:36 and 31:16, while the low number resides in bits 35:32 and 15:0.

Dn.WH[19:16] = Dn[39:36]; Dn.WH[15:0] = Dn[32:16].

Dn.WL[19:16] = Dn[35:32]; Dn.WL[15:0] = Dn[15:0].

  - Either an integer or a fractional number
  - Either a signed or unsigned number. If signed then bit 39 is the sign of the high word and bit 35 is the sign of the low word.
- 16-bit fixed point number.
  - Two numbers reside in a single DALU register.



The high number resides in bits 31:16, while the low number resides in bits 15:0.

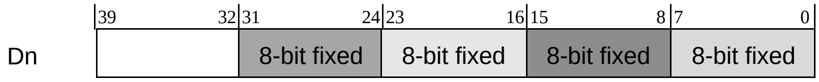
Dn.H = Dn[32:16].

Dn.L = Dn[15:0].

The upper 8-bit (extension) is not used as input and is undefined. Zero extended, sign extended or 20-bit value can be written to it according to the used instruction.

  - Either an integer or a fractional number
  - Either a signed or an unsigned number. If signed bit 31 is the sign of the high word and bit 15 is the sign of the low word.
  - Optionally saturated if cannot be represented as a 16-bit signed number.
- 8-bit fixed point number.

- Four numbers reside in a single DALU register.



The four numbers resides in bits 31:24, 23:16, 15:8 and 7:0.

Dn.HH = Dn[32:24].

Dn.HL = Dn[23:16].

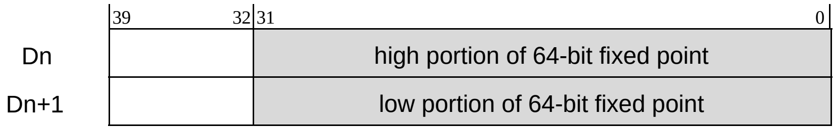
Dn.LH = Dn[15:8].

Dn.HL = Dn[7:0].

The upper 8-bit (extension) is not used as input and is undefined. Zero or other value can be written to it according to the used instruction.

- Very few application-specific instructions are supporting this format in the SC3900FP core (for example SAD8).

- 64-bit fixed point number:
  - Reside in low 32-bits of two DALU register.


39
32
31
0
Dn
high portion of 64-bit fixed point
Dn+1
low portion of 64-bit fixed point

The upper 8-bit (extension) f both registers is not used as an input and is undefined. Zero extended, sign extended can be written to the extension of the upper |(first) register according to the used instruction and zero is written to the extension of the lower (second) register.

Dn.LL[63:32] = Dn[31:0].

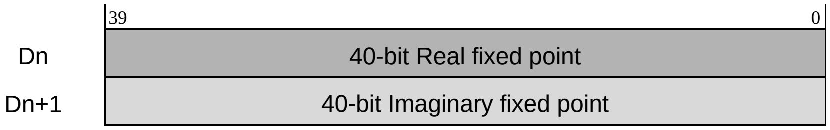
Dn.LL[31: 0] = Dn+1[31:0].

- An integer-only number (no fractional support)
- Either a signed or an unsigned number

3.2.1.2 Complex fixed representation in DALU registers

The following complex fixed point numerical data types are supported by different DALU instructions:

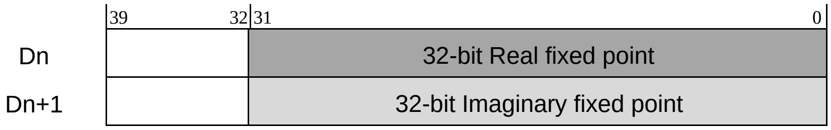
- 40-bit fixed point complex number.
  - Reside in two DALU registers.



- Real part reside in Dn and Imaginary part in Dn+1
- Either integer or fractional number.
- Either signed or unsigned number. If signed bit 39 is the sign.

- Optionally saturated if cannot be represented as a 32-bit signed number.

- 32-bit fixed point complex number:
  - Reside in low 32-bits of two DALU registers.



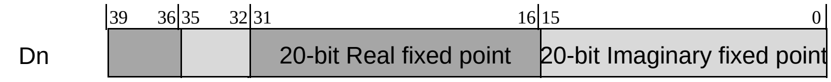
The upper 8-bit (extension) of both registers is not used as an input and is undefined. Zero extended, sign extended or 40-bit value can be written to it according to the used instruction.

Real part=Dn.M = Dn[31:0].

Imaginary part=Dn+1.M = Dn+1[31:0].

  - Either integer or fractional number.
  - Either signed or unsigned number.

- 20-bit fixed point complex number.
  - A single complex number reside in a single DALU register.



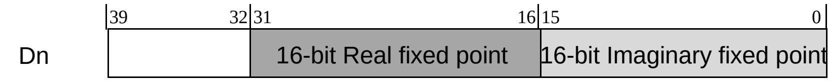
The real part resides in bits 39:36 and 31:16, while the imaginary part resides in bits 35:32 and 15:0.

Real part in Dn.WH – Dn.WH[19:16] = Dn[39:36]; Dn.WH[15:0] = Dn[32:16].

Imaginary part in Dn.Wl – Dn.WL[19:16] = Dn[35:32]; Dn.WL[15:0] = Dn[15:0].

  - Either an integer or a fractional number
  - Either a signed or an unsigned number. If signed then bit 39 is the sign of the real part and bit 35 is the sign of the imaginary part.

- 16-bit fixed point complex number.
  - A single complex number reside in a single DALU register.



The real part resides in bits 31:16, while the imaginary part resides in bits 15:0.

Real part=Dn.H = Dn[32:16].

Imaginary part=Dn.L = Dn[15:0].

The upper 8-bit (extension) is not used as input and is undefined. Zero extended, sign extended or 20-bit value can be written to it according to the used instruction.

  - Either integer or fractional number
  - Either signed or unsigned number. If signed bit 31 is the sign of the real part and bit 15 is the sign of the imaginary part.
  - Optionally saturated if cannot be represented as a 16-bit signed number.

3.2.2 Load and store support by the DALU register file

The DALU supports both a wide set of load instructions that load data from the memory to the DALU register file and store instructions that store data from the DALU register file to the memory, with various conversions and manipulations on-the-fly.

Load and store instructions enable conversion (casting) of numerical types while moving data from memory to Dn registers and back. This conversion enables compression of data in memory (for example, calculate 40-bit accumulate data and save to memory is only a 16-bit fraction value of it), and good support for types, such as such as integer bytes and fraction bytes, that do not have full, direct support in Dn registers.

In addition, the store operation optionally can Scale, Round and Saturate (SRS) the value while storing it to memory, thereby reducing the application instruction count.

Load and store instructions can load or store multiple numbers in a single instruction, according to the requirement (SIMD2, SIMD4, SIMD8, SIMD16 or SIMD32). When two load instructions, two store instructions or a load/store instruction with move instruction need to be grouped in the same VLES, each load or store instruction can access memory with up to 256-bit wide and access up to eight registers. A single load transaction, a single store transaction or a grouped load transaction with store transaction can access memory with up to 512-bit wide and access up to 16 registers (yielding a maximum bandwidth of 1024-bit per cycle between the core and the memory).

Note that in the following paragraphs all DALU register sequences are shown as linear (each register number after the first register is the number of the previous register plus one, for example, D5:D6:D7:D8), although other sequences are allowed for some of the instructions. See the load and store description in Appendix C, “Instruction Set” for more details.

3.2.2.1 Load of fixed point numbers

Signed fix point numbers are loaded from memory of 8-bit, 16-bit, 32-bit or 64-bit width into a Dn register of 20-bit or 40-bit width. During the load operation, the numbers are “Sign Extended,” meaning that the MSBit of the number in memory (the sign) is copied to all of the bit above the value in the register. This behavior is marked in the following register maps as ‘SE.’ Signed fix numbers are also marked in blue.

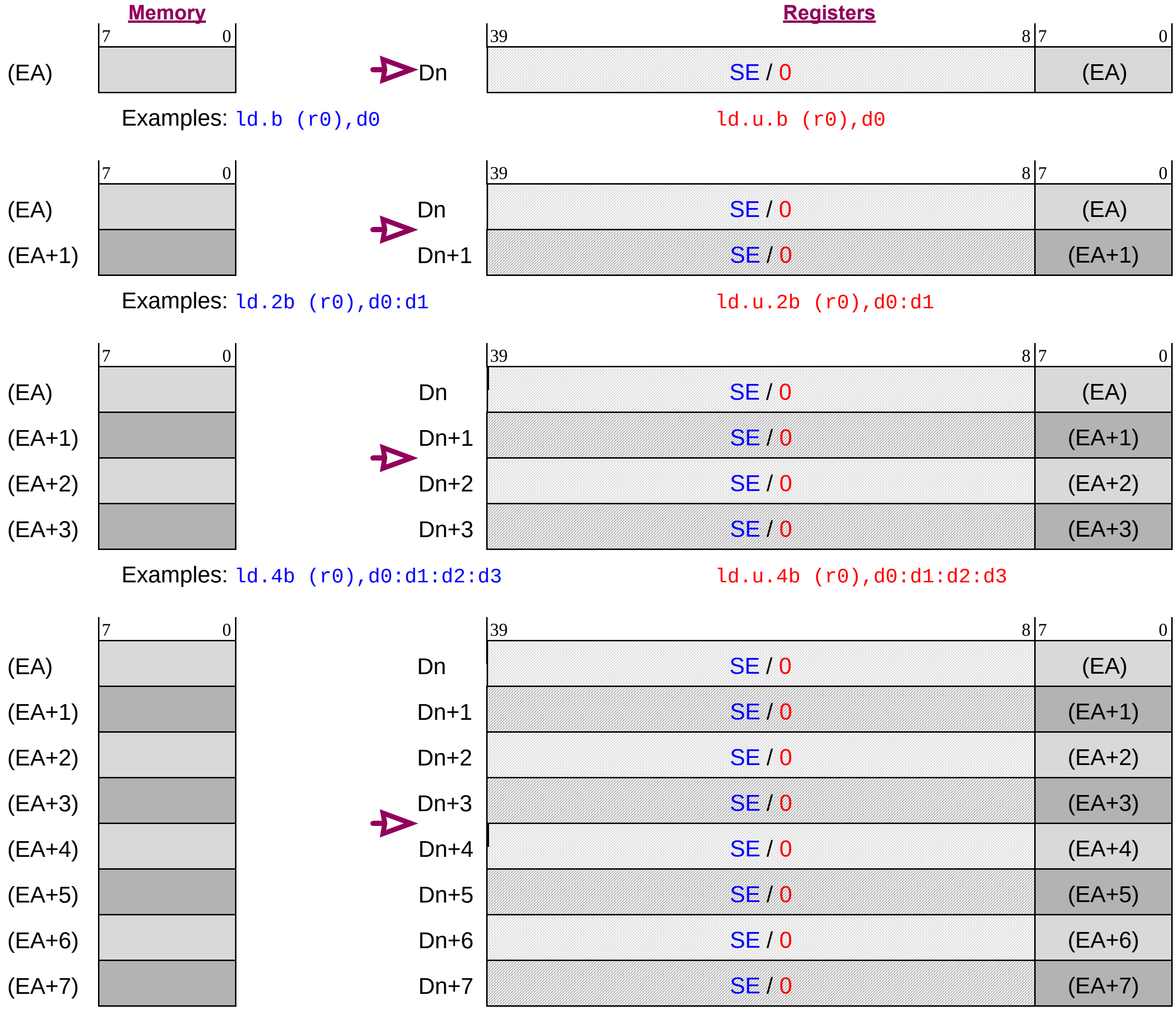
Unsigned numbers are loaded from memory of 8-bit, 16-bit, 32-bit or 64-bit width into a Dn register of 20-bit or 40-bit width. During the load operation, the numbers are “Zero Extended,” meaning that zero is copied to all of the bit above the value in the register. This behavior is marked in the following register maps as ‘0.’ Unsigned fix numbers are also marked in red.

When fraction numbers are loaded from memory of 8-bit width into a Dn register of 20-bit or 40-bit width or from memory of 16-bit width into a Dn register of 40-bit width, all of the bits below the value are padded with zeros. This behavior is marked in the following register maps as ‘0.’

Fixed point 40-bit wide numbers can be loaded to the Dn register from 64-bit wide numbers in memory. The lower 40 LSBits of the 64-bit value are loaded into the register. High 24 bits of the memory are not used by this load.

The fixed point numbers load instructions are supported by LSU instructions, with different addressing generation and register allocations.

The following figures show the SC3900FP support of fixed point numbers load from memory to Dn registers.



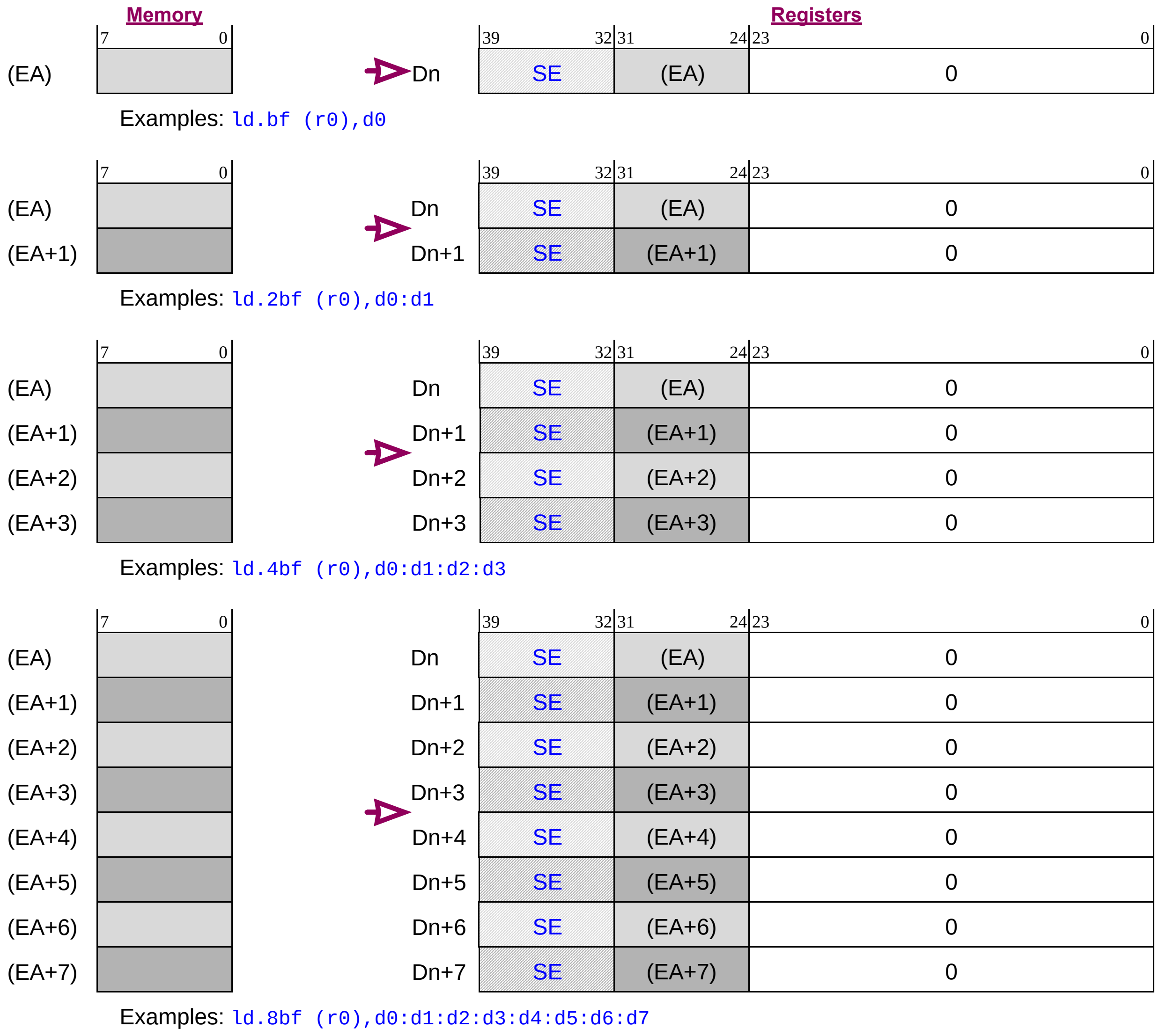
Examples: `ld.b (r0),d0` `ld.u.b (r0),d0`

Examples: `ld.2b (r0),d0:d1` `ld.u.2b (r0),d0:d1`

Examples: `ld.4b (r0),d0:d1:d2:d3` `ld.u.4b (r0),d0:d1:d2:d3`

Examples: `ld.8b (r0),d0:d1:d2:d3:d4:d5:d6:d7` `ld.u.8b (r0),d0:d1:d2:d3:d4:d5:d6:d7`

Figure 3-2. 8-bit signed/unsigned integer in memory into 40-bit signed/unsigned integer



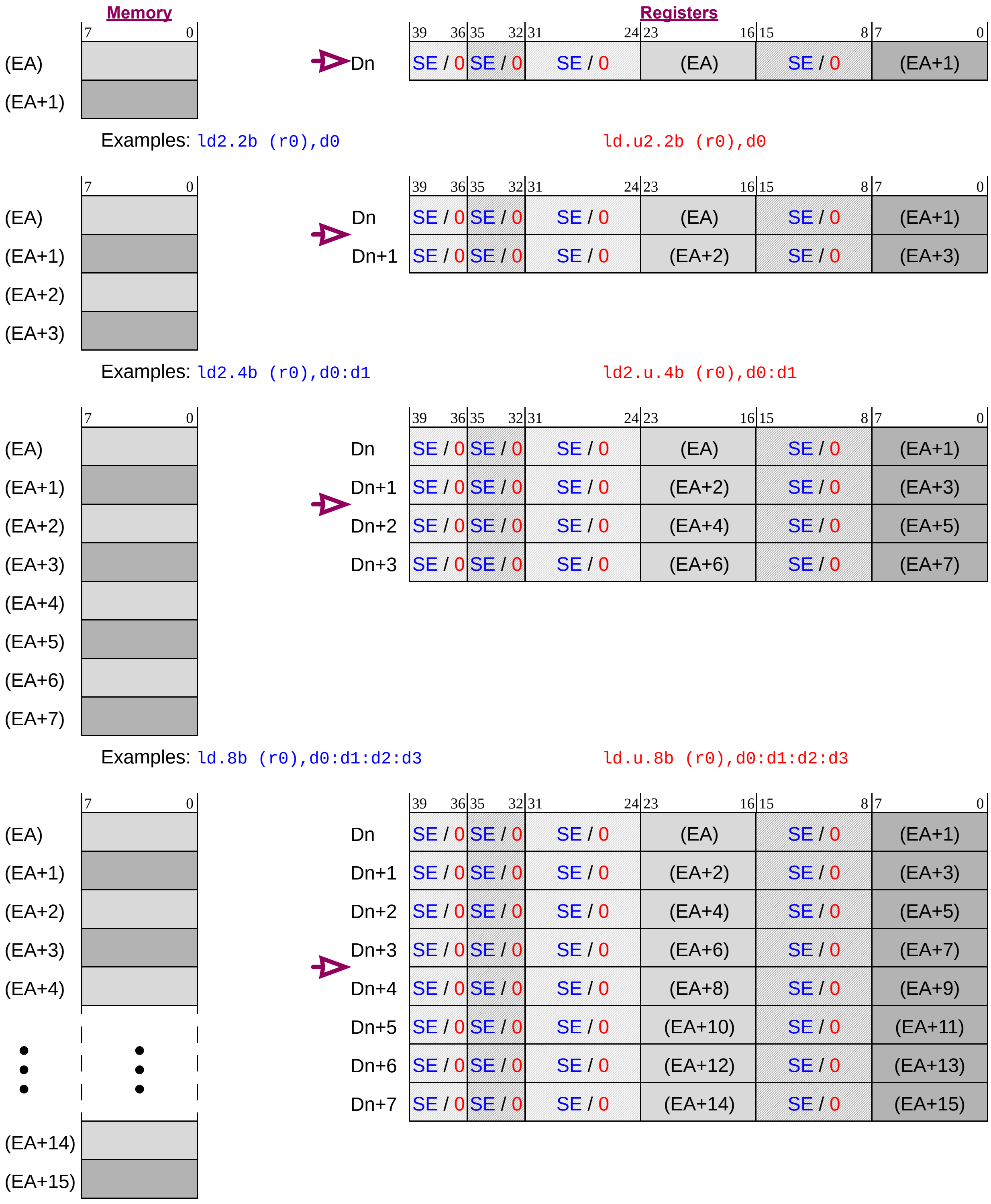
Examples: `ld.bf (r0),d0`

Examples: `ld.2bf (r0),d0:d1`

Examples: `ld.4bf (r0),d0:d1:d2:d3`

Examples: `ld.8bf (r0),d0:d1:d2:d3:d4:d5:d6:d7`

Figure 3-3. 8-bit signed fraction in memory into 40-bit signed accumulator (fraction)



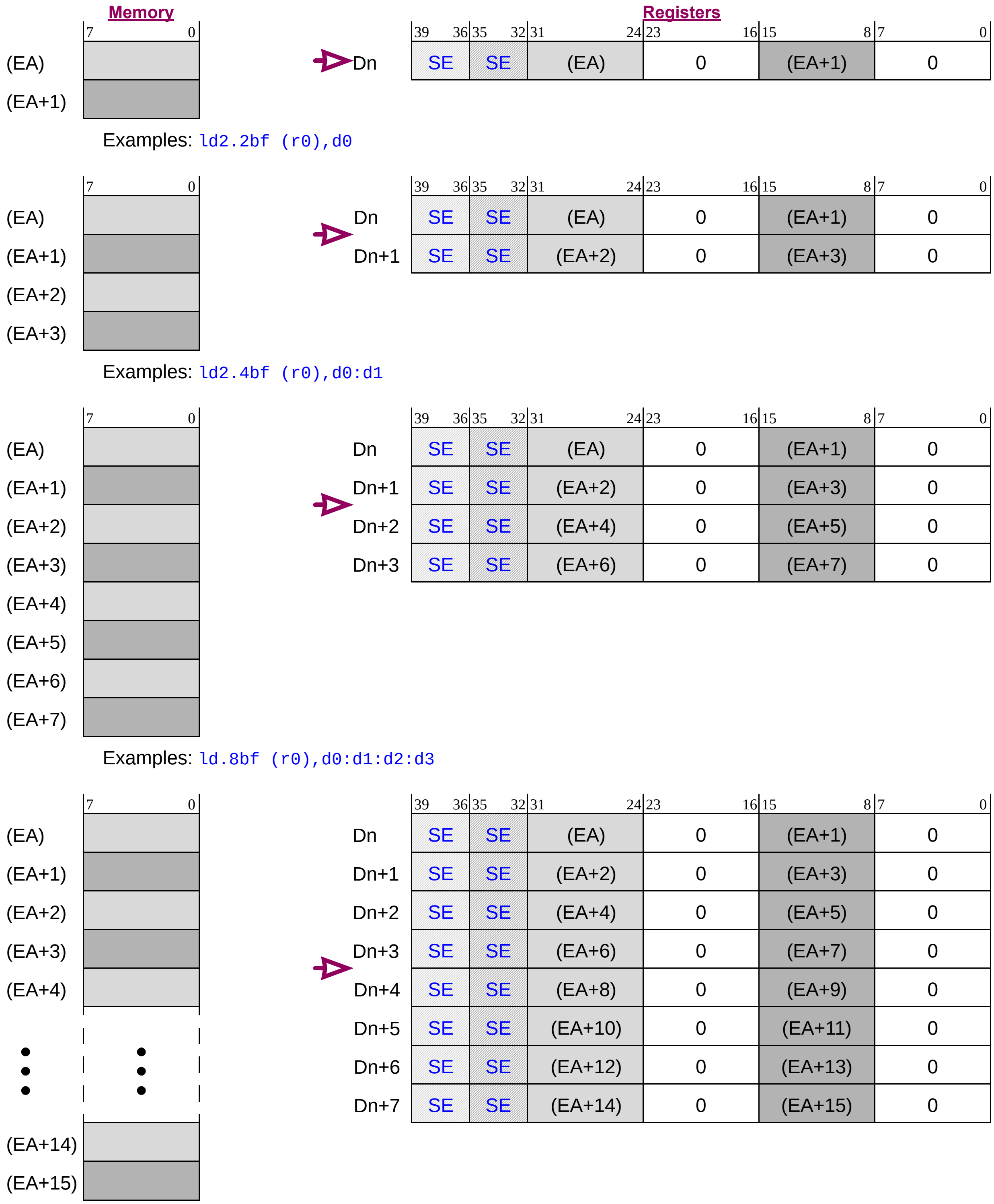
Examples: `ld2.2b (r0),d0` `ld.u2.2b (r0),d0`

Examples: `ld2.4b (r0),d0:d1` `ld2.u.4b (r0),d0:d1`

Examples: `ld.8b (r0),d0:d1:d2:d3` `ld.u.8b (r0),d0:d1:d2:d3`

Examples: `ld.16b (r0),d0:d1:d2:d3:d4:d5:d6:d7` `ld.u.16b (r0),d0:d1:d2:d3:d4:d5:d6:d7`

Figure 3-4. 8-bit signed/unsigned integer in memory into 20-bit signed/unsigned integer



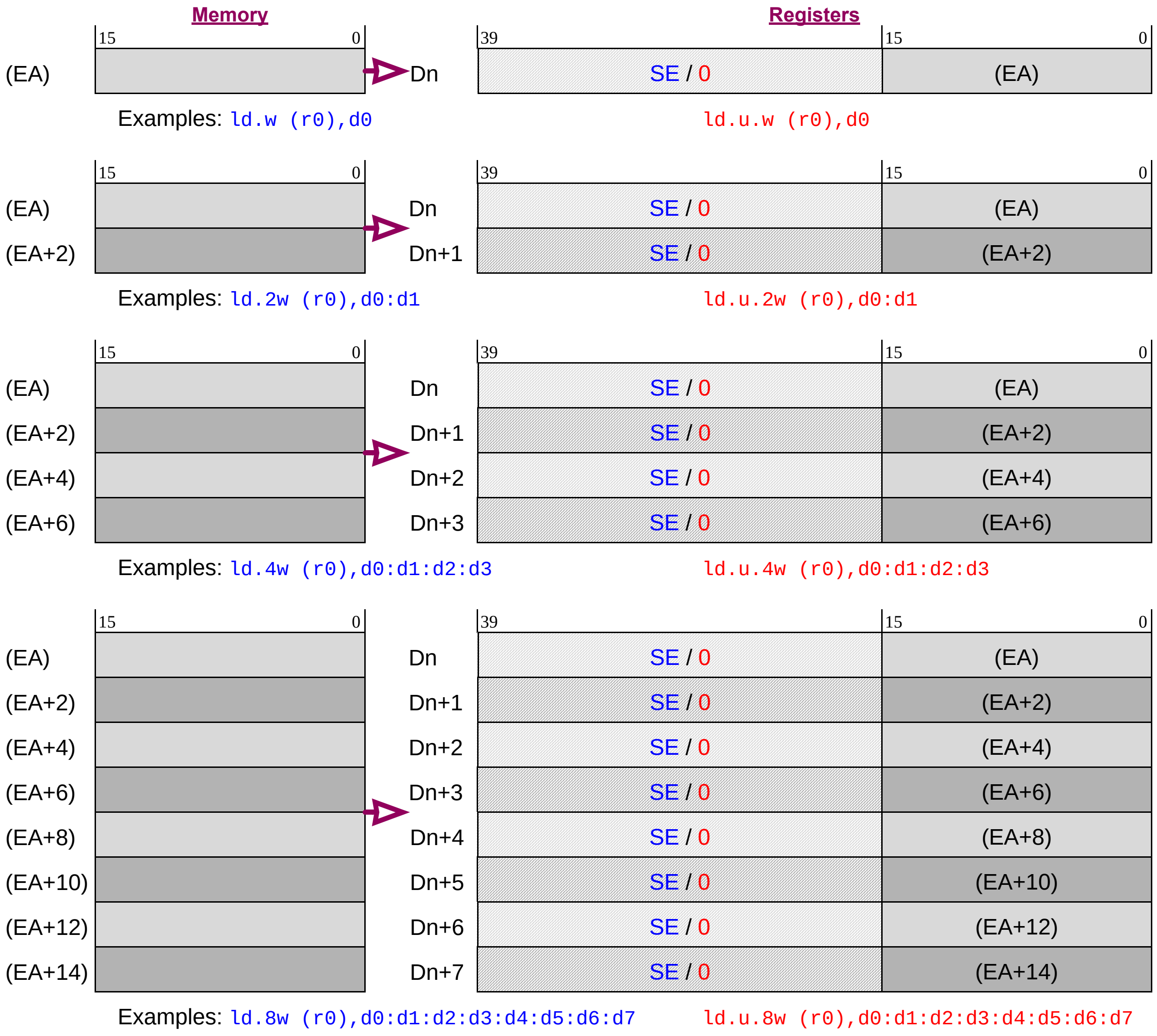
Examples: `ld2.2bf (r0),d0`

Examples: `ld2.4bf (r0),d0:d1`

Examples: `ld.8bf (r0),d0:d1:d2:d3`

Examples: `ld.16bf (r0),d0:d1:d2:d3:d4:d5:d6:d7`

Figure 3-5. 8-bit signed fraction in memory into 20-bit signed accumulator (fraction)



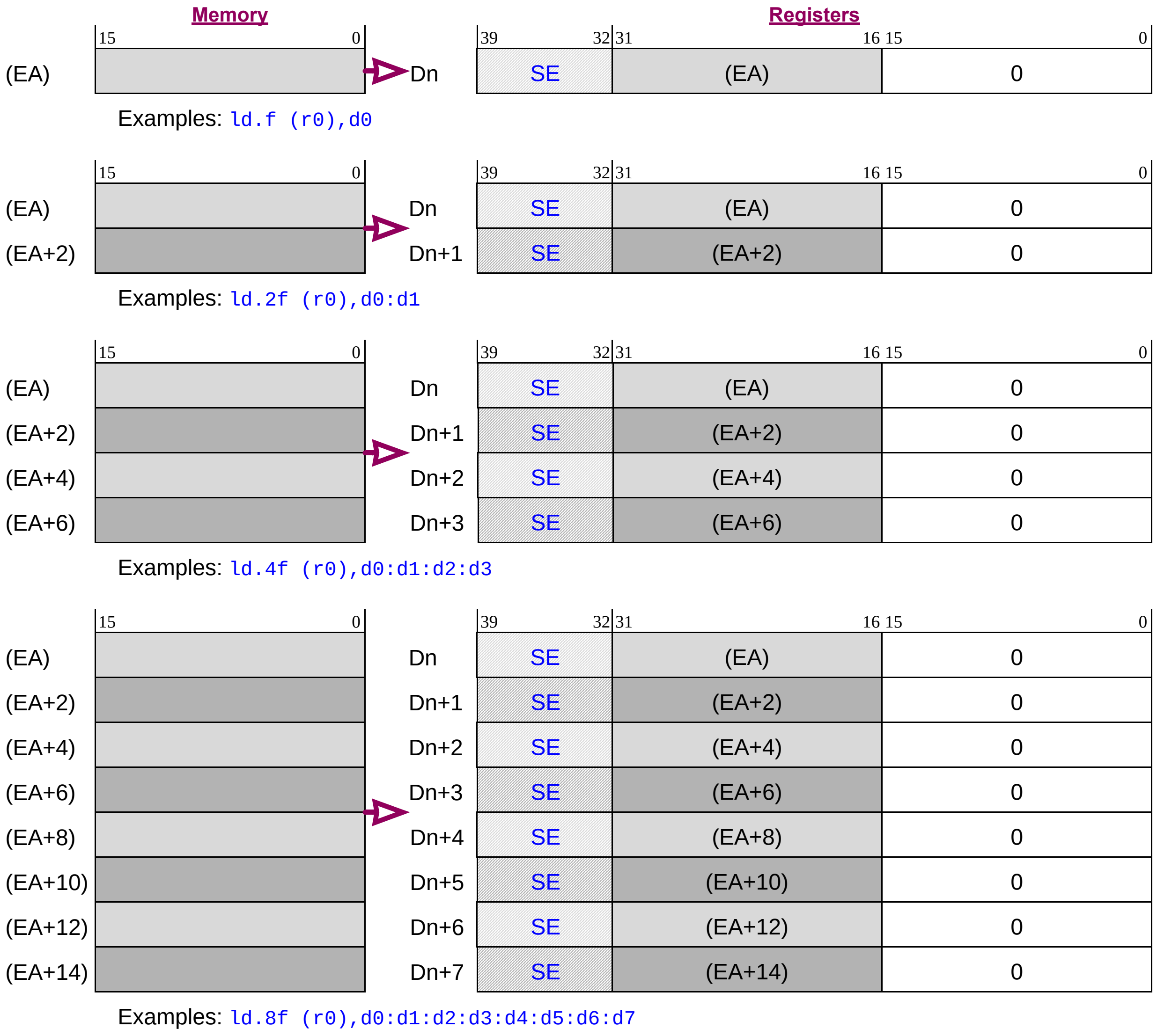
Examples: `ld.w (r0),d0` `ld.u.w (r0),d0`

Examples: `ld.2w (r0),d0:d1` `ld.u.2w (r0),d0:d1`

Examples: `ld.4w (r0),d0:d1:d2:d3` `ld.u.4w (r0),d0:d1:d2:d3`

Examples: `ld.8w (r0),d0:d1:d2:d3:d4:d5:d6:d7` `ld.u.8w (r0),d0:d1:d2:d3:d4:d5:d6:d7`

Figure 3-6. 16-bit signed/unsigned integer in memory into 40-bit signed/unsigned integer



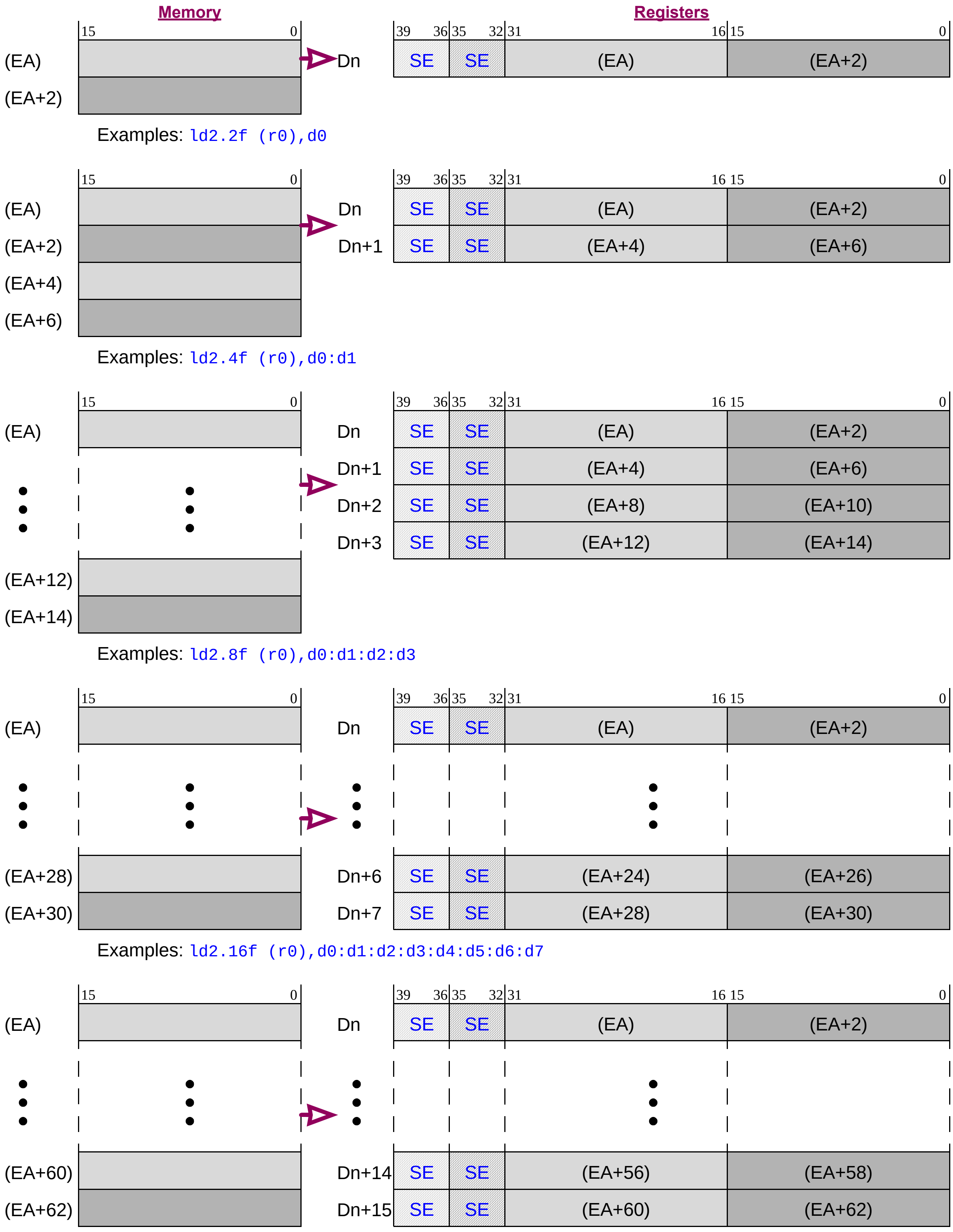
Examples: `ld.f (r0),d0`

Examples: `ld.2f (r0),d0:d1`

Examples: `ld.4f (r0),d0:d1:d2:d3`

Examples: `ld.8f (r0),d0:d1:d2:d3:d4:d5:d6:d7`

Figure 3-7. 16-bit signed fraction in memory into 40-bit signed accumulator (fraction)



Examples: `ld2.2f (r0),d0`

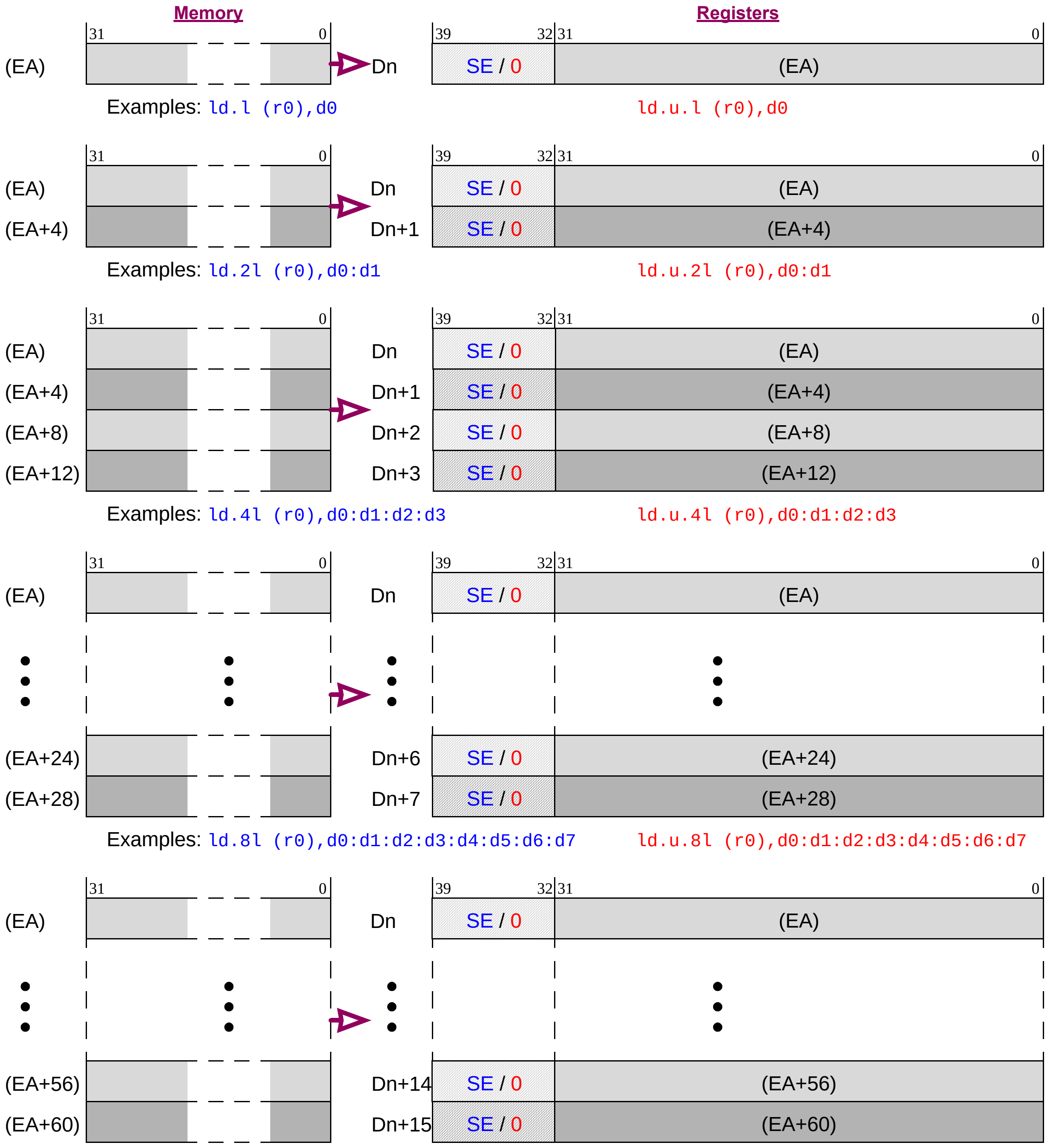
Examples: `ld2.4f (r0),d0:d1`

Examples: `ld2.8f (r0),d0:d1:d2:d3`

Examples: `ld2.16f (r0),d0:d1:d2:d3:d4:d5:d6:d7`

Examples: `ld2.32f (r0),d0:d1:d2:d3:d4:d5:d6:d7:d8:d9:d10:d11:d12:d13:d14:d15`

Figure 3-8. 16-bit signed integer/fraction in memory into 20-bit signed integer/fraction



Examples: `ld.l (r0),d0` `ld.u.l (r0),d0`

Examples: `ld.2l (r0),d0:d1` `ld.u.2l (r0),d0:d1`

Examples: `ld.4l (r0),d0:d1:d2:d3` `ld.u.4l (r0),d0:d1:d2:d3`

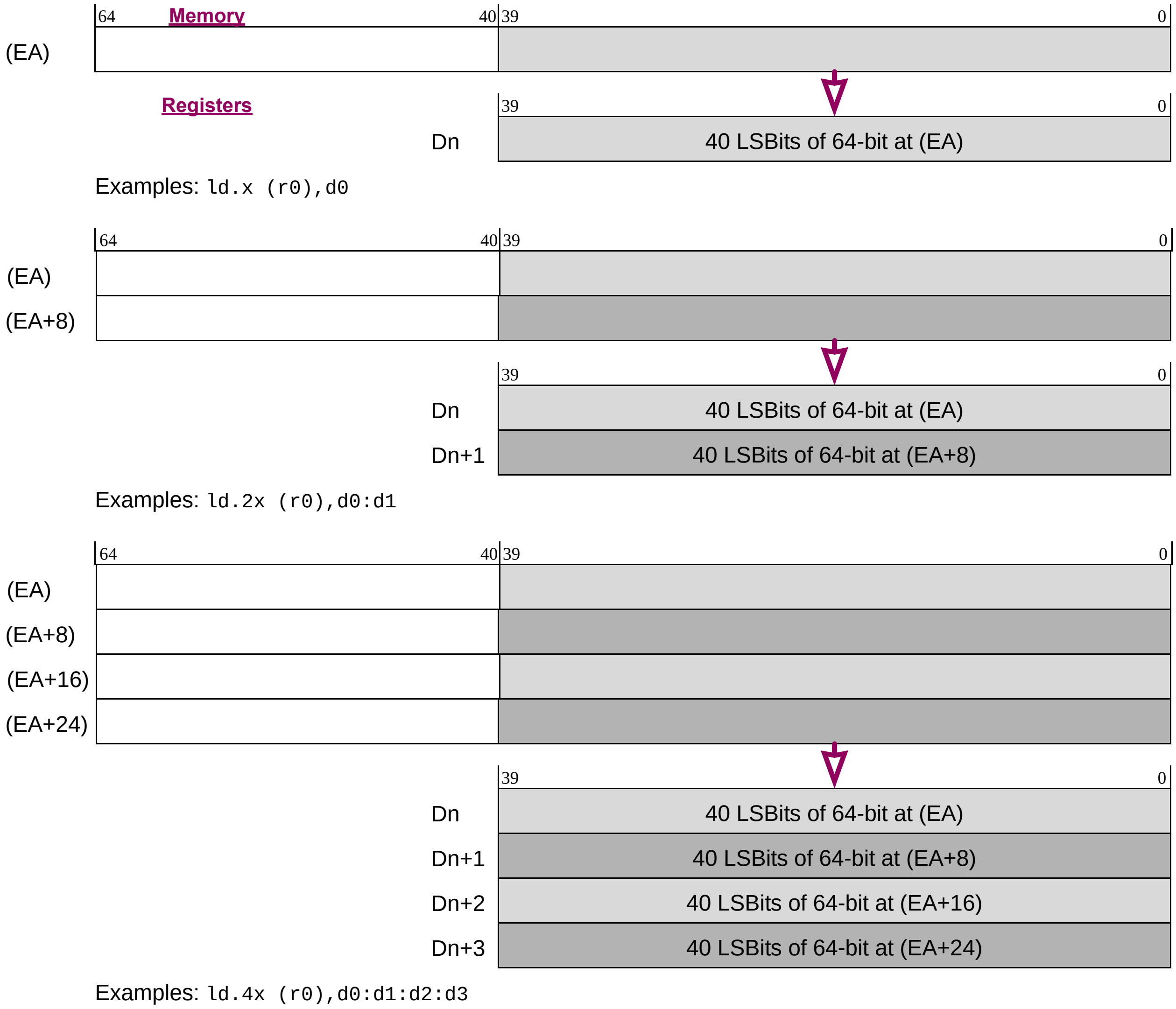
Examples: `ld.8l (r0),d0:d1:d2:d3:d4:d5:d6:d7` `ld.u.8l (r0),d0:d1:d2:d3:d4:d5:d6:d7`

Examples: `ld.16l (r0),d0:d1:d2:d3:d4:d5:d6:d7:d8:d9:d10:d11:d12:d13:d14:d15`

`ld.u.16l (r0),d0:d1:d2:d3:d4:d5:d6:d7:d8:d9:d10:d11:d12:d13:d14:d15`

Figure 3-9. 32-bit signed/unsigned integer/fraction in memory into 40-bit signed/unsigned integer/accumulator (fraction)

Note that a 32-bit unsigned integer to 40-bit unsigned integer can also be used as a 16-bit unsigned integer to 20-bit unsigned integer.


64
Memory
40
39
0
(EA)
Registers
39
0
Dn
40 LSBits of 64-bit at (EA)
64
40
39
0
(EA)
(EA+8)
39
0
Dn
40 LSBits of 64-bit at (EA)
Dn+1
40 LSBits of 64-bit at (EA+8)
64
40
39
0
(EA)
(EA+8)
(EA+16)
(EA+24)
39
0
Dn
40 LSBits of 64-bit at (EA)
Dn+1
40 LSBits of 64-bit at (EA+8)
Dn+2
40 LSBits of 64-bit at (EA+16)
Dn+3
40 LSBits of 64-bit at (EA+24)

Examples: `ld.x (r0),d0`

Examples: `ld.2x (r0),d0:d1`

Examples: `ld.4x (r0),d0:d1:d2:d3`

Figure 3-10. 64-bit signed/unsigned integer/fraction in memory into 40-bit signed/unsigned integer/accumulator (fraction)

3.2.2.2 Store of fixed point numbers

Fix point numbers are stored from a Dn register of 20-bit or 40-bit width into memory of 8-bit, 16-bit or 32-bit width. During the store operation, the numbers are wrapped around and truncated, meaning that only the bits that are required in the memory are used; bits above the value are discarded (wrapped around) and bits below the value are discarded (truncated).

There is no difference between signed and unsigned numbers, since no sign extension or zero extension should be performed.

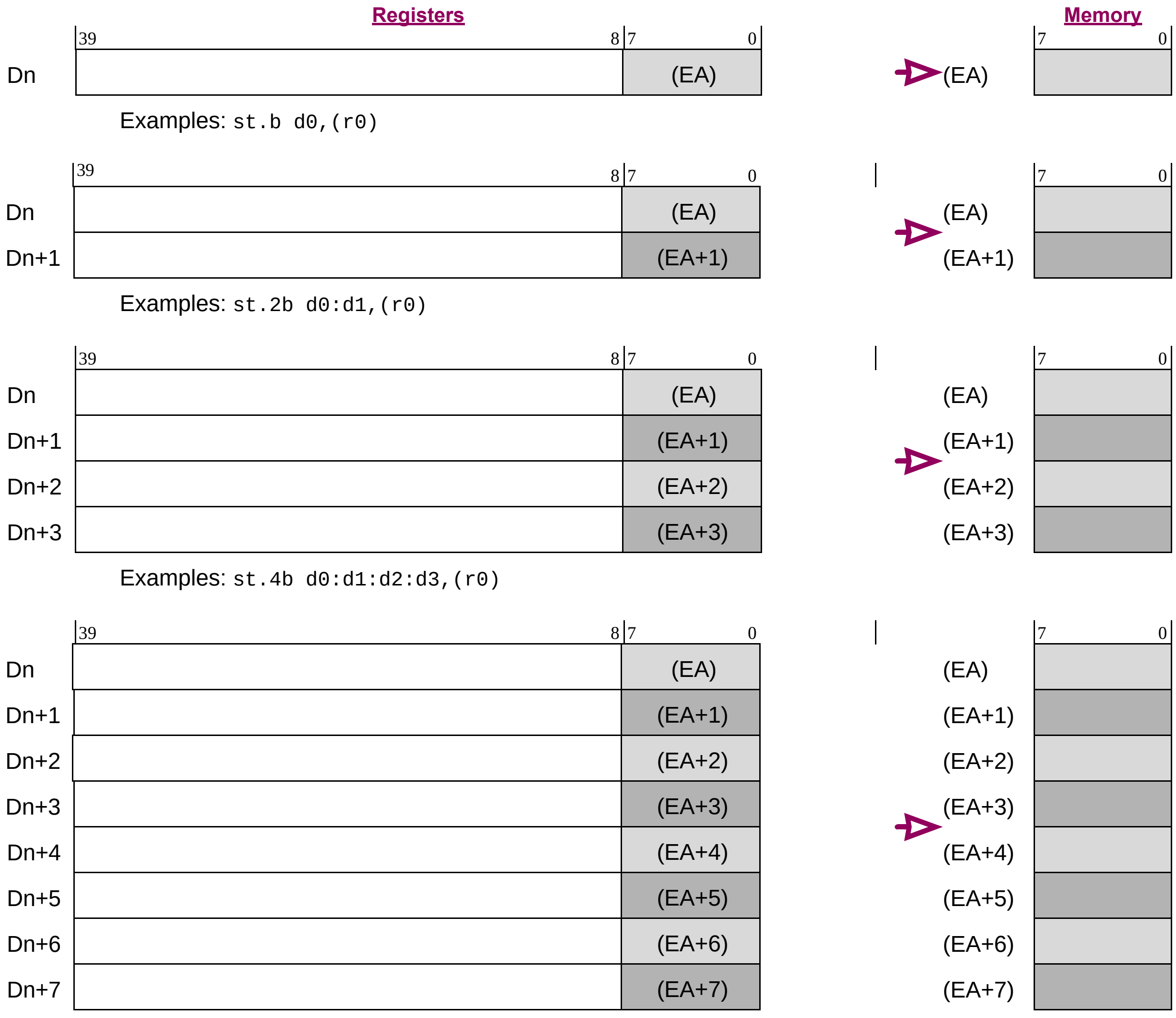
Fixed point 40-bit wide numbers can be stored from a Dn register into 64-bit wide memory. The register value is written to the 40 LSBits of the 64-bit value and zero extended (high 24 bits of the memory are set to zero).This behavior is marked in the following memory maps as ‘0.’

The fixed point numbers store instructions are supported by LSU instructions, with different addressing generation and register allocations.

NOTE

Special Scale, Round and Saturate (SRS) instructions are described in a separate paragraph. For details, see Chapter 3.2.2.3, “Scale, round and saturate store instructions.”

The following figures show how the SC3900FP supports fixed point numbers store from Dn registers to memory.



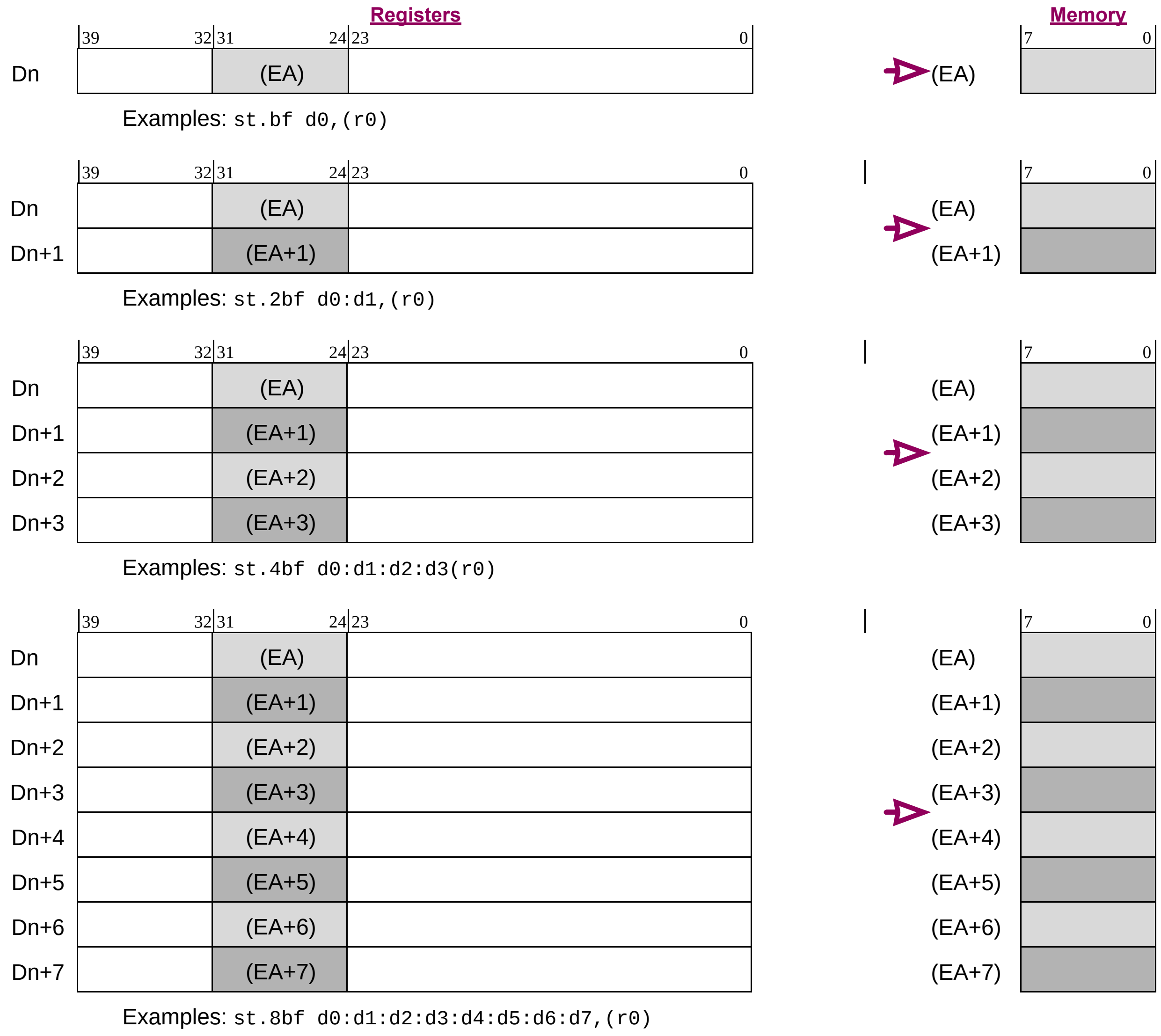
Examples: `st.b d0,(r0)`

Examples: `st.2b d0:d1,(r0)`

Examples: `st.4b d0:d1:d2:d3,(r0)`

Examples: `st.8b d0:d1:d2:d3:d4:d5:d6:d7,(r0)`

Figure 3-11. 40-bit signed/unsigned integer into 8-bit signed/unsigned integer in memory



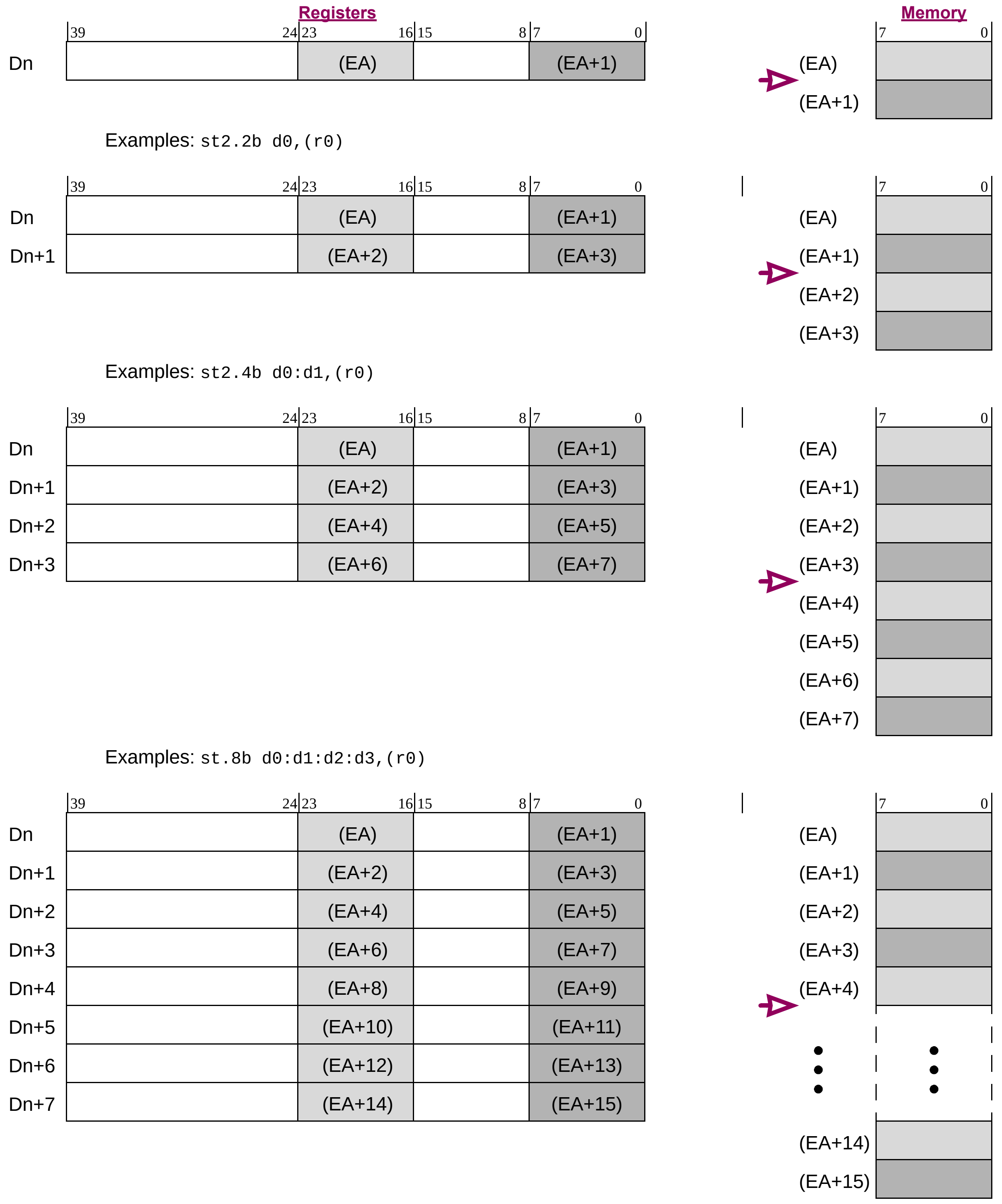
Examples: `st.bf d0,(r0)`

Examples: `st.2bf d0:d1,(r0)`

Examples: `st.4bf d0:d1:d2:d3(r0)`

Examples: `st.8bf d0:d1:d2:d3:d4:d5:d6:d7,(r0)`

Figure 3-12. 40-bit signed accumulator (fraction) into 8-bit signed fraction in memory



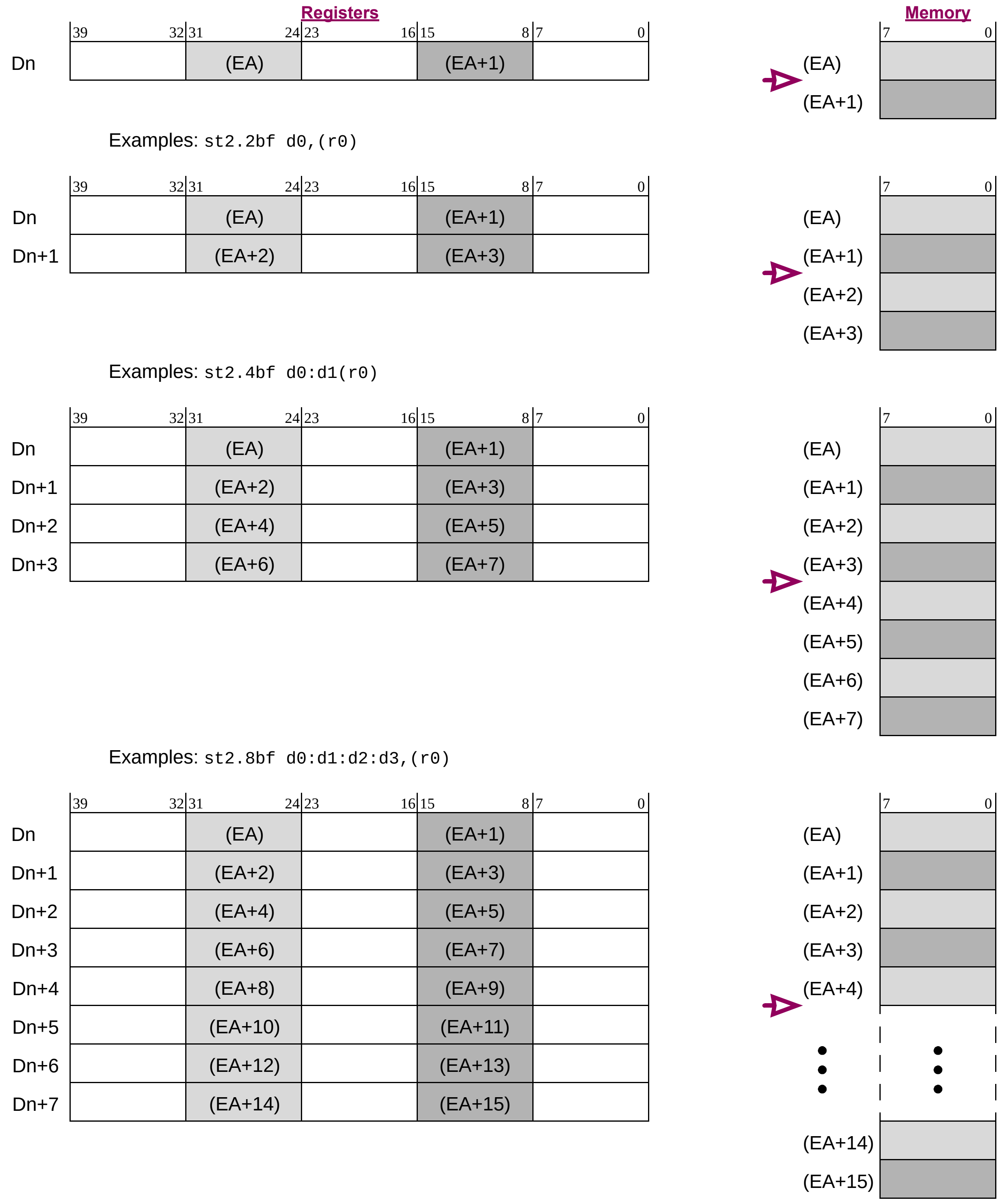
Examples: `st2.2b d0,(r0)`

Examples: `st2.4b d0:d1,(r0)`

Examples: `st.8b d0:d1:d2:d3,(r0)`

Examples: `st.16b d0:d1:d2:d3:d4:d5:d6:d7(r0)`

Figure 3-13. 20-bit signed/unsigned integer into 8-bit signed/unsigned integer in memory



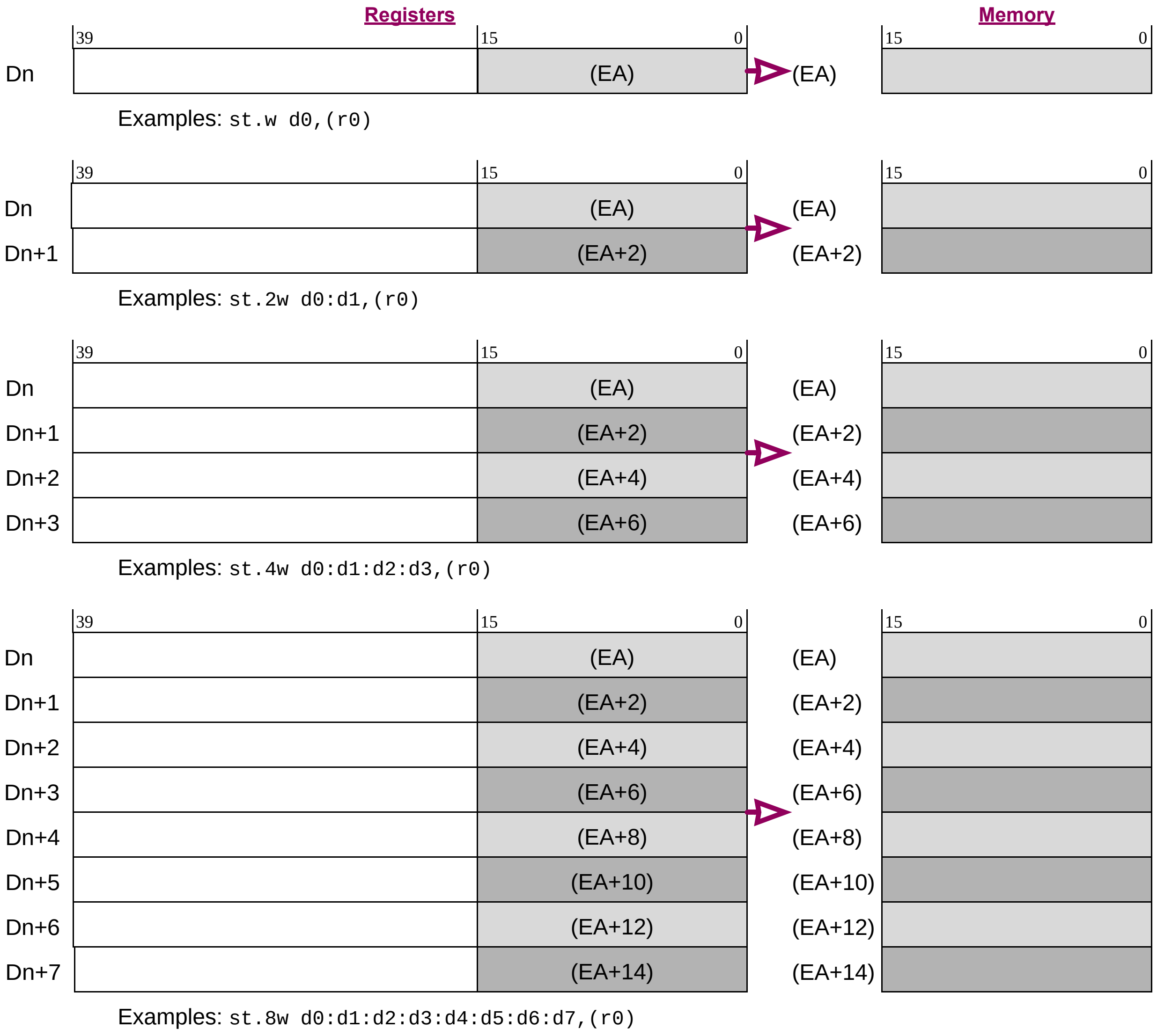
Examples: `st2.2bf d0,(r0)`

Examples: `st2.4bf d0:d1(r0)`

Examples: `st2.8bf d0:d1:d2:d3,(r0)`

Examples: `st2.16bf d0:d1:d2:d3:d4:d5:d6:d7,(r0)`

Figure 3-14. 20-bit signed accumulator (fraction) into 8-bit signed fraction in memory



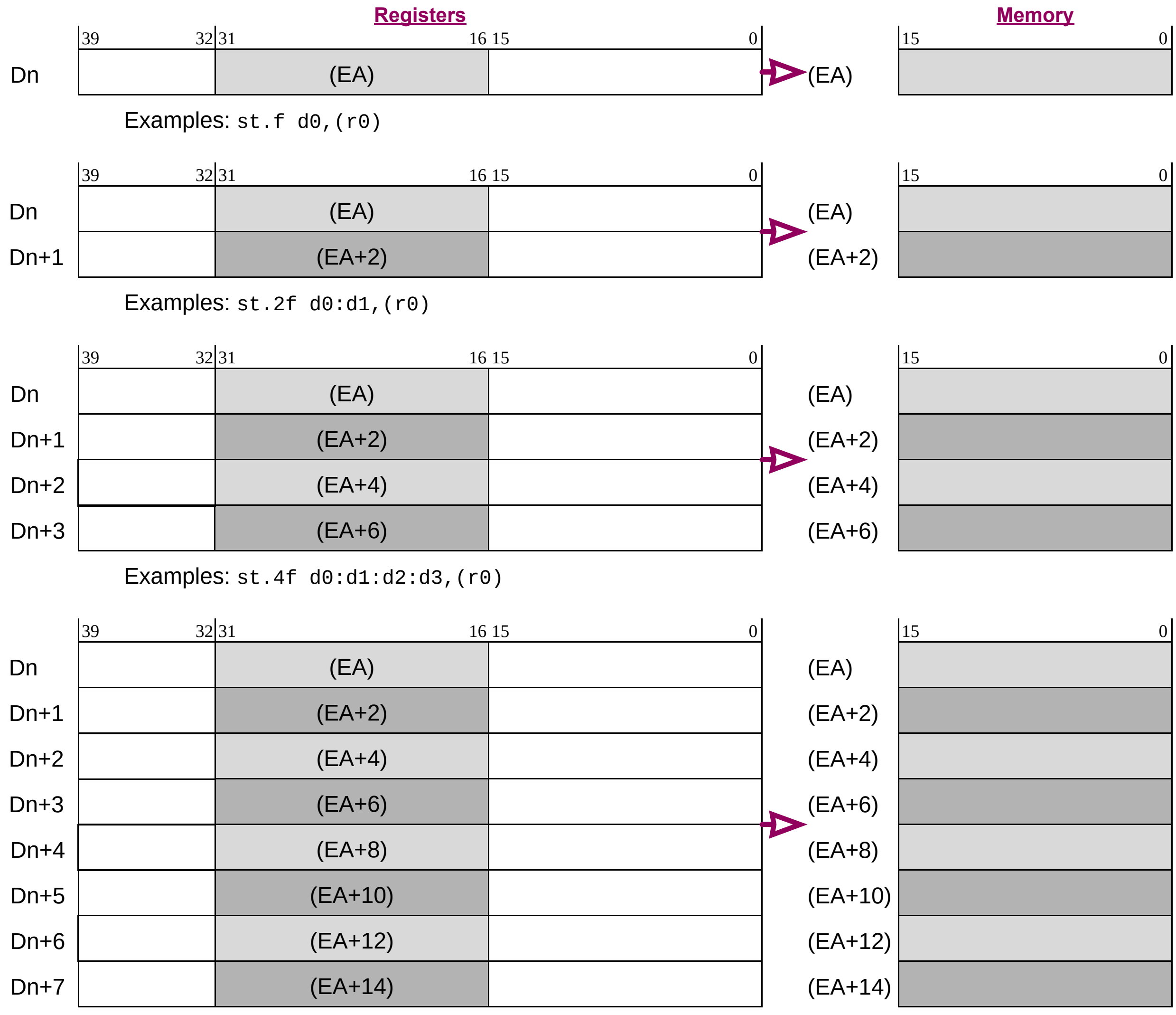
Examples: `st.w d0,(r0)`

Examples: `st.2w d0:d1,(r0)`

Examples: `st.4w d0:d1:d2:d3,(r0)`

Examples: `st.8w d0:d1:d2:d3:d4:d5:d6:d7,(r0)`

Figure 3-15. 40-bit signed/unsigned integer into 16-bit signed/unsigned integer in memory



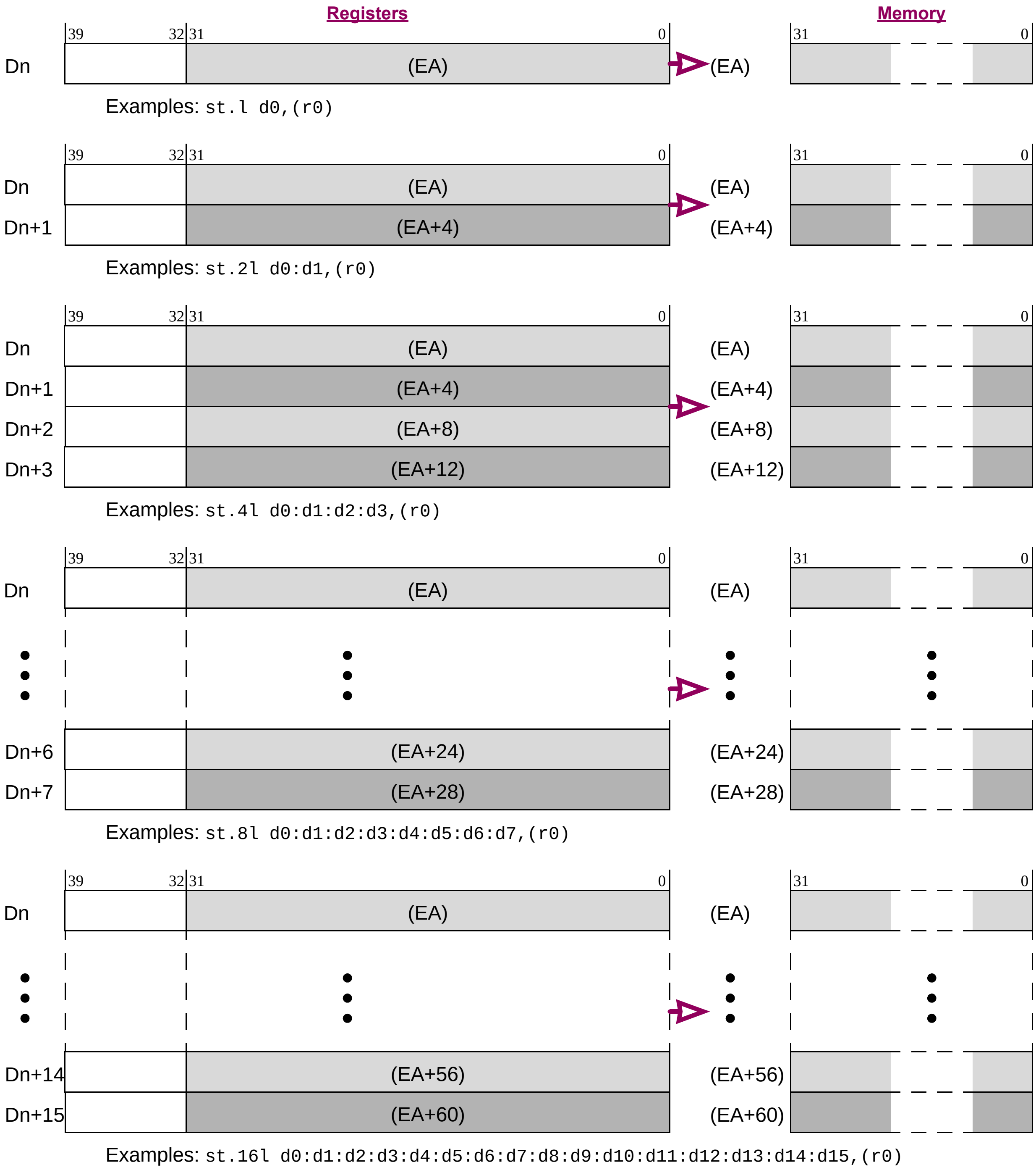
Examples: `st.f d0,(r0)`

Examples: `st.2f d0:d1,(r0)`

Examples: `st.4f d0:d1:d2:d3,(r0)`

Examples: `st.8f d0:d1:d2:d3:d4:d5:d6:d7,(r0)`

Figure 3-16. 40-bit signed accumulator (fraction) into 16-bit signed fraction in memory



Examples: `st.l d0,(r0)`

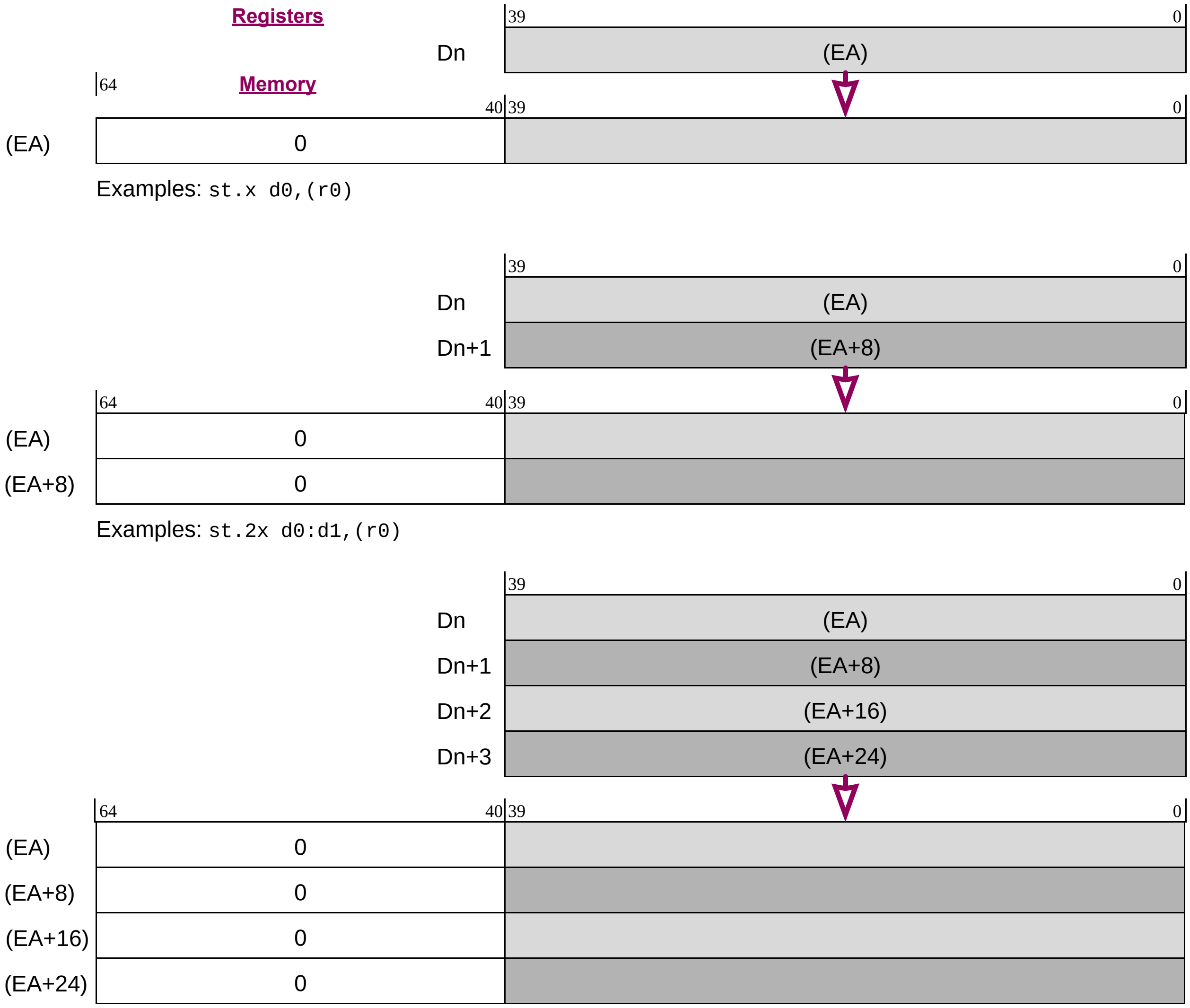
Examples: `st.2l d0:d1,(r0)`

Examples: `st.4l d0:d1:d2:d3,(r0)`

Examples: `st.8l d0:d1:d2:d3:d4:d5:d6:d7,(r0)`

Examples: `st.16l d0:d1:d2:d3:d4:d5:d6:d7:d8:d9:d10:d11:d12:d13:d14:d15,(r0)`

Figure 3-17. 40-bit signed/unsigned integer/accumulator (fraction) into 32-bit signed/unsigned integer/fraction in memory



Examples: `st.x d0,(r0)`

Examples: `st.2x d0:d1,(r0)`

Examples: `st.4x d0:d1:d2:d3,(r0)`

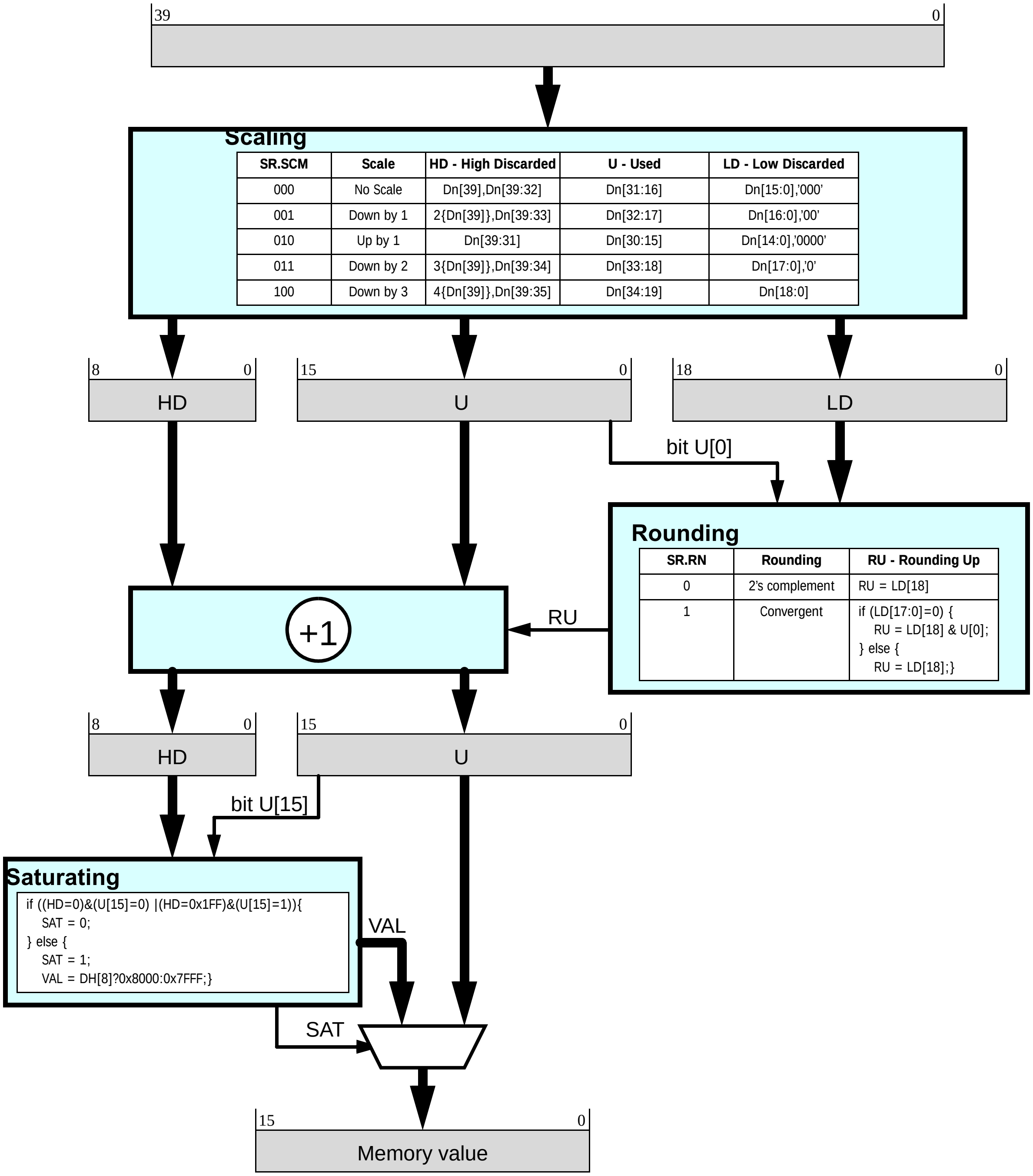
Figure 3-18. 40-bit fixed into 64-bit fixed in memory

3.2.2.3 Scale, round and saturate store instructions

The SC3900FP core can perform additional processing on numbers on their way to be stored in memory. Usually during DSP calculation, numbers are kept in registers with better precision and dynamic range than in memory. For example, accumulate 16-bit x 16-bit fraction multiplication results in a 40-bit accumulator type, and stores the result to memory as 16-bit fraction. Lowering value precision and dynamic range is done in order to reduce the required memory bandwidth, which improves performance. In order to avoid overflow, in many cases scaling is required before a value is stored to memory. The Scale, Round and Saturate (SRS) store instructions take an input number of 20-bit or 40-bit from the Dn register, scale it according to the required mode, round it if the memory format precision is lowered (that is, LSBits are thrown away), saturate the result (if the value cannot be represented in the required memory format), and store the result in memory. In maximal capacity, 32 values of 20 bits each could be stored to memory simultaneously, every cycle after scaling, rounding, and saturation.

The store SRS instruction works on signed fraction numbers. The following five types of store SRS are supported: 20-bit/40-bit into 8-bit, 20-bit/40-bit into 16-bit and 40-bit into 32-bit.

The following diagram shows the SRS process in the case of a 40-bit value into 16-bit value. A similar process is performed for the other SRS flavors.



SR.SCM	Scale	HD - High Discarded	U - Used	LD - Low Discarded
000	No Scale	Dn[39],Dn[39:32]	Dn[31:16]	Dn[15:0],'000'
001	Down by 1	2{Dn[39]},Dn[39:33]	Dn[32:17]	Dn[16:0],'00'
010	Up by 1	Dn[39:31]	Dn[30:15]	Dn[14:0],'0000'
011	Down by 2	3{Dn[39]},Dn[39:34]	Dn[33:18]	Dn[17:0],'0'
100	Down by 3	4{Dn[39]},Dn[39:35]	Dn[34:19]	Dn[18:0]

SR.RN	Rounding	RU - Rounding Up
0	2's complement	RU = LD[18]

1	Convergent	if (LD[17:0]=0) {
		RU = LD[18] & U[0];
		} else {
		RU = LD[18];}

```
if ((HD=0)&(U[15]=0) |(HD=0x1FF)&(U[15]=1)){
   SAT = 0;
} else {
   SAT = 1;
   VAL = DH[8]?0x8000:0x7FFF;}
```

Figure 3-19. Scaling, rounding and saturating process for 40-bit into 16-bit

The Scaling, Rounding and Saturating (SRS) is done in three steps, each of which is described in the following sections.

3.2.2.3.1 Understanding scaling

Scaling is optionally shifting the output value down by one, two or three bits or up by one bit. Scaling is set according to SCM field value in the SR register, see Section 2.1.5.1, “Status Register (SR).”

Note that there are few legacy DALU arithmetic instructions for which rounding is also affected by the scaling mode. For details, see Section 3.4.2, “Configuring the operation of store with scale, round, and saturate instructions.”

The following table shows the bit of the input 20-bit or the 40-bit value, which is taken for the result (before the rounding and saturation calculation).

Table 3-1. Signed integer values

Register value width	Output memory width	Scaling factor				
		Up by 1	No Scale	Down by 1	Down by 2	Down by 3
40-bit	8-bit	30:23	31:24	32:25	33:26	34:27
20-bit	8-bit	14:7	15:8	16:9	17:10	18:11
40-bit	16-bit	30:15	31:16	32:17	33:18	34:19
20-bit	16-bit	14:0 [1]	15:0	16:1	17:2	18:3
40-bit	32-bit	30:0 [1]	31:0	32:1 [2]	33:2 [2]	34:3 [2]

[1] LSBit of the result is taken as 0.

[2] Discarded bits are truncated (no rounding).

3.2.2.3.2 Understanding the rounding modes

The store SRS instruction supports two rounding modes: 2’s complement rounding and convergent rounding. The rounding mode is set according to value of RM bit in the SR, see Section 2.1.5.1, “Status Register (SR).” In 2’s complement rounding, the result is rounded up if the discarded remainder is greater or equal half of the result LSBit. In convergent rounding, the result is rounded up if the discarded remainder is greater than half of the result LSBit. If the discarded remainder is exactly half, that the number is rounded to the nearest even, that is, it is rounded up only if the LSBit of the result is set.

NOTE

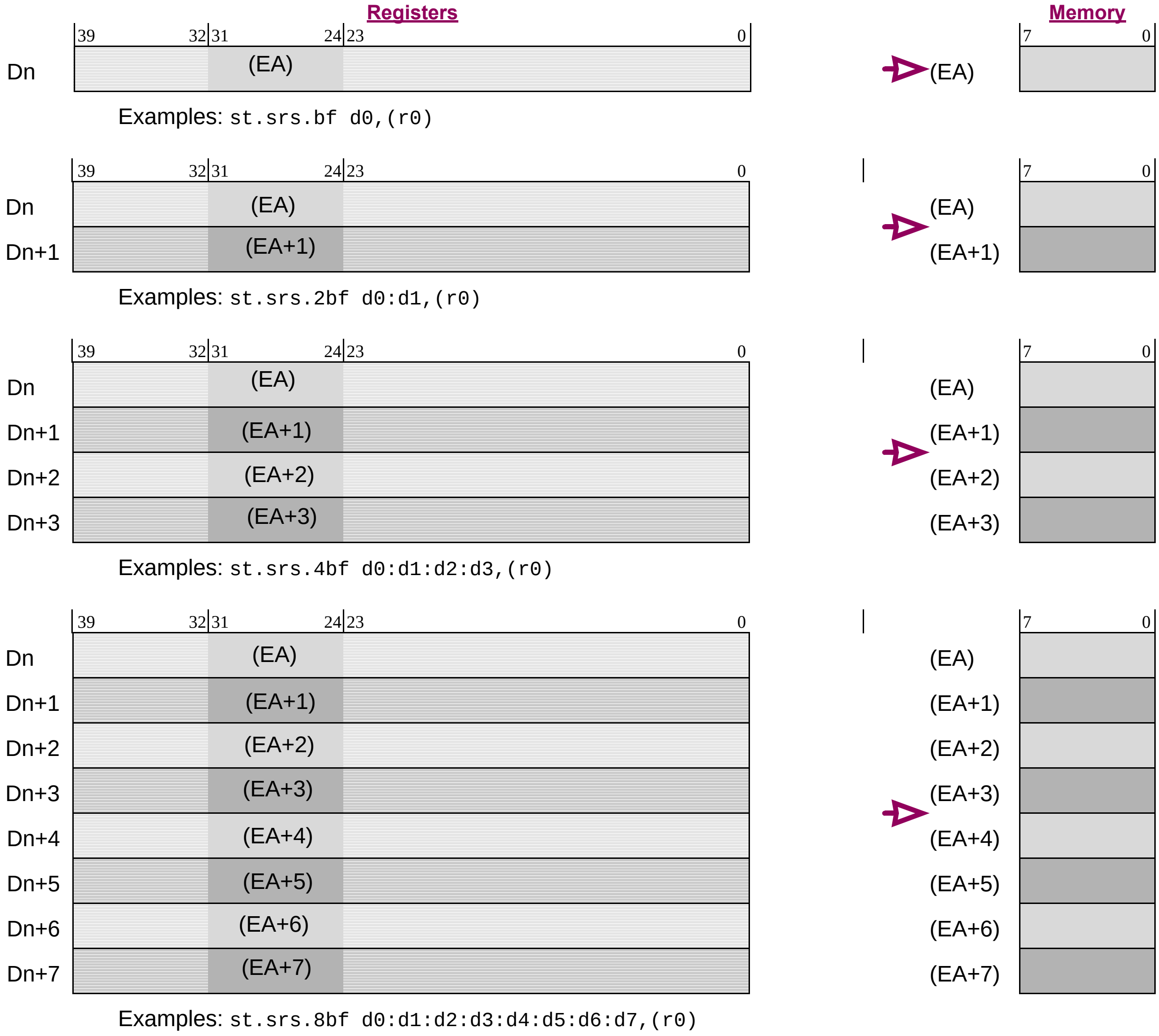
When 40-bit value is stored into 32-bit with store SRS instruction, the discarded reminder is truncated, and no rounding occurs.

3.2.2.3.3 Understanding saturation

Saturation is performed after the number is scaled and rounded. The number is saturated if it cannot be represented in the output format. The saturation is to a positive maximum value or negative minimum value according to the sign of the original 20-bit or 40-bit value.

3.2.2.4 How the SC3900FP supports fixed point numbers store from Dn registers to memory

The following diagrams show how the SC3900FP supports fixed point numbers store from Dn registers to memory.



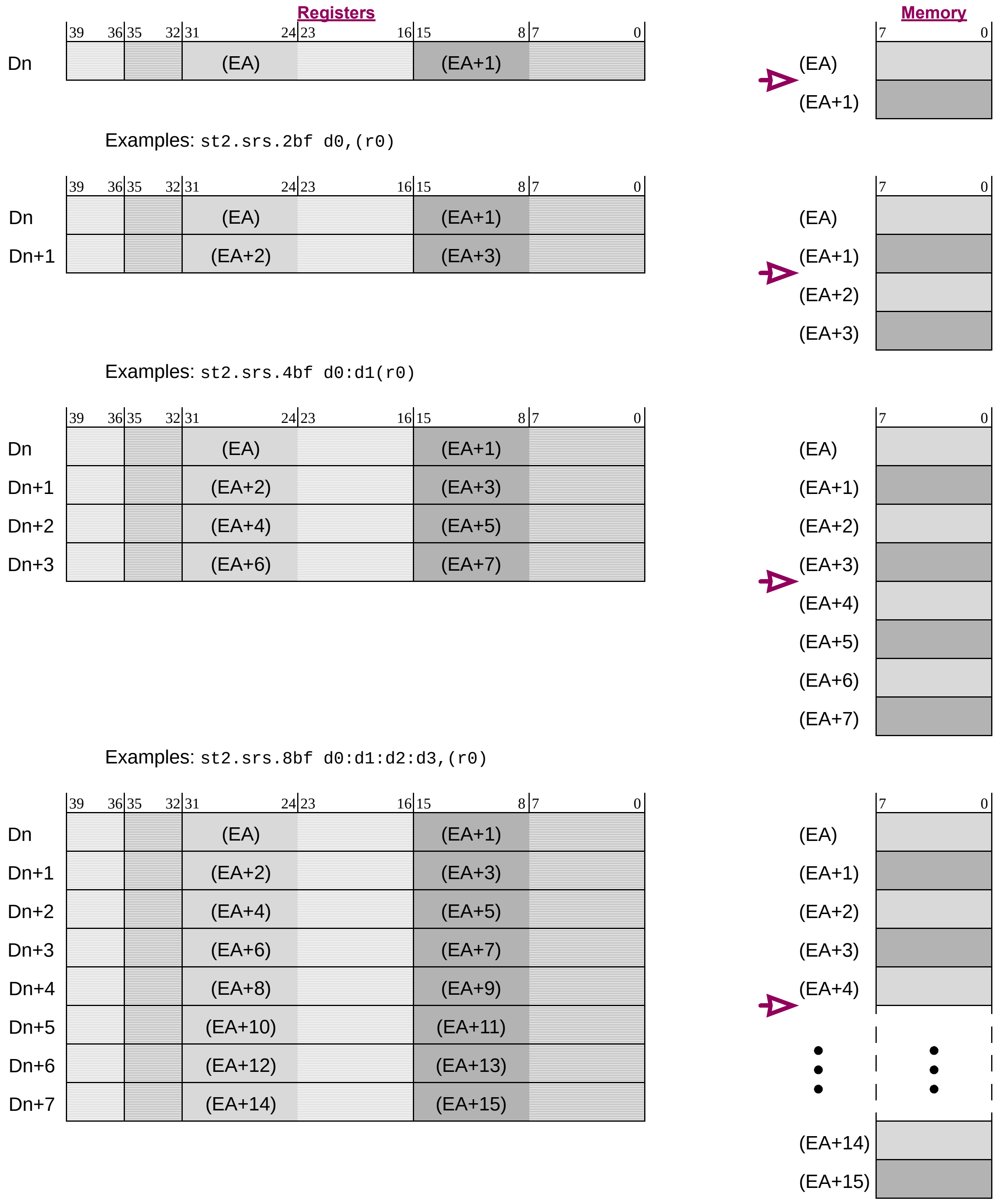
Examples: `st.srs.bf d0,(r0)`

Examples: `st.srs.2bf d0:d1,(r0)`

Examples: `st.srs.4bf d0:d1:d2:d3,(r0)`

Examples: `st.srs.8bf d0:d1:d2:d3:d4:d5:d6:d7,(r0)`

Figure 3-20. SRS 40-bit signed accumulator (fraction) into 8-bit signed fraction in memory



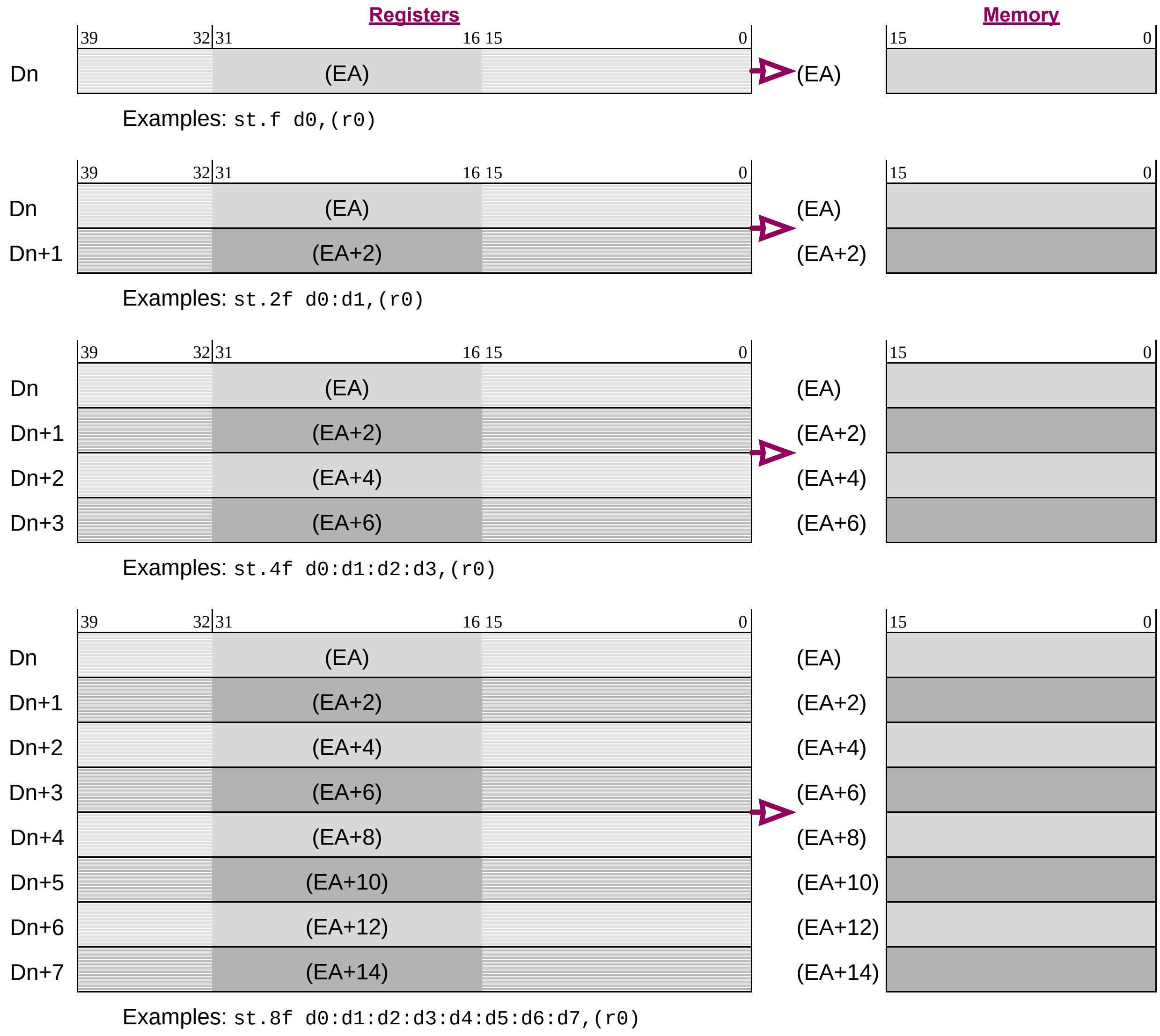
Examples: `st2.srs.2bf d0,(r0)`

Examples: `st2.srs.4bf d0:d1(r0)`

Examples: `st2.srs.8bf d0:d1:d2:d3,(r0)`

Examples: `st2.srs.16bf d0:d1:d2:d3:d4:d5:d6:d7,(r0)`

Figure 3-21. SRS 20-bit signed accumulator (fraction) into 8-bit signed fraction in memory



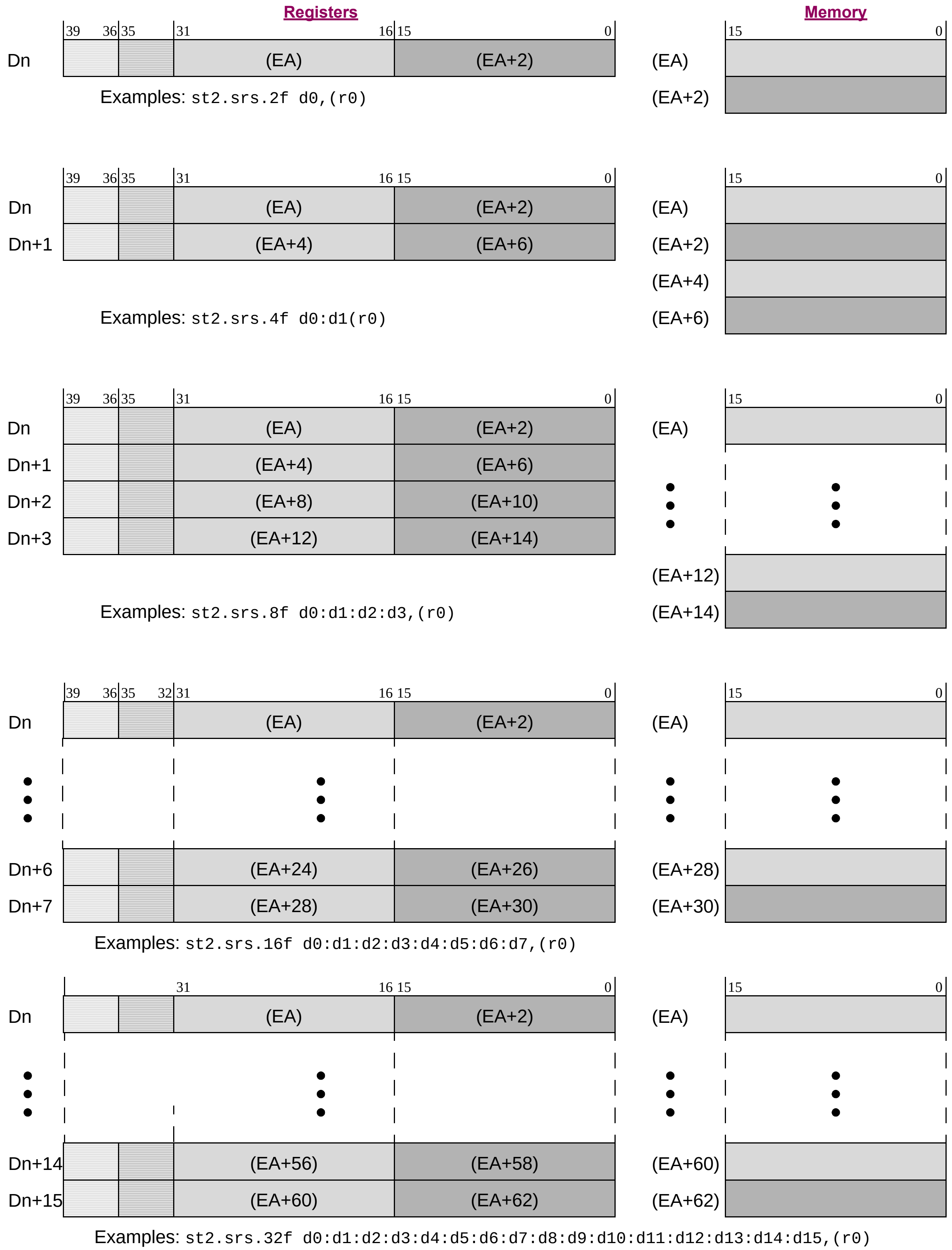
Examples: `st.f d0,(r0)`

Examples: `st.2f d0:d1,(r0)`

Examples: `st.4f d0:d1:d2:d3,(r0)`

Examples: `st.8f d0:d1:d2:d3:d4:d5:d6:d7,(r0)`

Figure 3-22. SRS 40-bit signed accumulator (fraction) into 16-bit signed fraction in memory



Examples: `st2.srs.2f d0,(r0)`

Examples: `st2.srs.4f d0:d1(r0)`

Examples: `st2.srs.8f d0:d1:d2:d3,(r0)`

Examples: `st2.srs.16f d0:d1:d2:d3:d4:d5:d6:d7,(r0)`

Examples: `st2.srs.32f d0:d1:d2:d3:d4:d5:d6:d7:d8:d9:d10:d11:d12:d13:d14:d15,(r0)`

Figure 3-23. SRS 20-bit signed accumulator (fraction) into 16-bit signed fraction in memory

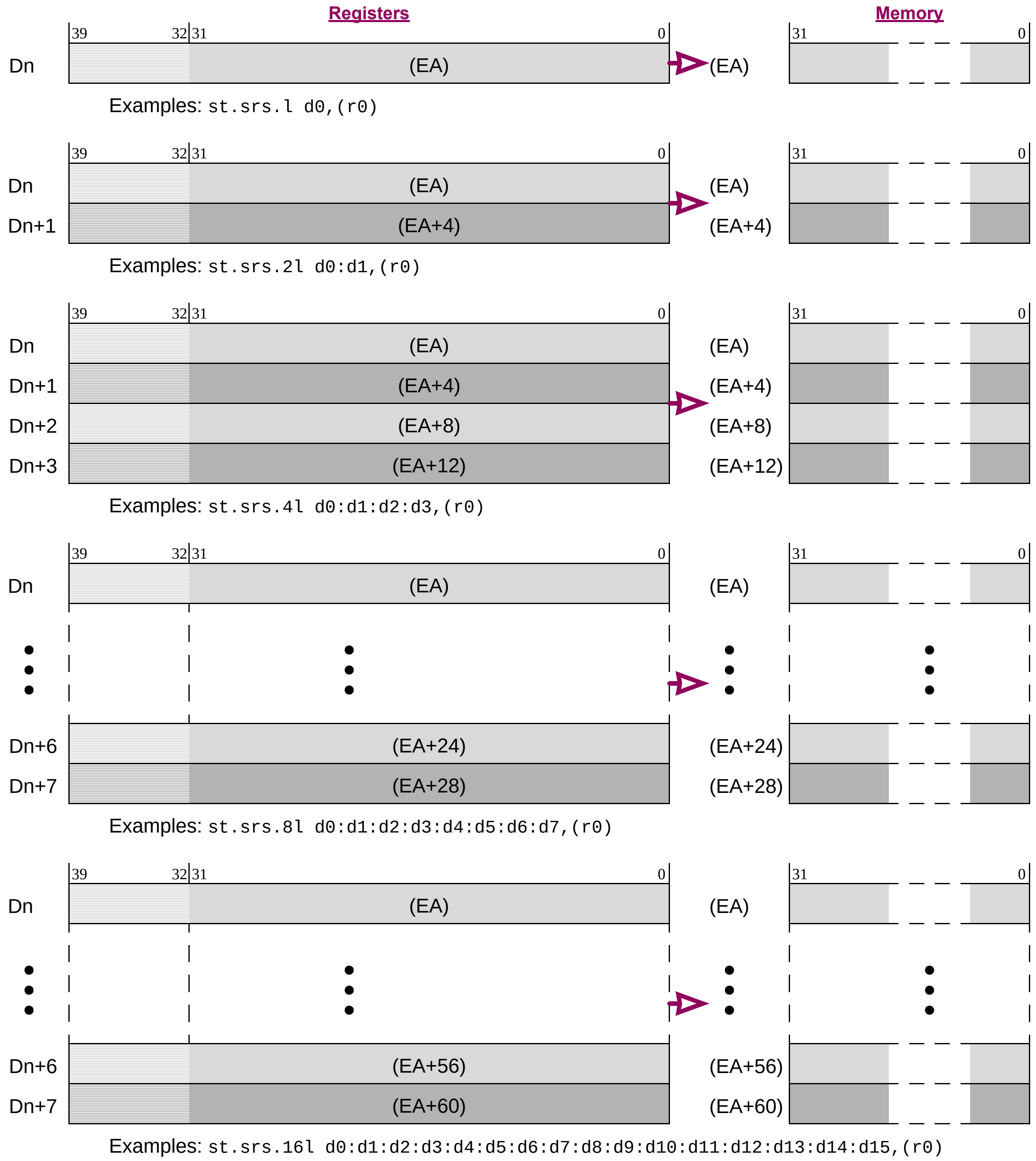

Registers
Memory
39
32
31
0
31
0
Dn
(EA)
(EA)
Examples: st.srs.l d0,(r0)
Dn
Dn+1
(EA)
(EA+4)
(EA)
(EA+4)
Examples: st.srs.2l d0:d1,(r0)
Dn
Dn+1
Dn+2
Dn+3
(EA)
(EA+4)
(EA+8)
(EA+12)
(EA)
(EA+4)
(EA+8)
(EA+12)
Examples: st.srs.4l d0:d1:d2:d3,(r0)
Dn
Dn+6
Dn+7
(EA)
(EA+24)
(EA+28)
(EA)
(EA+24)
(EA+28)
Examples: st.srs.8l d0:d1:d2:d3:d4:d5:d6:d7,(r0)
Dn
Dn+6
Dn+7
(EA)
(EA+56)
(EA+60)
(EA)
(EA+56)
(EA+60)
Examples: st.srs.16l d0:d1:d2:d3:d4:d5:d6:d7:d8:d9:d10:d11:d12:d13:d14:d15,(r0)

Figure 3-24. SRS 40-bit signed/unsigned integer/accumulator (fraction) into 32-bit signed/unsigned integer/fraction in memory

3.2.2.5 Implicit load and store instructions

In the sections above, all the implemented formats of load, store and store SRS are described. Note that in many cases, load and store instructions that are not explicitly defined can be implemented by using an equivalent instruction that is defined in another format. For example, load of four packed bytes from memory into a single Dn register should be done with the LD.L instruction (there is no explicit LD4.4B instruction defined).

The following table shows examples of explicit instruction that should be used for implicit load and store instructions.

Table 3-2. Implicit load and store instructions

Implicit instruction	Implicit width		Explicit width		Explicit syntax example
	From	To	From	To	
Load	4*N bytes	Packed 4 bytes in N registers	N long words	N registers	ld.l (r0),d0 ld.2l (r0),d0:d1 ld.4l (r0),d0:d1:d2:d3:d4 ld.8l (r0),d0::d7 ld.16l (r0),d0::d15
	2*N words	Packed two words in N registers			
Store	Packed 4 bytes in N registers	4*N bytes	N registers	N long words	st.l d0,(r0) st.2l d0:d1,(r0) st.4l d0:d1:d2:d3:d4,(r0) st.8l d0::d7,(r0) st.16l d0::d15,(r0)
	Packed two words in N registers	2*N words			
	Packed two 20-bit words in N registers	2*N words			

3.2.3 Floating point format

The SC3900FP core support IEEE 754 single precision floating point operations. Floating point numbers are stored in Dn registers, residing in the low 32-bit of it. The upper 8-bit (extension) is not used as an input and is written as zero by floating point instructions. Native floating-point operations for single precision numbers are supported (Double-precision floating-point is supported via software emulation).

The representation of the single precision floating point number in the Dn register is as follows:

	39 32	31	30 23	22 0
Dn	0	S	E - Exponent	M - Mantissa

The floating point value for normalized (E>0) is calculated as: $FP = (-1)^s \times 1.M \times 2^{E-127}$.

The floating point value for de-normalized (E=0) is calculated as: $FP = (-1)^s \times 0.M \times 2^{-127}$.

De-normalized input is considered by the SC3900FP to have zero value. The SC3900FP does not generate a denormalized number other than zero as a calculation result.

SC3900FP native support includes multiply, add, subtract, fused multiply-add, compare, and convert operations. In addition, internal lookup tables help fast implementation of 1/x and 1/sqrt(x) functions. Each DMU is capable of performing up to two fused multiply-add operations, giving a total of 16 floating point operations per cycle – 19.2 GFLOPS when running at 1.2 GHz clock.

The SC3900FP floating-point is partially compliant with the IEEE 754, except for the following issues:

- Only round-to-nearest-even rounding method is supported (IEEE default rounding).
- De-normalized input is considered by the SC3900FP to have zero value (flushed to zero). The SC3900FP does not generate denormalized numbers other than zero as a calculation result.
- No hardware exceptions are automatically generated.
- When using a two predicates compare instruction, the second predicate (the predicate which is set when a condition is not met) is set to true even when either input is NaN (in IEEE, both conditions, for example, EQ and NE would return false for a NaN).
- There is no quieting of signaling NaNs.
- Float-to-Fixed (FLT2FIX) instructions do not perform rounding.

Besides normalized and denormalized numbers, IEEE floating-point number representation also consists of special symbols such as NaN (not a number), and infinity. The representation for these symbols is summarized in the following table.

Table 3-3. Floating Point Special Symbols

Symbol	Sign (S)	Exponent (E)	Mantissa (M)	Hexadecimal representation
+ 0	0	0	0	0X0000_0000
– 0	1	0	0	0X8000_0000
+ ∞	0	255	0	0X7F80_0000
– ∞	1	255	0	0XFF80_0000
QNaN (quiet NaN)	x	255	Bit 22 is 1.	0X7FC0_0000 (example)
SNaN (signaling NaN)	x	255	Bit 22 is 0. Other bits cannot be all zeros.	0X7FBF_FFFF (example)
QNaN output	0	255	All ones	0X7FFF_FFFF

3.3 Main numerical capabilities

The SC3900FP DALU is capable of performing high computational calculations required for Digital Signal Processing, such as multiplication, addition, multi-bit shift, logic operation and special application-specific functions. The following is a description of the main functionality of the DALU instructions. Note that not all of the instructions are herein described. A full description of the instruction set can be found in Appendix C, “Instruction Set”.

3.3.1 Fixed point multiplication

Each DMU within the DALU contains eight 16-bit x 16-bit multipliers. All or part can participate in multiplication or multiply-accumulate (MAC) operations.

The operation of part of the instructions changes according to the value of bits SR.W20 and SR.SM2 in the SR (see Section 3.4.1, “Arithmetic modes,” for details). These instructions are described here, according to the operation in which both SR.W20 and SR.SM2 bits are negated (default mode). In general, these instructions include MAC instructions with a fraction numeric type, no saturation and a 16-bit/20-bit destination (legacy SC3850 instructions).

There are many flavors of multiplication instructions implemented in the DMU. Main multiplication groups are described in the following sections.

3.3.1.1 Single/SIMD Real 16-bit x 16-bit into 32/40-bit multiplication

The “basic” multiplication operation of the SC3900FP core. A variety of instructions are included in this group, with the following flavors (not all combinations are implemented):

- MPY and MAC
  - “mpy” instruction: Multiplication, without accumulate value (for example $\mathrm{Result} = \pm A \times B$)
  - “mac” instructions: Multiplication, with accumulation (for example $\mathrm{Acc} = \mathrm{Acc} \pm A \times B$)
- Numeric type
  - Signed fraction
  - Signed integer (flag “i”)
  - Unsigned integer (flag “uu.i”, or mixed with signed “us.i” or “su.i”
- Saturation
  - Not saturated
  - Saturated (flag “s”) – if the result (after the accumulation) cannot be represented in 32-bit, it is saturated to 0x00_7FFF_FFFF if positive and to 0xFF_8000_0000 if negative
    (no integer, saturated multiply, since result cannot overflow)
- SIMD (how many outputs are calculated by the instruction
  - No SIMD (suffix “.x”) – Single output value is calculated and written into a single register
  - SIMD2 (suffix “.2x”) – Two output values calculated and written into two registers
  - SIMD2 with shard input (suffix “.2x”) – One of the inputs of the two multiplications is shared

Examples:

```
mpy.x d0.h,d1.h,d2; d2 = (long)d0.h * (long)d1.h << 1
mac.i.x d2.l,d4.h,d7; d7 = d7 + (long)d2.h * (long)d4.h
mac2.s.2x -d5,d7,d8:d9; d8 = Sat32(d8 - ((long)d5.h * (long)d7.h) << 1))
      ; d9 = Sat32(d9 - ((long)d5.l * (long)d7.l) << 1))
mpy2.h.i.2x d5,d6,d3:d4; d3 = d8 + ((long)d5.h * (long)d6.h))
      ; d4 = d9 + ((long)d5.h * (long)d6.l))
```

3.3.1.2 SIMD Real 16-bit x 16-bit into 16/20-bit multiplication

Multiplication with scaling into 16/20-bit. A variety of instructions are included in this group, with the following flavors (not all combinations are implemented):

- MPY and MAC
  - “mpy” instruction: Multiplication, without accumulate value (for example $\mathrm{Result} = \pm A \times B$).
  - “mac” instructions: Multiplication, with accumulation (for example $\mathrm{Acc} = \mathrm{Acc} \pm A \times B$).
- Numeric type
  - Signed fraction
  - Signed integer (flag “i”)
  - Unsigned integer (flag “su.i”
  - Signed integer low (flag “l.i”)
    Write low 16-bit of the 32-bit multiplication result to the result register (“C” compliant)
- Saturation
  - Not saturated. Output is 20-bit value
  - Saturated (flag “s”) – Accumulate value is read as 16-bit (ignoring the high nibble of the 20-bit word). Output is 16-bit value High nibble of the result (or accumulate output) is always written as zero. If the result (after the round, scale and accumulation) cannot be represented in 16-bit, it is saturated to 0x7FFF if positive and to 0x8000 if negative.
- Round
  - Round (flag “r”) – Most of the instruction has round of the result (for example $\mathrm{Acc} = \mathrm{Acc} \pm (((A \times B) + 0x8000) \gg 16)$ for integer accumulation with round or $\mathrm{Result} = \pm((A \times B) + 0x4000) \gg 15$ for fraction multiplication with round)
  - Truncate – Just shift right the result by 16
- SIMD (how many outputs are calculated by the instruction
  - SIMD2 (suffix “.2w” or “.2t”) – Two output values calculated and written into one registers
  - SIMD4 (suffix “.4w” or “.4t”) – Four output values calculated and written into two registers
  - SIMD4 with shard input (suffix “.4w” or “.4t”) – Couple of the inputs of the four multiplications is shared

Examples:

```
mpy.2t d0,d1; d2.wh = (((long)d0.h * (long)d1.h) + 0x4000) >> 15
      ; d2.wl = (((long)d0.l * (long)d1.l) + 0x4000) >> 15
mpy.l.i.4t d5:d6,d8:d9,d0:d1; d0.wh = d5.h * d8.h
      ; d0.wh = d5.l * d8.l
      ; d1.wh = d6.h * d9.h
      ; d1.wh = d6.l * d9.l
mac.sr.2w -d5,d6,d7; d7.h = Sat16(d7.h - ((((long)d5.h * (long)d6.h) + 0x4000) >> 15))
      ; d7.l = Sat16(d7.l - ((((long)d5.l * (long)d6.l) + 0x4000) >> 15))
      ; d7.e = 0
```

3.3.1.3 Single/SIMD Dot Product 16-bit x 16-bit into 32/40-bit multiplication

SIMD of dot product multiplication operation of the SC3900FP core into wide 32/40-bit result, summing two multiplications. Flavors of the instruction enable generation of the real part, imaginary part or conjugate imaginary part as well. A variety of instructions are included in this group, with the following flavors (not all combinations are implemented):

- MPY and MAC
  - — “mpyd” instruction: Multiplication, without accumulate value (for example $\text{Result} = A \times B \pm C \times D$).
  - — “macd” instructions: Multiplication, with accumulation (for example $\text{Acc} = \text{Acc} \pm A \times B \pm C \times D$).
- Numeric type
  - — Signed fraction
  - — Signed integer (flag “i”)
- Complex operation
  - — Real dot multiplication. Also real part of complex conjugate multiplication
  - — Real part of complex multiplication (flag “re”). Also real dot multiplication, subtracting the second multiplication result
  - — Imaginary part of complex multiplication (flag “im”)
  - — Imaginary part of complex conjugate multiplication (flag “cim”)
- Saturation
  - — Not saturated
  - — Saturated (flag “s”) – if the result (after the accumulation) cannot be represented in 32-bit, it is saturated to 0x00_7FFF_FFFF if positive and to 0xFF_8000_0000 if negative
- SIMD (how many outputs are calculated by the instruction
  - — No SIMD (suffix “.x”) – Single output value is calculated and written into a single register
  - — SIMD2 (suffix “.2x) – Two output values calculated and written into two registers
  - — SIMD2 with shard input (suffix “.2x”) – One of the inputs is shared

Examples:

```
mpyd.x d0,d2,d5; d5 = ((long)d0.h * (long)d2.h + (long)d0.l * (long)d2.l) << 1
macd.s.x d3,d7,d9; d9 = Sat32(d9 + ((long)d3.h * (long)d3.h + (long)d3.l * (long)d7.l) << 1))
macd.re.i.2x d5:d6,d7:d8,d0:d1
; d0 = d0 + ((long)d5.h * (long)d7.h - (long)d5.l * (long)d7.l)
; d1 = d1 + ((long)d6.h * (long)d8.h - (long)d6.l * (long)d8.l)
mpyd.s.2x d4,d0:d1,d2:d3
; d2 = Sat32((long)d4.h * (long)d0.h + (long)d4.l * (long)d0.l) << 1
; d3 = Sat32((long)d4.h * (long)d1.h + (long)d4.l * (long)d1.l) << 1
macd.im.is.2x d7:d8,d3:d4,d0:d1
; d0 = Sat32(d0 + ((long)d7.h * (long)d3.l + (long)d7.l * (long)d3.h))
; d1 = Sat32(d1 + ((long)d8.h * (long)d4.l + (long)d8.l * (long)d4.h))
macd.cim.s.2x -d5:d6,d3:d4,d7:d8
; d7 = Sat32(d7 - (((long)d5.h * (long)d3.l - (long)d5.l * (long)d3.h) << 1))
; d8 = Sat32(d8 - (((long)d6.h * (long)d4.l - (long)d6.l * (long)d4.h) << 1))
```

3.3.1.4 SIMD Dot Product 16-bit x 16-bit into 16/20-bit multiplication

SIMD of dot product multiplication operation of the SC3900FP core into 16/20-bit result, summing two multiplications. Flavors of the instruction enable generation of the real part, imaginary part or conjugate imaginary part as well. A variety of instructions are included in this group, with the following flavors (not all combinations are implemented):

- MPY and MAC
  - “mpyd” instruction: Multiplication, without accumulate value (for example $\mathrm{Result} = A \times B \pm C \times D$).
  - “macd” instructions: Multiplication, with accumulation (for example $\mathrm{Acc} = \mathrm{Acc} \pm A \times B \pm C \times D$).
- Numeric type
  - Signed fraction
  - Signed integer (flag “i”)
- Complex operation
  - Real dot multiplication. Also real part of complex conjugate multiplication
  - Real part of complex multiplication (flag “re”). Also real dot multiplication, subtracting the second multiplication result
  - Imaginary part of complex multiplication (flag “im”)
  - Imaginary part of complex conjugate multiplication (flag “cim”)
- Saturation
  - Not saturated
  - Saturated (flag “s”) – Accumulate value is read as 16-bit (ignoring the high nibble of the 20-bit word). Output is 16-bit value High nibble of the result (or accumulate output) is always written as zero. If the result (after the round, scale and accumulation) cannot be represented in 16-bit, it is saturated to 0x7FFF if positive and to 0x8000 if negative.
- Round (all the instructions in the group are rounded)
  - Round (flag “r”) – Round of the result. Note that round is done on each complex multiplication and not on the sum of two complex multiplications (for example $\mathrm{Acc} = \mathrm{Acc} \pm (((A \times B) + 0x8000) » 16) \pm (((C \times D) + 0x8000) » 16)$ for integer accumulation with round or $\mathrm{Result} = (((A \times B) + 0x4000) » 15) \pm (((A \times B) + 0x4000) » 15)$ for fraction multiplication with round)
- SIMD (how many outputs are calculated by the instruction
  - SIMD2 (suffix “.2w” or “.2t”) – Two output values calculated and written into one registers

Examples:

```
mpyd.r.2t d5:d6,d7:d8,d0
; d0.wh = ((long)d5.h * (long)d7.h + (long)d5.l * (long)d7.l + 0x4000) >> 15
; d0.wl = ((long)d6.h * (long)d8.h + (long)d6.l * (long)d8.l + 0x4000) >> 15
macd.re.ir.2t d8:d9,d6:d7,d1
; d1.wh = d1.wh + (((long)d8.h * (long)d6.h - (long)d8.l * (long)d6.l + 0x8000) >> 16)
; d1.wl = d1.wl + (((long)d9.h * (long)d7.h - (long)d9.l * (long)d7.l + 0x8000) >> 16)
mpyd.im.sr.2w d4:d5,d0:d1,d2
; d2.h = Sat16(((long)d4.h * (long)d0.l + (long)d4.l * (long)d0.h + 0x8000) >> 16)
; d2.l = Sat16(((long)d5.h * (long)d1.l + (long)d5.l * (long)d1.h + 0x8000) >> 16)
; d2.e = 0
macd.cim.isr.2w d5:d6,d7:d8,d9
; d9.h = Sat16(d9.h + (((long)d5.h * (long)d7.l - (long)d5.l * (long)d7.h + 0x8000) >> 16))
; d9.l = Sat16(d9.l + (((long)d6.h * (long)d8.l - (long)d6.l * (long)d8.h + 0x8000) >> 16))
; d9.e = 0
```

3.3.1.5 Quad Dot Product 16-bit x 16-bit into 32/40-bit multiplication

Quad dot product multiplication operation of the SC3900FP core into wide 32/40-bit result, summing four multiplications. A variety of instructions are included in this group, with the following flavors (not all combinations are implemented):

- MPY and MAC
  - — "mpyq" instruction: Multiplication, without accumulate value (for example $\mathrm{Result} = A \times B + C \times D + E \times F + G \times H$).
  - — "macq" instructions: Multiplication, with accumulation (for example $\mathrm{Acc} = \mathrm{Acc} \pm (A \times B + C \times D + E \times F + G \times H)$).
- Numeric type
  - — Signed fraction
  - — Signed integer (flag "i")
- SIMD (how many outputs are calculated by the instruction)
  - — No SIMD (suffix ".x") – Single output value is calculated and written into a single register

Examples:

```
mpyq.x d0:d1,d5:d6,d8; d8 = ((long)d0.h * (long)d5.h + (long)d0.l * (long)d5.l +
     ;       (long)d1.h * (long)d6.h + (long)d1.l * (long)d6.l) << 1
macq.x -d7:d8,d4:d5,d3; d3 = d3 - (((long)d7.h * (long)d4.h + (long)d7.l * (long)d4.l +
     ;             (long)d8.h * (long)d5.h + (long)d8.l * (long)d5.l) << 1)
macq.i.x d1:d2,d3:d4,d0; d3 = d3 + ((long)d1.h * (long)d3.h + (long)d1.l * (long)d3.l +
     ;             (long)d2.h * (long)d4.h + (long)d2.l * (long)d4.l)
```

3.3.1.6 SIMD Complex 16-bit x 16-bit into 32/40-bit multiplication

The complex multiplication operation of the SC3900FP core into wide 32/40-bit result. A variety of instructions are included in this group, with the following flavors (not all combinations are implemented):

- MPY and MAC
  - — "mpycx" instruction: Multiplication, without accumulate value (for example $\mathrm{Result} = \pm A \times B$).
  - — "maccx" instructions: Multiplication, with accumulation (for example $\mathrm{Acc} = \mathrm{Acc} \pm A \times B$).

- Numeric type
  - Signed fraction
  - Signed integer (flag “i”)
- Complex operation
  - Complex multiplication
  - Complex multiplication by Conjugate (flag “c”)
  - Complex multiplication multiplied by J (flag “j”)
- Saturation
  - Not saturated
  - Saturated (flag “s”) – if the result (after the accumulation) cannot be represented in 32-bit, it is saturated to 0x00_7FFF_FFFF if positive and to 0xFF_8000_0000 if negative
- SIMD (how many outputs are calculated by the instruction
  - SIMD2 (suffix “.2x”) – One complex output value calculated and written into two registers

Examples:

```
mpycx.2x d0,d1,d2:d3; d2 = ((long)d0.h * (long)d1.h - (long)d0.l * (long)d1.l)) << 1
      ; d3 = ((long)d0.h * (long)d1.l + (long)d0.l * (long)d1.h)) << 1
maccx.c.i.2x d2,d4,d7:d8; d7 = d7 + (long)d2.h * (long)d4.h + (long)d2.l * (long)d4.l)
      ; d8 = d8 + (long)d2.h * (long)d4.l - (long)d2.l * (long)d4.h)
mpycx.j.is.2x d5,d7,d8:d9; d8 = Sat32(-(long)d5.h * (long)d7.l - (long)d5.l * (long)d7.h)))
      ; d9 = Sat32( (long)d5.h * (long)d7.h - (long)d5.l * (long)d7.l)))
```

3.3.1.7 SIMD Complex 16-bit x 16-bit into 16/20-bit multiplication

Complex multiplication with scaling into 16/20-bit. A variety of instructions are included in this group, with the following flavors (not all combinations are implemented):

- MPY and MAC
  - “mpycx” instruction: Multiplication, without accumulate value (for example $\mathrm{Result} = \pm A \times B$).
  - “maccx” instructions: Multiplication, with accumulation (for example $\mathrm{Acc} = \mathrm{Acc} \pm A \times B$).
- Numeric type
  - Signed fraction
  - Signed integer (flag “i”)
- Complex operation
  - Complex multiplication
  - Complex multiplication by Conjugate (flag “c”)
  - Complex multiplication multiplied by J (flag “j”)
- Saturation
  - Not saturated. Output is 20-bit value
  - Saturated (flag “s”) – Accumulate value is read as 16-bit (ignoring the high nibble of the 20-bit word). Output is 16-bit value High nibble of the result (or accumulate output) is always written as zero. If the result (after the round, scale and accumulation) cannot be represented in 16-bit, it is saturated to 0x7FFF if positive and to 0x8000 if negative.

- Round (all the instructions in the group are rounded)
  - Round (flag “r”) – Most of the instruction has round of the result (for example $\mathrm{Acc} = \mathrm{Acc} \pm (((A \times B) + 0x8000) \gg 16)$ for integer accumulation with round or $\mathrm{Result} = \pm((A \times B) + 0x4000) \gg 15$ for fraction multiplication with round)
- SIMD (how many outputs are calculated by the instruction
  - SIMD2 (suffix “.2w” or “.2t”) – One complex output values calculated and written into one registers
  - SIMD4 (suffix “.4w” or “.4t”) – Two complex output values calculated and written into two registers
  - SIMD4 with shard input (suffix “.4w” or “.4t”) – One of the complex inputs is shared

Examples:

```
mpycx.r.2t d0,d1,d2
; d2.wh = (((long)d0.h * (long)d1.h - (long)d0.l * (long)d1.l)) + 0x4000) >> 15
; d2.wl = (((long)d0.h * (long)d1.l + (long)d0.l * (long)d1.h)) + 0x4000) >> 15
maccx.c.ir.2t d2,d4,d7
; d7.wh = d7.wh + ((long)d2.h * (long)d4.h + (long)d2.l * (long)d4.l + 0x8000) >> 16
; d8.wl = d8.wl + ((long)d2.h * (long)d4.l - (long)d2.l * (long)d4.h + 0x8000) >> 16
mpycx.j.isr.2w d5,d7,d8
; d8.h = Sat16((-(long)d5.h * (long)d7.l - (long)d5.l * (long)d7.h + 0x8000) >> 16)
; d8.l = Sat16(( (long)d5.h * (long)d7.h - (long)d5.l * (long)d7.l + 0x8000) >> 16)
; d8.e = 0
maccx.c.sr.4w d2:d3,d4:d5,d7:d8
; d7.h = d7.h + Sat16(((long)d2.h * (long)d4.h + (long)d2.l * (long)d4.l + 0x4000) >> 15)
; d7.l = d7.l + Sat16(((long)d2.h * (long)d4.l - (long)d2.l * (long)d4.h + 0x4000) >> 15)
; d8.h = d8.h + Sat16(((long)d3.h * (long)d5.h + (long)d3.l * (long)d5.l + 0x4000) >> 15)
; d8.l = d8.l + Sat16(((long)d3.h * (long)d5.l - (long)d3.l * (long)d5.h + 0x4000) >> 15)
; d7.e = 0 ; d8.e = 0
mpycx.ir.4w d9,d4:d5,d7:d8
; d7.h = ((long)d9.h * (long)d4.h - (long)d9.l * (long)d4.l + 0x8000) >> 16
; d7.l = ((long)d9.h * (long)d4.l + (long)d9.l * (long)d4.h + 0x8000) >> 16
; d8.h = ((long)d9.h * (long)d5.h - (long)d9.l * (long)d5.l + 0x8000) >> 16
; d8.l = ((long)d9.h * (long)d5.l + (long)d9.l * (long)d5.h + 0x8000) >> 16
; d7.e = 0 ; d8.e = 0
```

3.3.1.8 SIMD Complex Dot 16-bit x 16-bit into 32/40-bit multiplication

The dot product complex multiplication operation of the SC3900FP core into wide 32/40-bit result. Variety of instructions are included in this group, with the following flavors (not all combinations are implemented):

- MPY and MAC
  - “mpycxd” instruction: Multiplication, without accumulate value (for example $\mathrm{Result} = A \times B \pm C \times D$).
  - “maccxd” instructions: Multiplication, with accumulation (for example $\mathrm{Acc} = \mathrm{Acc} \pm A \times B \pm C \times D$).
- Numeric type
  - Signed fraction

  - Signed integer (flag “i”)
- Complex operation
  - Complex multiplication
  - Complex multiplication by Conjugate (flag “c”)
- Addition/Subtraction
  - Add both multiplication (flag “pp”)
  - Add first multiplication and subtract second multiplication (flag “pn”)
  - subtract first multiplication and add second multiplication (flag “np”) - only for accumulation
  - subtract both multiplication (flag “nn”) - only for accumulation
- Saturation
  - Not saturated
  - Saturated (flag “s”) – if the result (after the accumulation) cannot be represented in 32-bit, it is saturated to 0x00_7FFF_FFFF if positive and to 0xFF_8000_0000 if negative
- SIMD (how many outputs are calculated by the instruction
  - SIMD2 (suffix “.2x) – One complex output value calculated and written into two registers
  - SIMD2 with shard input (suffix “.2x”) – One of the complex inputs is shared

Examples:

```
mpycxd.pp.2x d0:d1,d2:d3,d5:d6
; d5 = ((long)d0.h * (long)d2.h - (long)d0.l * (long)d2.l
      + (long)d1.h * (long)d3.h - (long)d1.l * (long)d3.l) << 1
; d6 = ((long)d0.h * (long)d2.l + (long)d0.l * (long)d2.h
      + (long)d1.h * (long)d3.l + (long)d1.l * (long)d3.h) << 1
maccxd.pn.i.2x d3:d4,d0:d1,d8:d9
; d8 = d8 + ((long)d3.h * (long)d0.h - (long)d3.l * (long)d0.l
           - (long)d4.h * (long)d1.h - (long)d4.l * (long)d1.l)
; d9 = d9 + ((long)d3.h * (long)d0.l + (long)d3.l * (long)d0.h
           - (long)d4.h * (long)d1.l + (long)d4.l * (long)d1.h)
maccxd.cnp.s.2x d7,d5:d6,d0:d1
; d0 = Sat32(d0 + (-(long)d7.h * (long)d5.h + (long)d7.l * (long)d5.l
                  + (long)d7.h * (long)d6.h + (long)d7.l * (long)d6.l) << 1)
; d1 = Sat32(d1 + (-(long)d7.h * (long)d5.l - (long)d7.l * (long)d5.h
                  + (long)d7.h * (long)d6.l - (long)d7.l * (long)d6.h) << 1)
```

3.3.1.9 SIMD Complex Dot 16-bit x 16-bit into 16/20-bit multiplication

Dot product of two complex multiplication with scaling into 16/20-bit. Variety of instructions are included in this group, with the following flavors (not all combinations are implemented):

- MPY and MAC
  - “mpycxd” instruction: Multiplication, without accumulate value (for example $\text{Result} = \pm A \times B \pm C \times D$).
  - “maccxd” instructions: Multiplication, with accumulation (for example $\text{Acc} = \text{Acc} \pm A \times B \pm C \times D$).
- Numeric type
  - Signed fraction

  - Signed integer (flag “i”)
- Complex operation
  - Complex multiplication
  - Complex multiplication by Conjugate (flag “c”)
- Saturation (all the instructions in the group are saturated)
  - Saturated (flag “s”) – Accumulate value is read as 16-bit (ignoring the high nibble of the 20-bit word). Output is 16-bit value High nibble of the result (or accumulate output) is always written as zero. If the result (after the round, scale and accumulation) cannot be represented in 16-bit, it is saturated to 0x7FFF if positive and to 0x8000 if negative.
- Round (all the instructions in the group are rounded)
  - Round (flag “r”) – Round of the result. Note that round is done on each complex multiplication and not on the sum of two complex multiplications (for example $Acc = Acc\pm (((A \times B) + 0x8000) » 16) \pm (((C \times D) + 0x8000) » 16)$ for integer accumulation with round or $Result = (((A \times B) + 0x4000) » 15) \pm(((A \times B) + 0x4000) » 15)$ for fraction multiplication with round)
- SIMD (how many outputs are calculated by the instruction
  - SIMD2 (suffix “.2w”) – One complex dot output value calculated and written into one registers
  - SIMD2 (suffix “.2w”) with shard input – One of the complex inputs is shared

Examples:

```
mpycxd.pp.sr.2w d0,d1:d2,d5
; d2.w = Sat16((((long)d0.h * (long)d1.h - (long)d0.l * (long)d1.l + 0x4000) >> 15)
;            + (((long)d0.h * (long)d2.h - (long)d0.l * (long)d2.l + 0x4000) >> 15))
; d2.w = Sat16((((long)d0.h * (long)d1.l + (long)d0.l * (long)d1.h + 0x4000) >> 15)
;            + (((long)d0.h * (long)d2.l + (long)d0.l * (long)d2.h + 0x4000) >> 15))
; d2.e = 0
maccxd.cpn.isr.2w d2:d3,d4:d5,d7
; d7.h = Sat16(d7.h + (((long)d2.h * (long)d4.h + (long)d2.l * (long)d4.l + 0x8000) >> 16)
;                   - (((long)d3.h * (long)d5.h + (long)d3.l * (long)d5.l + 0x8000) >> 16))
; d7.l = Sat16(d7.l + (((long)d2.h * (long)d4.l - (long)d2.l * (long)d4.h + 0x8000) >> 16)
;                   - (((long)d3.h * (long)d5.l - (long)d3.l * (long)d5.h + 0x8000) >> 16))
; d7.e = 0
maccxd.nn.sr.2w d0:d1,d7:d8,d9
; d9.h = Sat16(d9.h - (((long)d0.h * (long)d7.h - (long)d0.l * (long)d7.l + 0x4000) >> 15)
;                   - (((long)d1.h * (long)d8.h - (long)d0.l * (long)d8.l + 0x4000) >> 15))
; d9.l = Sat16(d9.l - (((long)d0.h * (long)d7.l + (long)d1.l * (long)d7.h + 0x4000) >> 15)
;                   - (((long)d1.h * (long)d8.l + (long)d1.l * (long)d8.h + 0x4000) >> 15))
; d9.e = 0
```

3.3.1.10 Real 16-bit x 32-bit into 32/40-bit multiplication

Mix precision multiplication operation of the SC3900FP core into wide 32/40-bit result. Variety of instructions are included in this group, with the following flavors (not all combinations are implemented):

- MPY and MAC
  - “mpym” instruction: Multiplication, without accumulate value (for example $Result = A \times B$).

  - “macm” instructions: Multiplication, with accumulation (for example $\mathrm{Acc} = \mathrm{Acc} \pm \mathrm{A} \times \mathrm{B}$).
- Numeric type
  - Signed fraction
  - Signed integer (flag “i”)
- Round (all the instructions in the group are rounded)
  - Round (flag “r”) – Round of the result from 48-bit to 32-bit (for example $\mathrm{Acc} = \mathrm{Acc} \pm (((\mathrm{A} \times \mathrm{B}) + 0\mathrm{x}8000) \gg 16)$ for integer accumulation with round or $\mathrm{Result} = \pm((\mathrm{A} \times \mathrm{B}) + 0\mathrm{x}4000) \gg 15$ for fraction multiplication with round)
- Saturation
  - Not saturated
  - Saturated (flag “s”) – if the result (after the accumulation) cannot be represented in 32-bit, it is saturated to 0x00_7FFF_FFFF if positive and to 0xFF_8000_0000 if negative
- SIMD (how many outputs are calculated by the instruction
  - No SIMD – Single output value is calculated and written into a single register (32-bit result marked with “.x” suffix) or two registers (64-bit result marked with “.ll” suffix)
  - SIMD2 (suffix “.2x”) – Two output values calculated and written into two registers

Examples:

```
mpym.r.x d0.h,d1,d5; d5 = ((longlong)d0.h * (longlong)d1.m + 0x4000) >> 15
macm.ir.2x -d7,d8:d9,d0:d1
; d0 = d0 - (((longlong)d7.h * (longlong)d8.m + 0x8000) >> 16)
; d1 = d1 - (((longlong)d7.l * (longlong)d9.m + 0x8000) >> 16)
mpym.i.ll d5.l,d7,d8:d9; {d8.m:d9.m} = (longlong)d5.l * (longlong)d7.m
     ; d8.e = SignExtend(d8.m), d9.e = 0
```

3.3.1.11 SIMD Complex 16-bit x 32-bit into 32/40-bit multiplication

Mix precision complex multiplication operation of the SC3900FP core into wide 32/40-bit result. Variety of instructions are included in this group, with the following flavors (not all combinations are implemented):

- MPY and MAC
  - “mpycxm” instruction: Multiplication, without accumulate value (for example $\mathrm{Result} = \pm \mathrm{A} \times \mathrm{B}$).
  - “maccxm” instructions: Multiplication, with accumulation (for example $\mathrm{Acc} = \mathrm{Acc} \pm \mathrm{A} \times \mathrm{B}$).
- Numeric type
  - Signed fraction
  - Signed integer (flag “i”)
- Complex operation
  - Complex multiplication
  - Complex multiplication by Conjugate (flag “c”)
- Round (all the instructions in the group are rounded)

- Round (flag “r”) – Round of the result from 48-bit to 32-bit (for example $\mathrm{Acc} = \mathrm{Acc} \pm (((\mathrm{A} \times \mathrm{B}) + 0\mathrm{x}8000) \gg 16)$ for integer accumulation with round or $\mathrm{Result} = \pm((\mathrm{A} \times \mathrm{B}) + 0\mathrm{x}4000) \gg 15$ for fraction multiplication with round)

- Saturation
  - Not saturated
  - Saturated (flag “s”) – if the result (after the accumulation) cannot be represented in 32-bit, it is saturated to 0x00_7FFF_FFFF if positive and to 0xFF_8000_0000 if negative
- SIMD (how many outputs are calculated by the instruction
  - SIMD2 (suffix “.2x”) – One complex output value calculated and written into two registers

Examples:

```
   mpycxm.r.2x d0,d1:d2,d5:d6
; d5 = ((longlong)d0.h * (longlong)d1.m - (longlong)d0.l * (longlong)d2.m) + 0x4000) >> 15
; d6 = ((longlong)d0.h * (longlong)d2.m + (longlong)d0.l * (longlong)d1.m) + 0x4000) >> 15
   maccxm.c.ir.2x -d5,d7:d8,d0:d1
; d0 = d0 - ((longlong)d5.h * (longlong)d7.m + (longlong)d5.l * (longlong)d8.m) + 0x8000) >> 16
; d1 = d1 - ((longlong)d5.h * (longlong)d8.m - (longlong)d5.l * (longlong)d7.m) + 0x8000) >> 16
   mpycxm.sr.2x d7,d8:d9,d2:d3
; d2 = Sat32((longlong)d7.h * (longlong)d8.m - (longlong)d7.l * (longlong)d9.m) + 0x4000) >> 15)
; d3 = Sat32((longlong)d7.h * (longlong)d9.m + (longlong)d7.l * (longlong)d8.m) + 0x4000) >> 15)
```

3.3.1.12 Single/SIMD real 32-bit x 32-bit into 32-bit or 64-bit multiplication

Double precision multiplication operation of the SC3900FP core. A 64-bit target is supported by multiply only. Variety of instructions are included in this group, with the following flavors (not all combinations are implemented):

- MPY and MAC
  - “mpy32” instruction: Multiplication, without accumulate value (for example $\mathrm{Result} = \pm\mathrm{A} \times \mathrm{B}$).
  - “mac32” instructions: Multiplication, with accumulation (for example $\mathrm{Acc} = \mathrm{Acc} \pm \mathrm{A} \times \mathrm{B}$).
- Numeric type
  - Signed fraction
  - Signed integer (flag “i”)
  - Unsigned integer (flag “uu.i”, or mixed with signed “us.i” or “su.i”
  - Signed integer low (flag “l.i”)
    Write low 32-bit of the 64-bit multiplication result to the result register (“C” compliant)
- Saturation
  - Not saturated
  - Saturated (flag “s”) – When taking the high part of a fraction, if the result (after the accumulation) cannot be represented in 32-bit, it is saturated to 0x00_7FFF_FFFF if positive and to 0xFF_8000_0000 if negative
- Round
  - Round (flag “r”) – round of the result (for example $\mathrm{Acc} = \mathrm{Acc} \pm (((\mathrm{A} \times \mathrm{B}) + 0\mathrm{x}8000\overline{0000}) \gg 32)$ for integer accumulation with round or $\mathrm{Result} = \pm((\mathrm{A} \times \mathrm{B}) + 0\mathrm{x}4000000) \gg 31$ for fraction multiplication with round)

  - No LSBits lost
- SIMD (how many outputs are calculated by the instruction
  - No SIMD – Single output value is calculated and written into a single register (32-bit result marked with ".x" suffix) or two registers (64-bit result marked with ".ll" suffix)
  - SIMD2 (suffix ".2x") – Two 32-bit output values calculated and written into two registers
  - SIMD2 with shard input (suffix ".2x") – One of the inputs of the two multiplications is shared

Examples:

```
mpy32.i.ll d0,d1,d2:d3; {d2.m:d3.m} = (longlong)d0.m * (longlong)d1.m << 1
     ; d2.e = SignExtend(d2.m), d3.e = 0
mac32.l.i.l -d2,d4,d7; d7.m = d7.m - (d2.m * d1.M)%32
mpy32.h.r.2x d5:d6,d7:d8,d1:d2; d1 = ((longlong)d5 * (longlong)d7) + 0x4000_0000 >> 31))
     ; d2 = ((longlong)d6 * (longlong)d8) + 0x4000_0000 >> 31))
```

3.3.2 Fixed point addition and subtraction

Each DMU within the DALU contains two 40-bit adders and four 20-bit adders. All or part can participate in addition or subtraction operations.

The operation of part of the instructions changes according to the value of bits SR.W20 and SR.SM2 in the SR (see Section 3.4.1, "Arithmetic modes," for details). These instructions are described here, according to the operation in which both SR.W20 and SR.SM2 bits are negated (default mode). In general, these instructions include SOD instructions with no saturation and 16-bit/20-bit destination (legacy SC3850 instructions).

There are many flavors of add/sub instructions implemented in the DMU. The main groups are described in the following sections.

NOTE

> Unlike multiplication, there is no difference between integer and fraction as well as signed and unsigned; since all are supported, these terms do not appear in the instruction's description.

3.3.2.1 Single/SIMD 16-bit/20-bit addition/subtraction

The "basic" 16-bit/20-bit addition/subtraction operation of the SC3900FP core. A variety of instructions are included in this group, with the following flavors (not all combinations are implemented):

- ADD, SUB and SOD
  - "add" instruction: addition of two operands (for example $\mathrm{Result} = A + B$).
  - "sub" instruction: Subtraction of two operands (for example $\mathrm{Result} = A - B$).
  - "sod" instructions: SIMD instructions with combination of addition and Subtraction, possible with "cross" between one of the additives input vector (for example $\mathrm{Result}[\mathrm{High}] = B[\mathrm{High}] - A[\mathrm{Low}]$; $\mathrm{Result}[\mathrm{Low}] = B[\mathrm{Low}] + A[\mathrm{High}]$).
    Note that SIMD2/4 addition and subtraction are actually aliases of SOD instruction flavors
- Add and sub flavors of SOD
  - Add (flag "a") – This result (one of two or one of four) is calculated as addition

  - subtract (flag “s”) – This result (one of two or one of four) is calculated as Subtraction

  The “a” and “s” always come as a couple (for SIMD2) or quadruplets (for SIMD4), unlike single “s” that denotes saturation. Not all combinations of four a/s are available.

- Cross flavors of SOD
  - No cross (flag “i” or “ii”) – SIMD vector addition/Subtraction
  - Cross (flag “x” or “xx”) – High and low portions of the first input are swapped (for both first inputs in case of SIMD4)
- Saturation
  - Not saturated
  - Saturated (flag “s”) – Input values are read as 16-bit (ignoring the high nibble of the 20-bit word). Output is 16-bit value. High nibble of the result is always written as zero. If the result cannot be represented in 16-bit, it is saturated to 0x7FFF if positive and to 0x8000 if negative.
- SIMD (how many outputs are calculated by the instruction
  - No SIMD (suffix “.t”) – Single output value is calculated and written into a half register (20-bit). Second half remains unchanged.
  - SIMD2 (suffix “.2t” or “.2w”) – Two output values calculated and written into one register
  - SIMD4 (suffix “.4t” or “.4w”) – Four output values calculated and written into two registers

Examples:

```
add.t d0.l,d1.h; d1.wh = d1.wh + d0.wl (d1.wl unchanged)
sub.t d2.h,d5.l; d5.wl = d5.wl - d2.wh (d5.wh unchanged)
sod.aaii.2t d6,d8,d9; (actually a “add.2t” instruction)
      ; d9.wh = d8.wh + d6.wh
      ; d9.wl = d8.wl + d6.wl
sod.asxx.s.2w d7,d4,d2; d2.h = Sat16(d4.h + d7.l)
      ; d2.l = Sat16(d4.l - d7.h)
      ; d2.e = 0
sod.aassx.4t d0:d1,d3:d4,d8:d9; d8.wh = d3.wh + d0.wl
      ; d8.wl = d3.wl + d0.wh
      ; d9.wh = d4.wh - d1.wl
      ; d9.wl = d4.wl - d1.wh
sod.asasi.s.4w d2:d3,d6:d7,d6:d7; d6.h = Sat16(d6.h + d2.wh
      ; d6.l = Sat16(d6.l - d2.wl
      ; d7.h = Sat16(d7.h + d3.wh
      ; d7.l = Sat16(d7.l - d3.wl
```

3.3.2.2 Single/SIMD 32-bit/40-bit addition/subtraction

The “basic” 32-bit/40-bit addition/subtraction operation of the SC3900FP core. A variety of instructions are included in this group, with the following flavors (not all combinations are implemented):

- ADD, SUB, ADDC, SUBC and SOD
  - “add” instruction: addition of two operands (for example $\text{Result} = A + B$).
  - “sub” instruction: subtraction of two operands (for example $\text{Result} = A - B$).
  - “addc” instruction: addition of two operands (for example $\text{Result} = A + B$), which either updates the carry bit (with result’s carry) or adds it.

- “subc” instruction: subtraction of two operands (for example $\mathrm{Result} = A - B$), which either updates the carry bit (with result’s borrow) or subtracts it.
- “sod” instructions: SIMD instructions with combination of addition and subtraction, possible with “cross” between one of the additives input vector (for example, $\mathrm{Result[High]} = \mathrm{B[High]} - \mathrm{A[Low]}$; $\mathrm{Result[Low]} = \mathrm{B[Low]} + \mathrm{A[High]}$).
  Note that SIMD2 addition and subtraction are actually aliases of SOD instruction flavors

- Add and sub flavors of SOD
  - Add (flag “a”) – This result (one of two or one of four) is calculated as addition
  - subtract (flag “s”) – This result (one of two or one of four) is calculated as Subtraction

  The “a” and “s” always come as a couple (for SIMD2) or quadruplets (for SIMD4), unlike single “s” that denotes saturation. Not all combinations of four a/s are available.
- Cross flavors of SOD
  - No cross (flag “ii”) – SIMD vector addition/subtraction
  - Cross (flag “xx”) – High and low portions of the first input are swapped
- Carry flavors of ADDC and SUBC
  - Read only (flag “ro”) – Add or subtraction carry flag value to the result
  - Write only (flag “wo”) – Update the carry flag with carry or borrow value.
    Note that only one instruction in a VLES can have the “wo” or “rw” attribute.
  - Read and write (flag “ro”) – Add or subtraction carry flag value to the result and update the carry flag with carry or borrow value.
    Note that only one instruction in a VLES can have the “wo” or “rw” attribute.

  For more details on carry calculation, see Section 3.4.3.1, “Carry bit flag.”
- Saturation
  - Not saturated
  - Saturated (flag “s”) – if the result cannot be represented in 32-bit, it is saturated to 0x00_7FFF_FFFF if positive and to 0xFF_8000_0000 if negative.
    Not supported for ADDC and SUBC
- SIMD (how many outputs are calculated by the instruction)
  - No SIMD (suffix “.x” or “.l”) – Single output value is calculated and written into a single register. The “.l” suffix refers to ADDC/SUBC instructions that update the carry flag according to 32-bit result rather than according to 40-bit result. The result has the same 40-bit as in equivalent “.x” instruction (although upper 8-bit will be normally ignored).
  - SIMD2 (suffix “.2x”) – Two output values calculated and written into two registers
    Not supported for ADDC and SUBC.

Examples:

```
add.x d0,d1,d2; d2 = d1 + d0
sub.s.x d2,d5,d8; d8 = Sat32(d5 - d2)
add.x 37,d8,d2; d2 = d8 + 37
sod.aaii.2x d6:d7,d8:d9,d3:d4; (actually a “add.2x” instruction)
      ; d3 = d8 + d6
      ; d4 = d9 + d7
sod.asxx.s.2x d7:d8,d4:d5,d2:d3; d2 = Sat32(d4 + d8)
```

```
      ; d3 = Sat32(d5 - d7)
addc.rw.l d5,d6,d7; d7 = d6 + d5 + SR.C
      ; SR.C = Carry(d6.m + d5.m + SR.C)
subc.wo.x d8,d9,d0; d0 = d9 - d8
      ; SR.C = Borrow(d9 - d8)
subc.ro.x d7,d3,d2; d2 = d3 - d7 - SR.C
subc.wo.l #17,d7,d1; d1 = d7 - 17
      ; SR.C = Borrow(d7.m - 17)
```

3.3.2.3 Mixed 16-bit/20-bit and 32-bit/40-bit integer addition

Add one or two 16-bit integer number with 32-bit/40-bit number. A variety of instructions are included in this group, with the following flavors (not all combinations are implemented):

- ADDM
  - “addm” instruction: addition of one or two 16-bit integer with 32-bit/40-bit integer (for example, $\mathrm{Result} = A + B$ or $\mathrm{Result} = A + B + C$).
- HL flavor
  - Single 16-bit– Adds a 16-bit integer value (sign extended) to 32-bit/40-bit value
  - Two 16-bit (flag “HL”) – Adds two 16-bit values that are packed in one register (both sign extended), and add the result to 32-bit/40-bit value
- SIMD (how many outputs are calculated by the instruction
  - No SIMD (suffix “.x”) – Single output value is calculated and written into a single register
  - SIMD2 (suffix “.2x”) – Two output values calculated and written into two registers

Examples:

```
addm.x d0.h,d1,d2; d2 = d1 + d0.h
addm.hl.x d5,d6,d7; d7 = d6 + d5.h + d5.l
addm.hl.2x d7:d8,d2:d3,d6:d7; d6 = d2 + d7.h + d7.l
      ; d7 = d3 + d8.h + d8.l
```

3.3.2.4 64-bit addition/subtraction

Add or subtract two 64-bit numbers. A variety of instructions are included in this group, with the following flavors (not all combinations are implemented):

- ADD and SUB
  - “add” instruction: addition of two operands (for example, $\mathrm{Result} = A + B$).
  - “sub” instruction: Subtraction of two operands (for example, $\mathrm{Result} = A - B$).
- SIMD (how many outputs are calculated by the instruction
  - No SIMD (suffix “.ll”) – Single output value is calculated and written into two registers

Examples:

```
add.ll d0:d1,d3:d4,d5:d6; {d5.m:d6.m} = {d3.m:d4.m} + {d0.m:d1.m}
      ; d5.e = SignExtend(d5.m), d6.e = 0
sub.ll d2:d3,d6:d7,d8:d9; {d8.m:d9.m} = {d6.m:d7.m} - {d2.m:d3.m}
      ; d8.e = SignExtend(d8.m), d9.e = 0
```

3.3.3 Multi-bit shift operations

Each DMU within the DALU contains one 64-bit multi-bit shifter (or barrel shifter), two 32-bit/40-bit multi-bit shifters and four 16-bit/20-bit multi-bit shifters. Each multi-bit instruction as well as other related instructions.

The operation of part of the instructions changes according to the value of bits SR.W20 and SR.SM2 in the SR (see Section 3.4.1, “Arithmetic modes for more details). These instructions are described here according to the operation when both SR.W20 and SR.SM2 bits are negated (default mode). In general these instructions include shift instructions with no saturation and 16-bit/20-bit destination (legacy SC3850 instructions).

NOTE

> Unlike multiplication, there is no difference between integer and fraction; since all are supported, these terms do not appear in the instruction’s description.

Unlike other instructions, there is saturation support for 20-bit saturated into 20-bit and into 16-bit values as well as 32-bit values into 32-bit values.

The shift offset is either an immediate field or signed value taken as the lower bits of 40-bit value stored in a Dn register. In SIMD shift instructions all of the operands are shifted by the same amount.

There are many flavors of multi-bit shift instructions implemented in the DMU. The main groups are described in the following sections.

3.3.3.1 SIMD 16-bit/20-bit Multi-bit shift

Multi-bit shift operation of 16-bit and 20-bit operands in the SC3900FP core. Variety of instructions are included in this group, with the following flavors (not all combinations are implemented):

- Arithmetic shift and logical shift
  - “ash” instruction: arithmetic shift, sign extended on shift right (for example, $\text{Result} = A \gg B$).
  - “lsh” instruction: logical shift, zero extended on shift right (for example, $\text{Result} = A \ll B$). Supported only by non-saturated 16-bit instructions
- Shift direction
  - Right (flag “rgt”) – Shift right (shift left if shift value is negative)
  - Left (flag “lft”) – Shift left (shift right if shift value is negative)
- Saturation
  - Not saturated
  - Saturated (flag “s”) –
    - For 16-bit instruction (suffix “.2w” or “.4w”), input values are read as 16-bit (ignoring the high nibble of the 20-bit word). Output is 16-bit value. High nibble of the result is always written as zero. If the result cannot be represented in 16-bit, it is saturated to 0x7FFF if source is positive (bit 15 is cleared) and to 0x8000 if source is negative (bit 15 is set)

    - For 20-bit instruction (suffix “.2t” or “.4t”), input values are read as 20-bit. Output is 20-bit value. If the result cannot be represented in 16-bit, it is saturated to 0x0_7FFF if source is positive (bit 19 is cleared) and to 0x0_8000 if source is negative (bit 19 is set)
  - Saturated (flag “s20”) –
    - For 20-bit instruction (suffix “.2t” or “.4t”), input values are read as 20-bit. Output is 20-bit value. If the result cannot be represented in 20-bit, it is saturated to 0x7_FFFF if source is positive (bit 19 is cleared) and to 0x8_0000 if source is negative (bit 19 is set)
- SIMD (how many outputs are calculated by the instruction
  - SIMD2 (suffix “.2t” or “.2w”) – Two output values calculated and written into one register
  - SIMD4 (suffix “.4t” or “.4w”) – Four output values calculated and written into two registers

  The same shift value is used for all the shifted operands (two or four).

Examples:

```
ash.rgt.2t d0,d1,d2; shift = (d0 << (40 - 6)) >> (40 - 6) ; Take 6 LSBits
      ; d2.wh = (shift<0) ? d1.wh << -shift : d1.wh >> shift
      ; d2.wl = (shift<0) ? d1.wl << -shift : d1.wl >> shift
ash.lft.2w d3,d4,d5; shift = (d3 << (40 - 5)) >> (40 - 5) ; Take 5 LSBits
      ; d5.h = (shift<0) ? d1.h >> -shift : d1.h << shift
      ; d5.l = (shift<0) ? d1.l >> -shift : d1.l << shift
      ; d5.e = 0
ash.rgt.s20.2t d4,d7,d9; shift = (d4 << (40 - 6)) >> (40 - 6) ; Take 6 LSBits
      ; d9.wh = Sat20((shift<0) ? d7.wh << -shift : d7.wh >> shift)
      ; d9.wl = Sat20((shift<0) ? d7.wl << -shift : d7.wl >> shift)
ash.rgt.s.2t d3,d5,d8; shift = (d3 << (40 - 6)) >> (40 - 6) ; Take 6 LSBits
      ; d8.wh = Sat16((shift<0) ? d5.wh << -shift : d5.wh >> shift)
      ; d8.wl = Sat16((shift<0) ? d5.wl << -shift : d5.wl >> shift)
ash.lft.s.2w d3,d4,d5; shift = (d3 << (40 - 5)) >> (40 - 5) ; Take 5 LSBits
      ; d5.h = Sat16((shift<0) ? d4.h >> -shift : d4.h << shift)
      ; d5.l = Sat16((shift<0) ? d4.l >> -shift : d4.l << shift)
      ; d5.e = 0
ash.rgt.4t #15,d3:d4,d5:d6; d5.wh = d3.wh >> 15
      ; d5.wl = d3.wl >> 15
      ; d6.wh = d4.wh >> 15
      ; d6.wl = d4.wl >> 15
ash.lft.s.4w d0,d2:d3,d6:d7
      ; shift = (d0 << (40 - 5)) >> (40 - 5) ; Take 5 LSBits
      ; d6.h = Sat16((shift<0) ? d2.h >> -shift : d2.h << shift)
      ; d6.l = Sat16((shift<0) ? d2.l >> -shift : d2.l << shift)
      ; d7.h = Sat16((shift<0) ? d3.h >> -shift : d3.h << shift)
      ; d7.l = Sat16((shift<0) ? d3.l >> -shift : d3.l << shift)
      ; d6.e = 0, d7.e = 0
lsh.rgt.2w d7,d8,d9; shift = (d7 << (40 - 5)) >> (40 - 5) ; Take 5 LSBits
; d9.h = (shift<0) ? (unsigned word)d8.h << -shift : (unsigned word)d8.h >> shift
; d9.l = (shift<0) ? (unsigned word)d8.l << -shift : (unsigned word)d8.l >> shift
; d9.e = 0
lsh.lft.4w d5,d8:d9,d8:d9; shift = (d5 << (40 - 5)) >> (40 - 5) ; Take 5 LSBits
; d8.h = (shift<0) ? (unsigned word)d8.h >> -shift : (unsigned word)d8.h << shift
; d8.l = (shift<0) ? (unsigned word)d8.l >> -shift : (unsigned word)d8.l << shift
; d9.h = (shift<0) ? (unsigned word)d9.h >> -shift : (unsigned word)d8.h << shift
; d9.l = (shift<0) ? (unsigned word)d9.l >> -shift : (unsigned word)d8.l << shift
; d8.e = 0, d9.e = 0
```

3.3.3.2 SIMD 32-bit/40-bit multi-bit shift

Multi-bit shift operation of 32-bit and 40-bit operands in the SC3900FP core. A variety of instructions are included in this group, with the following flavors (not all combinations are implemented):

- Arithmetic shift and logical shift
  - “ash” instruction: arithmetic shift, sign extended on shift right (for example $\mathrm{Result} = A \gg B$).
  - “lsh” instruction: logical shift, zero extended on shift right (for example $\mathrm{Result} = A \ll B$). Supported only by non-saturated instructions
- Shift direction
  - Right (flag “rgt”) – Shift right (shift left if shift value is negative)
  - Left (flag “lft”) – Shift left (shift right if shift value is negative)
- Saturation
  - Not saturated
  - Saturated (flag “s”)
    - For 32-bit instruction (suffix “.l” or “.2l”), input values are read as 32-bit (ignoring the high byte of the 40-bit word). Output is 32-bit value. High byte of the result is always written as zero. If the result cannot be represented in 32-bit, it is saturated to 0x7FFF_FFFF if source is positive (bit 31 is cleared) and to 0x8000_0000 if source is negative (bit 31 is set)
    - For 40-bit instruction (suffix “.x” or “.2x”), input values are read as 40-bit. If the result cannot be represented in 32-bit, it is saturated to 0x00_7FFF_FFFF if source is positive (bit 39 is cleared) and to 0xFF_8000_0000 if source is negative (bit 39 is set)
- SIMD (how many outputs are calculated by the instruction
  - No SIMD (suffix “.x” or “.l”) – Single output value is calculated and written into a single register
  - SIMD2 (suffix “.2x” or “.2l”) – Two output values calculated and written into two registers

    The same shift value is used for both shifted operands.

Examples:

```
ash.rgt.x d0,d1,d2; shift = (d0 << (40 - 7)) >> (40 - 7) ; Take 7 LSBits
     ; d2 = (shift<0) ? d1 << -shift : d1 >> shift
ash.lft.l d3,d4,d5; shift = (d3 << (40 - 6)) >> (40 - 6) ; Take 6 LSBits
     ; d5.m = (shift<0) ? d1.m >> -shift : d1.m << shift
     ; d5.e = 0
ash.rgt.s.x d3,d5,d8; shift = (d3 << (40 - 7)) >> (40 - 7) ; Take 7 LSBits
     ; d8 = Sat32((shift<0) ? d5 << -shift : d5 >> shift)
ash.lft.s.l d3,d4,d5; shift = (d3 << (40 - 6)) >> (40 - 6) ; Take 6 LSBits
     ; d5.m = Sat32((shift<0) ? d4.m >> -shift : d4.m << shift)
     ; d5.e = 0
ash.rgt.2x #15,d3:d4,d5:d6; d5 = d3 >> 15
     ; d6 = d4 >> 15
ash.lft.s.2l d0,d2:d3,d6:d7
     ; shift = (d0 << (40 - 6)) >> (40 - 6) ; Take 6 LSBits
     ; d6.m = Sat32((shift<0) ? d2.m >> -shift : d2.m << shift)
     ; d7.m = Sat32((shift<0) ? d3.m >> -shift : d3.m << shift)
     ; d6.e = 0, d7.e = 0
```

```
lsh.rgt.l d7,d8,d9; shift = (d7 << (40 - 6)) >> (40 - 6) ; Take 6 LSBits
; d9.m = (shift<0) ? (unsigned long)d8.m << -shift : (unsigned long)d8.m >> shift
; d9.e = 0
lsh.lft.2x d5,d8:d9,d8:d9; shift = (d5 << (40 - 7)) >> (40 - 7) ; Take 7 LSBits
; d8 = (shift<0) ? (unsigned long40)d8 >> -shift : (unsigned long40)d8 << shift
; d9 = (shift<0) ? (unsigned long40)d9 >> -shift : (unsigned long40)d8 << shift
```

3.3.3.3 SIMD 64-bit multi-bit shift

Multi-bit shift operation of 64-bit operands in the SC3900FP core. A variety of instructions are included in this group, with the following flavors (not all combinations are implemented):

- Arithmetic shift and logical shift
  - “ash” instruction: arithmetic shift, sign extended on shift right (for example, $\text{Result} = A \gg B$).
  - “lsh” instruction: logical shift, zero extended on shift right (for example, $\text{Result} = A \ll B$).
- Shift direction
  - Right (flag “rgt”) – Shift right (shift left if shift value is negative)
  - Left (flag “lft”) – Shift left (shift right if shift value is negative)
- Saturation
  - Not saturated
- SIMD (how many outputs are calculated by the instruction
  - No SIMD (suffix “.ll”) – Single output value is calculated and written into two registers

Examples:

```
ash.rgt.ll #15,d3:d4,d5:d6; shift = (d0 << (40 - 7)) >> (40 - 7) ; Take 7 LSBits
     ; {d5.m,d6.m} = {d3.m,d4.m} >> 15
     ; d5.e = 0, d6.e = 0
ash.lft.ll d0,d2:d3,d6:d7; shift = (d0 << (40 - 7)) >> (40 - 7) ; Take 7 LSBits
; {d6.m,d7.m} = (shift<0) ? {d2.m:d3.m} >> -shift : {d2.m:d3.m} << shift
     ; d6.e = 0, d7.e = 0
lsh.lft.ll d5,d8:d9,d8:d9; shift = (d5 << (40 - 7)) >> (40 - 7) ; Take 7 LSBits
; {d8.m,d9.m} = (shift<0) ? (unsigned longlong){d8.m,d9.m} >> -shift :
;       (unsigned longlong){d8.m,d9.m} << shift
```

3.3.3.4 40-bit multi-bit rotate

One bit rotate operation of 40-bit operands through the carry in the SC3900FP core. Two instructions are included in this group, with the following flavors (not all combinations are implemented):

- Rotate
  - “ror” instruction: rotate right.
  - “rol” instruction: rotate left

Examples:

```
ror.x d1,d2; d2 = ((d1 >> 1) & 0x7F_FFFF_FFFF) | (SR.C << 39)
     ; SR.C = d1 (take only the LSBit)
rol.x d6,d9; d9 = ((d6 << 1) & 0xFF_FFFF_FFFE) | SR.C
     ; SR.C = (d6 >> 39) (take only the MSBit)
```

3.3.4 Compare operations

Each DMU within the DALU contains two 32-bit/40-bit compare units and four 16-bit/20-bit compare units. A single compare can be executed and update one predicate bit (Pn) or a predicate bit couple (Pm:Pn). Two to four SIMD compares can be executed in parallel and update Dn register.

NOTE

> Unlike multiplication, there is no difference between integer and fraction; since all are supported, these terms do not appear in the instruction's description.

There are many flavors of multi-bit shift instructions implemented in the DMU. The main groups are described in the following sections.

3.3.4.1 Compare into predicate bit

A single compare of 16-bit, 32-bit or 40-bit into a predicate bit in the SC3900FP core. Compare can write a single result into a single predicate Pn, or write the result and opposite result into predicate couple Pm:Pn. Few DMU instructions can write to the same single predicate bit, resulting in "OR" Between the results of each DMU. Variety of instructions are included in this group, with the following flavors (not all combinations are implemented):

- Compare instruction
  - "cmp" instruction: Compare values.
- Compare operation
  - Equal (flag "eq") – Two values (two registers or a register and immediate value) have equal value
  - Not-equal (flag "ne") – Two values (two registers or a register and immediate) do not have equal value
  - Greater-than (flag "gt") – First value (register) has a value greater than the second value (either register or immediate value)
  - Greater-equal (flag "ge") – First value (register) has a value greater than or equal to the second value (register)
  - Less-equal (flag "le") – First value (register) has a value less than or equal to the second value (immediate value)

  Note that other required compares such as first register less than or equal second register can be implemented with the above flavors by manipulation of the registers order or immediate value.
- Signed / Unsigned
  - Signed – Compare two signed numbers, for example 0x0035 (53) is positive and thus greater than 0xFF00 (–256), which is negative.
  - Unsigned – Compare two unsigned numbers, for example 0x0035 (53) is smaller and thus less than 0xFF00 (65280).
- Compare Width
  - Word (suffix ".w") – Compare low word (16-bit) of a register

- — Long (suffix “.l”) – Compare long (32-bit) of a register
- — A whole register (suffix “.x”) – Compare a whole register (40-bit)

- Compare Target
  - — Single predicate (“Pn”) – Result is written to the pointed predicate bit Pn (set Pn if condition is true and clear it if condition is false).
  - — Predicate couple (“Pm:Pn”) – Result is written to the first pointed predicate bit Pm (set Pm if condition is true and clear it if condition is false), and inverse result is written to the second pointed predicate bit Pn (clear Pn if condition is true and set it if condition is false).

Examples:

```
cmp.eq.x d0,d1,p2; p2 = (d1 == d2)
cmp.gt.x d3,d7,p4:p5; p4 = (d7 > d3) ; p5 = !(d7 > d3)
cmp.ne.l d9,d0,p3; p3 = (d0 != d9)
cmp.le.u.l d5,d6,p1; p1 = ((unsigned long)d6.m <= (unsigned long)d5.m)
cmp.ge.w d3.l,d5.l,p2; p2 = (d5.l >= d3.l)
cmp.le.l #19,d4,p4; p4 = (d4.m <= 19)
```

3.3.4.2 Single/SIMD compare into DALU register

A single or SIMD compare of 16-bit, 20-bit or 40-bit into a DALU register in the SC3900FP core. Compare result is written into a DALU register (or part of it) at the same width as the compare. True value is all ones (0xFFFF for 16-bit, 0xF_FFFF for 32-bit and 0xFF_FFFF_FFFF for 40-bit) and false value is all zeros (0x0000 for 16-bit, 0x0_0000 for 32-bit and 0x00_0000_0000 for 40-bit). Signed compare only is supported. A variety of instructions are included in this group, with the following flavors (not all combinations are implemented):

- Compare instruction
  - — “cmpd” instruction: Compare values into a Dn register.
- Compare operation
  - — Equal (flag “eq”) – Two values (two registers or a register and immediate value) have equal value
  - — Not-equal (flag “ne”) – Two values (two registers or a register and immediate) do not have equal value
  - — Greater-than (flag “gt”) – First value (register) has a value greater than the second value (either register or immediate value)
  - — Greater-equal (flag “ge”) – First value (register) has a value greater than or equal to the second value (either register or immediate value)
  - — Less-than (flag “lt”) – First value (register) has a value less than the second value (immediate value)
  - — Less-equal (flag “le”) – First value (register) has a value less than or equal to the second value (immediate value)

  Note that immediate values are supported only for 40-bit compare (immediate value is signed 32-bit).
- Compare Width

- Word (suffix “.2w” or “.4w”) – Compare 16-bit word of a register
- 20-bit word (suffix “.2t” or “.4t”) – Compare 20-bit word of a register
- A whole register (suffix “.x” or “.2x”) – Compare a whole register (40-bit)

- SIMD (how many outputs are calculated by the instruction
  - No SIMD (suffix “.x”) – Single output value is calculated and written into a single register
  - SIMD2 (suffix “.2x”) – Two output values calculated and written into two registers
  - SIMD2 (suffix “.2w” or “.2t”) – Two output values calculated and written into a single register
  - SIMD4 (suffix “.4w” or “.4t”) – Four output values calculated and written into two registers

Examples:

```
cmpd.ne.x d0,d1,d2; d2 = (d1 == d2) ? 0xFF_FFFF_FFFF : 0x00_0000_0000
cmpd.gt.2x d3:d4,d7:d8,d5:d6; d5 = (d7 > d3) ? 0xFF_FFFF_FFFF : 0x00_0000_0000
      ; d6 = (d8 > d4) ? 0xFF_FFFF_FFFF : 0x00_0000_0000
cmpd.le.x #0x1234_5678,d1,d2; d2 = (d1 <= 0x1234_5678) ? 0xFF_FFFF_FFFF : 0x00_0000_0000
cmp.ne.2w d9,d0,d3; d3.wh = (d0.h != d9.h) ? 0xF_FFFF : 0x0_0000
      ; d3.wl = (d0.l != d9.l) ? 0xF_FFFF : 0x0_0000
cmp.gt.2w d0:d1,d2:d3,d4:d5; d4.wh = (d2.wh != d0.wh) ? 0xF_FFFF : 0x0_0000
      ; d4.wl = (d2.wl != d0.wl) ? 0xF_FFFF : 0x0_0000
      ; d5.wh = (d3.wh != d1.wh) ? 0xF_FFFF : 0x0_0000
      ; d5.wl = (d3.wl != d1.wl) ? 0xF_FFFF : 0x0_0000
```

3.3.5 Floating point

Each DMU has two single precision floating point units, each of which is capable of executing two floating point multiplication, two floating point additions, two floating point fused multiply-add operation (floating point MAC), compare operations and more.

The main groups of floating points instructions flavors implemented in the DMU are described in the following sections.

3.3.5.1 Single/SIMD floating point multiplication

Single precision floating point multiplication operation of the SC3900FP core. Multiply, fused multiply add, fused multiply sub and fused multiply sod. A variety of instructions are included in this group, with the following flavors (not all combinations are implemented):

- MPY and MAC
  - “fmpy” instruction: Multiplication, without additive value (for example $\mathrm{Result} = \pm A \times B$).
  - “fmadd”, “fmsub” and “fmsod” instructions: Multiplication, with additive value (for example, $\mathrm{Result} = C \pm A \times B$).
- Add and sub flavors of FMSOD
  - Add (flag “a”) – This result (one of two or one of four) is calculated as addition
  - subtract (flag “s”) – This result (one of two or one of four) is calculated as Subtraction
- Cross flavors of FMSOD and FMPY
  - No cross (flag “i”) – SIMD vector addition/Subtraction
  - Cross (flag “x”) – First and second inputs are swapped

- Addition/Subtraction flavors of FMPY
  - Add both multiplication (flag “pp”)
  - Add first multiplication and subtract second multiplication (flag “pn”)
  - subtract first multiplication and add second multiplication (flag “np”)
  - subtract both multiplication (flag “nn”)
- Special flags for FMADD (see Appendix C, “Instruction Set” for details)
  - Inverse Square Root (flag “invsq”) – Special version for inverse square root implementation
  - Log 2 (flag”log2”) – Special version for log base two implementation
  - Reciprocal (flag “recip”) – Special version for reciprocal (division) implementation
- SIMD (how many outputs are calculated by the instruction
  - No SIMD (suffix “.sp”) – Single output value is calculated and written into a single register
  - SIMD2 (suffix “.2sp”) – Two output values calculated and written into two registers
  - SIMD2 with shard input (suffix “.2sp”) – One of the inputs is shared

Examples (all variables are single precision float, extension of the output is written as zero):

```
fmpy.sp d0,d3,d6; d6 = d0 * d3
fmpy.pp.2sp d2:d3,d4:d5,d6:d7; d6 = d2 * d4
      ; d7 = d3 * d5
fmpy.2sp d7,d8:d9,d0:d1; d0 = d7 * d8
      ; d1 = d7 * d9
fmpy.npx.2sp d8,d2:d3,d5:d6; d5 = -d8 * d3
      ; d6 =  d8 * d2
fmadd.sp d5,d7,d0; d0 = d0 + d5 * d7
fmadd.2sp d6:d7,d8:d9,d1:d2; d1 = d1 + d6 * d8
      ; d2 = d2 + d7 * d9
fmsub.sp d5,d7,d0; d0 = d0 - d5 * d7
fmsub.2sp d6:d7,d8:d9,d1:d2; d1 = d1 - d6 * d8
      ; d2 = d2 - d7 * d9
fmsod.asx.2sp d7,d8:d9,d2:d3; d2 = d2 + d7 * d9
      ; d3 = d3 - d7 * d8
```

3.3.5.2 Single/SIMD floating point addition

Single precision floating point addition operation of the SC3900FP core. A variety of instructions are included in this group, with the following flavors (not all combinations are implemented):

- Add and subtract
  - “fadd” instruction: Addition (for example $\mathrm{Result} = A + B$).
  - “fsub” instruction: Subtraction (for example $\mathrm{Result} = A - B$).
  - “faddsub” and “fsubadd”: Addition and Subtraction (for example $\mathrm{Result} = A \pm B$).
- Cross flavors
  - No cross – SIMD vector addition/Subtraction
  - Cross (flag “x”) – First and second inputs are swapped
- SIMD (how many outputs are calculated by the instruction
  - No SIMD (suffix “.sp”) – Single output value is calculated and written into a single register

— SIMD2 (suffix “.2sp”) – Two output values calculated and written into two registers

Examples (all variables are single precision float, extension of the output is written as zero):

```
fadd.sp d0,d3,d6; d6 = d3 + d0
fadd.2sp d2:d3,d4:d5,d6:d7; d6 = d4 + d2
     ; d7 = d5 + d3
fadd.x.2sp d6:d7,d8:d9,d0:d1; d0 = d9 + d6
     ; d1 = d8 + d7
faddsub.x.2sp d8:d9,d2:d3,d5:d6; d5 = d2 + d9
     ; d6 = d3 - d8
fsub.sp d0,d3,d6; d6 = d3 - d0
fsub.2sp d2:d3,d4:d5,d6:d7; d6 = d4 - d2
     ; d7 = d5 - d3
fsub.x.2sp d6:d7,d8:d9,d0:d1; d0 = d9 - d6
     ; d1 = d8 - d7
fsubadd.x.2sp d8:d9,d2:d3,d5:d6; d5 = d2 - d9
     ; d6 = d3 + d8
```

3.3.5.3 Floating point corner cases

The following tables describe the corner cases definition for floating point operations.

The symbol ‘x’ means don’t care.

The add operation is defined as $Dn = Da + Db$. Corner case rules for Da and Db are commutative.

Table 3-4. Floating point add operation special cases

Da	Db	Dn	Remarks
SNaN	x	QNaN output	Invalid flag is set.
QNaN	x (x ≠ SNaN)	QNaN output	—
– ∞	+ ∞	QNaN output	Invalid flag is set.
± ∞	x (x ≠ NaN or opposite sign Infinity)	± ∞	—
value	– value	+ 0	—
+0	– 0	+ 0	—
– 0	– 0	– 0	—

The subtract operation is defined as $Dn = Db - Da$.

Table 3-5. Floating point subtract operation special cases

Da	Db	Dn	Remarks
SNaN	x	QNaN output	Invalid flag is set.
x	SNaN	QNaN output	Invalid flag is set.
QNaN	x (x ≠ SNaN)	QNaN output	—
x (x ≠ SNaN)	QNaN	QNaN output	—
– ∞	– ∞	QNaN output	Invalid flag is set.
+ ∞	+ ∞	QNaN output	Invalid flag is set.

Table 3-5. Floating point subtract operation special cases (continued)

Da	Db	Dn	Remarks
x (x ≠ NaN or same sign Infinity)	± ∞	± ∞	—
value	value	+0	—
+0	– 0	– 0	—
–0	+0	+0	—

The multiply operation is defined as $Dn = Da \times Db$. Corner case rules for Da and Db are commutative.

Table 3-6. Floating point multiply operation special cases

Da	Db	Dn	Remarks
SNaN	x	QNaN output	Invalid flag is set.
QNaN	x (x ≠ SNaN)	QNaN output	—
± ∞	x (x ≠ 0 or denormalized)	± ∞	—
± ∞	± 0	QNaN output	Invalid flag is set.
± 0	x (x ≠ ∞ or NaN)	± 0	—

The multiply-add operation is defined as $Dn = Dn + Da \times Db$. Corner case rules for Da and Db are commutative.

Table 3-7. Floating point multiply-add operation special cases

Da	Db	Dn (in)	Dn (out)	Remarks
SNaN	x	x	QNaN output	Invalid flag is set.
x	x	SNaN	QNaN output	Invalid flag is set.
QNaN	x (x ≠ SNaN)	x (x ≠ SNaN)	QNaN output	—
x (x ≠ SNaN)	x (x ≠ SNaN)	QNaN	QNaN output	—
0	± ∞	x	QNaN output	invalid flag is set.
x (x ≠ 0 or denormalized)	± ∞	–(± ∞)	QNaN output	Invalid flag is set. **Note:** Db and Dn are infinities of opposite sign
x (x ≠ 0 or denormalized)	± ∞	± ∞	± ∞	**Note:** Db and Dn are infinities of the same sign.
Da*Db overflowed to + ∞		+ ∞	+ ∞	—
Da*Db overflowed to – ∞		+ ∞	QNaN output	Invalid flag is set.
Da*Db overflowed to + ∞		– ∞	QNaN output	Invalid flag is set.
Da*Db overflowed to – ∞		– ∞	– ∞	—

The multiply-subtract operation is defined as $Dn = Dn - Da \times Db$. Corner case rules for Da and Db are commutative.

Table 3-8. Floating point multiply-subtract operation special cases

Da	Db	Dn (in)	Dn (out)	Remarks
SNaN	x	x	QNaN output	Invalid flag is set.
x	x	SNaN	QNaN output	Invalid flag is set.
QNaN	x (x ≠ SNaN)	x (x ≠ SNaN)	QNaN output	—
x (x ≠ SNaN)	x (x ≠ SNaN)	QNaN	QNaN output	—
0	± ∞	x	QNaN output	Invalid flag is set.
x (x ≠ 0 or denormalized)	± ∞	–(± ∞)	QNaN output	Invalid flag is set. **Note:** Db and Dn are infinities of opposite sign.
x (x ≠ 0 or denormalized)	± ∞	± ∞	± ∞	**Note:** Db and Dn are infinities of the same sign.
Da*Db overflowed to + ∞		+ ∞	QNaN output	Invalid flag is set.
Da*Db overflowed to –∞		+ ∞	+ ∞	—
Da*Db overflowed to + ∞		– ∞	– ∞	—
Da*Db overflowed to –∞		– ∞	QNaN output	Invalid flag is set.

Compare operations are defined as Pn = Db cond Da. In the first general table and in compare-equal Corner case rules for Da and Db are commutative.

Note that when using a two predicate output, the second predicate is written with the inverse value of the first one.

Table 3-9. Floating Point Compare Operation Special Cases

Da	Db	Pn	Remarks
x	NaN	False	Invalid flag is set.
NaN	NaN	False	Invalid flag is set.

Table 3-10. Floating Point Compare-Equal (Db=Da) Operation Special Cases

Da	Db	Pn	Remarks
+ ∞	+ ∞	True	—
– ∞	– ∞	True	—
+0	–0	True	—
Denormalized	0	True	—

Table 3-11. Floating Point Compare-Not-Equal (Db!=Da) Operation Special Cases

Da	Db	Pn	Remarks
+0	–0	False	—
±Denormalized	±0	False	—

Table 3-12. Floating Point Compare-Greater Than (Db>Da) Operation Special Cases

Da	Db	Pn	Remarks
x (x ≠ NaN)	+ ∞	True	—
+/–0	+/–0	False	—
± Denormalized	± Denormalized	False	Denormalized input flag is set.
+/–0	± Denormalized	False	Denormalized input flag is set.

Table 3-13. Floating Point Compare-Greater Equal (Db≥Da) Operation Special Cases

Da	Db	Pn	Remarks
x (x ≠ NaN)	+ ∞	True	—
± 0	± 0	True	—
± Denormalized	± Denormalized	True	Denormalized input flag is set.
± 0	± Denormalized	True	Denormalized input flag is set.
+ ∞	+ ∞	True	—
– ∞	– ∞	True	—

Table 3-14. Floating Point Compare-Bound (-Db≤Da≤Db) Operation Special Cases

Da	Db	Pn	Remarks
x	any negative (except NaN, ± 0, ± denormalized)	False	—
x (x ≠ NaN)	+ ∞	True	—
± ∞	+ ∞	True	—
± 0	+ 0	True	—
+/–Denormalized	+Denormalized	True	Denormalized input flag is set.
± 0	+Denormalized	True	Denormalized input flag is set.

Absolute operation is defined as $Dn = |Da|$:

Table 3-15. Floating Point Absolute Operation Special Cases

Da	Dn	Remarks
SNaN	QNaN output	Invalid flag is set.
QNaN	QNaN output	—

Table 3-15. Floating Point Absolute Operation Special Cases (continued)

Da	Dn	Remarks
± ∞	+ ∞	—
–0	+0	—
± Denormalized	+0	Denormalized input flag is set.

Floating point to fixed point conversion $\mathrm{Dn} = (\mathrm{finxed})(\mathrm{Da} \times 2^{\mathrm{Scale}})$:

Table 3-16. Floating Point to Fixed Point Conversion special Cases

Da	Dn	Remarks
NaN	± Saturated fixed point value	Invalid flag is set.
± ∞	± Saturated fixed point value	—
± 0	+0	—
± Denormalized	+0	Denormalized input flag is set.

Fixed point to floating point conversion $\mathrm{Dn} = (\mathrm{float})(\mathrm{Da} \times 2^{-\mathrm{Scale}})$ Dn = (float) (Da) / 2^SCALE:

Table 3-17. Fixed Point to Floating Point Conversion special cases

Da	Dn	Remarks
0	+0	—
biased Da exponent < –126	± 0	Inexact and underflow flags are set.
biased Da exponent > 127	± ∞	Inexact and overflow flags are set.

Table 3-18. Floating Point Reciprocal Operation Special Cases

Da	Dm	Dn	Remarks
± Denormalized	+0	± ∞	Division by zero and denormalized input flags are set.
± 0	+0	± ∞	Division by zero flag is set.
± ∞	± 0	± 0	—
SNaN	QNaN output	QNaN output	Invalid flag is set.
QNaN	QNaN output	QNaN output	—

Table 3-19. Floating Point Reciprocal Square Root Operation Special Cases

Da	Dm	Dn	Remarks
Any negative, but -0 or -denormalized	QNaN output	QNaN output	Invalid flag is set.
± Denormalized	+0	+ ∞	Division by zero and denormalized input flags are set.
± 0	+0	+ ∞	Division by zero flag is set.
+ ∞	+0	+0	—

Table 3-19. Floating Point Reciprocal Square Root Operation Special Cases (continued)

Da	Dm	Dn	Remarks
SNaN	QNaN output	QNaN output	Invalid flag is set.
QNaN	QNaN output	QNaN output	—

Table 3-20. Floating point LOG2 operation special cases

Da	Dm	Dn	Remarks
Any negative	QNaN output	QNaN output	Invalid flag is set.
+ Denormalized	+0	– ∞	Division by zero and denormalized input flags are set.
+ 0	+0	– ∞	Division by zero flag is set.
+ ∞	+0	+ ∞	—
SNaN	QNaN output	QNaN output	Invalid flag is set.
QNaN	QNaN output	QNaN output	—

3.3.6 Application-specific instructions

3.3.6.1 Maximum and minimum find

In many cases, fast search of array is needed to return the maximum or minimum value as well as the pointer to the location of this value. If there is more than one maximum or minimum value (winner), index points to the first one.

The instruction holds in one register the winner value (as a 40-bit value), and in another register the pointer to this winner and the pointer to the current search input. Each time the instruction is activated, four new inputs are read from registers and compared to each other and to the previous winner. If one of newly read data is the new winner, both winner value and index are updated. The current pointer is updated in any case. At the end of the search routine the winner value register holds the maximum or minimum value of the vector as required while the winner index is kept in the other register.

The following diagram shows the data flow for find instructions.

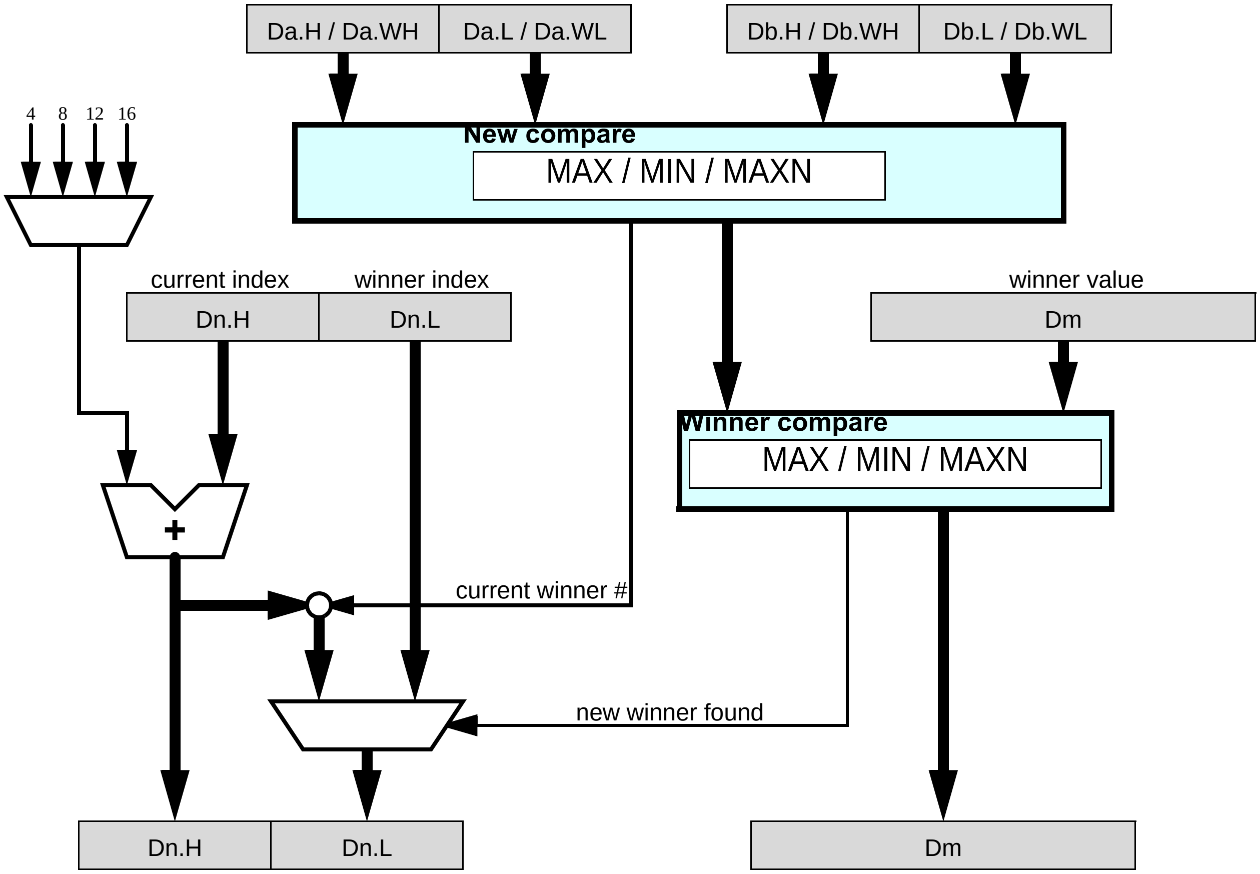


Figure 3-25. Data flow for find instructions

Single maximum and minimum search operation of the SC3900FP core, enables acceleration of such searches. Following instructions are included in this group, with the following flavors:

- Find instruction
  - — “fnd” instruction: Find maximum or minimum value and index.
- Maximum and minimum flavors
  - — Maximum (flag “max”) – Find the maximum value
  - — Minimum (flag “min”) – Find the minimum value
  - — Absolute maximum (flag “maxm”) – Find the absolute maximum value. The winner is the value with the maximum absolute value. If two highest absolute value are with opposite signs, take the positive one
- SIMD (how many outputs are calculated by the instruction
  - — SIMD4 (suffix “.4t” or “.4w”) – Compare four new input values against previous “winner”, and if needed replace the winner and update the pointer to it.

Examples:

```
find.max.4t #2,d0,d3,d6:d7; tmp_winner = max(d0.wh,d0.wl,d3.wh,d3.wl)
      ; tmp_index = index_of_tmp_winner (2-bit number: 0-3)
      ; if (tmp_winner > d6) {
      ;    d6 = tmp_winer- Set new winner
      ;    d7.l = {(d7.h>>2),tmp_index}- Set new index
      ; }
      ; d7.h = d7.h + 12 (#0=4,#1=8,#2=12,#3=16) update pointer
find.min.4w #0,d5,d6,d0:d1; tmp_winner = max(d5.h,d5.l,d6.h,d6.l)
      ; tmp_index = index_of_tmp_winner (2-bit number: 0-3)
      ; if (tmp_winner < d0) {
      ;    d0 = tmp_winer- Set new winner
      ;    d1.l = {(d1.h>>2),tmp_index}- Set new index
      ; }
      ; d1.h = d1.h + 4 (#0=4,#1=8,#2=12,#3=16) update pointer
```

3.3.6.2 FFT and DFT

Special instructions accelerate FFT and DFT calculation for Radixes 2, 3, 4 and 5. The instruction calculate the butterfly result of the required radix in either single phase or two phases, according to the case. The following instructions are included in this group, with the following flavors (not all flavors implemented):

- FFT and DFT instruction
  - “fft” instruction: FFT or DFT butterfly calculation.
- Radix
  - Radix 2 (flag “r2”) – Radix 2 calculation (first stage calculation only)
  - Radix 3 (flag “r3”) – Instructions for two phases implemented
  - Radix 4 (flag “r4”) – Full butterfly in one instruction. Different instructions for the first stage and later stages
  - Radix 5 (flag “r5”) – Instructions for two phases implemented
- Stage
  - Stage 1 (flag “s1”) – Radix 2/4 calculation of stage 1 (all twiddle factors are “1”)
  - Stage N (flag “sn”) – Radix 4 calculation of stage N
- Phase
  - Phase 1 (flag “p1”) – Radix 3/5 phase 1 (twiddle factor multiplication)
  - Phase 2 (flag “p2”) – Radix 3/5 phase 2 (radix 3/5 constant multiplication)
- Saturation
  - Not saturated – Most of FFT instruction need no saturation
  - Saturated (flag “s”) – Few first stage instruction need saturation, since its output is an input for stage two multiplication
- Inverse FFT/DFT
  - Normal FFT/DFT– Do FFT/DFT math
  - Inverse FFT/DFT or IFFT/IDFT (flag “i”) – All fft instruction (excluding radix 2 symmetric function) has an IFFT/IDFT flavor
- SIMD (how many outputs are calculated by the instruction

- — SIMD4 (suffix “.4t” or “.4w”) – Single DMU generating four results in two registers
- — SIMD8 (suffix “.8t”) – Two DMU generating wight results in four registers, encoded in 64-bit

Examples:

```
fft.r2.s1.4t d1,d0,d16:d27; d0 holds IA, D1 holds IB,
     ; write OA to d16 and OB to d27
fft.r3.sp1.4t d0:d1,d48:d49,d31:d40; d0:d1 holds IB:IC, d48:d49 holds twiddle value
     ; d31:d40 holds mid result for phase 2
fft.r3.p2.8t d31:d32,d30,d12,d40:d41,d30,d13,d31:d40,d32:d41
; A two DMU 64-bit instruction.
; d31:d32 and d40:d41 are inputs from phase 1, d30 is the radix 3 constant and d12,d13
; holds the IA value, calculating OB and OC for two butterflies into d31:d40 and d32:d41
fft.r4.sn.8t d4,d5:d6,d38:d39,d4,d6:d7,d41,d17:d28,d33:d44
; Radix 4 two DMU instruction 64-bit instruction,
; calculating a full radix 4 butterfly in a single instruction
; d4,d5,d6,d7 hold IA, IB, IC and ID input of a butterfly
; d38, d39,d41 hold three twiddle values AB, AC and WD
; Four output OA,OB,OC,OD are written to d17:d28 andd33:d44
```

3.3.6.3 Floating Point Arithmetic Functions

The DALU supports special look-up table instructions which aid in the implementation of the following functions:

- 1/x (reciprocal)
- $1/\sqrt{x}$ (inverse square-root)
- Log_2 (base 2 logarithm)

Each look-up table instruction generates two co-efficients as an output. Using linear interpolation on these outputs yields an approximately 16-bit precision result. Note there are specific fused multiply-add instructions that should be used in order for the result to be correct. Dedicated ‘preparation’ instructions should be used as well.

That result can be followed by a Newton-Raphson iteration in order to obtain a higher precision result (note that Newton-Raphson is not applicable for Log2).

The following are example for a proper usage of these functions.

3.3.6.3.1 Floating Point Reciprocal (FRECIP)

Example:

```
frecip d0,d1:d2            ; d0 is the input
frecip.pre d0,d3           ; preparation instruction
fmadd.recip.sp d3,d1,d2    ; recip-specific linear interpolation
fmsub.sp d0,d2,d4
fmpy.sp d2,d4,d5           ; final Newton-Raphson result d5 = 1/d0
```

3.3.6.3.2 Floating Point Inverse Square-Root (FINVSQRT)

Example:

```
finvsqrt.2sp d0,d1:d2      ; d0 is the input
finvsqrt.pre d0,d3          ; preparation instruction
fmadd.invsq.sp d3,d1,d2     ; invsqrt-specific linear interpolation
fmpy.sp d2,d2,d1
fmpy.sp d2,d4,d6
fmsub.sp d0,d1,d5
fmpy.sp d6,d5,d5            ; final Newton-Raphson result d5 = 1/SQRT(d0)
```

3.3.6.3.3 Floating Point Base 2 Logarithm (FLOG2)

Example:

```
flog2 d0,d1:d2              ; d0 is the input
flog2.pre1 d0,d3            ; preparation instruction
flog2.pre2 d0,d4            ; preparation instruction
fmadd.log2.sp d0,d3,d2      ; log2-specific linear interpolation
fix2flt.l.sp #0,d4,d5
fadd.sp d2,d5,d5            ; result d5 = LOG2(d0)
```

3.4 DALU configuration and flags

The DALU has dedicated mode bits that affect the DALU behavior and dedicated flags that are set by DALU instructions according to the execution all implemented in the Status Register (SR). For details about the SR, see Section 2.1.5.1, “Status Register (SR).” The following sections describe the functionality of the DALU with the SR bits.

3.4.1 Arithmetic modes

In previous DALU generations, different arithmetic modes such as integer versus fraction, saturation and rounding were controlled almost exclusively by using mode bits in the SR. Setting these mode bits changes the mathematical behavior of legacy instructions.

Two arithmetic modes control 32-bit/40-bit mathematics, according to the value of the saturation mode bit in the Status Register (SR.SM):

- SR.SM=0 – No saturation, 40-bit mathematics
- SR.SM=1 – Saturation, 40-bit output value saturated to 0x00_7FFF_FFFF or 0xFF_8000_0000 if cannot be represented in 32-bit.

Default mode in SC3900FP (as in previous generations) is SR.SM set (although reset value is cleared).

Three arithmetic modes control 16-bit/20-bit mathematics, according to the value of Saturation Mode 2 bit and Word 20 bit in the Status Register (SR.SM2 and SR.W20):

- SR.SM2=0,SR.W20=0 – No saturation, 16-bit mathematics
- SR.SM2=1,SR.W20=0 – Saturation, 16-bit output value saturated to 0x7FFF or 0x8000 if cannot be represented in 16-bit. Extension nibble always written as zero
- SR.SM2=0,SR.W20=1 – No saturation, 20-bit mathematics

Default mode in S3900 (as in previous generations) is both SR.SM2 and SR.W20 cleared.

In SC3900FP, access to different mathematical modes is done using different instruction flavors (it is more flexible since it allows mix of variables with different mathematical modes in the same code with no penalty). Legacy instructions preserve the change of behavior according to the arithmetic mode. These instruction are divided into two groups: 32-bit/40-bit instructions are marked as legacy instruction (marked with flag "leg" in the instruction name), and 16-bit/20-bit instructions should be used in new code, assuming default arithmetic mode setting (16-bit without saturation).

In the case of "pure" legacy instructions, there are new instructions that are not affected by the arithmetic mode bits, and have static setting of their mathematical mode. For example, legacy instruction `ADD.LEG.X Da,Db,Dn` is saturating the result to 32-bit value if SR.SM bit is set, and not saturating it if the bit is cleared. New instructions `ADD.S.X Da,Db,Dn` always saturate the result (same as `ADD.LEG.X Da,Db,Dn` with SR.SM bit set) and `ADD.X Da,Db,Dn` never saturate the result (same as `ADD.LEG.X Da,Db,Dn` with SR.SM bit cleared).

Instruction that depend on SR.SM2 and SR.W20, are expected to operate in new code assuming that SR.SM2 and SR.W20 are cleared (default mode). For example `ASH.LFT.2W Da,Db,Dn` instruction shift 20-bit values if W20 set, shift and saturate 16-bit values if SM2 is set, and shift 16-bit values if both are cleared. In new SC3900FP code `ASH.LFT.2W Da,Db,Dn` should be used to shift 16-bit values with no saturation (as implied by it's name). New `ASH.LFT.S.2W Da,Db,Dn` should be used to implement 16-bit shift with saturation and new `ASH.LFT.2T Da,Db,Dn` should be used to implement 20-bit shift.

NOTE

According to the SC3850 definition, the extension nibble of 20-bit variables was unused as input and undefined as output when SR.W20 bit was cleared. Since the low 16-bit of the result of 20-bit addition is the same as 16-bit addition result, legacy 16-bit/20-bit add/mac instruction when both SR.SM2 and SR.W20 are cleared is the same as the behavior when SR.SM2 is cleared and SR.W20 is set. For example `SOD.AAII.2T Da,Db,Dn` generate the 20-bit result (compatible with 16-bit result), when SR.SM2=0,SR.W20=0 and SR.SM2=0,SR.W20=1 (and thus it's name suggest its default behavior of 20-bit add), while new instruction `SOD.AAII.S.2W Da,Db,Dn` generate saturated 16-bit result.

3.4.2 Configuring the operation of store with scale, round, and saturate instructions

The SC3900FP is capable of scale, round and saturate numbers, according to two fields in the Status Register (SR). The two fields are:

- SR.SCM—Scaling size
  - 000—No scaling
  - 001—Scale down by one
  - 010—Scale up by one
  - 011—Scale down by two
  - 100—Scale down by three
- SR,RM—Rounding mode

— 0—Convergent rounding
— 1—Two's complement rounding

The fields are used in the SC3900FP to configure the operation of store with scale, round and saturate instructions. For details, see Section 3.2.2.3, "Scale, round and saturate store instructions."

These bits also affect few DALU legacy instructions including: `RND`, `MPY.RLEG.X` and `MAC.RLEG.X`. Usage of these instruction enable the ability to scale and round directly into a register. Note that scale down by two and three are not supported by DALU instructions.

NOTE

> Enabling the 32-bit arithmetic saturation mode (set SR.SM bit) disables scaling for non-packed data memory transaction (`ST.SRS.*`), and enabling the 16-bit arithmetic saturation mode (set SR.SM2 bit) disables scaling for packed data memory transaction (`ST2.SRS.*`).

3.4.3 Arithmetic flags

3.4.3.1 Carry bit flag

The DALU generates a single carry bit which is reflected into the SR register carry flag.

Carry bit can be affecting and/or affected by the following instructions:

- ADDC

  These instruction update and/or use 32-bit (suffix ".l") or 40-bit (suffix ".x") carry.

  Carry definition for these instruction is:

  — Input carry

  If SR.C is set, add one to the result

  — 32-bit output carry

  If sum of 32 LSBits of the two additives cannot be represented in a 32-bit number set SR.C bit, else clear SR.C bit.

  — 40-bit output carry

  If sum of the two 40-bit additives cannot be represented in a 40-bit number set SR.C bit, else clear SR.C bit.

- SUBC

  These instruction update and/or use 32-bit (suffix ".l") or 40-bit (suffix ".x") borrow.

  Borrow definition for these instruction is:

  — Input borrow

  If SR.C is set, subtract one from the result

  — 32-bit output borrow

  If difference of 32 LSBits of the two additives cannot be represented in a 32-bit number set SR.C bit, else clear SR.C bit.

  — 40-bit output borrow

If difference of the two 40-bit additives cannot be represented in a 40-bit number set SR.C bit, else clear SR.C bit.

- ROR and ROL
  - ROL - MSBit if the input written to SR.C and SR.C written to LSBit of the result
  - ROR - LSBit if the input written to SR.C and SR.C written to MSBit of the result
- ASH and LSH

  Last shifted out bit is written to the SR.C

Only one instruction that write SR.C bit is allowed in a VLES.

3.4.3.2 Saturation flag

The saturation flag is a sticky bit that reside in the Status Register (SR.SAT), by a DALU instruction that saturates at least one of it's result. It can be explicitly cleared by SW using an IPU instruction.

The saturation flag, SR.SAT is set in case any of the four DMU results was saturated, on either way of SIMD operations.

Saturation may occur in one of several cases:

- One of the results of 16-bit saturating instruction (“s” flag and “w” suffix), cannot be represented in 16-bit and thus saturated.
- Result or one of the results of 32-bit/40-bit saturating instruction (“s” flag and “x” or “l” suffix), cannot be represented in 32-bit and thus saturated.
- One of the results of 20-bit saturating instruction (“s” flag and “t” suffix), cannot be represented in 20-bit and thus saturated (barrel shift only instructions).
- SR.SM bit is set and the result of a legacy instruction affected by it cannot be represented in 32-bit and thus saturated.
- SR.SM2 bit is set, SR.W20 bit is cleared and the result of a legacy instruction affected by them cannot be represented in 16-bit and thus saturated.

3.4.3.3 Overflow flag

The overflow flag is a sticky bit that reside in the Status Register (SR.OVF), by a DALU instruction that at least one of it's result cannot be represented in the output format. It can be explicitly cleared by SW using an IPU instruction.

The overflow flag, SR.OVF is set in case any of the four DMU results was overflowed, on either way of SIMD operations.

Overflow set may occur in one of several cases:

- One of the results of 20-bit non-saturating instruction (“t” suffix), cannot be represented in 20-bit.
- One of the results of 16-bit non-saturating instruction (“w” suffix), cannot be represented in 16-bit.
- Result or one of the results of 40-bit non-saturating instruction (“x” suffix), cannot be represented in 40-bit

- Result or one of the results of 32-bit non-saturating instruction (“l” suffix), cannot be represented in 32-bit
- SR.SM bit is set and the result of a legacy instruction affected by it cannot be represented in 32-bit and thus saturated (note that legacy mode dependant saturation assert the SR.OVF flag)
- SR.SM2 bit is set, SR.W20 bit is cleared and the result of a legacy instruction affected by them cannot be represented in 16-bit and thus saturated (note that legacy mode dependant saturation assert the SR.OVF flag)

The SR.OVF behavior retains legacy behavior of EMR.DOVF for legacy instructions, indicating for such instructions that either saturation or overflow occurred.

Note that, in the case of saturation, a legacy mode-dependent saturating instruction sets both SR.OVF and SR.SAT.

3.4.3.4 Floating point flags

The floating point related flags are located in the GCR register. These flags are sticky and are not affected by false-predicated instructions.

The following flags are implemented:

3.4.3.4.1 Float Inexact Flag (FINEX)

This flag is set in case a result differs from the result that would have been computed had the exponent range and precision been unbounded:

- Rounding is effectively performed
- Denormalized result was flushed to zero in output
- Overflow or underflow occurred (in such case the appropriate overflow or underflow flag is set as well)

3.4.3.4.2 Float Invalid Flag (FINVAL)

This flag is set in any of the following cases:

- A signaling NaN (SNaN) is used as source
- Infinity is subtracted from infinity
- Iinfinity is multiplied by zero, or by a denormalized number
- Reciprocal-square-root or LOG2 operation is attempted on a negative operand

3.4.3.4.3 Float Division by Zero Flag (FDIVZ)

This flag is set in case a zero or a denormalized value is used as source for reciprocal or reciprocal-square-root instruction.

Note that a reciprocal-square-root on a negative zero or a negative denormalized would set the invalid flag.

3.4.3.4.4 Float Overflow Flag (FOVER)

This flag is set in case the result of finite inputs overflowed. The output is signed infinity (definition for nearest-even rounding).

3.4.3.4.5 Float Underflow Flag (FUNDR)

This flag is set in case a non-zero result of finite inputs underflowed (less than smallest normalized number). The output is signed zero (definition for nearest-even rounding).

3.4.3.4.6 Float Denormalized input Flag (FDENI)

This flag is set in case a denormalized number is used as input. The value is flushed to the correctly signed zero and computation continues as if the input value was zero.

Chapter 4
Address Generation and Memory Interface

4.1 Address Generation Unit (AGU) overview

The AGU is a group of execution units that performs load and store operations as well as general integer arithmetic and logic operations. The arithmetic operations can be used for effective address calculations or for general integer arithmetic not related to address calculations in particular. The AGU operates in parallel with other core resources to minimize address generation overhead. The AGU supports many features defined to optimize DSP kernels, among them arithmetic modes such as modulo, multiple wrap-around modulo, and reverse-carry. The AGU also has specialized vectored load and store instructions that activate the two parallel data buses (each 256-bits wide), and can perform packing and unpacking of multiple variables during accesses between registers and memory.

The AGU is composed of the following components:

- A general register file (also used to hold address pointers)
- Two identical Load Store Unit (LSUs)
- An Integer Processing Unit (IPU)

The AGU can execute up to three instructions every cycle: one in each of the two LSUs, and one in the IPU. Figure 4-1 shows a block diagram of the AGU. The two buses are illustrated functionally; in the implementation, the data buses (CAD, CBD) are split into read and write directions.

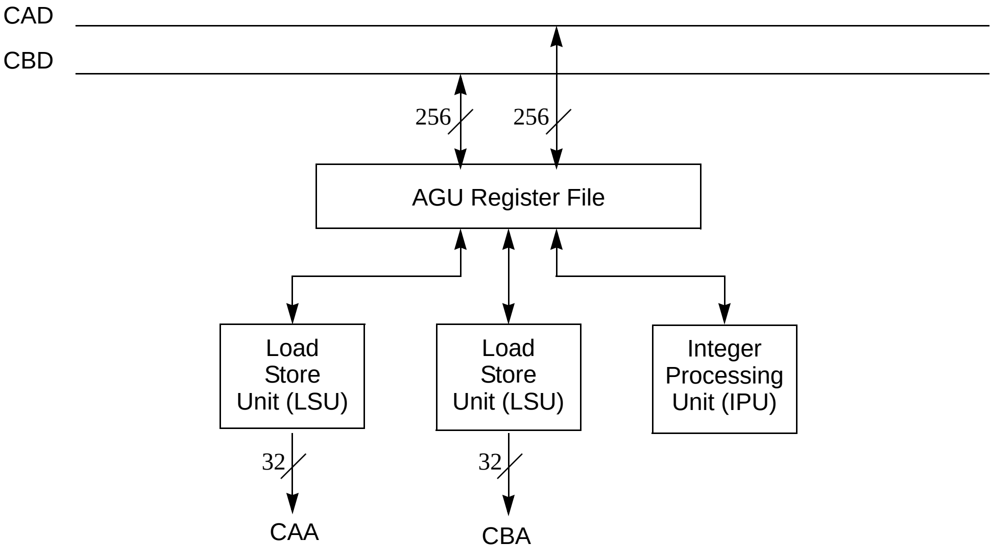


Figure 4-1. AGU block diagram

The full bandwidth of the two 256-bit buses can be used to interface the data register file in the DALU (D0–D63). Although the buses connect also with the AGU register file (R0–R31), there are throughput limitations on the number of address registers that can be written per VLES. For details, see Section 4.6.7, “Register bandwidth limitations on MOVE and memory accesses.”

4.1.1 AGU register file

The AGU register file contains the following registers:

- Thirty two 32-bit general registers and pointers (R0–R31)
- A 32-bit modulo control register (MCTL)
- Three 32-bit stack pointers (TSP, ESP and DSP), only one of which is active at a time, and referenced as “SP” in the source code

Some of the modulo registers are physically aliased to R registers:

- B0–B7 (modulo base registers) are aliased to R8–R15 (same as in SC3850)
- M0–M3 (modulo modifier registers) are aliased to R24–R27 (new aliasing in SC3900FP)

For more information, see Section 2.1.2, “Address registers (R0–R31).”

4.1.2 Load Store Unit

The AGU has two essentially identical Load Store Units (LSUs). These units generate addresses for load/store memory access operations that initiate data transfers between the core and memory. Each LSU can independently generate a memory access per cycle. In addition, they can perform address or integer arithmetic and logical instructions.

The LSUs support the following four types of arithmetic:

- Linear
- Modulo
- Multiple wrap-around modulo
- Reverse-carry

Linear addressing is available for all address registers, while the non-linear arithmetic modes are available only for registers R0–R7. The MCTL register is used to specify, per R register, what type of arithmetic will be used, and using what modulo register. Non-linear arithmetic modes are available for most memory LD/ST instructions, and for a subset of the LSU arithmetic instructions.

The LSU also supports a range of 32-bit integer arithmetic and logical operations on address registers. Supported instructions include the following:

- Arithmetic (Add, Sub, and so on)
- Comparisons and Predicate bit manipulation
- Logical operations
- Multi-bit shifts (with explicit shift count)

4.1.3 Integer Processing Unit

The Integer Processing Unit (IPU) can perform a rich set of general 32-bit integer arithmetic instructions on address registers. The IPU cannot generate memory accesses, but it can generate some non-R register transfers (such as MOVE.L Dn,Rn). Only linear addressing arithmetic is supported in this unit. Most of the arithmetic and logical instructions supported by the LSUs can also be performed by the IPU. These dual LSU-IPU instructions use the same mnemonic. The assembler automatically selects which unit will execute the instruction, and encodes it accordingly. For example, ZXTA.W R0,R1 is supported by both LSU and IPU. If it is grouped with two load/store instructions (which can only be executed by an LSU), the ZXTA.W will be encoded as an IPU instruction. However, if it is grouped with an IPU-only instruction (such as MPY32A), the ZXTA.W instruction will be encoded as an LSU instruction.

On top of instructions shared with the LSU, the IPU supports some unique arithmetic instructions, such as the following:

- 32-bit Integer Multiply (32x32)
- Multi-bit Shift (register based shift count)

The IPU also manages the configuration access to all of the control registers. Thus, for example, TFRA SR,Rn is an IPU instruction.

4.2 Memory addressing

4.2.1 Effective, virtual and physical addresses

The 32-bit address that is calculated by the programer is known as the effective address. The MMU differentiates between the effective addresses of different tasks by including a task ID tag (PID for program accesses, DID for data accesses) as part of the address. This allows the MMU to keep memory descriptors of different tasks enabled concurrently. The effective address + ID is the virtual address, which is the input to address translation in the MMU. When in exception, and for system OS code (SR2.EXP = 1), the core drives a fixed value of 0 as the ID field. The translated address is a 36-bit wide physical address that is used to access the system memory. The physical memory space is unified for program and data; however, the MMU allows to configure different protection, translation and caching attributes to program and data accesses. The 40-bit virtual address is part of the core architecture. Translation and the width of the physical address are part of the MMU architecture; certain aspects of the MMU definition, such as the width of the physical address, may change in other products, see the *SC3900FP FVP Subsystem Reference Manual*.

The types of memory addresses and the translation flow are described in this figure.

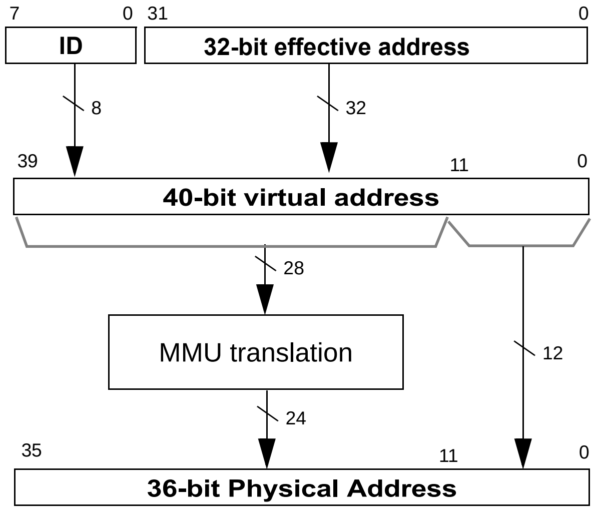


Figure 4-2. Types of memory addresses and address translation flow

Both the DID and PID are configured in different fields in the TID core register, and are generally managed by the OS, see Section 2.1.7.1, "Using the Task ID register (TID)." The separation of task ID to data ID (DID) and Program ID (PID) is intended to allow cache line re-use when several instances of the same program are running, each with its unique data (for example, several similar channels that require the same type of processing). In general, the DID is specific per task while the PID may be shared between a few tasks. For the sake of OS management operations and advanced cache maintenance operations, the core supports an option to override the value that is programmed in DID or in PID and issues accesses or cache management operations with an alternate DID or PID value. Table 4-1 describes the possible sources for driving the 8-bit task ID on the data bus. Note that the exclusiveness of some rows is enforced via programming rules, see Rule A.9.

Table 4-1. Sources of driving the Task ID on the data bus

Case		Source	Details/comments
ACATOR.P instruction with the AID attribute bit set. See Section 4.11, "Instruction attribute override."		AID	Instructions with Access Attribute Override: overriding DID/PID and SR2.EXP settings, for data accesses and cache instructions (program and data)
PUSHC*/POPC* instruction. See Section 4.9.1, "Stack pointer selection."		DID	Instructions for saving/restoring the context of the current/next task, regardless of the EXP setting
If not one of the instructions above, then:			
	Data memory access or data cache instruction	DID	When not in an exception, the default behavior for data access and data cache instructions (see Table 4-11, instructions starting with "D")

Table 4-1. Sources of driving the Task ID on the data bus (continued)

Case		Source	Details/comments
SR2.EXP = 0	A program cache instruction	PID	When not in an exception, the default behavior for program cache instructions (see Table 4-11, instructions starting with “P”). These instructions are communicated to the CME through the data buses.
SR2.EXP = 1	A memory access or cache instruction (program and data)	0x00	The default behavior of data accesses and cache instructions (program and data) when in an exception

Additional information on this topic is described in Section 4.11, “Instruction attribute override,” Section 4.10, “Cache control instructions,” and Section 4.9, “Stack model.”

4.2.2 Byte and bit ordering

The SC3900FP memory architecture is big endian. In big-endian byte order architectures, the system stores a multi-byte value with the big end first, in other words, the most significant byte occupies the lowest address. For example, in big-endian byte order, the `st.w d0,(r0)` instruction stores bits 15–8 of D0 into address (R0) and bits 7–0 into address (R0 + 1).

Figure 4-3 illustrates how the core transfers data from a register to memory in the 2-byte store example:

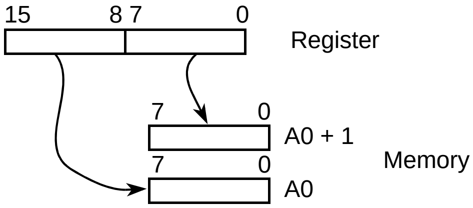


Figure 4-3. Register-to-memory big endian byte mapping

The bit-ordering convention in SC3900FP is little endian, meaning that the lowest bit index has the least significance. This convention can be seen as an example in the bit ordering of the register in Figure 4-3.

4.3 Data addressing modes

This section describes the addressing modes for data accesses. An addressing mode defines how the effective address is calculated. Some addressing modes also include a post-update of the address register, in preparation for the next access. The SC3900FP supports three types of data addressing modes:

- Indexed addressing modes
- Indirect addressing modes (with or without post-update)
- Implicit SP reference addressing mode

Post-update addressing modes could be defined to use non-linear arithmetic to support modulo buffers and FFT kernels, see Section 4.4, “Address modifier modes.”

4.3.1 Indexed addressing modes

In this addressing mode, the effective address is calculated through an arithmetic operation before it is used. The address registers used in that calculation are not modified and remain unchanged after the instruction is executed.

This table shows the different types of pre-calculated indexed addressing modes.

Table 4-2. Types of pre-calculated indexed addressing modes

Addressing mode	Description
Indexed by Address Register, scaled: (R*n* + R*k*)	The effective address is the sum of the contents of the R*n* address register and the contents of an R*k* address register, pre-shifted to the left by up to 6 bit positions according to the access width. The contents of the R*n* and R*k* registers remain unchanged. For example: `ld.l (r0+r17),d6`. Here, the access width is four bytes, so the value in R17 is shifted left by two bits before it is added to the value of R0 and used as the effective address. **Note:** Only address registers (R0–R7 and R16–R23) can be used as offsets (R*k*) in this and in similar addressing modes.
Long Displacement: (R*n* + *xxxx*)	The effective address is the sum of the contents of Rn with an explicitly specified offset, which is not scaled by the hardware. The offset is specified as an absolute value (signed 16 bits), and is not required to be aligned to the access width in the assembly source code.
SP Long Displacement: (SP + *xxxx*)	The effective address is the sum of the contents of SP with an explicitly specified offset. SP is the active stack pointer as determined by the SPSEL field in SR2 (or by ASPSEL for relevant instructions). The offset is specified as an absolute value (signed 16 bits), and is not required to be aligned to the access width in the assembly source code.
SP Short Displacement: (SP – *xx*)	The effective address is calculated by subtracting an explicitly specified offset from the contents of SP. The offset is encoded as an absolute value (unsigned 9 bits). The assembler automatically chooses this form if the displacement can fit in this format, saving code size. The offset as specified in the source code must be aligned to the width of the access. The assembler does not encode the LS zero bits of the displacement and thus increase the allowed offset range for accesses that are wider than a byte access.

NOTE

In previous StarCore architectures, most of the instructions using pre-calculated addressing modes (for example, indexed by Address Register: (R*n* + R*m*)) were defined as 2-cycle instructions. In the SC3900FP all these instructions are single cycle instructions, however, they support only the linear arithmetic mode.

4.3.2 Indirect addressing modes

In indirect addressing modes, the effective address is the value in the R*n* address register. The address register is used to point to a memory location. After the address is issued, the R*n* register may be updated (incremented or decremented) according to the different modes.

Table 4-3. Addressing modes descriptions

Mode	Description
No Update: (R*n*)	The effective address is stored in R*n*, which remains unaffected after the operation.
Post-Increment: (R*n*)+	The effective address is stored in R*n*. After the access R*n* is incremented by 1, 2, 4, 8, 16, 32 or 64 according to the width of the access to memory (byte, word, long, double-long, quad-long, etc., respectively). Hence "+" signifies incrementing Rn by the number of bytes that were transferred by a single access of this type.
Post-Decrement: (R*n*)–	Similar to the Post-Increment mode, only decrements by the number of bytes that were transferred by a single access of this type.
Post-Increment by Address Register: (R*n*)+R*k*	After the effective address is issued, R*n* is incremented or decremented by the signed value in the R*k* register, pre-shifted to the left by up to 6 bit positions according to the access width. The result is stored in the same address register. The contents of the R*k* register is unchanged. For example: `st.w d3,(r2)+r3`. The access width is two bytes, so the increment is twice the value in the R3 register. **Note:** Only address registers (R0–R7, R16–R23) can be used as offsets in this addressing mode.

For the post-update addressing modes, the type of arithmetic used for updating R0–R7 by relevant instructions is determined by programming the MCTL register.

4.3.3 Implicit SP reference addressing mode

Some instructions make implicit reference to the active Stack Pointer (SP). For example, "`push d2`" stores the data register D2 into the memory location pointed by the SP, and post-increments the SP by 8. For more information on the stack model of the SC3900FP, see Section 4.9, "Stack model."

4.4 Address modifier modes

The LSUs support special types of arithmetic for addressing modes that involve a calculation (post update only), and for a subset of arithmetic instructions such as ADDA. These arithmetic modes are available only for calculations that involve registers R0–R7 as their R*n* in pointer post-update calculation:

- Linear addressing
- Reverse-carry addressing
- Modulo addressing mode
- Multiple wrap-around modulo addressing

These arithmetic types allow the easy creation of data structures in memory for first-in-first-out (FIFO) queues, delay lines, circular buffers, stacks, and reverse-carry FFT buffers.

The core supports the manipulation of data by updating address registers (R*n*) used as pointers rather than by moving large blocks of data. The fields in the MCTL register define the type of arithmetic to be performed for address calculations on each address register.

4.4.1 Linear addressing

Linear addressing is useful for general-purpose addressing such as for stacks. In linear addressing, the address is calculated using standard binary arithmetic. The entire memory space is addressable. This is the default address modifier mode.

4.4.2 Reverse-carry addressing

Reverse-carry addressing supports 2^k-point FFT addressing. The address modification is performed in the hardware by propagating the carry from each pair of added bits in the reverse direction (from the most significant bit to the least significant bit). For the (R*n*)+R*k* post-update addressing mode, reverse-carry is equivalent to the following process:

1. Bit-reverse the contents of R*n* (redefining the most significant bit as the least significant bit, the next most significant bit as bit 1, and so on).
2. Shift the offset value in R*k* 0 to 6 positions to the left according to the access width.
3. Bit-reverse the shifted R*k*.
4. Add normally.
5. Bit-reverse the result.

This address modification is useful for addressing the twiddle factors in 2^k-point FFT addressing as well as for unscrambling 2^k-point FFT data. The range of values for R*k* is 0 to (2^{32} – 1), which allows reverse-carry addressing for FFTs up to 4,294,967,296 points. The user should align the base address of the arrays to $2^k \times W$, where W is the number of bytes in a data element. For example, in a 256-point FFT on a 16-bit array, you should align the base address of the array to 256 × 2 = 512 bytes.

NOTE

> To achieve correct reverse-carry accessing for access widths of 2, 4, or 8, the 1, 2, or 3 least significant bits (respectively) of the address calculation result are forced to zero.

4.4.3 Modulo addressing mode

Modulo address modification supports the creation of circular buffers for FIFO queues, delay lines, and sample buffers up to 2^{31} bytes long.

Modulo addressing is selected by writing to the appropriate AM_R*n* field of the MCTL register (as shown in Section 2.1.4, “Registers for non-linear arithmetic support”) as well as writing the desired modulus value M to the corresponding M*j* register. Address modification is performed in modulo M, where M ranges from 1 to 2^{31} – 1. Modulo M arithmetic causes the address register values to remain within an address range of size M, thus defining a buffer with a lower and an upper address boundary.

Each base address register (B*n* register) is associated with an R*n* register (B0 with R0, and so on). Each R*n* register has one M*j* register assigned to it by encoding in the MCTL register. The lower boundary value of the buffer resides in the B*n* register, and the upper boundary is calculated as B*n* + M*j* – 1. M*j* must be smaller than 2^{31} – 1 (M*j* < 2^{31} – 1).

B0–B7 are physically aliased to registers R8–R15, and M0–M3 are physically aliased to R24–R7.

The modulo addressing definition, using a base register (B*n*) and a modulo register (M*j*), enables the programmer to locate the modulo buffer at any address. The buffer start address is required to be aligned only to the access width.

The address pointer R*n* is not required to start at the lower address boundary nor to end on the upper address boundary. Initially, R*n* can point anywhere (aligned to its access width) within the defined modulo address range, B*n* ≤R*n* < B + M*j*. Assuming the (R*n*)+ indirect addressing mode, if the address register pointer increments past the upper boundary of the buffer (base address + M*j* – 1), it wraps around through the base address (lower boundary). Alternatively, assuming the (R*n*)– indirect addressing mode, if the address decrements past the lower boundary (base address), it wraps around through the upper boundary (base address + M*j* – 1).

The following constraints apply:

1. For proper modulo addressing, if an offset R*k* is used in the address calculation, the 32-bit absolute effective value |R*k*| must be less than or equal to M*j*, where “effective” means that the programmed R*k* is multiplied by the access width. For example, `st.w (r0)+r5,d0` translates to the restriction 2 × R5 ≤M*j*, and `st.l (r0)+,d0` translates to 4 ≤ M*j*. If the effective R*k* is greater than M*j*, the result of the address calculation is undefined. However, multiple wrap-around modulo addressing does support effective R*k* values greater than M*j*.
2. M*j* must be aligned to the access width used. For example, if the buffer is used with a LD.2L instruction, M*j* must be aligned to 8 (be a multiple of 8). If the modulus is less than the access width, the data accessed as well as the address calculations are undefined.
3. When B*n* is used as a base address register, the use of R<*n* + 8> as a pointer is illegal because this is the same physical register. Likewise, when M*j* is used as a modulo register, the use of R<24+*j*> as a pointer is illegal because this is the same physical register.

Modulo addressing is illustrated in Figure 4-4. Addresses are kept within the 11 addresses shown. For the instruction `st.w (r0)+r5,d0`, if R0 is 0x24, R5 is 0xE, the base address is 0x20, the modulus is 0xC, and so R0 is updated to 0x26. The operation is 36 + 14 = 50 = 38 in modulus 12, base address 32 (50 – 44 + 32 = 38).

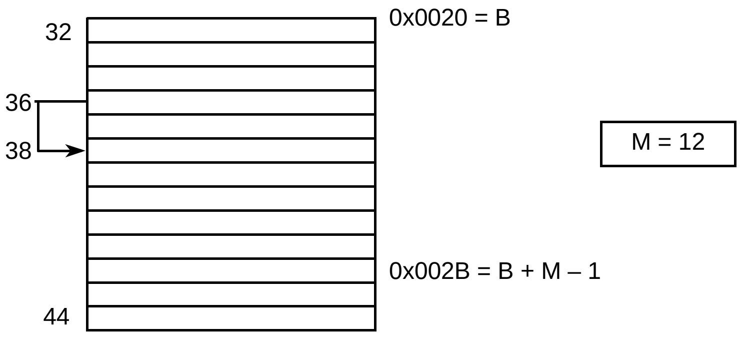


Figure 4-4. Modulo addressing example

Table 4-4 describes the modulo register values and the corresponding address calculation:

Table 4-4. Modulo register values for modulo addressing mode

Modifier M*j*	Address calculation arithmetic
0000 0000h	Unused
0000 0001h	Modulo 1
0000 0002h	Modulo 2
7FFF FFFEh	Modulo $2^{31} – 2$
7FFF FFFFh	Modulo $2^{31} – 1$

4.4.4 Multiple wrap-around modulo addressing

Multiple wrap-around addressing is useful for decimation, interpolation, and waveform generation. The multiple wrap-around capability can be used for argument reduction. In multiple wrap-around modulo addressing mode, the modulus M is a power of 2 in the range of 2^1 to 2^{31}. The value M – 1 is stored in the modifier register (M*j*). The B registers (B0 to B7) are not used for multiple wrap-around modulo addressing; therefore, the corresponding R8–R15 registers can be used for other purposes in parallel.

The lower and upper boundaries are derived from the contents of M*j*. The lower boundary (base address) value has zeros in the *k* least significant bits, where $M = 2^k$, and therefore must be a multiple of M. The R*n* register involved in the memory access is used to set the most significant bits of the base address. The base address is set so that the initial value in the R*n* register is within the lower and upper boundaries. The upper boundary is the lower boundary plus the modulo size minus one (base address + M – 1).

The size of the modulo buffer must be aligned to (be a multiple of) the access width. If the modulus is less than the access width, the data accessed as well as the address calculations are undefined.

If an offset R*k* is used in the address calculations, it is not required to be less than or equal to M for proper modulo addressing. The multiple wrap-around modulo addressing mode supports unlimited boundary wraps.

When using the (R*n*)+ and (R*n*)– addressing modes with a modulus of $2^k \geq 8$, there is no functional difference between the multiple wrap-around and normal modulo modes because the address can be wrapped around only once.

As an example, consider the instruction `st.w (r0)+r5,d0`. If the MCTL is set to 0xC and M0 is set to 0xF, then M0 = 16. If R0 is initially 0x24 (36), the lower boundary is 0x20 (32) and the upper boundary is 0x2F (47). R*k* is 0x42 (66), R0 is updated to 0x26 (38), calculated by 36 + 66 = 102, 102 – 48 = 54, 54 – 3 × 16 = 6, and 6 + 32 = 38.

Table 4-5 describes the modulo (M*j*) register values and the corresponding multiple wrap-around modulo address calculation.

Table 4-5. Modulo register values for wrap-around modulo addressing mode

Modifier M*j*	Address calculation arithmetic
0000 0001h	Multiple wrap-around modulo 2
0000 0003h	Multiple wrap-around modulo 4
0000 0007h	Multiple wrap-around modulo 8
7FFF FFFFh	Multiple wrap-around modulo 2^{31}
FFFF FFFFh	Linear

4.4.5 Arithmetic instructions affected by non-linear arithmetic

For some arithmetic instructions, address modification modes can affect the arithmetic results using the LSU instructions ADDA, SUBA, ADDLA.*n* (n = 1..6). The *second* source operand of these instructions (for example, R0 in the instruction `adda r2,r0,r7`) is the one used for defining the modulo mode affected by MCTL. Only R0–R7 can be used as this second operand if non-linear calculation is intended. The respective B*n* and M*j* are used in the same way as in pointer non-linear addressing.

For a detailed description of the operation of these instructions, see the SC3900FP Instruction Set.

4.5 Arithmetic capabilities of the AGU

The SC3900FP core provides arithmetic instructions on the address registers (R0–R31). These instructions can mostly be executed both in the LSUs and in the IPU.

4.5.1 Computational instructions of the LSU

The LSU supports the following main types of arithmetic instructions:

- Compare and predicate manipulation instructions
- Addition/subtraction
- Logic instructions (such as ANDA)
- Register transfers
- Sign/zero extend
- Multi-bit shifts

As described in Section 4.4.5, “Arithmetic instructions affected by non-linear arithmetic,” a subset of the LSU addition/subtraction instructions that operate on R0–R7 as the second source operand support non-linear arithmetic in the same way as it is done for load and store operations (configured in MCTL).

4.5.2 Computational instructions of the IPU

The IPU supports the following main types of arithmetic instructions:

- Compare and predicate manipulation instructions (a superset of the LSU predicate updating instructions)

- Addition/subtraction instructions, only in linear mode (in linear mode, same functionality as the LSU)
- Logic instructions such as ANDA (duplicated with the LSU)
- Register transfer instructions (most TFRA, MOVE D↔R — some are duplicated with the LSU)
- Sign/zero extend (duplicated with the LSU)
- Integer multiplication instructions (unique)
- Multi-bit shift instructions (some variants duplicated with the LSU, others unique to the IPU)
- Bit manipulation and test of R and control registers (unique)

To ease the dispatching process, IPU instructions have a dedicated encoding group. However, instructions that have exact functionality as their counterparts in the LSU have the same assembly mnemonic and syntax. The assembler determines, according to the availability of execution units, whether to encode them as IPU or LSU instructions. For example:

```
ld.l (r0),r5      anda r4,r15       tfra r20,r21
```

The LD has to be an LSU instruction; the assembler will encode either ANDA or TFRA as an IPU instruction.

4.6 Load and store instructions

Store (ST) and Load (LD) are the main instructions that transfer data from the core to the memory or vice versa. The MOVE mnemonic is generally used for data transfers between core registers. Also, some specialized memory transfers use different mnemonics (for example, PUSH, and POP).

In SC3900FP, load and store instructions play a critical role in supporting the different data types by performing the implicit casting of the transferred data to and from the register which is usually of a different size. For an overview of the data types, see Section 2.4, “Data type support.”

The various types of memory LD and ST instructions differ in the following properties:

- Access width:
  - Byte (8 bits)
  - Word (16 bits)
  - Long word (32 bits)
  - Double-long word (64 bits)
  - Quad-long word (128 bits)
  - 8 long words (256 bits)
  - 16 long words (512 bits)
- Position in the register:
  - Aligned to the right, for integer data types
  - Aligned to the right of the implicit decimal point, for fractional data types
- Sign or zero extension (for loads), in support of signed and unsigned data types
- Addressing mode
- Multi-element accesses: The number of data elements that are loaded/stored together

- Multi-register accesses: Some multi-element moves may read/write the data from/to several registers
- Scaling, rounding, and limiting data during a memory store (the ST.SRS instruction)
- Specialized load/store instructions that are application specific for DSP kernels

Multiple registers that are accessed together belong in most cases to a contiguous index range, and are marked with a “:” separating the registers, for example:

```
ld.4w (r0),d2:d3:d4:d5
```

A shorthand notation for a contiguous register range is with “::”, for example:

```
ld.4w (r0),d2::d5
```

4.6.1 Width and register alignment of memory load/store instructions

The data type (width and register alignment) that is handled by the access is marked in the width suffix of the mnemonic. In case of several flags, the width suffix is the last:

- LD/ST.B: Integer byte
- LD/ST.BF: Fractional byte (used for data in D registers only)
- LD/ST.W: Integer word
- LD/ST.F: Fractional word (used for data in D registers only)
- LD/ST.L: Integer long word, fractional long word (D registers only), or single precision floating point number (D registers only)
- LD/ST.X: 40-bit long word (used for data in D registers only)

The data type affects the width of the access and its alignment in the source or destination register. Examples of the basic load types are given in Figure 4-5. Unless otherwise specified, accesses are considered as signed (see Section 4.6.2, “Signed and unsigned memory load instructions”). Matching store accesses take the date from the same positions in the D register.

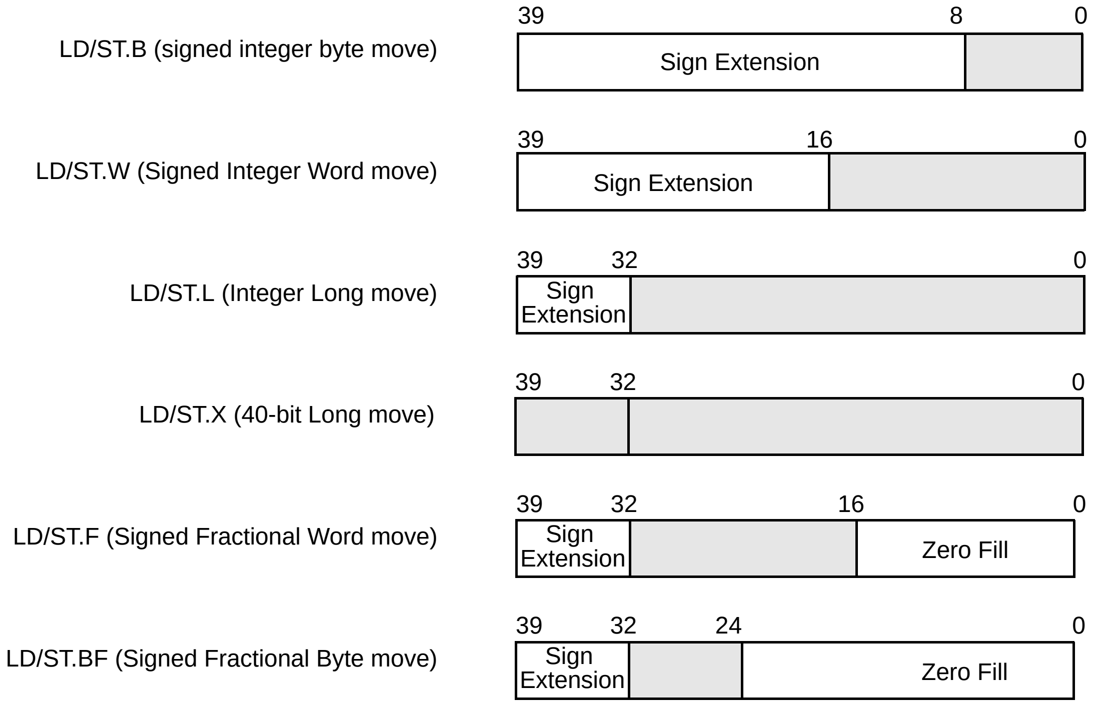

LD/ST.B (signed integer byte move)
39
8
0
Sign Extension
LD/ST.W (Signed Integer Word move)
39
16
0
Sign Extension
LD/ST.L (Integer Long move)
39
32
0
Sign Extension
LD/ST.X (40-bit Long move)
39
32
0
LD/ST.F (Signed Fractional Word move)
39
32
16
0
Sign Extension
Zero Fill
LD/ST.BF (Signed Fractional Byte move)
39
32
24
0
Sign Extension
Zero Fill

Figure 4-5. Basic load types to data registers

Load instructions to R registers use the same conventions, but use only a subset of these transfer types: LD.B, LD.W and LD.L.

ST.X and LD.X are special accesses intended to transfer a whole D register to/from memory. They are actually 64-bit accesses to memory, where the high order bits (63:40) are written as zeros and ignored during a read, see Figure 4-6.

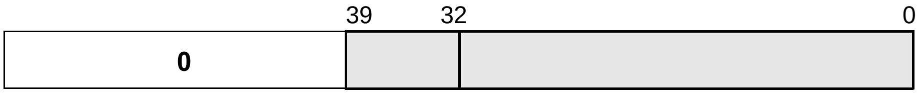


Figure 4-6. Organization of a 40-bit data type in memory

4.6.2 Signed and unsigned memory load instructions

Data that is loaded from memory could be interpreted as signed or as unsigned. This property determines if the data is sign- or zero-extended in the destination register. By convention, all loads are signed unless explicitly marked as unsigned. Fractional loads are always signed. Some of the integer loads are marked as unsigned by a ‘U’ flag following the LD mnemonic, as follows:

- LD.U.B: Unsigned byte load from memory.
- LD.U.W: Unsigned word load from memory.
- LD.U.L: Unsigned long word load from memory (used only to D registers)

Examples of signed accesses were given in Figure 4-5. Unsigned accesses are given in Figure 4-7.

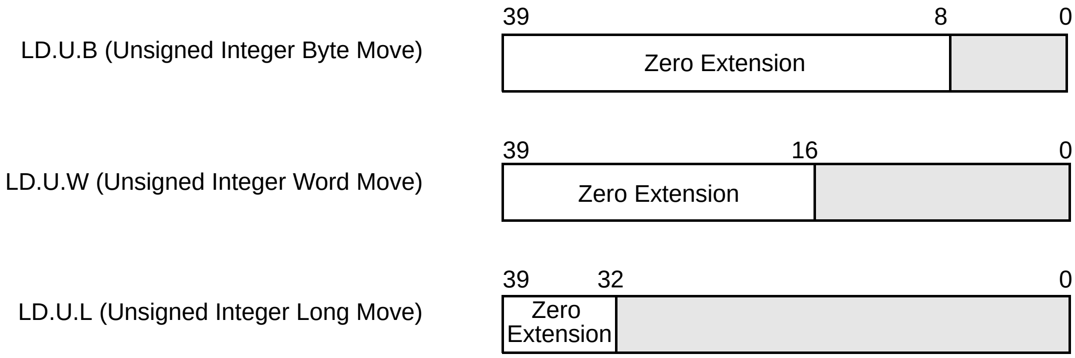

LD.U.B (Unsigned Integer Byte Move)
39
8
0
Zero Extension
LD.U.W (Unsigned Integer Word Move)
39
16
0
Zero Extension
LD.U.L (Unsigned Integer Long Move)
39
32
0
Zero Extension

Figure 4-7. Examples of Unsigned Integer MOVE Instructions

As sign extension occurs only when loading data from memory to a core register, there is no special variant for an unsigned store. The same instruction (by convention the one without the U) is used for unsigned memory stores, for example:

```
        st.b    d0,(r0)          ; store a signed or an unsigned byte to memory
```

Unsigned loads are supported for R registers as well (LD.U.B and LD.U.W).

4.6.3 Multi-element memory load and store instructions

Many memory load and store instructions can transfer several data elements with the same instruction. The number of data elements that are transferred by the access is marked by a number in the width suffix, right after the dot separator. For example:

- LD.2L or ST.2L: A 64-bit access, of two 32-bit long words.
- LD.4F or ST.4F: A 64-bit access, of four 16-bit fractional words.

Unless otherwise specified, each multi-variable accesses assumes that each variable is in a different register (see Section 4.6.4, "Packing several data elements in a single data register"). The set of registers that are used in such accesses are separated with ":", and are usually restricted to a consecutive index range. For example, accesses involving two data registers are limited to the register pairs d0:d1, d1:d2, d2:d3, and so on. Accesses involving four registers are usually limited to the register quads: d0:d1:d2:d3, d1:d2:d3:d4, and so on. Examples of register specifications are as follows:

```
        st.2l d0:d1,(r0)         ; store two 32-bit elements, taken respectively
                                 ; from d0 and d1
        ld.4f (r0),d8:d9:d10:d11 ; load four 16-bit elements, and place as four
                                 ; fractional words in each of d8, d9, d10 and d11
```

A shorthand notation for specifying a long contiguous register range is with "::", for example "d0::d7" is equivalent to "d0:d1:d2:d3:d4:d5:d6:d7".

There are multi-element accesses that use R or D registers. The following is the list of SC3900FP multi-element load instructions. There are equivalent store instructions, except for unsigned variants.

- LD.2B, LD.U.2B, LD.2W, LD.U.2W, LD.2L—D and R registers
- LD.2BF, LD.2F, LD.U.2L, LD.2X—D registers only

- LD.4W, LD.U.4W, LD.4L—D and R registers
- LD.U.L, LD.4B, LD.4BF, LD.U.4B, LD.4F—D registers only
- LD.8B, LD.U.8B, LD.8W, LD.U.8W, LD.8F,
  LD.8L, LD.U.8L—D registers only
- LD.16L, LD.U.16L—D registers only

The 16L accesses are unique—a single instruction drives both data buses and transfers 512 bits. No other memory access in the same direction is allowed in the VLES; however, the second LSU can perform in parallel an access in the other direction. For example, it is possible to perform simultaneous 512-bit load and store in the same VLES, as follows.

```
ld.16l (r0),d0::d15       st.16l d16::d31,(r1)
```

4.6.4 Packing several data elements in a single data register

By default, moving more than one data element is an unpacked move, meaning that each element is loaded (stored) from (to) one register. So, a 2-element access uses two registers.

The user is free to use a wide data access (such as a 32-bit LD.L) to transfer several smaller elements, for example, 4 bytes or 2 words. This is the main way that the architecture supports multi-variable packing.

While packing two 16-bit words or four bytes into a 40-bit register (not using the upper 8-bit extension) can utilize normal LD.L/ST.L instructions, loading to or storing from two 20-bit values in each Dn register requires dedicated instructions. These are cases that pack two data elements (bytes or fractional words) per register. This is marked by placing a ‘2’ right after the base mnemonic, for example:

- LD2.2B or ST2.2B: A 16-bit access of 2 bytes, that are organized in one data register in a packed 20-bit format. The load instruction casts the two signed integer bytes into two signed 20-bit integers.
- LD2.4F or ST2.4F: A 64-bit access of 4 words, that are organized as 4 packed fractional pairs in 4 registers. The load instruction casts the two signed integer words into two signed 20-bit fractions.

Note that for bytes, as a result of the casting into a wider format, the data in the register is not contiguous; however, the data in memory is. The interface logic of the register file combines or separates the data accordingly. Figure 4-8 shows examples for some packed multi-element byte accesses. A load of two packed fractional words (LD2.2F for example) is different from LD.L because the loaded data is sign extended independently to two 20 bit portions.

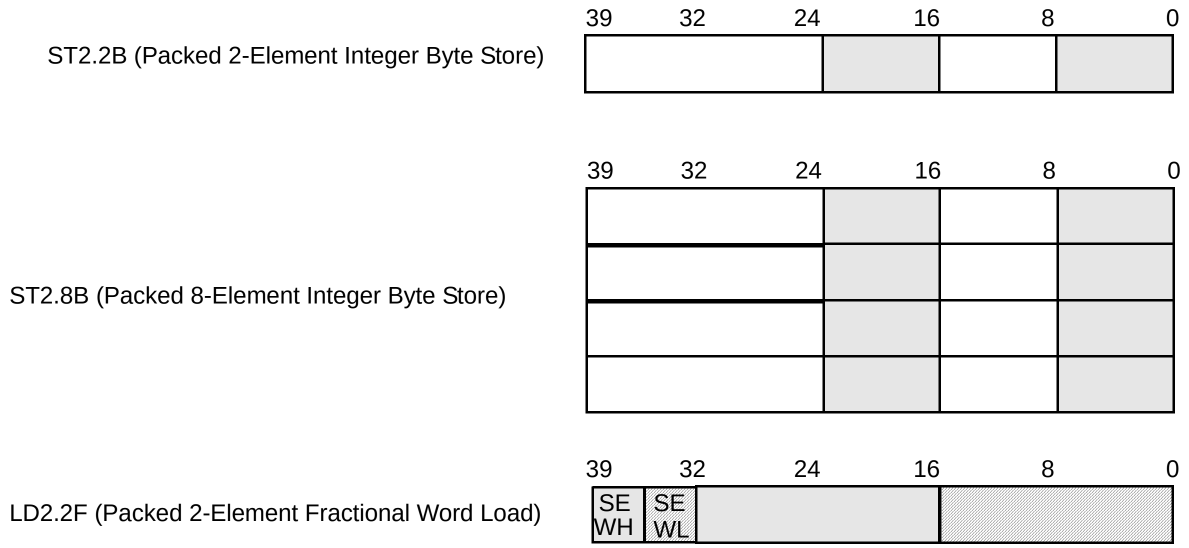

ST2.2B (Packed 2-Element Integer Byte Store)
39
32
24
16
8
0
ST2.8B (Packed 8-Element Integer Byte Store)
39
32
24
16
8
0
LD2.2F (Packed 2-Element Fractional Word Load)
39
32
24
16
8
0
SE WH
SE WL

Figure 4-8. Examples of multi-element packed load and store instructions

Packed data accesses are supported only for D registers. The following is the list of supported packed loads. There are equivalent store accesses, except for unsigned bytes and fractional words.

- LD2.2B, LD2.U.2B, LD2.2BF, LD2.2F
- LD2.4B, LD2.U.4B, LD2.4BF, LD2.4F
- LD2.8B, LD2.U.8B, LD2.8BF, LD2.8F
- LD2.16B, LD2.16BF, LD2.16F
- LD2.32B, LD2.32BF, LD2.32F

4.6.5 Addressing mode support in different load/store instructions

The addressing modes supported by nearly all LD and ST instructions are as follows:

- (Rn)
- (Rn)+
- (Rn)-
- (Rn+Rk)
- (Rn)+Rk
- (SP+s16)
- (Rn+s16)

These addressing modes are collectively referred to as “SAM” (Standard Addressing Modes), which are allocated to nearly every load or store instruction in SC3900FP.

A subset of control-oriented instructions support in addition a shorter form of SP-relative addressing:

- (SP-u9)

This form is functionally equivalent to (SP+s16) if the s16 value is in range of u9. The only difference is the encoding size of the instruction. The assembler automatically chooses the shorter form when it can.

Instructions that issue control commands to the data cache such as DFETCH support only the (Rn) addressing mode.

Some application-specific load and store instructions may have fewer addressing modes; normally, these instructions are not used outside of specific kernels.

4.6.6 Register transfers between R and D registers

The MOVE mnemonic is used to signify register to register transfers, using internal data paths that are shared with memory accesses, hence have similar pipeline behavior as memory loads and stores. Register transfers that use the arithmetic data path are named TFR or TFRA. For more details, see Chapter 7, "The Interlocked Pipeline." AGU supports the following register move instructions:

- MOVE.L Da,Ra
- MOVE.L Ra,Da
- MOVE.L Da,Dn
- MOVE.X Da,Dn
- MOVE.2L Da:Db,Dm:Dn
- MOVE.2X Da:Db,Dm:Dn

In addition, there are some unique instructions:

- MOVE.2X Da,Dm:Dn
  This instruction duplicates the value of Da in two destination registers Dm and Dn.
- MOVE.2L Da:Db,Rn,Rm
  Transfers two D to R registers with one instruction. There is no encoding dependency between the R register pair. The D register pair has to be of consecutive registers (D0:D1, D1:D2 etc.). Since this is an IPU instruction, it could be executed in parallel with two other LSU instructions.
- MOVE.4X Da,Db,Dc,Dc,Dm:Dn:Do:Dp
  The source quad operands are selectable from a wide combination of non-contiguous sources, which may also repeat some of the source registers. The destination is a contiguous register quad.

There are some limitations on the allowed bandwidth of register updates by MOVE instructions, as is described in Section 4.6.7, "Register bandwidth limitations on MOVE and memory accesses."

4.6.7 Register bandwidth limitations on MOVE and memory accesses

The SC3900FP has some limits on the bandwidth of transferring data to and from address, data and control registers, on top of the limit of the data bus width itself (256-bits per bus, or instruction). These limits originate from the connectivity structure of the register files. These limits relate to the maximal register selective capacity per VLES, in the context of memory loads, stores, and MOVE instructions. They are not related to bandwidth between the register file and the arithmetic units. It should be noted that all these rules are backward compatible with SC3850 and before, as they relate to the additional bandwidth that is new to the SC3900FP.

The instructions that transfer 512-bits in a single instruction (ST.16L, LD.16L, LD2.32F, and so on) are special. A 512-bit load actually activates both load buses from the memory (under the control of a single LSU), therefore, another load cannot be grouped with it in the same VLES. Similarly, a 512-store instruction activates both store buses from the core, so another store instruction cannot be grouped in the same VLES. However, a load and a store can be grouped, allowing a VLES that concurrently exchanges 1024 bits between the core and memory (512 bits in each direction). Because of the special way they operate, these instructions have several rules on their legal use, see Chapter 8, "Programming Rules and Guidelines."

An additional rule relates to the selection capacity, or the ability to specify independent operands (see Rule A.11 in Chapter 8, "Programming Rules and Guidelines"). A "selection group" is defined as a single or a group of operands that can be specified in an instruction without a relation to other operands in this instruction or in other instructions in the VLES. For example, in the context of sources or destinations of load instructions, the following instruction:

```
        ld.4l (r9),d0:d1:d2:d3
```

has one operand selection group because the register quartet elements cannot be independently selected. Loosely, an operand group for which the operands are separated by ":" is considered a single selection group, while an operand group for which the operands are separated by "," belong to different selection groups (however, some exceptions to this rule apply, see Table 8-3). The rule relating to selection groups is as follows: Independently for R and D registers, and for reads and writes, a VLES could support up to two selection groups of each type.

Examples are as follows:

```
        st.srs.f d5,(r0)  st.l d0,(r2)      move.l d1,r2      ; not allowed: 3 D source
                                                              ; selection groups

        ld.l (r0),d0:d1:d2:d3     ld.l (r1),d4:d5:d6:d7       ; allowed - each instruction
                                                              ; uses 1 D source sel. group
```

The operands that are relevant for this rule are only registers that are used as sources or destinations of data read from or written to the register file for memory transfers or with register-to-register MOVE instructions. Registers used as pointers or operands in arithmetic instructions are not subject to this rule.

4.7 Interlocks for linear and modulo arithmetic instructions

In order to support pre-calculation with no cycle penalty, the SC3900FP LSU has a clear distinction between linear calculations and non-linear ones. Linear calculations is a generalized term related to the following:

- Any AGU addition/subtraction operation (including post update address calculations) in linear mode, whether due to instruction type (ADDA.LIN), specific register (for example, R8) or dynamic MCTL configuration (for example, the value for the AM_Rn field in MCTL related to the base pointer is zero)
- Pre-calculated indexed addressing mode (Rn+Rk), (Rn+immediate) and SP-relative calculation
- All LSU non-move instructions not affected by MCTL (logic, arithmetic, predicate manipulation)
- All IPU instructions

Non-linear calculations are as follows:

- Any LSU addition/subtraction operation, which is part of a post update address calculations, in any of the non-linear modes (reverse-carry, linear/multi-wrap modulo)
- A subset of LSU arithmetic instructions (ADDA, SUBA, ADDLA.*x*) in any of the non-linear modes (reverse-carry, linear/multi-wrap modulo)

The LSU performs linear calculations in stage G of the pipeline, and non-linear calculations are delayed to stage A. This separation may cause interlocks when combining linear and non-linear calculations in sequence. This is exemplified by the following cases:

```
; mctl = 0x80 - affecting r1
adda r8,r9,r0              ; r0 calculation is linear (linearity determined by r9)
cmpa.eq #$13,r0,p2         ; no stall

; mctl = 0x80 - affecting r1
adda r8,r1,r9              ;r9 calculation is non-linear (linearity determined by r1)
cmpa.eq #$13,r9,p2         ; 1 cycle stall

; mctl = 0x80 - affecting r1
adda r1,r2,r9              ; r9 calculation is linear (linearity determined by r2)
tfra r9,r0                 ; no stall
```

Critical forwarding from non-linear operations to themselves (base only) is maintained to support maximal performance of DSP kernels:

```
; forwarding to memory address:
; mctl = 0x80000 - affecting r4
addla.1 r2,r4,r3           ; r3 calculation is non-linear (due to r4)
ld.w (r3),d0               ; no stall

; forwarding of the updated base pointer:
; mctl = 0x80000 - affecting r4
ld.w (r4)+r17,d0           ; r4 calculation is non-linear (due to r4)
ld.w (r4)+r17,d0           ; no stall
```

Note that the following cases, unlike the SC3850, incur stalls:

```
; non linear calculations of offsets: offset is most likely to be either a constant
; or of a “linear nature”
; mctl = 0x80000 - affecting r4
adda r17,r4,r4             ; r4 calculation is non-linear (due to r4)
ld.w (r6)+r4,d0            ; 1 cycle stall (forwarding to offset and not base)

; pre-calculation using non-linear inputs:
; mctl = 0x80000 - affecting r4
adda r17,r4,r8             ; r8 calculation is non-linear (due to r4)
ld.w (r8+r6),d0            ; 1 cycle stall - despite r4 being the “base” pointer
                           ; since pre-calculation has no non-linear forwarding
```

4.8 Unaligned memory access support

The SC3900FP core supports full, unaligned memory accesses for both loads and stores. Accesses in any width may use any address; for example, LD.L may be issued to address 0x103. Likewise, multi-element accesses are not required to be aligned with any specific address boundary. Both of the two load or store instructions that can be generated per VLES may be unaligned without limitation.

If an unaligned access crosses a 4 KB-aligned address boundary then a single hold cycle is generated; otherwise, there is no penalty on an unaligned access.

NOTE

> The exact boundary for hold generation depends on the implementation of the sub-system, the 4 KB boundary is for the current definition of the L1 cache.

The following are some special cases in which the address of the transaction must be aligned:

- Semaphore accesses: “load reserve” and “store conditional.” For more information, see Section 4.12, “Semaphore and atomic operations support.”
- Peripheral accesses. Usually some memory-mapped registers of peripheral units are defined as such in the MMU. See the MMU description in the *SC3900FP FVP Cluster Reference Manual*.
- These examples, as well as additional special memory accesses that must be aligned, are listed in Rule A.9 in Chapter 8, “Programming Rules and Guidelines.”

A memory error is generated if an unaligned address is issued in these cases.

4.9 Stack model

The AGU has three stack pointer registers, which provide hardware assistance for managing the software stack:

TSP — The Task Stack Pointer is intended for usage of a task.

ESP — The Exception Stack Pointer is automatically selected when the core jumps to an exception other than the Debug exception. This is intended for use during exceptions and for the OS kernel that is not associated with a task

DSP — The Debug Stack Pointer is automatically selected when the core jumps to service a Debug exception.

Only one stack pointer is active at any time, according to the bits stored in the SPSEL field in SR2. This value is set automatically upon jumping to an exception, and restored when returning from an exception. The SC3900FP has dedicated instructions that assist in managing and using the stack, as follows:

- SP-relative accesses that do not update the SP itself. These instructions help the compiler in passing parameters to functions when the stack pointer is more or less stable (for example, the SP value is explicitly set to account for the memory space allocated for parameter passing).
- Push and Pop instructions, which update the stack pointer while transferring data to and from the stack. These instructions are mainly used for context save and restore and for caller-saved parameter passing.
- Instructions that help manipulate the value of SP directly, such as adding a value to it and so on.

All the stack pointer registers have their 3 LS bits tied to zero, meaning that the SP is by definition 8-byte aligned. However, when using SP-relative addressing such as (SP+s16), there is no alignment limitation on the address that is accessed.

The user can manage the stack explicitly by manipulating the value of SP and load or store data into memory using SP-relative addressing. It is also possible to use PUSH and POP instructions that implicitly update SP upon every access. PUSH and POP instructions assume that SP is pointing to the next unwritten address; a push stores to memory with the address in SP and then increments it while a pop first decrements SP and then loads from the memory.

4.9.1 Stack pointer selection

At any point, only one stack pointer is active. The active stack pointer is used by default, whenever an instruction that specifies “SP” as an operand is used, such as LD.L (SP-100),D0 or TFRA SP, Rn. The active stack pointer is also used when SP is used implicitly, for example, by the JSR or POP instructions. The active SP is selected in the SPSEL field in SR2. The value of SPSEL is modified in one of the following cases:

- Upon jumping to an exception: a non-debug exception changes SPSEL to select ESP, and in debug exceptions to select DSP
- Upon returning from an exception (RTE instructions): the value of SPSEL is restored from the active stack. When the OS dispatches a new task, it configures the stack location that would be restored to SR2.SPSEL with the value selecting TSP, and performs RTE*.

Attempting to write SPSEL not through RTE (for example, with TFRA) will be ignored, although other fields may be updated by this instruction. In this case, no error exception is generated.

The ASPSEL (Alternate SP Select) field in SR2 allows the user to specify, for specific instructions, a different SP, thereby overriding the SP selection specified in SPSEL. Such an ability is useful when the OS wishes to peek or manage data in the stack of a task while maintaining the ability to work with its OS stack at the same time.

The ASPSEL field can be modified in the following ways:

- During exception processing, the previous value of SPSEL is copied into ASPSEL
- During RTE* the value of SR2, including ASPSEL, is restored from memory
- Using the CHCF #u2 instruction. Selecting the alternate SP directly.
- Using the CHCF Rn instruction. Allowing to restore the value of ASPSEL after it was saved aside with a TFRA SR2,Rn instruction, without modifying the other fields in SR2.
- TFRA Rn,SR2 modifies ASPSEL with the rest of the writable fields in SR2.

As mentioned, at the beginning of the exception service routine, ASPSEL holds the identity of the previous SP. This feature allows the interrupt or task switch routine an easy way to access and/or save the context of the task that is swapped out on the stack of that task.

A dedicated subset of instructions that use SP have a variant that chooses SP according to ASPSEL instead of SPSEL. These instructions are marked with a “C” in the base mnemonic, as follows:

- POPC and PUSHC are variants of PUSH and POP, and are defined specifically for accessing the previous context stack by the OS. SP, from ASPSEL, holds the SP by default before the jump to exception. The ID driven on the bus is from DID, which is not changed by default when jumping to an exception.
- TFRCA SP,Rn and TFRCA Rn,SP transfer a value to and from non-active SP registers.

In order to access explicitly one of the three stack pointers, update ASPSEL with the required value and then use one of the instructions mentioned above. If the required operation is not included in this subset, the user can use TFRCA SP,Rn, work with Rn addressing and, if required, restore the updated value with TFRCA Rn,SP.

4.9.2 PUSH and POP instructions

PUSH and POP instructions increment and decrement the stack pointer by different steps, depending on the width of the data that is transferred. PUSH writes to the address pointed by SP and then increments it; POP first decrements SP and then performs the memory read. The minimum stack updating granularity is 8 bytes.

Table 4-6 lists the update granularity of the different PUSH instructions.

Table 4-6. SP update granularity of PUSH instructions

Instruction[1]	Description	Memory access	SP update	Grouping restrictions with other PUSH instructions
PUSH OPe	Push an operand belonging to the “even” group	OPe → [SP:SP+3]	8	Cannot be grouped with another single PUSH of an even register
PUSH OPo	Push an operand belonging to the “odd” group	OPo → [SP+4:SP+7]	8	Cannot be grouped with another single PUSH of an odd register
[PUSH OPe PUSH OPo]	Push an “even” and “odd” operand pair, grouped in a VLES	OPe:OPo → [SP:SP+7]	8	No other PUSH instructions
BSR and JSR	Jump or Branch to a subroutine	PC:SR → [SP:SP+7]	8	Not allowed with another PUSH
PUSH.2L OP1:OP2	Push two operands with a single instructions	OP1:Op2 → [SP:SP+7]	8	Cannot be grouped with PUSH.4L and PUSH.4X.
PUSH.4L OP1:OP2:OP3:OP4	Push 4 operands with a single instruction	OP1:OP2:OP3:OP4 → [SP:SP+15]	16	Can only be grouped with PUSH.4X
PUSH.4X Da:Db:Dc:Dd	Push 4 40-bit data registers with a single instruction	Da:Db:Dc:Dd → [SP:SP+31] Each 40-bit D register is zero padded to 64 bits before storing	32	Can only be grouped with PUSH.4L and PUSH.4X

[1] For each PUSH instruction there is a corresponding PUSHC instruction

This table lists the update granularity of the different pop instructions. The table assumes the value of SP before the instruction is executed.

Table 4-7. SP update granularity of POP instructions

Instruction[1]	Description	Memory access	SP update	Grouping restrictions with other POP instructions
POP OPe	Pop an operand belonging to the “even” group	[SP-8:SP-5] → OPe	8	Cannot be grouped with another single POP of an even register
POP OPo	Pop an operand belonging to the “odd” group	[SP-4:SP-1] → OPo	8	Cannot be grouped with another single POP of an odd register
[POP OPe POP OPo]	Pop an “even” and “odd” operand pair, grouped in a VLES	[SP-8:SP-1] → OPe:OPo	8	No other POP instructions
RTS*	Return from subroutine	[SP-8:SP-5] → PC	8	Not allowed with another POP
POP.2L OP1:OP2	Pop two operands with a single instructions	[SP-8:SP-1] → OP1:Op2	8	Cannot be grouped with POP.4L or POP.4X
POP.4L OP1:OP2:OP3:OP4	Pop 4 operands with a single instruction	[SP-16:SP-1] → OP1:OP2:OP3:OP4	16	Can only be grouped with POP.4X
POP.4X Da:Db:Dc:Dd	Pop 4 40-bit data registers with a single instruction	[SP-32:SP-1] → Da:Db:Dc:Dd for each register, only the lower 40 bits of each 64-bit access are used	32	Can only be grouped with POP.4L and POP.4X
RTE*	Return from exception	[SP-32:SP-1] → PC:SR:SR2:EIDR	32	Cannot be grouped with another memory access

[1] For each POP instruction there is a corresponding POPC instruction

The “even” and “odd” operand attribute relates to the functionality of PUSH and POP instructions that operate on a single register, as follows:

- “Even” registers: R*n*, D*n* where *n* is an even number; in addition: GCR, BTR0, TMTAG
- “Odd” registers: R*n*, D*n* where *n* is an odd number; in addition: SR, MCTL, BTR1, TID

Up to two PUSH instructions can be grouped together in a VLES. The semantics of this grouping is serial, meaning that the second push is performed after the first push has incremented the SP, according to the granularity of the instruction, as specified in Table 4-6. This means that at the VLES level, SP can be updated in jumps of either 8, 16, 24, 32, 40, 48 and 64. As long as the POP operations are performed in a reversed VLES order, the integrity of the data is maintained. Inside the VLES, the order of POP instructions should match that of the PUSH instructions. The following is an example:

```
push.2l d0:d1     push d5
pop d5            pop.2l d0:d1      ; d0, d1, d5 will hold the original values
```

There are some restrictions on the allowed combinations of PUSH instructions, according to their width (respectively on POP instructions). Table 4-6 lists the allowed combinations.

Note that on average, it is probable that the value in SP will not be aligned with the size of the access. For example, during a task switch, two 256-bit accesses may be performed in parallel, which in previous architectures would require that SP be aligned to 32. However, the non-alignment support of SC3900FP nearly always allows to perform such accesses without a stall. Only upon passing a boundary of 4 KB will a stall cycle be added (see Section 4.8, “Unaligned memory access support”). This allows the user to ignore any consideration of SP alignment in most scenarios.

4.9.3 Spilling 40-bit registers to the stack

The SC3900FP has additional instructions that enable the core to read and write a full Dn register (40-bits) from/to the memory. The mnemonic convention for this instruction is the suffix “.X”. The LD.X and ST.X instructions issue 64-bit accesses, of which the lower 40 bits are the contents of the D register. The upper 24 bits are written as zeros. A single instruction may be used to save/restore two such registers using a 128-bit transaction. The following is an example.

```
st.2x d0:d1,(sp+s16)
```

For details, see Section 4.6.1, “Width and register alignment of memory load/store instructions.”

To aid in context save and restore, dedicated push and pop instructions were added which transfers four full D registers at once using a 256-bit access:

```
push.4x Da:Db:Dc:Dd
pop.4x Da:Db:Dc:Dd
```

Figure 4-9 shows the memory map after executing a PUSH.4X Da:Db:Dc:Dd instruction. A POP.4X instruction works in the opposite way.

0	3	8	11	16	19	24	27	31
0	Da	0	Db	0	Dc	0	Dd	

Figure 4-9. Memory map after a PUSH.4X instruction

The byte numbers represent the offset from SP before the push.

4.9.4 Saving and restoring the context to/from the stack

Using the instructions defined in Table 4-6, saving the register context of SC3900FP takes 14 cycles (optionally, it takes 15 when crossing a 4 KB memory boundary) and looks as follows:

```
push.4x d0:d1:d2:d3        push.4l r0:r1:r2:r3
push.4x d4:d5:d6:d7        push.4l r4:r5:r6:r7
push.4x d8:d9:d10:d11      push.4l r8:r9:r10:r11
push.4x d12:d13:d14:d15    push.4l r12:r13:r14:r15
push.4x d16:d17:d18:d19    push.4l r16:r17:r18:r19
push.4x d20:d21:d22:d23    push.4l r20:r21:r22:r23
push.4x d24:d25:d26:d27    push.4l r24:r25:r26:r27
push.4x d28:d29:d30:d31    push.4l r28:r29:r30:r31          tfra lc0,r0
push.4x d32:d33:d34:d35    push.4x d36:d37:d38:d39          tfra lc1,r1
```

```
        push.4x d40:d41:d42:d43    push.4x d44:d45:d46:d47            tfra lc2,r2
        push.4x d48:d49:d50:d51    push.4x d52:d53:d54:d55            tfra lc3,r3
        push.4x d56:d57:d58:d59    push.4l r0:r1:r2:r3                tfra procid,r0
        push.4x d60:d61:d62:d63    push.4l gcr:mctl:btr0:btr1         tfra tmtag,r1
        push.2l r0:r1
```

Notes:

- PC, SR, SR2 and EIDR are copied to the appropriate link registers and should be saved as well with a PUSH.4L LR*n* instruction, usually as one of the first instructions in the ISR. This push is usually followed by re-enabling some critical exceptions (such as MMU exceptions), therefore, it is generally viewed as part of the interrupt management overhead and it is not included as part of the context save.
- MOCR holds general core configuration bits that are usually not task-specific, hence this register is not part of the saved context. Restoring MOCR requires a CLRIC instruction in parallel, which adds a significant cycle overhead to the task switch routine and thus is better avoided.
- Replacing all PUSH.* instructions with the respective PUSHC.* instructions allows the OS, which runs in exception mode using ESP, to save the context of the aborted task to its own stack instead of saving it to the stack of the OS.
- SP, TID.DID and, if using PUSHC, also SR2.ASPSEL should be saved independently, normally in the TCB (Task Control Block) in the OS memory, because restoring these two registers (at least the SP and the DID field in TID) is a pre-condition to restore the rest of the saved context.

Similar to the context saving routine, the assembly sequence performing the restoration of the context from the stack looks like the following.

```
        pop.2l r0:r1
        pop.4l gcr:mctl:btr0:btr1  pop.4x d60:d61:d62:d63    tfra r1,tmtag
        pop.4l r0:r1:r2:r3         pop.4x d56:d57:d58:d59    tfra r0,procid
        pop.4x d52:d53:d54:d55     pop.4x d48:d49:d50:d51    tfra r3,lc3
        pop.4x d44:d45:d46:d47     pop.4x d40:d41:d42:d43    tfra r2,lc2
        pop.4x d36:d37:d38:d39     pop.4x d32:d33:d34:d35    tfra r1,lc1
        pop.4l r28:r29:r30:r31     pop.4x d28:d29:d30:d31    tfra r0,lc0
        pop.4l r24:r25:r26:r27     pop.4x d24:d25:d26:d27
        pop.4l r20:r21:r22:r23     pop.4x d20:d21:d22:d23
        pop.4l r16:r17:r18:r19     pop.4x d16:d17:d18:d19
        pop.4l r12:r13:r14:r15     pop.4x d12:d13:d14:d15
        pop.4l r8:r9:r10:r11       pop.4x d8:d9:d10:d11
        pop.4l r4:r5:r6:r7         pop.4x d4:d5:d6:d7
        pop.4l r0:r1:r2:r3         pop.4x d0:d1:d2:d3
```

Notes:

- In a similar way to the context saving routine, replacing all POP.* instructions with the respective POPC.* instructions restores the context of the restored task from its own stack instead of restoring it from the stack of the OS
- Prior to restoring the context registers, the OS should restore the registers that allow it to perform the POP instructions from the right location. If restoring is done with POPC, then first SR2.ASPSEL should be restored (it could be restored efficiently using the CHCF Ra,SR2.ASP instruction). Then the proper SP and TID should be restored
- PC, SR. SR2 and EIDR are restored automatically by the RTE instruction, and thus no additional instruction is required to restore them.

4.10 Cache control instructions

The SC3900FP cache system supports a rich set of control functions that the user can activate on the caches. Control functions can be, for example, pre-fetching operations, coherency operations, and status query operations. The unit managing these operations is named the CME (Cache Management Engine), which is divided into sub-units that are responsible for managing the instruction and data caches, and also includes an interface unit that handles the configuration requests. All cache operations from the core are communicated to the caches via the core data buses (including program cache instructions). The cache instructions are marked with attributes that identify them as cache management operations, and specify the requested operation. The CME holds a set of task channels that the user configures with cache control instructions. Once configured, the CME operates independently while the core continues to run. The CME monitors the respective address buses (program and data) and drives accesses on them from the active task channels using free access slots that are not used by functional core accesses. The high-level connection of the CME in the SC3900FP subsystem is illustrated in Figure 4-10. This figure is just for illustration purposes as the actual connections are more complex.

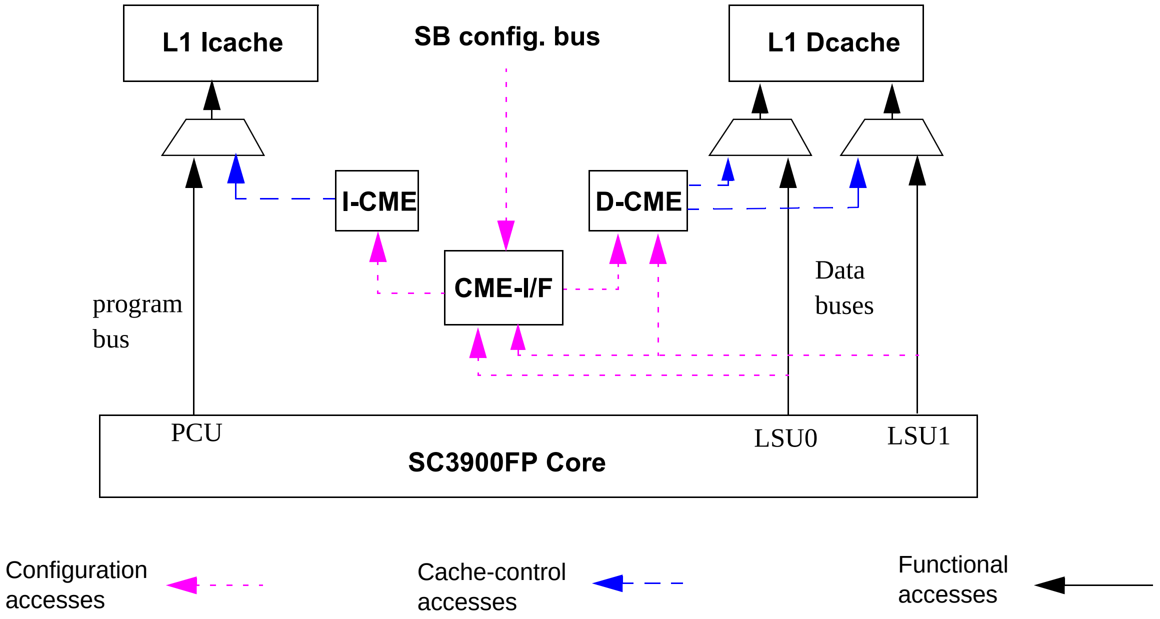


Figure 4-10. General connection of the Cache Management Engine (CME)

Cache control operations in the CME may be configured either by the core, using dedicated cache control instructions, or by an external master issuing accesses to memory mapped CME registers on the Sky-Blue configuration bus, which is used to configure all the subsystem memory mapped registers. This section describes only the core instructions used to perform cache control operations. Some types of cache operations are possible only through the external SB bus. For more information on the CME, see the *SC3900FP FVP Subsystem Reference Manual*.

4.10.1 Types of cache control commands

All cache control instructions are LSU instructions. This also includes instructions that control the program cache, which are configured by data accesses, as seen in Figure 4-10. Most cache control instructions are executed as load operations, and a few as stores operations. Some of them may return status information as the load return data (in which case destination registers are specified in the instruction), while others are dummy loads that do not return data. Cache control commands could be characterized in several ways:

Data/instruction: Instructions that control the Data Cache, or instructions that control the Instruction Cache.

Destination cache: Some instructions affect only the L1 cache, others affect only the L2 cache, and some affect both.These instructions are marked with an L1, L2 or L12 suffix, respectively. The L2 cache is shared for both program and data caches, however selection of the program or data cache instruction is important also for instructions that affect only the L2 cache because of the different MMU translation path.

Block/Granular: Granular cache instructions are instructions that operate architecturally on one “Cache Granule”, which is the minimum data unit on which cache coherency is managed. A Cache Granule is an aligned, contiguous range of 64-bytes. A granular cache instruction operates on the cache without consuming a user-visible CME resource. A Block cache command is a more complex operation that can be defined on a set of cache granules, such as a contiguous memory range, or even a two-dimensional memory range. Block cache commands are allocated a CME channel and may contend between them on the available channels.

Functional type: There are four types of cache instructions: instructions that perform pre-fetching, instructions that help manage the coherency of the caches, instructions for CME control, and cache query instructions. Some of these functional types have variants such as locking during an L2 fetch, and more.

4.10.2 Granular cache instructions

Granular instructions are always accepted for execution by the respective cache, hence do not return a success status. These instructions specify the cache granule they operate on as an address operand of the instruction. The general form of these instructions is as follows:

```
<instruction> <granule address>
```

Granular data cache instructions support only the (Rn) addressing mode.

4.10.3 Block cache commands

Block cache commands configure a CME “channel,” which then starts to operate independently in the background until it generates all the accesses it was configured to perform. The number of channels is limited, and the user usually has no knowledge of how many channels are available for use. Therefore, block cache commands return a status that communicates whether or not the block command was allocated a free channel and was accepted for operation. Note that a success status only signifies successful allocation of a CME channel, not a successful completion of the task that the command had configured the CME to perform.

Block cache commands are specified by the user as a single mnemonic:

```
<command> Ra,Rb,Rn
```

for example:

```
DFETCHM Ra,Rb,Rn
```

where Ra and Rb are source registers that hold the configuration data of the various fields that are required to configure the command, and Rn is the destination register that samples the success status of the command (acceptance or rejection in the CME). <command> is the specific cache operation mnemonic, which informs the CME of the type of the specific action that is required.

Block cache commands are a “meta instruction”, encoded as two LSU instructions and a PCU instruction that are grouped together, as follows:

```
<instruction> Rb,Rn       BCCAS Ra         SYNC.B
```

for example, the DFETCHM command mentioned above is encoded as:

```
DFETCHB Rb,Rn             BCCAS Ra         SYNC.B
```

The BCCAS instruction (Block Cache Command Assist) is a generic instruction that is added in all block cache commands and performs a shared task: drives the second operand (Rb) on the address bus, so that the CME could sample it. The SYNC.B instruction is required to ensure that the cache instructions are not executed speculatively. These three instructions are encoded automatically by the assembler when it processes the single cache command mnemonic written by the user.

Figure 4-11 and Figure 4-12 show the configuration fields in Ra and Rb. There are two configuration options, one for a sequential mode (in which the CME generates sequential memory accesses), and the other is a two-dimensional mode (in which the CME generates accesses to rows of sequential data that are separated in memory, each starting a “stride” apart). The two modes require a different field partition of Ra, which is determined by the TDAG field. See the *SC3900FP FVP Subsystem Reference Manual* for more details.

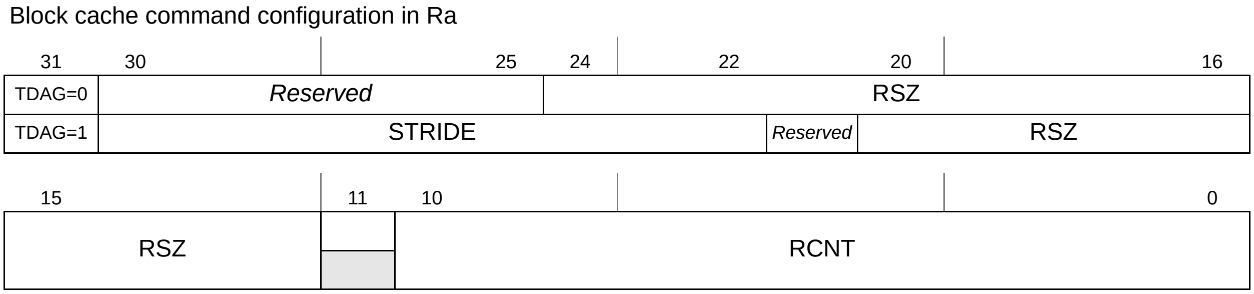


Figure 4-11. Block Cache Command Configuration by Ra (driven by BCCAS)

Block cache command configuration in Rb

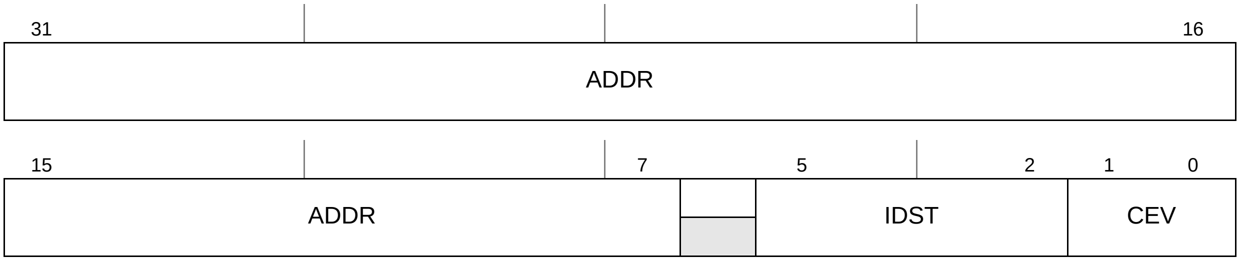


Figure 4-12. Block Cache Command Configuration by Rb

This table provides a detailed description of the function of every field in Ra and Rb.

Table 4-8. Block cache command Ra, Rb field description

Field	Description
Ra[31] TDAG	Two dimensional address generation enable bit. 0 Address generation is strictly sequential (one dimensional). Stride size is forced to 0. 1 Address generation is two dimensional.
Ra[30–22] STRIDE	In case of two dimensional address generation this field represents the 9 most significant bits of the stride size. The 7 least significant bits are always 0 (the stride has 128-byte resolution). This field is ignored for sequential address generation. When valid, the stride should be equal or greater than the row size.
Ra[20–12] or [24–12] RSZ	Represents the most significant portion of the row size: 9 bits for two dimensional mode, or 13 bits of the sequential mode. The 7 least significant bits are always 0 (the row size has 128-byte resolution).
Ra[10–0] RCNT	Number of rows
Rb[31–6] ADDR	Bits [31:7] of the block command start address. Address bits [6:0] are always 0 (it has 128-byte resolution). The address is virtual.
Rb[5–2] IDST	Block command external interrupt destination. Used only if external interrupt completion event is chosen by CEV field below. Index in the doorbell register of MMU.
Rb[1–0] CEV	Block command completion event. Controls special action that is executed after completion of the command. 00 No special action is done on completion. 01 An interrupt to the local EPIC is asserted, following a heavy weight memory barrier. 10 An external interrupt is sent to the destination defined by IDST, following a heavy weight memory barrier. 11 A Heavy weight memory barrier is generated.

The result of a block cache command is updated in the destination register Rn and has the following format:

	31 16
R W	—

	15 8	7 5	4 2	1	0
R W	—	CHANNEL	BER	BRES	0

Figure 4-13. Block Cache Command Result format

This table describes the bit fields.

Table 4-9. Block Cache Command Result description

Field	Description
31–8	Reserved
7–5 CHANNEL	Index of the CME channel allocated for block cache command. Type of the channel (instruction or data) is defined by maintenance instruction’s type. Valid only if BRES is 0.
4–2 BER	Result details 000 The command was successfully allocated to an empty CME channel 001Reserved 010 The command failed to allocate to a CME channel due to lack of space (all available channels are taken by commands). 011 The command failed to allocate to a CME channel because the CME is disabled or full during error or suspend state 100Reserved 10xReserved 110Reserved 111 The command failed to allocate to a CME channel because it is invalid (wrong combination of parameters and values)
1 BRES	Block cache command result 0 The command was successfully allocated 1 The command was not allocated
0	Reserved

4.10.4 Query instructions

The DQUERY.L*x* instructions allow users to receive detailed status information from the MMU and relevant caches regarding a certain address. The instruction has the following format:

```
DQUERY.Lx (Ra),Rm:Rn
```

Ra holds the address that is queried. Rm:Rn are destination operands, which are loaded with the status results from the memory system. The data query instruction has two variants: one querying the L1 cache, and one querying both the L1 and L2 caches (the two variants are collectively described above as L*x*). The Returned status includes the following information:

- Index of the MMU descriptor to which the address belongs
- In case the address would have caused a memory exception in a memory load, an error indication is given (but no exception is generated)

- The hit status in the L1 cache
- In case there is an L1 cache hit, in what Way (in general, enough information to configure a direct access to the direct mapped array for debug purposes)
- Returns the hit and locking status in the L2 cache. This information is provided only for the instruction variant that also queries L2, which takes more cycles relative to the L1-only variant.
- In case there is an L2 cache hit, same type of cache status information as described for L1 above.
- Returns the full physical address of the access (after translation).

The query instructions are neither Block nor Granular. They do not occupy a CME channel, and the core stalls until the information is provided. No memory exception is generated by these instructions. A query access is presented to the memory as a byte-wide access (the minimum width).

Figure 4-15 and Figure 4-14 describe the field organization of the data returned by the QUERY instructions. It is a superset of information returned by the various instructions, for example fields that describe the L2 cache status are not valid for QUERY variants that probe only the L1 cache. The result reflects the current organization of the MMU and L1 + L2 caches. See the *SC3900FP FVP Subsystem Reference Manual* for more details.

Rm result format in Rm:Rn destination operands

31 29	28 25	24	23 22	21 18	17	16
L2B	L2W	L2L	L2S	L1W	SGBH	L1H

15	14	13	12	11 9	8 4	3 0
MER	NC		CSS	MMUE	SDHI	QPA

Figure 4-14. QUERY result format in Rn

Rn result format in Rm:Rn destination operands

31 16
QPA

15 0
QPA

Figure 4-15. QUERY result format in Rm

This table provides a detailed description of the fields of QUERY destination operands.

Table 4-10. QUERY destination Rm:Rn field description

Field	Description
Rn: 31–29 L2B	L2 cache bank Reflects the L2 cache bank for the related query address. Only valid for the L2 query and cacheable address. Not valid in case of error.
Rn: 28–25 L2W	L2 cache way Reflects the L2 cache way for the related query address. Only valid for the L2 query and cacheable address. Not valid in case of error.

Table 4-10. QUERY destination Rm:Rn field description (continued)

Field	Description
Rn[24] L2L	L2 lock status Reflects the L2 cache lock status for the related query address. Only valid for the L2 query and cacheable address. Not valid in case of error.
Rn[23–22] L2S	L2 cache coherency status. Reflects the L2 cache hit and coherency status for the related query address. Value updated only for L2 query and cacheable address. Not valid in case of error. 00 Invalid 01 Shared (includes Shared-Last Reader) 10 Exclusive 11 Modified
Rn[21–18] L1W	L1 cache way Reflects the L1 cache way for the related query address. Not valid in case of error or L2 query and non cacheable address.
Rn[17] SGBH	SGB hit indication Reflects SGB hit indication for the query address. Always 0 for PQUERY, in case of error, or L12 query on a non cacheable address. 0 No hit in SGB 1 Hit in SGB
Rn[16] L1H	L1 cache hit indication Reflects the L1 cache hit indication for the related query address. Not valid in case of error or L2 query and non cacheable address. 0 No hit in L1 cache 1 Hit in L1 cache
Rn[15] MER	L1 external memory error Valid only for memory query. Not valid in case of MMU error or L2 query and non cacheable address. 0 No external memory error in L1. 1 External memory error was detected.
Rn[14] NC	Non cacheable descriptor hit. Not valid in case of MMU error, except for peripheral error. 0 Cacheable descriptor hit. 1 Non cacheable descriptor hit. If L1 hit is also asserted then non cacheable hit error was detected.
Rn[13]	Always 0 for the DQUERY instructions
Rn[12] CSS	Bank 0 indication Set when a query address matches bank 0 address space.Not valid in case of error, except for peripheral error. 0 External access that goes through the L2 cache. 1 Bank 0 access. L2 cache statistics above are meaningless. For DQUERY.L12 peripheral error is generated.
Rn[11–9] MMUE	MMU error. 000 Query has multiple segment descriptor hit error reported by the MMU. 001 Query has segment miss error reported by the MMU. 011 Query has peripheral error reported by the MMU. DQUERY.L12 accesses bank 0 descriptor. 111 Query has no errors reported by MMU.

Table 4-10. QUERY destination Rm:Rn field description (continued)

Field	Description
Rn[8–4] SDHI	Segment descriptor hit index Reflects the segment descriptor hit index for the related query address. Not valid in case of error, except for peripheral error. When MSD is set, this field reflects the higher hit segment descriptor index.
Rn[3–0]: Rm[31–0] QPA	Query physical address Holds the 40-bit physical address related to the last query virtual address. Not valid in case of error or memory query.

For more information on the exact data format that is returned by query instructions, see the *SC3900FP FVP Subsystem Reference Manual*.

4.10.5 Attribute override of cache instructions

Like most other memory accesses, some cache commands could be specified to use alternate attributes:

- Alternate DID (for data cache commands) and PID (for program cache commands)
- STRM. Marking a data pre-fetch instruction or block command with a “streaming” attribute, informing the cache that this data is not expected to be used again after it is loaded to the core
- ITW. Marking a data pre-fetch instruction or block command with an “intent to write” attribute, informing the cache that the core will modify this data soon

For details of the data access attribute override, see Section 4.11, “Instruction attribute override.”

4.10.6 Cache command summary

This table lists the instruction cache instructions, their variants and functionality.

Table 4-11. Summary of cache commands and their functionality

Functional group	mnemonic	type	Variants	Mem. type	Attribute override	Memory exception	Accept?	Specul-ation	Functionality
Query	DQUERY.Lx (Ra),Rn:Rm	Instruction	Lx = L1, L12	LD	AID	no	yes	yes	Queries the MMU, L1 data cache and optionally the L2 cache

Table 4-11. Summary of cache commands and their functionality (continued)

Functional group	mnemonic	type	Variants	Mem. type	Attribute override	Memory exception	Accept?	Specul-ation	Functionality
Pre-fetch & locking	DFETCH.Lx (Rn)	instruction	Lx = L2,L12	LD, no data	AID, ITW	config	yes	yes	Fetches the specified data granule to the cache (L2, and optionally L1 data cache)
	DFETCHM.Lx Ra,Rb,Rn	meta-instruction: [DFETCHB.Lx Rb,Rn BCCAS Ra SYNC.B]	Lx = L2,L12	LD	AID, ITW	no	maybe	no	Fetches the specified data block to the cache (L2, and optionally L1 data cache)
	DFETCHM.LCK.L2 Ra,Rb,Rn	meta-instruction: [DFETCHB.LCK.L2 Rb,Rn BCCAS Ra SYNC.B]	—	LD	AID, ITW	no	maybe	no	Fetches the specified data block to the cache (L2, and optionally L1 data cache), and lock the data in L2
	DUNLOCKM.L2 Ra,Rb,Rn	meta-instruction: [DUNLOCKB.L2 Rb,Rn BCCAS Ra SYNC.B]	—	LD	AID	no	maybe	no	Unlocks the specified data block lines in the L2 cache
	PFETCHM.Lx Ra,Rb,Rn	meta-instruction: [PFETCHB.Lx Rb,Rn BCCAS Ra SYNC.B]	Lx = L2,L12	LD	AID	no	maybe	no	Fetches the specified instruction block to the cache (L2, and optionally L1 data cache)
	PFETCHM.LCK.L2 Ra,Rb,Rn	meta-instruction: [PFETCHB.LCK.L2 Rb,Rn BCCAS Ra SYNC.B]	—	LD	AID	no	maybe	no	Fetches the specified instruction block to the cache (L2, and optionally L1 data cache), and lock the data in L2
	PUNLOCKB.L2 Ra,Rb,Rn	meta-instruction: [PUNLOCKB.L2 Rb,Rn BCCAS Ra SYNC.B]	—	LD	AID	no	maybe	no	Unlocks the specified instruction block lines in the L2 cache

Table 4-11. Summary of cache commands and their functionality (continued)

Functional group	mnemonic	type	Variants	Mem. type	Attribute override	Memory exception	Accept?	Specul-ation	Functionality
Coherency	DCM.FLUSH.Lx (Rn)	instruction	Lx = L1, L12	ST, no data	AID	yes	yes	no	Flush the specified data granule from L1 or both L1 and L2 caches (with SGB flush)
	DCMM.FLUSH.L12 Ra,Rb,Rn	meta-instruction: [DCMB.FLUSH.L12 Rb,Rn BCCAS Ra SYNC.B]	—	LD	AID	no	maybe	no	Flush the specified data block from L1 and L2 caches (SGB is not flushed)
	DCM.SYNC.L12 (Rn)	instruction	—	ST, no data	AID	yes	yes	no	Synchronize a data granule from L1 with L2 with the main memory.
	DCMM.SYNC.Lx Ra,Rb,Rn	meta-instruction: [DCMB.SYNC.Lx Rb,Rn BCCAS Ra SYNC.B]	Lx = L1,L2	LD	AID	no	maybe	no	Synchronize a data block either from L1 with L2 (SGB flush), or L2 with main memory (without SGB flush).
	DCM.INVAL.Lx (Rn)	instruction	Lx = L1, L12	ST, no data	AID	yes	yes	no	Invalidate the specified data granule from L1 or both L1 and L2 caches.
	DCMM.INVAL.Lx Ra,Rb,Rn	meta-instruction: [DCMB.INVAL.Lx Rb,Rn BCCAS Ra SYNC.B]	Lx = L1, L12	LD	AID	no	maybe	no	Invalidate the specified data block from L1 and L2 caches. Both do not flush the SGB
	PCMM.INVAL.Lx Ra,Rb,Rn	meta-instruction: [PCMB.INVAL.Lx Rb,Rn BCCAS Ra SYNC.B]	Lx = L1, L12	LD	AID	no	maybe	no	Invalidate the specified program block from L1 or both L1 and L2 caches

Table 4-11. Summary of cache commands and their functionality (continued)

Functional group	mnemonic	type	Variants	Mem. type	Attribute override	Memory exception	Accept?	Specul-ation	Functionality
CME control	CCMDM.SUSP	meta-instruction: [CCMD.SUSP SYNC.B]	—	LD, no data	—	N/A	yes	no	Suspend all active channels in the CME
	CCMDM.RESM	meta-instruction: [CCMD.RESM SYNC.B]	—	LD, no data	—	N/A	yes	no	Resume all suspended channels in the CME

Comments on the rows of Table 4-11:

- Memory access type:
  - — LD: a load access, that returns data
  - — LD, no data: a load access, where no data is sampled by the core
  - — ST, no data: a store access, where no data is written to memory
- Attribute override: see Section 4.11, “Instruction attribute override”
- Configurable MMU exception: the MMU could be configured to generate a memory exception for DFETCH granular instructions
- Acceptance:
  - — “yes” means that the instruction is always accepted for execution
  - — “maybe” means that the instruction may not always be accepted for execution, usually the return status should be consulted
- Speculation:
  - — “no”: the instruction is never speculatively executed due to its nature and restrictions on it (for example if it must be grouped with a SYNC.B instruction)
  - — “yes”: the instruction may be speculatively executed

4.10.7 Coding restrictions on cache control instructions

There are several coding restrictions which apply when using cache control instructions. These are described in detail in Rule A.9 in Chapter 8, “Programming Rules and Guidelines.” The main ones are as follows:

- Only one cache instruction can be specified per VLES
- Cache instructions cannot be predicated

- Most cache instructions cannot be grouped with another memory access in parallel
- Block cache commands must be specified as meta instructions, or be grouped with the exact set of instructions that is encoded by the meta-instruction

4.11 Instruction attribute override

When a data access is issued on the bus, it inherits several attributes that are not specified as part of the instruction, such as the Data ID (DID). In other cases, the memory system supports adding side attributes with the access that allows the user to specify variations on the behavior of the access in the memory system. The instruction override feature allows the user to specify, per VLES, alternate attributes for accesses that are issued on the data buses (some program cache instructions can also be modified this way, see Section 4.10, “Cache control instructions”).

The Attribute Override feature is activated by grouping an IPU instruction with regular LSU instructions that access memory. These instructions are as follows:

- ACATOR.P #u6,#u4
- ACATOR #u6

Several programming rules apply on the use of ACATOR, see Rule A.9 in Chapter 8, “Programming Rules and Guidelines”. In most cases only one LSU instruction that is affected by ACATOR is allowed. The user has two ways to specify these instructions: via the ACATOR instruction, where each bit in an immediate field controls a feature to override, or by directly specifying a mnemonic per feature. The assembler generates the appropriate ACATOR instruction as a result.

Examples of cases where DID/PID attribute override can be used are as follows:

- The OS can perform accesses to user memory using the active MMU configurations, which include a task-specific DID.

Table 4-12 lists the attributes that could be overridden with this mechanism.

Table 4-12. Instruction Attribute Override options

Attribute	Default setting	Override setting	ACATOR bit	Allowed for...	Comments
Data task ID	TID.DID	TID.AID	u4[1]	• Load and store instructions • Supporting data cache instructions, see Table 4-11	—
Program task ID	TID.PID	TID.AID	u4[1]	Supporting instruction cache instructions, see Table 4-11	—

Table 4-12. Instruction Attribute Override options (continued)

Attribute	Default setting	Override setting	ACATOR bit	Allowed for...	Comments
LRU cache state	The access is “most recently used.”	The access sets the cache status as “least recently used”	u6[1:0]	Load and store instructions	Used for data streaming: data that is loaded for short term usage, or stored and not expected to be used so does not need to linger in the cache.
Intent to write	Do not inform	Inform the coherency domain that this data is about to be modified	u6[2]	Supporting data cache pre-fetch instructions, see Table 4-11	Used when the loaded data is intended to be written soon. Communicates to the coherency domain that this data is about to be in the “Modified” MESI+L state.

Note that although, in most cases, the AID field is replacing the DID field (for load, store and data cache commands), the AID field can be used to specify the alternate PID for program cache instructions. This cannot be done to other program-related instructions.

Note that the group of memory access instructions that have the “C” tag in their base mnemonic perform implicit attribute override for the DID attribute. The instructions are as follows:

- PUSHC, POPC: use the DID, regardless of the state of SR2.EXP. See Table 4-1.

4.12 Semaphore and atomic operations support

The SC3900FP supports atomic operations and semaphores using a reservation mechanism.

The reservation mechanism enables the user to load data from a shared memory space, updating it in any needed manner, and storing it back, with guaranteed atomic behavior (that is, guaranteeing that the destination memory was not updated by any other master between the load and the store). This mechanism enables the user to implement simple and efficient semaphores in any supported memory location and to perform more advanced atomic operations as well.

4.12.1 The reservation mechanism

A load instruction can be tagged as a “load reserve,” meaning that the memory system reserves this address, and monitors transactions to it by any master in the system with the help of a special hardware module called the reservation station, which is supported in the SoC.

Data that is loaded by this instruction to the SC3900FP core can be updated and/or manipulated by any sequence of the SC3900FP instruction set.

The following store is then tagged as “store conditional.” This store is tested by the hardware reservation station, and it is allowed to change the destination memory only if the reservation did not fail. The reservation operation is guaranteed to fail if a successful write transaction by any master was completed between the load reserve transaction and the current store conditional transaction. Note that reservation may fail even if such a write did not occur due to resource conflicts at the reservation station, exceptions and implementation issues.

Reservation success or failure during the store conditional instruction is reported by the reservation station to the core and sampled into a new bit in the SR named RF (Reservation Failed). A JMPRF instruction enables an efficient test on the SR.RF bit value.

4.12.2 SR and ISA support

4.12.2.1 Reservation failed bit in the SR

The SR.RF (Reservation Failed) bit in the SR is implicitly written by a conditional store, and is implicitly read by the JMPRF instruction. It can also be accessed by explicit instructions that access the SR. For example, the instruction BMTSTA.S #$2000,SR.L,P2 sets SR.P2, according to the value of SR.RF.

4.12.2.2 Reservation and conditional attribute

The SC3900FP has an attribute instruction (in IPU) that is grouped with a general purpose load or a store instruction, thus marking the load as “load reserve” and the store as “store conditional.” This attribute is marked in the assembly code as LDRSTC (Load Reserve/Store Conditional), which should precede the load or store instruction.

For correct operation of the semaphore access, some restrictions should be maintained. See Rule A.9 in Chapter 8, “Programming Rules and Guidelines.” The following are included among these restrictions:

- A VLES with LDRSTC attribute can have only a single memory transaction.
- The load reserve and store conditional instructions should access the same address with the same transaction width.
- Load reserve and store conditional address must be aligned to the transaction width (no support for non-aligned semaphore accesses).

4.12.2.3 Jump On Reservation Failed

The JMPRF instruction enables to efficiently test the success of a previous store conditional operation. This instruction jumps to the supplied address if SR.RF bit is set. The JMPNF instruction is not BTB-able but can be speculatively executed, that is, when SR.RF is cleared, the instruction does not jump and has one cycle latency. When SR.RF is set, the instruction jumps to the label and takes 12 cycles.

There is a four cycle interlock between a store conditional instruction and a JMPRF instruction.

4.12.3 Usage of the reservation mechanism

The following sections provide examples that illustrate the use of the reservation mechanism in order to implement atomic operations.

4.12.3.1 Semaphore set

In order to take a semaphore the software should load the semaphore from memory, test the value (to see if the semaphore is currently free) and store the updated value back to memory, thereby validating that no other master has written to this location between the load and the store.

The following code example sets a bit in a word and tests if it was set to begin with, and retries until the semaphore is taken. Note BRA.NOBTB in parallel with the conditional store - cannot be a BTB-able COF as restricted in programming rule Rule A.9:

```
_retry:                            ; The semaphore is LSBit of a word located at (r5)
        sync.b ldrstc ld.w (r5),d7 ; Load and reserve semaphore word
      [ cmp.ne #0,d7,p0            ; Test if semaphore is already taken
        or.l #1,d7]                ; Set the semaphore bit (do nothing if already set)
      [ if.p0 bra.nobtb _retry     ; Retry if semaphore was already taken
        ldrstc st.w d7,(r5)]       ; Store conditional the set semaphore back
        jmprf _retry               ; Retry if reservation failed
        .
        .                          ; Semaphore is now taken
```

4.12.3.2 Shared counter

In this example, the software should read a shared counter (that can be updated by other masters or tasks), increment it and write it back atomically (without any other master writing in between).

A standard semaphore based solution requires allocation of the counter *and* of a semaphore. Each time the counter has to be updated the software needs to set the semaphore, update the counter and release the semaphore.

The reservation mechanism enables the user to update of the counter without a separate semaphore. The counter is loaded with reservation, updated and written conditionally, assuring that no other master or task updated it in between. The example code is as follows:

```
_retry:                                    ; Counter is a long word located at (r5)
        sync.b ldrstc ld.l (r5),d7         ; Load and reserve semaphore word
        add.x #1,d7                        ; Increment the counter value
        ldrstc st.l d7,(r5)                ; Write conditional the updated counter back
        jmprf _retry                       ; Retry if reservation failed
```

4.12.3.3 Update a Buffer Descriptor

In this example, a separate entity (either software or hardware) is working on 32 byte buffer descriptors. Each descriptor has a busy bit. In order to write a new descriptor, the busy bit should be tested and, if set, the software should jump to the next buffer location and retry. The following code shows an efficient implementation based on a wide memory transaction with reservation support:

```
; d0:d1:d2:d3 holds the new required buffer descriptor (with busy bit already set)
; d4:d5:d6:d7 are used to read the current buffer descriptor and test the busy bit
; r5 is pointing to the current buffer descriptor

_retry:
        sync.b ldrstc ld.4l (r5),d4:d5:d6:d7; Load and reserve current BD
        tstbm.s.l #$8000_0000,d4,p0         ; Test if busy bit is set
      [ if.p0 adda #32,r5                   ; Advanced to next BD if current BD is busy
        if.p0 bra _retry]                   ; Retry if current BD is busy
        ldrstc st.4l d0:d1:d2:d3,(r5)       ; Write conditional the updated BD
        jmprf _retry                        ; Retry if reservation failed
```

4.12.4 Support of legacy semaphore instructions

In the previous StarCore architecture, there was a single instruction, Bit-Mask Test and Set (BMTSET.W), that provided support for simple hardware semaphores. This instruction, along with a following branch to retry, is replaced in SC3900FP with a code sequence similar to the one shown in Section 4.12.3.1, “Semaphore set.”

Although the code of simple semaphore handling is a bit longer, performance is similar (legacy BMTSET.W takes around 16 cycles to execute, without reservation failed reception delay). Note that, in more complex use cases of the reservation mechanism (such as Section 4.12.3.2, “Shared counter,” and Section 4.12.3.3, “Update a Buffer Descriptor”), SC3900FP support is much more efficient.

4.13 Memory barriers

A memory barrier instruction is an instruction the user should add in the code in order to ensure the correct semantic behavior of load and store instructions in cases that require enforcing execution order, and that access non-standard memories.

Barriers are required in several cases, which fall under the following general categories:

- There is a need to ensure that data written from the core has reached its destination, in preparation for allowing another processor to take it. The barrier should be executed before communicating to the other processor that the data is ready
- Writing a configuration register in some peripheral module may require waiting for the write to take effect before continuing, such as clearing the cause of an interrupt before RTE (so that the interrupt will not be re-executed)
- Between a load and a store to an address of non-standard memory, where the data that is read is not necessarily equal to the data that was written. For example, some peripheral control registers which implement a “write 1 to clear” logic, some types of decorated accesses, etc. The barrier ensures that the load is done from the destination memory and not from an intermediate buffer

The SC3900 supports several types of barriers, tailored for different cases, in order to allow the user to minimize the number of stall cycles that are caused by the barrier. The user is advised not to use the strongest barriers uniformly “to be on the safe side” because it may degrade the performance of the application.

All barrier instructions are LSU instructions. The memory barrier instructions only perform actions in the memory system. In general, the barrier instruction is placed in a VLES without parallel memory accesses, and ensures the semantic order between the relevant accesses from the previous VLESes to those in the following VLESes. In some cases, the barrier should be supplemented by core pipe synchronization instructions that make sure that the core pipeline is also synchronized. In such a case, there is a meta-instruction mnemonic that causes the assembler to generate the instruction pair. This table lists the memory barrier instructions of the SC3900FP.

Table 4-13. SC3900FP memory barrier instructions

Mnemonic	Type	Access type	Functionality
DBARS.IBSS.L1	instruction	ST, no data	Store-to-store barrier. The barrier ensures that the two stores will not be packed into a single access in the SGB, and it also drains all prior entries in the SGB. This barrier does not stall the core.
DBARS.IBSS.L12	instruction	ST, no data	Store-to-store barrier, between two cacheable stores. This barrier ensures that the first store is updated in the coherency domain before the second one updates the L2 cache. On the way it clears the SGB from all previous entries. This barrier does not stall the core.
DBARM.IBSL	meta-instruction: [DBAR.IBSL SYNC.B]	LD, no data	Store-to-load barrier, between a cacheable store and a cacheable load. This barrier ensures that the previous stores are drained from the SGB before the load is issued
DBARM.IBLL	meta-instruction: [DBAR.IBLL SYNC.B]	LD, no data	Load-to-load barrier, between two cacheable loads. This barrier ensures that the first load updated the coherency domain before the second load is sent out. All previous load misses are also serviced before the second load is issued.
DBARM.L1SYNC	meta-instruction: [DBAR.L1SYNC SYNC.B]	LD, no data	L1 synchronization: hold the core until the L1 data cache is idle, and all open SkyBlue accesses are closed
DBARM.SCFG	meta-instruction: [DBAR.SCFG SYNC.B]	LD, no data	Store configuration: assert hold until SGB is clear and the cluster memory mapped registers were written. The barrier is used after writes to subsystem or cluster registers, making sure the write finished before carrying on
DBARM.EIEIO	meta-instruction: [DBAR.EIEIO SYNC.B]	LD, no data	This barrier is issued between two non-cacheable stores, ensuring the first one was written before the second store is issued. No load following the first store may bypass it.
DBARM.HWSYNC	meta-instruction: [DBAR.HWSYNC SYNC.B]	LD, no data	Heavy-wight sync, without flushing the pre-fetch queue. This barrier is used between all other access combinations not mentioned above. The second access is held until the fabric reports that the first access had reached its final destination
DQSYNCM	meta-instruction: [DQSYNC SYNC.B]	LD, no data	Global data cache invalidation

4.14 Other special access types

4.14.1 Access decoration

The CoreNet fabric supports the ability to add a 4-bit “decoration” attribute bus along with an access. The destination memory or peripheral may implement logic that reacts to specific decoration values in a way that is specific to that peripheral. See the specification of the SoC for more information on the units and addresses that support decorated accesses and the semantics of these accesses.

The user can decorate any load (LD) or store (ST) access by applying the DECOR IPU instruction in parallel with it. The 4-bit immediate operand of the DECOR instruction determines the 4-bit value driven on the decoration bus. For example:

```
        DECOR #5  ST.L d5,(r0)
```

Decoration may be done in principle on both load and store instructions; however, the implemented functionality is usually specific for one type of access, as defined in the SoC. Note that decorated loads in the current definition are destructive, meaning that the load may change state at the load address. Hence, the memory region that support decorated accesses should be defined as “guarded” in the MMU, and load accesses to it should conform to the requirements of Section 4.14.4, “Destructive loads.” MMU exceptions are generated normally for decorated accesses.

4.14.2 Notify accesses

“NOTIFY (Rn)” is a specialized store instruction that sends no data, that is intended to generate an event at the receiving end (address location specified by Rn). Such events could be a “doorbell event,” that is, an assertion of an interrupt at the receiving end, with no acknowledgement to the sending side. Acknowledgement of the event may be implemented at a higher software level, for example, by the code servicing the doorbell interrupt. The nature of the action performed at the receiving end is SoC-specific, and requires hardware support at the receiving end. See the SoC specification for more information on the units and addresses that support NOTIFY accesses and the semantics of these accesses.

Notify accesses must be decorated, by definition. Hence NOTIFY should be grouped with DECOR, which supplements the decoration value. For more information, see Section 4.14.1, “Access decoration.”

Memory exceptions are generated for NOTIFY accesses as for normal store instructions.

4.14.3 Message accesses

“MSGSND Rn” is a specialized store instruction that sends no data, driving the value of Rn on the address bus. It is used for sending a message to units that listen to this type of message in the same coherency domain. The contents of the message, as well as the ID of the addressee list of the message (all or a subset of the units that support it), are encoded on the address bus itself.

The message sending feature allows the user to define an independent low-bandwidth connectivity fabric on top of the CoreNet fabric, independent of general memory addressing. Supporting units listen on the bus, and respond only to these messages using the address bus as a means to pass the information payload of the message. There are two broad categories of messages: a normal message, directed at a single or group of receivers, and a broadcast message, addressed to all blocks that have the ability to receive messages.

Because the message mechanism does not use the address bus as proper addresses, the MMU ignores these accesses (that is, no descriptor match, access rights checks and translation). These accesses are treated as non-cacheable by nature.

The functionality of the MSGSND instruction is SoC-dependent, and requires dedicated hardware support by the units that respond to it.

The SC3900FP sub-system also supports reception of messages that are generated by other cores in the SoC. Upon reception of a broadcast message or a message destined to this core, a dedicated interrupt is generated. See the *SC3900FP FVP Subsystem Reference Manual* for more details.

4.14.4 Destructive loads

“Destructive load” is a load from a memory location, which changes some state in the hardware unit to which this memory location is associated. Examples of such hardware units could be as follows:

- A FIFO that is mapped to a single memory address. A load from that address returns the first entry from the queue and increments the internal pointer to the next entry
- A memory-mapped serial communication device: A load from the “receive” memory-mapped register returns the data just received, and triggers the fetching of the next data packet; usually some status indications are updated as well
- Decorated load accesses, per the CoreNet bus definition. Such load accesses could, for example, increment a counter that is associated in hardware with the address.

Memory locations from which loading may be destructive cannot be accessed speculatively, and therefore require special treatment, as follows:

- The memory segment, which includes the destructive address must be configured as “guarded” in the MMU.
- The VLES that includes a load from a destructive address must be marked as such explicitly in the assembly code, by grouping it with a dedicated SYNC.B.DL instruction. Several other restrictions apply, see Rule A.9.

Chapter 5
Program Control

This chapter describes the program control features of the SC3900FP DSP core by first providing an overview of the architectural features and then describing the main micro-architectural features (such as buffers, speculation, and prediction). This chapter intends to help the software-oriented reader understand both the cycle count and the performance numbers of this architecture. Please note that this chapter does not cover the details of the micro-architecture.

5.1 Overview of the PCU execution unit

The SC3900FP implements all program control instructions in a standalone execution unit called the PCU. As a result, unlike legacy StarCore architectures, SC3900FP does not require an AGU execution unit slot in most cases (that is, most PCU instructions can be grouped with two load/store instructions, or other LSU instructions, in addition to an IPU instruction). All PCU instructions that either use load/store unit resources (for example, subroutine call/return and RTE), or that access the AGU register file (for example, JMP Rn), use both a PCU slot and an LSU slot. Few program control-related instructions (for example, DOEN.n) use only an AGU execution slot.

5.2 Understanding the predication model

The SC3900FP core has a predication model with six physical symmetrical predicate bits P0÷P5. The predicate bits reside in the SR register (SR.P0, SR.P1…SR.P5).

The T-bit of previous StarCore architectures is mapped into the P0 predicate bit (SR.P0 is located at the same bit as legacy SR.T).

The physical predicate bits are common to all of the execution units. They can be generated and manipulated by both AGU and DALU instructions, and can be used to conditionally execute instructions in all of the execution units.

As in previous StarCore generations, a predicate bit has to be set according to a condition before it can be used to control a conditional change of flow or predicate an instruction.

The core supports hundreds of virtual predication bits. Most of the virtual P-bits are stored in a user-allocated Rn register, or in several registers, and six active P-bits are stored in the physical predicate bits (in the SR). Efficient instructions can select the six active predicate bits out of the full set[1]. SC3900FP does not have special flags that are calculated implicitly and can be used to predicate instructions like a zero bit and "jump on zero" instruction, excluding special cases such as jump if reservation failed. The predicate mechanism is a very flexible, orthogonal, and powerful means to control conditional execution and flow control.

[1] for readability, in most places the physical predicate bits are referred to as the predicate bits.

5.2.1 Setting the value of predicate bits

5.2.1.1 Calculate the value of the physical predicate bits

Both AGU and DALU instructions are able to calculate predicate bit values. The SC3900FP implements a fully orthogonal set of such instructions:

- Full condition set
  - All compare instructions (“`cmpa`”, “`cmp`” and “`cmpd`”) allow a higher-level compiler such as a C compiler to map a complete set of conditions: compare equal (“`eq`”) and the opposite compare not-equal (“`ne`”), compare greater than (“`gt`”) and the opposite compare less than or equal (“`le`”), compare greater than or equal (“`ge`”) and the opposite compare less than (“`lt`”). In some cases not all conditions exist, because the compiler can implement these conditions with the existing instructions, for example there is no encoding for “`cmp.le.x d0,d1,p0`”, since “`cmp.gt.x d1,d0,p0`” is an equivalent.
  - All the compare instructions with the variant conditions can write to each of the six physical predicate bits.
- Full set of destinations
  - Write to a single predicate bit: P0, P1, P2, P3, P4 or P5.
  - Write to a predicate pair: P0:P1, P1:P2, P2:P3, P3:P4, or P4:P5.
    Write to predicate pair is done with opposite values, for example, the instruction “`cmp.gt.x d0,d1,p0:p1`” is equivalent to “`[cmp.gt.x d0,d1,p0 cmp.le.x d0,d1,p1]`” but is taking only one execution slot.

As in previous StarCore architectures, updating the same predicate bit by several instructions in the same VLES is allowed under certain conditions, with the following semantics:

- When several instructions updating a single predicate are grouped, the destination predicate will be the logic OR of the output of each instruction
- When several instructions updating a predicate pair are grouped, the first predicate (lower index) of the pair will be the logic OR of the first predicate output of each instruction. The second predicate will be the inverse of the value of the first predicate as calculated.

In case the predicate updating instructions are themselves predicated, somewhat different semantics apply, see Section 5.2.3, “Understanding instruction predication”.

Several programming rules apply to grouping instructions updating the same predicates, among them:

- All the instructions that set the same predicate bit must execute in either AGU or DALU execution units.
- All instructions updating the same predicate must all update a single predicate destination, or the same destination pair.

For more details, see Rule G.G.4.

Setting different predication bits by different instructions in the same VLES is allowed with no restrictions. Note that all AGU and DALU execution units can set predicate bits values; and, thus, one can group up to seven instructions that set predicate bits values (into fewer predicate bits) all in the same VLES.

5.2.1.2 Preset the value of the physical predicate bits

An IPU instruction allows the user to preset the value of several predicate bits (1÷6) in a single instruction, as shown in the following example.

```
pinita #$0c,#$00            ; p2:p3 = 0

pinita #$3c,#$24            ; p5=1, p4=0, p3=0, p2=1
```

5.2.2 Manipulating the value of physical predicate bits

A flexible instruction allows to calculate of a value of a single predicate bit, according to any logic function of up to three predicate bits. There are two sets of this instruction: PCALCA, which is an IPU instruction, and PCALC, which is a DALU instruction. The PCALCA and PCALC instructions have the same functionality; however, as they belong to different execution units, there is a difference in the interlocks that are generated (similar to the difference between ADD and ADDA execution, although working on the same resources).

Any logic function of up to 3 input predicate values can be implemented using these instructions, for example:

```
pcalca ((p2 &! p4) ^ p5),p3          ; p3 = (p2 &! p4) ^ p5 with IPU interlocks

pcalc ((p0 | p1) &! (p3 ^ p0)),p1    ; p1 = (p0 | p1) &! (p3 ^ p0) with DALU interlocks

pcalc (!p2),p3                       ; p3 = !p2 (inverse p2) with DALU interlocks

pcalca p4,p2                         ; transfer (copy) p4 into p2 with IPU interlocks
```

Another flavor of these instruction can write a predicate pair. For example:

```
pcalca (p2 ^ p5),p3:p4               ; p3 = p2 ^ p5, p4 = !(p2 ^ p5) with IPU interlocks

pcalc p4,p2:p3                       ; p2 = p4, p3 = !p4 with DALU interlocks
```

Note that a “low level” syntax, explicitly marking the immediate fields needed to encode the “high level” syntax is supported by the assembler as well. The logic function is specified with a 3-bit field that specifies the truth table for each combination of 3 input predicates. For more information, see the description of these instructions in Appendix C, “Instruction Set”.

The following rule applies to the usage of these instructions:

- Writing to a predicate bit that is written by a PCALC or PCALCA by any other instruction in the VLES is not allowed.

For more details see Rule G.G.4.

5.2.3 Understanding instruction predication

Almost all instructions are able to execute conditionally (predicated), according to one of the physical predicate bits (few control instructions are implemented with reduced predication or no predication). Each instruction in a VLES can be predicated individually, without encoding constraints related to predication of other instructions. Note that predication by some of the predicate bits requires a prefix word that enlarges the code size.

The SC3900FP enables predication on positive predicate values; if the predicate bit value is set, an instruction is executed. A false condition can be achieved by first using the fully orthogonal predicate generation instruction, as well as the powerful predicates manipulation instructions, to generate the inverse value in the predicate. Good support of if-then-else implementation can be achieved by writing the true condition and false condition to a predicate pair (for more information on predicate pair generation, see Section 5.2.1.1, "Calculate the value of the physical predicate bits"), and conditioning each instruction in the VLES with the correct predicate.

NOTE

When two predicated instructions write to the same resource, it is the user's responsibility to make sure that predicate values are exclusive (that is, the predicate values cannot be set together), otherwise, the resulting value of that resource (Rn or Dn register) is undefined. For more details see Rule G.G.4a and Rule G.G.4b in Chapter 8, "Programming Rules and Guidelines."

Instructions that preset and/or manipulate the value of a predicate can be predicated themselves by either regular predication or by the "if else clear" predication. Regular predication means that if the predication condition is not met, the destination predicate bits remain unchanged. In "if else clear" predication, if the predication condition is not met, the destination predicate bits are cleared (assuring exclusive value of the result). For example, the instruction `if.p0.ec cmpa.eq r1,r2,p2:p3` gives the following result, regardless of the previous values of p2 and p3.

Table 5-1. Truth table for instruction if.p0.ec cmpa.eq r1,r2,p2:p3

P0	R1=R2	P2	P3
0	x	0	0
1	0	0	1
1	1	1	0

The combination of predicate generation with the "if else clear" condition and exclusive predication to the same resource enables very efficient coding in cases such as the following:

```
if (a = 0) {                        ; a - r0, b - r1, c - r2
    if (b > 0) {                    cmpa.eq #0,r0,p0              ; a = 0
         c += b;                    if.p0.ec cmpa.lt #0,r0,p1:p2  ; b > 0
    } else {                        [
         c--;                       if.p1 adda r1,r0,r0           ; c += b;
    }                               if.p2 suba #1,r0,r0           ; c--
}                                   ]
```

When grouping several predicate updating instructions that update two destination predicates, that are themselves predicated, the following semantics apply:

- In regular predication:
  - if the controlling predicate is true, the result is like the non-predicated case, i.e.: the lower index destination predicate is the OR of the respective destinations from each instruction, and the higher index predicate is the inverse value
  - If the controlling predicate is false, the two destination predicates are not changed
- In "if else clear" predication:

- — If the controlling predicate is true, the result is like the non-predicated case
- — If the controlling predicate is false, both destination predicates are cleared

In all cases of grouping several predicate updating instructions that are themselves predicated, the controlling predicate must be the same and on the same type (ex. “if else clear”) for all instructions. For more details see Rule G.G.4a and Rule G.G.4b in Chapter 8, “Programming Rules and Guidelines.”

5.2.4 Understanding the virtual predicate bits model

In many cases, more than six predicate bits are required simultaneously by the application (or by the compiler). To handle more than six predicate bits, the SC3900FP architecture supports many virtual predicate bits. The virtual predicate bits are allocated in two locations:

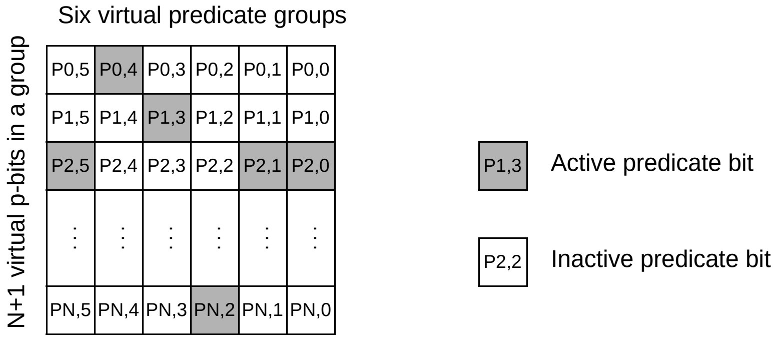
- Up to six predicate bits are the active bits. They reside in the physical predicate bits in SR register and can be manipulated and be predicated upon.
- The rest of the predicate bits reside in allocated Rn register or registers.

Each Rn register can store up to 18 inactive predicate bits[1]. A special swap instruction can exchange the active predicate bits that reside in the SR register with the inactive SR bits that reside in any Rn register, and vice versa.

5.2.4.1 A programmer’s view of virtual predicate bits

The programmer can allocate six groups of virtual predicates, each of which contain up to N predicate bits. N is calculated by $N = 1 + 3 \times k$, where k is the number of Rn registers allocated for inactive virtual predicate bits. For example, if up to 24 virtual predicates are needed, a single Rn register should be allocated; and if 42 virtual predicates are needed, two Rn registers should be allocated (as can be seen in the following explanation, usage of a single register and up to 24 virtual predicate bits is more efficient). In each of the six predicate groups, one of the N predicate bits can be set to be active, allowing six active predicate groups.

The virtual predicate bits are organized in a 2-D array, as shown in the following diagram:



P0,5	P0,4	P0,3	P0,2	P0,1	P0,0
P1,5	P1,4	P1,3	P1,2	P1,1	P1,0
P2,5	P2,4	P2,3	P2,2	P2,1	P2,0
⋮	⋮	⋮	⋮	⋮	⋮
PN,5	PN,4	PN,3	PN,2	PN,1	PN,0

Figure 5-1. Organization of virtual predicate bits in a 2-D array

[1] Theoretically, if all Rn registers are used to hold predicates, the machine can support 6 + 32 * 18 = 582. Obviously, in actual code, only a few Rn registers will be allocated as inactive virtual predicate bits holders. However, R registers that hold predicates may be stored in memory, allowing, in principle, an unlimited number of virtual predicates

Each predicate bit is marked by two numbers, “Pi,j.” The “i” number is the virtual set (out of N+1 sets); the “j” number is the virtual group (out of six groups).

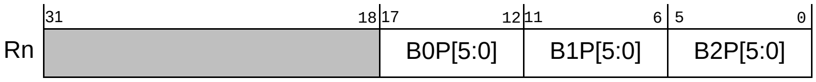
The active predicate bit in each of the six groups can be changed by an IPU instruction. All six active bits can be replaced as long as the new active bits currently reside in the same Rn register (guaranteed by definition in the case in which no more than 24 virtual predicate bits are required).

Note that the above programmers view of virtual predicates is not mandatory. Other predicate bits handling schemes can be implemented by utilizing the same resources and instructions; for example, a six predicate bits model with fast full or partial spill and restore into Rn register using the same swap Pn bit instruction. For information on actual implementation, see Section 5.2.4.2, “How the SC3900FP implements virtual predicate bits.”

5.2.4.2 How the SC3900FP implements virtual predicate bits

As mentioned above, the SC3900FP supports six physical predicate bits, which are implemented in the SR register. Each allocated Rn can hold three banks of six predicate bits stored in the lower 18 bit portion of the register. Dedicated instructions enable swapping of Pn bits between the six physical predicate bits and the corresponding bits in one of the three banks that are stored in an Rn register.

The structure of predicate bits information inside an Rn register is as follows:



BnP[i] - Predicate bit in group i and bank n

Figure 5-2. Structure of predicate bits information inside an Rn register

The instruction PRDA.SWP controls the swapping of predicate bits in one of the six groups between the physical predicate bit and the corresponding bit in one of the three banks. The instruction has an immediate field, divided into six 3-bit fields. Each 3-bit field is defined according to the following table.

Table 5-2. PRDA.SWP 3-bit field definition

3-bit control field of group i		Behavior
Swap	Bank	
0	xx	No change
1	00	B0P[i] ↔ Pi
1	01	B1P[i] ↔ Pi
1	10	B2P[i] ↔ Pi
1	11	Undefined

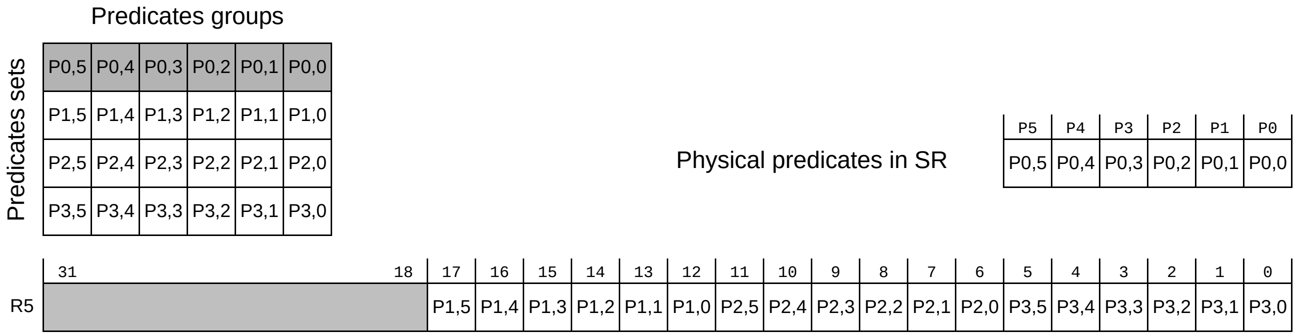
Note that, according to the assembler syntax, all the “Swap” bits are grouped together into a 6-bit immediate field and all the “Bank” bits are grouped together into a second 12-bit immediate field.

In the following example, 24 virtual predicate bits are supported. The active predicate bits are then manipulated, and actual mapping of the virtual predicate bits to actual SC3900FP resources is presented.

Note that, in this case, a single Rn register is required and, thus, a single instruction can always change all the six active predicate bits.

As before, each predicate bit is marked by two numbers “Pi,j.” The “i” number is the virtual set (out of four sets) and the “j” number is the virtual group (out of six groups).

In the initial state, the six virtual predicate bits “P0,5”÷”P0,0” are located in the physical predicate bits and the rest of the virtual predicates reside in three banks located in R5.



P0,5	P0,4	P0,3	P0,2	P0,1	P0,0
P1,5	P1,4	P1,3	P1,2	P1,1	P1,0
P2,5	P2,4	P2,3	P2,2	P2,1	P2,0
P3,5	P3,4	P3,3	P3,2	P3,1	P3,0

P5	P4	P3	P2	P1	P0
P0,5	P0,4	P0,3	P0,2	P0,1	P0,0

	31 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R5		P1,5	P1,4	P1,3	P1,2	P1,1	P1,0	P2,5	P2,4	P2,3	P2,2	P2,1	P2,0	P3,5	P3,4	P3,3	P3,2	P3,1	P3,0

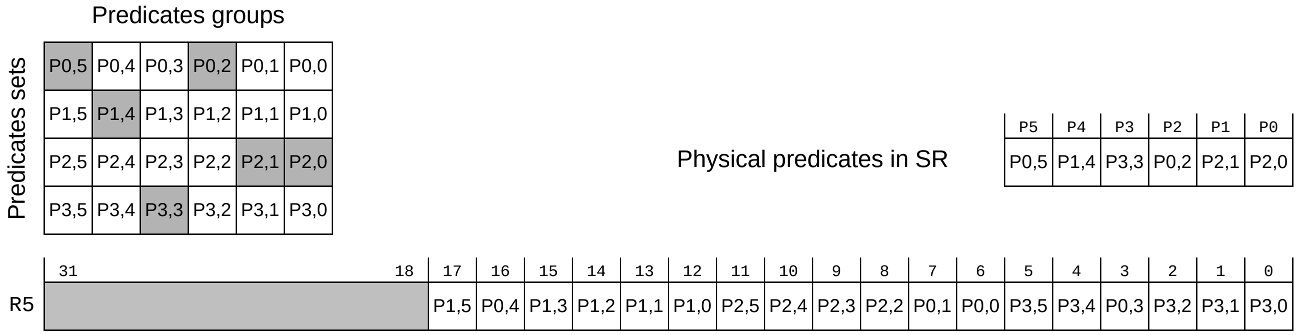
Figure 5-3. Example of virtual predicate bits organization

The map of active predicates can be changed by the following single IPU instruction:

```
; No swap for P5 and P2, Swap P4 with corresponding bit in bank 0,
; Swap P3 with corresponding bit in bank 2, Swap P1 and P0 with corresponding bits in bank 1
prda.swp noswp,swpb0,swpb2,noswp,swpb1,swpb1,r5      ; intermediate format for clarity

prda.swp #$1b,#$085,r5                               ; actual assembly format
```

The virtual predicate state following this instruction is as follows:



P0,5	P0,4	P0,3	P0,2	P0,1	P0,0
P1,5	P1,4	P1,3	P1,2	P1,1	P1,0
P2,5	P2,4	P2,3	P2,2	P2,1	P2,0
P3,5	P3,4	P3,3	P3,2	P3,1	P3,0

P5	P4	P3	P2	P1	P0
P0,5	P1,4	P3,3	P0,2	P2,1	P2,0

	31 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R5		P1,5	P0,4	P1,3	P1,2	P1,1	P1,0	P2,5	P2,4	P2,3	P2,2	P0,1	P0,0	P3,5	P3,4	P0,3	P3,2	P3,1	P3,0

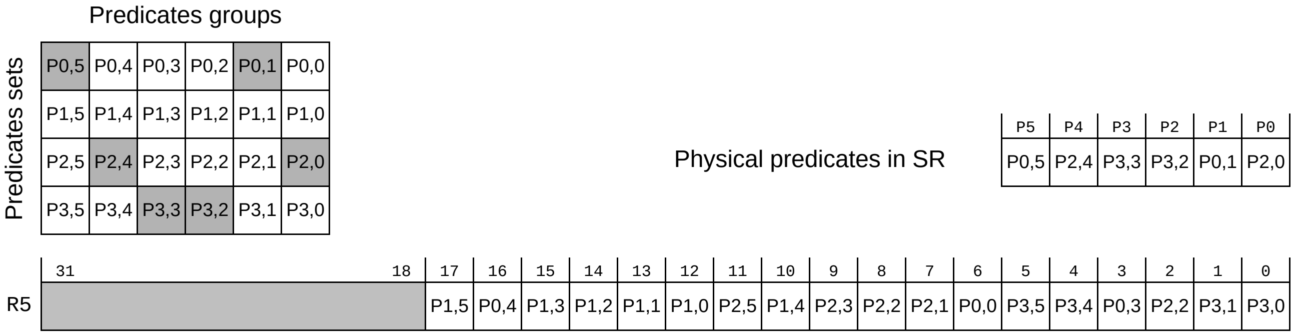
Figure 5-4. Virtual predicate state following the PRDA.SWP instruction

The map may be changed again with the following predicate swap instruction:

```
; No swap for P5, p3 and P0, Swap P4 with corresponding bit in bank 1,
; Swap P2 with corresponding bit in bank 2, Swap P1 with corresponding bits in bank 1
prda.swp noswp,swpb1,noswp,swpb2,swpb1,noswp,r5      ; intermediate format for clarity

prda.swp #$16,#$124,r5                               ; actual assembly format
```

The virtual predicate state following this instruction is shown in this figure.



Predicates groups

Predicates sets					
P0,5	P0,4	P0,3	P0,2	P0,1	P0,0
P1,5	P1,4	P1,3	P1,2	P1,1	P1,0
P2,5	P2,4	P2,3	P2,2	P2,1	P2,0
P3,5	P3,4	P3,3	P3,2	P3,1	P3,0

Physical predicates in SR

P5	P4	P3	P2	P1	P0
P0,5	P2,4	P3,3	P3,2	P0,1	P2,0

	31 … 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R5		P1,5	P0,4	P1,3	P1,2	P1,1	P1,0	P2,5	P1,4	P2,3	P2,2	P2,1	P0,0	P3,5	P3,4	P0,3	P2,2	P3,1	P3,0

Figure 5-5. Virtual predicate state following example instruction

Note that the user, or compiler, must track the location of each virtual predicate in the actual registers (the SR for the active predicates and, in this example, R5 for the non-active predicates) in order to map the active predicates correctly.

5.3 Understanding the program flow control architecture

The SC3900FP architecture has a program flow control architecture designed to meet the requirements of both the DSP number crunching kernels as well as the control code (for example, decision trees).

To handle complex algorithms, modern DSP cores rely on strong computational capabilities as well as out of core accelerators to crunch numbers efficiently. As a result, rather than only doing “number crunching,” an important portion of DSP algorithms could be classified as “control code”; thus, this type of code has to be handled in a very efficient way by modern DSP cores such as the SC3900FP.

The SC3900FP has several compiler-friendly hardware mechanisms, such as improved hardware looping and change of flow prediction, that lower the Change Of Flow (COF) penalty for compiled code as well as manually optimized code.

5.3.1 Optimizing loop execution with the hardware loop model

One of the most important features of a DSP algorithm is efficient loop execution. The SC3900FP core has a fully optimized looping mechanism that enables loop execution with up to four levels of loop nesting. The loop programming model includes four registers that specify the number of times each hardware loop is to be executed.

The VLESes that are iterated in the loop are called the “loop body.” At the assembly source code level, there is a single way to specify a loop; at the binary level, there are two types of hardware loops, as follows:

Sequential loops	Loops that have no BTB-able change of flow and/or another nested hardware loop inside it
Non-sequential loops	Loops that have BTB-able change of flow and/or another nested hardware loop inside it

If a sequential loop is short enough, the core executes it from an internal buffer in the core, ensuring low power and zero COF penalty in all cases. The core executes sequential loops that are too long to fit inside the internal buffer and non-sequential loops by re-fetching them from the instruction memory every iteration. A prediction mechanism of such long loops reduces the penalty to one time per loop. Note that while sequential and non-sequential loops have an identical programmer’s model, they differ in the actual encoding. The assembly program, or compiler, selects the correct format automatically.

Sequential loops can be one or more VLESes long. Non-sequential loops must be three or more VLESes long.

5.3.1.1 Loop Counter registers and the hardware loop programming model

There are four loop counter (LC*n*) registers (LC0, LC1, LC2, and LC3). Each register is responsible for a single hardware loop. The functionality of each register is described in the following sections.

The LC*n* registers are 32-bit read-write registers that specify the number of times each loop is to be executed. The LC*n* always holds a 32-bit unsigned value. This means that the largest number of loop iterations is $2^{32} – 1$. The DOEN.*n* instructions initialize the LC*n* register.

5.3.1.2 Defining loop notations

The notation used in the loop definitions is defined in Table 5-3.

Table 5-3. Loop notation

Term	Definition
Loop body	The VLESes that are iterated during loop execution
Sequential loop	A loop body that has no BTB-able change of flow or another nested hardware loop inside it
Non-sequential loop	A loop body that has at least one BTB-able change of flow or another nested hardware loop inside it
Start address (SA)	The address of the first VLES in a loop body. The start address is defined by the LOOPSTART*n* assembly directive.
Last address (LA)	The address of the last VLES in a loop body. LA is defined by the LOOPEND*n* assembly directive. In the case of a loop with only one VLES, SA is also the last address.
SA – 1	Address of the VLES that comes on VLES before SA

5.3.1.3 Encoding hardware loops

Loop operation relies on loop marks. The first loop mark is a branch hint instruction that is placed one VLES before the loop body (at SA – 1). The second loop mark is a branch instruction that is located in the loop body end (at LA). These instructions are automatically placed by the assembler and the user is not required to be aware of them. An instruction at SA – 1 informs the loop hardware of the start of the loop body iteration and causes the loop mechanism to save the loop in an inner buffer if the loop fits into it. The instruction at LA informs the loop hardware of the end of the loop body iteration and causes the loop mechanism to perform the iteration branch-back event sequence.

As a result, the PCU slot of these two VLESes is used and no other PCU instruction can be grouped with it. An exception is SKIP.*n* instruction that is allowed at SA – 1. If the SKIP.*n* or SKIP.U.*n* instruction is located at SA-1, the assembler encodes it with LPSKIP.*n* or LPSKIP.U.*n* instruction (respectively). The LPSKIP* instruction functions as both a SKIP and a loop start marker.

A loop with a body of only one VLES will not have a branch hint encoded at SA – 1.

5.3.1.4 How to initiate a hardware loop

To initiate a hardware loop:

1. Execute a DOEN.*n* instruction to load the corresponding LC*n* register with the number of iterations for the loop.
2. Mark the loop body by placing the LOOPSTART*n* assembly directive before it, and the LOOPEND*n* assembly directive after it. These directives allow the assembler to encode the correct loop marks, see Section 5.3.2, “LOOPSTARTn and LOOPENDn assembly directives”.
3. Optionally execute a SKIP.*n* or SKIP.U.*n* instruction before entering the loop to check the value of LC*n*. If the value of LC*n* is less than or equal to zero and SKIP.*n* was executed, or the value of LC*n* is equal to zero and SKIP.U.*n* was executed, then the loop is skipped and the program counter (PC) is loaded with the address specified in the SKIP.*n* or SKIP.U.*n* instruction. If it is guaranteed that LC*n* is greater than zero (for example, if the loop is initialized by an immediate value), the SKIP.*n* or SKIP.U.*n* instruction can be omitted. The SKIP.*n* and SKIP.U.*n* instructions provide the additional flexibility of skipping the steps in the loop completely if the loop count is initially zero or less.

5.3.1.5 Understanding loop execution

After the LC*n* is loaded with the DOEN.*n* instruction, the hardware loop is ready for operation. In sequential loops, whenever the program reaches the VLES marked by the loop end at LA, LC*n* is compared to the value 1 to detect loop termination. If the value of LC*n* is greater than 1, the program effectively jumps to the start address. LC*n* is decremented by 1 and the loop is repeated. If the value of LC*n* is equal to or less than 1, the loop terminates. Execution of instructions continues in sequence.

5.3.1.6 Loop nesting

The core has four hardware loops (LOOP0÷3) to execute up to four levels of loop nesting. A loop can be nested within a loop that has a different index.

Figure 5-6 illustrates an example of a loop nesting structure.

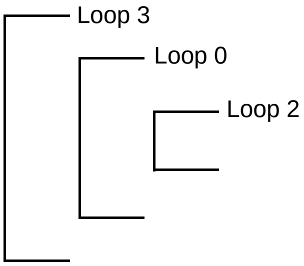


Figure 5-6. Loop nesting example

Initially, all three loops are not active. Loop 3 is the outermost loop. Loop 0 is the next loop, and loop 2 is the innermost loop of the three. In the normal program flow through the loops, loop 3 is starting and its first iteration takes it to loop 0, which is starting. In the first iteration of loop 0, loop 2 is starting. Loop 2 is executing until it has finished repeating, at which time loop 0 continue execution. When loop 0 stops repeating (including further complete cycles of loop 2), loop 3 continues execution.

5.3.1.7 Loop iteration and termination

The CONT.*n* instruction causes the active loop iteration to conditionally terminate before reaching the last VLES of the loop. If the value of LC*n* is greater than one, then the CONT.*n* instruction causes the program to jump to the first VLES of the loop (SA), LC*n* is decremented by one and the loop is repeated. If the value of LC*n* is less than or equal to 1, then the CONT.*n* instruction causes the program to branch to the address following the last VLES in the loop (LA + 1) and LC*n* is cleared. For correct operation as described above the offset for this instruction should point to LA + 1 of the loop while the value of the supplied Rn register should point to SA of the loop.

The BREAK.*n* instruction causes the active loop to terminate. The next PC to execute is specified by the operand of the BREAK instruction (should be the address following the last VLES in the loop (LA + 1)). The loop terminates regardless of the value of LC*n*, which is not changed.

5.3.2 LOOPSTART*n* and LOOPEND*n* assembly directives

The loop control instructions are assembled from source mnemonics in the conventional way. They are also disassembled normally. In addition, loop marks are encoded as implicit instructions. When coding a hardware loop in assembly, two loop-related assembly directives must be used to set the loop marks as follows:

- LOOPSTART*n*—Placed immediately before SA
- LOOPEND*n*—Placed immediately after LA

By definition, a loop body *n* is enveloped by the LOOPSTART*n* and LOOPEND*n* directives.

In disassembled code, the LOOPSTART and LOOPEND directives are not available.

5.3.2.1 Understanding hardware loop instructions

The programmer or the compiler writes a hardware loop by placing a DOEN.*n* instruction and two assembly directives: LOOPSTART*n* and LOOPEND*n* that envelope the iterated body of the loop. The index of the labels and instructions can be 0÷3. The programmer or the compiler can also use the loop COF instructions: SKIP.*n*, SKIP.U.*n*, CONT.*n* and BREAK.*n*. Note that although SKIP.*n*, SKIP.U.*n*, CONT.*n* and BREAK.*n* are non-BTBable instructions, they speculate with no penalty if not taken and thus have good performance when rarely taken.

The following example shows two nested hardware loops in the assembler:

```
   .
   doen.3 r7              ; Outer loop with index 3, number of iterations is in R7
   tfra.l _startloop1,r8  ; Loop start address stored in R8 for the CONT instruction
   .
   skip.3 _endloop3
loopstart3                ; Following VLES is the first in the outer loop
   .
   doen.1 #13             ; Inner loop with index 1, and 13 iterations
   .
   .
_startloop1:              ; A label marking SA of loop 1, for initializing R8 above
loopstart1                ; The following VLES is the first in the inner loop
   .
   .
   .                      ; Inner loop body
   if.p2 cont.1 _endloop1,r8; Conditional CONT instruction
   .
loopend1                  ; Previous VLES is the last in the inner loop
_endloop1:                ; Label marking LA+1 of loop 1, for the CONT instruction
   .
loopend3                  ; Previous VLES is the last in the outer loop
_endloop3:                ; Label marking LA+1 of loop 3, for the SKIP instruction
   .
```

The assembler detects whether the hardware loop is sequential and encodes the instructions listed in the following table.

Table 5-4. Instructions encoded by the assembler

Instruction	Description
DOEN.*n*	Initializes the loop counter register LC*n* according to index *n* (IPU instruction).
LPST.*n*[1]	Encoded one VLES before the loopstart*n* directive (SA - 1), marks the beginning of the loop.
LPST.SQ.*n* [1]	Similar to LPST.*n* for sequential loops.
LPEND.*n* [1]	Encoded at LA, decrements the loop counter register LC*n* according to index *n* and changes the program flow to the loop start if the loop counter is bigger than one.
LPEND.SQ.*n* [1]	Similar to LPEND.*n* for sequential loops.
SKIP.*n*	Encoded after the DOEN.*n* and before the LOOPSTART*n*, changes the program flow to the label specified in the instruction, if the loop counter register is less than one.
LPSKIP.*n* [1]	In case the SKIP instruction is encoded at SA - 1, the LPST and SKIP instructions are combined to a single PCU instruction. If the loop counter is less than one, changes the label specified in the instruction.

Table 5-4. Instructions encoded by the assembler (continued)

Instruction	Description
LPSKIP.SQ.*n* [1]	Similar to LPSKIP.*n* for sequential loop.
BREAK.*n*	Stop execution of current loop *n* and change the program flow to the label specified in the instruction.
CONT.*n*	Stop execution of current iteration of loop *n*. If the loop counter register is greater than one change the program flow to address pointed by supplied R*m* (should point to the start address of the designated loop), else change the program flow to the label specified in the instruction.

Note:

[1] These are implicit instructions that are generated by the assembler following explicit instructions (SKIP.*n* and SKIP.U.*n*), and assembler directives (LOOPSTART*n* and LOOPEND*n*). They are not allowed to be specified in the assembly source file.

5.3.2.2 Restrictions on hardware loops

The following main restrictions apply to hardware loops:

- Two hardware loops cannot start at the same VLES, nor can they end at the same VLES.
- PCU instructions are not allowed at LA (excluding LPEND or LPEND.SQ).
- PCU instructions are not allowed at SA – 1 of loops longer than one VLES (excluding LPST.*n*, LPST.SQ.*n*, LPSKIP.*n*, LPSKIP.U.*n*, LPSKIP.SQ.*n* or LPSKIP.U.SQ.*n*).
- Explicit write to a LC register inside a hardware loop with the same index is not allowed.
- Loop COF within a loop cannot have a destination in the same loop.

For a full description of the programming rules of SC3900FP, see Chapter 8, “Programming Rules and Guidelines.”

5.3.3 Fast return from subroutines

The core supports a Return Address Shadow (RAS) register stack for speeding up the execution of the return from subroutine (RTS) instruction. The JSR, and BSR instructions update the RAS stack with a return address. This update process is known as making the RAS valid.

When not accelerated, the return from subroutine instruction (RTS) instruction obtains the return PC from memory, by implicitly popping it from the software stack prior to performing the change of flow.

When there is at least one valid entry in the RAS register stack, the RTS instructions obtains the PC directly by popping the return PC from the RAS stack.

Non-accelerated execution of an RTS instruction takes twelve execution cycles. If the RAS is valid, the RTS instruction executes in six cycles in case of a BTB miss, and in one cycle in case of a BTB hit that is correctly predicted.

The RTS.STK instruction bypasses the special logic that implements this fast RTS mechanism. Use an RTS.STK instruction when the subroutine return address is explicitly changed in the stack (and thus RAS data may be invalid).

The RTS.STK and RTE.CIC instructions, and some variants of the CLRIC instruction invalidate the RAS stack when they execute. When the RAS stack is invalid, any RTS instruction that may use the RAS must get the return address from the memory.

When the RAS value is not the same as the value on the memory stack, it is the user’s responsibility to use RTS.STK instead of RTS. See Rule J.4.

The programmer’s model of the RAS stack is illustrated in Figure 5-7.

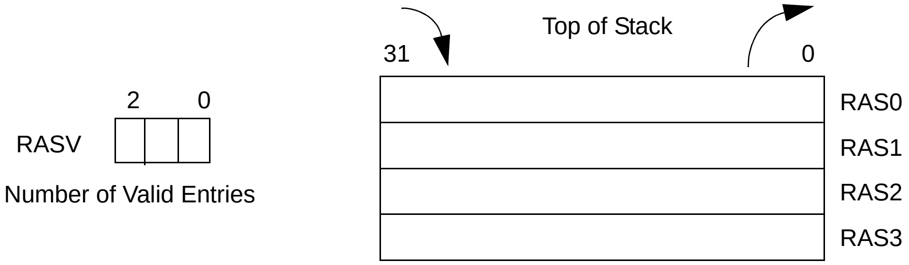


Figure 5-7. RAS stack programmer’s model

The operating principle: Upon execution of a JSR or a BSR instruction, the return address is pushed on the stack to RAS0 (in addition to the memory store), and RASV is incremented. If the RAS stack is already full upon JSR or BSR, the new return address is pushed and the oldest entry is discarded. Upon an RTS, if RASV is greater than zero, the RAS value from the top of the stack is popped and used as the return address, and RASV is decremented. A detailed functional description is given in the following table.

Table 5-5. Effect of instructions on the RAS stack

Instruction	Operation	Comments

JSR
BSR
	RASV := min (RASV+1, 4)
	RAS3 := RAS2
	RAS2 := RAS1
	RAS1 := RAS0
	RAS0 := return PC
		SP is updated as usual.
		PC and SR are stored in memory.

RTS	If (RASV>0)
	{ PC:=RAS0
	RASV--
	RAS0 := RAS1
	RAS1 := RAS2
	RAS2 := RAS3 }
	else
	{ PC:= from memory stack }
		SP is updated as usual in both cases.
		If RAS is valid (RASV>0), no load operation is executed on the bus.

RTS.STK	Functionality as usual per instruction definition, and in addition: RASV := 0	Always load PC from memory
RTE.CIC		Clears RASV if the appropriate immediate flag is set per RTE.CIC definition

Note that, similar to previous architectures, some RTE instruction variations do not clear the RAS stack. This can help to reduce the cycle loss of re-loading the RAS stack after returning from interrupts. When the interrupt causes a task switch, the RAS should be cleared so the OS is required to return from the interrupt using the RTE.CIC instruction with the appropriate RAS clearing immediate bit set, per the definition of this instruction.

5.3.4 Understanding flow prediction

The SC3900FP has both dynamic and static (compiled) mechanisms designed to reduce the change of flow penalty, which supports predicting and speculating the execution flow.

5.3.4.1 Performing dynamic prediction (BTB)

To perform dynamic prediction, the SC3900FP core has a unit that maintains a dynamic history of COF outcomes (taken or not taken), called the branch target buffer (BTB), and uses that history to predict COF direction. The BTB of the SC3900FP has 128 entries. The first time the core encounters a potential COF situation and that situation causes a change of flow, the branch prediction mechanism records information about that COF in a BTB entry. In other words, the core does not record a COF situation until the first time the COF is taken. The main portion of the BTB could be seen as a cache of previously encountered COF instructions, on which the BTB bases its prediction.

The user can override the default dynamic prediction by specifying static prediction hints in the code. The NOBTB attribute of a COF instruction prevents the insertion of the COF into the BTB. The use of this attribute may improve performance by blocking the BTB update in the case of an unpredictable change of flow, or a seldom taken change of flow, and so on.

5.4 Exception, idle and rewind processing

5.4.1 How exception processing works

Exceptions happen when an event occurs that causes the core to break the normal flow of the program in order to enable execution of the special exception handler, and continue the execution of the normal flow later on. There are two kinds of exception events—synchronous and asynchronous events. A synchronous event is generated due to the executed program flow (for example, illegal code, memory error, TRAP instruction) while an asynchronous event is generated due to an external source not related to the program flow (for example, timer interrupt, DMA).

The SC3900FP core handles exceptions due to a synchronous event in one of two ways, according to the event type: precise exception and imprecise exception.

In the case of a precise exception, the core breaks the program flow and saves the state of the machine before the execution of the instruction that is the cause of the event, that is, break-before-make. For example, when a memory error (for example, a memory protection violation) is detected on a transaction that is part of VLES “X,” the core stops the execution flow, restores the machine state (that is, all register values including the PC and SR) to the point before the execution of VLES “X,” and jumps to the memory exception routine. In this case, the instruction that caused the memory error is re-executed upon return from the exception. All precise exceptions are handled by the SC3900FP as break-before-make, except for the TRAP instructions that are handled as break-after-make; that is, the machine records the state following the TRAP instruction, and returns from the exception routine to the instruction following the TRAP instruction.

In the case of an imprecise exception, the core breaks the program flow and saves the machine state a few cycles/instructions following the VLES that generated the exception event.

Asynchronous events are handled by the core similar to imprecise event handling, that is, the program flow is broken a few VLESes after the assertion of the event.

The SC3900FP has the following sources of exception events:

- Hardware internal exceptions (for example, illegal code).
- Software exceptions (that is, TRAP).
- Debug events[1] - not supported in some implementations
- Memory exceptions (for example, memory errors/violations).
- Critical interrupts[2].
- Normal interrupts.

During exception handling, the operating system, utilizing core resources, can save the state of the interrupted flow in memory, execute the correct exception routine, restore the state of the interrupted flow and continue to execute the original program flow (in some cases exception routine may restore a program flow different from the interrupted one, for example, OS task switch handling). Note that since in many cases the OS stores the context of the interrupted task (or exception, in case of exception nesting) in the stack of that interrupted task, special hardware enables to accelerate this process, for both store and restore processes.

5.4.2 How the core enters and exists Idle state

When the core's pipe stages do not contain any instructions, the SC3900FP core is in Idle state.

Idle state is entered due to a request as described in this table.

Table 5-6. Entering and exiting Idle state

Enter Idle state reason	Enter Idle state details	Wake (exit Idle state) reason	PC of exit Idle state
Hardware reset	—	SoC hold-off signalling	SoC strap on value
Dedicated Debug instructions	DEBUGM.*n* SWB	Debugger "Go" signaling from the SoC	Debug return address (a register in the DTU)
out-of-core debug mode request	An SoC debug request, or any debug event configured in the DTU to cause entry into debug mode		
External halt request	SoC halt signalling	De-assertion of the SoC halt signal	First VLES that was killed due to the halt request
Nested irrecoverable exception	Exception received and SR2.IRR bit is set	Hardware Reset or Debug	N/A

[1] A debug event can cause the core to either jump to debug exception routine or enter the core into debug mode (idle state) to be controlled by a debug host. The debug event in this context leads to debug exception.

[2] Referred to as NMI (non-maskable interrupts) in previous StarCore architectures.

External halt requests are driven from the SoC using dedicated SoC signals, causing the core to stop executing instructions. There are various use cases where an external halt is asserted, such as entering a low-power state, stopping an errant core, and halting the core right out of reset (boot hold-off).

When the core exit reset (hardware reset signal is negated by the SoC), it may be kept in Idle state (external halt or debug mode) and waiting for further signaling before starting to fetch the first instruction. This mechanism enables a remote host, such as another core on the SoC, to master the boot process for all the SoC. Similarly it allows the external debug host to configure the debug logic in the SoC and the local DTU before the core starts to run.

For more information on the external halt situations, see the *SC3900FP FVP Subsystem Reference Manual*.

Entering Idle state from normal execution (other than by reset) is similar to an exception. Due to an event, the core breaks the normal flow of the program, but does not start fetching and executing instructions until it is “waked” by the SoC or the debugger hardware. As in exception handling, there are two kinds of enter Idle state requests: synchronous and asynchronous events. A synchronous event is generated due to the executed program flow, such as Break on PC event, while an asynchronous event is generated due to an external source not related to the program flow, such as an external request to enter debug mode, or low power (halted) mode.

Most of the synchronous events are handled by the SC3900FP in a precise manner (break on PC or on data address); in these cases, the core enters Idle state before the execution completion of the VLES that caused the event according to the case.

For some debug events, the debug mode request is imprecise, and the core breaks the program flow and enters Idle state a few cycles/VLESes following the VLES that generated the debug event.

Asynchronous events are handled by the core in a manner similar to that of imprecise event handling, that is, the program flow is broken a few instructions after the assertion of the event and the core enters Idle state.

Unlike exception handling, the programming model of the core is not changed when the core enters Idle state (PC, SR, SR2, EIDR and the link registers remain unchanged).

While the SC3900FP core is in debug mode it can do one of the two following debug actions:

- Core command: The debugger sets a single VLES (up to 128-bit wide) value in the DTU registers, and the core executes this VLES and returns to Idle state.
- Single step: The DTU sends a “single-wake” request to the core, the core starts fetching and executing from the debug return address and re-enters Idle state after the execution of a single VLES (killing the second VLES).

5.4.3 How the core handles rewind processing

The SC3900FP receives a rewind request from the memory system when a memory transaction cannot be completed successfully by the sub-system, but the memory system can fix it automatically until a retry is performed (for example, a tag match alias in the cache that is solved by invalidation of a line in the cache).

The SC3900FP core handles rewind requests in a way that is similar to that of precise memory error requests, except for the fact that it jumps to the interrupted VLES (re-executing it) and does not change the programming model. From the programmer's view, rewind can be perceived as additional memory latency cycles in the instruction execution.

5.4.4 How the core handles exceptions

Exception handling is done by the core hardware and operating system (OS) software. The following sections describe these steps.

5.4.4.1 Event generation

Each exception event is generated by one of the following four sources:

- PCU for internal exceptions
- DTU (Debug and Trace Unit) for debug exceptions - not supported for some implementations
- MMU (Memory Management Unit) for memory exceptions
- EPIC (Enhanced Programmable Interrupt Controller) for asynchronous exceptions/interrupts[1]

The event source supplies the following fields with the event request.

Table 5-7. Fields and attributes supplied by the exception source

Field	Description
Exception type (ETP)	A 6-bit field that defines the type of exception; for example, TRAP, illegal, memory error, precise debug, critical interrupt. The exception type is divided into three fields, as follows: • The 2 MSBits (bits 5:4) describe the exception source (PCU/DTU/MMU/EPIC). The core uses different resources (for example, stack pointers and link registers) and presets the machine according to the source. • The next bit (bit 3) is cleared if the exception is defined as fast or critical and set otherwise. • The 3 LSBits (bits 2:0) are used to further differentiate between the different types.
Exception Identification Number (EID)	A 10-bit number that supplies additional information regarding the exception event to the interrupt handler. For example, the TRAP.*n* instruction has a 10-bit immediate operand in which the user can specify the type of service that is requested from the TRAP handler. The concatenation of ETP and EID generates a 16-bit field that is unique for each exception request. This field is sampled by the core during exception handling into EIDR (see the EIDR description in Section 2.1.6.5, “Using the Exception ID Register (EIDR)”) and can be used by the exception routine software to determine the action needed to be performed in order to serve the specific exception request efficiently.
Dispatcher address	The address to which the core should jump after storing the current state of the interrupted flow. Normally, the OS has few such routines for handling all the exceptions. These routines are named “dispatchers.” This address replaces the need for legacy VBA register.
Priority level	A 5-bit field that is valid only for normal interrupts and determines the priority of the interrupt request and the masking of normal interrupts according to both this field and the IPM field in the SM2 register. See Section 2.1.6.4, “Using Status Register 2 (SR2).”

[1] EPIC supports external interrupts and internal asynchronous and imprecise events.

5.4.4.1.1 ETP, EID and dispatcher address values

The ETP, EID and dispatcher address values are defined according to the exception source/requestor according to the following table.

Table 5-8. Exceptions ETP, EID and dispatcher address generation

Exception source	Exception type[1]		ETP	EID	Dispatcher address	Comments
PCU	**TRAP.0**		00_0000	10-bit imm. field[2]	CESRA0 reg.	Fast TRAP instruction[3]
	TRAP.1		00_1000	10-bit imm. field[2]	CESRA1 reg.	Normal TRAP instruction[4]
	ILLEGAL		00_0001	00_0000_0000		Software emulation of illegal event
	Illegal source code	Illegal instruction		00_0000_0000		—
		Illegal VLES		00_0000_0001		
		Illegal fetch set		00_0000_0010		
		—		00_0000_0011		
		Illegal loop		00_0000_0100		
		—		00_0000_0101		—
		Illegal use of SWB		00_0000_0110		In case this instruction was used when the relevant partition in the DTU is disabled.
	Irrecoverable event		00_01xx[5]	Org. and req. ETP[5]		—

Table 5-8. Exceptions ETP, EID and dispatcher address generation (continued)

Exception source	Exception type[1]		ETP	EID	Dispatcher address	Comments
MMU	Program event	Multiple descriptor hit	01_f000[6]	00_0000_0000	MMU address[7]	For more details on MMU errors, see the Memory Management chapter in the SC3900FP Subsystem Reference Manual
		Segment miss		00_0000_0001		
		Permission violation		00_0000_0010		
		Unaligned program access		00_0000_0100		
		Cache command to non-cachable area		00_0000_0111		
		Non-cachable hit		00_0000_1001		
		External Fetch error		00_0000_1110		
		EDC error		00_0000_1111		
	Data event	Multiple descriptor hit	01_f100[6]	00_0000_0000		
		Segment miss		00_0000_0001		
		Permission violation		00_0000_0010		
		Peripheral access size error		00_0000_0011		
		Semaphore access error		00_0000_0101		
		Stack overrun		00_0000_0110		
		Cache command to non-cachable area		00_0000_0111		
		Illegal bus command		00_0000_1000		
		Non-cachable hit		00_0000_1001		
		External read error		00_0000_1110		
		EDC error		00_0000_1111		
DTU[8] (Debug)	Hardware debug events	Async./imprecise debug event[9]	10_0110[6]	00_0000_0000	Debug address[10]	All debugger hardware events, for example, hardware breakpoint
		Precise debug event[11]	10_0111[6]	00_0000_0000		
EPIC	Critical interrupts		11_0000	Exception source[12]	AIC addr.[13]	Critical interrupts (legacy NMI)
	Normal interrupts		11_1000			Normal interrupts

[1] Bolded in capital letters stands for a core instruction.

[2] The TRAP instruction has a 10-bit unsigned immediate field (u10). The immediate field value is copied from the instruction to the EID field upon successful execution of this instruction.

[3] This instruction enables implementation of fast TRAP routine with a private dispatcher in order to reduce the required overhead.

[4] TRAP.0 #u10 can support 1024 variations of TRAP

[5] The Irrecoverable event records the last ETP that was used and the ETP of the exception that could not be served, in order to help with post-mortem analysis. The 2 MSBits of the ETP of the exception that could not be served are stored in the field “xx” (2 LSBits) of the irrecoverable ETP, The 4 LSBits of the ETP of the exception that could not be served are stored in the 4 MSBits of the EID field and the last ETP that was used is stored in the 6 LSBits of the EID field.

[6] The Value of “f” is set according to the “Fast” attribute: 1 if “fast” (or “critical”) and 0 if not. For MMU exceptions, “fast” means that M_DESRA0 or M_PESRA0 MMU register was used as the dispatcher address (for data or program exceptions, respectively). If not fast, register M_DESRA1 or M_PESRA1 was used.

[7] MMU address is set according to the MMU programming model. In the current MMU, each event can be configured to be either fast, using fast MMU address or non-fast using non-fast address. See the *SC3900FP FVP Subsystem Reference Manual* for more details.

[8] In some implementations, the DTU does not generate debug exceptions

[9] An asynchronous debug event is caused by SoC external debug exception request. Imprecise debug event can be IEU events, or on floating error flags, and so on, that are detected by the hardware too late for a precise handling.

[10] The debugger can configure a single dispatcher address that serves all debug exceptions. For details, see the debug chapter in the *SC3900FP FVP Subsystem Reference Manual*.

[11] The core is supporting precise break-before-make handling of both Break on PC and break on data address events.

[12] Interrupt source is encoded into the EID field (only nine LSBits are in use by current EPIC).

[13] EPIC sets the dispatcher address according to the interrupt source.

5.4.4.2 Exception, Idle and Rewind priority and protection

Exception, Idle and Rewind priority is evaluated when aborting the first VLES that is killed due to the exception.

Priority of the event that is handled after this VLES is killed is as follows, where the break-after-make exception has the highest priority.

Table 5-8. Event priorities

Event	Description	Priority
Break-after-make exception: TRAP instruction	In this case, the previous instruction (TRAP) has already been executed, and the exception related to it should be handled before any other imprecise exception, break-before-make exception, IDLE request or rewind is handled.	Highest
Failure in the next VLES fetch due to program memory error	The following priority is a failure in the next VLES fetch due to program memory error such as MMU page miss, protection error, or data error (EDC) in a fetch set that includes, or is part of, the current VLES. In this case the core handles the program memory error exception, even if there are other pending exception or debug requests (such as interrupts).	—
Rewind request on the program flow	The following priority is a rewind request on the program flow. In this case, either the program memory or the debugger requires fetching the same address again (rewind request). This request is transparent to the user excluding the wasted cycles.	—
Imprecise/asynchronous IDLE requests, including enter debug mode and enter external halt modes	The following priority are imprecise/asynchronous IDLE requests, including enter debug mode and enter external halt modes. Imprecise debug request can be due to asynchronous external debug request, an indirect event. Such events are generated by the DTU based on core events.	—
Imprecise exception	The following priority are all the imprecise exceptions. Since these exceptions stop the program flow before the execution of the following exception it also mask any break-before-make due to this killed exception excluding the program memory error. If the break-before-make exception/IDLE request reason is yet valid, it will be generated again once the core returns from the other exception/IDLE state and served only then.	—
ILLEGAL exception when a complete legal VLES could not be decoded	The following priority is the ILLEGAL exception, in the case in which a complete legal VLES could not be decoded. Note that, although technically an illegal VLES is treated as a break-before-make exception, execution should not be resumed to this location after the exception handler (no point in re-executing this VLES again).	—

Table 5-8. Event priorities (continued)

Event	Description	Priority
Break on debug request	The following priority is for debug requests. Note that, in order to break on a PC, this PC must point to a legal VLES that can be fetched from memory without error. Note also that, if there is a break on PC request and a pending imprecise exception simultaneously, the exception will be handled and the debug request will be handled only upon the return from this exception.	—
Data memory error due to a transaction that was initiated by an instruction in the current VLES	The following priority is a data memory error due to a transaction that was initiated by an instruction in the current VLES. In this case, the core handles the data memory error exception only if the fetched VLES is legal, there are no pending imprecise exceptions and there is no break on PC debug request (since all the above stop the program flow before the memory transaction has began).	—
Break on address debug requests	The following priority is a break on address debug mode requests. Note that, in order to break on an address, there must be a legal fetched VLES, with no pending imprecise exception, and the address value must point to a legal memory location without data error. Note also that, if there is a break on address request and a pending imprecise exception or any other precise exception request simultaneously, the other exception will be handled and the debug address breakpoint request will be handled only upon the return from debug mode.	—
Data memory rewind request	The least priority is a data memory rewind request. In this case, data memory requires fetching the same address again (rewind request). The core starts re-fetching the VLES from the program memory and then re-executes the required memory transactions. This request is transparent to the user excluding the wasted cycles.	Lowest

Exceptions are protected against unintentional or illegal reentry and against irrecoverable nesting, using protection bits in the SR2 register (see Section 2.1.6.4, “Using Status Register 2 (SR2)”). Imprecise exceptions should be driven as a level input signal to the core. Any exception can be masked by a bit (or bits) in the SR2 register, which is set automatically by the core once handling of this exception begins, preventing undesired self-recurrence. If nesting is required, software can re-enable the exception after clearing the interrupt request level at the device holding it (this can be the originator peripheral for level interrupts, or the interrupt controller for edge interrupts).

Precise exceptions are also protected by bits in the SR2 register. When a precise event occurs while this event is protected, the event is either masked (that is, ignored by the core) or the event generates an irrecoverable error according to the case. The priority of such an irrecoverable error is the same as the priority of the original exception that cannot be handled. When an irrecoverable exception is generated, the bit SR2.IRR is set, and the core starts irrecoverable exception handling. Note that, although technically irrecoverable exceptions behave as break before make, for correct operation either the task causing the irrecoverable should be terminated, or system boot is required when this exception is asserted, since it is not possible to recover the machine state prior to the irrecoverable exception. In case of nested irrecoverable exceptions (an irrecoverable exception when the SR2.IRR bit is set) the core automatically attempts to enter debug mode. If the DTU host partition is enabled, then the core enters debug mode and halts. If the DTU host partition is not enabled, the core enters an endless loop of attempted re-execution of the VLES causing the second irrecoverable condition. No programming model is changed in this process, so if the DTU host partition is enabled while the core is in this loop, debug mode is entered at the correct location. For nearly all causes, this endless loop could be broken by an external halt request or an exception that is not otherwise blocked. An irrecoverable exception caused by a nested program MMU error can be broken only by entering debug mode.

This table summarizes the exceptions, IDLE request and Rewind priority, and re-entry behavior.

Table 5-9. Priority of system exceptions, IDLE requests and Rewind requests for SC3900FP

Name	Assertion scenario	Type	Reentry protection bit	Reentry behavior if protected	Entry point	Priority
TRAP	Software exception	Exception	SR2.DIN	Irrecoverable	Break-after-make	Highest
Program memory error	Program memory error fetching part or all of the current VLES	Exception	SR2.DMN	Irrecoverable	Break-before-make	—
Program memory or debug rewind request	Re-fetch the current VLES due to program fetch request or debug sync request	Rewind	N/A		Break-before-make	—
Imprecise debug mode request	Imprecise debug event, external (asynchronous) debug request.	Idle Request	N/A		Imprecise	—
Imprecise low power mode req.	Assertion of enter low power mode signal (halt)	Idle Request	N/A		Imprecise	—
Imprecise debug exception	Imprecise debug events, break always (single step), external (asynchronous) debug request.	Exception	SR2.DDN	Masked	Imprecise	—
Critical interrupt	Critical external interrupt. NMI (Non-Maskable Interrupt) In previous StarCore architectures	Exception	SR2.DIN / SR2.DCI	Masked	Imprecise	—
Normal interrupt	External interrupt	Exception	SR2.DIN / SR2.DI	Masked	Imprecise	—
Illegal exception	Illegal set, illegal instruction or core protection violation	Exception	SR2.DIN / SR2.ILL	Irrecoverable	Irrecoverable break-before-make	—
Debug mode on PC	Break on PC due to hardware breakpoint, SWB (software breakpoint) instruction or DEBUGM instruction	Idle Request	N/A		Break-before-make	—
Data memory error	Page miss or any other memory error on one of the data buses due to transaction of the current VLES	Exception	SR2.DMN	Irrecoverable	Break-before-make	-—
Debug mode on address	Break on address due to address hardware breakpoint	Idle Request	N/A		Break-before-make	—
Data Rewind	Re-execute the current VLES due to data transaction request	Rewind	N/A		Break-before-make	Lowest

When there is more than one exception event with the highest priority terminating the same VLES, the ETP field, EID field and the dispatcher address are set according to the event with the highest priority and lowest value of {ETP, EID}.

In general, all the above priorities are checked for all of the events related to the same VLES, starting with the program fetch, decoding, and memory transactions that are part of this VLES. The core chooses the highest priority of all of these events. Priority can be overruled in the following cases:

- A VLES that crosses the fetch set boundary (aligned 32-byte boundary in memory):
  In this case, some of the conditions are checked for the first fetch set, and will dictate the type of event, even if a higher priority condition exists for the second fetch set or for the whole VLES. For example, if a rewind condition is detected on the first fetch set and an error condition is detected on the second fetch set, the rewind condition will be handled for this VLES although it has lower priority.
- An unaligned data memory transaction that crosses the 4 KB boundary and thus splits into two transactions:
  In this case, if there is an error or rewind request on the first cycle of the split, it has higher priority on any error or rewind request on the second cycle.

5.4.4.3 Breaking the program flow

The SC3900FP core starts exception handling differently for precise and imprecise exceptions, IDLE request or rewind events.

A precise exception event is handled by the core only when the instruction that caused the event was committed (and not killed due to incorrect speculation). When a precise exception event is not handled due to a prior exception or incorrect speculation, it is ignored by the core.

Upon handling of a break-before-make event, the VLES causing the exception and all following VLESes are killed. In the case of a break-before-make exception, the core state is restored to the state prior to execution of the killed VLESes.

Note that, in such a case, memory transactions might start for killed transactions, changing the caches state, and so on. The L1 memory subsystem does not issue a transaction that is marked as destructive load outside the subsystem, as long as it might be killed, and as a result. Hence, there is no situation in which a destructive load transaction should be killed after it was issued from the L1 subsystem.

Imprecise events stop the execution of the current program flow, before any transaction of the instructions following the flow break are started. Therefore, when returning from an imprecise exception, instructions (and memory transactions) are not re-executed. This is different from precise exceptions, which kill instructions and memory transactions after they were issued.

Any imprecise event generator has to latch the request and sustain level assertion until the core is able to accept it (holding the exception request can be done by the EPIC for edge events or by the interrupts source for level events, holding of debug mode request is done by the DTU until Idle mode is reached). The core has hardware protection that prevents unwanted exception nesting due to the level request; software can enable nesting by clearing the relevant bits in SR2 once the level request is cleared (in the EPIC or in the source according to the case) by the software.

5.4.4.4 Saving machine state and hardware preparation for the exception handler

During exception processing, and before the actual jump, the core changes the value of the following registers: PC, SR, SR2 and EIDR (for a description of these registers, see Section 2.1.6, “Registers for program and exception control”). Since the value of these registers prior to exception handling is required in order to restore the original program flow upon returning from the exception, their original content is saved in one of three 128-bit Link Registers: LR0, LR1 or LR2. LR1 is used to service precise memory error exceptions, LR2 is used servicing debug exceptions, and LR0 is used for all other exceptions.

Note that, in order to avoid loss of data, nesting of another exception writing to a link register should be disabled until the content of the link register is saved to memory by the software. Nesting of exceptions using another link register is allowed, for example, an MMU exception (storing the previous state in LR1) is allowed before the storing of another exception state from LR0 (this is the reason why more than a single link register is required to begin with).

When handling an exception, the core performs the following actions:

- Copy the current value of PC:SR:SR2:EIDR registers into one of the three link registers according to the exception source:
  - — All exceptions but MMU and DTU:Copy the value of PC:SR:SR2:EIDR registers to LR0.
  - — MMU exception: Copy the value of PC:SR:SR2:EIDR registers to LR1.
  - — DTU exception: Copy the value of PC:SR:SR2:EIDR registers to LR2.
- Set the PC value to the dispatcher address that was received from the originator of the exception (core, MMU, DTU, EPIC). See Section 5.4.4.1.1, “ETP, EID and dispatcher address values,” for more details.
- Clear the whole SR register.
- Preset the SR2 register according to the exception source, as defined in the following table:

Table 5-10. SR2 preset value during exception handling

SR2 field	Exception source			
	PCU	MMU	DU	EPIC
SPSEL	01 b (EXP SP)	01 b (EXP SP)	11 b (Debug SP)	01 b (EXP SP)
ASPSEL	**Copy SPSEL current value**[1]			
IDE	0 b	0 b	1 b	0 b
PE	**0 b**			
EXP	**1 b**			
IPM	11111b	11111b	11111b	If critical (11111b) else (EPIC Priority)[2]
DI	**1 b**[3]			
DCI	1 b[3]	1 b[3]	1 b[3]	Set if critical[4]
DIN	**1 b**			
DMN	Keep[3]	1 b	Keep[3]	Keep[3]
DDN	Keep	Keep	1 b	Keep
ILL	Set on Illegal[5]	Keep[3]	Keep[3]	Keep[3]
IRR	Set on Irrecoverable[6]	N/A	N/A	N/A

[1] This action is done in order to preserve the stack selection of the interrupted flow in the alternate stack selector, since this is the required default value for the OS to which interrupted flow context should be saved.

[2] In the case of an EPIC critical interrupt, priority is all one (all non-critical exceptions are masked). In the case of an EPIC normal interrupt, IPM is set according to EPIC priority, as received with the interrupt request.

[3] Note that this behavior is changed from legacy StarCore.

[4] If the exception is critical, set DCI; otherwise, keep DCI cleared (non-critical exceptions are masked when this bit is set).

[5] Set in the case of an illegal exception (if it was set for illegal exception an irrecoverable exception is generated); otherwise, keep the previous value.

[6] Set in the case of irrecoverable; otherwise, keep cleared (all exceptions are masked when this bit is set)

- Copy the exception ETP and EID values (see Section 5.4.4.1.1, “ETP, EID and dispatcher address values for more details) to EIDR.
- Start the exception routine execution (fetching from the now updated PC).

5.4.4.5 Guidelines for writing an OS exception routine

The OS is expected to save the link register in memory and optionally enable nesting at the start of the exception dispatcher routine (and thus enable exception nesting according to priority). After saving the link register or registers, the OS can save all the required state of the machine (all registers, or just a subset of them as required), and start the actual exception handling. After the exception handling is done, the machine state prior to the exception routine should be restored by software. The value of the saved linked register should be stored to the active stack and then the core can return from the exception routine to the interrupted routine at the interrupted location and continue execution. Nesting of imprecise exceptions can be enabled by software once the appropriate machine state is stored and the exception request is cleared by software (exception request is cleared by software in the EPIC for edge exceptions and in the remote exception requestor in case of level exception).

Upon an exception, the core saves the current minimal context (PC, SR, SR2 and EIDR) into one of the three link registers (as explained in the previous paragraph), and restores them from the active stack (the stack pointed by SPSEL) upon executing the return from exception instruction (RTE). For proper operation, the OS should write this minimal context of the machine, which was written by the core to the appropriate link register on the current active stack, just before it is restored directly to PC, SR, SR2 and EIDR by using an RTE instruction at the end of the exception routine. Normally, the context of the interrupted task/exception can be saved on its own stack, or on the current exception stack at the start of the exception routine, and restored from the same stack at the end of the exception routine.

A standard simplified non-MMU/DEBUG exception routine saving the context in the exception stack should look as follows:

```
[ push.4l lr0                   ; save the link register to the current stack
  bmclra #$0100,sr2.h]          ; enable nesting of other exceptions using lr0
  push.4l r0::r3 push.4x d0::d3 ; start storing full context
  .
  .
  push.4l procid:tid:tmtag      ; end storing all context
  tfra eidr,r1                  ; move the interrupt ID to a register and
                                ; start working on it
  .
  .                             ; exception body
  .
  ei                            ; optional enable interrupts
  chcf r5                       ; optionally update IPL as required
  .
  .
  pop.4l procid:tid:tmtag       ; start restoring the context
  .
  .
  pop.4x d0::d3 pop.4l r0::r3   ; end restoring the context
  rte                           ; PC:SR:SR2:EIDR are restored from the stack
```

The same simplified non-MMU/DEBUG exception routine, this time storing the core context in the “interrupted” task, uses PUSHC and POPC to access the stack that was interrupted or to be restored. Note that in this case, the stack pointer of the task, TID and SR2.ASPSEL have to be saved elsewhere, for example in the Task Control Block (an OS data construct) because these values are required to define what memory will be accessed by POPC. The simplified routine should look as follows:

```
[ pushc.4l lr0                  ; save the link register to the “interrupted” stack
  bmclra #$0080,sr2.h]          ; enable nesting of other exceptions
  pushc.4l r0::r3 pushc.4x d0::d3; start storing full context to the “interrupted” stack
  .
  .
  pushc.2l r0:r1       ; like the procedure described in
                       ; Section 4.9.4, “Saving and restoring the context to/from the stack,
                       ; without TID, SP, SR2.ASPSEL
  tfra sr2,r4                   ; Save SR2 including the ASPSEL value
  tfra tid,r5
  tfrca sp,r6                   ; save the SP of the previous task
```

```
..
st.4l r4:r5:r6:r7,(r8)          ; save SR2.ASPSEL,TID, SP and unspecified data to the TCB
                                ; (Task Control Block - OS data construct)
..
  tfra eidr,r1                  ; move the interrupt ID to a register and
                                ; start working on it
  .
  .                             ; exception body
  .
  ei                            ; optional enable interrupts
  chcf #$10,sr2.ipl             ; optionally update IPL as required
  .
  .
  ld.4l (r8),r4:r5:r6:r7        ; load data from the TCB
  chcf r4,sr2.asp               ; restore ASPSEL from r4
  tfra r5,tid                   ; restore TID
  tfrca r6,sp                   ; restore SP of the task
  popc.2l r0:r1                 ; start restoring the context from the “interrupted” stack
  .                             ; see section/page: 4.9.4/4-25
  .
  popc.4x d0::d3 popc.4l r0::r3 ; end restoring the context
[ popc.4l lr0                   ; restore the link register
  bmseta #$0100,sr2.h]          ; disable nesting of other exceptions using lr0
  push.4l lr0                   ; save the link register to the current stack
  rte                           ; PC:SR:SR2:EIDR are restored from current stack
```

Note that the routine cycle count may be improved (remove read after write interlock between the two last instructions) by saving the link register later and transferring it to the active stackearlier. The downside is that nested interrupts are disabled for some additional cycles. Also, restoring of TID could be spaced more from the following TFRCA.

The MMU exception has a dedicated link register because an MMU exception could occur during the ISR of a non-MMU exception, before the respective link register of that exception (LR0 or LR2) was saved.

In the example below, it is assumed that the MMU ISR would like to use TRAP instructions, so it has to save LR0. Also, it would like to allow the nested MMU exceptions for other causes, for example, servicing EDC errors.

The user may select a different set of link registers to save, based on the nesting requirements of this routine in the specific application.

A standard simplified MMU exception routine should look as follows:

```
  push.4l lr1                   ; save the MMU link register
  push.4l lr0                   ; save the non-MMU/DEBUG link register
  bmclra #$0300,sr2.h           ; enable nesting of non-debug exceptions
  push.4l r0::r3 push.4x d0::d3 ; start storing full context
  .
  .
  .                             ; Same as non-MMU/DEBUG routine
  .
  .
  pop.4x d0::d3 pop.4l r0::r3   ; end restoring the context
  bmseta #$0300,sr2.h           ; disable nesting of non-debug exceptions
  pop.4l lr0                    ; restore non-MMU/DEBUG link register
  rte                           ; PC:SR:SR2:EIDR are restored from the stack
```

The debug exception has a dedicated link register because some use cases could occur during the ISR of a non-debug exception, before the respective link register of that exception (LR0 or LR1) was saved.

In the example below, it is assumed that the debug ISR would like to use TRAP instructions, so it has to save LR0. It may also like to allow nested debug exceptions; in this case, it should also save LR2.

The user may select a different set of link registers to save, based on the nesting requirements of this routine in the specific application.

A standard simplified DTU exception routine should look as follows:

```
push.4l lr2                         ; save the DEBUG link register
push.4l lr0                         ; save the non-MMU/DEBUG link register
bmclra #$0500,sr2.h                 ; enable nesting of non-MMU exceptions
push.4l r0::r3    push.4x d0::d3     ; start storing full context
.
.
.                                   ; Same as non-MMU/DEBUG routine
.
.
pop.4x d0::d3     pop.4l r0::r3      ; end restoring the context
bmseta #$0500,sr2.h                 ; disable nesting of non-MMU exceptions
pop.4l lr0                          ; restore non-MMU/DEBUG link register
rte                                 ; PC:SR:SR2:EIDR are restored from the stack
```

5.4.4.6 Understanding the Return from Exception (RTE) instruction

The RTE (Return from Exception) instruction restores the registers PC, SR, SR2 and EIDR directly from the current active stack, and thus restore the state of the machine (working mode, stack source, flags, exception enable state, and so on) according to the value of the relevant bits in the stack memory. Note that there is no return instruction, operating directly from the “interrupted” stack, and thus the exception handler has to make sure that the return data resides in the current active stack before RTE is executed (as demonstrated in the above examples).

The RTE instruction can optionally clear the core internal context, such as the BTB history and the RAS buffers (named RTE.CIC). Cleaning is required when the memory map changes. For example, a task that reuses the same effective addressing is switched when memory translation of the executing task is changed. Cleaning is also required when the program itself is modified in memory. In such cases, the core internal context may be misleading and need to be cleared.

5.4.4.6.1 Understanding the RTE instruction variations

There are two variations of the return from exception instruction. When the exception does not return to the place it interrupted, RTE.CIC (Return from Exception, Clear internal Content) should be used. This instruction clears the RAS stack, the BTB content, and any other internal content. If the exception returns to the interrupted location, and the exception did not change the program in any way, an RTE that preserves the internal content can be used to improve the performance.

Chapter 6
Encoding

This chapter describes the high-level encoding scheme of the SC3900FP core, describing issues like the encoding map, the prefix and suffix encoding, the grouping encoding, and so on.

The SC3900FP instructions are encoded in 16-, 32-, 48-, and 64-bit forms. Up to eight instructions can be grouped together to be executed in parallel by forming a Variable Length Execution Set (VLES).

The main features of the encoding scheme are as follows:

- No architectural limit on VLES size, instruction size, and their mixture
- Compiler-friendly 3-operands instructions
- Fully predicated ISA
- Support for 32 R registers and up to 64 D registers (implementation dependant)
- Hardware-friendly decoding

6.1 Understanding instruction grouping

The SC3900FP uses dedicated grouping bits in the encoding of instructions to determine what instructions are grouped together for concurrent execution.

Almost any mixture, of any instruction size, can be grouped to form any VLES size and in any order. The few grouping limitations are detailed in the following sections.

Each instruction word (16 bits) includes grouping bits, as follows:

- An MSBit equal to '0' indicates that the word is not last in the instruction, which can mean one of the following:
  - — It is the first word of a 2-word (32-bit) instruction.
  - — It is the first or the second word of a 3-word (48-bit) instruction.
  - — It is the first, second or third word of a 4-word (64-bit) instruction.
- Two MSBits equal to '10' indicates that it is the last word in an instruction, or a one word instruction, but not the last word of a VLES (that is, more instructions of the same VLES follow).
- Two MSBits equal to '11' indicates that it is the last word in a VLES.

6.1.1 Grouping examples

Figure 6-1 describes instruction grouping of several patterns, as follows:

- A single 16-bit instruction
- A single 32-bit instruction

- A 32-bit instruction grouped with a 16-bit instruction
- A 32-bit instruction grouped with a two 16-bit instructions

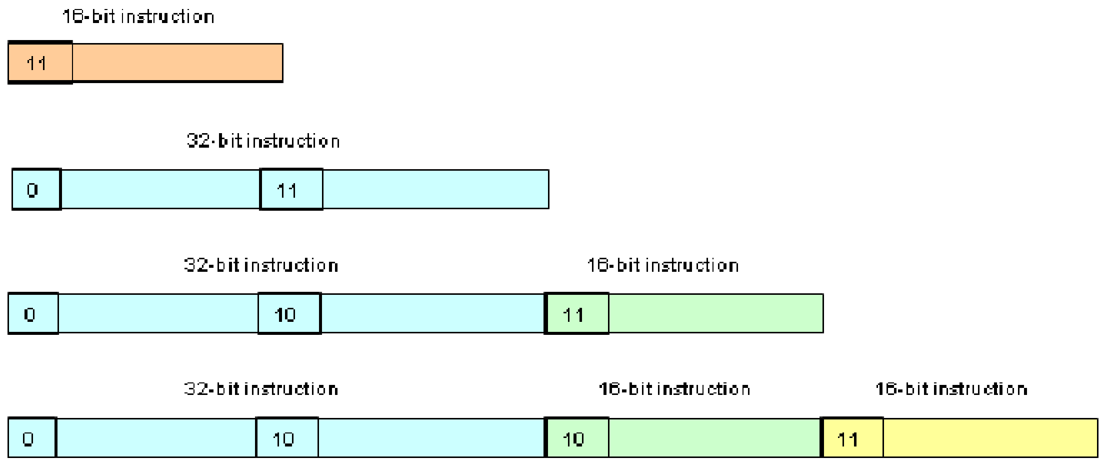


Figure 6-1. Grouping encoding examples

6.1.2 VLES ordering and limitations

The instructions are placed in the VLES in the following order (lower numbers are closer to the start of the VLES, that is, with a lower address):

1. PCU
2. LSU (up to two)
3. IPU
4. DALU (up to four)

This ordering is done automatically by the assembler tool and is generally transparent to the programmer. There are several cases where the ordering of instructions that belong to the same execution unit (DMU or LSU) has semantic meaning. For information on such cases, see Section 2.5.2.1, “Cases of serial semantics within a VLES.” Nevertheless, unlike previous StarCore architectures, if a VLES contains more than a single LSU instruction, the first LSU instruction is mapped to LSU0 and the second to LSU1. In some cases, this behavior has architectural implications; for example, two stores to memory are executed serially, one after another. This behavior also has an impact on encoding (for information on the impact on LSU0/1 encoding, see Section 6.1.3, “VLES suffix word/s”).

6.1.2.1 Ordering DALU instructions

The DALU instructions can be encoded in 64-bit encoding and 32-bit encoding.

A 64-bit encoding can carry one of the following instructions:

- Container of two encapsulated 32-bit DALU instructions operating on two DMU units. The first instruction is allocated to an even DMU and the second is allocated to the following, odd DMU. The majority of the DALU instructions are encoded as 32-bit instructions encapsulated in 64-bit encoding. If only a single DALU instruction is required, the second DMU can be programmed to execute a DALU NOP[1].

- A 64-bit instruction operating on a single DMU execution unit. In this case the instruction is allocated to an odd DMU with the preceding even DMU getting a NOP instruction.
- A 64-bit dual compounded instruction operating on two DMU execution units.

The 32-bit encoding forms include a premium subset of DALU 32-bit instructions. The premium subset includes instructions that are commonly used in VLESes with either one DALU instruction or three DALU instructions in a VLES. The DALU 32-bit premium encoding is aimed at code size reduction. A 32-bit premium DALU instructions also have an encapsulated form in the 64-bit containers. Hence the assembler can always choose the encoding variant that consumes less encoding words.

VLES encoding of DALU instructions can use a mixture of 64-bit encoding and 32-bit premium encoding. The following options are available for grouping DALU instructions:

- 32-bit premium DALU encoding (encoded into DMU0)
- 64-bit encoding (encoded into DMU0 and DMU1)
- 64-bit followed by 32-bit premium encoding (64-bit encoded into DMU0 and DMU1 and 32-bit encoded into DMU2)
- 64-bit encoding followed by 64-bit encoding (first 64-bit encoded into DMU0 and DMU1 and the second 64-bit encoded into DMU2 and DMU3)

For details on grouping limitations, see Chapter 8, “Programming Rules and Guidelines.”

6.1.3 VLES suffix word/s

A suffix word is located at the end of a VLES and is aimed to extend the encoding of the instructions in this VLES or pad the VLES with NOPs as follows:

- Extended encoding for instruction predication.
- Extended encoding for DALU instruction using high 32 registers (D32÷D63).
- NOP padding may be encoded using several sequential suffix words.

6.1.3.1 Detailed suffix encoding

The following figures describe the suffix.

31	30 23	22 19	18 16	15	14	13 5	4 0
0	PR_Hi_Bits	Reserved	DALo[1]	1	E	DALU_Hi_Reg	Suffix

[1] Upper bits of DALU Hi Registers field

Figure 6-2. 32-bit suffix encoding format

15	14	13										0
0	E	x	x	x	x	x	x	x	Reserved	0	1	0

Figure 6-3. 16-bit suffix encoding format

[1] Another option is DALU 32-bit premium encoding, as explained in the following paragraphs

The 32-bit suffix carries extension bits for instructions predication and DALU hi registers, while the 16-bit suffix is used for NOP padding only.

Each bit in the PR_Hi_Bits field extends the conditional field of an execution unit allocation (for specific bit allocation, see the following table). With the absence of a suffix, each instruction with optional conditional execution has a two bit field (bits 13:12 in most instructions) that defines the conditional execution, as follows:

- 00—Unconditional execution (IFA)
- 01—Conditioned by P0 (IF.P0)
- 10—Conditioned by P0 (IF.P1)
- 11—Conditioned by P0 (IF.P2)

When a suffix exists, the conditional behavior of each instruction stays the same when the corresponding bit in PR_Hi_Bits field is cleared; when the corresponding bit in PR_Hi_Bits field is set, the conditional behavior changes to the following:

- 00—Conditioned by P0 (IF.P3)
- 01—Conditioned by P0 (IF.P4)
- 10—Conditioned by P0 (IF.P5)
- 11—Reserved

Each bit in the DALU_Hi_Reg field extends the operand field of one of the operands of each DMU from 5-bit to 6-bit wide, enabling support of 64 registers (see the specific bit allocation in the following table). With the absence of a suffix, each operand in each DMU instruction can point to DALU registers D0÷D31, according to a 5-bit operand field allocated to it in the instruction.

When a suffix exists, each operand to which its corresponding bit in the DALU_Hi_Reg field is asserted points to a DALU register D32÷D63, according to a 5-bit operand field allocated to it in the instruction.

The suffix field is a constant that is used to decode this 32-bit word as a suffix.

The following table details the bit allocation of the dedicated suffix fields and their usage.

Table 6-1. Suffix fields description

Bit number	Part of field	Extended field	of Execution Unit
30	PR_Hi_Bits	Conditional execution	PCU
29			LSU0
28			LSU1
27			IPU
26			DMU0
25			DMU1
24			DMU2
23			DMU3

Table 6-1. Suffix fields description (continued)

Bit number	Part of field	Extended field	of Execution Unit
18	DALU_Hi_Reg	Operand A	DMU0
17		Operand B	
16		Operand C	
13		Operand A	DMU1
12		Operand B	
11		Operand C	
10		Operand A	DMU2
9		Operand B	
8		Operand C	
7		Operand A	DMU3
6		Operand B	
5		Operand C	

6.2 Instruction encoding

The following sections and figures illustrate the outline of the encoding scheme, based on the SC3900FP grouping method.

The main features are as follows:

- The instruction set encoding is 32-bit based. Some units include 16-, 48-, or 64-bit instructions as well for improved usability and code size density.
- A dedicated unit field (bits 4:0) of any instruction is used by the core to identify the execution unit to which the instruction should be dispatched. In the following figure examples, this field is labeled with the unit name: PCU, DALU, LSU, or IPU. Note that several entries out of the 32 possibilities belong to the same unit.
- Almost all operand fields are orthogonal and un-compressed (that is, each field describes a single operand). For easy hardware implementation, these fields are allocated at the same place within the instruction, as much as possible, with the predication field included.
- Each sub-unit (LSU0 and LSU1, DMU0÷DMU3) is considered identical and there is no encoding difference between them. When differentiation is required, it is done according to the ordering of the instructions within the encoded VLES.
- The 64-bit DALU encoding uses three types of encoding, as follows:
  - — Container of two encapsulated 32-bit DALU instructions operating on two DMU units.
  - — A 64-bit instruction operating on a single DMU execution unit.
  - — A 64-bit compounded instruction operating on two DMU execution units.

  The three types of 64-bit DALU formats encoding differ as follows:

  - The 64-bit dual-instructions container differs from the other 64-bit formats by a single bit in the 64-bit space (bit 46).
  - The 64-bit compounded instruction differs from a 64-bit single instruction by using different entries in the unit field (bits 4:0).
- Encoding of 32-bit encapsulated DALU instructions is as follows:
  - The encapsulated DALU instructions has a dedicated 32-bit encoding space in which all of the unit field (bits 4:0) values are allocated to the DALU.
  - Bit 15 of the encapsulated DALU instructions (CG) is set to 0 in the first instruction (bits 63:32) and to 1 in the second instruction (bits 31:0)
- The 32-bit DALU premium encoding is defined according to the following:
  - These instructions are a subset of the ‘regular’ encapsulated 32-bit DALU instructions.
  - Only one ‘premium’ instruction may be used in a VLES.
  - These instructions are encoded as part of the 32-bit encoding space, including the PCU, LSU and IPU instructions, and are identified as DALU instructions, according to the unit field (bits 4:0) value.

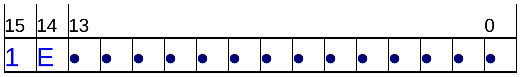
The rules listed above do not apply to the 16-bit encoding scheme, as the bit budget is very tight:

- There is no dedicated unit field.
- Operand fields are located in the same area; however, several formats exist, differing both in sizes and in widths.
- This scheme is unconditional; thus, no predicate register is encoded except for the change-of-flow instruction/s.

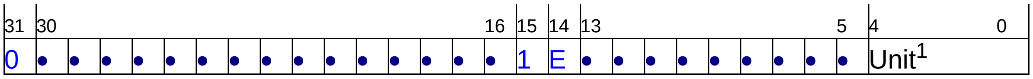
6.2.1 Instructions formats

The following encoding formats are supported by the SC3900FP core:

16-bit encoding of PCU, LSU and IPU:

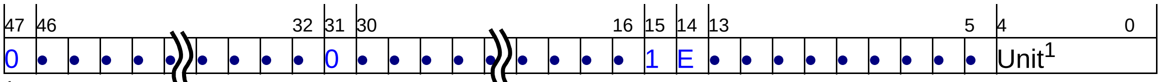


32-bit encoding of PCU, LSU, IPU, premium DALU and Suffix:



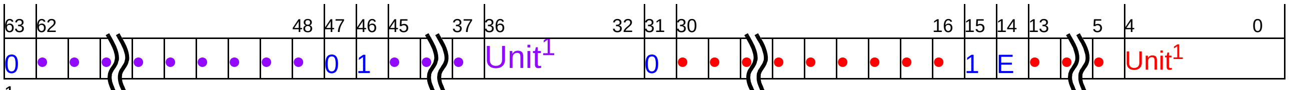
[1] Unit can be allocated to PCU, LSU, IPU, premium DALU or Suffix

48-bit encoding of PCU, LSU and IPU:



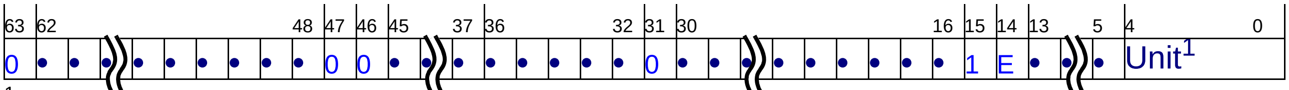
[1] Unit can be allocated to PCU, LSU or IPU

64-bit encoding of DALU container:



[1] All Unit values for both instructions are allocated to the DALU

64-bit encoding of DALU single/dual instruction:



[1] All Unit values are allocated to the DALU

6.2.2 Encoding scheme examples

The following sections describe several encoding schemes, according to the various instruction size.

The following provides a legend to the encoding schemes:

Da,Db,Dn	Data registers
Ra,Rb,Rn	Addressing registers
Rk	Offset register
PR	Predicate registers
PM	Post-modify field
Un	Immediate field of size n (unsigned)
Sn	Immediate field of size n (signed)

An, RelAddn Immediate field of size n (used as address)

CG Bit 15 of 32-bit DALU encapsulated instructions. Set to 0 in the first instruction (bits 63:32) and to 1 in the second instruction (bits 31:0).

Bits marked as ‘x’ are reserved and encoded as ‘1.’

6.2.2.1 16-bit encoding scheme

The 16-bit encoding currently contain only few PCU instructions (LSU and IPU instructions will be added in future revisions of the SC3900FP core).

instruction	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SWB	1	E	0	0	0	0	0	0	0	0	0	0	0	0	0	0
NOP (16-bit Suffix)	1	E	x	x	x	x	x	x	x	0	0	0	0	0	1	0

Figure 6-4. PCU and suffix 16-bit encoding formats examples

6.2.2.2 32-bit encoding scheme

The 32-bit encoding contains PCU, LSU, IPU and DALU-premium. The following are examples of a few instructions of each execution unit.

example instruction	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BRA RelAdd20	0	RelAdd20[1]															1	E	PR		N[2]	P[3]	RelAa…[1]				1	PCU				
JMP Rn	0	x	x	x	x	x	Sw-Rn[4]					x	x	opcode			1	E	PR		x	opcode						PCU				
RTE.CIC #u5	0	U5					x	x	x	x	x	x	x	x	x	x	1	E	PR		opcode							PCU				

[1] Relative address is a 19-bit field, with bits 18:15 encoded in bits 9:6 of the instruction and bits 14:0 encoded in bits 30:16 of the instruction

[2] NB field is NOBTB field. When cleared, the instruction is BRA and when set the instruction is BRA.NOBTB

[3] This instruction has the conditioned by extension bit encoded within the instruction (replacing the need for a suffix, in order to condition the instruction execution by predicates P4, P5 or P6).

[4] Swapped Rn is: Bit 4 of Rn encoded in bit 25 of the instruction, bit 3 of Rn encoded in bit 21 of the instruction and bits 2:0 of Rn encoded in bits 24:22 of the instruction

Figure 6-5. PCU 32-bit Encoding Formats Examples

example instruction	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SUBA Ra,Rb,Rn	0	Ra					Sw-Rb[1]					Rn					1	E	PR		opcode							LSU				
SUBA #u5,Ra,Rn	0	Ra					U5					Rn					1	E	PR		opcode							LSU				
ST.2L Da:Db,(Rn)	0	Rn					0	x	x	x	Da						1	E	PR		opcode							LSU				
ST.2L Ra:Rb,(Rn)PM	0	Rn					1	x	x	PM	x	Ra					1	E	PR		opcode							LSU				
LD.2L (Rn+Rk),Da:Db	0	Rn					Rk[2]				Da						1	E	PR		opcode							LSU				

[1] Swapped Rb is: Bit 4 of Rb encoded in bit 25 of the instruction, bit 3 of Rb encoded in bit 21 of the instruction and bits 2:0 of Rb encoded in bits 24:22 of the instruction.

[2] Offset register Rk can be either R0÷R7 or R16÷R23. To support this, Bit 4 of Rk encoded in bit 25 of the instruction, bit 3 of Rk is zero and bits 2:0 of Rb encoded in bits 24:22 of the instruction.

Figure 6-6. LSU 32-bit encoding formats examples

example instruction	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MPY32A.I Ra,Rb,Rn	0	Ra					Sw-Rb[1]					Rn					1	E	PR		opcode							IPU				
ASHA.RGT #u5,Ra,Rn	0	Ra					U5					Rn					1	E	PR		opcode							IPU				

[1] Swapped Rb is: Bit 4 of Rb encoded in bit 25 of the instruction, bit 3 of Rb encoded in bit 21 of the instruction and bits 2:0 of Rb encoded in bits 24:22 of the instruction.

Figure 6-7. IPU 32-bit encoding formats examples

example instruction	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NOP.DALU	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	1	E	PR		x	x	opcode					DALU				

Figure 6-8. DALU 32-bit 'premium' encoding formats examples

6.2.2.3 48-bit encoding scheme

The 48-bit encoding contains PCU, LSU and IPU instructions. The following are examples of a few instructions of each execution unit.

example instruction	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP a31	0	a31[1]															0	a31[1]															1	E	PR		N[2]	P[3]	a[1]	opc				PCU				

[1] Absolute address is a 32-bit field, with bits 31:17 encoded in bits 46:32 of the instruction, bits 16:2 encoded in bits 30:16 of the instruction, bit 1 encoded in bit 9 of the instruction and bit 0 is set to zero (program space is 16-bit word addressable, while address is defined in bytes).

[2] NB field is NOBTB field. When cleared instruction is BRA and when set instruction is BRA.NOBTB

[3] This instruction has the conditioned by extension bit encoded within the instruction (replacing the need for a suffix, in order to condition the instruction execution by predicates P4, P5 or P6).

Figure 6-9. PCU 48-bit encoding formats example

example instruction	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LD.B (SP+s16),Da:Db	0	s16[1]															0	x	x	x	x	x	s[1]	x	x	x	Da:Db						1	E	PR		opcode							LSU				
ST.B Ra,(Rn+s16)	0	s16[1]															0	Rn					s[1]	x	x	x	x	Ra					1	E	PR		opcode							LSU				
MOVE.L #s32,Dn	0	s32[2]															0	s32[2]									Da						1	E	PR		s32[2]								LSU[3]			

[1] Signed 16-bit field, with bits 15:1 encoded in bits 46:32 of the instruction, and bit 0 encoded in bit 25 of the instruction.

[2] Signed 32-bit field, with bit 31 encoded in bit 4 of the instruction, bits 30:22 encoded in bits 30:22 of the instruction, bits 21:15 encoded in bits 11:5 of the instruction and bits 14:0 encoded in bits 46:32 of the instruction.

[3] The unit field is only four bytes in this case. Meaning is that two consecutive 5-bit unit values are assigned to this instruction alone (and to the LSU).

Figure 6-10. LSU 48-bit encoding formats examples

example instruction	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CMPA.LE.U #u16,Ra,Pn	0	u16[1]															0	Ra					x	u[1]	x	x	x	F[2]	o	Pn			1	E	PR		opcode							IPU				

[1] Unsigned 16-bit field, with bits 15:1 encoded in bits 46:32 of the instruction, and bit 0 encoded in bit 25 of the instruction.

[2] IFEC - IF-Else-Clear field

Figure 6-11. IPU 48-bit encoding formats examples

6.2.2.4 64-bit encoding scheme

The 64-bit encoding contains single and dual DALU instructions. The following are examples of a few instructions of each.

example instruction

TFR #s32,Dn

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0	s32[1]															0	0	s32[1]													

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	x	x	x	x	x	s32[1]			x	x	Dn					1	E	PR		opcode							DALU[2]				

CMPEQ.L #s32,Da,Pm:Pn

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0	s32[1]															0	0	s32[1]													

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Da					s32[1]			x	x	F[3]	x	Pm:Pn			1	E	PR		opcode							DALU[2]				

[1] Signed 32-bit field, with bits 31:17 encoded in bits 62:48 of the instruction, bits 16:3 encoded in bits 45:32 of the instruction and bits 2:0 encoded in bits 25:23 of the instruction.

[2] All unit values of 64-bit encoding are DALU

[3] IFEC - IF-Else-Clear field

Figure 6-12. DALU 64-bit single instruction encoding format examples

FFTR4SN.8T Da,Dc:Dd,De:Df,Db,Dg:Dh,Di:Dj,Dx:Dy,Dm:Dn

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0	Dc:Dd					De:Df					Da					0	0	Dx:Dy[1]						Dm:Dn[1]						x	x

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Dg:Dh					Di:Dj					Db					1	E	PR		opcode							DALU[2]				

[1] Note that Dx:Dy and Dm:Dn are the 4th operand of each DALU, and as such does not have allocated expanding bit in the suffix. For this reason these operands are decoded as 6-bit field operands.

[2] All unit values of 64-bit encoding are DALU

Figure 6-13. DALU 64-bit dual (compound) instruction encoding format example

6.2.2.5 32-bit Encapsulated Encoding Scheme

The 32-bit encapsulated encoding contains single DALU instructions, that every couple of them (including encapsulated DALU NOP), can fit into one 64-bit DALU container. The following is an example of a 32-bit encapsulated DALU instruction.

example instruction	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MAC.X Da,Db,Dn	0	Da					Db					Dn					C[1]	E[2]	PR		opcode							DALU[3]				
ADD.L Da,#u5,Dn	0	Da					u5					Dn					C[1]	E[2]	PR		opcode							DALU[3]				
ABS.L Da,Dn	0	Da					x	x	x	x	x	Dn					C[1]	E[2]	PR		opcode							DALU[3]				

[1] CG - When encapsulated instruction is located in upper 32-bit word of a 64-bit container this bit is 0 and when encapsulated instruction is located in lower 32-bit word of a 64-bit container this bit is 1

[2] E - When encapsulated instruction is located in upper 32-bit word of a 64-bit container this bit is 1 and when encapsulated instruction is located in lower 32-bit word of a 64-bit container this bit is E (End of VLES, set if this instruction is the last in the VLES)

[3] All unit values of 32-bit encapsulating encoding are DALU

Figure 6-14. DALU 32-bit encapsulated encoding formats examples

Chapter 7
The Interlocked Pipeline

This chapter describes the interlocked pipeline of the SC3900FP DSP core.

7.1 Pipeline overview

The core pipeline consists of the fourteen pipeline stages shown in this table.

Table 7-1. Pipeline stages

SC3900FP pipeline stage	Symbol	Description
Program Address	P	Program memory address phase
Read Memory	R	The memory interface accesses program memory
Fetch	F	Program memory data phase
VLES Dispatch	V	• The core dispatches the VLES instructions to the different execution units • The core decodes the suffix instructions
Decode	D	The core decodes the instructions in the different execution units
Generate Address	G	• The core precalculates the indexed data addressing, and PC-relative COF destinations • The core processes AGU arithmetic and logical instructions for linear addressing • The core generates the next address for both data address (load/store) operations and program address (COF) operations
Address to Memory	A	• Data memory address phase (for both load and store) • The core processes AGU arithmetic instructions for non-linear addressing • The core finalize complex AGU arithmetic operations (AGU multiply and shift ISA)
Access Memory	C	The memory interface accesses data memory (load)
Sample Memory	S	Data memory data phase (load)
Multiply	M	• The core preprocesses and distributes the data (load) • The DALU performs the first portion of any Multiply or 3-cycle instruction • The core moves data between registers
Extended Multiply	Y	The DALU performs the second portion of any Multiply or 3-cycle instruction, or the first portion of a 2-cycle instruction
Execute	E	• The DALU performs the third portion of any Multiply or 3-cycle instruction, or the second portion of a 2-cycle instruction, or a single cycle instruction • The core completes all DALU operations (aside from floating point instructions) • The core performs data scaling and limiting and reorders bytes (store)
Write Back	W	Data memory data phase (store) The core completes floating point DALU operations
Late	L	Sampling of atomic result

DALU numerical processing takes three cycles (stages M, Y, E). DALU instructions that take less than three processing cycles are aligned to the end of the pipeline. For example, instructions that need only one DALU processing stage will execute at stage E.

The numerical floating point instructions (fadd.sp, fsub.sp, fmpy.sp, fmad.sp etc.) write their outputs in stage W.

L stage is used for the sampling of reservation result following a store conditional instruction.

7.2 PCU timing and speculation

7.2.1 Change of flow speculation

To achieve one-cycle execution for conditional COF instructions (no latency), the instructions executed by the core following the COF are determined by the prediction value in the BTB entry for that COF. In the case of a not-taken prediction (either miss or hit with not-taken prediction), the core executes the instructions immediately following the COF, while for a BTB hit with taken prediction, the core executes the instructions starting at the destination of the conditional COF. As part of the branch prediction mechanism, the core may execute the instructions after a conditional COF in such a way that the instructions can be cancelled if the COF resolution (the updating of the predicate bit) does not match the branch prediction mechanism (an incorrect prediction). This process is called speculation.

7.2.1.1 Terminology

This section defines some of the pertinent terms that will be used in the following sections.

Table 7-1. Terminology

Term	Definition
Conditional COF	COF instructions that depend on a predicate bit value to determine whether to change the flow to the destination or continue with serial execution.
COF taken	A COF that architecturally continues to the COF destination. By definition, all non-conditional COFs are taken.
COF not taken	A conditional COF that resolves so that execution continues in the serial flow, without jumping to the destination.
COF speculation	In the case of a conditional COF, for which the condition is not yet known, when it is in stage G, speculation allows the following VLESes in one of the two possible flows (taken or not taken) to continue execution after stage G, with the possibility of being aborted if the condition resolves for the COF to be taken in the other direction.
Prediction direction	The COF direction (taken or not taken) that is used in the COF speculation of conditional COFs is as follows (see the BTB document for the definition of the BTB terms): • Prediction to taken: in the case of a BTB hit taken. • Prediction to not taken: in the case of a BTB miss, or a BTB hit not taken.
Loops prediction	Loops that use the hardware loop BTB to assist in end of loop prediction in their work. This includes non-sequential loops and sequential loops with a size exceeding the fetch buffer size. Cases of mis-prediction may occur on the first iteration(s) of a loop when it is not in the BTB, on the last iteration when the loop termination predictor was wrong, and in cases in which the BTB entry is replaced in the middle of the loop (for example, interrupt/function call inside a loop that uses the same loop index).

7.2.1.2 Predicate-bit update timing

Speculation occurs when a conditional COF that requires the value of a predicate bit (Pn) in an early pipeline stage follows an instruction that updates a predicate bit in a late pipeline stage. The following instructions update a Pn bit in late pipeline stages (all DALU instructions that update predicate bits):

- DALU float/fixed point compare instruction (ex. CMP.EQ.X Da,Db,Pn)
- DALU test instructions (TSTBM)
- DALU DEC instructions
- DALU PCALC instructions (ex. PCALC #u8_t2,Pabc,Pn)

All instructions that update the predicates late in the pipeline are contained within the DAU_Pbit and FLT _Pbit tables.

No speculation is required when a Pn bit update occurs in an early pipeline stage. The following instructions update a Pn bit in the early pipeline stages (all AGU instructions that update predicate bits):

- AGU compare instructions (for example, CMPA.EQ Ra,Rb,Pn)
- AGU DECA instructions
- AGU predicate calculation instructions (for example, PCALCA #u8_t2,Pabc,Pn)
- AGU predicate swap/restore instructions (for example, PRDA.SWP #uena,#u12_t1,Rn)
- AGU BMTSTA instructions (for example, BMTSTA.C #u16,Ra.h,Pmn)
- Direct writes to the SR register either from an AGU register or an immediate value
- POP.L SR (Note that in this case there is a load to use stall for subsequent AGU usage)

7.2.1.3 Speculation situations

The core performs speculation on VLESes following a conditional COF instruction that is speculative (see the following description), and that is preceded by a DALU instruction that updated the relevant predicate bit (see Section 7.2.1.2, “Predicate-bit update timing”).

The core performs speculation for VLESes in the following cases:

- VLESes that follow a conditional COF instruction, when the COF is either a BTB miss or a BTB hit not taken
- VLESes at the target of a conditional COF instruction, when the COF is a BTB hit taken

Speculative PCU instructions (instructions that can be speculated under condition)[1] are as follows:

- JMP
- BRA
- JSR
- BSR
- RTS
- BREAK

[1] Note that some of these instructions (for example, BREAK) are not BTBable, and thus not taken prediction on a miss is the only speculation done for them.

- CONT
- SKIP

For non-speculative PCU instruction, the core generates interlock (hold them in stage G) until their condition is resolved and thus no speculation is required.

Non-speculative PCU instructions (instructions that cannot be speculated under condition):

- RTE
- RTS.STK
- DEBUGE.*n*
- DEBUGM.*n*
- SWB

The following PCU instructions cannot be predicted and, thus, they don’t break the speculation window (but they do incur a pipeline flush behind them):

— CLRIC, SYNC*, RTE.CIC

7.2.1.4 Cancelled speculation latency

When branch prediction for the conditional COF is correct, the speculated instructions are executed following the COF instruction and no latency occurs as a result of this COF.

If the prediction is incorrect, the core cancels the speculated instructions and the pipeline must be repopulated, resulting in a COF latency. Speculated memory transactions that get cancelled may increase the effective COF latency because of memory stall cycles such as cache miss penalty. Note that such cancelled speculated memory transactions can change the memory system state (for example, enter the cache thrashing another previously cached data).

In some cases (such as guarded accesses), the memory system actually stalls memory transactions until all core speculation has ended.

7.2.1.5 COF speculation window

The speculation window is the group of VLESes that are speculatively dispatched for execution after a conditional COF, while the predicate conditioning the COF remains unresolved. In this context, the predicate is resolved eight cycles after the DALU instruction that updated the predicate bit value, and thus the maximum length of the speculation window is 6 VLESes when the predicate bit is generated by a DALU instruction (subtracting the COF itself and at least one cycle interlock before it). The speculation window can be less when there is a greater distance between the DALU compare instruction and the speculative COF, or if the conditional COF and/or part of the speculated VLES have interlocks between them. Figure 7-2 gives some examples of speculation windows.

The following interlocks are generated when the predicate modification is performed by a DALU instruction:

- One cycle interlock is inserted between the predicate update and a conditional PCU instruction, which is conditioned by the same predicate bit when the COF does not modify the SP.

- Two cycle interlocks are inserted between the predicate update and the conditional PCU instruction when the COF modifies the SP (relevant for JSR,BSR and RTS).

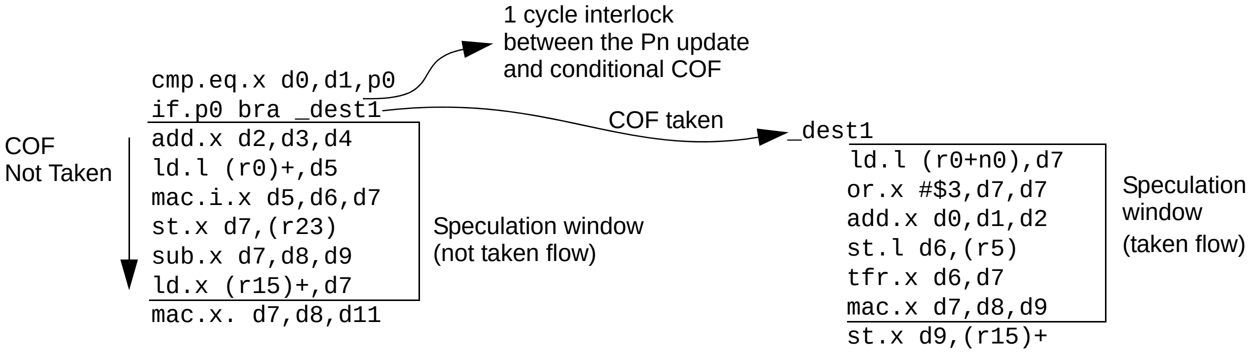


Figure 7-2. Example of COF speculation windows

7.2.1.6 Nested change of flow speculation

A COF executing inside the speculation window of a conditional COF will not start executing until the speculation is complete (end of the speculation window). As a result, this COF suffers up to six more interlock cycles.

7.3 IF.Pn AGU speculation

IF.Pn AGU speculation addresses the fact that, while DALU instructions that update a predicate value (for example, cmp.eq.x) do it late in the pipeline (that is, at stage E), conditional AGU instructions are required to read the predicates value early in the pipeline (at stage G). For example:

```
cmp.eq.x d0,d1,p0          ;DALU update p0 in stage E
if.p0 ld.l (r1),d3         ;AGU executes in stage G
```

Without a dedicated speculation mechanism, seven interlock cycles are needed to resolve the read after write hazard in the example above, in order to start the load transaction only if required. The IF.Pn AGU speculation mechanism bridges this gap for important use cases and, as a result, the core endures only two interlock cycles in order to resolve the hazard. The core actually starts the execution of the conditioned instructions speculatively, without waiting for the predicate value, and cancels the speculated instruction before it updates the programming model registers in the case of an incorrect speculation (similar to the PCU speculation mechanism. For more details, see Section 7.2, “PCU timing and speculation.”) Speculation of such an instructions sequence can continue until the predicate value is resolved, as long as the hardware can anticipate the input value for the speculated instructions (for example, when successive instructions have the same condition, the second instruction can assume the first instruction was executed, since this will be the case if the second instruction executes itself).

Speculation is required for conditional execution of AGU instructions (and PCU instructions that read AGU resources such as the Rn register value) that are following **all** predicate updates that originated from DALU instructions, as follows:

- DALU float/fixed point compare instruction (for example, CMP.EQ.X Da, Db, Pn)
- DALU test instructions (TSTBM)

- DALU DEC instructions
- DALU PCALC instructions (for example, PCALC #u8_t2,Pabc,Pn)

For information on minimum RSU interlocks on predicate bits, see Section 7.8.12.2, “Predicates (SR.Pn) interlock scheme.”

NOTE

> Conditional execution in the AGU, following the predicate update in the AGU and the conditional execution in the DALU, does not cause any interlock cycles

```
cmpa.eq r0,r5,p1
if.p1 adda r1,r2,r3        ; no stall

cmpa.eq r0,r1,p1
if.p1 add.x d1,d2,d3       ; no stall

cmp.eq.x d0,d1,p1
if.p1 add.x d1,d2,d3       ; no stall
```

7.3.1 IF.Pn AGU speculation principle

Immediately after a DALU predicate-updating instruction (see the list above), a two cycles interlock is incurred on a VLES with an AGU instruction that is conditioned with the updated predicate bit. The following provides an example:

```
cmp.eq.x d0,d1,p1
if.p1 adda r1,r2,r3        ; 2 interlock cycles
```

The conditioned AGU instruction starts executing speculatively after the two cycle interlock (note that PCU instructions that read an Rn register value following a DALU predicate update instruction incur 1÷2 cycle interlock, see Section 7.8.12.2, “Predicates (SR.Pn) interlock scheme” for more details).

At this point, a speculation window is opened for the instructions following the first conditioned instructions and remains open until the predicate value is resolved. The predicate is considered to be resolved for an AGU instruction either five cycles or eight cycles after the DALU predicate update instruction, according to the pipe stage in which the instruction reads its input value. For example, “`if.p1 adda r1,r2,r3`” reads R1 and R2 at stage A and thus eight cycles are required to resolve P1 value while “`if.p1 move.l r1,d2`” reads R1 at stage M and, thus, only five cycles are required to resolve P1 value. In the example “`if.p1 st.l r5,(r8)`” R5 is read at stage M while R8 is read at stage A. Therefore, the maximum size of the speculation window is either two VLESes or five VLESes (following the two cycle interlock and the first conditional instruction), according to the actual resource the AGU or PCU needs to read.

The speculation cannot be supported if one conditional AGU instruction updates an Rn register (either one VLES before the speculation window or in it) and another conditional AGU instruction reads the new value of the same Rn register with a different condition; and, thus, the hardware cannot predict the required value of Rn in order to proceed with the speculation.

When speculation cannot be supported, the core hardware terminates the speculation window, and inserts interlock cycles until the predicate is resolved (that is, five or eight cycles from the predicate update by a

DALU instruction). For more details, see Section 7.3.1.2, “Conditions for IF.Pn AGU speculation window termination.”

Instructions that either move Rn registers to Dn registers (MOVE.L Rn.Dn) or store Rn registers value to memory (ST* R*,(*)) have two VLESes speculation window while other AGU instructions (that can be speculative) have five VLESes speculation window.

The following instruction shows an example of a five VLESes speculation window:

```
cmp.eq.x d0,d1,p0
if.p0 adda r2,r3,r4 ;First speculated instruction; 2 interlock cycles
if.p0 ld.l (r0)+,d5 mac d5,d6,d7
if.p0 adda r7,r11
if.p0 ld.l (r0),d6                     ; 5 VLES speculation window
if.p0 tfra r3,r4
if.p0 ld.l (r4)+,d11
if.p0 ld.l (r4)+,d12    ;For this VLES the predicate is resolved
```

In the example above, the core hardware can speculate the first conditioned VLES (after the two cycle interlock) and the following conditioned VLESes that reside in the speculation window (since the hardware can predict the input value of the instructions under the p0 asserted condition).

The two interlocks VLES slots can be used for VLESes with DALU instructions, or for VLESes otherwise not conditioned by the un-resolved condition, without penalty, as demonstrated in the following example.

```
cmp.eq.x d0,d1,p0
if.p0 add.leg.x d5,d6,d6;VLES with only DALU inst.
adda r5,r6,r6;VLES not conditioned by p0
if.p0 adda r2,r3,r4 ;First speculated instruction; No interlock
if.p0 ld.l (r0)+,d5 mac d5,d6,d7
if.p0 adda r7,r11
if.p0 ld.l (r0),d6                     ; 5 VLES speculation window
if.p0 tfra r3,r4
if.p0 ld.l (r4)+,d11
if.p0 ld.l (r4)+,d12    ;For this VLES the predicate is already resolved
```

The following example shows instructions with only a two VLESes speculation window.

```
cmp.eq.x d0,d1,p0
if.p0 adda r2,r3,r4    ;First speculated instruction; 2 interlock cycles
if.p0 move.l r4,d6
if.p0 move.l r4,d7                      ; 2 VLES speculation window
if.p0 move.l st.l r4,(r8)     ;For r4 in this VLES the predicate is resolved
```

Note that, like any interlock due to a predicate value read, when the predicate update instruction is conditional as well, the two interlock cycles following it are inserted statically, disregarding the predicate value. This means that, as shown in the example below, even if the IF.P0 condition is false, an interlock will still be inserted.

```
if.p0 cmp.eq.x d0,d1,p1
if.p1 adda r1,r2,r3        ; 2 interlock cycles
```

Note that an implicit read of the Rn value due to a non-linear calculation (Bn and Mn implicit read) requires six cycle static interlocks (for details, see Section 7.8.3, “Address (Rn) register interlock scheme”), and thus generates no speculation window.

7.3.1.1 Static interlock (with no speculation) on predicate bits

Predicates bits, like any other resource, also have an explicit interlock (between DALU and AGU). For details, see Section 7.8.12.2, “Predicates (SR.Pn) interlock scheme.” In the case of such an instruction that has interlock due to Pn update, the following conditioned instructions already read a predicate with a resolved value and thus no speculation is required.

One example is the predicate manipulation instruction:

```
cmp.eq.x d0,d1,p0
pcalca p0,p1,p2           ;7 cycle interlock (p0 is written by DALU and read by AGU
if.p0 adda r1,r2,r3       ; not speculative - p0 is already resolved
```

In addition, certain instructions cannot execute speculatively and thus conditional execution of such instruction with an un-resolved predicate generate a static 7 cycle interlock from the instruction that updated the relevant predicate bit.

The core does not execute speculatively a conditional instruction, with an un-resolved condition in the following cases:

- AGU instructions that update a Pn bit (AAU_Pbit instructions).
- Load or store of BTRx register.
- PUSH*, and POP* instructions (all push and pop instructions, i.e. MOVE_STK instructions).
- Instructions that explicitly update any of the following registers (AAU_CTRL_REG instructions excluding TMDAT update, AAU_SP instructions that update SP, DOEN.n and SETALIGN Ra,GCR.BAM):
  - SR
  - SR2
  - EIDR
  - GCR
  - MCTL
  - PROCID
  - SP
  - TID
  - MOCR
  - TMTAG
  - LCn
  - LRx
  - CESRAx

As a result, any instruction from the above list, following a DALU instruction updating the same Pn bit, has a seven cycle interlock. In addition, the AGU conditional execution speculation is not performed for an instruction from that list.

Examples are as follows:

```
V1: cmp.eq.x d0,d1,p0
V2: if.p0 adda.lin #32,sp                  ;7 interlock cycles

V1: cmp.gt.x d15,d3,p2:p3
V2: if.p3 tfra r0,GCR                      ;7 interlock cycles

V1: cmp.ge.u.l d23,d2,p4
V2: if.p4 if.p0 st.2l btr0:btr1,(r2)+      ;7 interlock cycles

V1: cmp.le.l #$5fff,d8,p3:p4
V2: if.p3 push.4x d0:d1:d2:d3              ;7 interlock cycles

V1: cmp.eq.x d0,d1,p0
V2: if.p0 cmpa.eq r2,r3,p4                 ;7 interlock cycles
```

There are three exceptions to the above, as follows:

- B\JSR and RTS (SP update) **CAN** be speculated.

```
V1: cmp.eq.x d0,d1,p0
V2: if.p0 jsr #_label      ;2 interlock cycles
```

- CONT.n (LCn update) can be speculated.
- Instructions that change specific fields in control registers such as TMDAT

7.3.1.2 Conditions for IF.Pn AGU speculation window termination

In some cases, the speculated instruction uses an input operand, which was itself updated by a speculated instruction controlled by a different condition. In such cases, the core cannot determine what value to use for this operand, so the core stops speculation and inserts stall cycles until the condition predicating the generation of this operand resolves. For consecutive VLESes, the stall may be of 5 or 8 cycles, depending on the pipe stage where this operand is required[1] as input.

The following are examples:

```
V1: cmp.eq.x d0,d1,p0
V2: if.p0 tfra r0,r1       ;2 interlock cycles
V3: ld.l (r1),d0           ;stall until predicate resolution (5 cycles)
```

In the example above, the value of R1, used as a pointer in V3, is not known without first resolving the predicate bit; thus, the core cannot start its speculative execution. In this case, the core stalls on V3 for five cycles, until the P0 bit is resolved (since R1 is needed at stage A)[2].

```
V1: tfra r17,r1
V2: cmp.eq.x d0,d1,p0
 V3: cmp.eq.x d1,d5,p2
V4: if.p0 tfra r0,r1      ;1 interlock cycles
V5: if.p2 ld.l (r1),d0    ;5 interlock cycles until p0 is resolved
```

In the example above, the input value of R1 in the V5 instruction has two options: one from the tfra in V4, and one from V1, according to P0 resolution. Therefore, it is considered part of the P0 update five cycle speculation window, and once the speculation window is terminated, interlocks are inserted to guarantee eight cycles from V2 (p0 update) to V5. Note that, in this case, no speculation window break is done for P2.

```
V1: tfra r13,r1
V2: cmp.eq.x d0,d1,p0
V3: if.p0 tfra r0,r1       ;2 interlock cycles
V4: cmpa.eq r5,r29,p0
V5: if.p0 ld.l (r1),d0     ;3 cycles interlock speculation break
```

In the example above, p0 is modified twice: first in V2 and second in V4. In fact, these are two different conditions and, as a result, the value of r1 in V5 cannot be determined until the predicate P0 in V3 is resolved.

[1] Note that termination of the speculation is done only due to a previous speculated instruction that prevent the speculation of the current instruction, and thus speculation window start only after the first speculated VLES execution.

[2] In this extreme case, the distance between the compare instruction and the instruction that breaks the speculation is 8 cycles, which is more than the 7 cycles required for non-speculated instructions, but in most cases the speculation mechanism yields better cycles count.

To illustrate the meaning of the condition *AGU source register cannot be architecturally determined without knowing the predicate value* as a reason for the speculation break, consider a use case that behaves a little differently from the majority of AGU instructions. The data register, used as a source by the memory store instruction, is the reason for breaking speculation.

```
V1: cmp.eq.x d0,d1,p0
V2: if.p0 tfra r0,r5;2 interlock cycles
V3: st.b r5,(r29);speculation break 2 cycles interlock (short speculative win)
```

In the example above, R5 (read at V3) is read at stage S of the pipeline, unlike most arguments in the AGU, which are read at stage G. As a result, the speculation window required is only five cycles and the corresponding penalty is only two cycles.

The following provides an example:

```
V1: tfra r13,r1
V2: cmp.eq.x d0,d1,p0
V3: cmpa.eq r5,r29,p0
V4: if.p0 tfra r0,r1;1 interlock cycles
V5: ld.l (r1),d0;5 cycles interlock speculation break
```

In theory, the V4 condition is not affected by the V2 compare (since P0 is overwritten by V3). Due to corner cases and hardware implementation issues, the hardware will consider this case as a break of the speculation window from V2 and, as a result, enter a five cycle interlock before V5 execution.

7.4 Understanding subsystem latencies

NOTE

> This section deals with latencies that are caused by the subsystem and related to core instructions flow. This paragraph may change with subsystem implementation updates, even if the core is unchanged.

The SC3900FP core is guaranteed to be stalled by the subsystem in certain instructions flows, such as read after write latency and non-linear latency. The actual latency depends on subsystem implementation.

7.4.1 Read after write latencies

The SC3900FP core supplies the write data to the subsystem, in the case of a store instruction, at a very late pipe stage (stage W), while read data, in the case of a load instruction, is required by the core in an earlier pipe stage (stage S). As a result, the subsystem stalls the core when a load instruction is issued shortly after a store instruction to the same address (or transactions with overlapping bytes). The following table summarizes the number of stalls that the current subsystem asserts due to a load after store scenario.

Table 7-2. Read after write latency

Distance (in cycles) between load and store	Cache status during store (read cache and SGB)	Latency (stall cycles)
0	Hit in L1 cache (either read cache or SGB) [1]	4
1		3
2		2
3		1
≥ 4		0
< 6	Miss in L1 cache (both read cache and SGB)	Cache miss latency
≥ 6 [2]		0

[1] If all the bytes of the load transaction are “hit” in one of the caches when the store transaction is issued.

[2] If there are at least six cycles between the store and the load, the store enters the SGB and is thus a “hit” with zero penalty. Note that, if the entry in the SGB is evacuated (and not updating existing entry in the read cache), a later load (with distance greater than 6 cycles from the store) will suffer from cache miss penalty.

7.4.2 Unalignment latencies

The SC3900FP core supports the unaligned address of load and store transactions. When this kind of transaction crosses a 4 KByte boundary ($(\text{Address})/4096 \neq (\text{Address} + \text{Length} - 1)/4096$), one cycle stall is entered. Note that, in such a case, the unaligned transaction is performed as two transactions, each of which can cause stall cycles due to cache latencies (for example, a cache miss penalty).

Note that while for load with unalignment split the first part of the data is required in the first cycle and the second part of the data is required in the second cycle, the core supply the whole data of store with unalignment split at the second cycle[1].

7.5 Pipeline synchronization instructions

NOTE

> This section deals with the core pipeline synchronization ISA. For the memory synchronization ISA, see Section 4.13, “Memory barriers.”

The SC3900FP programming model contains control registers that influence the global behavior of the core, or influence the program fetch portion of the pipeline. For example, various fields in the MOCR register control the activation of speculation features.

When modifying these types of fields, the pipeline synchronization commands must be used to ensure that no pipeline effects will take place.

[1] The calculation of actual cycle count in case of unalignment split with other memory hold requirements such as read-after-write and with interlocks should take this behavior into account.

7.5.1 SYNC.D (SYNC Data)

By inserting seven interlock cycles between the instruction and the sequential VLES, this instruction ensures that the next VLES is not speculative.

7.5.1.1 SYNC.D use cases

The SYNC.D instruction must be executed in parallel to the following:

- Explicit write to EIDR (RTE* does not require SYNC.D)
- BM* instruction writing on MOCR.H registers

For a formal definition of these use cases, see Rule J.6 in Chapter 8, “Programming Rules and Guidelines”.

7.5.2 CLRIC instructions (Program synchronization)

The CLRIC #u5 instruction flushes the whole pipe (program + execution) and jumps to the next sequential PC from stage E. It also ensures synchronization between the execution pipe and the program pipe. In it’s basic form (u5 = 0), the CLRIC instruction performs only the pipe flush. For u5 values larger than 1, CLRIC performs also other internal clearing operations.

The CLRIC instruction takes 12 cycles to execute (1 + 11 bubbles).

NOTE

SYNC.P is an aliased to CLRIC #0. This mnemonic could be used in cases when only a pipe flush is required

7.5.2.1 Pipe flushing use cases

CLRIC #0 (SYNC.P) must be grouped with any instruction writing the following registers:

- CESRA0–1

CLRIC #1 or #3 must be grouped with any instruction writing the following registers:

- Any modification to MOCR.L

For a formal definition of these use cases, see Rule J.6 in Chapter 8, “Programming Rules and Guidelines”.

7.5.3 SYNC.B (Sync Before)

The SYNC.B instruction stalls the VLES in stage G until all previous VLESes are committed (seven cycles penalty).

7.5.3.1 SYNC.B use cases

The SYNC.B instruction must be executed in parallel to some special memory accesses, memory barriers and block cache commands. If the user uses the meta-instruction mnemonic, the barriers are automatically inserted by the assembler. The list of instructions that must be grouped with a SYNC.B is described in Rule A.9 in Chapter 8, “Programming Rules and Guidelines”.

7.5.4 SYNC.B.DL (Sync Before Destructive Load)

The SYNC.B.DL instruction stalls the VLES in stage G until all previous VLESes are committed (seven cycles penalty) and marks a load grouped with it as a destructive load (that is, destructive read), in order to guarantee correct memory system behavior. For more information on destructive loads, see Section 4.14.4, “Destructive loads.”

7.6 Pipeline interlocks

The SC3900FP DSP core has a fully interlocked pipeline that detects and solves resource conflicts and hazards through the operation of the resource stall unit (RSU). This section describes the read/write timing models for resources, allowing the use of this section for calculating the number of inserted interlock cycles. The timing models illustrated in this document may not match the implementation of a particular core exactly, but they do show the accurate interlock cycle calculation. For example, this model does not cover cases such as reading the value of a resource within the same VLES that modifies that resource. Resources within a VLES always follow the VLES semantics.

7.6.1 Conflict cases

This table lists resource conflict situations.

Table 7-3. Resource conflict cases

Conflict case	Description
Read after write	When the core writes to a resource and attempts to read from the same resource in the following VLES or VLESes. The conflict situation involves resolving whether the updated data is ready to be read. The distance in the read-write diagrams depicts the number of interlock cycles inserted. A distance of one, two, or three pipeline stages results in one, two, or three interlock cycles, respectively.
Write after read	When the core reads from a resource and attempts to write to the same resource in the following VLES or VLESes. In the SC3900FP DSP core, this case is resolved by the core hardware with no need for interlock cycles.
Write after write	When the core writes to a resource and attempts to write to the same resource again in the following VLES or VLESes. In the SC3900FP DSP core, this case is resolved by the core hardware with no need for interlock cycles.
Instruction spacing requirement	Specific, sporadic instructions utilize specific core/system resources, which do not support full core frequency throughput. In this case, subsequent instructions that utilize the same resources need to be stalled until the first instruction finishes its execution.

7.7 Instruction spacing interlock timing

The following sections describe specific scenarios in which instruction groups, following specific instructions, endure stall cycles to ensure the hardware instruction spacing requirement.

7.7.1 Memory Query spacing

All CME instructions immediately following P/DQUERY instructions endure one stall cycle to ensure the instruction spacing requirement.

7.8 Read after write resource pipeline interlock timing

The following sections illustrate the SC3900FP core pipeline interlock timing.

The table below shows an example of how interlocks are displayed in the document.

- Every table focuses on a single type of resource (for example, the Rn register).
- The horizontal headline contains the different types/occurrences of writing instructions of a register type.
- The right-most vertical column contains the different types/occurrences of reading instructions of the register type.
- The numbers in the table represent the minimum number of cycles that are needed to space between the write and the read of the related resource. When the program sequence executed by the core has fewer cycles than the listed number then the listed number, the RSU will insert bubbles accordingly, to accommodate for the required latency.
- The “0” entries in the table represent write/read combinations that do not incur any interlock in the pipeline.
  The following provides an example.

Table 7-3. Rn interlock scheme example

Rn writing instruction → Rn reading instruction ↓	move—memory destination	**adda**
ld/st—pointer read G	3 S	0 G
↑ Row’s Read Stage	↑ Column’s Write Stage	↑ Column’s Write Stage

```
ld.l (r1),r2
ld.l (r2),d1       ;In this case there will be 3 interlock cycles between the two VLESes.

adda r3,r4,r1
ld.l (r1),r2       ;In this case there would not be any interlock cycle
```

7.8.1 Instruction families

Each instruction is associated with a single instruction family that represents it. The different interlock classifications correspond to one or more families and are detailed, when required, after each table.

Note that the families are inclusive, meaning that not necessarily all the instructions in the family access the same resources. Information detailing which resource each instruction accesses is available in the ADL databases. Note that in ADL database, all AGU related instruction families start with “instr_tbl_agu_”, all DALU related instruction families start with “instr_tbl_dalu_” and all PCU related instruction families start with “instr_tbl_pcu_”, e.g. “AAU_MCTL” will appear as “instr_tbl_agu_AAU_MCTL”.

In latter spec releases, the families themselves will be included in an appendix. For this release, the families are available in the ADL database only.

Table 7-4. Instruction table

Unit/Type	Name	Description
AGU - Arithmetic and Logic Instructions	AAU	Linear arithmetic (add and subtract)
	AAU_LOGIC	Non-addition arithmetic (logic)
	AAU_SP	Explicit SP related arithmetic
	AAU_MCTL	Non-Linear arithmetic (add and subtract, based on MCTL configuration)
	AAU_2CYCLE	Two cycle linear arithmetic
	AAU_2CYCLE_MPYADDA	MPYADDA
	AAU_Pbit	Predicate modifying
	AAU_CTRL_REG	Explicit control register modification
AGU - Load instructions	MOVE_LD	Load
	MOVE_LD_PARTIALW	Load to partial Dn register (read modify write to the register)
	MOVE_LD_PRECALC	Pre-calculated load
	MOVE_LD_PRECALC_PARTIALW	Pre-calculated load to partial Dn register (read modify write to the register)
	MOVE_LD_POSTINC	Post update loads (endures MCTL effects)
	MOVE_LD_POSTINC_PARTIALW	Post update load to partial Dn register (read modify write to the register), endures MCTL effects
	MOVE_LD_cmd	Cache/barrier Load-Like ISA (Has the same behavior as MOVE_LD)
	MOVE_LD_nodat	Cache/barrier Load-Like ISA (Has the same behavior as MOVE_LD). Does not sample data from memory
AGU - Store instructions	MOVE_STK	Push and Pop
	MOVE_ST	Store
	MOVE_ST_PRECALC_IMM	Pre-calculated store with immediate offset
	MOVE_ST_PRECALC_R	Pre-calculated store with register offset
	MOVE_ST_POSTINC	post update stores (endures MCTL effects)
	MOVE_ST_cmd	Cache/barrier store-like ISA (Has the same behavior as MOVE_ST)
	MOVE_ST_nodat	Cache/barrier store-like ISA (Has the same behavior as MOVE_ST) that does not write data
AGU - Initialization and register move instructions	MOVE_REG_INIT	Register to register or initialization
	MOVE_REG_INIT_DD	D register to D register initialization
	MOVE_REG_INIT_IMM	Register initialization from immediate
	MOVE_REG_INIT_IMM_PARTIALW	Partial register initialization from immediate
AGU - Loop setup	LPSETUP	Loop setup

Table 7-4. Instruction table (continued)

Unit/Type	Name	Description
DALU - Integer Arithmetic and Logic instructions	DAU	Arithmetic Instructions
	DAU_Y	Arithmetic Instructions that read their non accumulator input in stage Y
	DAU_Pbit	Predicate modifying
	DAU_ACS2H	ACS2H instructions
	DAU_ACS2L	ACS2L instructions
	DAU_CNEGADDMD	CNEGADDMD instruction
	ADDM	Single mixed (S16+S40) addition
	LBM	Logical and Bit Manipulation Instructions
	LBM_Y	Logical and Bit Manipulation Instructions that read the size indicator in stage Y
	LBM_CARRY	Explicit carry handling
	MPY	multiplication
	MPY_Acc	MAC
DALU - Floating point Arithmetic Instructions	FLT_NUMERIC	numeric float
	FLT _Pbit	compare float
	FLT_TO_FIXED	float to fixed
	FLT_FROM_FIXED	fixed to float
	FLT_MISC	Miscellaneous float (ex. abs/exponent arithmetic)
PCU	COF	COF instructions
	COF_LP	Non sequential loop COFs
	COF_LPSEQ	Sequential loop COFs

7.8.2 Data Register (Dn) interlock scheme

Table 7-4. Dn interlock scheme

Dn writing instruction → **Dn reading instruction ↓**	FLT_NUMERIC FLT_FROM_FIXED	All DALU instructions (writing Dn), excluding: FLT_NUMERIC &FLT_FROM_FIXED	All AGU instructions (writing Dn)
MOVE_REG_INIT MPY MPY_Acc (Multiplicands) DAU_CNEGADDMD (Non accumulator) Non Src0 of FLT_NUMERIC [1] Non Src0 of FLT_FROM_FIXED [1] Non Src0 of FLT_TO_FIXED [1] M	3 W	2 E	0 M
MOVE_REG_INIT_DD [2] MOVE_REG_INIT_IMM_PARTIALW MOVE_LD_PARTIALW MOVE_LD_PRECALC_PARTIALW MOVE_LD_POSTINC_PARTIALW M	2 E	2 E	0 M
Src0 of FLT_NUMERIC [1] Src0 of FLT_FROM_FIXED [1] Src0 of FLT_TO_FIXED [1] M	3 W	3 W	0 W
DAU_ACS2H - Da and Dn (Viterbi dedicated) DAU_ACS2L - Da and Db (Viterbi dedicated) DAU_Y - non accumulator source operands LBM_Y - size specifier LBM_MASKSEL (New data) DAU_CNEGADDMD (accumulator) ADDM FLT_MISC, FLT _Pbit Y	3 L	1 E	0 M
MPY_Acc - Additive DAU DAU_Pbit DAU_Y - accumulator LBM, LBM_CARRY LBM_Y - non size sources LBM_MASKSEL (control and accumulator) E	3 L+1	0 E	0 M
MOVE_ST* (Memory store transactions) E	0 E	0 E	0 M

[1] The Src0 refers to the following operands: Db or Dc:Dd for add/sub floating instruction (FADD*, FSUB*, FADDSUB*, Dn or Dm:Dn (additive) of fused multiply addition/subtraction floating instruction (FMADD*, FMSUB*, FMSOD*) and for data (not scale) input of fix to float and float to fix conversion instructions.

[2] Note that, in the case of MOVE.4X Da,Db,Dc,Dd,Dmnop, the registers Da,Db,Dc and Dd should all be part of four consecutive registers (for example, if the lowest source register is D9 then other registers should be either D9, D10, D11 or D12), and even if not all registers out of this consecutive quartet are actually used, all of them are considered as sources of the instruction (for example, for “MOVE.4X D10,D9,D12,D10,D22::D25”, D11 will be considered as a source for interlock calculation even though it is not actually used by the instruction or explicitly mentioned in it).

7.8.2.1 Dn Register interlock examples

- Floating Point numerical instructions output

```
fadd.sp d1,d2,d3
fadd.sp d3,d4,d5  ;3 interlock cycles

fadd.sp d1,d2,d3
move.l d3,d5      ;3 interlock cycles

fadd.sp d1,d2,d3
move.l d3,r5      ;2 interlock cycles

fadd.sp d1,d2,d3
fcmp.eq.sp d3,d4,p0;3 interlock cycles

fadd.sp d1,d2,d3
fabs.sp d3,d4     ; 3 interlock cycles

fadd.sp d1,d2,d3
tfr.x d3,d4       ; 3 interlock cycles

fadd.sp d1,d2,d3
st.l d3,(r0)      ;no interlock cycles
```

- All DALU instructions (that write Dn), excluding FP numerical instructions

```
add.x d1,d2,d3
extract.x d3,d4,d5;1 interlock cycle

add.x d1,d2,d3
extract.u.x d4,d3,d5;no interlock cycles

add.x d1,d2,d3
mpy.x d3,d4,d5    ;2 interlock cycles

fabs.sp d1,d2,d3
fcmp.eq.sp d3,d4,P0;3 interlock cycles

mac.leg.x d1.h,d2.h,d3
mac.leg.x d4.h,d5.h,d3;no interlock cycles

mac.leg.x d1.h,d2.h,d3
mac.leg.x d4.h,d3.h,d5;2 interlock cycles
tfr.x d1,d2
fadd.sp d2,d4,d5  ;3 interlock cycles (src0 example)

tfr.x d1,d2
fadd.sp d4,d2,d5  ;2 interlock cycles
```

7.8.3 Address (Rn) register interlock scheme

In both address arithmetic and AGU addition, the input operands are divided into two groups, as follows:

- Pointer—The base address from which the offset is incremented/decremented (first operand) for address calculation of load and store instructions and the second source for addition instructions.
- Offset—The stride size from the pointer value (second operand) for address calculation of load and store instructions and the first source for addition instructions.

This has significance for two types of instructions, as follows:

1. AAU_MCTL—non linear arithmetic in which the second operand is the pointer and the first operand is the offset. The following provides an example:

```
adda r5, r6, r7; r5 - offset r6 - pointer
```

2. Address calculation of load and store instructions, as follows:

```
ld.l (r5)+r7, d0; r7 - offset r5 - pointer
st.l (r6+r8), r15; r8 - offset r6 - pointer
ld.w (r3), d7; r3 - pointer (no offset)
```

Instructions that do not read MCTL are linear for the purpose of the table above.

Legacy Nn registers behave like any Rn offset register.

This table shows the Rn interlock scheme.

Table 7-5. Rn interlock scheme

Rn writing instruction → **Rn reading instruction ↓**	Destination of 2-cycle (excluding MPYADDA) [1]	Destination of 2-cycle MPYADDA [2]	Destination of non-addition, linear addition or linear address update [3]	Destination of non-linear addition or non-linear address update [4]	Destination of load instructions[5]	Destination of move instructions[6]
Source of data store or move instructions [7] S	0 A	0 A	0 G	0 A	0 S	0 S
Source of non-addition, 2-cycle and COF instructions [8] G	1 A	1 A	0 G	1 S	3(4[9]) S(M)	4 M
Source of linear addition, offset of non-linear addition, address with pre-calc, offset of address with post-calc and loop setup[10] G	1 A	1 A	0 G	1 A	3 (4[9]) S(M)	3 S
Source of address without pre or post calculation for load or store instructions [11] G	1 A	0 G	0 G	0 G	3 (4[9]) S(M)	3 S
Source of pointer of non-linear addition and pointer of address with post calc [12] G	1 A	1 A	0 G	0 G	3 (4[9]) S(M)	3 S
Bn - Implicit modulo base address Mn - Implicit Buffer size G	6 E	6 E	6 E	6 E	6 E	6 E

[1] Destination of two cycle AGU AKU instructions, excluding MPYADDA, that are part of AAU_2CYCLE.

[2] Destination of MPYADDA instruction, which is part of AAU_2CYCLE_MPYADDA (has a forwarding path to address with no pre/post calculation).

[3] - Destination of instruction in the following groups is Rn in linear calculation mode: AAU, AAU_LOGIC, AAU_MCTL (when relevant MCTL field is in linear mode), AAU_Pbit and AAU_SP.
- Address of load/store instruction with post calculation in the following groups is Rn in linear calculation mode (when relevant MCTL field is in linear mode): MOVE_LD_POSTINC, MOVE_LD_POSTINC_PARTIALW and MOVE_ST_POSTINC.
- Destination of move instruction is Rn: MOVE_REG_INIT_IMM.

[4] - Destination of instruction in the following groups is Rn in non-linear calculation mode: AAU_MCTL (when relevant MCTL field is in non-linear mode).
- Address of load/store instruction with post calculation in the following groups is Rn in non-linear calculation mode (when relevant MCTL field is in non-linear mode): MOVE_LD_POSTINC, MOVE_LD_POSTINC_PARTIALW and MOVE_ST_POSTINC.

[5] Destination of instruction in the following groups is Rn: MOVE_LD* and MOVE_STK.

[6] Destination of instruction in the group MOVE_REG_INIT is Rn.

[7] Source of data (not address) for store instructions and source of move instructions in the following groups: MOVE_ST* (stored data), MOVE_REG_INIT and MOVE_STK (pushed data).

[8] Source of the following groups:
- Sources of non-addition instructions: AAU_LOGIC, AAU_Pbit and AAU_SP, AAU_2CYCLE and AAU_2CYCLE_MPYADDA
- Source of COF (with Rn source) instruction - COF*

[9] Load instructions that read four 32 bits operands to the Rn register file (ld.4l->Rn, pop.4l->Rn) have another interlock cycle (four cycles interlock) between the load and the usage of the two high destination registers (three for the lower two registers).

[10] Source of the following groups:
- Sources of linear only addition instructions: AAU, AAU_MCTL (when relevant MCTL field is in linear mode)
- The “offset” source (first operand) of non-linear addition (by mode): AAU_MCTL (relevant MCTL field is in non-linear mode)
- The “pointer” and “offset” of address calculation for load and store instructions with pre-calculation: MOVE_LD_PRECALC, MOVE_LD_PRECALC_PARTIALW and MOVE_ST_PRECALC
- The “offset” of address calculation (second argument) for load and store instructions with post-calculation: MOVE_LD_POSTINC, MOVE_LD_POSTINC_PARTIALW and MOVE_ST_POSTINC
- The source of Loop setting instructions (with Rn source): LPSETUP

[11] Source of address with no pre-calculation or post calculation (“pointer” only)—“(Rn)”: MOVE_LD and MOVE_ST.

[12] - The “pointer” source (second operand) of non-linear addition (by mode): AAU_MCTL (relevant MCTL field is in non-linear mode).
- The “pointer” of address calculation (first argument) for load and store instructions with post-calculation: MOVE_LD_POSTINC, MOVE_LD_POSTINC_PARTIALW and MOVE_ST_POSTINC

7.8.3.1 Rn Register interlock examples:

- Load-to-use / Move-to-use

```
ld.l (r0),r1
adda r1,r2,r3     ;3 interlock cycles

ld.4l (r0),r0:r3
adda r2,r6,r7     ;4 interlock cycles

move.l d1,r3
adda r3,r4,r5     ;3 interlock cycles

move.l d3,r5
anda r5,r6,r7     ;4 interlock cycles
```

- Modify-to-use

```
adda.lin r1,r2,r3
jmp r3            ;0 interlock cycle

mpy32a.l r3,r4,r5
adda r1,r5,r3     ;1 interlock cycle

adda r1,r2,r3     ;
tfra r3,r4        ;no interlock

mpyadda.i #u16_t6,R1,R2,R3
lb.b (R3),d0      ;no interlock

mpyadda.i #u16_t6,R1,R2,R3
lb.b (R3+R4),d0   ;1 interlock cycle
```

For non linear examples see:
Section 4.7, “Interlocks for linear and modulo arithmetic instructions”

7.8.4 Back Trace Register (BTR0, BTR1) interlock scheme

NOTE

BTR does not generate any read/write conflicts.

Table 7-6. BTRn interlock scheme

BTRn Writing Instruction → **BTRn Reading Instruction ↓**	MOVE_LD_* (Memory Load) move.l Da,C3 MOVE_STK (pop)	DAU_ACS2H DAU_ACS2L
MOVE_ST* (Memory Store) MOVE.L C3,Dn MOVE_STK (push) E	0 M	0 E
DAU_ACS2H DAU_ACS2L E	0 M	0 E

7.8.5 General Configuration Register (GCR) interlock scheme

Table 7-7. GCR interlock scheme

GCR Writing Instruction → **GCR Reading Instruction ↓**	SETALIGN, TFRA BM[SET\CLR\CHG]A	MOVE_STK (pop)	DIVP.n, instr_tbl_FLT* FLOG2 FINVSQRT FRECIP
DIVPn DOALIGN E	0 E	0 E	0 E
MOVE_STK (push) E	0 E	0 E	0 L
BMTSTA BM[SET\CLR\CHG]A TFRA G	6 E	6 E	8 L

7.8.5.1 GCR interlock examples

```
fadd d0,d1,d2
bmtsta.s #$ffff,gcr.h;8 interlock cycles

divp.0 d0,d1
push gcr                   ;no interlock cycles

divp.0 d0,d1
bmtsta.s #$0001, gcr       ;no interlock cycles

divp.0 d0,d1
divp.0 d0,d1               ;no interlock cycles

pop.l gcr
bmtsta.s #1,gcr,p0         ;6 interlock cycles
```

7.8.6 TMDAT

Write-only register (no interlocks).

7.8.7 Modifier Control (MCTL) register interlock scheme

Table 7-8. MCTL interlock scheme

MCTL writing instruction → **MCTL reading instruction ↓**	POP, TFRA BM[SET\CLR\CHG]A
Implicit MCTL read [1] BM[SET\CLR\CHG]A BMTSTA TFRA G	6 E
PUSH E	0 E

[1] Includes the following groups: AAU_MCTL, MOVE_LD_POSTINC, MOVE_LD_POSTINC_PARTIALW and MOVE_ST_POSTINC

All the instructions that write to Rn can be either Bn or Mn (R8÷R15 or R24÷R27).

A full list can be found at the first row (Rn Writing Instructions) of table "Rn interlock scheme" on page 7-19

7.8.7.1 MCTL interlock examples

```
tfra r22,mctl
ld.l (r1)+n0,r0   ;6 interlock cycles

pop.l mctl
adda r1,r0,r0     ;6 interlock cycles

pop.l mctl
bmclra #7,mctl    ;6 interlock cycles
```

7.8.8 Stack Pointer (SP) register interlock scheme

The SP does not have interlock cycles associated with it. There are interlocks associated with the SP selection (see Section 7.8.13.3, "SR2: ASPSEL fields in the SR2").

The SP explicit conditional modification (for example, IF.P0 ADDIA #S10_3,SP) breaks AGU speculation (see Section 7.3.1, "IF.Pn AGU speculation principle").

7.8.9 Loop Counter (LCn) interlock scheme

Table 7-9. LCn interlock scheme

LCn writing instruction → **LCn reading instruction ↓**	DOEN.n TFRA	CONT.n	LPEND.n	LPEND.SQn[1]
LPEND.n G	0 G	0 G	0 G	x G
LPEND.SQn[1] V	2 G	0 V	x	0 V

Table 7-9. LCn interlock scheme (continued)

LCn writing instruction → **LCn reading instruction ↓**	DOEN.n TFRA	CONT.n	LPEND.n	LPEND.SQn[1]
SKIP.n LPSKIP.n CONT.n G	0 G	0 G	0 G	0 V
TFRA G	6 E	6 E	6 E	6 E

[1] LPEND.SQn interlocks are introduced in stage V by the PDU, unlike the rest of the interlocks inserted by the RSU in stage G, and, as a result, they may not merge with stage G stalls in all cases.

7.8.9.1 LCn interlock examples

```
doen.0 r0
lpend.sq.0       ;2 interlock cycles

lpend0
tfra lc0,r0      ;6 interlock cycles
```

7.8.10 Link Register (LRx)

Table 7-10. LRx interlock scheme

LRx writing instruction → **LRx reading instruction ↓**	TFRA	POP
PUSH E	0 G	0 E
TFRA G	6 E	6 E

Interlocks are incurred on an LRx portion (PC/SR/SR2/EIDR) read/write register. Only relevant stalls will be incurred.

7.8.10.1 LRx interlock examples

```
tfra r5,lr0.sr2
tfra lr0.sr,r0    ;no interlock cycles

pop.4l lr0
tfra lr0.pc,r0    ;6 interlock cycles
```

7.8.11 Task ID Register interlock scheme

TID.PID and TID.DID can only be explicitly modified when the core is in exception mode (SR2.EXP = 1), and is implicitly relevant (for MMU) only when the core is not in exception mode (SR2.EXP = 0). The transition between the two is protected with the RTE instruction. As a result, no interlock exist between update of this register and the implicit use of it by memory transactions of any kind, excluding the use of ACATOR.P instruction.

Table 7-11. TID interlock scheme

TID writing instruction → **TID reading instruction ↓**	POP	TFRA
PUSH E	0 E	0 E
ACATOR.P TFRA G	6 E	6 E

7.8.12 Status Register (SR) interlock scheme

The SR is also updated by the RTE instruction. However, as a result of the fact that, in RTE, the entire pipeline is flushed, no additional interlocks are needed.

7.8.12.1 Carry Bit (SR.C) interlock scheme

Table 7-12. SR.C interlock scheme

SR.C writing instruction → **SR.C reading instruction ↓**	DAU LBM	LBM_CARRY	POP	TFRA BM[SET\CLR\CHG]A
DAU LBM E	0 E	1 W	0 E	0 G
TFRA BM[SET\CLR\CHG]A BMTSTA G	8 L	8 L	6 E	0 G
PUSH Implicit PUSH (JSR/BSR) E	0 E	0 E	0 E	0 G

7.8.12.1.1 SR.C interlock examples

```
rol.x d0
bmtsta.s #5,sr,p1          ;8 interlock cycles

addc.rw.l d0,d1,d2
addc.ro.l d3,d4,d5         ;0 interlock cycles

addc.wo.l d0,d1,d2
rol.x d2                   ;0 interlock cycles

extc.lft #16,d5
rol.x d2                   ;1 interlock cycle
```

7.8.12.2 Predicates (SR.Pn) interlock scheme

Table 7-13. SR.Pn interlock scheme

SR.Pn writing instruction → **SR.Pn reading instruction ↓**	AAU_Pbit AAU_CTRL_REG	DAU_Pbit FLT_Pbit	pop.l sr
Conditional execution of DALU instruction	0	0	0
DAU_Pbit FLT_Pbit	0	0	0
Speculative Conditional execution of AGU instruction (not in non-speculative list [1])	0	2 [2]	6
Non-Speculative Conditional execution of AGU instruction (in non-speculative list [1])	0	7	6
Speculative conditional execution of BSR, JSR, CONT or RTS instruction	0	2 [3]	6
Speculative conditional execution of BRA, JMP, BREAK or SKIP instruction	0	1 [3]	6
Non-Speculative conditional execution of COF instructions [4]	0	7	6
AAU_Pbit [5] AAU_CTRL AAU	0	7	6
MOVE_STK(push[c].l)	0	0	0

[1] Following instructions cannot execute speculatively, when conditionally executed with unresolved predicate bit (For more details see section "IF.Pn AGU speculation" on page 7-5):
- AGU instructions that update a Pn bit (AAU_Pbit)
- Load or store of BTRx register
- PUSH*, and POP* instructions (MOVE_STK)
- Instructions that explicitly update some control registers (AAU_CTRL_REG excluding TMDAT update, AAU_SP updating SP, DOEN.n and SETALIGN Ra,GCR.BAM)

[2] This case applies for conditional execution of an AGU instruction (IFPn AGU) after Pbit setting by a DALU instruction. The number represents the minimum interlock penalty. Additional stalls may be incurred. For more details see section "IF.Pn AGU speculation" on page 7-5.

[3] This case applies for speculative conditional execution of a COF instruction after Pbit setting by a DALU instruction. The number represents the minimum interlock penalty. Additional stalls may be incurred due to incorrect prediction. For more details see section "PCU timing and speculation" on page 7-2.

[4] COF executes as non-speculation (with interlocks), if it is either a non-speculative COF instruction (RTE*, DEBUG*, SWB) or encoded as non-speculative. For more details see section "PCU timing and speculation" on page 7-2.

[5] For PRDA.SAV and PRDA.SWP, Each Pn bit is read (and thus may cause an interlock) only if bit "n" in "UENA" immediate field is asserted.

For PRDA.RST, Each Pn bit is read (and thus may cause an interlock) only if bit "n" in "UENA" immediate field is asserted, and the two related bits in "U12" immediate field have a 0b11 value.

7.8.12.2.1 Predicate interlock examples

Conditional DALU execution

```
cmpa.eq r0,r1,p1
if.p1 add.x d1,d2,d3        ; no stall
```

```
        cmp.eq.x d0,d1,p1
        if.p1 mac.leg.x d1.h,d2.h,d3 ; no stall
```

DAU_Pbit

```
        cmpa.eq r0,r1,p1 cmp.eq.x d0,d0,p2
        pcalc p1,p2,p3              ; no stall
```

Conditional AGU execution

```
        cmpa.eq r0,r5,p1
        if.p1 adda r1,r2,r3         ; no stall

        cmp.eq.x d0,d5,p1
        if.p1 adda r1,r2,r3         ; 2 interlock cycles

        pop.l sr
        if.p1 ld.l (r0)+,d1         ; 6 interlock cycles
```

Speculative Conditional PCU execution

```
        cmpa.eq r0,r1,p1
        if.p1 bra _label            ; no stall

        cmp.eq.x d0,d1,p1
        if.p1 bra _label            ; 1 interlock cycles

        cmp.eq.x d0,d1,p1
        if.p1 jsr _label            ; 2 interlock cycles
```

Non-Speculative Conditional PCU execution

```
        cmp.eq.x d0,d1,p1
        if.p1 trap.0 #15            ; 7 interlock cycles
```

AAU_Pbit

```
        cmp.eq.x d0,d1,p1
        pcalca #5,p1,p2,p3,p5       ; 7 interlock cycles
```

7.8.12.3 Scaling (SR.S) interlock scheme

Table 7-14. SR.S interlock scheme

SR.S writing instruction → **SR.S reading instruction ↓**	MOVE_ST* (Memory store) that updates SR.S	POP	TFRA BM[SET\CLR\CHG]A
PUSH Implicit PUSH (JSR/BSR) E	1 W	0 E	0 G
BM[SET\CLR\CHG]A BMTSTA TFRA G	9 L+1	6 E	0 G

7.8.12.3.1 SR.S interlock examples

```
        st.srs.l d4,(r4)+
        tfra sr,r5        ;9 interlock cycles

        st.srs.l d4,(r7)+
        push.l sr         ;1 interlock cycles

        pop.l sr
        bmclra #$6,sr.l   ;6 interlock cycles
```

7.8.12.4 SR DALU mode bits: SM, SM2, W20, and RM interlock scheme

Table 7-15. SR.SM/SR.SM2/SR.W20/SR.RM interlock scheme

SR.xx writing instruction → **SR.xx reading instruction ↓**	POP	TFRA BM[SET\CLR\CHG]A CHCF
PUSH Implicit PUSH (JSR/BSR) E	0 E	0 G
st.srs instr_tbl_dalu* (implicit DALU usage) Y	0 E	0 G
BM[SET\CLR\CHG]A BMTSTA TFRA CHCF [1] G	6 E	0 G

[1] CHCF instructions operate in a read modify write manner. All CHCF instructions read all the SR arithmetic mode fields (SM, SM2, W20, RM and SCM) and writes only the relevant updated fields.

7.8.12.4.1 SR.SM/SR.SM2/SR.W20/SR.RM interlock examples

```
pop.l sr
rnd.leg.x d0,d1   ;no interlock cycle

pop.l sr
tfra sr,r0        ;6 interlock cycles

addc.wo.l d1,d2,d3;carry generation
chcf #1,sr.rm     ;0 interlock cycles
```

7.8.12.5 SR.SCM interlock scheme

Table 7-16. SR.SCM interlock scheme

SR.SCM writing instruction → **SR.SCM reading instruction ↓**	POP	TFRA BM[SET\CLR\CHG]A CHCF
PUSH Implicit PUSH (JSR/BSR) E	0 E	0 G
st.srs—MOVE_ST* Y	0 Y	0 G
MPY, MPY_Acc M	1 Y	0 G

Table 7-16. SR.SCM interlock scheme (continued)

SR.SCM writing instruction → **SR.SCM reading instruction ↓**	POP	TFRA BM[SET\CLR\CHG]A CHCF
DAU* LBM* E	0 E	0 G
BM[SET\CLR\CHG]A BMTSTA, TFRA CHCF[1] G	6 E	0 G

[1] CHCF instructions operate in a read modify write manner. All CHCF instruction read the all the SR arithmetic mode fields (SM, SM2, W20, RM and SCM) and writes only the relevant updated fields.

7.8.12.5.1 SR.SCM interlock examples

```
pop.l sr
mpy.leg.x d1.h,d2.h,d3;1 interlock cycle

pop.l sr
st.srs.l d4,(r0)  ;1 interlock cycle

pop.l sr
chcf #1,rm        ;6 interlock cycles
```

7.8.12.6 SR OVF/SAT interlock scheme

Table 7-17. SR.SAT interlock scheme

SR.OVF/SAT writing instruction → **SR.OVF/SAT reading instruction ↓**	POP	BM[SET\CLR\CHG]A TFRA	DALU OVF/SAT implicit write[1]
PUSH Implicit PUSH (JSR/BSR) E	0 E	0 G	0 E
BM[SET\CLR\CHG]A BMTSTA TFRA G	6 E	0 G	8 L

[1] Contains all the non-float DALU ISA and FLT_TO_FIXED instructions.

7.8.12.6.1 SR SAT interlock examples

```
add.leg.x d1,d3,d4
tfra sr,r5        ;8 interlock cycles
```

7.8.12.7 SR.RF (Reservation Failed) interlock scheme

Table 7-18. SR.RF interlock scheme

SR.RF writing instruction → **SR.RF reading instruction ↓**	POP	LDRSTC Usage	TFRA BM[SET\CLR\CHG]A
JMPRF,PUSH Implicit PUSH (JSR/BSR) Y	0 Y	3 L	0 G
BM[SET\CLR\CHG]A BMTSTA TFRA G	6 E	9 L+1	0 G

7.8.12.7.1 SR.RF interlock examples

```
_lbl:
        ldrstc st.l r1,(r3)
        jmprf  _lbl                 ;3 interlock cycles in case of not taken

        ldrstc st.l r1,(r3)
        bmtsta.s  #$8,sr.l,p1       ;9 interlock cycles
```

7.8.13 Status Register 2 (SR2) interlock scheme

7.8.13.1 SR2 Update From Pop instruction

Pop is the only instruction that can update the SR2 directly from memory.

The update is done from stage W and, as a result of the update, seven interlock cycles are incurred between pop SR2 and the sequential VLES.

7.8.13.2 SR2: IDE/EXP/TE

The IDE/EXP/TE fields of the SR2 can be updated only upon interrupt and RTE instruction.

In both cases, the pipe flush accompanying the write removes the need for RSU stalls.

7.8.13.3 SR2: ASPSEL

Table 7-19. SR2: ASPSEL interlock scheme

ASPSEL writing instruction → **ASPSEL reading instruction ↓**	BM[SET\CLR\CHG]A TFRCA, CHCF
TFRCA G	0 G
BMCHGA BMTSTA G	0 G
ASPSEL usage [1] D	1 G

[1] MOVE_LD\ST_PRECALC*, AAU_SP which read ASPSEL (TFRC,POPC,PUSHC)

Note that SPSEL can only be modified by RTE or interrupt (which are pipeline flushed protected), so it does not have stall associated with it.

7.8.13.3.1 SR.ASPSEL interlock examples

```
chcf #2,sr2.asp
tfrca sp,r9       ;1 interlock cycle
```

7.8.13.4 SR2: IPM/DI/DCI/DIN/DMN/DDN/ILL/IRR

NOTE

Modification of SR2 IPM/DI/DCI/DIN/DMN/DDN/ILL/IRR fields do not incur any interlock cycles.

Table 7-20. SR2: DI/DCI/DIN/DMN/DDN/ILL/IRR interlock scheme

IPM/DI/DCI/DIN/DMN/DDN/ILL/IRR writing instruction → **IPM/DI/DCI/DIN/DMN/DDN/ILL/IRR reading instruction ↓**	TFRA BM[SET\CLR\CHG]A CHIPL
PUSH E	0 G
TFRA BMCHGA BMTSTA G	0 G

7.8.14 Process ID Register (PROCID) interlock scheme

PROCID can only be explicitly modified when the core is in exception mode (SR2.EXP = 1) and is implicitly relevant (for DTU) only when the core is not in exception mode (SR2.EXP = 0). The transition between the two is protected with RTE instruction.

Table 7-21. PROCID interlock scheme

PROCID writing instruction → **PROCID reading instruction ↓**	POP TFRA
PUSH E	0 E
TFRA G	6 E

7.8.15 TMTAG register interlock scheme

Table 7-22. TMTAG interlock scheme

TMTAG writing instruction → **TMTAG reading instruction ↓**	POP TFRA BM[SET\CLR\CHG]A
PUSH E	0 E
TFRA BMCHGA BMTSTA G	6 E

Chapter 8
Programming Rules and Guidelines

The SC3900FP core instructions share core resources, so the core has restrictions on its ability to group or sequence certain instructions that activate various execution units. These restrictions are expressed as programming rules and guidelines. Software applications must follow these rules and guidelines to ensure that the application produces the expected results and is compatible with future processor implementations. The SC3900FP core assembler and simulator assist the programmer in complying with these programming rules. This chapter describes the SC3900FP core programming rules and guidelines.

8.1 Terminology and conventions

This section describes some conventions used in the this chapter.

8.1.1 Referencing instruction groups

8.1.1.1 Mnemonic wildcard

The SC3900FP mnemonic naming system includes a base mnemonic, which specifies the basic functionality followed by flags in three hierarchies, separated by dots, which specify variants of the instruction. For example: ADD.S.X. The ADD base mnemonic identifies the instruction as performing addition, while the “S” flag specifies it performs saturation, and the “X” flag specifies it operates on 40-bit data and destination.

When relating to the whole group of instructions which share the same base mnemonic, the “*” symbol can be used as a wildcard that captures all the instructions which share the same base mnemonic. For example, RTS* stands for:

- RTS
- RTS.NOBTB
- RTS.STK

Note that the “*” captures also the dot itself, as RTS does not have a dot in the mnemonic. Additional examples:

LD* captures LD.L and LD2.2W, but LD.* does not include LD2.2W.

8.1.1.2 Control (CTL) registers

Control registers (or CTL registers) are all core registers that are specified in Section 2.1, “Description of the SC3900FP registers,” and are not D registers, R registers or SP registers (TSP, ESP and DSP).

8.1.1.3 COF instructions

This group of instructions includes instructions that perform an explicit Change Of Flow. It does not include the LPEND* instructions that are not exposed in the assembly. In case a programming rule includes also them, they are mentioned.

The COF instruction group is: JMP*, BRA*, JSR*, BSR*, RTS*, RTE*, BREAK*, CONT*, SKIP*, LPSKIP*, TRAP*, DEBUGE.*

8.1.1.4 BTB-able COF instructions

This instruction group is a subset of the COF instruction group, and includes instructions that are stored and accelerated in the BTB. This group includes: JMP <label>, BRA, JSR <label>, BSR, RTS (note that RTS.STK is non-BTB-able)

8.1.1.5 Loads and stores of 512 bits

Loads and Store instructions of 512 bits have special rules associated with them because they use resources from both LSUs, unlike other memory accesses.

- 512-bit loads include: LD.16L, LD.8X, LD2.32F
- 512-bit stores include: ST.16L, ST.8X, ST2.SRS.32F

8.1.1.6 Special access LD/ST instructions

A subset of memory load and store instructions is defined which can support special memory accesses such as load-reserve, store conditional, decoration etc. The list below includes both loads and stores, and is referred to by the rules that deal with these instructions (mainly Rule A.9). If some special access is limited only to loads or stores (for example load-reserve), the appropriate subset of the list should be used.

Mnemonic base of instructions that belong to this group:

- LD.[*n*]<width>
- LD.U.[*n*]<width>
- ST.[*n*]<width>

where [*n*] is the number of transferred variables (omitted if only one variable is transferred), and <width> is the variable width such as B, BF, W, F, X etc. So for example the following base instructions are supported: LD.BF, LD.U.2X, ST.8L.

Examples of instruction categories that are not in this group:

- Accesses of 512-bit wide, see Section 8.1.1.5, “Loads and stores of 512 bits”
- Packed accesses: LD2, ST2
- Accesses with scaling: ST*.SRS

8.1.1.7 Cache instructions

This instruction group relates to all instructions that are used to control the cache operation, either directly or through the CME. They are listed in Table 4-8, in Section 4.10, “Cache control instructions.”

Following are subsets of the cache instructions:

- CME control instructions: DQUERY.*, CCMD*
- Granular cache instructions: DFETCH.*, DCM.*
- Block cache instructions: DUNLOCKB.*, PUNLOCKB.*, DFETCHB.*, PFETCHB.*, DCMB.*, PCMB.*

The same instructions (with the exception of CCMD*) could be divided also as:

- Data cache instructions: starting with D
- Program cache instructions: starting with P

8.1.1.8 Memory barrier instructions

The memory barrier instructions are LSU instructions that access memory without loading or storing data, in order to activate a memory hold that is required for keeping memory ordering semantics. These are: DBAR*, and DQSYNC

8.1.1.9 Positioning of VLES in a hardware loop

The following conventions apply for specifying VLES of a hardware loop:

- LA: The last iterated VLES of the loop (right before the LOOPENDn directive)
- SA: The first iterated VLES of the loop (right after the LOOPSTARTn directive)

Based on this convention, SA-1 means the VLES preceding the VLES of SA, and similarly for SA+1, LA-2 etc.

8.2 Summary of SC3900FP programming rules

The SC3900FP has many changes relative to previous architectures, hence many of the legacy programming rules were removed or relaxed. Table 8-1 lists all the remaining SC3900FP programming rules, including the few new ones, and specifies their status.

Table 8-1. Summary of SC3900FP programming rules

Rule	Rule Header	Rule Type	Legacy Status	Comments	Reference
A.3	Limitation on instructions at the end of a code section	Static	Modified	—	page 8-6
A.9	Grouping limitations for special memory accesses and cache instructions	Static	Modified	Modified for the new semaphore instructions in SC3900FP, and expanded to include limitations for new SC3900FP special instructions	page 8-7
A.11	Limitation on the number of R/D selection groups in a VLES	Static	New	Backwards compatible, the limitation is relevant only to new instructions and features	page 8-11
D.3	No instruction can be grouped with RTE instructions	Static	Modified	In SC3850 some instructions were allowed to be grouped	page 8-14

Table 8-1. Summary of SC3900FP programming rules (continued)

Rule	Rule Header	Rule Type	Legacy Status	Comments	Reference
D.5	No instructions that affect SR grouped with a subroutine call	Static	Modified	Expanded: in SC3850 only SR writes were restricted, now any SR change (including status)	page 8-15
G.G.2	Instruction word & encoding format limits	Static	Modified	Updated to SC3900FP fetch width total dispatch length to execution units, and encoding format grouping limitations	page 8-16
G.G.3	Execution unit capacity and allocation	Static	Modified	Updated to SC3900FP dispatcher limits, plus a list of instructions that are unique to specific LSUs or DMUs, or consume more than one unit. Name change to better reflect the content	page 8-17
G.G.4	Multiple writes to the same register or status bit	Static	Modified	Now relates only to instructions with the same predication	page 8-19
G.G.4a	Multiple writes to the same register or status bit predicated by different predicates (static)	Static	New	Relates to instructions predicated with different predicates which are not allowed regardless of the predicate value	page 8-22
G.G.4b	Multiple writes to the same register or status bit predicated by different predicates (dynamic)	Dynamic	New	Relates to cases where mutually exclusive predication is allowed	page 8-23
G.P.3	Mutually exclusive instructions in a VLES	Static	Modified	List modified	page 8-24
G.S.1	Instruction-specific rules	Static	New	These limitations will be described in the ISA page of each relevant instruction	page 8-25
J.1	COF destination must be to a start of a VLES	Guideline	Maintained		page 8-28
J.4	RAS value equal to value pointed to by SP	Dynamic	Modified	Instruction list modified	page 8-29
J.6	Limitations of updates of special control registers	Static	Modified	Changed from a guideline to a static rule, updated register list, requiring SYNCP in parallel	page 8-30
L.C.7	Loop COF within a hardware loop cannot have destination in the same loop	Static	Maintained	—	page 8-31
L.D.8	Read from LCn at LA of a hardware loop	Static	Modified	Restricted only at LA	page 8-32
L.L.1	Limitations on contents of LA of hardware loops	Static	Modified	Restriction now relates only to LA	page 8-33
L.L.2	No DOEN.n or write to LCn at in the same hardware loop	Static	Modified	Expanded to all the loop	page 8-34
L.L.6	Limitations on instructions at (SA-1) of a hardware loop	Static	New	Not backwards compatible - the mapper should insert NOP if relevant in legacy code	page 8-35

Table 8-1. Summary of SC3900FP programming rules (continued)

Rule	Rule Header	Rule Type	Legacy Status	Comments	Reference
L.N.1	Nested hardware loops cannot have the same LA	Static	Maintained	—	page 8-36
L.N.2	Hardware loop bodies and nesting	Static	Modified	Allow any nesting order	page 8-37
L.N.7	Rules for sequential loops	Static	Modified	Will now describe sequential loops	page 8-38
S.1	rules for assembly sections and assembly file content	Static	New	Rules for the assembler regarding section alignment and content consistency	page 8-39

8.3 Programming rule reference

This section describes each rule. The description will be expended in future versions of the document.

Rule A.3 Limitation on instructions at the end of a code section

Description

The last COF in a code section must be one of the following instructions: JMP*, BRA*, RTE* or RTS*. This last COF instruction should not be predicated.

A code section in this context is a consecutive program area, for which memory in the following addresses is not guaranteed to be valid machine code after linking. The end of a code section can be the end of the assembler file, the last line of the code area that is followed by the data array, and so on.

NOTE

Instructions located between the VLES following the last COF of a code section and the last VLES in the code section should not be targets of COF instructions, however must be valid SC3900FP code even though they can never be executed.

Rule A.9 Grouping Limitations for Special Memory Accesses and Cache Instructions

Description

This rule lists limitations when using special memory accesses and cache instructions. These instructions are used for specific, non-standard purposes, and are not relevant for general load and store instructions. Table 8-2 also has references to CRM sections where the functionality of these instructions is described. The table also refers to instruction groups that are relevant to definition of the rule, see Section 8.1.1, "Referencing instruction groups" for a definition of the following instruction groups:

- Cache instructions
- Memory barrier instructions
- COF instructions
- Special access LD or ST instructions

Table 8-2. Grouping limitations for special memory access and cache instructions

#	Instruction	Reference	Must be grouped with (per instruction, listed alternate options)...	Cannot be grouped with...	Additional limitations/ comments
1	LDRSTC	Semaphore accesses, see Section 4.12, "Semaphore and atomic operations support."	special access LD + SYNC.B	A second memory access A cache instruction or barrier A BTB-able COF inst., LA of a hardware loop SYNC.B.DL	The parallel memory access must be aligned LDRSTC must immediately precede the LD or ST it affects Both LDRSTC and the affected LD/ST cannot be predicated The following VLES cannot have a memory access, cache instruction or barrier
2			special access ST		
3	DECOR	Decorated accesses, see section Section 4.14.1, "Access decoration."	special access LD + SYNC.B.DL	A second memory access A cache instruction or barrier A COF inst., LA of a hardware loop	The parallel memory access must be aligned DECOR must immediately precede the LD, ST or NOTIFY it affects Both DECOR and the affected LD/ST/NOTIFY cannot be predicated The following VLES cannot have a memory access, cache instruction or barrier
4			special access ST		
5			NOTIFY		

Table 8-2. Grouping limitations for special memory access and cache instructions (continued)

#	Instruction	Reference	Must be grouped with (per instruction, listed alternate options)...	Cannot be grouped with...	Additional limitations/ comments
6	ACATOR* cases:	Access Attribute Override, see Section 4.11, “Instruction attribute override”			ACATOR* or the tag (STRM etc.) must immediately precede the memory access it affects. Must have the same predication as the instruction it affects
	STRM		One of: LD* LD* + non-stream DFETCH LD* + non-stream ST* ST* ST* + non-stream DFETCH	- LD*/ST* combination not explicitly in the options specified to the left - AID - Barrier, cache instruction (except for DEFETCH) - PUSHC, POPC - SYNC.B.DL	
	ITW		One of: LD* DFETCH, DFETCHB	A second memory access, cache instruction or barrier PUSHC, POPC SYNC.B.DL	
	AID		One of: LD*, ST*, DFETCH, DFETCHB*, [DP]UNLOCKB.L2 DCM.* DQUERY.*		
7	NOTIFY	Notify accesses, see Section 4.14.2, “Notify accesses”	DECOR	Dictated by DECOR: A second memory access A cache instruction or barrier A COF inst., LA of a hardware loop	Dictated by DECOR: The address must be aligned cannot be predicated The following VLES cannot have a memory access, cache instruction or barrier
	NOTIFYM meta instruction				
8	MSGSND	Message accesses, see Section 4.14.3, “Message accesses”	—	A cache instruction or barrier A 512-bit load or store, see 8.1.1.5	
9	SYNC.B.DL	Destructive loads, see Section 4.14.4, “Destructive loads”	LD (except for 512-bit loads) DECOR + LD (except for 512-bit loads)	A second memory access A cache instruction or barrier A COF inst., LA of a hardware loop	The parallel memory access must be aligned The following VLES cannot have a memory access, cache instruction or barrier The addr. must be labeled “Guarded” in MMU

Table 8-2. Grouping limitations for special memory access and cache instructions (continued)

#	Instruction	Reference	Must be grouped with (per instruction, listed alternate options)...	Cannot be grouped with...	Additional limitations/ comments
10	DQUERY.*	Cache query instructions, see Section 4.10.5, “Attribute override of cache instructions”	—	A memory access A cache instruction or barrier A COF inst., LA of a hardware loop	Cannot be predicated The following VLES cannot have a memory access, cache instruction or barrier
11	Block cache instructions	See Section 4.10.3, “Block cache commands”	BCCAS + SYNC.B	—	The block cache instruction, BCCAS and the SYNC.B cannot be predicated The following VLES cannot have a memory access, cache instruction or barrier
	BCCAS		A block cache instruction	—	—
	Block cache meta instructions		—	—	Cannot be predicated The following VLES cannot have a memory access, cache instruction or barrier
12	CCMD.*	Cache command instructions, see Section 4.10.6, “Cache command summary”	SYNC.B	A memory access A cache instruction or barrier	Cannot be predicated The following VLES cannot have a memory access, cache instruction or barrier
	CCMDM* (meta instruction)		—		
13	DFETCH.*	See Section 4.10.6, “Cache command summary”	—	A cache instruction or barrier A 512-bit load or store, see 8.1.1.5	Cannot be predicated
14	DCM.*	See Section 4.10.6, “Cache command summary”	—	A memory access A cache instruction or barrier A COF inst., LA of a hardware loop	Cannot be predicated The following VLES cannot have a memory access, cache instruction or barrier
16	Data load mem. barriers: DBAR.* and DQSYNC	See Section 4.13, “Memory barriers”	SYNC.B	A memory access A cache instruction Another barrier	Cannot be predicated The following VLES cannot have a memory access, cache instruction or barrier
	DBARM.* meta instruction		—		

Table 8-2. Grouping limitations for special memory access and cache instructions (continued)

#	Instruction	Reference	Must be grouped with (per instruction, listed alternate options)...	Cannot be grouped with...	Additional limitations/ comments
17	Data store memory barrier: DBARS.*	See Section 4.13, “Memory barriers”	—	A memory access A cache instruction Another barrier A COF inst., LA of a hardware loop	Cannot be predicated The following VLES cannot have a memory access, cache instruction or barrier
19	PUSHC*	Accessing the alternate stack, see Section 4.9.1, “Stack pointer selection”	—	Any memory access other than PUSHC	When grouped with another PUSHC, all GG4 grouping restrictions for PUSHC apply See limitation on using regular SP in Rule G.P.3
20	POPC*	Accessing the alternate stack, see Section 4.9.1, “Stack pointer selection”	—	Any memory access other than POPC	When grouped with another POPC, all GG4 grouping restrictions for POPC apply See limitation on using regular SP in Rule G.P.3

Rule A.11 Limitation on the number of R/D/CTL selection groups in memory accesses & MOVE instructions in a VLES

Definition

Definition: a “selection group” is a data path resource in the AGU which is used when transferring registers to or from memory, and between registers in MOVE instructions. In most cases, an operand group for which the operands are separated by “:” or “::” is considered a single selection group, while an operand group for which the operands are separated by “,” belong to different selection groups. The “:” syntax implies that the operands in this group, for example d0:d1:d2:d3 cannot be independently selected, and usually must belong only to a consecutive set. However, some exceptions to this lexical convention apply, see Table 8-3 below. For more detail on this definition, see Section 4.6.7, “Register bandwidth limitations on MOVE and memory accesses.” Table 8-3 defines the number of selection groups for the relevant instructions.

Table 8-3. Selection groups

Instruction	DALU (D) selection groups		AGU (R/CTL) selection groups		Comments
	Source	Dest.	Source	Dest.	
Memory load to D register(s) of up to 256 bits		1			Including POP*
Memory load of 512 bits		2			—
Memory store from D register(s) of up to 256 bits	1				Including PUSH*
Memory store of 512 bits	2				—
Memory load to R or Ra:Rb				1	Including POP*, block cache instructions (which update R as return status) and DQUERY, PQUERY which update Ra:Rb
Memory load to R register quad Ra:Rb:Rc:Rd				2	Including POP*
Memory load to a single control register		1		1	Including POP* Note it uses also a DALU group
128-bit memory load to control registers: POP*.4L to control registers		1		2	Note it uses also a DALU group
RTS*				1	—
Memory store from R or Ra:Rb, or from a single control register Ca			1		Including PUSH* and JSR*, BSR (which push the PC:SR)
128-bit memory store from R or control registers: ST*/PUSH* from Ra:Rb:Rc:Rd or PUSH*.4L from control registers			2		Including PUSH*
MOVE.[LX] Da,Dn MOVE.2[LX] Da:Db,Dm:Dn	1	1			—

Table 8-3. Selection groups (continued)

Instruction	DALU (D) selection groups		AGU (R/CTL) selection groups		Comments
	Source	Dest.	Source	Dest.	
MOVE.L Ra,Dn		1	1		—
MOVE.L Da,Rn	1			1	—
MOVE.2L Da:Db,Ra,Rb	2			2	Exception case - DALU 2 source groups
MOVE.2L Da:Db,Ra:Rb	2			1	
MOVE.4X Da::Dd,Dm:Dq	1	1			Da::Dd is a quad permutation
MOVE.[WL] #imm,Dn		1	1		Note it uses also an AGU source sel. group
MOVE.PAR.W #imm,Dn.[HL]	1	1	1		Implemented as read-modify-write
LD.PAR.W <addr. mode>,Dn.[HL]	1	1			Implemented as read-modify-write
LDSH2.*	1	1			Implemented as read-modify-write
MOVE.L Da,TMDAT	1			1	—

Description

Independently for R/CTL and D registers, and for reads and writes, a VLES could support up to two selection groups of each type. In a VLES, four counts should be checked (DALU source, DALU destination, AGU source, AGU destination), and the appropriate values for each instruction listed in Table 8-3 should be added. For the VLES as a whole, each count value cannot exceed 2.

The rule is applied regardless of the predication state of the relevant instructions.

The operands that are relevant for this rule are only registers that are used as sources or destinations of data written to/read from memory or with register MOVE instructions. Control registers (all registers that are not R, D or SP) are considered in AGU selection groups for this rule. Registers used as pointers or operands in arithmetic instructions are not subject to this rule.

NOTE

This is a new rule for SC3900FP. It is backwards compatible with SC3850 code, as the limitation is relevant only to new instructions and execution capacity.

Examples:

```
ld.l (r0),r1      move.2l d0:d1,r2,r3        ; not allowed: 3 R dest. selection groups
st.4l d0:d1:d2:d3,(r0) move.2l d0:d1,r2,r3   ; not allowed - 3 D src. selection groups
move.l d0,d1      st.l d2,(r5)      st.b d3,(r6)
                                             ; not allowed - 3 DALU source selection
                                             ; groups
ld.16l (r0),d0::d15    ld.l (r1),d20; not allowed - ld.16l uses 2 dest.
                                    ; selection groups
st.4l r0:r1:r2:r3,(r0) move.l r7,d8          ; not allowed, 3 R source selection groups
st.4l r0:r1:r2:r3,(r19) ld.4l(r20+r16),r4:r5:r6:r7   ; allowed
ld.4l (r10),r0:r1:r2:r3 ld.4l (r11),d0:d1:d2:d3      ; allowed
pop.4l r0:r1:r2:r3         move.l d8,r7              ; not allowed
pop.4l gcr:mctl:btr0:btr1  tfra mctl,r5              ; allowed - TFRA not in the rule
move.2l d0:d1,r0,r1        push.l d5        ; not allowed - 3 DALU source sel. groups
ld2.32f (r0),d0::d15       move.l r0,d20    ; not allowed - 3 DALU dest. sel. groups
ld.16l (r0),d0::d15        st.16l d0::d15,(r20)      ; allowed
```

Rule D.3 No instructions can be grouped with RTE instructions

Description

RTE* instructions (all variants of RTE) must be alone in the VLES and cannot be grouped with other instructions.

Rule D.5 No instructions that affect SR can be grouped with a subroutine call

Description

Any instruction that affects SR (whole register or individual bits, explicitly or implicitly) cannot be grouped in a VLES with a BSR or JSR* instruction.

Example

This line of code illustrates an invalid grouping of a write to the SR with a subroutine call:

```
tfra r5,sr        jsr r0           ; Not allowed - SR write
cmp.eq.x d0,d1,p0 bsr _dest        ; Not allowed - cmp.eq updates p0
rol.x d0,d1       jsr _dest        ; Not allowed - rol.x updates the carry bit
```

Rule G.G.2 Instruction word length and encoding format grouping limitations

Description

A VLES can be at most 16 program words long (256 bits), including suffix words.

NOTE

> When a VLES is encoded some instruction words may be added that are not explicit in the assembly code, such as a suffix, LPST* and LPEND* (at SA-1 and LA, respectively) and in very rare occasions NOP.AGU or NOP.DALU instructions required for instruction padding

In addition, there are limits to the allowed grouping of DALU instructions as follows:

- The combined length of all DALU instructions in a VLES cannot exceed 8 program words (128 bits)

  NOTE

  > Limitations on execution units do not include suffix instructions, even if they affect instructions in the execution units (for example adding predication information).

- Only the following combinations of encoding formats are allowed for DALU instructions in a VLES:
  - One 32-bit instruction
  - One 64-bit encoding format
  - One 64-bit encoding format and one 32-bit instruction (in this order)
  - Two 64-bit encoding formats

  A single 64-bit DALU encoding format can encode either one instruction or two instructions. Also, each 32-bit instruction has a variant that is included in a 64-bit encoding format. Hence this limitation does not affect the functional ability to group up to 4 DALU instructions.

Rule G.G.3 Execution Unit Capacity and Allocation

Description

A VLES cannot attempt to use more than the available number of execution units. The maximum allowed number of instructions in a VLES is:

- One PCU instruction
- Two LSU instructions (one using LSU0, and one LSU1)
- One IPU instruction
- Four DALU instructions

The list above reflects the order by which the instructions should be encoded in the VLES.

NOTE

This rule does not apply to prefix or suffix instructions because they are not dispatched to execution units.

Some instructions that use the LSU are rigidly assigned to operate on either LSU0 or LSU1. Normally this requirement is transparent to the user as the assembler reorders the instructions in the encoded VLES to conform with the rigid assignments. However, two instructions that rigidly use the same LSU cannot be grouped together. Table 8-4 lists the instructions that can operate only on a specific LSU. There is no specific relations between instructions in the same row.

Table 8-4. Instructions executing on a specific LSU

Instructions executing only on LSU0	Instructions executing only on LSU1
LD* of 512 bits	ST* of 512 bits[1]
Instructions that activate the CME (see Section 8.1.1.7, “Cache instructions”): • Block cache instructions • CME control instructions • Program cache instructions • PQUERY* and DQUERY* • Program memory barriers: PBAR*	BCCAS
LD/ST that is grouped with LDRSTC	A PCU instruction that consumes an LSU slot, see Rule G.G.3.

[1] If there is no other LSU instruction grouped with it, the assembler adds a NOPA instruction in the encoding

The following DALU instructions execute by specific DMUs, and are not re-ordered by the assembler:

- FFT.R5.P11.4T is supported only by DMU0 and DMU2 - must be the 1st or 3rd DALU instruction
- FFT.R5.P12.4T is supported only by DMU1 and DMU3 - must be the 2nd or 4th DALU instruction
- FFT.R5.IP11.4T is supported only by DMU0 and DMU2 - must be the 1st or 3rd DALU instruction
- FFT.R5.IP12.4T is supported only by DMU1 and DMU3 - must be the 2nd or 4th DALU instruction

Some instructions consume more than one execution unit, which should be considered in the total execution unit count for that VLES.

Instructions that consume both a PCU and an LSU slot:

- JMP Rn
- JSR, BSR
- RTE, RTE.CIC
- RTS, RTS.STK
- CONT.*n*

64-bit DALU instructions consume 2 DALU execution slots, regardless if they actually activate one or two DMUs.

Rule G.G.4 Multiple writes to the same register or status bit

Description

Instructions that write to the same register or affect the same status bit cannot be grouped in the same VLES.

Cases covered by this rule are considered violations also in cases some or both of the instructions are predicated as follows:

- Both write instructions are not predicated
- Both write instructions are predicated with the same predicate and condition
- One instruction is not predicated and the other one is

Instructions predicated with *different* predicates writing to the same register or bit field are covered by different rules: Rule G.G.4a and Rule G.G.4b.

Cases

This table lists the less obvious cases for this rule:

Table 8-5. Case rules

Case	Example	Page
Duplicate address pointer register destinations—A VLES cannot group multiple writes by instructions to the same address pointer register R*n*. The no-update addressing modes (R*n*), (R*n*+R*k*) etc. are not considered an address register write.		
Duplicate stack pointer destinations—A VLES cannot group multiple writes or updates to the active stack pointer. This case applies regardless of the physical stack pointer being used. Some exceptions apply, see Table 8-6.		
Duplicate R/D register destinations—A VLES cannot group multiple instructions that write different portions of the same register. Exceptions apply to some cases of predication and mutually exclusive writes, see Table 8-6 and Rule G.G.4a.		
Duplicate control register destinations—A VLES cannot group instructions that write to a control register (SR, GCR etc.) as a whole or a half of it, with similar instructions or instructions that affect individual bit fields in the same register.		
Duplicate status bit destinations—A VLES cannot group multiple instructions that affect the same status bit (for example P0-P5, C, bits in the SR). Some exceptions apply, see Table 8-6.		
The carry bit (SR.C) can be written by one instruction per VLES		

Exceptions

This table lists exceptions for rule G.G.4. In general these exceptions apply also only to instructions that have the same predication status (both without a predicated or predicated with the same predicate). However, some exceptions allow to disregard the predication status altogether, in which case it is specified in the table per case:

Table 8-6. Rule G.G.4 exceptions

Exception	Examples (page)
Dual even/odd push/pop instructions—It is allowed to group two PUSH.L instructions, or two POP.L instructions if they access different, OP*e* (even) and OP*o* (odd), registers (see Table 4-4). The core ensures that the SP is written to correctly. The exception also applies to the respective PUSHC.L and POPC.L instructions.	—
Dual 4-element push/pop instructions—It is allowed to group two 4-element PUSH instructions (4L or 4X) or two 4-element POP instructions in the same VLES. The core ensures that the SP is written to correctly. Note that some register bandwidth limitations apply for R and CTL register accesses, see Rule A.11. The exception also applies to the respective PUSHC and POPC instructions.	—
A 2-element push/pop can be grouped with a single or dual element push/pop instruction—It is allowed to group a PUSH.2L instruction with another PUSH.2L instruction or PUSH.L instruction; Likewise it is allowed to group a POP.2L instruction with another POP.2L instruction or a POP.L instruction. The exception also applies to the respective PUSHC and POPC instructions.	—
Writes to different status/control bits and fields in SR and GCR—It is allowed to group instructions that affect different status/control bits or fields in SR and GCR. These include P0-5, and S status bits in the SR; RM, SM, SM2, W20 and SCM in SR (if written by the CHCF instruction); and AS*n*, BAM and floating point status bits in GCR. This exception relates to instructions that write to field/bit specific destinations, not as part of register-level or half-register destination (such as TFRA, BMSETA, POP.L). The instructions writing to the different bits or fields may have unrelated predication status. Note: only one carry (SR.C) destination is allowed per VLES, regardless of predication.	—
Multiple updates of a predicate —It is allowed under certain conditions, see detailed explanation with examples in "Multiple updates of a predicate" below	—
Multiple S, SAT and floating point status bit destination—It is allowed to group multiple instructions that affect SR.S, and SR.SAT and floating point flags in GCR. These status bits are sticky and are set by the logical OR of the respective conditions from all executed instructions in a VLES that affect them. The instructions writing to these status bits may have unrelated predication status.	—
Write to SR2 by instructions that cause an exception— Instructions that update SR2 as part of the process of generating an exception may be grouped with any other instruction that affects SR2. Instructions that cause exceptions include, among others: TRAP.* and illegal instructions. Remaining as cases restricted by G.G.4 are instructions that update SR2 as a whole, which cannot be grouped with other instructions of this type. This group includes BMSETA, BMCLRA, BMCHGA, POP*, TFRA, CHCF where the destination operand is SR2.	—
Parallel ACS.H.2W or ACS.L.2W instructions— Several ACS.H.2W or ACS.L.2W instructions can update the BTR registers in parallel. The ACS.H.2W or ACS.L.2W instructions must be used only in groups of two or four instructions in a VLES. For this and additional limitations concerning ACS* instructions, see Rule G.S.1	—

Multiple updates of a predicate

Two predicate generating instructions that update the same destination predicate cannot be grouped together in a VLES if any of the following conditions occur:

- One of them is a special predicate updating instruction (considered updating all predicates): PINITA, PRDA.SWP, PRDA.RST.

- One of them is PCALCA or PCALC, even if updating a different predicate
- One of them updates one predicate and one of them updates two predicates
- The two instructions update two predicates which are not the same pair, for example p0:p1 and p1:p2
- One instruction is a DALU instruction and the other is an AGU (LSU or IPU) instruction
- The two instructions are predicated by different predicates
- One of the instructions is predicated and one is not
- One of the instructions is predicated as IF.P*n* and the other as IF.P*n*.EC

If none of these conditions apply, several predicate generating instructions that update the same predicate can be grouped together, and then the destination predicate holds the logical OR of the values generated by each individual instruction.

Examples

These lines of code illustrate invalid grouping cases of multiple predicate updates:

```
         pinita #2,#3      cmpa.eq r0,r1,p4  ; Not allowed - special multiple predicate update
         bmseta #3,sr.l    cmpa.gt r1,r2,p1  ; Not allowed - whole SR write (see Table 8-5)
         cmp.eq.x d0,d1,p0 cmp.gt.x d2,d3,p0:p1; Allowedcmp.eq.x d0,d1,p0:p1 cmp.gt.x d2,d3,p1
; Not allowed - mix of 1 and 2 predicate update
         cmp.eq.x d0,d1,p0:p1 cmp.gt.x d2,d3,p1:p2; Not allowed - not the same predicate pair
         cmp.eq.x d0,d1,p0 cmpa.eq r0,r1,p0  ; Not allowed - update by DALU and AGU
         if.p0 cmp.eq.x d0,d1,p2    if.p1 cmp.gt.x d2,d3,p2   ; Not allowed - predicated by
                                                              ; different predicates
         if.p0 cmp.eq.x d0,d1,p2    cmp.gt.x d2,d3,p2         ; Not allowed - one predicated
                                                              ; and one is not
         if.p0 cmp.eq.x d0,d1,p2    if.p0.ec cmp.eq.x d2,d3,p2; Not allowed - mixing IF.Pn and
                                                              ; IF,Pn.EC predication
```

These lines of code illustrate some valid grouping cases of multiple predicate updates:

```
         cmp.eq.x d0,d1,p0 cmp.ne.x d2,d3,p0 tstbm.s.l #$16,d5,p0      ; Allowed - all DALU
         cmpa.eq r0,r1,p2  cmpa.gt r2,r3,p2  bmtsta.c #$12,r4,p2; Allowed - all AGU
         if.p0 cmp.eq.x d0,d1,p3    if.p0 cmp.gt.x d2,d3,p3    ; Allowed - same predication
         if.p0.ec cmp.eq.x d0,d1,p3 if.p0.ec cmp.gt.x d2,d3,p3 ; Allowed - same predication
         cmp.eq.x d0,d1,p0:p1 cmp.gt.x d2,d3,p0:p1; Allowed - updating the same pair
         if.p5.ec cmp.eq.x d0,d1,p0:p1 if.p5.ec cmp.gt.x d2,d3,p0:p1   ; Allowed - same
                                                                       ; predication (TBD)
```

Rule G.G.4a Multiple writes to the same register or status bit, predicated by different predicates (static)

Description

This rule restricts some cases of instructions predicated by *different* predicates that write to the same register or status bit, that are statically detectable (not related to the value of the predicates). It supplements Rule G.G.4.

In any the following cases, it is not allowed to group together in the same VLES instructions writing to the same register or status bit, which are predicated by different predicates:

- Instructions that write to a full SP or CTL register. In this context, a write to H or L portion of a register is considered a full register write
- Instructions that write to the carry bit (SR.C)

Examples

```
cmp.eq.x d0,d1,p0:p1
if.p0 tfra r0,lc0          if.p1 tfra r1,lc0         ; Not allowed, even though
                                                     ; p0 and p1 cannot both be true
if.p0 add.x d0,d1,d2       if.p1 sub.x d2,d3,d2      ; Allowed (mutual exclusiveness
                                                     ; covered by Rule G.G.4b
```

Rule G.G.4b Multiple writes to the same register or status bit, predicated by different predicates (dynamic)

Description

Two instructions predicated by two *different* predicates, and that write to the same register, and are not restricted by Rule G.G.4a, cannot be grouped together in the same VLES according to this rule if at least one of the following conditions apply:

- The destination is potentially updated by more than two instructions in that VLES
- One of the instructions is an AGU instruction, and the other is a DALU instruction
- The predicates of the two instructions are both resolved to be true

It is the users' responsibility to ensure that predicating conditions are mutually exclusive.

Rule G.G.4b is dynamic and cannot be checked beforehand by the assembler. The most common way to ensure that two predicates are mutually exclusive is to generate them together by a predicate-generating instruction that updates two predicates with inverse values.

Examples

```
cmp.eq.x d0,d1,p0:p1
if.p0 adda.lin r0,r1,r2    if.p1 suba.lin r4,r0,r2  ; Allowed, p0 and p1 cannot both
                                                    ; be true
if.p0 add.x d0,d1,d2       if.p1 sub.x d2,d3,d2     ; Allowed if mutually exclusive

[ if.p0 adda r0,r1,r2      if.p1 suba r4,r0,r2
 if.p2 ora r5,r6,r2 ]                               ; Not allowed - more than 2 inst.
                                                    ; potentially writing to r2

if.p0 ld.l (r0),d2         if.p1 add.x d0,d1,d2     ; not allowed - AGU and DALU
```

Rule G.P.3 Mutually exclusive instructions in a VLES

Description

The following sets of instructions are mutually exclusive and cannot be grouped in the same VLES, regardless of predication.

For each item below, which includes an instruction or list of instructions, grouping of any two of the instructions, as well as the instruction with itself is not allowed:

a) Only one instruction that updates a link register (LR0, LR1, LR2) or a portion of it is allowed per VLES

For each item below, which includes two instruction lists, delimited with parenthesis: grouping of any instruction from one list with an instruction from the other list is not allowed:

1. (Any write to LC*n*) and (DOEN.*, any PCU instruction)
2. (An instruction that stores 512 bits) and (any instruction performing a memory store, RTS*)
3. (An instruction that loads 512 bits) and (any instruction performing a memory load, JSR*, BSR)
4. (JSR*, BSR, RTS*), and right of it in the VLES a memory store. Note: a store to the left of the mentioned PCU instruction is allowed.
5. (RTS*, POP*.L GCR) and MOVE.L Dn,TMDAT
6. (PUSH*.L TMTAG, PUSH*.4L PROCID:TID:TMTAG) and any ST* of BTR0
7. (An instruction using a stack pointer selected by ASPSEL: PUSHC.*, POPC.*, TFRCA) and (an instruction using a stack pointer selected by SPSEL: PUSH.*, POP.*, SP-relative LD/ST, TFRA using SP, ADDA using SP)

Rule G.S.1 Single instruction rules

This rule is used to describe limitations for single instructions, which is not related to other instructions that may be grouped with it or in adjacent VLES.

Description

Instructions that must be alone in a VLES (cannot be grouped with any other instruction):

1. TRAP.0, TRAP.1
2. RTE* (see Rule D.3)
3. DEBUGM.*n*
4. DEBUGE.*n*

Instructions that cannot be predicated:

1. RTE*
2. SWB
3. LPST*, LPEND*, LPSKIP*
4. All cache instructions, see Rule A.9
5. Memory barriers, see Rule A.9
6. Some special memory access instructions, see Rule A.9
7. ILLEGAL
8. SYNC.D, SYNC.B, SYNC.B.DL
9. CLRIC
10. JMPRF
11. NOP

Rules related to FFT instructions:

1. FFT.R4.S1.8T Da,Dc:Dd,Db,Dg:Dh,Di,Dx:Dy,Dz:Dw
   FFT.R4.IS1.8T Da,Dc:Dd,Db,Dg:Dh,Di,Dx:Dy,Dz:Dw
   (Da,Dc,Db,Dg,Dx,Dz are encoded in 64-bit instruction)
   When encoded for DMU0 and DMU1 Dx and Dz must be even
   When encoded for DMU2 and DMU3 Dx and Dz must be odd
2. FFT.R4.SN.8T Da,Dc:Dd,De:Df,Db,Dg:Dh,Di,Dx:Dy,Dz:Dw
   FFT.R4.SN.8T Da,Dc:Dd,De:Df,Db,Dg:Dh,Di,Dx:Dy,Dm:Dn
   FFT.R4.ISN.8T Da,Dc:Dd,De:Df,Db,Dg:Dh,Di,Dx:Dy,Dz:Dw
   FFT.R4.ISN.8T Da,Dc:Dd,De:Df,Db,Dg:Dh,Di,Dx:Dy,Dm:Dn
   FFT.R4B.SN.8T Da,Dc:Dd,De:Df,Db,Dg:Dh,Di:Dj,Dx:Dy,Dz:Dw
   FFT.R4B.SN.8T Da,Dc:Dd,De:Df,Db,Dg:Dh,Di:Dj,Dx:Dy,Dm:Dn
   FFT.R4B.ISN.8T Da,Dc:Dd,De:Df,Db,Dg:Dh,Di:Dj,Dx:Dy,Dz:Dw
   FFT.R4B.ISN.8T Da,Dc:Dd,De:Df,Db,Dg:Dh,Di:Dj,Dx:Dy,Dm:Dn
   (Da,Dc,De,Db,Dg,Di,Dj,Dx,Dz or Dm are encoded in 64-bit instruction)
   When encoded for DMU0 and DMU1 Dx and Dz or Dm must be even
   When encoded for DMU2 and DMU3 Dx and Dz or Dm must be odd
   For the FFT.4B.* instructions, Di must be encoded as De+3 (the hardware computes De value from Di-3 in the second execution unit)
3. FFT.R3.SP1.4T Da:Db,Dc:Dd,Dj:Dk
   FFT.R3.ISP1.4T Da:Db,Dc:Dd,Dj:Dk
   (Da,Dc,Dj are encoded in 32-bit instructions)
   When encoded for DMU0 or DMU2 Dj must be odd
   When encoded for DMU1 or DMU3 Dj must be even
4. FFT.R3.P2.8T Da:Db,Dc,Dd,De:Df,Dg,Dh,Dj:Dk,Dw:Dx
   FFT.R3.IP2.8T Da:Db,Dc,Dd,De:Df,Dg,Dh,Dj:Dk,Dw:Dx
   (Da,Dc,Dd,De,Dg,Dh,Dj,Dw are encoded in 64-bit instruction)
   If Da is even De must be odd and if Da is odd De must be even
5. FFT.R5.P11.4T Da:Db,Dc:Dd,Dm:Dn
   FFT.R5.P12.4T Da:Db,Dc:Dd,Dm:Dn
   (Da,Dc,Dm are encoded in 32-bit instruction)
   FFT.R5.P11.4T and FFT.R5.P12.4T must appear together in pairs, only in the following options, no assembly re-ordering is performed:
   FFT.R5.P11.4T in DMU0 and FFT.R5.P12.4T in DMU1
   FFT.R5.P11.4T in DMU2 and FFT.R5.P12.4T in DMU3
   FFT.R5.P11.4T in DMU0 + DMU2, and FFT.R5.P12.4T in DMU1 + DMU3
6. FFT.R5.IP11.4T Da:Db,Dc:Dd,Dm:Dn
   FFT.R5.IP12.4T Da:Db,Dc:Dd,Dm:Dn
   (Da,Dc,Dm are encoded in 32-bit instruction)
   FFT.R5.IP11.4T and FFT.R5.IP12.4T must appear together in pairs, only in the following options, no assembly re-ordering is performed:
   FFT.R5.IP11.4T in DMU0 and FFT.R5.IP12.4T in DMU1
   FFT.R5.IP11.4T in DMU2 and FFT.R5.IP12.4T in DMU3
   FFT.R5.IP11.4T in DMU0 + DMU2, and FFT.R5.IP12.4T in DMU1 + DMU3

7. FFT.R5.P2.8T Da:Db,Dc:Dd,De,Dg:Dh,Di:Dj,Dk,Du:Dv,Dw:Dx
   FFT.R5.IP2.8T Da:Db,Dc:Dd,De,Dg:Dh,Di:Dj,Dk,Du:Dv,Dw:Dx
   (Da,Dc,De,Dg,Di,Dk,Du,Dw are encoded in 64-bit instruction)
   If Da is even Dg must be even and if Da is odd Dg must be odd

Rules related to ACS instructions (application specific for the Viterbi kernel):

1. It is not allowed to have an odd number of ACS* instructions in a VLES (if they appear, can be only 2 or 4)
2. If only 2 ACS* instructions appear in a VLES, they must appear in the source assembly as the 1st and 2nd DALU instructions, or as the 3rd and 4th DALU instructions. The assembler does not reorder the ACS* instructions in the VLES - the order of the ACS* instructions affects the semantics of how the BTR0-1 are updated, see Section 2.5.2, “Parallel and serial semantics inside a VLES.”
3. The mentioned 2 or 4 ACS* instructions must not be predicated or if predicated, predicated by the same predicate

Rule J.1 COF destination must be the start of a VLES (guideline)

Description

A COF destination must be the start (lowest) address of a VLES. A COF destination cannot jump to the middle of a VLES.

Example

This segment of code illustrates a COF destination in the middle of a VLES:

```
  jmp _dest+2                        ; Not allowed
  ...
_dest  mac.x d0,d1,d2    mac.x d3,d4,d5
```

The assembler evaluates the address of a VLES label as the start (lowest) address of a VLES, regardless of its source position in the VLES. Good programming practice always places COF destination labels before or at the start of a VLES. Programmers should be careful to ensure that computed COF destinations are at the start of a VLES. This ensures that rule J.1 is enforced.

Rule J.4 RAS value equal to value pointed to by SP (dynamic)

Description

Upon execution of an RTS instruction, if the RAS is valid, the value of the RAS (used to restore the PC) must be equal to the value in the stack, pointed to by the stack pointer, that would have been used if the RAS were not valid.

Example

This segment of code illustrates improper manipulation of the SP, relying on the return address using RAS:

```
st.l d6,(sp-8)
rts                    ; Not allowed since RAS may be valid
```

Rule J.6 Limitations on updates of the special control registers

Description

The following instructions must be grouped with a SYNC.P or a CLRIC instruction:

- Any write to CESRA0, CESRA1

The following instructions must be grouped with a CLRIC #1 or a CLRIC #3 instruction:

- TFRA to MOCR
- BMSETA, BMCLRA, BMCHGA to MOCR.L

The following instructions must be grouped with a SYNC.D instruction:

- BMSETA, BMCLRA, BMCHGA to MOCR.H
- An explicit write to EIDR: TFRA, BMSETA, BMCLRA, BMCHGA
  Note: RTE* which implicitly writes to EIDR is not subject to this rule

Rule L.C.7 Loop COF within a hardware loop cannot have destination in the same loop

Description

A loop COF instruction (BREAK.*, CONT.*, SKIP.* or LPSKIP.*) in the body of a hardware loop *n* cannot have a COF destination in the same loop. For CONT*, the limitation refers to the destination in case of the last iteration (the other destination is to SA).

Example 1

This segment of code illustrates a BREAK instruction in a loop with its destination in the same loop, which is not allowed:

```
         doen.3 #5
         nop
         loopstart3
label1   add.x d0,d1,d1
         add.x d2,d3,d4d2
         break.3 label2         ; Not allowed
         sub.x d3,d4,d3
         sub.x d4,d5,d6
label2   sub.x d7,d8,d5
         loopend3
```

Example 2

This segment of code illustrates a SKIP.*n* instruction in a loop with its destination outside of the loop, which is allowed:

```
         doen.3 r0
         nop
         skip.3 label2        ; Allowed
         loopstart3
label1  inc d1
         add.x d0,d1,d1
         add.x d0,d1,d2
         add.x d0,d1,d3
         add.x d0,d1,d4
         loopend3
label2   nop
```

Rule L.D.8 Reading from LC*n* at LA of a hardware loop

Description

An instruction that uses LC*n* as a source (TFRA LC*n*,R) is not allowed at LA of the same hardware loop *n*.

Rule L.L.1 Limitations on contents of LA of a hardware loop

Description

PCU instructions are not allowed at LA of a hardware loop, except for:

- LPEND.*n*
- LPEND.SQ.*n*

Note: these two instructions are generated by the assembler and should not be written by the user explicitly.

Rule L.L.2 No DOEN*n* or explicit write to LC*n* inside the same hardware loop

Description

An instruction that explicitly writes to an LC*n* register (such as DOEN.*n* or TFRA) is not allowed inside the same hardware loop *n*.

Rule L.L.6 Limitations on contents of (SA–1) of a hardware loop

Description

PCU instructions are not allowed at (SA–1) of a hardware loop, except for:

- SKIP*
- LPST.*
- LPST.SQ.*
- LPSKIP.*
- LPSKIP.SQ.*

In addition, DOEN.*n* is not allowed at SA–1 of loop *n*, which has more than one VLES (DOEN.*n* allowed at SA–1 for a 1-VLES hardware loop).

NOTES

The LP* instructions are generated by the assembler and should not be written by the user explicitly. LPST* is generated in response to the LOOPSTART*n* directive placed right before SA.

The mapper may insert a NOP at SA–1 in legacy code that does not comply with this rule.

Rule L.N.1 Nested hardware loops cannot have same LA or SA

Description

Nested hardware loops cannot have the same LA or SA.

Example

This segment of code illustrates invalid nested loops with the same LA:

```
...
st.w  r3,(r4)     ; LA
loopend1
loopend0            ; Not allowed
```

Rule L.N.2 hardware Loop bodies and nesting

Description

A loop body *n* must be surrounded by the LOOPSTART*n* and LOOPEND*n* assembly directives.

Rule L.N.7 Rules for Sequential Loops

Description

This rule is for the assembler. The programmers’ model of a sequential loop is the same as for a non-sequential loop.

- A sequential loop cannot have any loop nested within it.
- The following instructions cannot be inside a sequential loop:
  - BTB-able COF

Rule S.1 Rules for assembly sections and assembly file content

Description

This rule is for the assembler. Following are some consistency requirements for the contents of assembly files and sections:

- A hardware loop body must be contained in the same assembly section, meaning that the LOOPSTART directive must be closed by a matching LOOPEND directive
- An assembly section must be aligned to 16-words (the start PC must have the 5 LS bits zero)

Chapter 9
Debug and Trace Support

9.1 Introduction

This section describes the features in the SC3900FP core for supporting debug and trace. Most of the logic and programming model for activating debug and trace features resides in the Debug and Trace Unit (DTU) which is a block located outside the core. All the DTU registers are memory-mapped, and are accessible as part of the internal register space of the SC3900FP subsystem (also known as bank0). The DTU, as well as the debug concept and flows used to debug the SC3900FP core and subsystem are described in detail in the *SC3900FP FVP Subsystem and Cluster Reference Manual*. The user should refer to that document to get a full description of the debug and trace support of the SC3900FP subsystem, including the core.

This chapter describes a subset of the SC3900FP debug features that are directly connected with the core instruction set and program flow control, which are:

- Instructions for debug support
- Core registers for debug support
- Debug mode, and in particular:
  - Single stepping
  - Core command injection

9.2 Overview of the debug and trace features of the SC3900FP

This section describes in high level the debug features supported for the SC3900FP subsystem, in order to familiarize the reader with the main concepts and abilities of the debug logic. A full description is given in the *SC3900FP FVP Subsystem and Cluster Reference Manual*.

The main concepts of the SC3900FP subsystem debug support are as follows:

- Unified support for the core and the subsystem, in one Debug and Trace Unit (DTU)
- The debug and trace architecture is compatible with the debug chassis definition of the Power architecture
- Debug configuration is done by accessing memory mapped registers
- Debug configuration can be done by an external host, accessing the DTU registers through the SoC debug configuration bus (DCSR), or by the core itself, accessing the DTU registers locally.
- The DTU resources could be split into two partitions, independently enabled, one servicing the external host and the other the on-core debug drivers. On-core drivers are limited to self-profiling scenarios.
- Multi-core tracing is based on Nexus (IEEE ISTO 5001) standard, with an independent trace output bus

The logic supporting debug and trace in SC3900FP subsystem includes the following resources:

- Dedicated core instructions

- Independent runtime core resources for on-core debug drivers: debug exception, debug stack pointer and a debug link register
- A dedicated debug mode, allowing to single step and force execute core commands
- An address detection unit, including the following:
  - — Four address range detectors, which can be configured to detect up to four PC ranges or four data address ranges. Alternatively, each range detector can be configured to detect two exact PCs.
  - — A task ID comparators
- A profiling unit, containing the following:
  - — 6 event counters, organized in 2 triads, for profiling
  - — A reloadable counter, for debug control
- An indirect event unit for advanced cross triggering, which enables implementing user-defined state machines and event sequence detectors based on debug events
- A trace unit, which supports (partial list):
  - — Program flow trace
  - — User defined trace messages (data acquisition), based on writing to dedicated core registers
  - — Profiling messages, with a snapshots of the profiling counters

As defined in the debug chassis definition of the Power architecture, the following debug functionality is supported at the SoC level, and is described or referenced in the documentation of the SoC:

- JTAG controller, that translates JTAG transactions to memory-based accesses to the memory mapped registers in the SoC, including those of the SC3900FP subsystem
- A central run control block, which controls enabling the debug unit, and centralized debug entry and exit from debug mode, or per core
- A central debug event controller and cross trigger unit, to define, combine and transfer debug events between blocks in the SoC, including the SC3900FP subsystem
- SoC-level support of the Nexus trace streams coming from each SC3900FP subsystem

9.3 Debug Instructions

This section describes the instructions that are part of the SC3900FP core ISA and are part of debug control. The instructions are summarized in this table.

Table 9-1. Instructions for debug support

Instruction	Enabling condition[1]	Behavior when enabled	Behavior when disabled	Breaking behavior	Size	Usage
SWB	Host partition enabled	Core enters Debug mode	Illegal exception	Break before make	16 bits	Debugger inserted software breakpoint - external master

Table 9-1. Instructions for debug support (continued)

Instruction	Enabling condition[1]	Behavior when enabled	Behavior when disabled	Breaking behavior	Size	Usage
DEBUGM.*n* *n*=0..3	Host partition enabled, and per *n* in DMCSR	Core enters Debug mode	NOP	Break before make	32 bits	Compiled-in core halt
DEBUGEV.*n* *n*=0..7	—	Configurable in the DTU	NOP	After instruction, preciseness depends on the case	32 bits	Generate an event like (partial list): Start/end trace Star/end count Trigger an external unit

[1] See the *SC3900FP FVP Subsystem and Cluster Reference Manual* for more details on the host and monitor partitions and how they are enabled.

The DEBUGEV.*n* instructions may be predicated (conditioned by IF.P*x*). The other instructions have limitations on the ability to predicate them, as specified in Chapter 8, “Programming Rules and Guidelines.” Illegal exceptions caused by these instructions have a unique value written to EIDR.EID which enables the service routine to know the reason for exception, see Section 5.4.4.1.1, “ETP, EID and dispatcher address values,” and Table 5-6.

9.4 Debug Mode

Debug mode is a special processing state of the core, where the pipeline is flushed, and the core is halted by suspending the dispatch of new instructions until the SoC informs the core to continue, via subsystem interface signals. Debug mode can be used only by the external host, so the ability to enter this mode requires the host partition to be enabled, see the debug chapter in the *SC3900FP FVP Subsystem and Cluster Reference Manual*.

While the core is in debug mode, debug event generation, event counting and tracing are suspended. The subsystem timers, the core-associated watchdog timer and the trace timestamp are frozen while the core is in debug mode.

9.4.1 Entering and Exiting Debug Mode

If the host partition is enabled, the core can enter debug mode because of one of the following reasons:

- Assertion of an SoC signal, typically controlled from the SoC run control block
- Execution of the SWB instruction
- Execution of one of the DEBUGM.*n* instructions which are individually enabled in a DTU control register
- Occurrence of a debug event, which is configured to cause entry into debug mode, as configured in the DTU control registers
- Occurrence of an irrecoverable core exception, when already inside a routine servicing an irrecoverable exception (SR2.IRR set). For a description of the irrecoverable exception, see Section 5.4.4.2, “Exception, Idle and Rewind priority and protection.”

Debug mode can be exited only when a dedicated signal from the SoC typically controlled from the SoC run control block, is asserted, or when the core is reset.

When exiting debug mode, the core performs a COF operation to the address programmed in the DTU register PC_NEXT.

9.4.2 Operations Supported During Debug Mode

When the core is in debug mode, the host debugger can perform the following actions:

- Single step - execute one VLES as pointed by PC_NEXT - see Section 9.6, “Single stepping.”
- Force-execute a VLES supplied by the host (called “core command” in SC3900FP terminology). With core commands, the host debugger can download, upload or change the core state (registers), read or write memory using the virtual memory as seen by the application, or generate a direct memory access to the local memory mapped registers, see Section 9.7, “Core commands.”
- Read or write the debug control registers of the DTU via the DCSR port. Configurations will take effect after the core exits from debug mode.
- Read or write from main memory or subsystem configuration registers directly as a master of the system bus, without involving the core. Through this path the core can also perform memory and cache management operations using physical addresses.

All these operations (except for single stepping) are usually supported also while the core is in a low power mode. Unless if a full power gated mode, a debug request causes the core or the SoC to activate clocking to enable the debugger to perform these actions, however the core or SoC power controller remember the low power state the core was in and return the core to it upon exiting from debug mode.

9.4.3 instruction attributes in debug mode

When the core enters debug mode, the core programming model does not change, which includes the various attributes programmed in SR2.

When single stepping, the core performs instructions and memory accesses using the attributes of the native application, as configured in SR2. Instruction attribute override will also work as it would out of debug mode, see Section 9.6, “Single stepping.”

Core commands that access SR2 and the TID registers may modify their fields regardless of the state of the SR2.EXP bit which normally protects some fields in these registers, see Section 2.1.6.4, “Using Status Register 2 (SR2).”

Core commands can access all the DTU registers: Registers that belong to the host partition can be accessed using the DCSR memory space or, if a configuration bit is set in the DTU, via the core in the direct path to bank0 bypassing the MMU. The registers that belong to the monitor partition can be accessed using the CCSR memory space, or through the local path if configured in an MMU descriptor - in the same way the monitor itself accesses them.

SP selection and the value of DID that are used by memory accesses retain their values as programmed in the fields of SR2 and TID, respectively. Core commands could be issued with user privilege, and the values of ADID ASPSEL using the Instruction Attribute Override mechanism, see Section 4.11, “Instruction attribute override.” In case the debugger wishes to use an SP or DID value that is not programmed as the active or alternate value for these fields, it can program a different value into ASPSEL or ADID and then use attribute override.

9.5 Debug exceptions

The SC3900FP core has a single debug exception, with a user-configured address for the service routine, configured in the DTU. Debug exceptions are not supported in the DTU in this implementation.

9.6 Single stepping

Single stepping by the external host is performed, upon receiving the “exit from debug” strobe from the SoC, by performing a Change of Flow (COF) to the address programmed in the PC_NEXT register in the DTU, and immediately entering debug mode again. PC_NEXT is updated automatically to point to the next un-executed VLES. The host debugger can change the value of PC_NEXT between steps.

To the external world, the core exits debug mode for a short period and enters debug mode again once the pipe is idle after the attempted execution of one VLES.

Memory accesses that were generated by the single stepped VLES will have the attributes and privilege of the native code section (as configured in SR2). In particular, single stepped instructions would not have extra privilege that would enable them to execute privileged instructions or access otherwise protected memory.

Enabled interrupts and exceptions are serviced normally during single stepping. The debugger can configure a control bit in the DTU which disables maskable interrupts during single step, as if SR2.DI was set, without changing the value of this bit.

It should be noted that the VLES that is executed upon receiving the exit from debug strobe from the SoC may not complete due to various micro-architectural situations. The reasons for not completing a step are as follows:

- The VLES was killed due to a pending exception. This may include an external request, or an Illegal encoding or MMU exception caused by the stepped VLES itself. In such a case, PC_NEXT will point to the first VLES of the ISR.
- The VLES was killed due to a pipe re-wind condition accepted from the memory system. In this case, the host debugger should re-attempt executing the step.
- The VLES was killed due to an attempt to execute the SWB instruction. This instruction is usually inserted as part of the software breakpoint flow, so this case should be handled by the debugger as part of this flow.
- The VLES was killed due to an attempt to execute the DEBUGM.*n* instruction. In this case, the debugger should modify PC_NEXT so that the next step will skip it. DEBUGM* cannot be grouped with other instructions, so usually PC_NEXT should only be incremented by 4.
- There is a pending request from the DTU for the core to enter debug mode. In this case the debugger should clear all pending requests, as reflected in DTU registers, before attempting the next single step.

- The core is in a low power state. In this case, unless the debugger exits the core from this state, no single stepping could be performed.

Various status bits in the DTU enable the debugger to understand that the last step was killed, and because of what reason, see the debug chapter in *SC3900FP FVP Subsystem and Cluster Reference Manual*.

During single stepping the DTU ignores debug events that may be generated in parallel, such as trace generation, address detection, event counting etc. Note. however, that single stepping instructions that generate debug exceptions (such as DEBUGE.*n*) will perform a jump to the ISR.

9.7 Core commands

A core command is executed by presenting the core with an encoded VLES occupying up to 128-bits in dedicated DTU registers, see the debug chapter in *SC3900FP FVP Subsystem and Cluster Reference Manual*. This way flexible types of VLES can be executed—including multiple instructions, using any of the core execution units[1]. Only one execution set can be executed from this encoded sequence, so in most cases less than 128-bits could be configured in these registers. The hardware identifies the “end of VLES” encoding that is embedded in the 128-bit data and knows to ignore the following bits.

After writing the encoded VLES to the appropriate DTU registers, actual activation of the core command is done by writing to a bit in a DTU control register. For core commands that perform load or store instructions, an additional configuration bit enables to select if the access will be issued directly to the local cluster memory mapped registers (bank0, which includes the DTU registers), or will be handled like any other access by the MMU.

The core command may be terminated due to architectural or micro-architectural reasons, for example an Illegal encoding or an MMU exception, and a re-wind condition. The execution status of the core command is reflected in a status bit in a DTU register, see the debug chapter in *SC3900FP FVP Subsystem and Cluster Reference Manual*. The host debugger should check this bit after each execution attempt to ensure it was completed. In some cases, the host debugger has to re-issue the same core command again. The core ignores a VLES that caused an error. Exceptions of any kind are not serviced during core commands. Also, pending interrupts and exceptions do not abort the core command (except for errors that originate in the core command itself, such as MMU errors or illegal conditions). Note, however, that if the debugger issued a memory access that caused an imprecise interrupt (for example, a SkyBlue bus write error), then this interrupt remains pending and will be serviced by the core when it exits debug mode.

When a core command writes to SR2, the whole content of it could be written, unlike during execution, where some fields could be updated only upon a jump to exception or return from it. Other access attributes are inherited from the current configuration (SPSEL, PID, DID etc.). The host debugger can override the access attributes using the dedicated access override instructions and issue an access with an alternate ID using AID - see Section 4.11, “Instruction attribute override.”

The following instructions cannot be executed in the core command VLES:

- Any COF instruction
- LPSTART.*n*, LPEND.*n* and any of their flavors
- Any of the debug support instructions

[1] in previous StarCore architectures only a single AGU instruction was supported in core commands

Note that SYNC* and CLRIC instructions can be executed in core commands and are not considered a type of COF.

Appendix A
Revision History

This appendix provides a list of the major differences between the *SC3900 FVP Core Reference Manual*, Revision A through Revision C.

A.1 Changes from Revision B to Revision C

Major changes to the *SC3900 FVP Core Reference Manual*, Revision B through Revision C, are as follows:

Section, Page	**Changes**
Table 4-11./4-35	Updated table item from “PCMB.INVAL.Lx Ra,Rb,Rn” to “PCMM.INVAL.Lx Ra,Rb,Rn”
Appendix C	Updated the general description of PCMB instruction

A.2 Changes from Revision A to Revision B

Major changes to the *SC3900 FVP Core Reference Manual*, Revision A through Revision B, are as follows:

Section, Page	**Changes**
4.10.4/4-31	Corrected register allocation in the description of the destination registers of the QUERY instruction (Rm:Rn)
5.2.2/5-3	Added description of high-level syntax for the PCALC/PCALCA instructions
7.8.12.2/7-25	Added groups AAU and AAU_CTRL to the existing stall cases of AAU_PBIT
7.8.15/7-30	Removed the paragraph that restricts changes to TMTAG to when SR2.EXP is set
7.8.5/7-21	Correct G to W stalls of 8 cycles to G to L (same value)
7.8.12.1/7-24	Correct G to W stalls of 8 cycles to G to L (same value)
7.8.12.6/7-28	Correct G to W stalls of 8 cycles to G to L (same value)
7.8.12.3/7-26	Correct G to W stalls of 8 cycles to G to L+1, and to 9 cycles
Rule A.3 on page 8-6	Correct “JMP” and “BRA” to “JMP*” and “BRA*”
Rule G.S.1 on page 8-25	Added non-predicated case: “Some special memory access instructions, see Rule A.9” - for consistency with the existing definition of Rule A.9.
Rule L.L.2 on page 8-34	Correct “instructions that write to LCn” to “instructions that explicitly write to LCn”

Appendix C	Added Meta instructions and assembly aliases
Appendix C	Corrected operand ordering in the functionality of compare instructions (CMP, CMPA, FCMP)
Appendix C	Corrected the functionality of Radix 4 instructions in FFT.nT
2.5.2/2-35	Removed “An overflow indication of a DALU instruction.”

A.3 Revision A

Released in January 2013 - Baseline customer version

Appendix B Changes between SC3900 and SC3900FP

This appendix provides a list of the architectural differences between the SC3900 and SC3900FP. The changes described in this document do not include SC3900 errata corrections, which should be assumed as fixed.

B.1 Changes in the programming model (registers)

This section described additions to the register set of the core, and fields that were added in existing registers.

B.1.1 Added the THRDN register

The THRDN register is a read-only register that holds the hard-wired serial number of the thread running on this core. The thread number is across the SoC. The SC3900 supports only one thread per core, however some cores in the SoC, such as the e6500, may run multiple threads. Because of this reason, the serial SoC number of the thread running on the SC3900 may be different than the serial core number which is reflected in the COREREV register.

The new register is described in Section 2.1.7.3, “Using the Thread Number Register (THRDN)”.

B.1.2 Updated the core revision in COREREV

COREREV is a read-only register that holds among other things the hard-wired revision number of the core. The value configured in the REVMJ was changed from 0 to 1 to reflect to the software that it is running on SC3900FP.

See Section 2.1.7.2, “Using the Core Revision register (COREREV)”.

B.1.3 Added floating point status bits in GCR

As part of the added support for floating point operations, 6 new status bits were added to GCR, reflecting various arithmetic conditions of the last floating point operation. These bits are:

- FINEX - Floating point inexact flag
- FINVAL - Floating point invalid flag
- FDIVZ - Floating point divide by zero flag
- FOVER - Floating point overflow flag
- FUNDER - floating point underflow flag
- FDENI - Floating point denormalized input flag

The bits are described in Section 2.1.5.2, “General Configuration Register (GCR)”.

B.1.4 Configurable reset value for MOCR

The MOCR register, used to enable or disable various micro-architectural mechanisms in the core, will now have a reset value that can be hard-wired to a non-zero value. This will enable to disable features that are not supported in this architecture without requiring a software update during boot.

The functions controlled by MOCR are described in Section 2.1.6.7, “Using the Miscellaneous Option Configuration Register (MOCR)”.

B.2 Additions to the instruction set

This section describes in high level the new instructions that were added to the SC3900FP. The added instructions are marked in the ISA appendix.

B.2.1 Floating point instructions

The SC3900FP was added DALU instructions for supporting single precision floating point. Instructions also include DSP functions such as SIMD2 operations, multiply-add, and special lookup table functions for reciprocal, inverse square root and base 2 logarithms.

B.2.2 Integer DSP instructions

The DALU instruction set was supplemented with integer support for DSP operations, including complex, saturation, various multiplication and MAC instructions, and mode. This addition closes the gap between the ISA support for fractional data types and ISA support for integer data types.

B.2.3 Additional variants for multiplication instructions

The DALU instruction set was supplemented with additional MPY and MAC instruction variants, for better supporting baseband applications. In addition, some application-specific instructions were added for better support for FFT.

B.2.4 Miscellaneous instruction additions

The following instruction groups were supplemented:

- Supplementing the compare instruction set to allow better compiler support
- Supplementing load and store instructions with SP-relative addressing mode in cases it was not supported, making the addressing modes orthogonal across the load and store instruction set
- Bit expand instructions (DALU)
- AGU logic instructions (NANDA etc.)
- Supplementing legacy versions of ST.SRS instructions for better backwards compatibility with MOVER instructions

B.3 Relaxations in programming rules

This section describes relaxations done in the programming rules.

B.3.1 Rule A.9 - special memory accesses

B.3.1.1 Expansion of the “special accesses LD/ST” group

Rule A.9 uses in various cases a category of instructions named “special accesses load/store” group, allowing only members of this group in various situations. The definition of this group was extended to include any addressing mode, not only (Rn) and (Rn+offset) as before.

For the definition of this instruction group, see Section 8.1.1.6, “Special access LD/ST instructions”.

B.3.1.2 Added cases for the streaming attribute

The streaming attribute (STRM mnemonic) is allowed to be added to more cases of memory access, as follows:

STRM LD* + non streaming ST*

See Table 8-2 in Rule A.9.

B.3.2 Rule G.G.4 - multiple predicate update

B.3.2.1 Grouping compare instructions updating two predicates

It is now allowed to group multiple compare instructions that update predicate pairs, for example:

```
cmp.eq.x d0,d1,p0:p1 cmp.ge.x d2,d3,p0:p1
if.p5.ec cmp.eq.x d0,d1,p0:p1 if.p5.ec cmp.ge.x d2,d3,p0:p1
```

See the detailed definition and restrictions on grouping dual-predicate compares in Rule G.G.4, under “Multiple update of a predicate”.

B.3.3 Rule G.G.4a - multiple predicated writes to the same D register

It is now allowed to group two predicated DALU instructions with mutually exclusive conditions, having the same destination D register, for example:

```
if.p0 add.x d0,d1,d2       if.p1 sub.x d3,d4,d2
```

It is still the user’s responsibility to ensure that p1 and p0 are not both set, see Rule G.G.4a and Rule G.G.4b.

Appendix C
SC3900 Instruction Set Appendix

C.1 Introduction

This section describes how to read the information contained in the instruction entries and covers the following topics:

- Instruction Syntax Conventions
- Action Conventions
- Reading an Instruction Entry

C.1.1 Instruction Syntax Convention

The SC3900 mnemonic convention splits the semantical meaning into portions separated by dots, as follows:

base.[flags1].[flag2].[output lane] [Operands]

base - Indication for the basic functionality of the instruction.

- Examples -
  - ADD - addition
  - LD - Memory Load
  - ST - Memory Store
  - MAC - multiply accumulate
  - All AGU non load/store instructions end with "A". for example ADDA is the base of the AGU addition instruction, while ADD is DALU addition.

flag1 - (optional field) Minor (group specific) functional variation.

- Example - EQ in CMP.*EQ*.L #s32,Da,Pn represents that the compare instruction check equality between the input arguments.

flag2 - (optional field) Major arithmetic variation. usually supported vertically for large number of instructions.

- Examples
  - I-Integer arithmetic
  - S-Saturated arithmetic
  - R-rounded
  - U-unsigned

output lane - (optional field) Indicates the SIMD level and the native output type of the instruction.

- Convention - all DALU instruction have output lane specifier. For the AGU in case output lane is not specified the default output lane is L (32 bit).

Table C-1. Supported types

Field	Size
B	Byte
BF	Byte Fractional
W	word (16 bits)
F	Fractional (16 bits)
T	20 bits
L	Long (32 bit)
SP	single Precision (32 bit)
X	40 Bits
LL	64 bits

- Examples -
  - The output of ADD.W Da,Db,Dn is an argument of type W and the instruction is of a single instruction single data type
  - The output of MAC.2X Da:Db,Dc:Dd,Dm:Dn is of type X (40 bit) and the SIMD level is 2, i.e. two multiply accumulates are conducted in the instruction.

Syntax symbolic conventions -

Table C-2. Syntax Operands

Syntax	Description
uxxx_N	Unsigned immediate of xxx+N bits. Only xxx bits are encoded and the encoded value is shifted N bits to the left prior to usage. if the shift is 0 than it is made redundant (uxxx)
sxxx_N	Signed immediate of xxx+N bits. Only xxx bits are encoded and the encoded value is shifted N bits to the left prior to usage. if the shift is 0 than it is made redundant (sxxx)
:	Dual register concatenation. A register pair which is colon separated (ex. Da:Db) represents two registers which are not freely allocated. For example - in Da:Db the index of Db is always the index of Da+1
::	Short notation for a range of registers. For example, D0::D7 is equivalent to D0:D1:D2:D3:D4:D5:D6:D7. The registers indexes must be sequential

Table continues on the next page...

Table C-2. Syntax Operands (continued)

Syntax	Description
,	Operand separator. Two operands separated with a Comma are usually allocated without any restriction between them.
( )	Address operator. In instructions that access memory, the enclosed expression is used as the address issued to memory
SAM	Abbreviation for *Standard Addressing Modes*. For the vast majority of the load\store instructions the following SAM addressing modes are supported: • (Rn) • (Rn)+ • (Rn)- • (Rn+Rk) • (Rn)+Rk • (SP+s15) • (Rn+s15)
RSAM	Abbreviation for *Reduced Standard Addressing Modes*. For a group of the load\store instructions the following RSAM addressing modes are supported: • (Rn) • (Rn)+ • (Rn)- • (Rn+Rk) • (Rn)+Rk • (Rn+s15)

C.1.2 Conventions used in the description of the action of instructions

The action describes in pseudo code the main side-effects of the instruction. Some minor side effects, such as updating SR status bits, and some influencing factors, such as mode bits, are sometimes abstracted by means of function calls to improve clarity of the main functionality.

The table below presents semantic elements and conventions commonly used in the pseudo-code used to describe the action of an instruction.

Table C-3. Pseudo-code Syntax Elements

Syntax Element	Description
value (a) or [a]	The bit in position "a" extracted from the value. "Value" may be a register, immediate field or calculation result
value (a,b) or [a,b]	The bit range between bits in position "a" and "b" extracted from the value. "Value" may be a register, immediate field or calculation result
Mem (addr, width)	A data memory access from address "addr.", with a width of "width" bytes. In the context of a load, represents the returned data. In the context of a store, represents the destination memory that is written.
EA	Effective calculated address (after the addressing mode calculation was applied)
(EA)..(EA+x)	An alternative specification of a memory access, equivalent to: Mem(EA, x+1)
{ab}	A shorthand notation of two expressions with respective functionality, for example the expression "D{mn}.L = Da.{HL} + Db,{HL}" stands for the two expressions: "Dm.L=Da.H +Db.H" and "Dn.L=Da.L+Db.L"

Table continues on the next page...

Table C-3. Pseudo-code Syntax Elements (continued)

Syntax Element	Description
condition ? expression1 : expression2	If the condition is true, expression 1 is performed, else, expression 2 is performed.
{a,b,...n}	Concatenation operator
n{a}	Repeat operator: "a" is repeated n times, for example 3{Da[0]} is equivalent to {Da[0],Da[0],Da[0]}

The Table Below presents the commonly used operators used in action pseudo-code

Table C-4. Action Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Raising to the Power
/	Division
=	Assignment
==	Equality comparison
!=	Inequality comparison
>	Greater than comparison
<	Less than comparison
≤	Less than or equal comparison
≥	Greater than or equal comparison
|	Bitwise Or
&	Bitwise And
^	Bitwise exclusive or (XOR)
~	Bitwise Complement
!	Logic negation
|x|	Absolute value

The Table Below presents the commonly used functions used in action pseudo-code

Table C-5. Action Functions

Function	Description
(Sxx) Operand	Sign extension, or casting to xx bits a signed fixed point argument
(Uxx) Operand	Zero extension, or casting to xx bits an unsigned fixed point argument
MCTL_calc(exp)	Non-linear address modification of the expression "exp", based on the settings of MCTL. The first register mentioned in "exp" from the left is the one configured in MCTL.
NumberOfOnes( )	Counts the number of bits with the value "1" in the input expression

Table continues on the next page...

Table C-5. Action Functions (continued)

Function	Description
RND( )	Rounds the value in the expression (twos complement) to 16 bits
SATxx( )	Unconditional signed saturation to xx bits
SATU8( )	Unconditional unsigned saturation to 8 bits
SCALE_LEG( )	Scale the value up or down according to the settings of SR.SCM. In case SR.SM is set, no scaling is done
srRND( )	16-bit rounding. The rounding type is according to SR.RM. The rounding position is affected by SR.SCM
srRND_SAT32( )	Saturation to xx bits if SR.SM is set, and rounding to 16 bits
srSAT16( )	Dynamic signed saturation to 16 bits, controlled by SR.SM2
srSAT32( )	Dynamic signed saturation to 32 bits, controlled by SR.SM
SRS_xx_to_yy( )	Scale, round and saturate the value, from an input width of xx bits to an output width of yy bits. Scaling is done according to SR.SCM, rounding according to SR.RM, according to the output width. Saturation is signed, unconditioned by SR
SRS_xx_to_yy_leg( )	Scale, round and saturate the value, compatible with legacy architectures. Same functionality as SRS_xx_to_yy only that if SR.SM is set, no scaling is performed
SS_40_to_32( )	Scale and saturate the value without rounding, from an input width of 40 bits to an output width of 32 bits. Scaling is done according to SR.SCM, rounding according to SR.RM. Saturation is signed, unconditioned by SR
SS_40_to_32_leg( )	Scale and saturate the value without rounding, compatible with legacy architectures. Same functionality as SS_40_to_32 only that if SR.SM is set, no scaling is performed

C.1.3 Organization of the instructions in the document

The instructions in the appendix are organized in instruction groups, ordered alphabetically.

The instructions included within a group are *variants* with similar functionality with some differences, for example the LD.nB→ D group includes instructions that load 1, 2, 4 or 8 bytes to one or more D registers. The "n" in "nB" signifies that all SIMD variants of byte loads to D registers are included in the group.

Loosely speaking, an instruction group includes instructions with the same base mnemonic AND processing the same data type, as expressed by the output lane. SIMD variants are included in the same group. All flag1 and flag2 variants are included in the group. However, there are some exceptions to this convention.

C.1.3 Reading an Instruction Entry

Each instruction group in this appendix is organized in the following manner (note that not all sections are applicable to all of the instruction groups):

- **Instruction group header:** includes 3 items:

  - The name of the instruction group: usually the mnemonic base with the output lane. When there are some SIMD variants, the output lane is prefixed by an "n" which stands for different SIMD number, for example "ADD.nX" stands for both "ADD.X" and "ADD.2X".
  - A one-line description of the group
  - The execution unit or units involved in processing the instructions in the group. Usually it is just one execution unit, however there are some exceptions, for example, AGU instructions may frequently be executed by either an LSU or the IPU. Meta-instructions appear as "META INST". In such special cases the execution unit is specified per instruction variant in the attribute table

- **Flag1 Options** - Lists the relevant flag1 options for the instruction group with their corresponding description
- **Flag2 Options** - Lists the relevant flag2 options for the instruction group with their corresponding description
- **Instructions** - For each instruction variant the following information is provided:
  - Serial number in the table (under the column "#") - referenced by other tables in the entry
  - Syntax - Assembly form
  - Action - Pseudo code description
  - Description - brief description
- **Explicit Operands** - Lists the operands tables used explicitly in the syntax of the instruction, and the permitted values that each operand can accept. The list is accumulative across all instruction flavors in this instruction group.
  - Conventions -
    - Immediate values - The range of acceptable value is presented
    - Register fields- The index relation between the different registers in the field is presented. when the formula is complex the actual allowed values are also presented in a dedicated section: "Special Instruction Fields" at the end of the Appendix.
      - Example - The field "Da:Db" is described as "D_n:D_{n+1} 0≤n<62" where the subscript $_n$ represents the index of the first register.
- **Affect instructions** - Lists all programing model registers or mode bits which implicitly effect the behavior of the instructions. For each register or bit, there is a list of instruction variants which is affected by it. An instruction variant is referenced by its numerical index in the instruction variant table. In case all variants are affected, the text "All" appears instead.

- **Affected by Instructions** - Lists all programing model registers or status bits which are implicitly effected as a side effect of the instruction. For each register or bit, there is a list of instruction variants which are affected by it, using the same conventions as for the "Affect instructions" table.
- **Attributes** - Lists various attributes of the instructions. An attribute may be a binary attribute - meaning, if it appears, the instruction variants listed for it have this property. An attribute may also have a value, in which case the possible values are listed, and the list of relevant attributes for each is listed per value. The possible attributes in this table are described in the table below.

Table C-6. Instruction attributes

Attribute	Allowed Values	Description
Alias		This attribute indicates that the specified instructions are alias mnemonics, which could be seen as an additional name for an instruction. In case the specified instruction is a compressed entry representing several instructions, only a subset of them are aliases - the exact subset is described in the description of the entry.
Architecture	SC3900, SC3900FP	This attribute describes what architecture versions support the listed instructions. This attribute appears only if some variants are only in SC3900FP. If the attribute does not appear, it means that all variants are supported in both SC3900 and SC3900FP
Execution Unit	DALU, LSU, IPU, PCU	This attribute describes what execution units process the listed instruction variants. The attribute appears only if there is a difference between variants of the instruction. If it does not appear, then the value in the header of the instruction group applies to all variants. Note that an instruction mnemonic may stand for two encoded variants that are processed by different execution units, for example many AGU arithmetic instructions may be processed by both an LSU and the IPU. For Meta instructions, the list of execution units of the instructions it includes is given, for example 2xLSU+PCU.
Encoding Length		Specifies in how many bits the instruction is encoded, not including a suffix if needed. For Meta instructions, the accumulative length of the constituting instructions is given.
	16-bits	
	32-bits	
	32-bits in 64	A 32-bit DALU instruction that is part of a 64-bit container
	48-bits	
	64-bits	A 64-bit DALU instruction, occupying 2 DMUs
	96-bits	Relevant only for some Meta-instructions
Pipeline Behavior	(pipe template name)	Specifies the pipeline behavior of the instruction, including how many cycles it takes. The list of the pipeline template names is described in the Pipeline chapter.
No Predication		If the attribute appears, it means that it is not allowed to predicate the listed instruction with IF.Pn, as defined in programming rule G.S.1. If the attribute does not appear, all instruction variants could be predicated.

Table continues on the next page...

Table C-6. Instruction attributes (continued)

Attribute	Allowed Values	Description
IF.EC		If the attribute appears, it means that the listed instructions can also be predicated using the"If-Else-Clear" condition, in addition to normal predication. This attribute relates to a subset of predicate-generating instructions such as CMP.
Must be alone in VLES		If the attribute appears, it means that the listed instructions cannot be grouped with any other instructions and must be alone in the VLES, per the definition in programming rule G.S. 1.
Helper instruction		If the attribute appears, it means that the listed instructions require another instruction or instructions in parallel to perform the required operation. Typically such an instruction is usable as part of a Meta-instruction (which has its own instruction page).

Instructions

ABS.nL Absolute Value of 32-bit Number (DALU)

General Description

Calculate the absolute value of the signed 32-bit value (integer or fraction), with optional saturation of the result. The source extension is ignored. The destination extension is cleared.

Note that absolute value of 0x8000_0000 with no saturation results in 0x8000_0000.

Flag Options

Flag	Position	Description
S	flg2	Saturated result

Instruction Variants

#	Syntax	Operation	Description
1	ABS.L Da,Dn	`Dn = (U40)│ Da.M │`	32-bit absolute value

2	ABS.S.L Da,Dn	`if (Da.M == 0x80000000) {`
		`        Dn = 0x007fffffff`
		`        SR.SAT = 1`
		`} else {`
		`        Dn.M = │ Da.M │`
		`        Dn.E = 0`
		`}`
			32-bit absolute value with saturation

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Dn	D0-D63

Affected By Instructions

Field	Relevant Variants
SR.SAT	2

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

ABS.nT — Absolute Value of 20-bit Number — (DALU)

General Description

Calculate the absolute value of two or four packed signed 20-bit values (integer or fraction).

Note that absolute value of 0x8_0000 results in 0x8_0000.

Instruction Variants

#	Syntax	Operation	Description
1	ABS.2T Da,Dn	$Dn.WH = \lvert Da.WH \rvert$ $Dn.WL = \lvert Da.WL \rvert$	SIMD2 20-bit absolute value (two values packed in one register)
2	ABS.4T Da:Db,Dm:Dn	$Dm.WH = \lvert Da.WH \rvert$ $Dm.WL = \lvert Da.WL \rvert$ $Dn.WH = \lvert Db.WH \rvert$ $Dn.WL = \lvert Db.WL \rvert$	SIMD4 20-bit absolute value (four values packed in two register)

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

ABS.nW Absolute Value of 16-bit Number (DALU)

General Description

Calculate the absolute value of two or four packed signed 16-bit value (integer or fraction), with optional saturation of the result. The source extension is ignored. The destination extension is cleared.

Note that absolute value of 0x8000 with no saturation results in 0x8000.

Flag Options

Flag	Position	Description
S	flg2	Saturated result

Instruction Variants

#	Syntax	Operation	Description

1	ABS.2W Da,Dn	`if (SR.W20 == 1) {`
		`Dn.WH = │ Da.WH │`
		`Dn.WL = │ Da.WL │`
		`} else {`
		`Dn.E = 0`
		`Dn.H = │ Da.H │`
		`Dn.L = │ Da.L │`
		`}`
			SIMD2 16-bit absolute value (two values packed in one register)

2	ABS.4W Da:Db,Dm:Dn	`Dm.WH = (U20)│ Da.H │`
		`Dm.WL = (U20)│ Da.L │`
		`Dn.WH = (U20)│ Db.H │`
		`Dn.WL = (U20)│ Db.L │`
			SIMD4 16-bit absolute value four values packed in two registers)

3	ABS.S.2W Da,Dn	`Dn.H = SAT16(│ Da.H │)`
		`Dn.L = SAT16(│ Da.L │)`
		`Dn.E = 0`
			SIMD2 16-bit absolute value with saturation (two values packed in one register)

4	ABS.S.4W Da:Db,Dm:Dn	`Dm.H = SAT16(│ Da.H │)`
		`Dm.L = SAT16(│ Da.L │)`
		`Dn.H = SAT16(│ Db.H │)`
		`Dn.L = SAT16(│ Db.L │)`
		`Dm.E = 0`
		`Dn.E = 0`
			SIMD4 16-bit absolute value with saturation (four values packed in two registers)

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Affect Instructions

Field	Relevant Variants	Comments
SR.W20	1	

Affected By Instructions

Field	Relevant Variants
SR.SAT	3, 4

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

ABS.nX Absolute Value of 40-bit Number (DALU)

General Description

Calculate the absolute value of one or two 40-bit values (integer or fraction), optionally saturating them to a 32-bit or a 40-bit value.

The absolute value of 0x80_0000_0000 without saturation results in 0x80_0000_0000.
32-bit saturation saturates any value between 0x01_0000_0000 and 0x7F_FFFF_FFFF to the value 0x 00_7FFF_FFFF, and saturates 0x80_0000_0000 to 0x 00_7FFFF_FFFF.
40-bit saturation saturates 0x80_0000_0000 to 0x7F_FFFF_FFFF.

Flag Options

Flag	Position	Description
LEG	flg2	Legacy instruction
S	flg2	Saturated result
S40	flg2	40 bit (X) Saturation

Instruction Variants

#	Syntax	Operation	Description

1	ABS.2X Da:Db,Dm:Dn	Dm = | Da |
		Dn = | Db |
			SIMD2 40-bit absolute value (two values in two registers)

2	ABS.X Da,Dn	Da = |Dn|	40-bit absolute value
3	ABS.LEG.X Da,Dn	Da = (S40) srSAT32(|Dn|)	Legacy 40-bit absolute value - Saturate to 32-bit only if SR.SM bit is set

4	ABS.S.2X Da:Db,Dm:Dn	Dm = (S40) SAT32(|Da |)
		Dn = (S40) SAT32(|Db |)
			SIMD2 40-bit absolute value with saturation to 32-bit value (two values in two registers)

5	ABS.S.X Da,Dn	Dn = (S40) SAT32(|Da|)	40-bit absolute value with saturation to 32-bit value
6	ABS.S40.X Da,Dn	Dn = SAT40(|Da|)	40-bit absolute value with saturation to 40-bit value

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Affect Instructions

Field	Relevant Variants	Comments
SR.SM	3	

Affected By Instructions

Field	Relevant Variants
SR.SAT	3, 4, 5, 6

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

ABSA Absolute Value of 32-bit Number (LSU,IPU)

General Description

Calculates the absolute value of a signed 32-bit value.

Note that the absolute value of 0x8000_0000 results in 0x8000_0000.

Instruction Variants

#	Syntax	Operation	Description

1	ABSA Ra,Rn	`Rn = |Ra|`
		`|0x80000000| = 0x80000000`
			32-bit absolute value

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Execution unit	IPU	All
	LSU	All
Pipeline behavior	agu_AAU_LOGIC	All

ACATOR Access Attribute Override (IPU)

General Description

ACATOR is a helper instruction that is used to modify some attributes of a memory access that is grouped with it in the same VLES, such as the source of the task ID of the virtual address, or some cache attributes. The attributes are specified in one or two immediate operands of the instruction. The attribute is usually activated by setting the appropriate bit in one of the operands, in which case it affects both parallel accesses (although mostly not allowed in rule A.9). Some attributes allow explicit setting per bus

The attributes affected by the u4 flag for ACATOR.P:
u4[1] - AID - if set, forces the parallel access to use the task ID from AID in the virtual address instead of DID (or PID for program cache instructions)

The attributes affected by the u6 flag for both ACATOR variants:

u6[1:0] - STRM - the following settings determine if the parallel accesses on LSU0 and LSU1 use the attribute:
00 - no streaming
01 - streaming attribute only for LSU0
10 - streaming attribute for both LSU0 and LSU1
11 - streaming attribute only for LSU1
u6[2] - ITW - if set, marks the parallel accesses with the attribute

For a more detailed description of the attributes that could be affected by this instruction, see the Address Generation chapter in this manual. For a definition of what memory accesses are allowed to be grouped with ACATOR, and with what attributes, refer to programming rule A.9.

Flag Options

Flag	Position	Description
P	flg1	Privileged

Instruction Variants

#	Syntax	Description
1	**ACATOR #u6**	**Specify only cache-related attributes**

	`u6[1:0]: select the STRM attribute per bus`
	`u6[2]: if set, add the ITW attribute`	

2	**ACATOR.P #u6,#u4**	**Specify an alternate task ID source for the virtual address, and/or a cache related attributes**

	`u4[1]: If set, use the alternate AID`
	`u6[1:0]: select the STRM attribute per bus`
	`u6[2]:if set, use the ITW attribute`	

Explicit Operands

Operand	Permitted Values
u4	$0 \leq u4 < 2^4$
u6	$0 \leq u6 < 2^6$

Affect Instructions

Field	Relevant Variants	Comments
TID	2	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Helper instruction		All
Pipeline behavior	pcu_control	All

ACS Add, Compare, and Select High Portion (DALU)

General Description

The ACS instruction is aimed to support the Viterbi algorithm.

It shifts the 64-bit value in the BTR1:BTR0 registers two bits to the right (the two least significant bits of BRT1 are shifted to the two most significant bits of BTR0), and then performs two pairs of signed addition or subtraction operations between the specified high or low portions of the source data registers, compares and finds the largest result from each pair of operations, and writes it to the appropriate portion of the destination data register, and updates the appropriate bit of BTR1.

For the first pair of operations, writes the largest result to the high portion of the destination register and updates bit 30 of BTR1. For the second pair or operations, writes the largest result to the low portion of the destination and updates bit 31.

For each pair of operations, clears the appropriate BTR1 bit if the result of the first operation is greater than the result of the second; otherwise, sets the bit. If any of the addition or subtraction operations overflows, the result is saturated to a 16-bit maximum or minimum value. The extension byte of the destination register is cleared.

Flag Options

Flag	Position	Description
H	flg1	High 16-bit portion
L	flg1	Use the low portion

Instruction Variants

#	Syntax	Description
1	**ACS.H.2W Da.X,Da.Y,Db,Dn**	**ACS on Dn.H Add/Add**

```
BTR1:BTR0 >> 2
If ((Dn.H + Da.y) > (Db.H + Da.x)),
  then BTR1[30] = 0, Dn.H = (Dn.H + Da.y),
  else BTR1[30] = 1, Dn.H = (Db.H + Da.x)
If ((Dn.H - Da.y) > (Db.H - Da.x)),
  then BTR1[31] = 0, Dn.L = (Dn.H - Da.y),
  else BTR1[31] = 1, Dn.L = (Db.H - Da.x)
```

#	Syntax	Description
2	**ACS.H.2W Da.X,-Da.Y,Db,Dn**	**ACS on Dn.H Add/Sub**

```
BTR1:BTR0 >> 2
If ((Dn.H - Da.y) > (Db.H + Da.x)),
  then BTR1[30] = 0, Dn.H = (Dn.H - Da.y),
  else BTR1[30] = 1, Dn.H = (Db.H + Da.x)
If ((Dn.H + Da.y) > (Db.H - Da.x)),
  then BTR1[31] = 0, Dn.L = (Dn.H + Da.y),
  else BTR1[31] = 1, Dn.L = (Db.H - Da.x)
```

#	Syntax	Description
3	**ACS.H.2W -Da.X,Da.Y,Db,Dn**	**ACS on Dn.H Sub/Add**

Table continues on the next page...

#	Syntax	Description

```
BTR1:BTR0 >> 2
If ((Dn.H + Da.y) > (Db.H - Da.x)),
  then BTR1[30] = 0, Dn.H = (Dn.H + Da.y),
  else BTR1[30] = 1, Dn.H = (Db.H + Da.x)
If ((Dn.H - Da.y) > (Db.H + Da.x)),
  then BTR1[31] = 0, Dn.L = (Dn.H - Da.y),
  else BTR1[31] = 1, Dn.L = (Db.H + Da.x)
```

#	Syntax	Description
4	ACS.H.2W -Da.X,-Da.Y,Db,Dn	ACS on Dn.H Sub/Sub

```
BTR1:BTR0 >> 2
If ((Dn.H - Da.y) > (Db.H - Da.x)),
  then BTR1[30] = 0, Dn.H = (Dn.H - Da.y),
  else BTR1[30] = 1, Dn.H = (Db.H - Da.x)
If ((Dn.H + Da.y) > (Db.H + Da.x)),
  then BTR1[31] = 0, Dn.L = (Dn.H + Da.y),
  else BTR1[31] = 1, Dn.L = (Db.H + Da.x)
```

#	Syntax	Description
5	ACS.L.2W Da.X,Da.Y,Db,Dn	ACS on Dn.L Add/Add

```
BTR1:BTR0 >> 2
If ((Db.L + Da.x) > (Dn.L + Da.y)),
  then BTR1[30] = 0, Dn.H = (Db.L + Da.x),
  else BTR1[30] = 1, Dn.H = (Dn.L + Da.y)
If ((Db.L - Da.x) > (Dn.L - Da.y)),
  then BTR1[31] = 0, Dn.L = (Db.L - Da.x),
  else BTR1[31] = 1, Dn.L = (Dn.L - Da.y)
```

#	Syntax	Description
6	ACS.L.2W Da.X,-Da.Y,Db,Dn	ACS on Dn.L Add/Sub

```
BTR1:BTR0 >> 2
If ((Db.L + Da.x) > (Dn.L - Da.y)),
  then BTR1[30] = 0, Dn.H = (Db.L + Da.x),
  else BTR1[30] = 1, Dn.H = (Dn.L - Da.y)
If ((Db.L - Da.x) > (Dn.L + Da.y)),
  then BTR1[31] = 0, Dn.L = (Db.L - Da.x),
  else BTR1[31] = 1, Dn.L = (Dn.L + Da.y)
```

#	Syntax	Description
7	ACS.L.2W -Da.X,Da.Y,Db,Dn	ACS on Dn.L Sub/Add

```
BTR1:BTR0 >> 2
If ((Db.L - Da.x) > (Dn.L + Da.y)),
  then BTR1[30] = 0, Dn.H = (Db.L - Da.x),
  else BTR1[30] = 1, Dn.H = (Dn.L + Da.y)
If ((Db.L + Da.x) > (Dn.L - Da.y)),
  then BTR1[31] = 0, Dn.L = (Db.L + Da.x),
  else BTR1[31] = 1, Dn.L = (Dn.L - Da.y)
```

#	Syntax	Description
8	ACS.L.2W -Da.X,-Da.Y,Db,Dn	ACS on Dn.L Sub/Sub

```
BTR1:BTR0 >> 2
If ((Db.L - Da.x) > (Dn.L - Da.y)),
  then BTR1[30] = 0, Dn.H = (Db.L - Da.x),
  else BTR1[30] = 1, Dn.H = (Dn.L - Da.y)
If ((Db.L + Da.x) > (Dn.L + Da.y)),
  then BTR1[31] = 0, Dn.L = (Db.L + Da.x),
  else BTR1[31] = 1, Dn.L = (Dn.L + Da.y)
```

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Db	D0-D63

Table continues on the next page...

Operand	Permitted Values
Dn	`D0-D63`
X	`L, H`
Y	`L, H`

Affect Instructions

Field	Relevant Variants	Comments
BTR0, BTR1	All	
SR.SM2	All	

Affected By Instructions

Field	Relevant Variants
BTR0, BTR1	All
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Helper instruction		All
Pipeline behavior	dalu_DAU_ACS2H	1, 2, 3, 4
	dalu_DAU_ACS2L	5, 6, 7, 8

ADD.LL Add 64-bit Values (DALU)

General Description

Add two 64-bit values (integer or fraction) into a 64-bit result. A 64-bit value is held in a register pair. The extension of the sources is ignored, and the extension of the result is cleared.

In case of overflow result is wrapped arround.

Instruction Variants

#	Syntax	Description
1	**ADD.LL Da:Db,Dc:Dd,Dm:Dn**	**64-bit add**

```
Dm = (U40)(({Da.M, Db.M}) + ({Dc.M, Dd.M}))[63:32]
Dn = (U40)(({Da.M, Db.M}) + ({Dc.M, Dd.M}))[31:0]
```

Explicit Operands

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU_Y	All

ADD.nT Add 20-bit Values (DALU)

General Description

The instruction normally adds 20-bit values that are stored in the WH or WL portions of data registers. In Dual 16-bit saturation mode (SR.SM2 set), some variants add two 16-bit portions and saturate the result into 16-bits.

Variants that generate one result (ADD.T) are affected by SR.W20 and SR.SM2. In Dual 20-bit mode and non-saturating 16-bit mode (SM2 clear), two 20-bit portions (WH or WL) from the two source registers are added, and one 20-bit value written to a 20-bit portion in the destination. The other portion is not changed.
In Dual 16-bit saturation mode (SM2 set, W20 clear), only 16-bit portions from the H or L portions of the source registers are added, and the result is saturated to 16-bits. The result updates the H or L portion of the destination, and the other portion is not changed. the extension is cleared.
The SIMD2 version (ADD.2T) is not affected by SR and performs two 20-bit additions in parallel.

Note that when SM2 is clear, all variants can be used to add 16-bit values, if the extensions are ignored.

Instruction Variants

#	Syntax	Description
1	**ADD.2T Da,Db,Dn**	**SIMD2 20-bit addition (two values in two registers)**

	`Dn.WH = ((S21)Da.WH + (S21)Db.WH)[19:0]`
	`Dn.WL = ((S21)Da.WL + (S21)Db.WL)[19:0]`	

2	**ADD.T Da.h,Dn.h**	**High source and high destination 20-bit/16-bit addition**

	`if (SR.SM2==1 && SR.W20==0 )`
	`  Dn.H = srSAT16(Dn.H + Da.H); Dn.L = Dn.L; Dn.E= 0`
	`else`
	`  Dn.WH = (Dn.WH + Da.WH); Dn.WL = Dn.WL`	

3	**ADD.T Da.h,Dn.l**	**High source and low destination 20-bit/16-bit addition**

	`if (SR.SM2==1 && SR.W20==0)`
	`  Dn.L = srSAT16(Dn.L + Da.H); Dn.H = Dn.H; Dn.E = 0`
	`else`
	`  Dn.WL = (Dn.WL+ Da.WH); Dn.WH = Dn.WH`	

4	**ADD.T Da.l,Dn.h**	**Low source and high destination 20-bit/16-bit addition**

	`if (SR.SM2==1 && SR.W20==0)`
	`  Dn.H = (Dn.H + Da.L); Dn.L = Dn.L; Dn.E = 0`
	`else`
	`  Dn.WH = (Dn.WH + Da.WL); Dn.WL = Dn.WL`	

5	**ADD.T Da.l,Dn.l**	**Low source and low destination 20-bit/16-bit addition**

	`if (SR.SM2==1 && SR.W20==0)`
	`  Dn.L = srSAT16(Dn.L + Da.L); Dn.H = Dn.H; Dn.E = 0`
	`else`
	`  Dn.WL = (Dn.WL+ Da.WL); Dn.WH = Dn.WH`	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Db	D0-D63
Dn	D0-D63

Affect Instructions

Field	Relevant Variants	Comments
SR.SM2	2, 3, 4, 5	
SR.W20	2, 3, 4, 5	

Affected By Instructions

Field	Relevant Variants
SR.SAT	2, 3, 4, 5

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

ADD.nX Add 40-bit Values (DALU)

General Description

Adds two 40-bit values (integer or fraction) into 40-bit result. If the first source operand is an immediate value, it right aligns and zero-extends the immediate value to 40 bits before adding.
ADD.X performs no saturation on the addition result.
ADD.2X is an SIMD2 version which adds two pairs of D registers.
ADD.S.X saturates the addition result to 32-bit value represented in 40-bit.
ADD.LEG.X does not saturate the result if the SM bit in SR is cleared (default), and saturates the result to 32 bits if the SM bit in SR is set - matching the legacy behavior.

Flag Options

Flag	Position	Description
LEG	flg2	Legacy instruction
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**ADD.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 addition of two register pairs**

	`Dm = Da + Dc`
	`Dn = Db + Dd`
	`Alias, encoded as: SOD.AAII.2X Da:Db,Dc:Dd,Dm:Dn`	

2	**ADD.X Da,Db,Dn**	**Add two registers**
	`Dn = (Da + Db)[39:0]`	
3	**ADD.X #u5,Da,Dn**	**Add immediate and a register**
	`Dn = ((U40)u5 + Da)[39:0]`	
4	**ADD.LEG.X Da,Db,Dn**	**Add two registers, saturate according to mode (legacy behavior)**
	`Dn = srSAT32((Da + Db))`	
5	**ADD.LEG.X #u5,Da,Dn**	**Add immediate and a register, saturate according to mode (legacy behavior)**
	`Dn = srSAT32(((U40)u5 + Da))`	
6	**ADD.S.X Da,Db,Dn**	**Add two registers and saturate**
	`Dn = SAT32((Da + Db))`	

Explicit Operands

Operand	Permitted Values
Da	`D0-D63`

Table continues on the next page...

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	
u5	$0 \le u5 < 2^5$	

Affect Instructions

Field	Relevant Variants	Comments
SR.SM	4, 5	

Affected By Instructions

Field	Relevant Variants
SR.SAT	4, 5, 6

Instruction Attributes

Attribute	Value	Relevant Variant #
Alias		1
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

ADDA Add 32-bit Values (LSU,IPU)

General Description

Adds two 32-bit values into 32-bit result. If the first source operand is an immediate value, it is aligned to the right and sign-extended to 32 bits before adding.

In non-linear variants (without the LIN flag), the modifier mode specified in the MCTL register may affect the operation. For ADDA of two registers, the MCTL mode is determined by Rb (if one of R0-R7). For ADDA with an immediate value, the MCTL mode is determined by Ra (if one of R0-R7).

Flag Options

Flag	Position	Description
LIN	flg2	Linear (Unlike Modulo), no MCTL impact

Instruction Variants

#	Syntax	Description
1	**ADDA Ra,Rb,Rn**	**Add two registers according to MCTL**
	`Rn = MCTL_calc (Rb + Ra)`	
2	**ADDA #s16,Ra,Rn**	**Add signed 16-bit immediate and a register according to MCTL**
	`Rn = MCTL_calc (Ra + s16)`	
3	**ADDA #u5,Ra,Rn**	**Add unsigned 5-bit immediate and a register according to MCTL**
	`Rn = MCTL_calc (Ra + u5)`	
4	**ADDA.LIN Ra,Rb,Rn**	**Add two registers using linear arithmetic, not affected by MCTL**
	`Rn = Ra + Rb`	
5	**ADDA.LIN #s16,Ra,Rn**	**Add signed 16-bit immediate and a register using linear arithmetic, not affected by MCTL**
	`Rn = Ra + (S32)s16`	
6	**ADDA.LIN #s16,SP,Rn**	**Add signed 16-bit immediate and stack pointer, into a register, using linear arithmetic, not affected by MCTL**
	`Rn = SP + (S32)s16`	
7	**ADDA.LIN #s9_3,SP**	**Add signed 9-bit immediate and stack pointer, into the stack pointer, using linear arithmetic, not affected by MCTL**
	`SP = (SP + (S32)s9_3) & 0xfffffff8`	

Table continues on the next page...

#	Syntax	Description
8	**ADDA.LIN #s9,SP,Rn**	**Add signed 9-bit immediate and stack pointer, into a register, using linear arithmetic, not affected by MCTL**
	`Rn = SP + (S32)s9`	
9	**ADDA.LIN #u5,Ra,Rn**	**Add unsigned 5-bit immediate and a register using linear arithmetic, not affected by MCTL**
	`Rn = Ra + (U32)u5`	

Explicit Operands

Operand	Permitted Values	
Ra	R0-R31	
Rb	R0-R31	
Rn	R0-R31	
s16	$-2^{15} \le s16 < 2^{15}$	
s9	$-2^{8} \le s9 < 2^{8}$	
s9_3	$-2^{11} \le s9\_3 < 2^{11}$;	s9_3 & 0x7 == 0
u5	$0 \le u5 < 2^{5}$	

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2, 3	
SP, ESP, TSP, DSP	6, 7, 8	
SR2.SPSEL	6, 7, 8	

Affected By Instructions

Field	Relevant Variants
SP, ESP, TSP, DSP	7

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	6
Encoding length	32-bits	1, 3, 4, 7, 8, 9
	48-bits	2, 5, 6
Execution unit	IPU	4, 5, 6, 7, 8, 9
	LSU	1, 2, 3, 4, 5, 6, 8, 9
Pipeline behavior	agu_AAU	4, 5, 9
	agu_AAU_MCTL	1, 2, 3
	agu_AAU_SP	6, 7, 8

ADDC.L Add with Carry 40-bit Values and 32-bit carry out (DALU)

General Description

Adds two 40-bit values (integer or fraction) with/without carry into a 40-bit result, and update carry bit if result overflows 32-bit. If the first source operand is an immediate value, it right aligns and zero-extends the immediate value to 40 bits before adding.

Flag Options

Flag	Position	Description
RW	flg1	Carry Read and Write
WO	flg1	Carry Write Only
LEG	flg2	Legacy instruction

Instruction Variants

#	Syntax	Operation	Description

1	ADDC.WO.L Da,Db,Dn	`Dn = Da + Db`
		`SR.C = Carry (31)`
			Add two registers, update carry

2	ADDC.WO.L #u5,Da,Dn	`Dn = Da + (U40)u5`
		`SR.C = Carry (31)`
			Add immediate and a register, update carry

3	ADDC.RW.LEG.L Da,Db,Dn	`Dn = Da + Db + (U40)SR.C`
		`SR.C = Carry (31)`
			Add two registers and carry, update carry

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Db	D0-D63
Dn	D0-D63
u5	$0 \leq u5 < 2^5$

Affect Instructions

Field	Relevant Variants	Comments
SR.C	3	

Affected By Instructions

Field	Relevant Variants
SR.C	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

ADDC.X Add with Carry 40-bit Values (DALU)

General Description

Adds two 40-bit values (integer or fraction) with/without the carry (SR.C) into a 40-bit result, and updates the carry bit if the result overflows 40 bits. If the first source operand is an immediate value, it is right aligned and zero-extended to 40 bits before adding.

Flag Options

Flag	Position	Description
RO	flg1	Carry Read Only
WO	flg1	Carry Write Only
LEG	flg2	Legacy instruction

Instruction Variants

#	Syntax	Description
1	**ADDC.RO.X Da,Db,Dn**	**Add two registers and carry in**
	`Dn = (Da + Db + SR.C)[39:0]`	
2	**ADDC.WO.X Da,Db,Dn**	**Add two registers and carry in, update carry**

	`Dn = (Da + Db)[39:0]`
	`SR.C = Carry (39)`	

3	**ADDC.WO.X #u5,Da,Dn**	**Add immediate and a register, update carry**

	`Dn = ((U40)u5 + Da)[39:0]`
	`SR.C = Carry (39)`	

4	**ADDC.WO.LEG.X Da,Db,Dn**	**Add two registers and carry in, saturate according to SR, update carry**

	`SR.C = Carry (39)`
	`Dn = srSAT32((Da + Db))`	

5	**ADDC.WO.LEG.X #u5,Da,Dn**	**Add immediate and a register, saturate according to SR, update carry**

	`SR.C = Carry (39)`
	`Dn = srSAT32(((U40)u5 + Da))`	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Db	D0-D63
Dn	D0-D63
u5	$0 \le u5 < 2^5$

Affect Instructions

Field	Relevant Variants	Comments
SR.C	1	
SR.SM	4, 5	

Affected By Instructions

Field	Relevant Variants
SR.C	2, 3, 4, 5
SR.SAT	4, 5

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

ADDLA Add with N bits arithmetic shift (LSU,IPU)

General Description

Shifts the bits in the first 32-bit source register N positions to the left, and adds with the second 32-bit source to the result. The bits that were shifted out are discarded.

In non-linear variants (without the LIN flag), if the non-shifted source (Rb) is one of R0-R7, the modifier mode specified in the MCTL register affects the operation.

Flag Options

Flag	Position	Description
1	flg1	Pre-shift amount
2	flg1	Pre-shift amount
3	flg1	Pre-shift amount
4	flg1	Pre-shift amount
5	flg1	Pre-shift amount
6	flg1	Pre-shift amount
LIN	flg2	Linear (Unlike Modulo), no MCTL impact

Instruction Variants

#	Syntax	Description
1	**ADDLA.1 Ra,Rb,Rn**	**Add with 1 bit Left shift**
	`Rn = MCTL_calc (Rb + (Ra << 1))`	
2	**ADDLA.2 Ra,Rb,Rn**	**Add with 2 bit Left shift**
	`Rn = MCTL_calc (Rb + (Ra << 2))`	
3	**ADDLA.3 Ra,Rb,Rn**	**Add with 3 bit Left shift**
	`Rn = MCTL_calc (Rb + (Ra << 3))`	
4	**ADDLA.4 Ra,Rb,Rn**	**Add with 4 bit Left shift**
	`Rn = MCTL_calc (Rb + (Ra << 4))`	
5	**ADDLA.5 Ra,Rb,Rn**	**Add with 5 bit Left shift**
	`Rn = MCTL_calc (Rb + (Ra << 5))`	
6	**ADDLA.6 Ra,Rb,Rn**	**Add with 6 bit Left shift**
	`Rn = MCTL_calc (Rb + (Ra << 6))`	
7	**ADDLA.1.LIN Ra,Rb,Rn**	**Linear add with 1 bit Left shift**
	`Rn = (Ra << 1) + Rb`	

Table continues on the next page...

#	Syntax	Description
8	**ADDLA.2.LIN Ra,Rb,Rn**	**Linear add with 2 bit Left shift**
	`Rn = (Ra << 2) + Rb`	
9	**ADDLA.3.LIN Ra,Rb,Rn**	**Linear add with 3 bit Left shift**
	`Rn = (Ra << 3) + Rb`	
10	**ADDLA.4.LIN Ra,Rb,Rn**	**Linear add with 4 bit Left shift**
	`Rn = (Ra << 4) + Rb`	
11	**ADDLA.5.LIN Ra,Rb,Rn**	**Linear add with 5 bit Left shift**
	`Rn = (Ra << 5) + Rb`	
12	**ADDLA.6.LIN Ra,Rb,Rn**	**Linear add with 6 bit Left shift**
	`Rn = (Ra << 6) + Rb`	

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`
Rb	`R0-R31`
Rn	`R0-R31`

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2, 3, 4, 5, 6	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Execution unit	IPU	7, 8, 9, 10, 11, 12
	LSU	All
Pipeline behavior	agu_AAU	7, 8, 9, 10, 11, 12
	agu_AAU_MCTL	1, 2, 3, 4, 5, 6

ADDM.nX Add mixed - 16-bit and 40-bit Numbers (DALU)

General Description

Adds one or two 16-bit values with a 40-bit value into a 40-bit result. The 16-bit values are right aligned and sign extended.

Flag Options

Flag	Position	Description
HL	flg1	High 16 bits from the first argument and Low 16 bits from the second argument

Instruction Variants

#	Syntax	Description
1	**ADDM.X Da.h,Db,Dn**	**Add high 16-bit with 40-bit**
	`Dn = (Da.H + Db)[39:0]`	
2	**ADDM.X Da.l,Db,Dn**	**Add low 16-bit with 40-bit**
	`Dn = (Da.L + Db)[39:0]`	
3	**ADDM.HL.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 add high 16-bit with low 16-bit and 40-bit**

	`Dm = (Da.L + Da.H + Dc)[39:0]`
	`Dn = (Db.L + Db.H + Dd)[39:0]`	

4	**ADDM.HL.X Da,Db,Dn**	**Add high 16-bit with low 16-bit and 40-bit**
	`Dn = (Da.L + Da.H + Db)[39:0]`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_ADDM	All

AND.nL Bitwise AND (32-bit) (DALU)

General Description

Perform a bitwise AND operation on the lower 32 bits of a D register

Instruction Variants

#	Syntax	Description
1	**AND.L #u32,Da,Dn**	**AND the bits of a D register with an unsigned 32-bit immediate. The extension of the destination is not changed**
	`Dn = {Da.E, ((Da.M) & u32)}`	

Explicit Operands

Operand	Permitted Values
Da	`D0-D63`
Dn	`D0-D63`
u32	$0 \leq u32 < 2^{32}$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	64-bits	All
Pipeline behavior	dalu_LBM	All

AND.nX Bitwise AND (40-bit) (DALU)

General Description

Perform a bitwise AND operation on all the bits of a D register

Instruction Variants

#	Syntax	Operation	Description

1	AND.2X Da:Db,Dc:Dd,Dm:Dn	`Dm = Da & Dc`
		`Dn = Db & Dd`
			SIMD2 AND operation of two pairs of D registers into two destination registers, respectively

2	AND.X Da,Db,Dn	`Dn = Da & Db`	AND the bits of two D registers
3	AND.X #s32,Da,Dn	`Dn = (S40)s32 & Da`	AND the bits of a D register with a sign-extended 32-bit immediate

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	
s32	$-2^{31} \le s32 < 2^{31}$	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	1, 2
	64-bits	3
Pipeline behavior	dalu_LBM	All

ANDA Bitwise AND (32-bit) (LSU,IPU)

General Description

Performs a bitwise AND operation on R and control registers. Variants include 32-bit bitwise AND on values in two R registers, and 16-bit AND between an unsigend immediate value and a 16-bit register portion (wither high or low). The register is either an R regisetr or a control register. The other portion of the register is not changed.

Instruction Variants

#	Syntax	Description
1	**ANDA Ra,Rb,Rn**	**AND the bits of two R registers**
	`Rn = Ra & Rb`	
2	**ANDA #u16,C4.H**	**Bit-wise AND of the higher 16-bit control register portion with an unsigned immediate.**

	`C4.H = C4.H & u16`
	`Alias, encoded as: BMCLRA #(~u16) C4.H`	

3	**ANDA #u16,C4.L**	**Bit-wise AND of the lower 16-bit control register portion with an unsigned immediate.**

	`C4.L = C4.L & u16`
	`Alias, encoded as:`	

4	**ANDA #u16,Rn.H**	**Bit-wise AND of the higher 16-bit R register portion with an unsigned immediate.**

	`Rn.H = Rn.H & u16`
	`Alias, encoded as: BMCLRA #(~u16) Rn.H`	

5	**ANDA #u16,Rn.L**	**Bit-wise AND of the lower 16-bit R register portion with an unsigned immediate.**

	`Rn.L = Rn.L & u16`
	`Alias, encoded as: BMCLRA #(~u16) Rn.L`	

Explicit Operands

Operand	Permitted Values				
C4	EIDR, SR2,	GCR, TMTAG	MCTL,	MOCR,	SR,
Ra	R0-R31				
Rb	R0-R31				
Rn	R0-R31				
u16	$0 \leq u16 < 2^{16}$				

Instruction Attributes

Attribute	Value	Relevant Variant #
Alias		2, 3, 4, 5
Encoding length	32-bits	1
	48-bits	2, 3, 4, 5
Execution unit	IPU	All
	LSU	1
Pipeline behavior	agu_AAU_CTRL_REG	2, 3
	agu_AAU_LOGIC	1, 4, 5

ASH.LL Arithmetic Shift of 64 Bits (DALU)

General Description

Arithmetically shift a 64-bit operand taken from two concatenated 32-bit portions of a D register pair, and store the result in a data register pair. The shift could be to the left or to the right. The shift amount is specified either as an unsigned immediate operand, or taken from a signed field in an additional register. In such a case, a positive value in this field will shift the operand in the direction specified in the mnemonic, and a negative value will shift the operand by the same amount to the other direction. A shift to the right will sign extend the operand up to bit 64, and discard bits from the right. A left shift will discard bits from the left, and add zeros to the inserted bits to the right of the operand. The extensions of the source data registers do not participate in the shift. The extensions of the destination registers are written with zeros. A right shift by 64 will return a value of all zeros or all ones depending on the sign of the source operand.

Flag Options

Flag	Position	Description
LFT	flg1	Left
RGT	flg1	Right

Instruction Variants

#	Syntax	Description
1	**ASH.LFT.LL Da,Dc:Dd,Dm:Dn**	**Arithmetic shift left according to a signed field in Da[6:0]. A positive value is a left shift, a negative is a right shift.**

	`Dn = (U40)((((S16)Da[6:0]) > 0) ? ({Dc.M, Dd.M}) << ((S16)Da[6:0]) : ({Dc.M, Dd.M}) >>`
	`- ((S16)Da[6:0]))[31:0]`
	`Dm = (U40)((((S16)Da[6:0]) > 0) ? ({Dc.M, Dd.M}) << ((S16)Da[6:0]) : ({Dc.M, Dd.M}) >>`
	`- ((S16)Da[6:0]))[63:32]`	

2	**ASH.LFT.LL #u6,Da:Db,Dm:Dn**	**Arithmetic shift left according to an unsigned immediate filed.**

	`Dn = (U40)(({Da.M, Db.M}) << u6)[31:0]`
	`Dm = (U40)(({Da.M, Db.M}) << u6)[63:32]`	

3	**ASH.RGT.LL Da,Dc:Dd,Dm:Dn**	**Arithmetic shift right according to a signed field in Da[6:0]. A positive value is a right shift, a negative is a left shift.**

	`Dn = (U40)((((S16)Da[6:0]) > 0) ? ({Dc.M, Dd.M}) >> ((S16)Da[6:0]) : ({Dc.M, Dd.M}) <<`
	`- ((S16)Da[6:0]))[31:0]`
	`Dm = (U40)((((S16)Da[6:0]) > 0) ? ({Dc.M, Dd.M}) >> ((S16)Da[6:0]) : ({Dc.M, Dd.M}) <<`
	`- ((S16)Da[6:0]))[63:32]`	

4	**ASH.RGT.LL #u6,Da:Db,Dm:Dn**	**Arithmetic shift right according to an unsigned immediate filed.**

	`Dn = (U40)(({Da.M, Db.M}) >> u6)[31:0]`
	`Dm = (U40)(({Da.M, Db.M}) >> u6)[63:32]`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
u6	$0 \le u6 < 2^6$	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	2, 4
	dalu_LBM_Y	1, 3

ASH.nL Arithmetic Shift of 32 Bits in Data Registers (DALU)

General Description

Arithmetically shift a 32-bit operand taken from the low portion of a D register. The shift could be to the left or to the right. The shift amount is specified either as an unsigned immediate operand, or taken from a signed field in an additional register. In such a case, a positive value in this field will shift the operand in the direction specified in the mnemonic, and a negative value will shift the operand by the same amount to the other direction. A shift to the right will sign extend the operand up to bit 32, and discard bits from the right. A left shift will discard bits from the left, and add zeros to the inserted bits to the right of the operand. The extension of the source data register does not participate in the shift. The extension of the destination register is written with zeros. A right shift by 32 will return a value of all zeros or all ones depending on the sign of the source operand. SIMD variants perform two parallel shifts by the same amount on two input-output register pairs. Additional variants saturate the destination to either 0x7FFF_FFFF or 0x8000_0000 if the left shift spills meaningful bits. In case saturation occurs, SR.SAT is set.

Flag Options

Flag	Position	Description
LFT	flg1	Left
RGT	flg1	Right
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**ASH.LFT.2L Da,Dc:Dd,Dm:Dn**	**SIMD2 arithmetic shift left of Dc into Dm, and Dd into Dn, according to a signed field in Da[5:0]. A positive value is a left shift, a negative is a right shift.**

	`Dm = (U40)(((S7)Da[5:0]) > 0) ? Dc << ((S7)Da[5:0]) : Dc >> - ((S7)Da[5:0])`
	`Dn = (U40)(((S7)Da[5:0]) > 0) ? Dd << ((S7)Da[5:0]) : Dd >> - ((S7)Da[5:0])`	

2	**ASH.LFT.2L #u5,Da:Db,Dm:Dn**	**SIMD2 arithmetic shift left of Da into Dm, and Db into Dn, according to an unsigned immediate.**

	`Dm = (U40)Da << u5`
	`Dn = (U40)Db << u5`	

3	**ASH.LFT.L Da,Db,Dn**	**Arithmetic shift left of Db into Dn, according to a signed field in Da[5:0]. A positive value is a left shift, a negative is a right shift.**
	`Dn = (U40)(((S7)Da[5:0]) > 0) ? Db << ((S7)Da[5:0]) : Db >> - ((S7)Da[5:0])`	

Table continues on the next page...

#	Syntax	Description
4	**ASH.LFT.L #u5,Da,Dn**	**Arithmetic shift left of Da into Dn, according to an unsigned immediate.**
	`Dn = (U40)Da << u5`	
5	**ASH.RGT.2L Da,Dc:Dd,Dm:Dn**	**SIMD2 arithmetic shift right of Dc into Dm, and Dd into Dn, according to a signed field in Da[5:0]. A positive value is a right shift, a negative is a left shift.**

	`Dm.L = ((((S7)Da[5:0]) > 0) ? Dc >> ((S7)Da[5:0]) : Dc << - ((S7)Da[5:0]))[15:0]`
	`Dm.H = ((((S7)Da[5:0]) > 0) ? Dc >> ((S7)Da[5:0]) : Dc << - ((S7)Da[5:0]))[31:16]`
	`Dm.E = 0`
	`Dn.L = ((((S7)Da[5:0]) > 0) ? Dd >> ((S7)Da[5:0]) : Dd << - ((S7)Da[5:0]))[15:0]`
	`Dn.H = ((((S7)Da[5:0]) > 0) ? Dd >> ((S7)Da[5:0]) : Dd << - ((S7)Da[5:0]))[31:16]`
	`Dn.E = 0`	

6	**ASH.RGT.2L #u5,Da:Db,Dm:Dn**	**SIMD2 arithmetic shift right of Dc into Dm, and Dd into Dn, according to an unsigned immediate.**

	`Dm.M = (Da >> u5)[31:0]`
	`Dm.E = 0`
	`Dn.M = (Db >> u5)[31:0]`
	`Dn.E = 0`	

7	**ASH.RGT.L Da,Db,Dn**	**Arithmetic shift right of Db into Dn, according to a signed field in Da[5:0]. A positive value is a right shift, a negative is a left shift.**
	`Dn = (U40)(((S7)Da[5:0]) > 0) ? Db >> ((S7)Da[5:0]) : Db << - ((S7)Da[5:0])`	
8	**ASH.RGT.L #u5,Da,Dn**	**Arithmetic shift right of Da into Dn, according to an unsigned immediate.**
	`Dn = (U40)Da >> u5`	
9	**ASH.LFT.S.2L Da,Dc:Dd,Dm:Dn**	**SIMD2 arithmetic shift left with saturation of Dc into Dm, and Dd into Dn, according to a signed field in Da[5:0]. A positive value is a saturating left shift, a negative is a right shift.**

	`Dm = (U40)SAT32 ((((S7)Da[5:0]) > 0) ? Dc << ((S7)Da[5:0]) : Dc >> - ((S7)Da[5:0]))`
	`Dn = (U40)SAT32 ((((S7)Da[5:0]) > 0) ? Dd << ((S7)Da[5:0]) : Dd >> - ((S7)Da[5:0]))`	

10	**ASH.LFT.S.2L #u5,Da:Db,Dm:Dn**	**SIMD2 arithmetic shift left with saturation of Dc into Dm, and Dd into Dn, according to an unsigned immediate.**

	`Dm = (U40)SAT32 (Da << u5)`
	`Dn = (U40)SAT32 (Db << u5)`	

11	**ASH.LFT.S.L Da,Db,Dn**	**Arithmetic shift left with saturation of Db into Dn, according to a signed field in Da[5:0]. A positive value is a saturating left shift, a negative is a right shift.**
	`Dn = (U40)SAT32 ((((S7)Da[5:0]) > 0) ? Db << ((S7)Da[5:0]) : Db >> - ((S7)Da[5:0]))`	

Table continues on the next page...

#	Syntax	Description
12	**ASH.LFT.S.L #u5,Da,Dn**	**Arithmetic shift left with saturation of Da into Dn, according to an unsigned immediate.**
	`Dn = (U40)SAT32 (Da << u5)`	
13	**ASH.RGT.S.2L Da,Dc:Dd,Dm:Dn**	**SIMD2 arithmetic shift right with saturation of Dc into Dm, and Dd into Dn, according to a signed field in Da[5:0]. A positive value is a right shift, a negative is a saturating left shift.**

	`Dm = (U40)SAT32 (((S7)Da[5:0]) > 0) ? Dc >> ((S7)Da[5:0]) : Dc << -((S7)Da[5:0]))`
	`Dn = (U40)SAT32 (((S7)Da[5:0]) > 0) ? Dd >> ((S7)Da[5:0]) : Dd << -((S7)Da[5:0]))`	

14	**ASH.RGT.S.L Da,Db,Dn**	**Arithmetic shift right with saturation of Db into Dn, according to a signed field in Da[5:0]. A positive value is a right shift, a negative is a saturating left shift.**
	`Dn = (U40)SAT32 (((S7)Da[5:0]) > 0) ? Db >> ((S7)Da[5:0]) : Db << -((S7)Da[5:0]))`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \leq n \leq 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \leq n \leq 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \leq n \leq 62$
Dn	D0-D63	
u5	$0 \leq u5 < 2^5$	

Affected By Instructions

Field	Relevant Variants
SR.SAT	9, 10, 11, 12, 13, 14

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	2, 4, 6, 8, 10, 12
	dalu_LBM_Y	1, 3, 5, 7, 9, 11, 13, 14

ASH.nT Arithmetic Shift of 20 Bits in Data Registers (DALU)

General Description

Arithmetically shift pairs of packed 20-bit operands taken from the wide-low (WL) and wide-high (WH) portions of D registers. The shift could be to the left or to the right. The shift amount is specified either as an unsigned immediate operand, or taken from a signed field in an additional register. In such a case, a positive value in this field will shift the operand in the direction specified in the mnemonic, and a negative value will shift the operand by the same amount to the other direction. A shift to the right will sign extend the operand up to 20 bits, and discard bits from the right. A left shift will discard bits from the left, and add zeros to the inserted bits to the right of the operand. A right shift by more than 19 will return a value of all zeros or all ones depending on the sign of the source operand. A left shift by more than 19 will return 0. The basic operation shifts the two WH and WL portions that are packed in a register by the same amount. Additional variants shift four 20-bit portions that are packed in two registers. Some variants saturate 20 bits to 16 bits, marked by a stand-alone in flg2 position of the mnemonic. Other variants perform 20 bit saturation, marked by in flg2. 16-bit saturation occurs if either meaningful bits were shifted out in a left shift, or that after the shift (for both directions) the 4-bit extension is not a clean sign extension. The saturated value is 0x00_7FFF or 0xFF_8000. 20-bit saturation occurs for left shifts when meaningful bits are shifted out, saturating the destination to either 0x7F_FFFF or 0x80_0000. In case saturation occurs on any portion shifted by the instruction, SR.SAT is set.

Flag Options

Flag	Position	Description
LFT	flg1	Left
RGT	flg1	Right
S	flg2	Saturated result
S20	flg2	20 bit (T) Saturation

Instruction Variants

#	Syntax	Description
1	**ASH.LFT.2T Da,Db,Dn**	**SIMD2 arithmetic shift left of two packed portions in Db (WH,WL) into respective portions in Dn, according to a signed field in Da[5:0]. A positive value is a left shift, a negative is a right shift.**

```
Dn.WH = (((S6)Da[5:0]) > 0) ? Db.WH << ((S6)Da[5:0]) : Db.WH >> - ((S6)Da[5:0])
Dn.WL = (((S6)Da[5:0]) > 0) ? Db.WL << ((S6)Da[5:0]) : Db.WL >> - ((S6)Da[5:0])
```

Table continues on the next page...

#	Syntax	Description
2	**ASH.LFT.2T #u5,Da,Dn**	**SIMD2 arithmetic shift left of two packed portions in Db (WH,WL) into respective portions in Dn, according to an unsigned immediate.**

	`Dn.WH = Da.WH << u5`
	`Dn.WL = Da.WL << u5`	

3	**ASH.LFT.4T Da,Dc:Dd,Dm:Dn**	**SIMD4 arithmetic shift left of four packed portions in Dc and Dd into the respective portions in Dm and Dn, according to a signed field in Da[5:0]. A positive value is a left shift, a negative is a right shift.**

	`Dm.WH = (((S6)Da[5:0]) > 0) ? Dc.WH << ((S6)Da[5:0]) : Dc.WH >> - ((S6)Da[5:0])`
	`Dm.WL = (((S6)Da[5:0]) > 0) ? Dc.WL << ((S6)Da[5:0]) : Dc.WL >> - ((S6)Da[5:0])`
	`Dn.WH = (((S6)Da[5:0]) > 0) ? Dd.WH << ((S6)Da[5:0]) : Dd.WH >> - ((S6)Da[5:0])`
	`Dn.WL = (((S6)Da[5:0]) > 0) ? Dd.WL << ((S6)Da[5:0]) : Dd.WL >> - ((S6)Da[5:0])`	

4	**ASH.LFT.4T #u5,Da:Db,Dm:Dn**	**SIMD4 arithmetic shift left of four packed portions in Dc and Dd into the respective portions in Dm and Dn, according to an unsigned immediate.**

	`Dm.WH = Da.WH << u5`
	`Dm.WL = Da.WL << u5`
	`Dn.WH = Db.WH << u5`
	`Dn.WL = Db.WL << u5`	

5	**ASH.RGT.2T Da,Db,Dn**	**SIMD2 arithmetic shift right of two packed portions in Db (WH,WL) into respective portions in Dn, according to a signed field in Da[5:0]. A positive value is a right shift, a negative is a left shift.**

	`Dn.WH = (((S6)Da[5:0]) > 0) ? Db.WH >> ((S6)Da[5:0]) : Db.WH << - ((S6)Da[5:0])`
	`Dn.WL = (((S6)Da[5:0]) > 0) ? Db.WL >> ((S6)Da[5:0]) : Db.WL << - ((S6)Da[5:0])`	

6	**ASH.RGT.2T #u5,Da,Dn**	**SIMD2 arithmetic shift right of two packed portions in Db (WH,WL) into respective portions in Dn, according to an unsigned immediate.**

	`Dn.WH = Da.WH >> u5`
	`Dn.WL = Da.WL >> u5`	

7	**ASH.RGT.4T Da,Dc:Dd,Dm:Dn**	**SIMD4 arithmetic shift right of four packed portions in Dc and Dd into the respective portions in Dm and Dn, according to a signed field in Da[5:0]. A positive value is a right shift, a negative is a left shift.**

	`Dm.WH = (((S6)Da[5:0]) > 0) ? Dc.WH >> ((S6)Da[5:0]) : Dc.WH << - ((S6)Da[5:0])`
	`Dm.WL = (((S6)Da[5:0]) > 0) ? Dc.WL >> ((S6)Da[5:0]) : Dc.WL << - ((S6)Da[5:0])`
	`Dn.WH = (((S6)Da[5:0]) > 0) ? Dd.WH >> ((S6)Da[5:0]) : Dd.WH << - ((S6)Da[5:0])`
	`Dn.WL = (((S6)Da[5:0]) > 0) ? Dd.WL >> ((S6)Da[5:0]) : Dd.WL << - ((S6)Da[5:0])`	

Table continues on the next page...

#	Syntax	Description
8	**ASH.RGT.4T #u5,Da:Db,Dm:Dn**	**SIMD4 arithmetic shift right of four packed portions in Dc and Dd into the respective portions in Dm and Dn, according to an unsigned immediate.**

	`Dm.WH = Da.WH >> u5`
	`Dm.WL = Da.WL >> u5`
	`Dn.WH = Db.WH >> u5`
	`Dn.WL = Db.WL >> u5`	

9	**ASH.LFT.S.2T Da,Db,Dn**	**SIMD2 arithmetic shift left with 16-bit saturation of two packed portions in Db (WH,WL) into respective portions in Dn, according to a signed field in Da[5:0]. A positive value is a left shift, a negative is a right shift.**

	`Dn.WH = SAT16 ((((S6)Da[5:0]) > 0) ? Db.WH << ((S6)Da[5:0]) : Db.WH >> - ((S6)Da[5:0]))`
	`Dn.WL = SAT16 ((((S6)Da[5:0]) > 0) ? Db.WL << ((S6)Da[5:0]) : Db.WL >> - ((S6)Da[5:0]))`	

10	**ASH.LFT.S.2T #u5,Da,Dn**	**SIMD2 arithmetic shift left with 16-bit saturation of two packed portions in Db (WH,WL) into respective portions in Dn, according to an unsigned immediate.**

	`Dn.WH = SAT16 (Da.WH << u5)`
	`Dn.WL = SAT16 (Da.WL << u5)`	

11	**ASH.LFT.S.4T Da,Dc:Dd,Dm:Dn**	**SIMD4 arithmetic shift left with 16-bit saturation of four packed portions in Dc and Dd into the respective portions in Dm and Dn, according to a signed field in Da[5:0]. A positive value is a left shift, a negative is a right shift.**

	`Dm.WH = SAT16 ((((S6)Da[5:0]) > 0) ? Dc.WH << ((S6)Da[5:0]) : Dc.WH >> - ((S6)Da[5:0]))`
	`Dm.WL = SAT16 ((((S6)Da[5:0]) > 0) ? Dc.WL << ((S6)Da[5:0]) : Dc.WL >> - ((S6)Da[5:0]))`
	`Dn.WH = SAT16 ((((S6)Da[5:0]) > 0) ? Dd.WH << ((S6)Da[5:0]) : Dd.WH >> - ((S6)Da[5:0]))`
	`Dn.WL = SAT16 ((((S6)Da[5:0]) > 0) ? Dd.WL << ((S6)Da[5:0]) : Dd.WL >> - ((S6)Da[5:0]))`	

12	**ASH.LFT.S.4T #u5,Da:Db,Dm:Dn**	**SIMD4 arithmetic shift left with 16-bit saturation of four packed portions in Dc and Dd into the respective portions in Dm and Dn, according to an unsigned immediate.**

	`Dm.WH = SAT16 (Da.WH << u5)`
	`Dm.WL = SAT16 (Da.WL << u5)`
	`Dn.WH = SAT16 (Db.WH << u5)`
	`Dn.WL = SAT16 (Db.WL << u5)`	

13	**ASH.RGT.S.2T Da,Db,Dn**	**SIMD2 arithmetic shift right with 16-bit saturation of two packed portions in Db (WH,WL) into respective portions in Dn, according to a signed field in Da[5:0]. A positive value is a right shift, a negative is a left shift.**

	`Dn.WH = SAT16 ((((S6)Da[5:0]) > 0) ? Db.WH >> ((S6)Da[5:0]) : Db.WH << - ((S6)Da[5:0]))`
	`Dn.WL = SAT16 ((((S6)Da[5:0]) > 0) ? Db.WL >> ((S6)Da[5:0]) : Db.WL << - ((S6)Da[5:0]))`	

Table continues on the next page...

#	Syntax	Description
14	**ASH.RGT.S.2T #u5,Da,Dn**	**SIMD2 arithmetic shift right with 16-bit saturation of two packed portions in Db (WH,WL) into respective portions in Dn, according to an unsigned immediate.**

	Dn.WH = SAT16 (Da.WH >> u5)
	Dn.WL = SAT16 (Da.WL >> u5)	

15	**ASH.RGT.S.4T Da,Dc:Dd,Dm:Dn**	**SIMD4 arithmetic shift right with 16-bit saturation of four packed portions in Dc and Dd into the respective portions in Dm and Dn, according to a signed field in Da[5:0]. A positive value is a right shift, a negative is a left shift.**

	Dm.WH = SAT16 ((((S6)Da[5:0]) > 0) ? Dc.WH >> ((S6)Da[5:0]) : Dc.WH << - ((S6)Da[5:0]))
	Dm.WL = SAT16 ((((S6)Da[5:0]) > 0) ? Dc.WL >> ((S6)Da[5:0]) : Dc.WL << - ((S6)Da[5:0]))
	Dn.WH = SAT16 ((((S6)Da[5:0]) > 0) ? Dd.WH >> ((S6)Da[5:0]) : Dd.WH << - ((S6)Da[5:0]))
	Dn.WL = SAT16 ((((S6)Da[5:0]) > 0) ? Dd.WL >> ((S6)Da[5:0]) : Dd.WL << - ((S6)Da[5:0]))	

16	**ASH.RGT.S.4T #u5,Da:Db,Dm:Dn**	**SIMD4 arithmetic shift right with 16-bit saturation of four packed portions in Dc and Dd into the respective portions in Dm and Dn, according to an unsigned immediate.**

	Dm.WH = SAT16 (Da.WH >> (U6)u5)
	Dm.WL = SAT16 (Da.WL >> (U6)u5)
	Dn.WH = SAT16 (Db.WH >> (U6)u5)
	Dn.WL = SAT16 (Db.WL >> (U6)u5)	

17	**ASH.LFT.S20.2T Da,Db,Dn**	**SIMD2 arithmetic shift left with 20-bit saturation of two packed portions in Db (WH,WL) into respective portions in Dn, according to a signed field in Da[5:0]. A positive value is a left shift, a negative is a right shift.**

	Dn.WH = SAT20 ((((S6)Da[5:0]) > 0) ? Db.WH << ((S6)Da[5:0]) : Db.WH >> - ((S6)Da[5:0]))
	Dn.WL = SAT20 ((((S6)Da[5:0]) > 0) ? Db.WL << ((S6)Da[5:0]) : Db.WL >> - ((S6)Da[5:0]))	

18	**ASH.LFT.S20.2T #u5,Da,Dn**	**SIMD2 arithmetic shift left with 20-bit saturation of two packed portions in Db (WH,WL) into respective portions in Dn, according to an unsigned immediate.**

	Dn.WH = SAT20 (Da.WH << u5)
	Dn.WL = SAT20 (Da.WL << u5)	

19	**ASH.LFT.S20.4T Da,Dc:Dd,Dm:Dn**	**SIMD4 arithmetic shift left with 20-bit saturation of four packed portions in Dc and Dd into the respective portions in Dm and Dn, according to a signed field in Da[5:0]. A positive value is a left shift, a negative is a right shift.**

	Dm.WH = SAT20 ((((S6)Da[5:0]) > 0) ? Dc.WH << ((S6)Da[5:0]) : Dc.WH >> - ((S6)Da[5:0]))
	Dm.WL = SAT20 ((((S6)Da[5:0]) > 0) ? Dc.WL << ((S6)Da[5:0]) : Dc.WL >> - ((S6)Da[5:0]))
	Dn.WH = SAT20 ((((S6)Da[5:0]) > 0) ? Dd.WH << ((S6)Da[5:0]) : Dd.WH >> - ((S6)Da[5:0]))
	Dn.WL = SAT20 ((((S6)Da[5:0]) > 0) ? Dd.WL << ((S6)Da[5:0]) : Dd.WL >> - ((S6)Da[5:0]))	

Table continues on the next page...

#	Syntax	Description
20	**ASH.LFT.S20.4T #u5,Da:Db,Dm:Dn**	**SIMD4 arithmetic shift left with 20-bit saturation of four packed portions in Dc and Dd into the respective portions in Dm and Dn, according to an unsigned immediate.**

	`Dm.WH = SAT20 (Da.WH << u5)`
	`Dm.WL = SAT20 (Da.WL << u5)`
	`Dn.WH = SAT20 (Db.WH << u5)`
	`Dn.WL = SAT20 (Db.WL << u5)`	

21	**ASH.RGT.S20.2T Da,Db,Dn**	**SIMD2 arithmetic shift right with 20-bit saturation of two packed portions in Db (WH,WL) into respective portions in Dn, according to a signed field in Da[5:0]. A positive value is a right shift, a negative is a left shift.**

	`Dn.WH = SAT20 ((((S6)Da[5:0]) > 0) ? Db.WH >> ((S6)Da[5:0]) : Db.WH << - ((S6)Da[5:0]))`
	`Dn.WL = SAT20 ((((S6)Da[5:0]) > 0) ? Db.WL >> ((S6)Da[5:0]) : Db.WL << - ((S6)Da[5:0]))`	

22	**ASH.RGT.S20.4T Da,Dc:Dd,Dm:Dn**	**SIMD4 arithmetic shift right with 20-bit saturation of four packed portions in Dc and Dd into the respective portions in Dm and Dn, according to a signed field in Da[5:0]. A positive value is a right shift, a negative is a left shift.**

	`Dm.WH = SAT20 ((((S6)Da[5:0]) > 0) ? Dc.WH >> ((S6)Da[5:0]) : Dc.WH << - ((S6)Da[5:0]))`
	`Dm.WL = SAT20 ((((S6)Da[5:0]) > 0) ? Dc.WL >> ((S6)Da[5:0]) : Dc.WL << - ((S6)Da[5:0]))`
	`Dn.WH = SAT20 ((((S6)Da[5:0]) > 0) ? Dd.WH >> ((S6)Da[5:0]) : Dd.WH << - ((S6)Da[5:0]))`
	`Dn.WL = SAT20 ((((S6)Da[5:0]) > 0) ? Dd.WL >> ((S6)Da[5:0]) : Dd.WL << - ((S6)Da[5:0]))`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	
u5	$0 \le u5 < 2^5$	

Affected By Instructions

Field	Relevant Variants
SR.SAT	9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	2, 4, 6, 8, 10, 12, 14, 16, 18, 20
	dalu_LBM_Y	1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 22

ASH.nW Arithmetic Shift of 16 Bits in Data Registers (DALU)

General Description

Arithmetically shift pairs of packed 16-bit operands taken from the high (H) and low (L) portions of D register. The shift could be to the left or to the right. The shift amount is specified either as an unsigned immediate operand, or taken from a signed field in an additional register. In such a case, a positive value in this field will shift the operand in the direction specified in the mnemonic, and a negative value will shift the operand by the same amount to the other direction. A shift to the right will sign extend the operand up to 16 bits, and discard bits from the right. A left shift will discard bits from the left, and add zeros to the inserted bits to the right of the operand. A right shift by more than 15 will return a value of all zeros or all ones depending on the sign of the source operand. A left shift by more than 15 will return 0. The basic operation shifts the two H and L portions that are packed in a register by the same amount. Additional variants shift four 16-bit portions that are packed in two registers. Some variants saturate the result to 16 bits when meaningful bits are shifted out. The saturated value is 0x7FFF or 0x8000. In case saturation occurs on any portion shifted by the instruction, SR.SAT is set.

A few variants preserve the legacy behavior of performing a 20-bit shift on WH and WL instead of a 16-bit shift if SR.W20 is set, and performing 16-bit saturation if SR.SM2 is set. In general these SR bits should not be set in new code, and instructions that explicitly do a 20-bit shift or 16-bit saturation should be used instead.

Flag Options

Flag	Position	Description
LFT	flg1	Left
RGT	flg1	Right
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**ASH.LFT.2W Da,Db,Dn**	**If SR.W20 is clear (by default), perform dual 16-bit shift on Db.H and Db.L. If SR.W20 is set, perform dual 20-bit shift on Db.WH and Db.WL. If SR.SM2 is set, perform 16-bit saturation on the result. The shift size is taken from a signed field in Da. Negative shift sizes perform a right shift.**

```
If (SR.W20)
  Dn.WL = (Da[5:0] > 0) ? (S20) Db.WL << Da[5:0] : (S20) Db.WL >> |Da[5:0]|
  Dn.WH = (Da[5:0] > 0) ? (S20) Db.WH << Da[5:0] : (S20) Db.WH >> |Da[5:0]|
else
  Dn.L = (Da[4:0] > 0) ? (S16) Db.L << Da[4:0] : (S16) Db.L >> |Da[4:0]|
  Dn.H = (Da[4:0] > 0) ? (S16) Db.H << Da[4:0] : (S16) Db.H >> |Da[4:0]|
  Dn.E = 0
```

Table continues on the next page...

#	Syntax	Description
2	**ASH.LFT.2W #u5,Da,Dn**	**If SR.W20 is clear (by default), perform dual 16-bit shift on Db.H and Db.L. If SR.W20 is set, perform dual 20-bit shift on Db.WH and Db.WL. If SR.SM2 is set, perform 16-bit saturation on the result. The shift size is taken from an immediate field.**

	`If (SR.W20)`
	`  Dn.WL = (S20) Da.WL << u5`
	`  Dn.WH = (S20) Da.WH << u5`
	`else`
	`  Dn.L = (S16) Da.L << u5[3:0]`
	`  Dn.H = (S16) Da.H << u5[3:0]`
	`  Dn.E = 0`	

3	**ASH.LFT.4W Da,Dc:Dd,Dm:Dn**	**SIMD4 arithmetic shift left of four packed words in Dc and Dd into the respective portions in Dm and Dn, according to a signed field in Da[5:0]. A positive value is a left shift, a negative is a right shift.**

	`Dm.H = (((S6)Da[4:0]) > 0) ? Dc.H << ((S6)Da[4:0]) : Dc.H >> - ((S6)Da[4:0])`
	`Dm.L = (((S6)Da[4:0]) > 0) ? Dc.L << ((S6)Da[4:0]) : Dc.L >> - ((S6)Da[4:0])`
	`Dm.E = 0`
	`Dn.H = (((S6)Da[4:0]) > 0) ? Dd.H << ((S6)Da[4:0]) : Dd.H >> - ((S6)Da[4:0])`
	`Dn.L = (((S6)Da[4:0]) > 0) ? Dd.L << ((S6)Da[4:0]) : Dd.L >> - ((S6)Da[4:0])`
	`Dn.E = 0`	

4	**ASH.LFT.4W #u4,Da:Db,Dm:Dn**	**SIMD4 arithmetic shift left of four packed words in Dc and Dd into the respective portions in Dm and Dn, according to an unsigned immediate.**

	`Dm.H = Da.H << u4`
	`Dm.L = Da.L << u4`
	`Dm.E = 0`
	`Dn.H = Db.H << u4`
	`Dn.L = Db.L << u4`
	`Dn.E = 0`	

5	**ASH.RGT.2W Da,Db,Dn**	**If SR.W20 is clear (by default), perform dual 16-bit shift on Db.H and Db.L. If SR.W20 is set, perform dual 20-bit shift on Db.WH and Db.WL. If SR.SM2 is set, perform 16-bit saturation on the result. The shift size is taken from a signed field in Da. Negative shift sizes perform a left shift.**

	`If (SR.W20)`
	`  Dn.WL = (Da[5:0] > 0) ? (S20) Db.WL >> Da[5:0] : (S20) Db.WL << |Da[5:0]|`
	`  Dn.WH = (Da[5:0] > 0) ? (S20) Db.WH >> Da[5:0] : (S20) Db.WH << |Da[5:0]|`
	`else`
	`  Dn.L = (Da[4:0] > 0) ? (S16) Db.L >> Da[4:0] : (S16) Db.L << |Da[4:0]|`
	`  Dn.H = (Da[4:0] > 0) ? (S16) Db.H >> Da[4:0] : (S16) Db.H << |Da[4:0]|`
	`  Dn.E = 0`	

Table continues on the next page...

#	Syntax	Description
6	**ASH.RGT.2W #u5,Da,Dn**	**If SR.W20 is clear (by default), perform dual 16-bit shift on Db.H and Db.L. If SR.W20 is set, perform dual 20-bit shift on Db.WH and Db.WL. The shift size is taken from an immediate field.**

	`If (SR.W20)`
	`  Dn.WL = (S20) Da.WL >> u5`
	`  Dn.WH = (S20) Da.WH >> u5`
	`else`
	`  Dn.L = (S16) Da.L >> u5[3:0]`
	`  Dn.H = (S16) Da.H >> u5[3:0]`
	`  Dn.E = 0`	

7	**ASH.RGT.4W Da,Dc:Dd,Dm:Dn**	**SIMD4 arithmetic shift right of four packed words in Dc and Dd into the respective portions in Dm and Dn, according to a signed field in Da[5:0]. A positive value is a right shift, a negative is a left shift.**

	`Dm.H = (((S6)Da[4:0]) > 0) ? Dc.H >> ((S6)Da[4:0]) : Dc.H << - ((S6)Da[4:0])`
	`Dm.L = (((S6)Da[4:0]) > 0) ? Dc.L >> ((S6)Da[4:0]) : Dc.L << - ((S6)Da[4:0])`
	`Dm.E = 0`
	`Dn.H = (((S6)Da[4:0]) > 0) ? Dd.H >> ((S6)Da[4:0]) : Dd.H << - ((S6)Da[4:0])`
	`Dn.L = (((S6)Da[4:0]) > 0) ? Dd.L >> ((S6)Da[4:0]) : Dd.L << - ((S6)Da[4:0])`
	`Dn.E = 0`	

8	**ASH.RGT.4W #u4,Da:Db,Dm:Dn**	**SIMD4 arithmetic shift right of four packed words in Dc and Dd into the respective portions in Dm and Dn, according to an unsigned immediate.**

	`Dm.H = Da.H >> u4`
	`Dm.L = Da.L >> u4`
	`Dm.E = 0`
	`Dn.H = Db.H >> u4`
	`Dn.L = Db.L >> u4`
	`Dn.E = 0`	

9	**ASH.LFT.S.2W Da,Db,Dn**	**SIMD2 arithmetic shift left with 16-bit saturation of two packed words in Db into the respective portions in Dn, according to a signed field in Da[5:0]. A positive value is a left shift, a negative is a right shift.**

	`Dn.H = SAT16 ((((S6)Da[4:0]) > 0) ? Db.H << ((S6)Da[4:0]) : Db.H >> - ((S6)Da[4:0]))`
	`Dn.L = SAT16 ((((S6)Da[4:0]) > 0) ? Db.L << ((S6)Da[4:0]) : Db.L >> - ((S6)Da[4:0]))`
	`Dn.E = 0`	

10	**ASH.LFT.S.2W #u5,Da,Dn**	**SIMD2 arithmetic shift left with 16-bit saturation of two packed words in Da into the respective portions in Dn, according to an unsigned immediate.**

	`Dn.H = SAT16 (Da.H << (U6)u5[3:0])`
	`Dn.L = SAT16 (Da.L << (U6)u5[3:0])`
	`Dn.E = 0`	

11	**ASH.LFT.S.4W Da,Dc:Dd,Dm:Dn**	**SIMD4 arithmetic shift left with 16-bit saturation of four packed words in Dc and Dd into the respective portions in Dm and Dn, according to a signed field in Da[5:0]. A positive value is a left shift, a negative is a right shift.**

Table continues on the next page...

#	Syntax	Description

	Dm.H = SAT16 ((((S6)Da[4:0]) > 0) ? Dc.H << ((S6)Da[4:0]) : Dc.H >> - ((S6)Da[4:0]))
	Dm.L = SAT16 ((((S6)Da[4:0]) > 0) ? Dc.L << ((S6)Da[4:0]) : Dc.L >> - ((S6)Da[4:0]))
	Dm.E = 0
	Dn.H = SAT16 ((((S6)Da[4:0]) > 0) ? Dd.H << ((S6)Da[4:0]) : Dd.H >> - ((S6)Da[4:0]))
	Dn.L = SAT16 ((((S6)Da[4:0]) > 0) ? Dd.L << ((S6)Da[4:0]) : Dd.L >> - ((S6)Da[4:0]))
	Dn.E = 0	

12	**ASH.LFT.S.4W #u4,Da:Db,Dm:Dn**	**SIMD4 arithmetic shift left with 16-bit saturation of four packed words in Dc and Dd into the respective portions in Dm and Dn, according to an unsigned immediate.**

	Dm.H = SAT16 (Da.H << u4)
	Dm.L = SAT16 (Da.L << u4)
	Dm.E = 0
	Dn.H = SAT16 (Db.H << u4)
	Dn.L = SAT16 (Db.L << u4)
	Dn.E = 0	

13	**ASH.RGT.S.2W Da,Db,Dn**	**SIMD2 arithmetic shift right with 16-bit saturation of two packed words in Db into the respective portions in Dn, according to a signed field in Da[5:0]. A positive value is a right shift, a negative is a saturating left shift.**

	Dn.H = SAT16 ((((S6)Da[4:0]) > 0) ? Db.H >> ((S6)Da[4:0]) : Db.H << - ((S6)Da[4:0]))
	Dn.L = SAT16 ((((S6)Da[4:0]) > 0) ? Db.L >> ((S6)Da[4:0]) : Db.L << - ((S6)Da[4:0]))
	Dn.E = 0	

14	**ASH.RGT.S.4W Da,Dc:Dd,Dm:Dn**	**SIMD4 arithmetic shift right with 16-bit saturation of four packed words in Dc and Dd into the respective portions in Dm and Dn, according to a signed field in Da[5:0]. A positive value is a right shift, a negative is a saturating left shift.**

	Dm.H = SAT16 ((((S6)Da[4:0]) > 0) ? Dc.H >> ((S6)Da[4:0]) : Dc.H << - ((S6)Da[4:0]))
	Dm.L = SAT16 ((((S6)Da[4:0]) > 0) ? Dc.L >> ((S6)Da[4:0]) : Dc.L << - ((S6)Da[4:0]))
	Dm.E = 0
	Dn.H = SAT16 ((((S6)Da[4:0]) > 0) ? Dd.H >> ((S6)Da[4:0]) : Dd.H << - ((S6)Da[4:0]))
	Dn.L = SAT16 ((((S6)Da[4:0]) > 0) ? Dd.L >> ((S6)Da[4:0]) : Dd.L << - ((S6)Da[4:0]))
	Dn.E = 0	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	
u4	$0 \le u4 < 2^4$	
u5	$0 \le u5 < 2^5$	

Affect Instructions

Field	Relevant Variants	Comments
SR.SM2	1, 2, 5	
SR.W20	1, 2, 5, 6	

Affected By Instructions

Field	Relevant Variants
SR.SAT	1, 2, 5, 9, 10, 11, 12, 13, 14

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	2, 4, 6, 8, 10, 12
	dalu_LBM_Y	1, 3, 5, 7, 9, 11, 13, 14

ASH.nX Arithmetic Shift of 40 Bits in Data Registers (DALU)

General Description

Arithmetically shift 40-bit operands from D registers. The shift could be to the left or to the right. The shift amount is specified either as an unsigned immediate operand, or taken from a signed field in an additional register. In such a case, a positive value in this field will shift the operand in the direction specified in the mnemonic, and a negative value will shift the operand by the same amount to the other direction. A shift to the right will sign extend the operand up to 40 bits, and discard bits from the right. A left shift will discard bits from the left, and add zeros to the inserted bits to the right of the operand. A right shift by more than 39 will return a value of all zeros or all ones depending on the sign of the source operand. A left shift by more than 39 will return 0. SIMD variants shift 2 registers by the same amount, although one left shift variant allows to specify two independent shift sizes. Some variants, marked with in FLG2, explicitly saturate the result to 32 bits. Saturation occurs either if meaningful bits were shifted out in a left shift, or if the extension after the shift (in both directions) if not a clean sign extension of bit 31. The saturated value is 0x00_7FFF_FFFF or 0xFF_8000_0000. Legacy variants, marked with LEG in FLG2, saturate the result to 32 bits if SR.SM is set. In case saturation occurs, SR.SAT is set.

Flag Options

Flag	Position	Description
LFT	flg1	Left
RGT	flg1	Right
LEG	flg2	Legacy instruction
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**ASH.LFT.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 arithmetic shift left of Dc into Dm by an amount specified in Da[6:0], and Dd into Dn by an amount independently specified in Db[6:0]. A positive shift size is a left shift, a negative is a right shift.**

	`Dm = (Da[6:0] > 0) ? Dc << Da[6:0] : Dc >> - Da[6:0]`
	`Dn = (Db[6:0] > 0) ? Dd << Db[6:0] : Dd >> - Db[6:0]`	

2	**ASH.LFT.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 arithmetic shift left of Dc into Dm, and Dd into Dn, according to a signed field in Da[6:0], affecting both shifts. A positive value is a left shift, a negative is a right shift.**

	`Dm = (Da[6:0] > 0) ? Dc << Da[6:0] : Dc >> - Da[6:0]`
	`Dn = (Da[6:0] > 0) ? Dd << Da[6:0] : Dd >> - Da[6:0]`	

3	**ASH.LFT.2X #ue5,Da:Db,Dm:Dn**	**SIMD2 arithmetic shift left of Dc into Dm, and Dd into Dn, according to an unsigned immediate, affecting both shifts.**

Table continues on the next page...

#	Syntax	Description

	`Dm = Da << ue5`
	`Dn = Db << ue5`	

4	**ASH.LFT.X Da,Db,Dn**	**Arithmetic shift left of Db into Dn, according to a signed field in Da[6:0]. A positive value is a left shift, a negative is a right shift.**
	`Dn = (Da[6:0] > 0) ? Db << Da[6:0] : Db >> - Da[6:0]`	
5	**ASH.LFT.X #ue5,Da,Dn**	**Arithmetic shift left of Da into Dn, according to an unsigned immediate.**
	`Dn = Da << ue5`	
6	**ASH.RGT.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 arithmetic shift right of Dc into Dm, and Dd into Dn, according to a signed field in Da[6:0], affecting both shifts. A positive value is a right shift, a negative is a left shift.**

	`Dm = (Da[6:0] > 0) ? Dc >> Da[6:0] : Dc << - Da[6:0]`
	`Dn = (Da[6:0] > 0) ? Dd >> Da[6:0] : Dd << - Da[6:0]`	

7	**ASH.RGT.2X #ue5,Da:Db,Dm:Dn**	**SIMD2 arithmetic shift right of Dc into Dm, and Dd into Dn, according to an unsigned immediate, affecting both shifts.**

	`Dm = Da >> ue5`
	`Dn = Db >> ue5`	

8	**ASH.RGT.X Da,Db,Dn**	**Arithmetic shift right of Db into Dn, according to a signed field in Da[6:0]. A positive value is a right shift, a negative is a left shift.**
	`Dn = (Da[6:0] > 0) ? Db >> Da[6:0] : Db << - Da[6:0]`	
9	**ASH.RGT.X #ue5,Da,Dn**	**Arithmetic shift right of Da into Dn, according to an unsigned immediate.**
	`Dn = Da >> ue5`	
10	**ASH.LFT.LEG.X Da,Db,Dn**	**Arithmetic shift left of Db into Dn, optionally saturating to 32 bits if SR.SM is set. Shift size is according to a signed field in Da[6:0]. A positive value is a left shift, a negative is a right shift.**
	`Dn = (S40) srSAT32((Da[6:0] > 0) ? (S40) Db << Da[6:0] : (S40) Db >> |Da[6:0]|)`	
11	**ASH.LFT.LEG.X #u5,Da,Dn**	**Arithmetic shift left of Db into Dn, optionally saturating to 32 bits if SR.SM is set. Shift size is according to an unsigned immediate.**
	`Dn = srSAT32 (Da << (U7)u5)`	
12	**ASH.RGT.LEG.X Da,Db,Dn**	**Arithmetic shift right of Db into Dn, optionally saturating to 32 bits if SR.SM is set. Shift size is according to a signed field in Da[6:0]. A positive value is a right shift, a negative is a left shift.**
	`Dn = (S40) srSAT32((Da[6:0] > 0) ? (S40) Db >> |Da[6:0]| : (S40) Db << |Da[6:0]|)`	
13	**ASH.RGT.LEG.X #u5,Da,Dn**	**Arithmetic shift right of Db into Dn, optionally saturating to 32 bits if SR.SM is set. Shift size is according to an unsigned immediate.**

Table continues on the next page...

#	Syntax	Description
	`Dn = srSAT32 (Da >> u5)`	
14	**ASH.LFT.S.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 arithmetic shift left with 32-bit saturation of Dc into Dm, and Dd into Dn, according to a signed field in Da[6:0], affecting both shifts. A positive value is a left shift, a negative is a right shift.**

	`Dm = SAT32 ((Da[6:0] > 0) ? Dc << Da[6:0] : Dc >> - Da[6:0])`
	`Dn = SAT32 ((Da[6:0] > 0) ? Dd << Da[6:0] : Dd >> - Da[6:0])`	

15	**ASH.LFT.S.2X #ue5,Da:Db,Dm:Dn**	**SIMD2 arithmetic shift left with 32-bit saturation of Dc into Dm, and Dd into Dn, according to an unsigned immediate, affecting both shifts.**

	`Dm = SAT32 (Da << ue5)`
	`Dn = SAT32 (Db << ue5)`	

16	**ASH.LFT.S.X Da,Db,Dn**	**Arithmetic shift left with 32-bit saturation of Db into Dn, according to a signed field in Da[6:0]. A positive value is a left shift, a negative is a right shift.**
	`Dn = SAT32 ((Da[6:0] > 0) ? Db << Da[6:0] : Db >> - Da[6:0])`	
17	**ASH.LFT.S.X #ue5,Da,Dn**	**Arithmetic shift left with 32-bit saturation of Da into Dn, according to an unsigned immediate.**
	`Dn = SAT32 (Da << ue5)`	
18	**ASH.RGT.S.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 arithmetic shift right with 32-bit saturation of Dc into Dm, and Dd into Dn, according to a signed field in Da[6:0], affecting both shifts. A positive value is a right shift, a negative is a left shift.**

	`Dm = SAT32 ((Da[6:0] > 0) ? Dc >> Da[6:0] : Dc << - Da[6:0])`
	`Dn = SAT32 ((Da[6:0] > 0) ? Dd >> Da[6:0] : Dd << - Da[6:0])`	

19	**ASH.RGT.S.2X #ue5,Da:Db,Dm:Dn**	**SIMD2 arithmetic shift right with 32-bit saturation of Dc into Dm, and Dd into Dn, according to an unsigned immediate, affecting both shifts.**

	`Dm = SAT32 (Da >> ue5)`
	`Dn = SAT32 (Db >> ue5)`	

20	**ASH.RGT.S.X Da,Db,Dn**	**Arithmetic shift right with 32-bit saturation of Db into Dn, according to a signed field in Da[6:0]. A positive value is a right shift, a negative is a left shift.**
	`Dn = SAT32 ((Da[6:0] > 0) ? Db >> Da[6:0] : Db << - Da[6:0])`	
21	**ASH.RGT.S.X #ue5,Da,Dn**	**Arithmetic shift right with 32-bit saturation of Da into Dn, according to an unsigned immediate.**
	`Dn = SAT32 (Da >> ue5)`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \leq n \leq 62$
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \leq n \leq 62$
Dm:Dn	D_n:D_{n+1}	$0 \leq n \leq 62$
Dn	D0-D63	
u5	$0 \leq u5 < 2^5$	
ue5	$1 \leq ue5 \leq 32$	

Affect Instructions

Field	Relevant Variants	Comments
SR.SM	10, 11, 12, 13	

Affected By Instructions

Field	Relevant Variants
SR.SAT	10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 13, 15, 17, 19, 21
	dalu_LBM_Y	10, 12, 14, 16, 18, 20

ASHA Arithmetic Shift of 32 Bits in Address Registers (IPU,LSU)

General Description

Arithmetically shift a 32-bit operand taken from an R register. The shift could be to the left or to the right. The shift amount is specified either as an unsigned immediate operand, or taken from a signed field in an additional register. In such a case, a positive value in this field will shift the operand in the direction specified in the mnemonic, and a negative value will shift the operand by the same amount to the other direction. A shift to the right will sign extend the operand up to bit 32, and discard bits from the right. A left shift will discard bits from the left, and add zeros to the inserted bits to the right of the operand. A right shift by 32 will return a value of all zeros or all ones depending on the sign of the source operand.

Flag Options

Flag	Position	Description
LFT	flg1	Left
RGT	flg1	Right

Instruction Variants

#	Syntax	Description
1	**ASHA.LFT Ra,Rb,Rn**	**Arithmetic shift left of Rb into Rn, according to a signed field in Ra[5:0]. A positive value is a left shift, a negative is a right shift.**
	`Rn = (Ra[5:0] > 0) ? Rb << Ra[5:0] : Rb >> - Ra[5:0]`	
2	**ASHA.LFT #u5,Ra,Rn**	**Arithmetic shift left of Ra into Rn, according to an unsigned immediate.**
	`Rn = Ra << u5`	
3	**ASHA.RGT Ra,Rb,Rn**	**Arithmetic shift right of Rb into Rn, according to a signed field in Ra[5:0]. A positive value is a right shift, a negative is a left shift.**
	`Rn = (Ra[5:0] > 0) ? Rb >> Ra[5:0] : Rb << - Ra[5:0]`	
4	**ASHA.RGT #u5,Ra,Rn**	**Arithmetic shift right of Ra into Rn, according to an unsigned immediate.**
	`Rn = Ra >> u5`	

Explicit Operands

Operand	Permitted Values
Ra	R0-R31
Rb	R0-R31
Rn	R0-R31
u5	$0 \le u5 < 2^5$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Execution unit	IPU	All
	LSU	2, 4
Pipeline behavior	agu_AAU_LOGIC	All

AVG.4B Average 4 bytes (DALU)

General Description

Performs four average operations. Each average operation uses corresponding bytes from two source data registers and writes the result to the corresponding byte in the destination data register. The average operation consists of unsigned addition and rounding. Rounding is performed by adding 1 to the sum and discarding the least significant bit of the 9-bit result. The extension byte of the destination register is cleared.

Flag Options

Flag	Position	Description
U	flg2	Unsigned

Instruction Variants

#	Syntax	Description
1	**AVG.U.4B Da,Db,Dn**	**Four Byte Averages**

```
Dn.LL = (((U9)Db.LL + (U9)Da.LL + 1) >> 1)
Dn.LH = (((U9)Db.LH + (U9)Da.LH + 1) >> 1)
Dn.HL = (((U9)Db.HL + (U9)Da.HL + 1) >> 1)
Dn.HH = (((U9)Db.HH + (U9)Da.HH + 1) >> 1)
Dn.E = 0
```

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Db	D0-D63
Dn	D0-D63

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

BCCAS Block Cache Command Assist (LSU)

General Description

This instruction is a mandatory supplement of a block cache instruction. A block cache operation is a configuration operation of a CME (Cache Management Engine) task, and is encoded by 3 instructions that must appear together: the block instruction such as DFETCHB, PUNLOCKB and others, with BCCAS and SYNC.B. The block cache instruction must be positioned so that it is using LSU0, and the BCCAS must use LSU1. During its operation, BCASS drives the value in its operand Ra on the bus, with attributes that mark it as part of a CME configuration transaction. Some programming rules apply when using this instruction, refer to rule A.9 in the Programming Rules chapter. For more information on block cache operations and the required structure of the data operands, refer to the Address Generation chapter.

Instruction Variants

#	Syntax	Operation	Description
1	BCCAS Ra		Block cache command assist

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Helper instruction		All
No predication		All
Pipeline behavior	agu_MOVE_LD	All

BIT.COLPSH Pack the Most Significant Bits of Register Portions (DALU)

General Description

This group of instructions packs the MS (most significant) bits from bytes or words of the source registers into the specified portion of the destination register.

Most variants select eight MS bits of the source bytes into the specified byte in the destination register Dn, not changing the other bytes of Dn.M. The selected bits may be packed in matching order as they were in the source registers, or in reverse order.

A few variants select the MS bits from words rather than bytes in the source registers, in which case they are appended to the right, shifting the previous content of the destination register to the left.
The extension is cleared by all instruction variants.

Flag Options

Flag	Position	Description
COLPSH	flg1	Collapse High
REV	flg2	Reverse bit order
SH	flg2	Shift

Instruction Variants

#	Syntax	Description
1	**BIT.COLPSH.B Da:Db,Dn.hh**	**Pack MS bits of bytes from two registers into byte HH of the destination.**

	`Dn.HH = {Da[31], Da[23], Da[15], Da[7], Db[31], Db[23], Db[15], Db[7]}`
	`Dn[23:0] = Dn[23:0]`
	`Dn.E = 0`	

2	**BIT.COLPSH.B Da:Db,Dn.hl**	**Pack MS bits of bytes from two registers into byte HL of the destination.**

	`Dn.HL = {Da[31], Da[23], Da[15], Da[7], Db[31], Db[23], Db[15], Db[7]}`
	`Dn[15:0] = Dn[15:0]`
	`Dn.HH = Dn.HH`
	`Dn.E = 0`	

3	**BIT.COLPSH.B Da:Db,Dn.lh**	**Pack MS bits of bytes from two registers into byte LH of the destination.**

	`Dn.LH = {Da[31], Da[23], Da[15], Da[7], Db[31], Db[23], Db[15], Db[7]}`
	`Dn.LL = Dn.LL`
	`Dn[31:16] = Dn[31:16]`
	`Dn.E = 0`	

Table continues on the next page...

#	Syntax	Description
4	**BIT.COLPSH.B Da:Db,Dn.ll**	**Pack MS bits of bytes from two registers into byte LL of the destination.**

	Dn.LL = {Da[31], Da[23], Da[15], Da[7], Db[31], Db[23], Db[15], Db[7]}
	Dn[31:8] = Dn[31:8]
	Dn.E = 0	

5	**BIT.COLPSH.REV.B Da:Db,Dn.hh**	**Pack MS bits of bytes from two registers into byte HH of the destination, in reversed order.**

	Dn.HH = {Da[7], Da[15], Da[23], Da[31], Db[7], Db[15], Db[23], Db[31]}
	Dn[23:0] = Dn[23:0]
	Dn.E = 0	

6	**BIT.COLPSH.REV.B Da:Db,Dn.hl**	**Pack MS bits of bytes from two registers into byte HL of the destination, in reversed order.**

	Dn.HL = {Da[7], Da[15], Da[23], Da[31], Db[7], Db[15], Db[23], Db[31]}
	Dn[15:0] = Dn[15:0]
	Dn.HH = Dn.HH
	Dn.E = 0	

7	**BIT.COLPSH.REV.B Da:Db,Dn.lh**	**Pack MS bits of bytes from two registers into byte LH of the destination, in reversed order.**

	Dn.LH = {Da[7], Da[15], Da[23], Da[31], Db[7], Db[15], Db[23], Db[31]}
	Dn.LL = Dn.LL
	Dn[31:16] = Dn[31:16]
	Dn.E = 0	

8	**BIT.COLPSH.REV.B Da:Db,Dn.ll**	**Pack MS bits of bytes from two registers into byte LL of the destination, in reversed order.**

	Dn.LL = {Da[7], Da[15], Da[23], Da[31], Db[7], Db[15], Db[23], Db[31]}
	Dn[31:8] = Dn[31:8]
	Dn.E = 0	

9	**BIT.COLPSH.SH.W Da:Db,Dn**	**Shift the destination register left by 4 bits, and pack four MS bits of words from two registers into the least significant bits of the destination.**

	Dn[3:0] = {Da[31], Da[15], Db[31], Db[15]}
	Dn[31:4] = Dn[27:0]
	Dn.E = 0	

10	**BIT.COLPSH.SH.W Da,Dn**	**Shift the destination register left by 2 bits, and pack two MS bits of words from one register into the least significant bits of the destination.**

	Dn[1:0] = {Da[31], Da[15]}
	Dn[31:2] = Dn[29:0]
	Dn.E = 0	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	0 ≤ n ≤ 62
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

BIT.COLPSL — Pack the Least Significant Bits of Register Portions — (DALU)

General Description

The instruction selects eight least significant (LS) bits of the source bytes into the specified byte in the destination register Dn, not changing the other bytes of Dn.M. The selected bits may be packed in matching order as they were in the source registers, or in reverse order. The extension is cleared.

Flag Options

Flag	Position	Description
COLPSL	flg1	Collapse Low
REV	flg2	Reverse bit order

Instruction Variants

#	Syntax	Description
1	**BIT.COLPSL.B Da:Db,Dn.ll**	**Pack LS bits of bytes from two registers into byte LL of the destination.**

	`Dn.LL = {Da[24], Da[16], Da[8], Da[0], Db[24], Db[16], Db[8], Db[0]}`
	`Dn[31:8] = Dn[31:8]`
	`Dn.E = 0`	

2	**BIT.COLPSL.REV.B Da:Db,Dn.ll**	**Pack LS bits of bytes from two registers into byte LL of the destination, in reversed order.**

	`Dn.LL = {Da[0], Da[8], Da[16], Da[24], Db[0], Db[8], Db[16], Db[24]}`
	`Dn[31:8] = Dn[31:8]`
	`Dn.E = 0`	

Explicit Operands

Operand	Permitted Values
Da:Db	D_n:D_{n+1} $0 \le n \le 62$
Dn	D0-D63

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

BIT.DINTLV Two-Way or Three-Way Bit De-Interleave (DALU)

General Description

De-interleave bits from the source register into two or three portions in the destination register.
Two-way de-interleave splits 32 bits into two 16-bit portions; Three-way de-interleave splits 24 bits into 3 bytes. Unspecified portions of the destination (such as the extension) are cleared. SIMD2 variants perform this operation in parallel for two input and output registers.

Flag Options

Flag	Position	Description
2BI	flg1	Two Buffer Interleave Data
3BI	flg1	Three Buffer Interleave Data
DINTLV	flg1	De-Interleave
REV	flg2	Reverse bit order

Instruction Variants

#	Syntax	Description
1	**BIT.DINTLV2BI.2L Da:Db,Dm:Dn**	**SIMD2 2-way de-interleave: takes 32 bits from each of Da.M and Db.M and splits the bits to the H and L portions of the respective destination (Dm and Dn)**

	`for (i = 0; i < 16;i++)`
	`  Dm[i] = Da[2*i]`
	`  Dm[i + 16] = Da[2*i+1]`
	`  Dn[i] = Db[2*i]`
	`  Dn[i + 16] = Db[2*i+1]`
	`The other bits of Dm, Dn are cleared`	

2	**BIT.DINTLV2BI.L Da,Dn**	**2-way de-interleave: takes 32 bits from Da.M and splits the bits to the H and L portions of the destination**

	`for (i = 0; i < 16; i++)`
	`  Dn[i] = Da[2*i]`
	`  Dn[i + 16] = Da[2*i+1]`
	`rest of Dn is cleared`	

3	**BIT.DINTLV3BI.2L Da:Db,Dm:Dn**	**3-way de-interleave: takes 24 bits from the lower part of each of the source registers (Da and Db) and splits the bits to bytes LL, LH and HL of the respective destination (Dm and Dn)**

Table continues on the next page...

#	Syntax	Description

```
for (i = 0; i < 8; i++)
  Dm[i] = Da[3*i]
  Dm[i + 8] = Da[3*i + 1]
  Dm[i + 16] = Da[3*i + 2]
  Dn[i] = Db[3*i]
  Dn[i + 8] = Db[3*i + 1]
  Dn[i + 16] = Db[3*i + 2]
The other bits of Dm, Dn are cleared
```

#	Syntax	Description
4	**BIT.DINTLV3BI.L Da,Dn**	**3-way de-interleave: takes 24 bits from the lower part of Da and splits the bits to bytes LL, LH and HL of the destination**

```
for (i = 0; i < 8; i++)
  Dn[i] = Da[3*i]
  Dn[i + 8] = Da[3*i + 1]
  Dn[i + 16] = Da[3*i + 2]
rest of Dn is cleared
```

#	Syntax	Description
5	**BIT.DINTLV3BI.REV.2L Da:Db,Dm:Dn**	**3-way reverse de-interleave: takes 24 bits from the higher part of each of Da.M and Db.M, and splits the bits to bytes HH, HL and LH of the respective destination (Dm and Dn)**

```
Dm[31:24] = {Da[31],Da[28],Da[25],Da[22],Da[19],Da[16],Da[13],Da[10]}
Dm[23:16] = {Da[30],Da[27],Da[24],Da[21],Da[18],Da[15],Da[12],Da[09]}
Dm[15:8]  = {Da[29],Da[26],Da[23],Da[20],Da[17],Da[14],Da[11],Da[08]}
Dn[31:24] = {Db[31],Db[28],Db[25],Db[22],Db[19],Db[16],Db[13],Db[10]}
Dn[23:16] = {Db[30],Db[27],Db[24],Db[21],Db[18],Db[15],Db[12],Db[09]}
Dn[15:8]  = {Db[29],Db[26],Db[23],Db[20],Db[17],Db[14],Db[11],Db[08]}
The other bits of Dm, Dn are cleared
```

#	Syntax	Description
6	**BIT.DINTLV3BI.REV.L Da,Dn**	**3-way reverse de-interleave: takes 24 bits from the higher part of Da.M and splits the bits to bytes HH, HL and LH of the destination**

```
Dn[31:24] = {Da[31],Da[28],Da[25],Da[22],Da[19],Da[16],Da[13],Da[10]}
Dn[23:16] = {Da[30],Da[27],Da[24],Da[21],Da[18],Da[15],Da[12],Da[09]}
Dn[15:8]  = {Da[29],Da[26],Da[23],Da[20],Da[17],Da[14],Da[11],Da[08]}
The other bits of Dn are cleared
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \leq n \leq 62$
Dm:Dn	D_n:D_{n+1}	$0 \leq n \leq 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

BIT.EXPND Expand Bits to Register Portions (DALU)

General Description

Select two or more bits from a source register and duplicate their value in the specified portions of the destination register. Variants allow to select 2, 4 or 8 source bits, and expand them to 40, 20, 16 or 8 bits. In addition, variants allow to select if the destination portions are in the same order as the source bits or in reverse order.

Flag Options

Flag	Position	Description
EXPND	flg1	Expand
REV	flg2	Reverse bit order

Instruction Variants

#	Syntax	Description
1	**BIT.EXPND.2T #u2_1,Da,Dn**	**Select two bits from the lower byte of Da and expand them into two packed 20-bit portions**

	`Dn.WH = 20{Da[u2+1]}`
	`Dn.WL = 20{Da[u2+0]}`
	`u2 = offset, valid values are: (0,2,4,6)`	

2	**BIT.EXPND.2W Da,Dn**	**Expand the two lower bits of Da into two packed 16-bit portions. The extension is cleared.**

	`Dn.H = 16{Da[1]}`
	`Dn.L = 16{Da[0]}`
	`Dn.E = 0`	

3	**BIT.EXPND.2X #u2_1,Da,Dm:Dn**	**Select two bits from the lower byte of Da and expand them into two 40-bit registers**

	`Dm = 40{Da[u2+1]}`
	`Dn = 40{Da[u2+0]}`
	`u2 = offset, valid values are: (0,2,4,6)`	

4	**BIT.EXPND.4T #u2_2,Da,Dm:Dn**	**Select four bits from the lower part of Da and expand them into four packed 20-bit portions in two destinations**

	`Dm.WH = 20{Da[u2+3]}`
	`Dm.WL = 20{Da[u2+2]}`
	`Dn.WH = 20{Da[u2+1]}`
	`Dn.WL = 20{Da[u2+0]}`
	`u2 = offset, valid values are: (0,4,8,12)`	

5	**BIT.EXPND.4W Da,Dm:Dn**	**Expand the four lower bits of Da into four packed 16-bit portions in two destinations. The extensions are cleared.**

Table continues on the next page...

#	Syntax	Description

	Dm.H = 16{Da[3]}
	Dm.L = 16{Da[2]}
	Dn.H = 16{Da[1]}
	Dn.L = 16{Da[0]}
	Dm.E = 0, Dn.E = 0	

6	**BIT.EXPND.8B Da.hh,Dm:Dn**	**Expand the eight bits of Da.HH into eight packed bytes in two destinations. The extensions are cleared.**

	Dn.LL = 8{ Da[24] }
	Dn.LH = 8{ Da[25] }
	Dn.HL = 8{ Da[26] }
	Dn.HH = 8{ Da[27] }
	Dm.LL = 8{ Da[28] }
	Dm.LH = 8{ Da[29] }
	Dm.HL = 8{ Da[30] }
	Dm.HH = 8{ Da[31] }
	Dm.E = 0, Dn.E = 0	

7	**BIT.EXPND.8B Da.hl,Dm:Dn**	**Expand the eight bits of Da.HL into eight packed bytes in two destinations. The extensions are cleared.**

	Dn.LL = 8{ Da[16] }
	Dn.LH = 8{ Da[17] }
	Dn.HL = 8{ Da[18] }
	Dn.HH = 8{ Da[19] }
	Dm.LL = 8{ Da[20] }
	Dm.LH = 8{ Da[21] }
	Dm.HL = 8{ Da[22] }
	Dm.HH = 8{ Da[23] }
	Dm.E = 0, Dn.E = 0	

8	**BIT.EXPND.8B Da.lh,Dm:Dn**	**Expand the eight bits of Da.LH into eight packed bytes in two destinations. The extensions are cleared.**

	Dn.LL = 8{Da[8]}
	Dn.LH = 8{Da[9]}
	Dn.HL = 8{Da[10]}
	Dn.HH = 8{Da[11]}
	Dm.LL = 8{Da[12]}
	Dm.LH = 8{Da[13]}
	Dm.HL = 8{Da[14]}
	Dm.HH = 8{Da[15]}
	Dm.E = 0, Dn.E = 0	

9	**BIT.EXPND.8B Da.ll,Dm:Dn**	**Expand the eight bits of Da.LL into eight packed bytes in two destinations. The extensions are cleared.**

	Dn.LL = 8{Da[0]}
	Dn.LH = 8{Da[1]}
	Dn.HL = 8{Da[2]}
	Dn.HH = 8{Da[3]}
	Dm.LL = 8{Da[4]}
	Dm.LH = 8{Da[5]}
	Dm.HL = 8{Da[6]}
	Dm.HH = 8{Da[7]}
	Dm.E = 0, Dn.E = 0	

Table continues on the next page...

#	Syntax	Description
10	**BIT.EXPND.REV.2T #u2_1,Da,Dn**	**Select two bits from the lower byte of Da and expand them in revise order into two packed 20-bit portions**

	`Dn.WH = 20{Da[u2+0]}`
	`Dn.WL = 20{Da[u2+1]}`
	`u2 = offset, valid values are: (0,2,4,6)`	

11	**BIT.EXPND.REV.2X #u2_1,Da,Dm:Dn**	**Select two bits from the lower byte of Da and expand them in reverse order into two 40-bit registers**

	`Dm = 40{Da[u2+0]}`
	`Dn = 40{Da[u2+1]}`
	`u2 = offset, valid values are: (0,2,4,6)`	

12	**BIT.EXPND.REV.4T #u2_2,Da,Dm:Dn**	**Select four bits from the lower part of Da and expand them in reverse order into four packed 20-bit portions in two destinations**

	`Dm.WH = 20{Da[u2+0]}`
	`Dm.WL = 20{Da[u2+1]}`
	`Dn.WH = 20{Da[u2+2]}`
	`Dn.WL = 20{Da[u2+3]}`
	`u2 = offset, valid values are: (0,4,8,12)`	

13	**BIT.EXPND.REV.8B Da.hh,Dm:Dn**	**Expand the eight bits of Da.HH into eight packed bytes in reverse order in two destinations. The extensions are cleared.**

	`Dn.LL = 8{Da[31]}`
	`Dn.LH = 8{Da[30]}`
	`Dn.HL = 8{Da[29]}`
	`Dn.HH = 8{Da[28]}`
	`Dm.LL = 8{Da[27]}`
	`Dm.LH = 8{Da[26]}`
	`Dm.HL = 8{Da[25]}`
	`Dm.HH = 8{Da[24]}`
	`Dm.E = 0, Dn.E = 0`	

14	**BIT.EXPND.REV.8B Da.hl,Dm:Dn**	**Expand the eight bits of Da.HL into eight packed bytes in reverse order in two destinations. The extensions are cleared.**

	`Dn.LL = 8{Da[23]}`
	`Dn.LH = 8{Da[22]}`
	`Dn.HL = 8{Da[21]}`
	`Dn.HH = 8{Da[20]}`
	`Dm.LL = 8{Da[19]}`
	`Dm.LH = 8{Da[18]}`
	`Dm.HL = 8{Da[17]}`
	`Dm.HH = 8{Da[16]}`
	`Dm.E = 0, Dn.E = 0`	

15	**BIT.EXPND.REV.8B Da.lh,Dm:Dn**	**Expand the eight bits of Da.LH into eight packed bytes in reverse order in two destinations. The extensions are cleared.**

Table continues on the next page...

#	Syntax	Description

	`Dn.LL = 8{Da[15]}`
	`Dn.LH = 8{Da[14]}`
	`Dn.HL = 8{Da[13]}`
	`Dn.HH = 8{Da[12]}`
	`Dm.LL = 8{Da[11]}`
	`Dm.LH = 8{Da[10]}`
	`Dm.HL = 8{Da[9]}`
	`Dm.HH = 8{Da[8]}`
	`Dm.E = 0, Dn.E = 0`	

16	**BIT.EXPND.REV.8B Da.ll,Dm:Dn**	**Expand the eight bits of Da.LL into eight packed bytes in reverse order in two destinations. The extensions are cleared.**

	`Dn.LL = 8{Da[7]}`
	`Dn.LH = 8{Da[6]}`
	`Dn.HL = 8{Da[5]}`
	`Dn.HH = 8{Da[4]}`
	`Dm.LL = 8{Da[3]}`
	`Dm.LH = 8{Da[2]}`
	`Dm.HL = 8{Da[1]}`
	`Dm.HH = 8{Da[0]}`
	`Dm.E = 0, Dn.E = 0`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dm:Dn	D_n:D_{n+1}	0 ≤ n ≤ 62
Dn	D0-D63	
u2_1	0 ≤ u2_1 < 2^3;	u2_1 & 0x1 == 0
u2_2	0 ≤ u2_2 < 2^4;	u2_2 & 0x3 == 0

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 3, 4, 10, 11, 12
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

BIT.INTLV Two-Way or Three-Way Bit Interleave (DALU)

General Description

Interleave the bits of two 16-bit words from the source register, or of three bytes from the source register, into a destination register. Unspecified portions of the destination (such as the extension) are cleared. SIMD2 variants perform this operation in parallel for two input and output registers.

Flag Options

Flag	Position	Description
2BI	flg1	Two Buffer Interleave Data
3BI	flg1	Three Buffer Interleave Data
INTLV	flg1	Interleave
REV	flg2	Reverse bit order

Instruction Variants

<table>
<tr><th>#</th><th>Syntax</th><th>Description</th></tr>
<tr><td><b>1</b></td><td><b>BIT.INTLV2BI.2L Da:Db,Dm:Dn</b></td><td><b>SIMD2 two-way interleave: independently interleave the bits of the H and L portions of a source register (Da or Db) into the respective destination register (Dm or Dn)</b></td></tr>
<tr><td></td><td colspan="2"><pre>for (i = 0; i < 16; i++)
  Dm[2*i] = Da[i]
  Dm[2*i + 1] = Da[i + 16]
  Dn[2*i] = Db[i]
  Dn[2*i + 1] = Db[i + 16]
The other bits of Dm, Dn are cleared</pre></td></tr>
<tr><td><b>2</b></td><td><b>BIT.INTLV2BI.L Da,Dn</b></td><td><b>2-way interleave: Interleave the bits of Da.H and Da.L into Da.M</b></td></tr>
<tr><td></td><td colspan="2"><pre>for (i = 0; i < 16; i++)
  Dn[2*i] = Da[i]
  Dn[2*i + 1] = Da[i + 16]
The other bits of Dn are cleared</pre></td></tr>
<tr><td><b>3</b></td><td><b>BIT.INTLV3BI.2L Da:Db,Dm:Dn</b></td><td><b>SIMD2 3-way interleave: independently interleave the bits of the bytes LL, LH, HL from each of the source registers (Da,Db) into the lower 24 bits of the respective destinations (Dm, Dn)</b></td></tr>
<tr><td></td><td colspan="2"><pre>for (i = 0; i < 8; i++)
  Dm[3*i] = Da[i]
  Dm[3*i + 1] = Da[i + 8]
  Dm[3*i + 2] = Da[i + 16]
  Dn[3*i] = Db[i]
  Dn[3*i + 1] = Db[i + 8]
  Dn[3*i + 2] = Db[i + 16]
The other bits of Dm, Dn are cleared</pre></td></tr>
</table>

Table continues on the next page...

#	Syntax	Description
4	**BIT.INTLV3BI.L Da,Dn**	**3-way interleave: Interleave the bits of the bytes LL, LH, HL of the source register into the lower 24 bits of Dn**

```
for (i = 0; i < 8; i++)
  Dn[3*i] = Da[i]
  Dn[3*i + 1] = Da[i + 8]
  Dn[3*i + 2] = Da[i + 16]
The other bits of Dn are cleared
```

#	Syntax	Description
5	**BIT.INTLV3BI.REV.2L Da:Db,Dm:Dn**	**SIMD2 3-way reverse interleave: independently interleave the bits of the bytes HH, HL, LH from each of the source registers (Da,Db) into the upper 24 bits (excluding the extension) of the respective destinations (Dm, Dn)**

```
{Dm[31],Dm[28],Dm[25],Dm[22],Dm[19],Dm[16],Dm[13],Dm[10]} = Da[31:24]
{Dm[30],Dm[27],Dm[24],Dm[21],Dm[18],Dm[15],Dm[12],Dm[09]} = Da[23:16]
{Dm[29],Dm[26],Dm[23],Dm[20],Dm[17],Dm[14],Dm[11],Dm[08]} = Da[15:8]
{Dn[31],Dn[28],Dn[25],Dn[22],Dn[19],Dn[16],Dn[13],Dn[10]} = Db[31:24]
{Dn[30],Dn[27],Dn[24],Dn[21],Dn[18],Dn[15],Dn[12],Dn[09]} = Db[23:16]
{Dn[29],Dn[26],Dn[23],Dn[20],Dn[17],Dn[14],Dn[11],Dn[08]} = Db[15:8]
The other bits of Dm, Dn are cleared
```

#	Syntax	Description
6	**BIT.INTLV3BI.REV.L Da,Dn**	**3-way reverse interleave: Interleave the bits of the bytes HH, HL, LH of the source register into the upper 24 bits of Dn.M**

```
{Dn[31],Dn[28],Dn[25],Dn[22],Dn[19],Dn[16],Dn[13],Dn[10]} = Da[31:24]
{Dn[30],Dn[27],Dn[24],Dn[21],Dn[18],Dn[15],Dn[12],Dn[09]} = Da[23:16]
{Dn[29],Dn[26],Dn[23],Dn[20],Dn[17],Dn[14],Dn[11],Dn[08]} = Da[15:8]
The other bits of Dn are cleared
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

BIT.REV Bit Reverse (DALU)

General Description

Reverse the order of bits, either the 32 bits of Da.M or for each byte individually. The extension is cleared.

Flag Options

Flag	Position	Description
REV	flg1	Reverse

Instruction Variants

#	Syntax	Operation	Description

1	BIT.REV.4B Da,Dn	`Dn[24:31] = Da[31:24]`
		`Dn[16:23] = Da[23:16]`
		`Dn[8:15] = Da[15:8]`
		`Dn[0:7] = Da[7:0]`
		`Dn.E = 0`
			Reverse the order of bits in each byte of Da.M independently.

2	BIT.REV.L Da,Dn	`Dn[0:31] = Da[31:0]`
		`Dn.E = 0`
			Reverse the order of bits of Da.M

Explicit Operands

Operand	Permitted Values
Da	`D0-D63`
Dn	`D0-D63`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

BMCHGA Bit-Mask Change a 16-Bit Operand (IPU)

General Description

Inverts selected bits in the high or low portion of the destination control or address register using an immediate unsigned 16-bit value as a mask. Each bit that is set in the mask inverts the corresponding bit in the destination register. Bits that are not selected in the mask, as well as bits in the other portion of the register, are unaffected. The instruction reads the full value from the destination register, modifies the specified bits, and writes the new value back to that register. The operation is equivalent to the exclusive-or function.

Instruction Variants

#	Syntax	Operation	Description
1	BMCHGA #u16,C4.h	C4 = C4 ^ ({u16, 0x0000})	Bit mask change the high portion of a control register
2	BMCHGA #u16,C4.l	C4 = C4 ^ (U32)u16	Bit mask change the low portion of a control register
3	BMCHGA #u16,Rn.h	Rn = Rn ^ ({u16, 0x0000})	Bit mask change the high portion of an address register
4	BMCHGA #u16,Rn.l	Rn = Rn ^ ((U32)u16)	Bit mask change the low portion of an address register

Explicit Operands

Operand	Permitted Values				
C4	EIDR, SR2,	GCR, TMTAG	MCTL,	MOCR,	SR,
Rn	R0-R31				
u16	$0 \leq u16 < 2^{16}$				

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	48-bits	All
Pipeline behavior	agu_AAU_CTRL_REG	1, 2
	agu_AAU_LOGIC	3, 4

BMCLRA Bit-Mask Clear a 16-Bit Operand (IPU)

General Description

Clears selected bits in the high or low portion of the destination control or address register using an immediate unsigned 16-bit value as a mask. Each bit that is set in the mask clears the corresponding bit in the destination register. Bits that are not selected in the mask, as well as bits in the other portion of the register, are unaffected. The instruction reads the full value from the destination register, modifies the specified bits, and writes the new value back to that register. The operation is equivalent to the AND function.

Instruction Variants

#	Syntax	Description
1	**BMCLRA #u16,C4.h**	**Bit mask clear the high portion of a control register**
	`C4 = C4 & (~ ({u16, 0x0000}))`	
2	**BMCLRA #u16,C4.l**	**Bit mask clear the low portion of a control register**
	`C4 = C4 & ~ (U32)u16`	
3	**BMCLRA #u16,Rn.h**	**Bit mask clear the high portion of an address register**
	`Rn = Rn & ~ ({u16, 0x0000})`	
4	**BMCLRA #u16,Rn.l**	**Bit mask clear the low portion of an address register**
	`Rn = Rn & ~ ({0x0000, u16})`	

Explicit Operands

Operand	Permitted Values				
C4	EIDR, SR2,	GCR, TMTAG	MCTL,	MOCR,	SR,
Rn	R0-R31				
u16	$0 \le u16 < 2^{16}$				

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	48-bits	All
Pipeline behavior	agu_AAU_CTRL_REG	1, 2
	agu_AAU_LOGIC	3, 4

BMSETA Bit-Mask Set a 16-Bit Operand (IPU)

General Description

Sets selected bits in the high or low portion of the destination control or address register using an immediate unsigned 16-bit value as a mask. Each bit that is set in the mask sets the corresponding bit in the destination register. Bits that are not selected in the mask, as well as bits in the other portion of the register, are unaffected. The instruction reads the full value from the destination register, modifies the specified bits, and writes the new value back to that register. The operation is equivalent to the OR function.

Instruction Variants

#	Syntax	Operation	Description
1	BMSETA #u16,C4.h	`C4 = C4	{u16, 0x0000}`	Bit mask set the high portion of a control register
2	BMSETA #u16,C4.l	`C4 = C4	{0x0000, u16}`	Bit mask set the low portion of a control register
3	BMSETA #u16,Rn.h	`Rn = Rn	{u16, 0x0000}`	Bit mask set the high portion of an address register
4	BMSETA #u16,Rn.l	`Rn = Rn	{0x0000, u16}`	Bit mask set the low portion of an address register

Explicit Operands

Operand	Permitted Values				
C4	EIDR, SR2,	GCR, TMTAG	MCTL,	MOCR,	SR,
Rn	R0-R31				
u16	$0 \leq u16 < 2^{16}$				

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	48-bits	All
Pipeline behavior	agu_AAU_CTRL_REG	1, 2
	agu_AAU_LOGIC	3, 4

BMTSTA Bit Mask Test a 16-bit Operand (IPU)

General Description

Tests the specified bits in a 16-bit high or low portion of the source address or control register. The bits to test are specified in a 16-bit immediate operand which serves as a mask - only bits for which the mask is set are tested. The (.S) variant will return a true result if all the selected bits were set, and false otherwise. Similarly, the (.C) variant will return a true result if all the selected bits were clear. Variants with a single predicate destination will capture the test result in it. Variants which update a predicate pair will update the first predicate with the test result and the inverse result in the second predicate. As with other compare and test instructions, in some cases several instructions updating the same predicates may be grouped together in the same VLES, in which case the test result is the logical OR of the individual tests. The instruction may be in itself predicated by IF.Pn or IF.Pn.EC conditions. For more information on the semantics of predicate updating instructions, see the Program Control chapter.

Flag Options

Flag	Position	Description
C	flg1	Clear
S	flg1	Set

Instruction Variants

#	Syntax	Description
1	**BMTSTA.C #u16,C4.h,Pm:Pn**	**Bit mask test for clear the selected bits in the high portion of a control register, updating a predicate pair**

```
if ((C4[31:16] & u16) == 0x0000) {
        Pm = 1
        Pn = 0
} else {
        Pm = 0
        Pn = 1
}
```

#	Syntax	Description
2	**BMTSTA.C #u16,C4.h,Pn**	**Bit mask test for clear the selected bits in the high portion of a control register, updating one predicate**

```
if ((C4[31:16] & u16) == 0x0000) {
        Pn = 1
} else {
        Pn = 0
}
```

#	Syntax	Description
3	**BMTSTA.C #u16,C4.l,Pm:Pn**	**Bit mask test for clear the selected bits in the low portion of a control register, updating a predicate pair**

Table continues on the next page...

#	Syntax	Description

	if ((C4[15:0] & u16) == 0x0000) {
	Pm = 1
	Pn = 0
	} else {
	Pm = 0
	Pn = 1
	}	

4	**BMTSTA.C #u16,C4.l,Pn**	**Bit mask test for clear the selected bits in the low portion of a control register, updating one predicate**

	if ((C4[15:0] & u16) == 0x0000) {
	Pn = 1
	} else {
	Pn = 0
	}	

5	**BMTSTA.C #u16,Ra.h,Pm:Pn**	**Bit mask test for clear the selected bits in the high portion of an address register, updating a predicate pair**

	if ((Ra.H & u16) == 0x0000) {
	Pm = 1
	Pn = 0
	} else {
	Pm = 0
	Pn = 1
	}	

6	**BMTSTA.C #u16,Ra.h,Pn**	**Bit mask test for clear the selected bits in the high portion of an address register, updating one predicate**

	if ((Ra.H & u16) == 0x0000) {
	Pn = 1
	} else {
	Pn = 0
	}	

7	**BMTSTA.C #u16,Ra.l,Pm:Pn**	**Bit mask test for clear the selected bits in the low portion of an address register, updating a predicate pair**

	if ((Ra.L & u16) == 0x0000) {
	Pm = 1
	Pn = 0
	} else {
	Pm = 0
	Pn = 1
	}	

Table continues on the next page...

#	Syntax	Description
8	**BMTSTA.C #u16,Ra.l,Pn**	**Bit mask test for clear the selected bits in the low portion of an address register, updating one predicate**

	`if ((Ra.L & u16) == 0x0000) {`
	`        Pn = 1`
	`} else {`
	`        Pn = 0`
	`}`	

9	**BMTSTA.S #u16,C4.h,Pm:Pn**	**Bit mask test for set the selected bits in the high portion of a control register, updating a predicate pair**

	`if (((~ (C4[31:16])) & u16) == 0x0000) {`
	`        Pm = 1`
	`        Pn = 0`
	`} else {`
	`        Pm = 0`
	`        Pn = 1`
	`}`	

10	**BMTSTA.S #u16,C4.h,Pn**	**Bit mask test for set the selected bits in the high portion of a control register, updating one predicate**

	`if (((~ (C4[31:16])) & u16) == 0x0000) {`
	`        Pn = 1`
	`} else {`
	`        Pn = 0`
	`}`	

11	**BMTSTA.S #u16,C4.l,Pm:Pn**	**Bit mask test for set the selected bits in the low portion of a control register, updating a predicate pair**

	`if (((~ (C4[15:0])) & u16) == 0x0000) {`
	`        Pm = 1`
	`        Pn = 0`
	`} else {`
	`        Pm = 0`
	`        Pn = 1`
	`}`	

12	**BMTSTA.S #u16,C4.l,Pn**	**Bit mask test for set the selected bits in the low portion of a control register, updating one predicate**

	`if (((~ (C4[15:0])) & u16) == 0x0000) {`
	`        Pn = 1`
	`} else {`
	`        Pn = 0`
	`}`	

13	**BMTSTA.S #u16,Ra.h,Pm:Pn**	**Bit mask test for set the selected bits in the high portion of an address register, updating a predicate pair**

Table continues on the next page...

#	Syntax	Description

```
if (((~ (Ra.H)) & u16) == 0x0000) {
        Pm = 1
        Pn = 0
} else {
        Pm = 0
        Pn = 1
}
```

#	Syntax	Description
14	**BMTSTA.S #u16,Ra.h,Pn**	**Bit mask test for set the selected bits in the high portion of an address register, updating one predicate**

```
if (((~ (Ra.H)) & u16) == 0x0000) {
        Pn = 1
} else {
        Pn = 0
}
```

#	Syntax	Description
15	**BMTSTA.S #u16,Ra.l,Pm:Pn**	**Bit mask test for set the selected bits in the low portion of an address register, updating a predicate pair**

```
if (((~ (Ra.L)) & u16) == 0x0000) {
        Pm = 1
        Pn = 0
} else {
        Pm = 0
        Pn = 1
}
```

#	Syntax	Description
16	**BMTSTA.S #u16,Ra.l,Pn**	**Bit mask test for set the selected bits in the low portion of an address register, updating one predicate**

```
if (((~ (Ra.L)) & u16) == 0x0000) {
        Pn = 1
} else {
        Pn = 0
}
```

Explicit Operands

Operand	Permitted Values				
C4	EIDR, SR2,	GCR, TMTAG	MCTL,	MOCR,	SR,
Pm:Pn	P_n:P_{n+1}	$0 \le n \le 4$			
Pn	P0-P5				
Ra	R0-R31				
u16	$0 \le u16 < 2^{16}$				

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	48-bits	All
IF.EC		All
Pipeline behavior	agu_AAU_Pbit	All

BRA Branch to a Destination (PCU)

General Description

Transfers program execution to the specified PC-relative address. The destination is specified by the user as a label, and the assembler encodes the 19-bit PC-relative signed PC offset. An instruction variant may be marked not to be affected and not affect the BTB (NOBTB flag).

Flag Options

Flag	Position	Description
NOBTB	flg1	Does not use nor updates the BTB

Instruction Variants

#	Syntax	Operation	Description
1	BRA[.NOBTB] LABEL	`PC = PC + {RelAdd19,0}`	Branch. By default is subject to BTB acceleration, unless the NOBTB flag is used.

Explicit Operands

Operand	Permitted Values
RelAdd19	$-2^{18} \leq \text{RelAdd19} < 2^{18}$

Affect Instructions

Field	Relevant Variants	Comments
PC	All	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Pipeline behavior	pcu_COF	All

BREAK Terminate the Loop and Branch to the Specified Address (PCU)

General Description

Unconditionally terminates the loop of index n2 and branches to the specified PC-relative address. The destination is specified by the user as a label, and the assembler calculates and encodes the appropriate 19-bit signed offset field. The instruction does not change the loop programming model, however the variant with the right loop index should be used.

Flag Options

Flag	Position	Description
n2	flg1	Loop index (0, 1, 2, 3)

Instruction Variants

#	Syntax	Operation	Description
1	BREAK.n2 LABEL	`PC = PC + {RelAdd19,0}`	Break execution of the hardware loop with index n2, and branch to the specified destination.

Explicit Operands

Operand	Permitted Values
RelAdd19	$-2^{18} \le \texttt{RelAdd19} < 2^{18}$ $0 \le \texttt{n2} \le 3$

Affect Instructions

Field	Relevant Variants	Comments
PC	All	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	48-bits	All
Pipeline behavior	pcu_COF_LP	All

BSR Branch to a Subroutine (PCU_LSU)

General Description

Implicitly pushed SR and the PC of the next VLES (the return PC) to the software stack, pointed to by the active SP, and increments SP by 8. Then a branch is performed to the specified destination. The destination is specified by the user as a label, and the assembler calculates and encodes the 19-bit PC-relative signed PC offset. The return PC is also saved in the RAS, optionally accelerating a subsequent RTS instruction. The BSR instruction is affected and affecting the BTB.

Flag Options

Flag	Position	Description
NOBTB	flg1	Does not use nor updates the BTB

Instruction Variants

#	Syntax	Description
1	**BSR[.NOBTB] LABEL**	**Branch to a subroutine**

	`PC = PC + {RelAdd19,0}`
	`Mem(SP,8) = {return PC,SR}; SP = SP + 8`	

Explicit Operands

Operand	Permitted Values
RelAdd19	$-2^{18} \le \texttt{RelAdd19} < 2^{18}$

Affect Instructions

Field	Relevant Variants	Comments
PC	All	
SP, ESP, TSP, DSP	All	
SR	All	
SR2.SPSEL	All	

Affected By Instructions

Field	Relevant Variants
SP, ESP, TSP, DSP	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Pipeline behavior	pcu_COF	All

CAST.LL Cast 40-bit integer into 64-bit (DALU)

General Description

Right-aligns a 40-bit integer value, sign extends it to 64 bits, and writes into a register pair. The extension of the low register of the result is cleared, while the extension of the high register of the result is sign extended with the 64-bit value.

Flag Options

Flag	Position	Description
X	flg1	40 bit input

Instruction Variants

#	Syntax	Operation	Description

1	CAST.X.LL Da,Dm:Dn	`Dm = (S40)Da.E`
		`Dn.M = Da.M`
		`Dn.E = 0`
			Cast 40-bit integer into 64-bit result

Explicit Operands

Operand	Permitted Values	
Da	`D0-D63`	
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

CAST.nX Cast 20-bit fraction into 40-bit (DALU)

General Description

Extracts one 20-bit portion from a data register (WH or WL), and casts it to a standard 40-bit fractional format.

In the legacy variant (LEG) the result is saturated to a 32-bit value if SR.SM is set.

Flag Options

Flag	Position	Description
T	flg1	20 bits
LEG	flg2	Legacy instruction

Instruction Variants

#	Syntax	Description
1	**CAST.T.X Da.h,Dn**	**Cast high 20-bit fraction into 40-bit register**
	`Dn = (S40){Da[39:36], Da[31:16], 0x0000}`	
2	**CAST.T.X Da.l,Dn**	**Cast low 20-bit fraction into 40-bit register**
	`Dn = (S40){Da[35:32], Da[15:0], 0x0000}`	
3	**CAST.T.LEG.X Da.h,Dn**	**Cast high 20-bit fraction into 40-bit register and saturate by SR (legacy)**
	`Dn = srSAT32(({(S5)Da[39], Da[39:36], Da.H, 0x0000}))`	
4	**CAST.T.LEG.X Da.l,Dn**	**Cast low 20-bit fraction into 40-bit register and saturate by SR (legacy)**
	`Dn = srSAT32(({(S5)Da[35], Da[35:32], Da.L, 0x0000}))`	

Explicit Operands

Operand	Permitted Values
Da	`D0-D63`
Dn	`D0-D63`

Affect Instructions

Field	Relevant Variants	Comments
SR.SM	3, 4	

Affected By Instructions

Field	Relevant Variants
SR.SAT	3, 4

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

CCMD CME Commands (LSU)

General Description

This instruction is a constituent of the CCMDM meta-instruction, used to perform global operations on the CME (Cache Management Engine). The CCMD instruction should not be used in this form but only through DBARM; using the instruction on its own will give indeterminate results. For more information on the CCMDM instruction, see its description page and also the Address Generation chapter.

Flag Options

Flag	Position	Description
RESM	flg1	Resume
SUSP	flg1	Suspend

Instruction Variants

#	Syntax	Operation	Description
1	CCMD.RESM		Resume all suspended CME tasks
2	CCMD.SUSP		Suspend all active CME tasks

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Helper instruction		All
No predication		All
Pipeline behavior	agu_MOVE_LD_cmd	All

CCMDM CME Control Operation (META INST)

General Description

These meta-instructions perform global operations on the CME (Cache Management Engine). Variants allow to suspend all active CME tasks, and to resume all suspended tasks. The same operations could also be configured by writing to the CME control registers. The meta-instruction is encoded by a CCMD instruction of the right variant, grouped with a SYNC.B instruction. Some programming rules apply, refer to rule A.9 in the Programming Rules chapter. For more information on the CME and cache operations a, refer to the Address Generation chapter in the core RM and the Cache Management chapter in the Subsystem RM

Flag Options

Flag	Position	Description
RESM	flg1	Resume
SUSP	flg1	Suspend

Instruction Variants

#	Syntax	Description
1	**CCMDM.RESM**	**Resume all suspended CME tasks**
	`Encoded as: CCMD.RESM SYNC.B`	
2	**CCMDM.SUSP**	**Suspend all active CME tasks**
	`Encoded as: CCMD.SUSP SYNC.B`	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	64-bits	All
Execution unit	PCU+LSU	All
No predication		All
Pipeline behavior	per constructing instructions	All

CHCF Change Control Fields (IPU)

General Description

This instruction allows to modify individual fields in SR and SR2, while not affecting the other fields in the register. It allows to change the value of the specified field to any value, regardless of the previous state of the field. The destination field is updated either according to an immediate value, or, in some cases, from the bits in the same field position in an R operand. These variants allow to restore the previous settings of a field if SR2 was previously saved in Ra. In most cases, the instruction is not considered as a full update of the destination register in the context of hazard detection logic.

Instruction Variants

#	Syntax	Operation	Description
1	CHCF Ra,SR2.ASP	`SR2.ASPSEL = Ra[3:2]`	Change the alternate SP selection in SR2 according to a field in Ra. Allows to restore the previous ASPSEL settings if SR2 was previously saved in Ra.
2	CHCF Ra,SR2.IPM	`SR2.IPM = Ra[20:16]`	Change the interrupt priority mask in SR2 according to a field in Ra. Allows to restore the previous ASPSEL settings if SR2 was previously saved in Ra.
3	CHCF #u1,SR.RM	`SR.RM = u1`	Change Rounding Mode in SR to the state specified in the immediate.
4	CHCF #u1,SR.SM	`SR.SM = u1`	Change Saturation Mode in SR to the state specified in the immediate.
5	CHCF #u2,SR2.ASP	`SR2.ASPSEL = u2`	Change the alternate SP selection in SR2 to the state specified in the immediate.
6	CHCF #u2,SR.SWM2	`{SR.SM2,SR.W20} = u2`	Change both the W20 and SM2 modes in SR to the state specified in the immediate.
7	CHCF #u3,SR.SCM	`SR.SCM = u3`	Change Scaling Mode in SR to the state specified in the immediate.

Explicit Operands

Operand	Permitted Values
Ra	R0-R31
u1	$0 \le u1 < 2^1$
u2	$0 \le u2 < 2^2$
u3	$0 \le u3 < 2^3$

Affect Instructions

Field	Relevant Variants	Comments
SR.RM	3, 4, 6, 7	
SR.SCM	3, 4, 6, 7	
SR.SM	3, 4, 6, 7	
SR.SM2	3, 4, 6, 7	
SR.W20	3, 4, 6, 7	

Affected By Instructions

Field	Relevant Variants
SR.RM	3
SR.SCM	7
SR.SM	4
SR.SM2	6
SR.W20	6
SR2.ASPSEL	1, 5
SR2.IPM	2

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Pipeline behavior	agu_AAU_CTRL_REG	All

CLB.nX Count Leading Bits (DALU)

General Description

Counts the leading zeros or ones in the source data register, starting with bit 39, for assisting in normalizing the source value by a subsequent shift. For a negative number (bit 39 is 1), counts leading ones; for positive numbers (bit 39 is 0), counts leading zeros. The result is adjusted by 9 so that the required shift will normalize the number to a 32-bit value.
In LFT variants, 9 is substracted from the counted value, so the results are in the range of -8 to 31. A zero source gives 31.

In the LEG (legacy) variant, the counted value is subtracted from 9, so the results are in the range of -31 to 8. A zero source gives 0.

Flag Options

Flag	Position	Description
LFT	flg1	Left
LEG	flg2	Legacy instruction

Instruction Variants

#	Syntax	Description
1	**CLB.LFT.2X Da:Db,Dm:Dn**	**SIMD2 count leading bits**
	`SIMD2 of CLB.LFT.X`	
2	**CLB.LFT.X Da,Dn**	**Count leading bits**

	`if Da == 0 then Dn=31`
	`if Da[39] == 0 then Dn = (number of leading zeros in Da) - 9`
	`if Da[39] == 1 then Dn = (number of leading ones in Da) - 9`	

3	**CLB.LEG.X Da,Dn**	**Legacy count leading bits**

	`If (Da == 0)`
	`  Dn = 0`
	`else if (Da[39] == 0)`
	`  Dn = 9 - (number of consecutive leading zeros in Da[39:0])`
	`else`
	`  Dn = 9 - (number of consecutive leading ones in Da[39:0])`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

CLBA Count Leading Bits (LSU,IPU)

General Description

Counts the leading zeros or ones in the source address register, starting with bit 31, for assisting in normalizing the source value by a subsequent shift. For a negative number (bit 31 is 1), counts leading ones; for positive numbers (bit 31 is 0), counts leading zeros. It subtract 1 from count value into result. A zero source gives 31.

Flag Options

Flag	Position	Description
LFT	flg1	Left

Instruction Variants

#	Syntax	Description
1	**CLBA.LFT Ra,Rn**	

```
if Ra == 0 then Rn=31
if Ra[31] == 0 then Rn= (number of leading zeros in Ra) - 1
if Ra[31] == 1 then Rn= (number of leading ones in Ra) - 1
```

Explicit Operands

Operand	Permitted Values
Ra	R0-R31
Rn	R0-R31

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Execution unit	IPU	All
	LSU	All
Pipeline behavior	agu_AAU_LOGIC	All

CLIP.nB Clip signed 16-bit/20-bit/40-bit into unsigned byte (DALU)

General Description

Shift a 16-bit/20-bit/40-bit signed value right by an immediate field, round the result by adding the last bit that was shifteed out, and clip the result to an unsigned byte (saturate the result to 0xFF if larger than 0xFF and set result to zero if negative). If clipping occurred, SR.SAT is set.

Flag Options

Flag	Position	Description
T	flg1	20 bits
W	flg1	Word (16 bits)
X	flg1	40 bit input
U	flg2	Unsigned

Instruction Variants

#	Syntax	Description
1	**CLIP.T.U.2B #u5,Da,Dn**	**SIMD2 clip 20-bit signed value to unsigned byte**

```
If Dn.{WH/WL} < 0x00000 then
       Dn.{WH/WL} = 0x00000
else
      if RND (Da.{WH/WL} >> u5) > 0x000FF then
            Dn.{WH/WL} = 0x000FF
      else
            Dn.{WH/WL} = RND (Da.{WH/WL} >> u5)
```

#	Syntax	Description
2	**CLIP.W.U.2B #u5,Da,Dn**	**SIMD2 clip 16-bit signed value to unsigned byte**

```
If Da.{HL} < 0x0000 then
        Dn.{WH/WL} = 0x00000
 else
       if RND (Da.{HL} >> u5[3:0]) > 0x00FF then
            Dn.{WH/WL} = 0x000FF
       else
            Dn.{WH/WL} = RND (Da.{HL} >> u5[3:0])
```

#	Syntax	Description
3	**CLIP.X.U.B #u5,Da,Dn**	**Clip 40-bit signed value to unsigned byte**

```
If Da < 0x0 then
    Dn = 0x0
else
    if RND (Da >> u5) > 0xFF then
          Dn = 0xFF
    else
          Dn = RND (Da >> u5)
```

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Dn	D0-D63
u5	$0 \le u5 < 2^5$

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

CLR.X Clear a Data Register (DALU)

General Description

The instruction clears the contents of a D register, making the content all zeros. The instruction is an alias of a SUB instruction, subtracting the destination register from itself.

Instruction Variants

#	Syntax	Description
1	**CLR.X Dn**	**Clear a D register**

	`Dn = 0`
	`Alias, encoded as: SUB.X Dn,Dn,Dn`	

Explicit Operands

Operand	Permitted Values
Dn	`D0-D63`

Instruction Attributes

Attribute	Value	Relevant Variant #
Alias		All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

CLRA Clear an R register (LSU,IPU)

General Description

The instruction clears the contents of an R register, making the content all zeros. The instruction is an alias of a SUBA instruction, subtracting the destination register from itself.

Instruction Variants

#	Syntax	Description
1	**CLRA Rn**	**Clear an R register**

	`Rn = 0`
	`Alias, encoded as: SUBA.LIN Rn,Rn,Rn`	

Explicit Operands

Operand	Permitted Values
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Alias		All
Encoding length	32-bits	All
Execution unit	IPU	All
	LSU	All
Pipeline behavior	agu_AAU	All

CLRIC Clear Internal Context (PCU)

General Description

Flushes the instruction pipeline, performing a full re-fetch of the next VLES. In addition, may clear some micro-architectural information, according to the value of the immediate operand, as follows:
u5[0], if set, the BTB is cleared
u5[1], if set, the RAS is cleared
The other bits are reserved.

Clearing the BTB or the RAS may be required in some cases of task switch, self-modifying code, MMU program description change or explicit return address manipulation. Note that the RTE instruction has variants that can also perform such context clearing as part of its operation. RAS clearing may also be done by some variants of RTS.

Instruction Variants

#	Syntax	Operation	Description

1	CLRIC #u5	`Flush the pipe and,`
		`if (u5[0]) then clear BTB`
		`if (u5[1]) then clear RAS`
			Clear internal context according to the immediate field

Explicit Operands

Operand	Permitted Values
u5	$0 \leq u5 < 2^5$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
No predication		All
Pipeline behavior	pcu_control	All

CMP.nL Compare 32 Bits in a Data Register (DALU)

General Description

Performs a signed or unsigned comparison of the lower 32-bit data of a D register with another 32-bit operand - either in another D register or in an immediate value. The comparison type could be one of several conditions: equality (EQ), inequality (NE), greater than (GT), greater or equal (GE), less than (LT), and less or equal (LE). The condition is specified in position FLG1 of the mnemonic. In the variant table below, instructions that operate on the same sources and update the same destinations are described collectively under one entry, where the condition is generically specified as <op>. The functional semantics for evaluating conditions involving two registers for which order matters is from right to left, for example CMP.GT Da,Db is true if Db>Da. Not all instructions support all condition tests, so the exact substitution list is written in the description of each table entry. By default, comparisons are signed unless specified as unsigned (in FLG2 position of the mnemonic). Variants with a single predicate destination will capture the test result in it. Variants which update a predicate pair will update the first predicate with the test result and the inverse result in the second predicate. As with other compare and test instructions, in some cases several instructions updating the same predicates may be grouped together in the same VLES, in which case the test result is the logical OR of the individual tests. The instruction may be in itself predicated by IF.Pn or IF.Pn.EC conditions. For more information on the semantics of predicate updating instructions, see the Program Control chapter.

Flag Options

Flag	Position	Description
EQ	flg1	Equal comparison
GE	flg1	Greater or Equal comparison
GT	flg1	Greater Than comparison
LE	flg1	Less or Equal comparison
LT	flg1	Less Than comparison
NE	flg1	Not Equal comparison
U	flg2	Unsigned

Instruction Variants

#	Syntax	Description
1	**CMP.<*op*>.L #s32,Da,Pm:Pn**	**Compare Da.M with a 32-bit immediate for equal/ greater/less or equal/not equal, updating the result in two predicates. "<*op*>" is one of: EQ, GT, LE, NE in the syntax, and one of ==, >, ≤, != in the operation, respectively.**

```
if (Da.M <op> s32) {
        Pm = 1
        Pn = 0
} else {
        Pm = 0
        Pn = 1
}
```

Table continues on the next page...

#	Syntax	Description
2	**CMP.<*op*>.L #s32,Da,Pn**	**Compare Da.M register with a 32-bit immediate for equal/greater/less or equal/not equal, updating the result in one predicate. "<*op*>" is one of: EQ, GT, LE, NE in the syntax, and one of ==, >, ≤, != in the operation, respectively.**

	`if (Da.M <op> s32) {`
	`        Pn = 1`
	`} else {`
	`        Pn = 0`
	`}`	

3	**CMP.<*op*>.L #ux5,Da,Pm:Pn**	**Compare Da.M with a short immediate for equal/greater/less or equal/not equal, updating the result in two predicates. "<*op*>" is one of: EQ, GT, LE, NE in the syntax, and one of ==, >, ≤, != in the operation, respectively.**

	`imm = (ux5 == 31) ? (- 1) : ux5`
	`if (Da.M <op> imm) {`
	`        Pm = 1`
	`        Pn = 0`
	`} else {`
	`        Pm = 0`
	`        Pn = 1`
	`}`	

4	**CMP.<*op*>.L #ux5,Da,Pn**	**Compare Da.M register with a 32-bit zero-extended short immediate for equal/greater/less or equal/not equal, updating the result in one predicate. "<*op*>" is one of: EQ, GT, LE, NE in the syntax, and one of ==, >, ≤, != in the operation, respectively.**

	`imm = (ux5 == 31) ? (- 1) : ux5`
	`if (Da.M <op> imm) {`
	`        Pn = 1`
	`} else {`
	`        Pn = 0`
	`}`	

5	**CMP.<*op*>.L Da,Db,Pm:Pn**	**Compare two 32-bit portions of D registers for equal/greater or equal/greater/not equal/less than/less or equal, updating the result in two predicates. "<*op*>" is one of: EQ, GT, GE, NE, LE, LT in the syntax, and one of ==, >, ≥, != ,≤ ,< in the operation, respectively. Aliased variants are LE, LT.**

	`if (Db.M  <op>  Da.M) {`
	`        Pm = 1`
	`        Pn = 0`
	`} else {`
	`        Pm = 0`
	`        Pn = 1`
	`}`	

Table continues on the next page...

#	Syntax	Description
6	**CMP.*<op>*.L Da,Db,Pn**	**Compare two 32-bit portions of D registers for equal/greater or equal/greater/not equal/less than/less or equal, updating the result in one predicate. "*<op>*" is one of: EQ, GT, GE, NE, LE, LT in the syntax, and one of ==, >, ≥, != ,≤ ,< in the operation, respectively. Aliased variants are LE, LT.**

	`if (Db.M  <op>  Da.M) {`
	`        Pn = 1`
	`} else {`
	`        Pn = 0`
	`}`	

7	**CMP.*<op>*.U.L Da,Db,Pm:Pn**	**Compare two 32-bit portions of D registers for unsigned for greater/greater or equal/less than/ less or equal, updating the result in two predicates. "*<op>*" is one of: GT, GE, LE, LT in the syntax, and one of >, ≥ ,≤ ,< in the operation, respectively. Aliased variants are LE, LT.**

	`src1 = (U40)Da.M ; src2 = (U40)Db.M`
	`if (src2 <op> (src1)) {`
	`        Pm = 1`
	`        Pn = 0`
	`} else {`
	`        Pm = 0`
	`        Pn = 1`
	`}`	

8	**CMP.*<op>*.U.L Da,Db,Pn**	**Compare two 32-bit portions of D registers for unsigned for greater/greater or equal/less than/ less or equal, updating the result in one predicate. "*<op>*" is one of: GT, GE, LE, LT in the syntax, and one of >, ≥ ,≤ ,< in the operation, respectively. Aliased variants are LE, LT.**

	`src1 = (U40)Da.M ; src2 = (U40)Db.M`
	`if (src2 <op> (src1)) {`
	`        Pn = 1`
	`} else {`
	`        Pn = 0`
	`}`	

9	**CMP.*<op>*.U.L #u32,Da,Pm:Pn**	**Compare Da.M with a 32-bit immediate for unsigned for greater/less or equal, updating the result in two predicates. "*<op>*" is one of: GT, LE in the syntax, and one of >, ≤ in the operation, respectively.**

	`src1 = (U33)Da.M ; imm = (U33)u32`
	`if (src1 <op> (imm)) {`
	`        Pm = 1`
	`        Pn = 0`
	`} else {`
	`        Pm = 0`
	`        Pn = 1`
	`}`	

Table continues on the next page...

#	Syntax	Description
10	**CMP.<*op*>.U.L #u32,Da,Pn**	**Compare Da.M with a 32-bit immediate for unsigned for greater/less or equal, updating the result in one predicate. "<*op*>" is one of: GT, LE in the syntax, and one of >, ≤ in the operation, respectively.**

```
src1 = (U33)Da.M ; imm = (U33)u32
if (src1 <op> (imm)) {
        Pn = 1
} else {
        Pn = 0
}
```

11	**CMP.<*op*>.U.L #u5,Da,Pm:Pn**	**Compare Da.M with a short immediate for unsigned for greater/less or equal, updating the result in two predicates. "<*op*>" is one of: GT, LE in the syntax, and one of >, ≤ in the operation, respectively.**

```
src1 = (U33)Da.M ; imm = (U33)u5
if (src1 <op> (imm)) {
        Pm = 1
        Pn = 0
} else {
        Pm = 0
        Pn = 1
}
```

12	**CMP.<*op*>.U.L #u5,Da,Pn**	**Compare D with a short immediate for unsigned for greater/less or equal, updating the result in one predicate. "<*op*>" is one of: GT, LE in the syntax, and one of >, ≤ in the operation, respectively.**

```
src1 = (U33)Da.M ; imm = (U33)u5
if (src1 <op> (imm)) {
        Pn = 1
} else {
        Pn = 0
}
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Db	D0-D63	
Pm:Pn	$P_n:P_{n+1}$	$0 \le n \le 4$
Pn	P0-P5	
s32	$-2^{31} \le s32 < 2^{31}$	
u32	$0 \le u32 < 2^{32}$	
u5	$0 \le u5 < 2^{5}$	
ux5	$-1 \le ux5 \le 30$	

Instruction Attributes

Attribute	Value	Relevant Variant #
Alias		5, 6, 7, 8
Encoding length	32-bits in 64	3, 4, 5, 6, 7, 8, 11, 12
	64-bits	1, 2, 9, 10
IF.EC		All
Pipeline behavior	dalu_DAU_Pbit	All

CMP.nW — Compare 16 Bits in a Data Register — (DALU)

General Description

Perform a signed or unsigned comparison of the lower 16-bit data of a D register with another 16-bit operand - either in another D register or in an immediate value. The comparison type could be one of several conditions: equality (EQ), inequality (NE), greater than (GT), greater or equal (GE), less than (LT), and less or equal (LE). The condition is specified in position FLG1 of the mnemonic. In the variant table below, instructions that operate on the same sources and update the same destinations are described collectively under one entry, where the condition is generically specified as . The functional semantics for evaluating conditions for which order matters is from right to left, for example CMP.GT Da,Db is true if Db>Da. Not all instructions support all condition tests, so the exact substitution list for is written in the description of each table entry. By default, comparisons are signed unless specified as unsigned (in FLG2 position of the mnemonic). Variants with a single predicate destination will capture the test result in it. Variants which update a predicate pair will update the first predicate with the test result and the inverse result in the second predicate. As with other compare and test instructions, in some cases several instructions updating the same predicates may be grouped together in the same VLES, in which case the test result is the logical OR of the individual tests. The instruction may be in itself predicated by IF.Pn or IF.Pn.EC conditions. For more information on the semantics of predicate updating instructions, see the Program Control chapter.

Flag Options

Flag	Position	Description
EQ	flg1	Equal comparison
GE	flg1	Greater or Equal comparison
GT	flg1	Greater Than comparison
NE	flg1	Not Equal comparison
U	flg2	Unsigned

Instruction Variants

#	Syntax	Description
1	**CMP.<*op*>.W Da.l,Db.l,Pm:Pn**	**Compare two 16-bit portions of D registers for equal/greater or equal/greater/not equal, updating the result in two predicates. "<*op*>" is one of: EQ, GT, GE, NE in the syntax, and one of ==, >, ≥, != in the operation, respectively.**

```
if (Db.L  <op> (Da.L)) {
        Pm = 1
        Pn = 0
} else {
        Pm = 0
        Pn = 1
}
```

Table continues on the next page...

<table>
<tr><th>#</th><th>Syntax</th><th>Description</th></tr>
<tr><td>2</td><td><b>CMP.&lt;<i>op</i>&gt;.W Da.l,Db.l,Pn</b></td><td><b>Compare two 16-bit portions of D registers for equal/greater or equal/greater/not equal, updating the result in one predicate. "&lt;<i>op</i>&gt;" is one of: EQ, GT, GE, NE in the syntax, and one of ==, >, ≥, != in the operation, respectively.</b></td></tr>
<tr><td></td><td colspan="2"><pre>
if (Db.L <op> (Da.L)) {
        Pn = 1
} else {
        Pn = 0
}
</pre></td></tr>
<tr><td>3</td><td><b>CMP.&lt;<i>op</i>&gt;.U.W Da.l,Db.l,Pm:Pn</b></td><td><b>Compare two 16-bit portions of D registers for unsigned for greater/greater or equal, updating the result in two predicates. "&lt;<i>op</i>&gt;" is one of: GT, GE in the syntax, and one of >, ≥ in the operation, respectively.</b></td></tr>
<tr><td></td><td colspan="2"><pre>
if (Db.L <op> Da.L) {
        Pm = 1
        Pn = 0
} else {
        Pm = 0
        Pn = 1
}
</pre></td></tr>
<tr><td>4</td><td><b>CMP.&lt;<i>op</i>&gt;.U.W Da.l,Db.l,Pn</b></td><td><b>Compare two 16-bit portions of D registers for unsigned for greater/greater or equal, updating the result in one predicate. "&lt;<i>op</i>&gt;" is one of: GT, GE in the syntax, and one of >, ≥ in the operation, respectively.</b></td></tr>
<tr><td></td><td colspan="2"><pre>
if (Db.L <op> Da.L) {
        Pn = 1
} else {
        Pn = 0
}
</pre></td></tr>
</table>

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Db	D0-D63	
Pm:Pn	P_n:P_{n+1}	$0 \le n \le 4$
Pn	P0-P5	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
IF.EC		All
Pipeline behavior	dalu_DAU_Pbit	All

CMP.nX Compare 40 Bits in a Data Register (DALU)

General Description

Performs a signed or unsigned comparison of the 40-bit data of a D register with another 40-bit operand - either in another D register or in an immediate value. The comparison type could be one of several conditions: equality (EQ), inequality (NE), greater than (GT), greater or equal (GE), less than (LT), and less or equal (LE). The condition is specified in position FLG1 of the mnemonic. In the variant table below, instructions that operate on the same sources and update the same destinations are described collectively under one entry, where the condition is generically specified as <op>. The functional semantics for evaluating conditions involving two registers for which order matters is from right to left, for example CMP.GT Da,Db is true if Db>Da. Not all instructions support all condition tests, so the exact substitution list is written in the description of each table entry. By default, comparisons are signed unless specified as unsigned (in FLG2 position of the mnemonic). Variants with a single predicate destination will capture the test result in it. Variants which update a predicate pair will update the first predicate with the test result and the inverse result in the second predicate. As with other compare and test instructions, in some cases several instructions updating the same predicates may be grouped together in the same VLES, in which case the test result is the logical OR of the individual tests. The instruction may be in itself predicated by IF.Pn or IF.Pn.EC conditions. For more information on the semantics of predicate updating instructions, see the Program Control chapter.

Flag Options

Flag	Position	Description
EQ	flg1	Equal comparison
GE	flg1	Greater or Equal comparison
GT	flg1	Greater Than comparison
LE	flg1	Less or Equal comparison
LT	flg1	Less Than comparison
NE	flg1	Not Equal comparison
U	flg2	Unsigned

Instruction Variants

#	Syntax	Description
1	**CMP.<*op*>.X #s32,Da,Pm:Pn**	**Compare a D register with an immediate for equal/greater/less or equal/not equal, updating the result in two predicates. "<*op*>" is one of: EQ, GT, LE, NE in the syntax, and one of ==, >, ≤, != in the operation, respectively.**

```
if (Da <op> (S40)s32) {
        Pm = 1
        Pn = 0
} else {
        Pm = 0
        Pn = 1
}
```

Table continues on the next page...

#	Syntax	Description
2	CMP.<*op*>.X #s32,Da,Pn	Compare a D register with an immediate for equal/greater/less or equal/not equal, updating the result in one predicate. "<*op*>" is one of: EQ, GT, LE, NE in the syntax, and one of ==, >, ≤, != in the operation, respectively.

	`if (Da <op> (S40)s32) {`
	`        Pn = 1`
	`} else {`
	`        Pn = 0`
	`}`	

3	CMP.<*op*>.X #ux5,Da,Pm:Pn	Compare a D register with a short immediate for equal/greater/less or equal/not equal, updating the result in two predicates. "<*op*>" is one of: EQ, GT, LE, NE in the syntax, and one of ==, >, ≤, != in the operation, respectively.

	`imm = (ux5 == 31) ? (- 1) : ux5`
	`if (Da <op> imm) {`
	`        Pm = 1`
	`        Pn = 0`
	`} else {`
	`        Pm = 0`
	`        Pn = 1`
	`}`	

4	CMP.<*op*>.X #ux5,Da,Pn	Compare a D register with a short immediate for equal/greater/less or equal/not equal, updating the result in one predicate. "<*op*>" is one of: EQ, GT, LE, NE in the syntax, and one of ==, >, ≤, != in the operation, respectively.

	`imm = (ux5 == 31) ? (- 1) : ux5`
	`if (Da <op> imm) {`
	`        Pn = 1`
	`} else {`
	`        Pn = 0`
	`}`	

5	CMP.<*op*>.X Da,Db,Pm:Pn	Compare two D registers for equal/greater or equal/greater/not equal/less than/less or equal, updating the result in two predicates. "<*op*>" is one of: EQ, GT, GE, NE, LE, LT in the syntax, and one of ==, >, ≥, != ,≤ ,< in the operation, respectively. Aliased variants are LE, LT.

	`if (Db <op> (Da)) {`
	`        Pm = 1`
	`        Pn = 0`
	`} else {`
	`        Pm = 0`
	`        Pn = 1`
	`}`	

Table continues on the next page...

#	Syntax	Description
6	**CMP.<*op*>.X Da,Db,Pn**	**Compare two D registers for equal/greater or equal/greater/not equal/less than/less or equal, updating the result in one predicate. "<*op*>" is one of: EQ, GT, GE, NE, LE, LT in the syntax, and one of ==, >, ≥, != ,≤ ,< in the operation, respectively. Aliased variants are LE, LT.**

	`if (Db <op> (Da)) {`
	`        Pn = 1`
	`} else {`
	`        Pn = 0`
	`}`	

7	**CMP.<*op*>.U.X Da,Db,Pm:Pn**	**Compare two D registers for signed for greater/ greater or equal, updating the result in two predicates. "<*op*>" is one of: GT, GE in the syntax, and one of >, ≥ in the operation, respectively.**

	`if (Db <op> Da) {`
	`        Pm = 1`
	`        Pn = 0`
	`} else {`
	`        Pm = 0`
	`        Pn = 1`
	`}`	

8	**CMP.<*op*>.U.X Da,Db,Pn**	**Compare two D registers for signed for greater/ greater or equal, updating the result in one predicate. "<*op*>" is one of: GT, GE in the syntax, and one of >, ≥ in the operation, respectively.**

	`if (Db <op> Da) {`
	`        Pn = 1`
	`} else {`
	`        Pn = 0`
	`}`	

9	**CMP.<*op*>.U.X #u32,Da,Pm:Pn**	**Compare a D register with an immediate for unsigned for greater/less or equal, updating the result in two predicates. "<*op*>" is one of: GT, LE in the syntax, and one of >, ≤ in the operation, respectively.**

	`src1 = Da ; imm = u32`
	`if (src1 <op> (imm)) {`
	`        Pm = 1`
	`        Pn = 0`
	`} else {`
	`        Pm = 0`
	`        Pn = 1`
	`}`	

Table continues on the next page...

#	Syntax	Description
10	**CMP.<*op*>.U.X #u32,Da,Pn**	**Compare a D register with an immediate for unsigned for greater/less or equal, updating the result in one predicate. "<*op*>" is one of: GT, LE in the syntax, and one of >, ≤ in the operation, respectively.**

```
src1 = Da ; imm = u32
if (src1 <op> (imm)) {
        Pn = 1
} else {
        Pn = 0
}
```

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Db	D0-D63
Pm:Pn	$P_n:P_{n+1}$ $0 \le n \le 4$
Pn	P0-P5
s32	$-2^{31} \le s32 < 2^{31}$
u32	$0 \le u32 < 2^{32}$
ux5	$-1 \le ux5 \le 30$

Instruction Attributes

Attribute	Value	Relevant Variant #
Alias		5, 6
Architecture	SC3900FP only	9, 10
Encoding length	32-bits in 64	3, 4, 5, 6, 7, 8
	64-bits	1, 2, 9, 10
IF.EC		All
Pipeline behavior	dalu_DAU_Pbit	All

CMPA Compare 32 Bits in an Address Register (LSU,IPU)

General Description

Performs a signed or unsigned comparison of 32-bit data of an address register with another 32-bit operand - either in another R register or in an immediate value. The comparison type could be one of several conditions: equality (EQ), inequality (NE), greater than (GT), greater or equal (GE), less than (LT), and less or equal (LE). The condition is specified in position FLG1 of the mnemonic. In the variant table below, instructions that operate on the same sources and update the same destinations are described collectively under one entry, where the condition is generically specified as <op>. The functional semantics for evaluating conditions involving two registers for which order matters is from right to left, for example CMPA.GT Ra,Rb is true if Rb>Ra. Not all instructions support all condition tests, so the exact substitution list is written in the description of each table entry. By default, comparisons are signed unless specified as unsigned (in FLG2 position of the mnemonic). Variants with a single predicate destination will capture the test result in it. Variants which update a predicate pair will update the first predicate with the test result and the inverse result in the second predicate. As with other compare and test instructions, in some cases several instructions updating the same predicates may be grouped together in the same VLES, in which case the test result is the logical OR of the individual tests. The instruction may be in itself predicated by IF.Pn or IF.Pn.EC conditions. For more information on the semantics of predicate updating instructions, see the Program Control chapter.

Flag Options

Flag	Position	Description
EQ	flg1	Equal comparison
GE	flg1	Greater or Equal comparison
GT	flg1	Greater Than comparison
LE	flg1	Less or Equal comparison
LT	flg1	Less Than comparison
NE	flg1	Not Equal comparison
U	flg2	Unsigned

Instruction Variants

#	Syntax	Description
1	**CMPA.<*op*> #s16,Ra,Pm:Pn**	**Compare an R register with an immediate for equal/greater/less or equal/not equal, updating the result in two predicates. "<*op*>" is one of: EQ, GT, LE, NE in the syntax, and one of ==, >, ≤, != in the operation, respectively.**

```
if (Ra <op> s16) {
        Pm = 1
        Pn = 0
} else {
        Pm = 0
        Pn = 1
}
```

Table continues on the next page...

#	Syntax	Description
2	**CMPA.<*op*> #s16,Ra,Pn**	**Compare an R register with an immediate for equal/greater/less or equal/not equal, updating the result in one predicate. "<*op*>" is one of: EQ, GT, LE, NE in the syntax, and one of ==, >, ≤, != in the operation, respectively.**

	if (Ra <*op*> s16) {
	        Pn = 1
	} else {
	        Pn = 0
	}	

3	**CMPA.<*op*> #ux5,Ra,Pm:Pn**	**Compare an R register with a short immediate for equal/greater/less or equal/not equal, updating the result in two predicates. "<*op*>" is one of: EQ, GT, LE, NE in the syntax, and one of ==, >, ≤, != in the operation, respectively.**

	imm = (ux5 <*op*> 31) ? (- 1) : ux5
	if (Ra <*op*> imm) {
	        Pm = 1
	        Pn = 0
	} else {
	        Pm = 0
	        Pn = 1
	}	

4	**CMPA.<*op*> #ux5,Ra,Pn**	**Compare an R register with a short immediate for equal/greater/less or equal/not equal, updating the result in one predicate. "<*op*>" is one of: EQ, GT, LE, NE in the syntax, and one of ==, >, ≤, != in the operation, respectively.**

	imm = (ux5 <*op*> 31) ? (- 1) : ux5
	if (Ra <*op*> imm) {
	        Pn = 1
	} else {
	        Pn = 0
	}	

5	**CMPA.<*op*> Ra,Rb,Pm:Pn**	**Compare two R registers for equal/greater or equal/greater/not equal/less than/less or equal, updating the result in two predicates. "<*op*>" is one of: EQ, GT, GE, NE, LE, LT in the syntax, and one of ==, >, ≥, != ,≤ ,< in the operation, respectively. Aliased variants are LE, LT.**

	if (Rb <*op*> (Ra)) {
	        Pm = 1
	        Pn = 0
	} else {
	        Pm = 0
	        Pn = 1
	}	

Table continues on the next page...

#	Syntax	Description
6	**CMPA.<*op*> Ra,Rb,Pn**	**Compare two R registers for equal/greater or equal/greater/not equal/less than/less or equal, updating the result in one predicate. "<*op*>" is one of: EQ, GT, GE, NE, LE, LT in the syntax, and one of ==, >, ≥, != ,≤ ,< in the operation, respectively. Aliased variants are LE, LT.**

	if (Rb <*op*> (Ra)) {
	        Pn = 1
	} else {
	        Pn = 0
	}	

7	**CMPA.<*op*>.U Ra,Rb,Pm:Pn**	**Compare two R registers for unsigned for greater/greater or equal/less than/less or equal, updating the result in two predicates. "<*op*>" is one of: GT, GE, LE, LT in the syntax, and one of >, ≥ ,≤ ,< in the operation, respectively. Aliased variants are LE, LT.**

	val1 = (U33)Rb ; val2 = (U33)Ra
	if (val1 <*op*> (val2)) {
	        Pm = 1
	        Pn = 0
	} else {
	        Pm = 0
	        Pn = 1
	}	

8	**CMPA.<*op*>.U Ra,Rb,Pn**	**Compare two R registers for unsigned for greater/greater or equal/less than/less or equal, updating the result in one predicate. "<*op*>" is one of: GT, GE, LE, LT in the syntax, and one of >, ≥ ,≤ ,< in the operation, respectively. Aliased variants are LE, LT.**

	val1 = (U33)Rb ; val2 = (U33)Ra
	if (val1 <*op*> (val2)) {
	        Pn = 1
	} else {
	        Pn = 0
	}	

9	**CMPA.<*op*>.U #u16,Ra,Pm:Pn**	**Compare an R register with an immediate for unsigned for greater/less or equal, updating the result in two predicates. "<*op*>" is one of: GT, LE in the syntax, and one of >, ≤ in the operation, respectively.**

	src1 = (U33)Ra ; imm = (U33)u16
	if (src1 <*op*> (imm)) {
	        Pm = 1
	        Pn = 0
	} else {
	        Pm = 0
	        Pn = 1
	}	

Table continues on the next page...

#	Syntax	Description
10	**CMPA.<*op*>.U #u16,Ra,Pn**	**Compare an R register with an immediate for unsigned for greater/less or equal, updating the result in one predicate. "<*op*>" is one of: GT, LE in the syntax, and one of >, ≤ in the operation, respectively.**

```
src1 = (U33)Ra ; imm = (U33)u16
if (src1 <op> (imm)) {
        Pn = 1
} else {
        Pn = 0
}
```

#	Syntax	Description
11	**CMPA.<*op*>.U #u5,Ra,Pm:Pn**	**Compare an R register with a short immediate for unsigned for greater/less or equal, updating the result in two predicates. "<*op*>" is one of: GT, LE in the syntax, and one of >, ≤ in the operation, respectively.**

```
src1 = (U33)Ra ; src2 = (U33)u5
if (src1 <op> (src2)) {
        Pm = 1
        Pn = 0
} else {
        Pm = 0
        Pn = 1
}
```

#	Syntax	Description
12	**CMPA.<*op*>.U #u5,Ra,Pn**	**Compare an R register with a short immediate for unsigned for greater/less or equal, updating the result in one predicate. "<*op*>" is one of: GT, LE in the syntax, and one of >, ≤ in the operation, respectively.**

```
src1 = (U33)Ra ; src2 = (U33)u5
if (src1 <op> (src2)) {
        Pn = 1
} else {
        Pn = 0
}
```

Explicit Operands

Operand	Permitted Values
Pm:Pn	$P_n:P_{n+1}$ $0 \le n \le 4$
Pn	P0-P5
Ra	R0-R31
Rb	R0-R31
s16	$-2^{15} \le s16 < 2^{15}$
u16	$0 \le u16 < 2^{16}$
u5	$0 \le u5 < 2^{5}$
ux5	$-1 \le ux5 \le 30$

Instruction Attributes

Attribute	Value	Relevant Variant #
Alias		5, 6, 7, 8
Architecture	SC3900FP only	11, 12
Encoding length	32-bits	3, 4, 5, 6, 7, 8, 11, 12
	48-bits	1, 2, 9, 10
Execution unit	IPU	All
	LSU	All
IF.EC		All
Pipeline behavior	agu_AAU_Pbit	All

CMPD.nT Compare 20-bit Operands to a Data Register Destination (DALU)

General Description

Performs a signed SIMD2 comparison of two 20-bit operands packed in a D register (WH and WL), with another pair of packed 20-bit operands. SIMD4 variants operate on two independent source-destination register sets. The result of the comparison is written to a destination D register and not to a predicate. A true result writes a value of all-ones to a destination D register portion of the same width (WH or WL), and a false results writes an all-zero value. This convention allows to calculate a complex condition by a series of logic instructions performed in series on data registers, saving the need to consume and manipulate predicates. The comparison type could be one of several conditions: equality (EQ), inequality (NE), greater than (GT), greater or equal (GE), less than (LT), and less or equa (LE). The condition is specified in position FLG1 of the mnemonic. In the variant table below, instructions that operate on the same sources and update the same destinations are described collectively under one entry, where the condition is generically specified as <op>. The functional semantics for evaluating conditions involving two registers for which order matters is from right to left, for example CMPD.GT Da,Db is true if for the respective 20-bit portion Db>Da.

Flag Options

Flag	Position	Description
EQ	flg1	Equal comparison
GE	flg1	Greater or Equal comparison
GT	flg1	Greater Than comparison
LE	flg1	Less or Equal comparison
LT	flg1	Less Than comparison
NE	flg1	Not Equal comparison

Instruction Variants

#	Syntax	Description
1	**CMPD.<*op*>.2T Da,Db,Dn**	**SIMD2 compare for equal/greater or equal/ greater/not equal/less than/less or equal into a data register. "<*op*>" is one of: EQ, GT, GE, NE, LE, LT in the syntax, and one of ==, >, ≥, != ,≤ ,< in the operation, respectively. Aliased variants are LE, LT.**

	`Dn.WH = (Db.WH  <op>  Da.WH) ? 0xFFFFF : 0x00000`
	`Dn.WL = (Db.WL  <op>  Da.WL) ? 0xFFFFF : 0x00000`	

2	**CMPD.<*op*>.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 compare for equal/greater or equal/ greater/not equal/less than/less or equal into two data registers. "<*op*>" is one of: EQ, GT, GE, NE, LE, LT in the syntax, and one of ==, >, ≥, != ,≤ ,< in the operation, respectively. Aliased variants are LE, LT.**

	`Dm.WH = (Dc.WH  <op>  Da.WH) ? 0xFFFFF : 0x00000`
	`Dm.WL = (Dc.WL  <op>  Da.WL) ? 0xFFFFF : 0x00000`
	`Dn.WH = (Dd.WH  <op>  Db.WH) ? 0xFFFFF : 0x00000`
	`Dn.WL = (Dd.WL  <op>  Db.WL) ? 0xFFFFF : 0x00000`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Alias		All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

CMPD.nW Compare 16-bit Operands to a Data Register Destination (DALU)

General Description

Performs a signed SIMD2 comparison of two 16-bit operands packed in a D register (H and L), with another pair of packed 16-bit operands. SIMD4 variants operate on two independent source-destination register sets. The result of the comparison is written to a destination D register and not to a predicate. A true result writes a value of all-ones to a destination D register portion, sign-extending it to 20 bits (WH or WL), and a false results writes an all-zero value. This convention allows to calculate a complex condition by a series of logic instructions performed in series on data registers, saving the need to consume and manipulate predicates. The comparison type could be one of several conditions: equality (EQ), inequality (NE), greater than (GT), greater or equal (GE), less than (LT), and less or equal (LE). The condition is specified in position FLG1 of the mnemonic. In the variant table below, instructions that operate on the same sources and update the same destinations are described collectively under one entry, where the condition is generically specified as <op>. The functional semantics for evaluating conditions involving two registers for which order matters is from right to left, for example CMPD.GT Da,Db is true if for the respective 16-bit portion Db>Da.

Flag Options

Flag	Position	Description
EQ	flg1	Equal comparison
GE	flg1	Greater or Equal comparison
GT	flg1	Greater Than comparison
LE	flg1	Less or Equal comparison
LT	flg1	Less Than comparison
NE	flg1	Not Equal comparison

Instruction Variants

#	Syntax	Description
1	**CMPD.<*op*>.2W Da,Db,Dn**	**SIMD2 compare for equal/greater or equal/greater/not equal/less than/less or equal into a data register. "<*op*>" is one of: EQ, GT, GE, NE, LE, LT in the syntax, and one of ==, >, ≥, != ,≤ ,< in the operation, respectively. Aliased variants are LE, LT.**

	`Dn.WH = (Db.H  <op>  Da.H) ? 0xfffff : 0x00000`
	`Dn.WL = (Db.L  <op>  Da.L) ? 0xfffff : 0x00000`	

2	**CMPD.<*op*>.4W Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 compare for equal/greater or equal/greater/not equal/less than/less or equal into two data registers. "<*op*>" is one of: EQ, GT, GE, NE, LE, LT in the syntax, and one of ==, >, ≥, != ,≤ ,< in the operation, respectively. Aliased variants are LE, LT.**

	`Dm.WH = (Dc.H  <op>  Da.H) ? 0xfffff : 0x00000`
	`Dm.WL = (Dc.L  <op>  Da.L) ? 0xfffff : 0x00000`
	`Dn.WH = (Dd.H  <op>  Db.H) ? 0xfffff : 0x00000`
	`Dn.WL = (Dd.L  <op>  Db.L) ? 0xfffff : 0x00000`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Alias		All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

CMPD.nX Compare 40-bit Operands to a Data Register Destination (DALU)

General Description

Performs a signed comparison of two 40-bit operands in a D register, with another 40-bit operand (immediate or in a register). SIMD2 variants operate on two independent source-destination register sets. The result of the comparison is written to a destination D register and not to a predicate. A true result writes a value of all-ones to a destination D register, and a false results writes an all-zero value.
This convention allows to calculate a complex condition by a series of logic instructions performed in series on data registers, saving the need to consume and manipulate predicates. The comparison type could be one of several conditions: equality (EQ), inequality (NE), greater than (GT), greater or equal (GE), less than (LT) and less or equal (LE). The condition is specified in position FLG1 of the mnemonic. In the variant table below, instructions that operate on the same sources and update the same destinations are described collectively under one entry, where the condition is generically specified as <op>. Not all instructions support all condition tests, so the exact substitution list for is written in the description of each table entry. The functional semantics for evaluating conditions involving two registers for which order matters is from right to left, for example CMPD.GT Da,Db is true if Db>Da.

Flag Options

Flag	Position	Description
EQ	flg1	Equal comparison
GE	flg1	Greater or Equal comparison
GT	flg1	Greater Than comparison
LE	flg1	Less or Equal comparison
LT	flg1	Less Than comparison
NE	flg1	Not Equal comparison

Instruction Variants

#	Syntax	Description
1	**CMPD.<*op*>.X #s32,Da,Dn**	**Compare for equal/greater or equal/greater/less or equal/less/not equal of a register with an immediate value into a data register. "<*op*>" is one of: EQ, GT, GE, LE, LT, NE in the syntax, and one of ==, >, ≥, ≤, <, != in the operation, respectively.**

```
if (Da <op> (S40)s32) {
        Dn = 0xffffffffff
} else {
        Dn = 0
}
```

Table continues on the next page...

#	Syntax	Description
2	**CMPD.<*op*>.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 compare for equal/greater or equal/greater/not equal/less than/less or equal of two pairs of registers into two data registers. "<*op*>" is one of: EQ, GT, GE, NE, LE, LT in the syntax, and one of ==, >, ≥, != ,≤ ,< in the operation, respectively. Aliased variants are LE, LT.**

```
if (Dc <op> (Da)) {
        Dm = 0xffffffffff
} else {
        Dm = 0
}
if (Dd <op> (Db)) {
        Dn = 0xffffffffff
} else {
        Dn = 0
}
```

#	Syntax	Description
3	**CMPD.<*op*>.X Da,Db,Dn**	**Compare for equal/greater or equal/greater/not equal/less than/less or equal of two registers into a data register. "<*op*>" is one of: EQ, GT, GE, NE, LE, LT in the syntax, and one of ==, >, ≥, != ,≤ ,< in the operation, respectively. Aliased variants are LE, LT.**

```
if (Db <op> (Da)) {
        Dn = 0xffffffffff
} else {
        Dn = 0
}
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	
s32	$-2^{31} \le s32 < 2^{31}$	

Instruction Attributes

Attribute	Value	Relevant Variant #
Alias		2, 3
Encoding length	32-bits in 64	2, 3
	64-bits	1
Pipeline behavior	dalu_DAU	All

CNEG.nB Conditional Negate Four Packed Bytes (DALU)

General Description

Selectively copies or negates four bytes from a source data register.

Selection is done according to a nibble (4 bits) that is extracted from the first source operand using an immediate offset (with a value one of of 0, 4, 8, 12, 16, 20, 24, 28). Each bit in the nibble controls the selection of the corresponding byte in the second source operand: If it is set the byte is negated, and if cleared the byte is copied. When negated, the byte is saturated to 0x7F if the original value was 0x80.

The order of the controlling bits is reversed for the REV flavor.

Flag Options

Flag	Position	Description
N	flg1	Negative polarity
REV	flg2	Reverse bit order

Instruction Variants

#	Syntax	Description
1	**CNEG.N.4B #u3_2,Da,Db,Dn**	**Conditional Negate four packed bytes (negative polarity)**

```
Dn.HH = Da[<offset>+3] ? SAT8(-Db.HH) : (Db.HH)
Dn.HL = Da[<offset>+2] ? SAT8(-Db.HL) : (Db.HL)
Dn.LH = Da[<offset>+1] ? SAT8(-Db.LH) : (Db.LH)
Dn.LL = Da[<offset>+0] ? SAT8(-Db.LL) : (Db.LL)
Dn.E = 0
u3_2 = offset, valid values are: (0,4,8,12,16,20,24,28)
```

#	Syntax	Description
2	**CNEG.N.REV.4B #u3_2,Da,Db,Dn**	**Conditional Negate four packed bytes (negative polarity) - Reversed**

```
Dn.HH = Da[<offset>+0] ? SAT8(-Db.HH) : (Db.HH)
Dn.HL = Da[<offset>+1] ? SAT8(-Db.HL) : (Db.HL)
Dn.LH = Da[<offset>+2] ? SAT8(-Db.LH) : (Db.LH)
Dn.LL = Da[<offset>+3] ? SAT8(-Db.LL) : (Db.LL)
Dn.E = 0
u3_2 = offset, valid values are: (0,4,8,12,16,20,24,28)
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Db	D0-D63	
Dn	D0-D63	
u3_2	$0 \le u3\_2 < 2^5$;	u3_2 & 0x3 == 0

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

CNEG.nT Conditional Negate Packed 20-bit Wide Words (DALU)

General Description

Selectively copies or negates 20-bit portions from a data register or data register pair.

In variants without the SGN flag, selection is done according to a nibble (4 bits) that is extracted from the first source operand using an immediate offset (with a value one of of 0, 4, 8, 12, 16, 20, 24, 28). Each bit in the nibble controls the selection of the corresponding 20-bit portion in the second source operand (a register or register pair): If it is set the value is negated, and if cleared the value is copied. The order of the controlling bits is reversed for the REV variant.

In variants wiht the SGN flag - The sign bits of the 20-bit packed words of the first operand (a register or register pair) are used to control the negation of the 20-bit operands in the corresponding positions in the second source operand (register or register pair): If sign of the high 20-bit portion is set then the high 20-bit portion of the second source is negated and if cleared the high 20-bit word is copied, and similarly for the low 20-portion.

Flag Options

Flag	Position	Description
N	flg1	Negative polarity
REV	flg2	Reverse bit order
SGN	flg2	Use the sign bits as the negation condition

Instruction Variants

#	Syntax	Description
1	**CNEG.N.4T #u3_2,Da,Dc:Dd,Dm:Dn**	**Conditional Negate four packed 20-bit words (negative polarity)**

	Dm.WH = ((Da[u3_2+3] == 0) ? (S21)(Dc.WH) : (-(S21)(Dc.WH)))[19:0]
	Dm.WL = ((Da[u3_2+2] == 0) ? (S21)(Dc.WL) : (-(S21)(Dc.WL)))[19:0]
	Dn.WH = ((Da[u3_2+1] == 0) ? (S21)(Dd.WH) : (-(S21)(Dd.WH)))[19:0]
	Dn.WL = ((Da[u3_2] == 0) ? (S21)(Dd.WL) : (-(S21)(Dd.WL)))[19:0]	

2	**CNEG.N.REV.4T #u3_2,Da,Dc:Dd,Dm:Dn**	**Conditional Negate four packed 20-bit words (negative polarity) - reversed order**

	Dm.WH = ((Da[u3_2+0] == 0) ? (S21)(Dc.WH) : (-(S21)(Dc.WH)))[19:0]
	Dm.WL = ((Da[u3_2+1] == 0) ? (S21)(Dc.WL) : (-(S21)(Dc.WL)))[19:0]
	Dn.WH = ((Da[u3_2+2] == 0) ? (S21)(Dd.WH) : (-(S21)(Dd.WH)))[19:0]
	Dn.WL = ((Da[u3_2+3] == 0) ? (S21)(Dd.WL) : (-(S21)(Dd.WL)))[19:0]	

3	**CNEG.N.SGN.2T Da,Db,Dn**	**Conditional Negate two 20-bit (negative polarity)**

	Dn.WL = (Da[35] == 1) ? (-Db.WL) : Db.WL
	Dn.WH = (Da[39] == 1) ? (-Db.WH) : Db.WH	

4	**CNEG.N.SGN.4T Da:Db,Dc:Dd,Dm:Dn**	**Conditional Negate four 20-bit (negative polarity)**

	Dm.WH = (Da[39] == 1) ? (-Dc.WH) : Dc.WH
	Dm.WL = (Da[35] == 1) ? (-Dc.WL) : Dc.WL
	Dn.WH = (Db[39] == 1) ? (-Dd.WH) : Dd.WH
	Dn.WL = (Db[35] == 1) ? (-Dd.WL) : Dd.WL	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	
u3_2	$0 \le$ u3_2 $< 2^5$;	u3_2 & 0x3 == 0

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	3, 4
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

CNEG.nW Conditional Negate Four Packed 16-bit Words (DALU)

General Description

Selectively copies or negates four 16-bit words from a data register pair.

Selection is done according to a nibble (4 bits) that is extracted from the first source operand using an immediate offset (with a value one of of 0, 4, 8, 12, 16, 20, 24, 28). Each bit in the nibble controls the selection of the corresponding 16-bit word in the source register pair: If it is set the word is negated, and if cleared the word is copied. The order of the controlling bits is reversed for the REV variant.

Flag Options

Flag	Position	Description
N	flg1	Negative polarity
REV	flg2	Reverse bit order

Instruction Variants

#	Syntax	Description
1	**CNEG.N.4W #u3_2,Da,Dc:Dd,Dm:Dn**	**Conditional Negate four packed 16-bit words (negative polarity)**

	`Dm.H = SAT16 ((S40)((Da[u3_2+3] == 0) ? (Dc.H) : (-(Dc.H))))`
	`Dm.L = SAT16 ((S40)((Da[u3_2+2] == 0) ? (Dc.L) : (-(Dc.L))))`
	`Dn.H = SAT16 ((S40)((Da[u3_2+1] == 0) ? (Dd.H) : (-(Dd.H))))`
	`Dn.L = SAT16 ((S40)((Da[u3_2] == 0) ? (Dd.L) : (-(Dd.L))))`
	`Dm.E = 0`
	`Dn.E = 0`	

2	**CNEG.N.REV.4W #u3_2,Da,Dc:Dd,Dm:Dn**	**Conditional Negate four packed 16-bit words (negative polarity) - reversed order**

	`Dm.H = SAT16 ((S40)((Da[u3_2+0] == 0) ? (Dc.H) : (-(Dc.H))))`
	`Dm.L = SAT16 ((S40)((Da[u3_2+1] == 0) ? (Dc.L) : (-(Dc.L))))`
	`Dn.H = SAT16 ((S40)((Da[u3_2+2] == 0) ? (Dd.H) : (-(Dd.H))))`
	`Dn.L = SAT16 ((S40)((Da[u3_2+3] == 0) ? (Dd.L) : (-(Dd.L))))`
	`Dm.E = 0`
	`Dn.E = 0`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
u3_2	$0 \le$ u3_2 $< 2^5$;	u3_2 & 0x3 == 0

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

CNEGADD.nX Conditional Negate and Add (DALU)

General Description

For each of the registers in the source register pair (SIMD2), selectively copies or negates two 16-bit words, and adds them to the destination register.

Selection is done according to a nibble (4 bits) that is extracted from the first source operand using an immediate offset (with a value one of of 0, 4, 8, 12, 16, 20, 24, 28). Each bit in the nibble controls the selection of the corresponding 16-bit word in the source register pair: If it is set the word is negated, and if cleared the word is copied. If the value to negate is 0x8000 it is saturated to 0x7FFF. The order of the controlling bits is reversed for the REV variant.

Flag Options

Flag	Position	Description
N	flg1	Negative polarity
REV	flg2	Reverse bit order

Instruction Variants

#	Syntax	Description
1	**CNEGADD.N.2X #u3_2,Da,Dc:Dd,Dm:Dn**	**Conditional Negate and ADD (negative polarity)**

	`Dm = (Dm + SAT16 ((S40)((Da[u3_2+3] == 0) ? (Dc.H) : (-(Dc.H)))) + SAT16 ((S40)((Da[u3_2+2] == 0) ? (Dc.L) : (-(Dc.L)))))[39:0]`
	`Dn = (Dn + SAT16 ((S40)((Da[u3_2+1] == 0) ? (Dd.H) : (-(Dd.H)))) + SAT16 ((S40)((Da[u3_2+0] == 0) ? (Dd.L) : (-(Dd.L)))))[39:0]`	

2	**CNEGADD.N.REV.2X #u3_2,Da,Dc:Dd,Dm:Dn**	**Conditional Negate and ADD (negative polarity) - reversed order**

	`Dm = (Dm + SAT16 ((S40)((Da[u3_2+0] == 0) ? (Dc.H) : (-(Dc.H)))) + SAT16 ((S40)((Da[u3_2+1] == 0) ? (Dc.L) : (-(Dc.L)))))[39:0]`
	`Dn = (Dn + SAT16 ((S40)((Da[u3_2+2] == 0) ? (Dd.H) : (-(Dd.H)))) + SAT16 ((S40)((Da[u3_2+3] == 0) ? (Dd.L) : (-(Dd.L)))))[39:0]`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
u3_2	$0 \le u3\_2 < 2^5$;	u3_2 & 0x3 == 0

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU_CNEGADDMD	All

COB.nB Count One-Value Bits in Four Bytes (DALU)

General Description

Counts the number of bits whoÂs value is equal to 1 in each byte of the source register.

The bits in the extension are ignored. The count is right aligned and zero extended into the destination (unsigned integer format) at the corresponding byte position. The extension is cleared.

Instruction Variants

#	Syntax	Description
1	**COB.4B Da,Dn**	**Count One-Value Bits in 4 Bytes**
	`Dn = {0x00, NumberOfOnes (Da.HH), NumberOfOnes (Da.HL), NumberOfOnes (Da.LH), NumberOfOnes (Da.LL)}`	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Dn	D0-D63

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

COB.nL Count One-Value Bits (DALU)

General Description

Counts the number of bits whoÂs value is equal to 1 in bits 0 to 31 of the source register.

The bits in the extension are ignored. The count is right aligned and zero extended into the destination (unsigned integer format).

Instruction Variants

#	Syntax	Description
1	**COB.L Da,Dn**	**Count One-Value Bits**
	`Dn = (U40)NumberOfOnes (Da.M)`	

Explicit Operands

Operand	Permitted Values
Da	`D0-D63`
Dn	`D0-D63`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

CONT Continue to the Next Loop Iteration (PCU_LSU)

General Description

The instruction is used inside a hardware loop to skip the remainder of the current iteration and jump to the start of the loop to continue the next iteration. The instruction checks the unsigned loop counter (LC) of the specified loop index n2. If it is greater than 1, LC is decremented and a jump is performed to the start address of the loop, which should be pre-stored in Ra. If LC is equal to 1 or 0, it is cleared, and a branch is performed to the specified address label, which should point outside the loop. The assembler calculates and encodes the 19-bit signed PC-relative offset of that address. It is the responsibility of the user to load the start address of the loop into the R register used as the operand.

Flag Options

Flag	Position	Description
n2	flg1	Loop index (0, 1, 2, 3)

Instruction Variants

#	Syntax	Description
1	**CONT.n2 LABEL,Ra**	**Test LC of index n2. If greater than 1, jump to R1, and decrement LC. Else, branch to the PC-relative offset, and clear LC.**

```
if ( (u32) LCn>1 )
    then (PC = Ra); LCn= LCn - 1
else
    PC = PC + {RelAdd19,0} ; LCn=0
```

Explicit Operands

Operand	Permitted Values
Ra	R0-R31
RelAdd19	$-2^{18} \le \text{RelAdd19} < 2^{18}$
n2	$0 \le n2 \le 3$

Affect Instructions

Field	Relevant Variants	Comments
LC	All	
PC	All	

Affected By Instructions

Field	Relevant Variants
LC	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	48-bits	All
Pipeline behavior	pcu_COF_LP	All

DBAR Data Memory Load Barrier (LSU)

General Description

This instruction is a constituent of the DBARM meta-instruction, used for inserting a memory barrier before performing a memory load. The DBAR instruction should not be used in this form but only through DBARM; using the instruction on its own will give indeterminate results. For more information on the DBARM instruction, see its description page and also the Address Generation chapter.

Flag Options

Flag	Position	Description
EIEIO	flg1	Enforce In-order Execution of I/O
HWSYNC	flg1	Heavy-Weight Synchronization
IBLL	flg1	Barrier between two loads
IBSL	flg1	Barrier between a store and a load
L1SYNC	flg1	Barrier until reaching the L1 cache
SCFG	flg1	Sub-system Configuration

Instruction Variants

#	Syntax	Operation	Description
1	DBAR.EIEIO		A barrier following a non-cacheable write to SoC memory, ensuring it was written before subsequent accesses
2	DBAR.HWSYNC		Heavy-weight data memory synchronization
3	DBAR.IBLL		Load-to-load barrier
4	DBAR.IBSL		Store-to-load barrier
5	DBAR.L1SYNC		L1 data cache synchronization - assert hold until data cache is idle
6	DBAR.SCFG		Store configuration barrier - ensures that previous writes to the memory-mapped cluster registers were written

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Helper instruction		All
No predication		All
Pipeline behavior	agu_MOVE_LD_cmd	All

DBARM Data Memory Barrier (META INST)

General Description

This meta-instruction inserts a data memory barrier that is needed before some memory load instructions. A barrier should be placed between the two accesses that should be separated, and not in parallel to one of them. Such barriers are needed to ensure correct semantics that relate to access order. Variants relate to different hazard cases, and will insert a different number of holds depending on the state of the memory system. For minimal delays, the user should select the right barrier. The meta-instruction is encoded as an BDAR instruction of the right variant together with a SYNC.B instruction. Several programming rules apply, see rule A.9. For more explanations on the barriers, see the Address Generation chapter

Flag Options

Flag	Position	Description
EIEIO	flg1	Enforce In-order Execution of I/O
HWSYNC	flg1	Heavy-Weight Synchronization
IBLL	flg1	Barrier between two loads
IBSL	flg1	Barrier between a store and a load
L1SYNC	flg1	Barrier until reaching the L1 cache
SCFG	flg1	Sub-system Configuration

Instruction Variants

#	Syntax	Description
1	**DBARM.EIEIO**	**Enforce In-order Execution of I/O**
	`Encoded as: DBAR.EIEIO SYNC.B`	
2	**DBARM.HWSYNC**	**Heavy-Weight Synchronization**
	`Encoded as: DBAR.HWSYNC SYNC.B`	
3	**DBARM.IBLL**	**Load-to-load barrier**
	`Encoded as: DBAR.IBLL SYNC.B`	
4	**DBARM.IBSL**	**Store-to-load barrier**
	`Encoded as: DBAR.IBSL SYNC.B`	
5	**DBARM.L1SYNC**	**L1 data cache synchronization - assert hold until data cache is idle**
	`Encoded as: DBAR.L1SYNC SYNC.B`	
6	**DBARM.SCFG**	**Store configuration barrier - ensures that previous writes to the memory mapped cluster registers were written**
	`Encoded as: DBAR.SCFG SYNC.B`	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	64-bits	All
Execution unit	PCU+LSU	All
No predication		All
Pipeline behavior	per constructing instructions	All

DBARS Data Memory Store Barrier (LSU)

General Description

This instruction inserts a data memory barrier that is needed between two memory store instructions. A barrier should be placed between the two stores that should be separated, and not in parallel to one of them. Such barriers are needed to ensure correct semantics that relate to access order. For minimal delays, the user should select the right barrier. For more explanations on the barriers, see the Address Generation chapter.

Flag Options

Flag	Position	Description
IBSS	flg1	Barrier between two stores
L1	flg2	Direted to L1 cache
L12	flg2	Directed to L1 and L2 cache

Instruction Variants

#	Syntax	Operation	Description
1	DBARS.IBSS.L1		Store-to-Store barrier, ensuring that the accesses are not merged in the SGB
2	DBARS.IBSS.L12		Store-to-Store barrier, ensures that the first store is updated in the coherency domain before the second store is written to the L2 cache

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
No predication		All
Pipeline behavior	agu_MOVE_ST_cmd	All

DCM Granular Data Cache Management Operations (LSU)

General Description

This group of instructions perform various coherency management operations on the L1 data cache directly, without the involvement of the CME (Cache Management Engine). They operate on a single cache granule, which is an aligned, contiguous range of 64-bytes. Variants allow to synchronize, flush or invalidate a granule, either only from the L1 cache or from both the L1 and L2 caches. The address of the granule is specified in an Rn register.

Flag Options

Flag	Position	Description
FLUSH	flg1	Flush
INVAL	flg1	Invalidate
SYNC	flg1	Synchronize
L1	flg2	Direted to L1 cache
L12	flg2	Directed to L1 and L2 cache

Instruction Variants

#	Syntax	Operation	Description
1	DCM.FLUSH.L1 (Rn)		Flush the data cache granule from the L1 data cache
2	DCM.INVAL.L1 (Rn)		Invalidate the data cache granule in the L1 data cache
3	DCM.FLUSH.L12 (Rn)		Flush the data cache granule from both the L1 data cache and the L2 cache
4	DCM.INVAL.L12 (Rn)		Invalidate the data cache granule from both the L1 data cache and the L2 cache
5	DCM.SYNC.L12 (Rn)		Synchronize the data cache granule in the L1 data cache with the value in the L2 cache, and the L2 cache with main memory.

Explicit Operands

Operand	Permitted Values
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
No predication		All
Pipeline behavior	agu_MOVE_ST_nodat	All

DCMB Block Data Cache Management Operations (LSU)

General Description

This instruction is a constituent of the DCMM meta-instruction, used in configuring the CME (Cache Management Engine) to perform a cache management operation on a data block in the L2 and L1 caches. The DCMB instruction should not be used in this form but only through DCMM; using the instruction on its own will give indeterminate results. For more information on the DCMM instruction, see its description page and also the Address Generation chapter.

Flag Options

Flag	Position	Description
FLUSH	flg1	Flush
INVAL	flg1	Invalidate
SYNC	flg1	Synchronize
L1	flg2	Direted to L1 cache
L12	flg2	Directed to L1 and L2 cache
L2	flg2	Directed to L2 cache

Instruction Variants

#	Syntax	Description
1	**DCMB.INVAL.L1 Ra,Rn**	**Participate in configuring a CME task to invalidate a memory block from the L1 data cache.**
	`Rn = success status of the CME block command request`	
2	**DCMB.SYNC.L1 Ra,Rn**	**Participate in configuring a CME task to synchronize a memory block in L1 data with the L2 cache.**
	`Rn = success status of the CME block command request`	
3	**DCMB.FLUSH.L12 Ra,Rn**	**Participate in configuring a CME task to flush a memory block from both the L1 data cache and the L2 cache.**
	`Rn = success status of the CME block command request`	
4	**DCMB.INVAL.L12 Ra,Rn**	**Participate in configuring a CME task to invalidate a memory block from both the L1 data cache and the L2 cache.**
	`Rn = success status of the CME block command request`	
5	**DCMB.SYNC.L2 Ra,Rn**	**Participate in configuring a CME task to synchronize a memory block in the L2 cache with main memory.**
	`Rn = success status of the CME block command request`	

Explicit Operands

Operand	Permitted Values
Ra	R0-R31
Rn	R0-R31

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Helper instruction		All
No predication		All
Pipeline behavior	agu_MOVE_LD_cmd	All

DCMM Block Data Cache Coherency Operation (META INST)

General Description

This meta-instruction configures the CME (Cache Management Engine) to perform a cache coherency management operation on a block of data memory. The meta-instruction is encoded in 3 instructions - a DCMB instruction, a BCCAS instruction, and a SYNC.B instruction that must be grouped together for proper operation. The configuration data sent to the CME is taken from Ra and Rb. The CME returns a success or failure status of the configuration stage that is stored in the destination Rn. The exact format of the data in the input and output registers is described in the Address Generation chapter. Variants allow to synchronize, flush or invalidate a data memory range, either only from the L1 cache or from both the L1 and L2 caches. Some programming rules apply to these instructions, see rule A.9.

Flag Options

Flag	Position	Description
FLUSH	flg1	Flush
INVAL	flg1	Invalidate
SYNC	flg1	Synchronize
L1	flg2	Direted to L1 cache
L12	flg2	Directed to L1 and L2 cache
L2	flg2	Directed to L2 cache

Instruction Variants

#	Syntax	Description
1	**DCMM.INVAL.L1 Ra,Rb,Rn**	**Configure a CME task to invalidate a memory block from the L1 data cache**
	`Encoded as: DCMB.INVAL.L1 Rb,Rn  BCCAS Ra  SYNC.B`	
2	**DCMM.SYNC.L1 Ra,Rb,Rn**	**Configure a CME task to synchronize a memory block in L1 data with the L2 cache**
	`Encoded as: DCMB.SYNC.L1 Rb,Rn  BCCAS Ra  SYNC.B`	
3	**DCMM.FLUSH.L12 Ra,Rb,Rn**	**Configure a CME task to flush a memory block from both the L1 data cache and the L2 cache**
	`Encoded as: DCMB.FLUSH.L12 Rb,Rn  BCCAS Ra  SYNC.B`	
4	**DCMM.INVAL.L12 Ra,Rb,Rn**	**Configure a CME task to invalidate a memory block from both the L1 data cache and the L2 cache**
	`Encoded as: DCMB.INVAL.L12 Rb,Rn  BCCAS Ra  SYNC.B`	
5	**DCMM.SYNC.L2 Ra,Rb,Rn**	**Configure a CME task to synchronize a memory block in the L2 cache with the main memory**
	`Encoded as: DCMB.SYNC.L2 Rb,Rn  BCCAS Ra  SYNC.B`	

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`
Rb	`R0-R31`
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	64-bits	All
Execution unit	2xLSU+PCU	All
No predication		All
Pipeline behavior	per constructing instructions	All

DEBUGE — Generate a Compiled-in Debug Exception Request — (PCU)

General Description

If the instruction is enabled, executing it will trigger a debug exception. If not enabled, it is executed as a NOP. DEBUGE is typically used for user-inserted exception points, which may remain in the code and disabled externally, hence the term . Processing the exception includes the following actions:
1. Saving SR, SR2, EIDR and the PC of the VLES in LR2
2. Changing SR2 according to the settings of a debug exception
3. Setting EIDR with the value for DEBUGE, along with the immediate value of u10.
4. Jumping to the start of the debug exception routine

Debug exceptions are enabled only if the monitor partition of the DTU is enabled. In addition, the specific DEBUGE.n2 variant (n2=0..3) should be enabled in the appropriate DTU control register. The address of the exception routine should be configured in the DTU as well. The immediate operand is a 10-bit value that is sampled into EIDR during exception processing, and could be read by the exception software. It can be used to transfer information from the calling code to the ISR, for example specifying the exact type of debug service that is required - based on a pre-defined agreement.
The DEBUGE instruction is serviced as break-before-make, meaning that the return PC is that of the VLES that includes the DEBUGE instruction. Hence, in the normal case, the ISR should increment the return PC by 4 before returning (DEBUGE cannot be grouped with other instructions, so the size is of the VLES to skip is 4 bytes). For more information on exception processing, see the program control chapter. For more information on debug exceptions, see the debug chapters in both the core and subsystem RM.

Flag Options

Flag	Position	Description
n2	flg1	Variant index (0, 1, 2, 3)

Instruction Variants

#	Syntax	Description
1	**DEBUGE.n2 #u10**	**If enabled, enter Debug Exception, and sample u10 to EIDR**

	`PC = DESAR; LR2 = {return PC, SR, SR2, EIDR}`
	`EIDR = {16'b0, 4'b1000,n2,u10}`	

Explicit Operands

Operand	Permitted Values
	$0 \le n2 \le 3$
u10	$0 \le u10 < 2^{10}$

Affect Instructions

Field	Relevant Variants	Comments
EIDR	All	
SR	All	
SR2	All	

Affected By Instructions

Field	Relevant Variants
EIDR	All
LR	All
SR.C	All
SR.S	All
SR.[P0,P1,P2,P3,P4,P5]	All
SR2.IPM	All
SR2.SPSEL	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Must be alone in VLES		All
Pipeline behavior	pcu_control	All

DEBUGEV Trigger a Debug Event (IPU)

General Description

This instruction triggers a debug event to the DTU (Debug and Trace Unit). The outcome of the event is configured and enabled in the DTU registers, and may be different for each variant (0 to 7). Example events could be to enable trace, generate an event to count, stop profiling, etc. The instruction has no effect on the programming model of the core, and could be considered as a NOP. For more information on the events that could be triggered with this instruction, refer to the Debug chapter in the Subsystem RM.

Flag Options

Flag	Position	Description
n3	flg1	Variant index (0 to 7)

Instruction Variants

#	Syntax	Operation	Description
1	DEBUGEV.n3		Generate a debug event

Explicit Operands

Operand	Permitted Values
n3	`0, 1, 2, 3, 4, 5, 6, 7`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Pipeline behavior	pcu_control	All

DEBUGM Generate a Compiled-in Debug Mode Request (PCU)

General Description

If the instruction is enabled, executing it will cause the core to enter debug mode. If not enabled, it is executed as a NOP. Debug mode is enabled only if the host partition of the DTU is enabled. In addition, the specific DEBUGM.n2 variant (n2=0..3) should be enabled in the appropriate DTU control register (once the DTU is enabled, the instructions are enabled by default). DEBUGM is typically used for user-inserted halts, which may remain in the code and disabled externally, hence the term . The DEBUGM instruction is serviced as break-before-make, meaning that unless modified, the next PC to execute when exiting debug mode will be that of the VLES with the DEBUGM instruction. Hence, in the normal case, the host debugger should increment PC_NEXT by 4 before resuming (DEBUGM cannot be grouped with other instructions, so the size is of the VLES to skip is 4 bytes). Note that the debugger cannot single-step over this VLES. For more information on debug mode, see the debug chapters in both the core and subsystem RM.

Flag Options

Flag	Position	Description
n2	flg1	Variant index (0, 1, 2, 3)

Instruction Variants

#	Syntax	Operation	Description
1	DEBUGM.n2		If enabled, enter debug mode.

Explicit Operands

Operand	Permitted Values
	0 ≤ n2 ≤ 3

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Must be alone in VLES		All
Pipeline behavior	pcu_control	All

DEC.X Decrement and Set Pn conditionally (DALU)

General Description

Decrements a DALU register and sets the Pn bit or the Pn:Pm pair if the new value meets the condition.

Flag Options

Flag	Position	Description
GE	flg1	Greater or Equal comparison
NE	flg1	Not Equal comparison

Instruction Variants

#	Syntax	Operation	Description

1	DEC.GE.X Dn,Pm:Pn	`Dn--`
		`Pm = (Dn ≥ 0) ? 1 : 0`
			Decrement Dn and Set Pm If Greater Than or Equal to Zero or clear it otherwise, Pn written with complementary value of Pm

2	DEC.GE.X Dn,Pn	`Dn--`
		`Pn = (Dn ≥ 0) ? 1 : 0`
			Decrement Dn and Set Pn If Greater Than or Equal to Zero or clear it otherwise

3	DEC.NE.X Dn,Pm:Pn	`Dn--`
		`Pm = (Dn != 0) ? 1 : 0`
			Decrement Dn and Set Pm If Not Equal to Zero or clear it otherwise, Pn written with complementary value of Pm

4	DEC.NE.X Dn,Pn	`Dn--`
		`Pn = (Dn != 0) ? 1 : 0`
			Decrement Dn and Set Pn If Not Equal to Zero or clear it otherwise

Explicit Operands

Operand	Permitted Values	
Dn	D0-D63	
Pm:Pn	P_n:P_{n+1}	$0 \le n \le 4$
Pn	P0-P5	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU_Pbit	All

DECA Decrement and Set Pn conditionally (LSU,IPU)

General Description

Decrements an address register and set the Pn bit or the Pn:Pm pair if the new value meets the condition.

Flag Options

Flag	Position	Description
GE	flg1	Greater or Equal comparison
NE	flg1	Not Equal comparison

Instruction Variants

#	Syntax	Operation	Description

1	DECA.GE Rn,Pm:Pn	`Rn = Rn - 1`
		`Pm = (Rn ≥ 0) ? 1 : 0`
		`Pn = ~(new Pm)`
			Decrement Rn and Set Pm If Greater Than or Equal to Zero or clear it otherwise, Pn written with complementary value of Pm

2	DECA.GE Rn,Pn	`Rn = Rn - 1`
		`Pn = (Rn ≥ 0) ? 1 : 0`
			Decrement Rn and Set Pn If Greater Than or Equal to Zero or clear it otherwise

3	DECA.NE Rn,Pm:Pn	`Rn = Rn - 1`
		`Pm = (Rn != 0) ? 1 : 0`
		`Pn = ~(new Pm)`
			Decrement Rn and Set Pm If Not Equal to Zero or clear it otherwise, Pn written with complementary value of Pm

4	DECA.NE Rn,Pn	`Rn = Rn - 1`
		`Pn = (Rn != 0) ? 1 : 0`
			Decrement Rn and Set Pn If Not Equal to Zero or clear it otherwise

Explicit Operands

Operand	Permitted Values	
Pm:Pn	$P_n:P_{n+1}$	$0 \le n \le 4$
Pn	P0-P5	
Rn	R0-R31	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Execution unit	IPU	All
	LSU	All
Pipeline behavior	agu_AAU_Pbit	All

DECOR Decorate a Memory Access (IPU)

General Description

This is a helper instruction that is grouped with a load, store or NOTIFY instruction, and adds decoration attributes to it. Decoration information is signaled with 4 bits that accompany the access throughout the SoC CoreNet fabric, and dedicated peripheral units may respond to this signaling in pre-defined ways. For example, a decorated access may increment a counter at the destination. Decorated accesses are part of the CoreNet definition. Some programming rules apply to decorated accesses, see rule A.9. For more information on decorated accesses, see the Address Generation chapter.

Instruction Variants

#	Syntax	Operation	Description
1	DECOR #u4		Decorate the parallel memory access with decoration field u4

Explicit Operands

Operand	Permitted Values
u4	$0 \leq u4 < 2^4$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Helper instruction		All
Pipeline behavior	pcu_control	All

DFETCH Granular Pre-Fetch to the Data Cache (LSU)

General Description

This instructions pre-fetches a cache granule to either the L2 cache or to both the L1 and L2 cache directly, without the involvement of the CME (Cache Management Engine). A cache granule is an aligned, contiguous range of 64-bytes. The address of the granule is specified in an Rn register. For more information on pre fetching, refer to the Address Generation chapter.

Flag Options

Flag	Position	Description
L12	flg2	Directed to L1 and L2 cache
L2	flg2	Directed to L2 cache

Instruction Variants

#	Syntax	Operation	Description
1	DFETCH.L12 (Rn)		Pre-fetch a data granule to the L1 and L2 caches
2	DFETCH.L2 (Rn)		Pre-fetch a data granule to the L2 cache

Explicit Operands

Operand	Permitted Values
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
No predication		All
Pipeline behavior	agu_MOVE_LD_nodat	All

DFETCHB Block Pre-Fetch to the Data Cache (LSU)

General Description

This instruction is a constituent of the DFETCHM meta-instruction, used in configuring the CME (Cache Management Engine) to pre-fetch data to the L2 and L1 caches. The DFETCHB instruction should not be used in this form but only through DFETCHM; using the instruction on its own will give indeterminate results. For more information on the DFETCHM instruction, see its description page and also the Address Generation chapter.

Flag Options

Flag	Position	Description
LCK	flg1	Lock lines in L2
L12	flg2	Directed to L1 and L2 cache
L2	flg2	Directed to L2 cache

Instruction Variants

#	Syntax	Description
1	**DFETCHB.L12 Ra,Rn**	**Participate in configuring a CME task to pre-fetch a memory block to both the L2 cache and the L1 data cache.**
	`Rn = success status of the CME block command request`	
2	**DFETCHB.L2 Ra,Rn**	**Participate in configuring a CME task to pre-fetch a memory block to the L2 cache.**
	`Rn = success status of the CME block command request`	
3	**DFETCHB.LCK.L2 Ra,Rn**	**Participate in configuring a CME task to pre-fetch a memory block to the L2 cache, and lock the updated lines.**
	`Rn = success status of the CME block command request`	

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Helper instruction		All

Table continues on the next page...

Attribute	Value	Relevant Variant #
No predication		All
Pipeline behavior	agu_MOVE_LD_cmd	All

DFETCHM Block Prefetch to the Data Cache (META INST)

General Description

This meta-instruction configures the CME (Cache Management Engine) to pre-fetch a block of data to the L2 cache or also to the L1 data cache. The meta-instruction is encoded in 3 instructions - a DFETCHB instruction, a BCCAS instruction, and a SYNC.B instruction that must be grouped together for proper operation. The configuration data sent to the CME is taken from Ra and Rb. The CME returns a success or failure status of the configuration stage that is stored in the destination Rn. The exact format of the data in the input and output registers is described in the Address Generation chapter. Variants allow to fetch the data only to the L2, or also to L1; It is also possible to lock the fetched data in L2. Some programming rules apply to these instructions, see rule A.9. For more information on block cache instructions, refer to the Address Generation chapter.

Flag Options

Flag	Position	Description
LCK	flg1	Lock lines in L2
L12	flg2	Directed to L1 and L2 cache
L2	flg2	Directed to L2 cache

Instruction Variants

#	Syntax	Description
1	**DFETCHM.L12 Ra,Rb,Rn**	**Configure a CME task to pre-fetch a memory block to both the L2 cache and the L1 data cache.**
	`Encoded as: DFETCHB.L12 Rb,Rn  BCCAS Ra  SYNC.B`	
2	**DFETCHM.L2 Ra,Rb,Rn**	**Configure a CME task to pre-fetch a memory block to the L2 cache**
	`Encoded as: DFETCHB.L2 Rb,Rn  BCCAS Ra  SYNC.B`	
3	**DFETCHM.LCK.L2 Ra,Rb,Rn**	**Configure a CME task to pre-fetch a memory block to the L2 cache, and lock the updated lines**
	`Encoded as: DFETCHB.LCK.L2 Rb,Rn  BCCAS Ra  SYNC.B`	

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`
Rb	`R0-R31`
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	96-bits	All
Execution unit	2xLSU+PCU	All
No predication		All
Pipeline behavior	per constructing instructions	All

DI Disable Maskable interrupts (IPU)

General Description

The instruction sets the SR2.DI bit, disabling the maskable interrupts without affecting the Interrupt Priority Mask that is configured in SR2.IPM. For more information on maskable interrupts refer to the Program Control chapter. The instruction is an alias of the BMCLRA instruction.

Instruction Variants

#	Syntax	Description
1	**DI**	**Disable maskable interrupts**

	`SR2.DI = 1`
	`Alias, encoded as: BMSETA #$40,SR2.H`	

Affected By Instructions

Field	Relevant Variants
SR2.DI	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Alias		All
Encoding length	48-bits	All
Pipeline behavior	agu_AAU_CTRL_REG	All

DIVP Parallel Divide Iteration (DALU)

General Description

Calculates one quotient bit of the division of the value in the destination data register by the value in the source data register using a non-restoring fractional division algorithm.

Each of DIVP.0-DIVP.3 instructions uses a GCR bit (AS0-AS3, respectively) to store and use the carry from the previous iteration. This behavior enables to perform up to four division operations in parallel, on different operands. After the execution of the first DIVP.n instruction, the destination register holds both the partial remainder and the formed quotient. The partial remainder occupies the high portion of the register and is a signed fraction. The formed quotient occupies the low portion of the register and is a positive fraction. Each subsequent DIVP.n instruction shifts one bit of the formed quotient into bit 0 of the register at the start of the DIVP.n iteration. If the true quotient is positive, the formed quotient is the true quotient. If the true quotient is negative, the formed quotient must be negated. The instruction obtains

The result is valid only when the absolute value of the destination register is less than the absolute value of the source register and the operand values are interpreted as fractions. This condition ensures that the magnitude of the quotient is less than one (that is, is fractional) and precludes division by zero.

The DIVPn instruction calculates one quotient bit based on the divisor and the previous partial remainder. To produce an N-bit quotient, execute the DIVP.n instruction N times, where N is the number of bits of precision desired in the quotient ($1 \le N \le 16$). Thus, a full-precision (16-bit) quotient requires 16 DIVP.n iterations. In general, executing the DIVP.n instruction N times produces an N-bit quotient and a 32-bit remainder that has (32 - N) bits of precision and whose N most significant bits are zeros. The partial remainder is not a true remainder and must be corrected (due to the non-restoring nature of the division algorithm) before it can be used. Therefore, once the divide is complete, it is necessary to reverse the last DIVP.n operation and restore the remainder to obtain the true remainder.

The non-restoring fractional division algorithm used by DIVP.n consists of the following operations:
1. Compare the sign bits of both operands.
Performs an exclusive OR operation on the sign bits of the source and destination operands.
2. Shift the partial remainder and the quotient.
Shifts the bits in destination one position to the left and copies the Carry (ASn GCR) bit into the vacated bit 0. The Carry bit represents the quotient bit generated by the previous DIVP.n iteration.
3. Calculate the next quotient bit and the new partial remainder.

If the result of the exclusive OR operation in the first step was 1 (that is, the sign bits were different), adds the signed 16-bit divisor in Da.H to the contents of Dn.H; otherwise, subtracts Da.H from Dn.H. The sign extension of the signed 16-bit divisor causes the addition or subtraction operation to set the Carry (ASn GCR) bit correctly with the next quotient bit.

The DIVP.n instruction do not supports extended precision division (that is, for N-bit quotients where N > 16); therefore, you must provide a user-defined N-bit division routine.
For further information on division algorithms, see one of the following references or other references as required:
- Theory and Application of Digital Signal Processing by Rabiner and Gold (Prentice-Hall, 1975), pages 524-530.
- Computer Architecture and Organization by John Hayes (McGraw-Hill, 1978), pages 190-199.
- Computer Arithmetic: Principles, Architecture, and Design by Kai Hwang (John Wiley and Sons, 1979), pages 213-223.

Flag Options

Flag	Position	Description
0	flg1	Variant index
1	flg1	Variant index
2	flg1	Variant index
3	flg1	Variant index

Instruction Variants

#	Syntax	Description
1	**DIVP.0 Da,Dn**	**Parallel Divide Iteration, with Bit GCR.AS0**

	`(Dn[39] ^ Da[39])? Dn = 2 * Dn + GCR.AS0 + (Da & 0xFF_FFFF_0000)`
	`: Dn = 2 * Dn + GCR.AS0 - (Da & 0xFF_FFF_0000)`	

2	**DIVP.1 Da,Dn**	**Parallel Divide Iteration, with Bit GCR.AS1**

	`(Dn[39] ^ Da[39])? Dn = 2 * Dn + GCR.AS1 + (Da & 0xFF_FFFF_0000)`
	`: Dn = 2 * Dn + GCR.AS1 - (Da & 0xFF_FFF_0000)`	

3	**DIVP.2 Da,Dn**	**Parallel Divide Iteration, with Bit GCR.AS2**

	`(Dn[39] ^ Da[39])? Dn = 2 * Dn + GCR.AS2 + (Da & 0xFF_FFFF_0000)`
	`: Dn = 2 * Dn + GCR.AS2 - (Da & 0xFF_FFF_0000)`	

4	**DIVP.3 Da,Dn**	**Parallel Divide Iteration, with Bit GCR.AS3**

	`(Dn[39] ^ Da[39])? Dn = 2 * Dn + GCR.AS3 + (Da & 0xFF_FFFF_0000)`
	`: Dn = 2 * Dn + GCR.AS3 - (Da & 0xFF_FFF_0000)`	

Explicit Operands

Operand	Permitted Values
Da	`D0-D63`
Dn	`D0-D63`

Affect Instructions

Field	Relevant Variants	Comments
GCR.AS0	1	
GCR.AS1	2	
GCR.AS2	3	
GCR.AS3	4	

Affected By Instructions

Field	Relevant Variants
GCR.AS0	1
GCR.AS1	2
GCR.AS2	3
GCR.AS3	4

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

DOALIGN.L Extract Unaligned four Bytes (DALU)

General Description

Copies 4 consecutive bytes out of 8 bytes stored in a pair of source data registers, starting from the first, second, third, or fourth byte, as specified in the Byte Alignment Mode field of the GCR (GCR.BAM), into the destination.

Instruction Variants

#	Syntax	Description
1	**DOALIGN.L Da,Db,Dn**	**Extract Unaligned four Bytes**

```
switch (GCR.BAM)
   case 0b101: Dn.M = {Da[23:0],Db[31:24]}
   case 0b110: Dn.M = {Da[15:0],Db[31:16]}
   case 0b111: Dn.M = {Da[7:0],Db[31:8]}
   case 0b001: Dn.M = {Db[7:0],Da[31:8]}
   case 0b010: Dn.M = {Db[15:0],Da[31:16]}
   case 0b011: Dn.M = {Db[23:0],Da[31:24]}
```

Explicit Operands

Operand	Permitted Values
Da	`D0-D63`
Db	`D0-D63`
Dn	`D0-D63`

Affect Instructions

Field	Relevant Variants	Comments
GCR.BAM	All	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

DOEN Enable a Hardware Loop (IPU)

General Description

Enable hardware loop n by transferring the operand to the respective LC register (LC0, LC1 etc.). The loop count is either specified in an immediate operand or in an R register. The instruction should be placed outside the loop body and executed before it for the loop to iterate properly, some other programming rules apply. The hardware loop logic is generally interpreting the loop count as unsigned. In case the loop count value in R cannot be ensured to be unsigned, it is possible to test it with a signed variant of the SKIP instruction. For more information on hardware loops, see the Program Control chapter and the relevant programming rules in the Programming Rules chapter.

Flag Options

Flag	Position	Description
n2	flg1	Loop index (0, 1, 2, 3)

Instruction Variants

#	Syntax	Operation	Description
1	DOEN.n2 Ra	`LC (n2) = Ra`	Enable hardware loop n, iteration count specified in an R register
2	DOEN.n2 #u16	`LC (n2) = (U32)u16`	Enable hardware loop n, iteration count specified in an immediate operand

Explicit Operands

Operand	Permitted Values
Ra	R0-R31
n2	$0 \le n2 \le 3$
u16	$0 \le u16 < 2^{16}$

Affected By Instructions

Field	Relevant Variants
LC	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Pipeline behavior	agu_LPSETUP	All

DQSYNC Global Data Cache Synchronization (LSU)

General Description

This instruction is a constituent of the DQSYNCM meta-instruction, used to globally invalidate the L1 data cache. The DQSYNC instruction should not be used in this form but only through DQSYNCM; using the instruction on its own will give indeterminate results. For more information on the DQSYNCM instruction, see its description page and also the Address Generation chapter.

Instruction Variants

#	Syntax	Operation	Description
1	DQSYNC		Global data cache invalidation.

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Helper instruction		All
No predication		All
Pipeline behavior	agu_MOVE_LD_cmd	All

DQSYNCM Data Memory Global Barrier (META INST)

General Description

This meta-instruction globally invalidates the L1 data cache. It is encoded by the DQSYNC instruction, grouped with the SYNC.B instruction. For more information on cache invalidation, see the Address Generation chapter.

Instruction Variants

#	Syntax	Operation	Description
1	DQSYNCM	`Encoded as: DQSYNC SYNC.B`	Global data cache invalidation

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	64-bits	All
Execution unit	PCU+LSU	All
No predication		All
Pipeline behavior	per constructing instructions	All

DQUERY Query the MMU and Data Cache (LSU)

General Description

This instruction is used to query that status of an address in the data MMU and the data caches. The address to query is stored in Rn, and the query result is returned in Ra:Rb. A description of the information format returned by this instruction is described in the Address Generation chapter. This instruction could be used for example to make sure that a certain address is allowed for access in the MMU, without the risk of getting an MMU exception in case it is not. Variants return, in addition to the status in the MMU, the status of the address in the L1 cache, or in both the L1 and L2 caches. The query of the L2 cache takes a longer number of cycles. Some programming rules apply to DQUERY instructions, see rule A.9.

Flag Options

Flag	Position	Description
L1	flg2	Direted to L1 cache
L12	flg2	Directed to L1 and L2 cache

Instruction Variants

#	Syntax	Description
1	**DQUERY.L1 (Rn),Ra:Rb**	**Query the L1 data cache and MMU status of the address specified in Rn.**
	`Ra:Rb = cache and MMU status of (Rn)`	
2	**DQUERY.L12 (Rn),Ra:Rb**	**Query the L1 and L2 data cache and MMU status of the address specified in Rn.**
	`Ra:Rb = cache and MMU status of (Rn)`	

Explicit Operands

Operand	Permitted Values	
Ra:Rb	$R_n:R_{n+1}$	$0 \leq n \leq 30$
Rn	R0-R31	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
No predication		All
Pipeline behavior	agu_MOVE_LD	All

DUNLOCKB Block Unlock Lines in the Data Cache (LSU)

General Description

This instruction is a constituent of the DUNLOCKM meta-instruction, used in configuring the CME (Cache Management Engine) to unlock a block of data in the L2 cache. The DUNLOCKB instruction should not be used in this form but only through DUNLOCKM; using the instruction on its own will give indeterminate results. For more information on the DUNLOCKM instruction, see its description page and also the Address Generation chapter.

Flag Options

Flag	Position	Description
L2	flg2	Directed to L2 cache

Instruction Variants

#	Syntax	Description
1	**DUNLOCKB.L2 Ra,Rn**	**Participate in configuring a CME task to unlock lines in the L2 cache that belong to a memory block.**
	`Rn = success status of the CME block command request`	

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Helper instruction		All
Pipeline behavior	agu_MOVE_LD_cmd	All

DUNLOCKM Block Data Cache Unlock (META INST)

General Description

This meta-instruction configures the CME (Cache Management Engine) to unlock lines in the L2 cache that belong to a data memory block. The meta-instruction is encoded in 3 instructions - a DUNLOCKB instruction, a BCCAS instruction, and a SYNC.B instruction. The configuration data sent to the CME is taken from Ra and Rb. The CME returns a success or failure status of the configuration stage that is stored in the destination Rn. The exact format of the data in the input and output registers is described in the Address Generation chapter. Some programming rules apply to the this instruction, see rule A.9. For more information on block cache instructions, refer to the Address Generation chapter.

Flag Options

Flag	Position	Description
L2	flg2	Directed to L2 cache

Instruction Variants

#	Syntax	Description
1	**DUNLOCKM.L2 Ra,Rb,Rn**	**Configure a CME task to unlock lines in the L2 cache that belong to a memory block**
	`Encoded as: DUNLOCKB.L2 Rb,Rn  BCCAS Ra  SYNC.B`	

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`
Rb	`R0-R31`
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	96-bits	All
Execution unit	2xLSU+PCU	All
No predication		All
Pipeline behavior	per constructing instructions	All

EI Enable Maskable interrupts (IPU)

General Description

The instruction clears the SR2.DI bit, enabling the maskable interrupts according to the Interrupt Priority Mask that is configured in SR2.IPM. For more information on maskable interrupts refer to the Program Control chapter. The instruction is an alias of the BMSETA instruction.

Instruction Variants

#	Syntax	Description
1	**EI**	**Enable maskable interrupts**

	`SR2.DI = 0`
	`Alias, encoded as: BMCLRA #$40,SR2.H`	

Affected By Instructions

Field	Relevant Variants
SR2.DI	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Alias		All
Encoding length	48-bits	All
Pipeline behavior	agu_AAU_CTRL_REG	All

EOR.nL Bitwise Exclusive OR (32-bit) (DALU)

General Description

Perform a bitwise Exclusive OR (XOR) operation on the lower 32 bits of a D register

Instruction Variants

#	Syntax	Operation	Description
1	EOR.L #u32,Da,Dn	`Dn = {Da.E, ((Da.M) ^ u32)}`	32-bit Exclusive OR with an immediate value. The extension of the destination is not changed

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Dn	D0-D63
u32	$0 \le u32 < 2^{32}$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	64-bits	All
Pipeline behavior	dalu_LBM	All

EOR.nX Bitwise Exclusive OR (40-bit) (DALU)

General Description

Perform a bitwise Exclusive OR (XOR) operation on all the bits of a D register

Instruction Variants

#	Syntax	Operation	Description

1	EOR.2X Da:Db,Dc:Dd,Dm:Dn	`Dm = (Da ^ Dc)`
		`Dn = (Db ^ Dd)`
			SIMD2 exclusive OR operation of two pairs of D registers into two destination registers, respectively

2	EOR.X Da,Db,Dn	`Dn = Da ^ Db`	Exclusive OR of the bits of two D registers

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

EORA Bitwise Exclusive OR (32 bit) (LSU,IPU)

General Description

Performs a bitwise Exclusive Or (XOR) operation on R and control registers. Variants include 32-bit bitwise operation on values in two R registers, and 16-bit operation between an unsigend immediate value and a 16-bit register portion (wither high or low). The register is either an R regisetr or a control register. The other portion of the register is not changed.

Instruction Variants

#	Syntax	Description
1	**EORA Ra,Rb,Rn**	**Exclusive OR of the bits of two R registers**
	`Rn = Ra ^ Rb`	
2	**EORA #u16,C4.H**	**Bit-wise Exclusive Or of the higher 16-bit control register portion with an unsigned immediate.**

	`C4.H = C4.H ^ u16`
	`Alias, encoded as: BMCHGA #u16 C4.H`	

3	**EORA #u16,C4.L**	**Bit-wise Exclusive Or of the lower 16-bit control register portion with an unsigned immediate.**

	`C4.L = C4.L ^ u16`
	`Alias, encoded as: BMCHGA #u16 C4.L`	

4	**EORA #u16,Rn.H**	**Bit-wise Exclusive Or of the higher 16-bit R register portion with an unsigned immediate.**

	`Rn.H = Rn.H ^ u16`
	`Alias, encoded as: BMCHGA #u16 Rn.H`	

5	**EORA #u16,Rn.L**	**Bit-wise Exclusive Or of the lower 16-bit R register portion with an unsigned immediate.**

	`Rn.L = Rn.L ^ u16`
	`Alias, encoded as: BMCHGA #u16 Rn.L`	

Explicit Operands

Operand	Permitted Values

C4	`EIDR, GCR, MCTL, MOCR, SR,`
	`SR2, TMTAG`

Ra	`R0-R31`
Rb	`R0-R31`
Rn	`R0-R31`
u16	$0 \le u16 < 2^{16}$

Instruction Attributes

Attribute	Value	Relevant Variant #
Alias		2, 3, 4, 5
Encoding length	32-bits	1
	48-bits	2, 3, 4, 5
Execution unit	IPU	All
	LSU	1
Pipeline behavior	agu_AAU_CTRL_REG	2, 3
	agu_AAU_LOGIC	1, 4, 5

EXTC Extract the Last Shifted Bit Into the Carry (DALU)

General Description

These instructions perform an arithmetic or logical shift from D a register. The last bit that is shifted out is saved in the carry bit. No Data register is updated by this shift. In case the shifted value is needed, an additional shift instruction that updated a D register should be executed in parallel. The shift amount is specified either by an unsigned immediate or in a signed field inside an additional input. In such a case, a positive value in this field will shift the operand in the direction specified in the mnemonic, and a negative value will shift the operand by the same amount to the other direction. Arithmetic shifts to the right will insert sign extending bits from the left, while logical shifts will add zeros. This instruction can be used to emulate a legacy shift instruction which always updated the carry as a side effect, by supplementing the carry generation in parallel with another instruction that updated the destination D register. Variants include left or right shifts, logical or arithmetic shifts, 40-bit or 32-bit shifts and register or immediate based shift size specification. An absolute shift value of 40 or larger will behave like a 40-bit shift of the respective direction. A shift by 0 or -64 will clear the carry.

Flag Options

Flag	Position	Description
LFT	flg1	Left
RGT	flg1	Right

Instruction Variants

<table>
<tr><th>#</th><th>Syntax</th><th>Description</th></tr>
<tr><td>1</td><td>AEXTC.LFT Da,Db</td><td>Write the carry with the last shifted bit of an arithmetic 40-bit left shift of Db, based on the signed value in Da[6:0]. A positive value will shift to the left, a negative value will shift to the right.</td></tr>
<tr><td></td><td colspan="2">shift = max(39,|Da[6:0]|)
if (shift == 0 || Da[6:0]==-64) SR.C = 0
else SR.C = Da[6:0]>0 ? Db[40 - shift] : Db[shift - 1]</td></tr>
<tr><td>2</td><td>EXTC.LFT #u5,Da</td><td>Write the carry with the last shifted bit of an 40-bit left shift of Da, shifted by an unsigned immediate. The shift amount is limited to 32, so it is not marked as logical not arithmetic.</td></tr>
<tr><td></td><td colspan="2">if (u5 == 0) SR.C = 0
else SR.C = Db[40 - u5]</td></tr>
<tr><td>3</td><td>LEXTC.LFT.X Da,Db</td><td>Write the carry with the last shifted bit of a logical 40-bit left shift of Db, based on the signed value in Da[6:0]. A positive value will shift to the left, a negative value will shift to the right.</td></tr>
<tr><td></td><td colspan="2">shift = |Da[6:0]|
if (shift == 0 || Da[6:0]==-64 || shift > 39) SR.C = 0
else SR.C = Da[6:0]>0 ? Db[40 - shift] : Db[shift - 1]</td></tr>
</table>

Table continues on the next page...

#	Syntax	Description
4	**AEXTC.RGT Da,Db**	**Write the carry with the last shifted bit of an arithmetic 40-bit right shift of Db, based on the signed value in Da[6:0]. A positive value will shift to the right, a negative value will shift to the left.**

	shift = max(39,|Da[6:0]|)
	if (shift == 0 || Da[6:0]==-64) SR.C = 0
	else SR.C = Da[6:0]>0 ? Db[shift - 1] : Db[40 - shift]	

5	**EXTC.RGT #u5,Da**	**Write the carry with the last shifted bit of a 40-bit right shift of Da, shifted by an unsigned immediate. The shift amount is limited to 32, so it is not marked as logical not arithmetic.**

	if (u5 == 0) SR.C = 0
	else SR.C = Db[u5 - 1]	

6	**LEXTC.RGT.L Da,Db**	**Write the carry with the last shifted bit of an logical 32-bit right shift of Db[31:0], based on the signed value in Da[6:0]. A positive value will shift to the left, a negative value will shift to the right.**

	shift = |Da[6:0]|
	if (shift == 0 || Da[6:0]←39 || shift > 32) SR.C = 0
	else SR.C = Da[6:0]>0 ? Db[shift - 1] : Db[32 - shift]	

7	**LEXTC.RGT.X Da,Db**	**Write the carry with the last shifted bit of a logical 40-bit left shift of Db, based on the signed value in Da[6:0]. A positive value will shift to the right, a negative value will shift to the left.**

	shift = |Da[6:0]|
	if (shift == 0 || Da[6:0]==-64 || shift > 39) SR.C = 0
	else SR.C = Da[6:0]>0 ? Db[shift - 1] : Db[40 - shift]	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Db	D0-D63
u5	$0 \leq u5 < 2^5$

Affected By Instructions

Field	Relevant Variants
SR.C	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM_CARRY	All

EXTINS.LL Extract and Insert a Bit Field From 64-Bit Data (DALU)

General Description

The instruction allows to specify a bit field of arbitrary offset and width (up to 64 bits) in a 64-bit input, and inserts this field in an arbitrary offset in a 64-bit output. Both input and output data are specified as two pairs of concatenated registers (without the extension). The width and offset of the source field, and the offset of the destination field are specified in fixed fields in additional source registers. If the source field spills over the boundary of the 64-bit input, the destination registers are cleared. If the destination field spills over the boundary of the 64-bit output, the destination field is updated up to the 64-bit limit. The extensions in the destination registers are cleared.

Instruction Variants

#	Syntax	Description
1	**EXTINS.LL Da:Db,Dc:Dd,Dm:Dn**	**Extract and insert a bit field in 64-bit data. The offset and width of the extracted field are specified in Da. The offset of the inserted field in the destination is specified in Db. The 64-bit input is specified in Dc:Dd, and the 64-bit destination in Dm:Dn.**

```
width = Da[13:8] , offset = Da[5:0] , dest_offset = Db[5:0], src = {Dc.M,Dd.M}
{Dm.M,Dn.M}[width+dest_offset-1:dest_offset] =
              src[(offset + width - 1):offset]
Dm.E = 0, Dn.E = 0
```

Explicit Operands

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM_Y	All

EXTINS.X Extract and Insert a Bit Field From 64-bit Data to a 40-Bit Destination (DALU)

General Description

The instruction allows to specify a bit field of arbitrary offset and width (up to 64 bits) in a 64-bit input, and inserts this field in an arbitrary offset in a 40-bit output. The bits below the inserted field are cleared, and the bits above the inserted field are sign-extended. The input is specified in two pairs of concatenated registers (without the extension), the output is a full D register. The width and offset of the source field, and the offset of the destination field are specified in fixed fields in additional source registers. If the source field spills over the boundary of the 64-bit input, the destination register is cleared. If the destination field spills over the boundary of the 40-bit output, the destination field is updated up to the 40-bit limit. If the width of the field is zero it is treated as having a width of 1 bit.

Instruction Variants

#	Syntax	Description
1	**EXTINS.X Da:Db,Dc:Dd,Dn**	**Extract and insert a bit field for 64-bit data to a 40-bit destination. The offset and width of the extracted field are specified in Da. The offset of the inserted field in the destination is specified in Db. The 64-bit input is specified in Dc:Dd, and the 40-bit destination in Dn.**

```
width = Da[13:8] , offset = Da[5:0] , dest_offset = Db[5:0]
src = {Dc[31:0],Dd[31:0]}
Dn[dest_offset-1:0] = 0
Dn[width+dest_offset-1:dest_offset] = src[(offset + width - 1):offset]
Dn[39:width+dest_offset] = (40-(width+dest_offset)){src[offset + width - 1]}
```

Explicit Operands

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM_Y	All

EXTRACT.nL Extract a Bit Field Into a 32-Bit Destination (DALU)

General Description

SIMD2 Extract of two bit fields with shared offsets and widths (up to 32 bits) from two 32-bit inputs. The extracted fields are placed it in the lower part of the destinations and sign extended. If the source field spills over the 32-bit input boundary, the destination registers are cleared. The extensions are always cleared. If the width of the field is zero it is treated as having a width of 1 bit.

Instruction Variants

#	Syntax	Description
1	**EXTRACT.2L Da,Dc:Dd,Dm:Dn**	**SIMD2 extraction of a bit field from two 32-bit inputs. The offset and width of the extracted fields are specified in Da. The 32-bit inputs are specified in Dc and Dd, and the respective destinations in Dm and Dn.**

```
width = Da[13:8] , offset = Da[5:0]
Dm[(width - 1):0] = Dc[(offset + width - 1):offset]
Dm[31:width] = (S32) {Dm[(width - 1)]}
Dn[(width - 1):0] = Dd[(offset + width - 1):offset]
Dn[31:width] = (S32) {Dn[(width - 1)]}
Dm.E = 0, Dn.E = 0
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM_Y	All

EXTRACT.nX Extract a Bit Field Into a 40-Bit Destination (DALU)

General Description

Extract a bit field from the specified source (either 40 bits or 64 bits). The extracted field is placed at the lower part of the destination, and is either sign or zero extended to 40 bits. The offset and width of the field are specified either as explicit immediate operands or in fixed fields in an input register. If the field spills over the boundary of the input data (40 or 64 bits), the destination is cleared. If the width of the field is zero it is treated as having a width of 1 bit.

Flag Options

Flag	Position	Description
LL	flg1	SIMD multiplication: low by low portions Extract and Sat instructions: 64-bit input
U	flg2	Unsigned

Instruction Variants

#	Syntax	Description
1	**EXTRACT.X Da,Db,Dn**	**Extract a signed field from a 40-bit input. The width and offset are specified as immediate operands**

	`width = max (1,Da[13:8])`
	`offset = Da[5:0]`
	`Dn = (S40) Da[width + offset - 1 : offset]`	

2	**EXTRACT.X #width6,#offset6,Da,Dn**	**Extract a signed field from a 40-bit input. The width and offset are specified in Da.**

	`width = max(1,width6)`
	`Dn = (S40) Da[width + offset6 - 1 : offset6]`	

3	**EXTRACT.LL.X Da,Dc:Dd,Dn**	**Extract a signed field from a 64-bit input, specified in Dc:Dd (without the extensions). The width and offset are specified in Da.**

	`width = Da[13:8] , offset = Da[5:0]`
	`src = {Dc[31:0],Dd[31:0]}`
	`Dn[(width - 1):0] = src[(offset + width - 1):offset]`
	`Dn[39:width] = (S40) Dn[(width - 1)]`	

4	**EXTRACT.U.X Da,Db,Dn**	**Extract an unsigned field from a 40-bit input. The width and offset are specified as immediate operands**

	`width = max(1,Da[13:8])`
	`offset = Da[5:0]`
	`Dn = (U40) Db[width + offset - 1 : offset]`	

5	**EXTRACT.U.X #width6,#offset6,Da,Dn**	**Extract an unsigned field from a 40-bit input. The width and offset are specified in Da.**

Table continues on the next page...

#	Syntax	Description

	`width = max(1,width6)`
	`Dn = (U40) Da[width + offset6 - 1 : offset]`	

6	**EXTRACT.LL.U.X Da,Dc:Dd,Dn**	**Extract an unsigned field from a 64-bit input, specified in Dc:Dd (without the extensions). The width and offset are specified in Da.**

	`width = Da[13:8] , offset = Da[5:0]`
	`src = {Dc[31:0],Dd[31:0]}`
	`Dn[(width - 1):0] = src[(offset + width - 1):offset]`
	`Dn[39:width] = (U40) Dn[(width - 1)]`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	
width6	0 < width6 ≤ 40;	width6 + offset6 ≤ 40
offset6	0 ≤ offset6 ≤ 40;	width6 + offset6 ≤ 40

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	5
	dalu_LBM_Y	1, 2, 3, 4, 6

EXTRACTA Extract a Bit Field From and To an R Register (IPU)

General Description

Extract a bit field from the specified R register, up to 32 bits. The extracted field is placed at the lower part of the R destination, and is either sign or zero extended. If the field spills over the boundary of the input data (32 bits), or the width of the field is 0, the destination is cleared.

Flag Options

Flag	Position	Description
U	flg2	Unsigned

Instruction Variants

#	Syntax	Operation	Description
1	EXTRACTA #U5,#u5,Ra,Rn	`Rn = (S32) Ra[U5+u5-1:u5]`	Extract a signed field
2	EXTRACTA.U #U5,#u5,Ra,Rn	`Rn = (U32) Ra[U5+u5-1:u5]`	Extract an unsigned field

Explicit Operands

Operand	Permitted Values
Ra	R0-R31
Rn	R0-R31
U5	$0 \le U5 < 2^5$
u5	$0 \le u5 < 2^5$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Pipeline behavior	agu_AAU_LOGIC	All

FABS.nSP Floating Point Absolute (DALU)

General Description

Performs a single precision floating point absolute operation.

Instruction Variants

#	Syntax	Operation	Description

1	FABS.2SP Da:Db,Dm:Dn	Dm = |Da|
		Dn = |Db|
			SIMD2 Float Absolute

2	FABS.SP Da,Dn	Dn = |Da|	Float Absolute

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
GCR.FDENI	All
GCR.FINVAL	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_FLT_MISC	All

FADD.nSP Floating Point Add (DALU)

General Description

Performs single precision floating point addition.

Flag Options

Flag	Position	Description
X	flg1	Cross between the arguments

Instruction Variants

#	Syntax	Operation	Description

1	FADD.2SP Da:Db,Dc:Dd,Dm:Dn	`Dm = (U40)(Da.M + Dc.M)`
		`Dn = (U40)(Db.M + Dd.M)`
			SIMD2 Float Add

2	FADD.SP Da,Db,Dn	`Dn = (U40)(Da.M + Db.M)`	Float add

3	FADD.X.2SP Da:Db,Dc:Dd,Dm:Dn	`Dm = (U40)(Da.M + Dd.M)`
		`Dn = (U40)(Db.M + Dc.M)`
			SIMD2 Float add with swapped operands

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
GCR.FDENI	All
GCR.FINEX	All
GCR.FINVAL	All
GCR.FOVER	All
GCR.FUNDR	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_FLT_NUMERIC	All

FADDSUB.2SP Floating Point Add-Sub (DALU)

General Description

Performs single precision floating point addition on first lane and subtraction on the second lane.

Flag Options

Flag	Position	Description
X	flg1	Cross between the arguments

Instruction Variants

#	Syntax	Operation	Description

1	FADDSUB.2SP Da:Db,Dc:Dd,Dm:Dn	`Dm = (U40)(Da.M + Dc.M)`
		`Dn = (U40)(Dd.M - Db.M)`
			SIMD2 Float add-subtract

2	FADDSUB.X.2SP Da:Db,Dc:Dd,Dm:Dn	`Dm = (U40)(Da.M + Dd.M)`
		`Dn = (U40)(Dc.M - Db.M)`
			SIMD2 Float add-sub with swapped operands

Explicit Operands

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Affected By Instructions

Field	Relevant Variants
GCR.FDENI	All
GCR.FINEX	All
GCR.FINVAL	All
GCR.FOVER	All
GCR.FUNDR	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_FLT_NUMERIC	All

FCMP.nSP Floating Point Compare (DALU)

General Description

Performs single precision floating point compare into a single predicate or into to a predicate pair. The comparison type could be one of several conditions: equality (EQ), inequality (NE), greater than (GT), greater or equal (GE), less than (LT) and less or equal (LE). The condition is specified in position FLG1 of the mnemonic. In the variant table below, instructions that operate on the same sources and update the same destinations are described collectively under one entry, where the condition is generically specified as <op>.

Flag Options

Flag	Position	Description
BN	flg1	Bound Comparison
EQ	flg1	Equal comparison
GE	flg1	Greater or Equal comparison
GT	flg1	Greater Than comparison
LE	flg1	Less or Equal comparison
LT	flg1	Less Than comparison
NE	flg1	Not Equal comparison

Instruction Variants

#	Syntax	Description
1	**FCMP.BN.SP Da,Db,Pm:Pn**	**Float Compare Bound into predicate pair**

	`Pm = (-Db ≤ Da ≤ Db)`
	`Pn = ~(new Pm)`	

2	**FCMP.BN.SP Da,Db,Pn**	**Float Compare Bound**
	`Pn = (-Db ≤ Da ≤ Db)`	
3	**FCMP.<*op*>.SP Da,Db,Pm:Pn**	**Float Compare for equal/greater or equal/ greater/not equal/less than/less or equal into predicate pair. "<*op*>" is one of: EQ, GT, GE, NE, LE, LT in the syntax, and one of ==, >, ≥, != ,≤ ,< in the operation, respectively. Aliased variants are LE, LT.**

	`Pm = (Db <op> Da)`
	`Pn = ~(new Pm)`	

4	**FCMP.<*op*>.SP Da,Db,Pn**	**Float Compare for equal/greater or equal/ greater/not equal/less than/less or equal. "<*op*>" is one of: EQ, GT, GE, NE, LE, LT in the syntax, and one of ==, >, ≥, != ,≤ ,< in the operation, respectively. Aliased variants are LE, LT.**
	`Pn = (Db <op> Da)`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Db	D0-D63	
Pm:Pn	$P_n:P_{n+1}$	$0 \le n \le 4$
Pn	P0-P5	

Affected By Instructions

Field	Relevant Variants
GCR.FDENI	All
GCR.FINVAL	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Alias		3, 4
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
IF.EC		All
Pipeline behavior	dalu_FLT_Pbit	All

FFT.nT Fast Fourier Transform (DALU)

General Description

Each variant of this instruction are used in optimized kernels implementing the DFT (Discrete Fourier Transform), FFT (Fast Fourier Transform), IDFT (inverse Discrete Fourier Transform) or IFFT (inverse Fast Fourier Transform) algorithms in a different radix at a different stage. The instructions are application specific and normally will not be used outside these kernels.

For more information refer to documentation and software library implementation.

Flag Options

Flag	Position	Description
R2	flg1	Radix 2
R3	flg1	Radix 3
R4	flg1	Radix 4
R4B	flg1	Radix 4 type B
R5	flg1	Radix 5
I	flg2	Inverse FFT/DFT
P1	flg2	Phase 1
P11	flg2	Phase 1, variant 1
P12	flg2	Phase 1, variant 2
P2	flg2	Phase 2
S	flg2	Saturate
S1	flg2	Stage 1
SN	flg2	Stage N

Instruction Variants

#	Syntax	Description
1	**FFT.R5.2T Da:Db,Dc:Dd,Dn**	**DFT Radix 5 Output OA calculation (in a single phase)**

```
MX.H = ((  Da.H * Dc.H + Da.L * Dc.L)<<1 + 0x8000) >> 16
MX.L = (( -Da.H * Dc.L + Da.L * Dc.H)<<1 + 0x8000) >> 16
MY.H = (( Db.H * Dd.H + Db.L * Dd.L)<<1 + 0x8000) >> 16
MY.L = (( -Db.H * Dd.L + Db.L * Dd.H)<<1 + 0x8000) >> 16
Dn.WH = Dn.WH + MX.H + MY.H
Dn.WL = Dn.WL + MX.L + MY.L
```

Table continues on the next page...

#	Syntax	Description
2	**FFT.R5.4T Da:Db,Dc:Dd,De,Dg:Dh,Di:Dj,Dk,Dm,Dn**	**DFT Radix 5 Output OA calculation (in a single phase)**

```
MX.H = (( Da.H * Dc.H - Da.L * Dc.L)<<1 + 0x8000) >> 16
MX.L = (( Da.H * Dc.L + Da.L * Dc.H)<<1 + 0x8000) >> 16
MY.H = (( Db.H * Dd.H - Db.L * Dd.L)<<1 + 0x8000) >> 16
MY.L = (( Db.H * Dd.L + Db.L * Dd.H)<<1 + 0x8000) >> 16
MZ.H = (( Dg.H * Di.H - Dg.L * Di.L)<<1 + 0x8000) >> 16
MZ.L = (( Dg.H * Di.L + Dg.L * Di.H)<<1 + 0x8000) >> 16
MT.H = (( Dh.H * Dj.H - Dh.L * Dj.L)<<1 + 0x8000) >> 16
MT.L = (( Dh.H * Dj.L + Dh.L * Dj.H)<<1 + 0x8000) >> 16
Dm.WH = De.WH + MX.H + MY.H
Dm.WL = De.WL + MX.L + MY.L
Dn.WH = Dk.WH + MZ.H + MT.H
Dn.WL = Dk.WL + MZ.L + MT.L
```

3	**FFT.R5.I.2T Da:Db,Dc:Dd,Dn**	**IDFT Radix 5 Output OA calculation (in a single phase)**

```
MX.H = ((  Da.H * Dc.H - Da.L * Dc.L)<<1 + 0x8000) >> 16
MX.L = (( Da.H * Dc.L + Da.L * Dc.H)<<1 + 0x8000) >> 16
MY.H = (( Db.H * Dd.H - Db.L * Dd.L)<<1 + 0x8000) >> 16
MY.L = (( Db.H * Dd.L + Db.L * Dd.H)<<1 + 0x8000) >> 16
Dn.WH = Dn.WH + MX.H + MY.H
Dn.WL = Dn.WL + MX.L + MY.L
```

4	**FFT.R5.I.4T Da:Db,Dc:Dd,De,Dg:Dh,Di:Dj,Dk,Dm,Dn**	**IDFT Radix 5 Output OA calculation (in a single phase)**

```
MX.H = (( Da.H * Dc.H + Da.L * Dc.L)<<1 + 0x8000) >> 16
MX.L = ((-Da.H * Dc.L + Da.L * Dc.H)<<1 + 0x8000) >> 16
MY.H = (( Db.H * Dd.H + Db.L * Dd.L)<<1 + 0x8000) >> 16
MY.L = ((-Db.H * Dd.L + Db.L * Dd.H)<<1 + 0x8000) >> 16
MZ.H = (( Dg.H * Di.H + Dg.L * Di.L)<<1 + 0x8000) >> 16
MZ.L = ((-Dg.H * Di.L + Dg.L * Di.H)<<1 + 0x8000) >> 16
MT.H = (( Dh.H * Dj.H + Dh.L * Dj.L)<<1 + 0x8000) >> 16
MT.L = ((-Dh.H * Dj.L + Dh.L * Dj.H)<<1 + 0x8000) >> 16
Dm.WH = De.WH + MX.H + MY.H
Dm.WL = De.WL + MX.L + MY.L
Dn.WH = Dk.WH + MZ.H + MT.H
Dn.WL = Dk.WL + MZ.L + MT.L
```

5	**FFT.R5.IP11.4T Da:Db,Dc:Dd,Dm:Dn**	**IDFT Radix 5 phase 1 type 1**

```
MX.H = ((Da.H * Dc.H + Da.L * Dc.L)<<1 + 0x8000) >> 16
MX.L = ((Da.H * Dc.L - Da.L * Dc.H)<<1 + 0x8000) >> 16
MY.H = ((Db.H * Dd.H + Db.L * Dd.L)<<1 + 0x8000) >> 16
MY.L = ((Db.H * Dd.L - Db.L * Dd.H)<<1 + 0x8000) >> 16
Dm.WH = (MX.H + MY.H) ; (Xr + Yr)
Dm.WL = (MX.L - MY.L) ; (Xi - Yi)
Dn.WH = (MX.H - MY.H) ; (Xr - Yr)
Dn.WL = (MX.L + MY.L) ; (Xi + Yi)
```

6	**FFT.R5.IP12.4T Da:Db,Dc:Dd,Dm:Dn**	**IDFT Radix 5 phase 1 type 2**

```
MX.H = ((Da.H * Dc.H + Da.L * Dc.L)<<1 + 0x8000) >> 16
MX.L = ((Da.H * Dc.L - Da.L * Dc.H)<<1 + 0x8000) >> 16
MY.H = ((Db.H * Dd.H + Db.L * Dd.L)<<1 + 0x8000) >> 16
MY.L = ((Db.H * Dd.L - Db.L * Dd.H)<<1 + 0x8000) >> 16
Dm.WH = (MX.H - MY.H) ; (Xr - Yr)
Dm.WL = (MX.L + MY.L) ; (Xi + Yi)
Dn.WH = (MX.H + MY.H) ; (Xr + Yr)
Dn.WL = (MX.L - MY.L) ; (Xi - Yi)
```

7	**FFT.R3.IP2.8T Da:Db,Dc,Dd,De:Df,Dg,Dh,Du:Dv,Dw:Dx**	**IDFT Radix 3 Phase 2**

Table continues on the next page...

#	Syntax	Description

```
MX.H = ((  Da.H * Dc.H + De.L * Dc.L)<<1 + 0x8000) >> 16
MX.L = (( De.H * Dc.L - Da.L * Dc.H)<<1 + 0x8000) >> 16
MY.H = (( Da.H * Dc.H - De.L * Dc.L)<<1 + 0x8000) >> 16
MY.L = (( -De.H * Dc.L - Da.L * Dc.H)<<1 + 0x8000) >> 16
MZ.H = (( Db.H * Dg.H + Df.L * Dg.L)<<1 + 0x8000) >> 16
MZ.L = (( Df.H * Dg.L - Db.L * Dg.H)<<1 + 0x8000) >> 16
MW.H = (( Db.H * Dg.H - Df.L * Dg.L)<<1 + 0x8000) >> 16
MW.L = (( -Df.H * Dg.L - Db.L * Dg.H)<<1 + 0x8000) >> 16
Du.WH = Dd.WH + MX.H
Du.WL = Dd.WL + MX.L
Dv.WH = Dd.WH + MY.H
Dv.WL = Dd.WL + MY.L
Dw.WH = Dh.WH + MZ.H
Dw.WL = Dh.WL + MZ.L
Dx.WH = Dh.WH + MW.H
Dx.WL = Dh.WL + MW.L
```

#	Syntax	Description
8	**FFT.R5.IP2.8T Da:Db,Dc:Dd,De,Dg:Dh,Di:Dj,Dk,Du:Dv,Dw:Dx**	**IDFT Radix 5 Phase 2**

```
MBE1_Re = ((  Da.H * Dc.H + Dh.H * Dd.H)<<1 + 0x8000) >> 16
MBE2_Re = (( Da.L * Dc.L + Dh.L * Dd.L)<<1 + 0x8000) >> 16
MCD1_Re = (( Da.H * Dd.H + Dh.H * Dc.H)<<1 + 0x8000) >> 16
MCD2_Re = (( Da.L * Dd.L - Dh.L * Dc.L)<<1 + 0x8000) >> 16
MBE1_Im = (( Db.H * Di.L + Dg.H * Dj.L)<<1 + 0x8000) >> 16
MBE2_Im = (( -Db.L * Di.H - Dg.L * Dj.H)<<1 + 0x8000) >> 16
MCD1_Im = (( Db.H * Dj.L - Dg.H * Di.L)<<1 + 0x8000) >> 16
MCD2_Im = (( -Db.L * Dj.H - Dg.L * Di.H)<<1 + 0x8000) >> 16
Du.WH = De.WH + MBE1_Re + MBE2_Re
Du.WL = De.WH + MBE1_Re - MBE2_Re
Dv.WH = De.WH + MCD1_Re + MCD2_Re
Dv.WL = De.WH + MCD1_Re - MCD2_Re
Dw.WH = Dk.WL + MBE1.Im + MBE2.Im
Dw.WL = Dk.WL - MBE1.Im + MBE2.Im
Dx.WH = Dk.WL + MCD1.Im + MCD2.Im
Dx.WL = Dk.WL - MCD1.Im + MCD2.Im
```

#	Syntax	Description
9	**FFT.R4.IS1.8T Da,Dc:Dd,Db,Dg:Dh,Dx:Dy,Dz:Dw**	**IFFT Radix 4 stage 1**

```
MC.H = (( -Dc.H * 0x8000 )<<1 ) >> 16
 MC.L = (( -Dc.L * 0x8000 )<<1 ) >> 16
 MB.H = (( -Dd.H * 0x8000 )<<1 ) >> 16
 MB.L = (( Dg.L * 0x8000 )<<1 ) >> 16
 MD.H = (( -Dh.H * 0x8000 )<<1 ) >> 16
 MD.L = (( Dh.L * 0x8000 )<<1 ) >> 16
 Dx.WH = Da.WH + MB.H + MC.H + MD.H
 Dx.WL = Da.WL - MB.L + MC.L - MD.L
 Dy.WH = Da.WH - MB.H + MC.H - MD.H
 Dy.WL = Da.WL + MB.L + MC.L + MD.L
 Dz.WH = Db.WH + MB.L - MC.H - MD.L
 Dz.WL = Db.WL + MB.H - MC.L - MD.H
 Dw.WH = Db.WH - MB.L - MC.H + MD.L
 Dw.WL = Db.WL - MB.H - MC.L + MD.H
```

Table continues on the next page...

#	Syntax	Description
10	**FFT.R4.ISN.8T Da,Dc:Dd,De:Df,Db,Dg:Dh,Di,Dx:Dy,Dm:Dn**	**IFFT Radix 4 stage N**

	`MC.H = ((  Dc.H * Df.H + Dc.L * Df.L)<<1 + 0x8000) >> 16`
	`MC.L = (( -Dc.H * Df.L + Dc.L * Df.H)<<1 + 0x8000) >> 16`
	`MB.H = (( Dd.H * De.H + Dd.L * De.L)<<1 + 0x8000) >> 16`
	`MB.L = (( Dg.H * De.L - Dg.L * De.H)<<1 + 0x8000) >> 16`
	`MD.H = (( Dh.H * Di.H + Dh.L * Di.L)<<1 + 0x8000) >> 16`
	`MD.L = (( Dh.H * Di.L - Dh.L * Di.H)<<1 + 0x8000) >> 16`
	`Dx.WH = Da.WH + MB.H + MC.H + MD.H`
	`Dx.WL = Da.WL - MB.L + MC.L - MD.L`
	`Dy.WH = Da.WH - MB.H + MC.H - MD.H`
	`Dy.WL = Da.WL + MB.L + MC.L + MD.L`
	`Dm.WH = Db.WH + MB.L - MC.H - MD.L`
	`Dm.WL = Db.WL + MB.H - MC.L - MD.H`
	`Dn.WH = Db.WH - MB.L - MC.H + MD.L`
	`Dn.WL = Db.WL - MB.H - MC.L + MD.H`	

11	**FFT.R4.ISN.8T Da,Dc:Dd,De:Df,Db,Dg:Dh,Di,Dx:Dy,Dz:Dw**	**IFFT Radix 4 stage N (alternate register allocation)**

	`MC.H = ((  Dc.H * Df.H + Dc.L * Df.L)<<1 + 0x8000) >> 16`
	`MC.L = (( -Dc.H * Df.L + Dc.L * Df.H)<<1 + 0x8000) >> 16`
	`MB.H = (( Dd.H * De.H + Dd.L * De.L)<<1 + 0x8000) >> 16`
	`MB.L = (( Dg.H * De.L - Dg.L * De.H)<<1 + 0x8000) >> 16`
	`MD.H = (( Dh.H * Di.H + Dh.L * Di.L)<<1 + 0x8000) >> 16`
	`MD.L = (( Dh.H * Di.L - Dh.L * Di.H)<<1 + 0x8000) >> 16`
	`Dx.WH = Da.WH + MB.H + MC.H + MD.H`
	`Dx.WL = Da.WL - MB.L + MC.L - MD.L`
	`Dy.WH = Da.WH - MB.H + MC.H - MD.H`
	`Dy.WL = Da.WL + MB.L + MC.L + MD.L`
	`Dz.WH = Db.WH + MB.L - MC.H - MD.L`
	`Dz.WL = Db.WL + MB.H - MC.L - MD.H`
	`Dw.WH = Db.WH - MB.L - MC.H + MD.L`
	`Dw.WL = Db.WL - MB.H - MC.L + MD.H`	

12	**FFT.R4B.ISN.8T Da,Dc:Dd,De:Df,Db,Dg:Dh,Di:Dj,Dx:Dy,Dm:Dn**	**IFFT Radix 4 stage n instruction with ABCD order (type B)**

	`MB.H = ((  Dc.H * De.H + Dc.L * De.L)<<1 + 0x8000) >> 16`
	`MB.L = (( Dc.H * De.L - Dc.L * De.H)<<1 + 0x8000) >> 16`
	`MC.H = (( Dd.H * Df.H + Dd.L * Df.L)<<1 + 0x8000) >> 16`
	`MC.L = (( -Dg.H * Di.L + Dg.L * Di.H)<<1 + 0x8000) >> 16`
	`MD.H = (( Dh.H * Dj.H + Dh.L * Dj.L)<<1 + 0x8000) >> 16`
	`MD.L = (( Dh.H * Dj.L - Dh.L * Dj.H)<<1 + 0x8000) >> 16`
	`Dx.WH = Da.WH + MB.H + MC.H + MD.H`
	`Dx.WL = Da.WL - MB.L + MC.L - MD.L`
	`Dy.WH = Da.WH - MB.H + MC.H - MD.H`
	`Dy.WL = Da.WL + MB.L + MC.L + MD.L`
	`Dm.WH = Db.WH + MB.L - MC.H - MD.L`
	`Dm.WL = Db.WL + MB.H - MC.L - MD.H`
	`Dn.WH = Db.WH - MB.L - MC.H + MD.L`
	`Dn.WL = Db.WL - MB.H - MC.L + MD.H`	

13	**FFT.R4B.ISN.8T Da,Dc:Dd,De:Df,Db,Dg:Dh,Di:Dj,Dx:Dy,Dz:Dw**	**IFFT Radix 4 stage n instruction with ABCD order (type B, alternate register allocation)**

Table continues on the next page...

#	Syntax	Description

	MB.H = ((Dc.H * De.H + Dc.L * De.L)<<1 + 0x8000) >> 16
	MB.L = ((Dc.H * De.L - Dc.L * De.H)<<1 + 0x8000) >> 16
	MC.H = ((Dd.H * Df.H + Dd.L * Df.L)<<1 + 0x8000) >> 16
	MC.L = ((-Dg.H * Di.L + Dg.L * Di.H)<<1 + 0x8000) >> 16
	MD.H = ((Dh.H * Dj.H + Dh.L * Dj.L)<<1 + 0x8000) >> 16
	MD.L = ((Dh.H * Dj.L - Dh.L * Dj.H)<<1 + 0x8000) >> 16
	Dx.WH = Da.WH + MB.H + MC.H + MD.H
	Dx.WL = Da.WL - MB.L + MC.L - MD.L
	Dy.WH = Da.WH - MB.H + MC.H - MD.H
	Dy.WL = Da.WL + MB.L + MC.L + MD.L
	Dz.WH = Db.WH + MB.L - MC.H - MD.L
	Dz.WL = Db.WL + MB.H - MC.L - MD.H
	Dw.WH = Db.WH - MB.L - MC.H + MD.L
	Dw.WL = Db.WL - MB.H - MC.L + MD.H	

14	**FFT.R3.ISP1.4T Da:Db,Dc:Dd,Du:Dv**	**IDFT Radix 3 phase 1**

	MX.H = ((Da.H * Dc.H + Da.L * Dc.L)<<1 + 0x8000) >> 16
	MX.L = ((Da.H * Dc.L - Da.L * Dc.H)<<1 + 0x8000) >> 16
	MY.H = ((Db.H * Dd.H + Db.L * Dd.L)<<1 + 0x8000) >> 16
	MY.L = ((Db.H * Dd.L - Db.L * Dd.H)<<1 + 0x8000) >> 16
	Du.WH = (S20) SAT16 (MX.H + MY.H)
	Du.WL = (S20) SAT16 (MX.L + MY.L)
	Dv.WH = (S20) SAT16 (MX.H - MY.H)
	Dv.WL = (S20) SAT16 (MX.L - MY.L)	

15	**FFT.R5.P11.4T Da:Db,Dc:Dd,Dm:Dn**	**DFT Radix 5 phase 1 type 1**

	MX.H = ((Da.H * Dc.H - Da.L * Dc.L)<<1 + 0x8000) >> 16
	MX.L = ((Da.H * Dc.L + Da.L * Dc.H)<<1 + 0x8000) >> 16
	MY.H = ((Db.H * Dd.H - Db.L * Dd.L)<<1 + 0x8000) >> 16
	MY.L = ((Db.H * Dd.L + Db.L * Dd.H)<<1 + 0x8000) >> 16
	Dm.WH = (MX.H + MY.H) ; (Xr + Yr)
	Dm.WL = (MX.L - MY.L) ; (Xi - Yi)
	Dn.WH = (MX.H - MY.H) ; (Xr - Yr)
	Dn.WL = (MX.L + MY.L) ; (Xi + Yi)	

16	**FFT.R5.P12.4T Da:Db,Dc:Dd,Dm:Dn**	**DFT Radix 5 phase 1 type 2**

	MX.H = ((Da.H * Dc.H - Da.L * Dc.L)<<1 + 0x8000) >> 16
	MX.L = ((Da.H * Dc.L + Da.L * Dc.H)<<1 + 0x8000) >> 16
	MY.H = ((Db.H * Dd.H - Db.L * Dd.L)<<1 + 0x8000) >> 16
	MY.L = ((Db.H * Dd.L + Db.L * Dd.H)<<1 + 0x8000) >> 16
	Dm.WH = (MX.H - MY.H) ; (Xr - Yr)
	Dm.WL = (MX.L + MY.L) ; (Xi + Yi)
	Dn.WH = (MX.H + MY.H) ; (Xr + Yr)
	Dn.WL = (MX.L - MY.L) ; (Xi - Yi)	

17	**FFT.R3.P2.8T Da:Db,Dc,Dd,De:Df,Dg,Dh,Du:Dv,Dw:Dx**	**DFT Radix 3 Phase 2**

	MX.H = ((Da.H * Dc.H + De.L * Dc.L)<<1 + 0x8000) >> 16
	MX.L = ((-De.H * Dc.L + Da.L * Dc.H)<<1 + 0x8000) >> 16
	MY.H = ((Da.H * Dc.H - De.L * Dc.L)<<1 + 0x8000) >> 16
	MY.L = ((De.H * Dc.L + Da.L * Dc.H)<<1 + 0x8000) >> 16
	MZ.H = ((Db.H * Dg.H + Df.L * Dg.L)<<1 + 0x8000) >> 16
	MZ.L = ((-Df.H * Dg.L + Db.L * Dg.H)<<1 + 0x8000) >> 16
	MW.H = ((Db.H * Dg.H - Df.L * Dg.L)<<1 + 0x8000) >> 16
	MW.L = ((Df.H * Dg.L + Db.L * Dg.H)<<1 + 0x8000) >> 16
	Du.WH = Dd.WH + MX.H
	Du.WL = Dd.WL + MX.L
	Dv.WH = Dd.WH + MY.H
	Dv.WL = Dd.WL + MY.L
	Dw.WH = Dh.WH + MZ.H
	Dw.WL = Dh.WL + MZ.L
	Dx.WH = Dh.WH + MW.H
	Dx.WL = Dh.WL + MW.L	

Table continues on the next page...

#	Syntax	Description
18	**FFT.R5.P2.8T Da:Db,Dc:Dd,De,Dg:Dh,Di:Dj,Dk,Du:Dv,Dw:Dx**	**DFT Radix 5 Phase 2**

```
MBE1_Re = ((  Da.H * Dc.H + Dh.H * Dd.H)<<1 + 0x8000) >> 16
MBE2_Re = (( Da.L * Dc.L + Dh.L * Dd.L)<<1 + 0x8000) >> 16
MCD1_Re = (( Da.H * Dd.H + Dh.H * Dc.H)<<1 + 0x8000) >> 16
MCD2_Re = (( Da.L * Dd.L - Dh.L * Dc.L)<<1 + 0x8000) >> 16
MBE1_Im = (( -Db.H * Di.L - Dg.H * Dj.L)<<1 + 0x8000) >> 16
MBE2_Im = (( Db.L * Di.H + Dg.L * Dj.H)<<1 + 0x8000) >> 16
MCD1_Im = (( -Db.H * Dj.L + Dg.H * Di.L)<<1 + 0x8000) >> 16
MCD2_Im = (( Db.L * Dj.H + Dg.L * Di.H)<<1 + 0x8000) >> 16
Du.WH = De.WH + MBE1_Re + MBE2_Re
Du.WL = De.WH + MBE1_Re - MBE2_Re
Dv.WH = De.WH + MCD1_Re + MCD2_Re
Dv.WL = De.WH + MCD1_Re - MCD2_Re
Dw.WH = Dk.WL + MBE1_Im + MBE2_Im
Dw.WL = Dk.WL - MBE1_Im + MBE2_Im
Dx.WH = Dk.WL + MCD1_Im + MCD2_Im
Dx.WL = Dk.WL - MCD1_Im + MCD2_Im
```

19	**FFT.R2.S1.4T Da,Db,Dz:Dw**	**FFT/IFFT Radix 2 stage 1**

```
Dz.WH = Db.WH + Da.WH
Dz.WL = Db.WL + Da.WL
Dw.WH = Db.WH - Da.WH
Dw.WL = Db.WL - Da.WL
```

20	**FFT.R4.S1.8T Da,Dc:Dd,Db,Dg:Dh,Dx:Dy,Dz:Dw**	**FFT Radix 4 stage 1**

```
MC.H = (( -Dc.H * 0x8000 )<<1 ) >> 16
 MC.L = (( -Dc.L * 0x8000 )<<1 ) >> 16
 MB.H = (( -Dd.H * 0x8000 )<<1 ) >> 16
 MB.L = (( -Dg.L * 0x8000 )<<1 ) >> 16
 MD.H = (( -Dh.H * 0x8000 )<<1 ) >> 16
 MD.L = (( -Dh.L * 0x8000 )<<1 ) >> 16
 Dx.WH = Da.WH + MB.H + MC.H + MD.H
 Dx.WL = Da.WL + MB.L + MC.L + MD.L
 Dy.WH = Da.WH - MB.H + MC.H - MD.H
 Dy.WL = Da.WL - MB.L + MC.L - MD.L
 Dz.WH = Db.WH + MB.L - MC.H - MD.L
 Dz.WL = Db.WL - MB.H - MC.L + MD.H
 Dw.WH = Db.WH - MB.L - MC.H + MD.L
 Dw.WL = Db.WL + MB.H - MC.L - MD.H
```

21	**FFT.R4.SN.8T Da,Dc:Dd,De:Df,Db,Dg:Dh,Di,Dx:Dy,Dm:Dn**	**FFT Radix 4 stage N**

```
MC.H = ((  Dc.H * Df.H - Dc.L * Df.L)<<1 + 0x8000) >> 16
MC.L = (( Dc.H * Df.L + Dc.L * Df.H)<<1 + 0x8000) >> 16
MB.H = (( Dd.H * De.H - Dd.L * De.L)<<1 + 0x8000) >> 16
MB.L = (( Dg.H * De.L + Dg.L * De.H)<<1 + 0x8000) >> 16
MD.H = (( Dh.H * Di.H - Dh.L * Di.L)<<1 + 0x8000) >> 16
MD.L = (( Dh.H * Di.L + Dh.L * Di.H)<<1 + 0x8000) >> 16
Dx.WH = Da.WH + MB.H + MC.H + MD.H
Dx.WL = Da.WL + MB.L + MC.L + MD.L
Dy.WH = Da.WH - MB.H + MC.H - MD.H
Dy.WL = Da.WL - MB.L + MC.L - MD.L
Dm.WH = Db.WH + MB.L - MC.H - MD.L
Dm.WL = Db.WL - MB.H - MC.L + MD.H
Dn.WH = Db.WH - MB.L - MC.H + MD.L
Dn.WL = Db.WL + MB.H - MC.L - MD.H
```

Table continues on the next page...

#	Syntax	Description
22	**FFT.R4.SN.8T Da,Dc:Dd,De:Df,Db,Dg:Dh,Di,Dx:Dy,Dz:Dw**	**FFT Radix 4 stage N (alternate register allocation)**

	`MC.H = ((  Dc.H * Df.H - Dc.L * Df.L)<<1 + 0x8000)  >> 16`
	`MC.L = (( Dc.H * Df.L + Dc.L * Df.H)<<1 + 0x8000)  >> 16`
	`MB.H = (( Dd.H * De.H - Dd.L * De.L)<<1 + 0x8000)  >> 16`
	`MB.L = (( Dg.H * De.L + Dg.L * De.H)<<1 + 0x8000)  >> 16`
	`MD.H = (( Dh.H * Di.H - Dh.L * Di.L)<<1 + 0x8000)  >> 16`
	`MD.L = (( Dh.H * Di.L + Dh.L * Di.H)<<1 + 0x8000)  >> 16`
	`Dx.WH = Da.WH + MB.H + MC.H + MD.H`
	`Dx.WL = Da.WL + MB.L + MC.L + MD.L`
	`Dy.WH = Da.WH - MB.H + MC.H - MD.H`
	`Dy.WL = Da.WL - MB.L + MC.L - MD.L`
	`Dz.WH = Db.WH + MB.L - MC.H - MD.L`
	`Dz.WL = Db.WL - MB.H - MC.L + MD.H`
	`Dw.WH = Db.WH - MB.L - MC.H + MD.L`
	`Dw.WL = Db.WL + MB.H - MC.L - MD.H`	

23	**FFT.R4B.SN.8T Da,Dc:Dd,De:Df,Db,Dg:Dh,Di:Dj,Dx:Dy,Dm:Dn**	**FFT Radix 4 stage n instruction with ABCD order (type B)**

	`MB.H = ((  Dc.H * De.H - Dc.L * De.L)<<1 + 0x8000)  >> 16`
	`MB.L = (( Dc.H * De.L + Dc.L * De.H)<<1 + 0x8000)  >> 16`
	`MC.H = (( Dd.H * Df.H - Dd.L * Df.L)<<1 + 0x8000)  >> 16`
	`MC.L = (( Dg.H * Di.L + Dg.L * Di.H)<<1 + 0x8000)  >> 16`
	`MD.H = (( Dh.H * Dj.H - Dh.L * Dj.L)<<1 + 0x8000)  >> 16`
	`MD.L = (( Dh.H * Dj.L + Dh.L * Dj.H)<<1 + 0x8000)  >> 16`
	`Dx.WH = Da.WH + MB.H + MC.H + MD.H`
	`Dx.WL = Da.WL + MB.L + MC.L + MD.L`
	`Dy.WH = Da.WH - MB.H + MC.H - MD.H`
	`Dy.WL = Da.WL - MB.L + MC.L - MD.L`
	`Dm.WH = Db.WH + MB.L - MC.H - MD.L`
	`Dm.WL = Db.WL - MB.H - MC.L + MD.H`
	`Dn.WH = Db.WH - MB.L - MC.H + MD.L`
	`Dn.WL = Db.WL + MB.H - MC.L - MD.H`	

24	**FFT.R4B.SN.8T Da,Dc:Dd,De:Df,Db,Dg:Dh,Di:Dj,Dx:Dy,Dz:Dw**	**FFT Radix 4 stage n instruction with ABCD order (type B, alternate register allocation)**

	`MB.H = ((  Dc.H * De.H - Dc.L * De.L)<<1 + 0x8000)  >> 16`
	`MB.L = (( Dc.H * De.L + Dc.L * De.H)<<1 + 0x8000)  >> 16`
	`MC.H = (( Dd.H * Df.H - Dd.L * Df.L)<<1 + 0x8000)  >> 16`
	`MC.L = (( Dg.H * Di.L + Dg.L * Di.H)<<1 + 0x8000)  >> 16`
	`MD.H = (( Dh.H * Dj.H - Dh.L * Dj.L)<<1 + 0x8000)  >> 16`
	`MD.L = (( Dh.H * Dj.L + Dh.L * Dj.H)<<1 + 0x8000)  >> 16`
	`Dx.WH = Da.WH + MB.H + MC.H + MD.H`
	`Dx.WL = Da.WL + MB.L + MC.L + MD.L`
	`Dy.WH = Da.WH - MB.H + MC.H - MD.H`
	`Dy.WL = Da.WL - MB.L + MC.L - MD.L`
	`Dz.WH = Db.WH + MB.L - MC.H - MD.L`
	`Dz.WL = Db.WL - MB.H - MC.L + MD.H`
	`Dw.WH = Db.WH - MB.L - MC.H + MD.L`
	`Dw.WL = Db.WL + MB.H - MC.L - MD.H`	

25	**FFT.R3.SP1.4T Da:Db,Dc:Dd,Du:Dv**	**DFT Radix 3 phase 1**

	`MX.H = ((  Da.H * Dc.H - Da.L * Dc.L)<<1 + 0x8000)  >> 16`
	`MX.L = (( Da.H * Dc.L + Da.L * Dc.H)<<1 + 0x8000)  >> 16`
	`MY.H = (( Db.H * Dd.H - Db.L * Dd.L)<<1 + 0x8000)  >> 16`
	`MY.L = (( Db.H * Dd.L + Db.L * Dd.H)<<1 + 0x8000)  >> 16`
	`Du.WH = (S20) SAT16 (MX.H + MY.H)`
	`Du.WL = (S20) SAT16 (MX.L + MY.L)`
	`Dv.WH = (S20) SAT16 (MX.H - MY.H)`
	`Dv.WL = (S20) SAT16 (MX.L - MY.L)`	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Da:Db	$D_n:D_{n+1}$ 0 ≤ n ≤ 62
Db	D0-D63
Dc	D0-D63
Dc:Dd	$D_n:D_{n+1}$ 0 ≤ n ≤ 62
Dd	D0-D63
De	D0-D63
De:Df	$D_n:D_{n+1}$ 0 ≤ n ≤ 62
Dg	D0-D63
Dg:Dh	$D_n:D_{n+1}$ 0 ≤ n ≤ 62
Dh	D0-D63
Di	D0-D63
Di:Dj	$D_n:D_{n+1}$ 0 ≤ n ≤ 62
Dk	D0-D63
Dm	D0-D63
Dm:Dn	$D_n:D_{n+1}$ 0 ≤ n ≤ 62
Dn	D0-D63
Du:Dv	$D_n:D_{n+9}$ 0 ≤ n ≤ 54 Explicit list of all allowed combinations at Table C-18
Dw:Dx	$D_n:D_{n+9}$ 0 ≤ n ≤ 54 Explicit list of all allowed combinations at Table C-19
Dx:Dy	$D_n:D_{(n\&0x38+8)\%64+(n+3)\%8}$ 0 ≤ n ≤ 63 Explicit list of all allowed combinations at Table C-20
Dz:Dw	$D_n:D_{(n\&0x38+8)\%64+(n+3)\%8}$ 0 ≤ n ≤ 63 Explicit list of all allowed combinations at Table C-21

Affected By Instructions

Field	Relevant Variants
SR.SAT	14, 25

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	12, 13, 23, 24
Encoding length	32-bits in 64	1, 3, 5, 6, 14, 15, 16, 19, 25
	64-bits	2, 4, 7, 8, 9, 10, 11, 12, 13, 17, 18, 20, 21, 22, 23, 24
Execution unit	2xDALU	2, 4, 7, 8, 9, 10, 11, 12, 13, 17, 18, 20, 21, 22, 23, 24
	DALU	1, 3, 5, 6, 14, 15, 16, 19, 25
Pipeline behavior	dalu_DAU	19
	dalu_MPY	5, 6, 14, 15, 16, 25
	dalu_MPY_Acc	1, 2, 3, 4, 7, 8, 9, 10, 11, 12, 13, 17, 18, 20, 21, 22, 23, 24

FINVSQRT.nSP Floating Point Inverse Square Root (DALU)

General Description

Performs single precision floating point lookup table search for determining the Inverse Square Root of the input. For the final result, linear interpolation should be applied, which achieves accuracy of 14 bits in the mantissa. Further accuracy can be achieved using a Newton-Raphson iteration.
The following code is an example for the sequence:

```
FINVSQRT.2SP d0,d1:d2 ; d0 = input
FINVSQRT.PRE d0,d3
FMADD.INVSQ.SP d3,d1,d2
FMPY.SP d2,d2,d1 ; begin Newton-Raphson iteration
FMPY.SP d2,d4,d6
FMSUB.SP d0,d1,d5
FMPY.SP d6,d5,d8 ; d8 = output
```

Flag Options

Flag	Position	Description
PRE	flg1	Preliminary

Instruction Variants

#	Syntax	Description
1	**FINVSQRT.2SP Da,Dm:Dn**	**Reciprocal Square Root Approximation. The output are coefficients that can be used to linear interpolate the result.**

```
in =   { ~Da[23:23] , Da[22:18]}  ; 6b table entry
A = invsqrt_LUT_A( in ) ; 16b
tmp1 = { Da[23:23] , A[14:14] , A[15:15] }
casex tmp1
 00x : Dm = { 1_0111_1100 , A[12:0] , 00_0000_0000 }
 01x : Dm = { 1_0111_1101 , A[13:0] , 0_0000_0000 }
 1x0 : Dm = { 1_0111_1100 , A[13:0] , 0_0000_0000 }
 1x1 : Dm = { 1_0111_1101 , A[14:0] , 0000_0000 }
tmp2 = Da[23:23] ? (191 - (Da[30:23] >>1) ) : ( 190 - (Da[30:23] >>1) )
tmp3 = tmp2 - 1
B = invsqrt_LUT_B( in) ; 16b
if (B[15:15]==0 and Da[23:23]==0)
   Dn = { Da[31:31] , tmp3 , B[13:0] , 0_0000_0000 }
else
   Dn = { Da[31:31] , tmp2 , B[14:0] , 0000_0000 }
if (Da[30:23]]== 0x0) ; (zero or denorm)
 Dm = 0x00000000 ; +zero
 Dn = {Da[31:31] , 31h7F800000} ; inf , same sign as input
```

Table continues on the next page...

#	Syntax	Description
2	**FINVSQRT.PRE Da,Dn**	**Inverse Square Root Preparation operation**

```
If (Exp(Da) is Odd)
   Dn[30:23] = ( 191 - (Da[30:23] >>1) ) ; (-(Exp(Db) -127 + 1)/2 + 1 +127)
Else
   Dn[30:23] = ( 190 - (Da[30:23] >>1) ); (-(Exp(Db) -127)/2 +127)
Dn.E = 0
Rest of Dn bits = Da
```

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Dm:Dn	$D_n:D_{n+1}$ $0 \le n \le 62$
Dn	D0-D63

Affected By Instructions

Field	Relevant Variants
GCR.FDENI	1
GCR.FDIVZ	1
GCR.FINVAL	1

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU_Y	All

FIX2FLT.nSP Convert Fixed Point to Floating Point (DALU)

General Description

Converts 32-bit fixed point data into single precision floating point format, with rounding and scaling. For converting as Q31 fraction the scaling factor should be 31. For converting as integer the scaling factor should be 0.

Flag Options

Flag	Position	Description
L	flg1	32 bit input
U	flg1	Unsigned input

Instruction Variants

#	Syntax	Description
1	**FIX2FLT.L.2SP Da,Dc:Dd,Dm:Dn**	**SIMD2 32-bit fixed point to floating point. Scale factor taken from Da register.**

	`Dm = (float) ((S32)Dc / 2 ** ((S32)Da.LL))`
	`Dn = (float) ((S32)Dd / 2 ** ((S32)Da.LL))`	

2	**FIX2FLT.L.2SP #u5,Da:Db,Dm:Dn**	**SIMD2 32-bit fixed point to floating point. Scale factor taken from immediate.**

	`Dm = (float) ((S32)Da / 2 ** (U32)u5)`
	`Dn = (float) ((S32)Db / 2 ** (U32)u5)`	

3	**FIX2FLT.L.SP Da,Db,Dn**	**32-bit fixed point to floating point. Scale factor taken from Da register.**
	`Dn = (float) ((S32)Db / 2 ** ((S32)Da.LL))`	
4	**FIX2FLT.L.SP #u5,Da,Dn**	**32-bit fixed point to floating point. Scale factor taken from immediate.**
	`Dn = (float) ((S32)Da / 2 ** (U32)u5)`	
5	**FIX2FLT.UL.2SP Da,Dc:Dd,Dm:Dn**	**SIMD2 32-bit unsigned fixed point to floating point. Scale factor taken from Da register.**

	`Dm = (float) ((U32)Dc / 2 ** ((S32)Da.LL))`
	`Dn = (float) ((U32)Dd / 2 ** ((S32)Da.LL))`	

6	**FIX2FLT.UL.2SP #u5,Da:Db,Dm:Dn**	**SIMD2 32-bit unsigned fixed point to floating point. Scale factor taken from immediate.**

	`Dm = (float) ((U32)Da / 2 ** (U32)u5)`
	`Dn = (float) ((U32)Db / 2 ** (U32)u5)`	

7	**FIX2FLT.UL.SP Da,Db,Dn**	**32-bit unsigned fixed point to floating point. Scale factor taken from Da register.**
	`Dn = (float) ((U32)Db / 2 ** ((S32)Da.LL))`	

Table continues on the next page...

#	Syntax	Description
8	**FIX2FLT.UL.SP #u5,Da,Dn**	**32-bit unsigned fixed point to floating point. Scale factor taken from immediate.**
	`Dn = (float) ((U32)Da / 2 ** (U32)u5)`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	
u5	$0 \le u5 < 2^5$	

Affected By Instructions

Field	Relevant Variants
GCR.FINEX	All
GCR.FOVER	All
GCR.FUNDR	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_FLT_FROM_FIXED	All

FLOG2 Floating Point Log2 (DALU)

General Description

Performs single precision floating point lookup table search for determining the base 2 Log of the input. For the final result, linear interpolation should be applied, which achieves accuracy of 15 bits in the mantissa.
The following code is an example for the sequence:

```
FLOG2 d0,d1:d2 ; d0 = input
FLOG2.PRE1 d0,d3
FLOG2.PRE2 d0,d4
FMADD.LOG2.SP d0,d3,d2
FIX2FLT.L.SP #0,d4,d5
FADD.SP d2,d5,d8 ; d8 = output
```

Flag Options

Flag	Position	Description
PRE1	flg1	Preliminary 1
PRE2	flg1	Preliminary 2

Instruction Variants

#	Syntax	Description
1	**FLOG2 Da,Dm:Dn**	**Float Log2 approximation. The output are coefficients that can be used to linear interpolate the result.**

```
A = LOG2_LUT_A(Da[22:17]) ; 18b
B = LOG2_LUT_B(Da[22:17]) ; 17b
if Da[22:17] < 23
  Dm = {0_0111_1110 , A[15:0] , 000_0000}
else
  Dm = {0_0111_1111 , A[16:0] , 00_0000}
if B[15:15] = 0
  tmp = {1_0111_1101 , B[13:0] , 0_0000_0000}
else
  tmp = {1_0111_1110 , B[14:0] , 0000_0000}
if B[16:16] = 0
  Dn = tmp
else
  Dn = {1_0111_1111 , B[15:0] , 000_0000}
if (Exp(Da)== 0x0) ; in = +0,+denorm
   Dm = 0x0 ; +0
   Dn = 0xff800000 ; -inf
if (Exp(Da)== 0xff) ; in = +inf
   Dm = 0x0 ; +0
   Dn = 0x7f800000 ; + inf
if Da < 0 ; input is negative → a = b = QNaN output - note this has highest priority
    Dm = 7FFF_FFFF
    Dn = 7FFF_FFFF
```

Table continues on the next page...

#	Syntax	Description
2	**FLOG2.PRE1 Da,Dn**	**Float Log2 first Preparation operation**

	`Dn[30:23] = 0x7f    ; 0x7f → exp(x)`
	`Dn[22:0]  = Da[22:0]`
	`Dn[39:31] = 0`	

3	**FLOG2.PRE2 Da,Dn**	**Float Log2 second Preparation operation**

	`Dn.M = (S32) (Da[30:23]  - 127)   ;  exp(x) → int, range is -127...128`
	`Dn.E = 0`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
GCR.FDENI	1
GCR.FDIVZ	1
GCR.FINVAL	1

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	2
	dalu_DAU_Y	1, 3

FLT2FIX.nL Convert Floating Point to 32-bit Fixed Point (DALU)

General Description

Converts single precision floating point data into a 32-bit fixed point format, with scaling. Note that rounding is not performed (excessive bits are truncated). For converting as Q31 fraction the scaling factor should be 31. For converting as integer the scaling factor should be 0.

Flag Options

Flag	Position	Description
SP	flg1	Floating Point Single Precision

Instruction Variants

#	Syntax	Description
1	**FLT2FIX.SP.L Da,Db,Dn**	**Floating point to 32-bit fixed point. Scaled factor taken from Da register.**

```
SCALE = (S8) Da[7:0]
Dn.M = (int) (Db.M) * 2**SCALE
Dn.E = 0
Scale=0 means that the dot is to the right of the LSB.
No rounding is preformed.
```

#	Syntax	Description
2	**FLT2FIX.SP.L #u5,Da,Dn**	**Floating point to 32-bit fixed point. Scaled factor taken from immediate.**

```
SCALE = u5
Dn.M = (int) (Da.M) * 2**SCALE
Dn.E = 0
Scale=0 means that the dot is to the right of the LSB.
No rounding is preformed.
```

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Db	D0-D63
Dn	D0-D63
u5	$0 \le u5 < 2^5$

Affected By Instructions

Field	Relevant Variants
GCR.FDENI	All
GCR.FINVAL	All
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_FLT_TO_FIXED	All

FLT2FIX.nX Convert Floating Point to 40-bit Fixed Point (DALU)

General Description

Converts single precision floating point data into 40-bit fixed point format, with scaling. Note that rounding is not performed (excessive bits are truncated). For converting as Q31 fraction the scaling factor should be 31. For converting as integer the scaling factor should be 0.

Flag Options

Flag	Position	Description
SP	flg1	Floating Point Single Precision

Instruction Variants

#	Syntax	Description
1	**FLT2FIX.SP.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 Floating point to 40-bit fixed point. Scale factor taken from Da register.**

	SCALE = (S8) Da[7:0]
	Dm = (int40) (Dc.M) * 2**SCALE
	Dn = (int40) (Dd.M) * 2**SCALE
	Scale=0 means that the dot is to the right of the LSB.
	No rounding is preformed.	

2	**FLT2FIX.SP.2X #u5,Da:Db,Dm:Dn**	**SIMD2 Floating point to 40-bit fixed point. Scale factor taken from immediate.**

	SCALE = u5
	Dm = (int40) (Dc.M) * 2**SCALE
	Dn = (int40) (Dd.M) * 2**SCALE
	Scale=0 means that the dot is to the right of the LSB.
	No rounding is preformed.	

3	**FLT2FIX.SP.X Da,Db,Dn**	**Floating point to 40-bit fixed point. Scale factor taken from Da register.**

	SCALE = (S8) Da[7:0]
	Dn = (int40) (Db.M) * 2**SCALE
	Scale=0 means that the dot is to the right of the LSB.
	No rounding is preformed.	

4	**FLT2FIX.SP.X #u5,Da,Dn**	**Floating point to 40-bit fixed point. Scale factor taken from immediate.**

	SCALE = u5
	Dn = (int40) (Da.M) * 2**SCALE
	Scale=0 means that the dot is to the right of the LSB.
	No rounding is preformed.	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \leq n \leq 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \leq n \leq 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \leq n \leq 62$
Dn	D0-D63	
u5	$0 \leq u5 < 2^5$	

Affected By Instructions

Field	Relevant Variants
GCR.FDENI	All
GCR.FINVAL	All
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_FLT_TO_FIXED	All

FMADD.nSP Floating Point Fused Multiply-Add (DALU)

General Description

Performs single precision floating point fused multiply-add operation.

Flag Options

Flag	Position	Description
INVSQ	flg2	Inverse Square Root calculation assist
LOG2	flg2	Log base 2 calculation assist
RECIP	flg2	Reciprocal calculation assist

Instruction Variants

<table>
<tr><th>#</th><th>Syntax</th><th>Description</th></tr>
<tr><td>1</td><td><b>FMADD.2SP Da:Db,Dc:Dd,Dm:Dn</b></td><td><b>SIMD2 fused multiply-add</b></td></tr>
<tr><td></td><td colspan="2"><pre>Dm = (U40)(Dm.M + Da.M * Dc.M)
Dn = (U40)(Dn.M + Db.M * Dd.M)</pre></td></tr>
<tr><td>2</td><td><b>FMADD.SP Da,Db,Dn</b></td><td><b>Single fused multiply-add</b></td></tr>
<tr><td></td><td colspan="2"><pre>Dn = (U40)(Dn.M + Da.M * Db.M)</pre></td></tr>
<tr><td>3</td><td><b>FMADD.INVSQ.SP Da,Db,Dn</b></td><td><b>Fused multiply-add deducted for linear interpolation of Inverse Square Root. See FINVSQRT instruction for usage.</b></td></tr>
<tr><td></td><td colspan="2"><pre>Dn = (U40)Dn.M + Da.M * Db.M)
if (! (is_NaN (Dn.M) || is_NaN (Da.M) || is_NaN (Db.M))) {
        if ((Da[30:23] == 0xfe) || (Da[30:23] == 0xff)) {
            Dn = {Da[31], 0b00000000000000000000000000000000}
}
}</pre></td></tr>
<tr><td>4</td><td><b>FMADD.LOG2.SP Da,Db,Dn</b></td><td><b>Fused multiply-add deducted for linear interpolation of Log2. See FLOG2 instruction for usage.</b></td></tr>
<tr><td></td><td colspan="2"><pre>Dn = (U40)Dn.M + Da.M * Db.M)
if (! (is_NaN (Dn.M) || is_NaN (Da.M) || is_NaN (Db.M))) {
        if ((Da[30:23] == 0xfe) || (Da[30:23] == 0xff)) {
            Dn = {Da[31], 0b00000000000000000000000000000000}
}
}</pre></td></tr>
<tr><td>5</td><td><b>FMADD.RECIP.SP Da,Db,Dn</b></td><td><b>Fused multiply-add deducted for linear interpolation of Reciprocal. See FRECIP instruction for usage.</b></td></tr>
</table>

Table continues on the next page...

#	Syntax	Description

```
Dn = (U40)Dn.M + Da.M * Db.M)
if (! (is_NaN (Dn.M) || is_NaN (Da.M) || is_NaN (Db.M))) {
        if ((Da[30:23] == 0xff)) {
            Dn = {Da[31], 0b0000000000000000000000000000000}
}
}
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
GCR.FDENI	All
GCR.FINEX	All
GCR.FINVAL	All
GCR.FOVER	All
GCR.FUNDR	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_FLT_NUMERIC	All

FMPY.nSP Floating Point Multiply (DALU)

General Description

Performs single precision floating point multiply operation.

Flag Options

Flag	Position	Description
NN	flg1	Negative Negative
NP	flg1	Negative Positive
PN	flg1	Positive Negative
PP	flg1	Positive Positive
X	flg1	Cross between the arguments

Instruction Variants

#	Syntax	Operation	Description

1	FMPY.2SP Da,Dc:Dd,Dm:Dn	Dm = (U40)(Da.M * Dc.M)
		Dn = (U40)(Da.M * Dd.M)
			SIMD2 multiply by one value

2	FMPY.SP Da,Db,Dn	Dn = (U40)(Da.M * Db.M)	Single multiply

3	FMPY.NN.2SP Da,Dc:Dd,Dm:Dn	Dm = (U40)- (Da.M * Dc.M)
		Dn = (U40)- (Da.M * Dd.M)
			SIMD2 multiply by one value, both products are negated.

4	FMPY.NNX.2SP Da,Dc:Dd,Dm:Dn	Dm = (U40)- (Da.M * Dd.M)
		Dn = (U40)- (Da.M * Dc.M)
			SIMD2 multiply by one value, with swapped operands, both products are negated.

5	FMPY.NP.2SP Da,Dc:Dd,Dm:Dn	Dm = (U40)- (Da.M * Dc.M)
		Dn = (U40)(Da.M * Dd.M)
			SIMD2 multiply by one value, only first product is negated.

6	FMPY.NPX.2SP Da,Dc:Dd,Dm:Dn	Dm = (U40)- (Da.M * Dd.M)
		Dn = (U40)(Da.M * Dc.M)
			SIMD2 multiply by one value, with swapped operands, only first product is negated.

7	FMPY.PN.2SP Da,Dc:Dd,Dm:Dn	Dm = (U40)(Da.M * Dc.M)
		Dn = (U40)- (Da.M * Dd.M)
			SIMD2 multiply by one value, only second product is negated.

8	FMPY.PNX.2SP Da,Dc:Dd,Dm:Dn	Dm = (U40)(Da.M * Dd.M)
		Dn = (U40)- (Da.M * Dc.M)
			SIMD2 multiply by one value, with swapped operands, only second product is negated.

9	FMPY.PP.2SP Da:Db,Dc:Dd,Dm:Dn	Dm = (U40)(Da.M * Dc.M)
		Dn = (U40)(Db.M * Dd.M)
			SIMD2 multiply

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$

Table continues on the next page...

Operand	Permitted Values	
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
GCR.FDENI	All
GCR.FINEX	All
GCR.FINVAL	All
GCR.FOVER	All
GCR.FUNDR	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_FLT_NUMERIC	All

FMSOD.2SP Floating Point Fused Multiply-Add and Multiply-Subtract (DALU)

General Description

Performs single precision floating point fused multiply-add and multiply-subtract operation.

Flag Options

Flag	Position	Description
AAI	flg1	Add respective portions, without crossing
ASI	flg1	A: Add respective portions. S: Subtract respective portions. I: No cross of operand portions
ASX	flg1	A: Add respective portions. S: Subtract respective portions. X: Cross of operand portions
SAI	flg1	A: add respective portions. S: subtract respective portions. I: no cross of operand portions
SAX	flg1	A: add respective portions. S: subtract respective portions. X: cross of operand portions
SSI	flg1	Subtract respective portions, without crossing

Instruction Variants

#	Syntax	Description
1	**FMSOD.AAI.2SP Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - fused multiply-add, both products are added.**

	`Dm = (U40)(Dm.M + Da.M * Dc.M)`
	`Dn = (U40)(Dn.M + Da.M * Dd.M)`	

2	**FMSOD.ASI.2SP Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - fused multiply-add, first product is added and second product is subtracted.**

	`Dm = (U40)(Dm.M + Da.M * Dc.M)`
	`Dn = (U40)(Dn.M - Da.M * Dd.M)`	

3	**FMSOD.ASX.2SP Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - fused multiply-add with swapped operands, first product is added and second product is subtracted.**

	`Dm = (U40)(Dm.M + Da.M * Dd.M)`
	`Dn = (U40)(Dn.M - Da.M * Dc.M)`	

4	**FMSOD.SAI.2SP Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - fused multiply-add, first product is subtracted and second product is added.**

	`Dm = (U40)(Dm.M - Da.M * Dc.M)`
	`Dn = (U40)(Dn.M + Da.M * Dd.M)`	

5	**FMSOD.SAX.2SP Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - fused multiply-add with swapped operands, first product is subtracted and second product is added.**

	`Dm = (U40)(Dm.M - Da.M * Dd.M)`
	`Dn = (U40)(Dn.M + Da.M * Dc.M)`	

Table continues on the next page...

#	Syntax	Description
6	**FMSOD.SSI.2SP Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - fused multiply-add, both products are subtracted.**

```
Dm = (U40)(Dm.M - Da.M * Dc.M)
Dn = (U40)(Dn.M - Da.M * Dd.M)
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \leq n \leq 62$
Dm:Dn	D_n:D_{n+1}	$0 \leq n \leq 62$

Affected By Instructions

Field	Relevant Variants
GCR.FDENI	All
GCR.FINEX	All
GCR.FINVAL	All
GCR.FOVER	All
GCR.FUNDR	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_FLT_NUMERIC	All

FMSUB.nSP Floating Point Fused Multiply-Subtract (DALU)

General Description

Performs single precision floating point fused multiply-subtract operation.

Instruction Variants

#	Syntax	Description
1	**FMSUB.2SP Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fused multiply-subtract**

	`Dm = (U40)(Dm.M - Da.M * Dc.M)`
	`Dn = (U40)(Dn.M - Db.M * Dd.M)`	

2	**FMSUB.SP Da,Db,Dn**	**Single fused multiply-subtract**
	`Dn = (U40)(Dn.M - Da.M * Db.M)`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
GCR.FDENI	All
GCR.FINEX	All
GCR.FINVAL	All
GCR.FOVER	All
GCR.FUNDR	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_FLT_NUMERIC	All

FND.nT Find Maximum or Minimum 20-bit Value (DALU)

General Description

This instruction accelerate the search of a maximum or minimum 20-bit value and it's index within a long vector of 20-bit values.
The instruction read four 20-bit values out of the vector (Da.WH, Da.WL, Db.WH and Db.WL), current maximum/minimum value (stored in Dm.M in a 32-bit integer format) and two indexes - Dn.H holds index to the first of four read values while Dn.L holds the index of the current maximum/minimum value.

All five values are compared according to the flavor criteria (MAX - Maximum, MAXM - Maximum magnitude and MIN - minimum).
If the winner (e.g. highest value for MAX flavor) is the current winner then current winner (Dm.M) and current index (Dn.L) remain unchanged.
If the winner is one of the four input value, this value is signed extended to 32-bit and copied to Dm, while its index (Dn.H + location of the winner value out of the four input values) is written to Dn.L
The current index (Dn.H) is then incremented by 4, 8, 12 or 16 according to immediate field value in order to enable parallel execution of such one, two, three or four instructions.

For more information on MAX, MAXN and MIN flavors winner criteria see description of MAX.nT, MAXN.nT and MIN.nT instructions.

Flag Options

Flag	Position	Description
MAX	flg1	Maximum criteria
MAXM	flg1	Maximum absolute value criteria
MIN	flg1	Minimum criteria

Instruction Variants

#	Syntax	Description
1	**FND.MAX.4T #u2,Da,Db,Dm:Dn**	**Find maximum value and index**

```
Dm = S40(max(Dm.M, Da.WH, Da.WL, Db.WH, Db.WL)
- Max value of the 4 input 20-bit words and one input long, where max means > , not ≥
local_max_index = index to new max value (0 if Da.WH, 1 if Da.WL, 2 if Db.WH or 3 if
Db.WL).
- Priority is given to the lower index of 0,1,2,3 (If two values are equal - take the
first one)
Dn. L = ('new max found') ? {Dn[31:18], local_max_index} : Dn. L
Dn. H = Dn. H + (#u2+1)<<2
Dn.E = 0
```

Table continues on the next page...

#	Syntax	Description
2	**FND.MAXM.4T #u2,Da,Db,Dm:Dn**	**Find maximum magnitude value and index**

```
Dm = S40(maxm(Dm.M, Da.WH, Da.WL, Db.WH, Db.WL)
- Max absolute value of the 4 input 20-bit words and one input long, where max means
> , not ≥
- If two values are equal but with opposite sign - take the positive one.
local_max_index = index to new max value (0 if Da.WH, 1 if Da.WL, 2 if Db.WH or 3 if
Db.WL).
- Priority is given to the lower index of 0,1,2,3 (If two values are equal - take the
first one)
Dn. L = ('new max found') ? {Dn[31:18], local_max_index} : Dn. L
Dn. H = Dn. H + (#u2+1)<<2
Dn.E = 0
```

#	Syntax	Description
3	**FND.MIN.4T #u2,Da,Db,Dm:Dn**	**Find minimum value and index**

```
Dm = S40(min(Dm.M, Da.WH, Da.WL, Db.WH, Db.WL)
- Min value of the 4 input 20-bit words and one input long, where min means < , not ≤
local_min_index = index to new min value (0 if Da.WH, 1 if Da.WL, 2 if Db.WH or 3 if
Db.WL).
- Priority is given to the lower index of 0,1,2,3 (If two values are equal - take the
first one)
Dn. L = ('new min found') ? {Dn[31:18], local_min_index} : Dn. L
Dn. H = Dn. H + (#u2+1)<<2
Dn.E = 0
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Db	D0-D63	
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
u2	$0 \le u2 < 2^2$	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU_Y	All

FND.nW Find Maximum or Minimum 16-bit Value (DALU)

General Description

This instruction accelerate the search of a maximum or minimum 16-bit value and it's index within a long vector of 16-bit values.
The instruction read four 16-bit values out of the vector (Da.H, Da.L, Db.H and Db.L), current maximum/minimum value (stored in Dm.M in a 32-bit integer format) and two indexes - Dn.H holds index to the first of four read values while Dn.L holds the index of the current maximum/minimum value.

All five values are compared according to the flavor criteria (MAX - Maximum, MAXM - Maximum magnitude and MIN - minimum).
If the winner (e.g. highest value for MAX flavor) is the current winner then current winner (Dm.M) and current index (Dn.L) remain unchanged.
If the winner is one of the four input values, this value is signed extended to 32-bit and copied to Dm, while its index (Dn.H + location of the winner value out of the four input values) is written to Dn.L
The current index (Dn.H) is then incremented by 4, 8, 12 or 16 according to immediate field value in order to enable parallel execution of such one, two, three or four instructions.

For more information on MAX, MAXN and MIN flavors winner criteria see description of MAX.nW, MAXN.nW and MIN.nW instructions.

Flag Options

Flag	Position	Description
MAX	flg1	Maximum criteria
MAXM	flg1	Maximum absolute value criteria
MIN	flg1	Minimum criteria

Instruction Variants

#	Syntax	Description
1	**FND.MAX.4W #u2,Da,Db,Dm:Dn**	**Find maximum value and index**

```
Dm = S40(max(Dm.M, Da.H, Da.L, Db.H, Db.L)
- Max value of the 4 input 16-bit words and one input long, where max means > , not ≥
local_max_index = index to new max value (0 if Da.H, 1 if Da.L, 2 if Db.H or 3 if
Db.L).
- Priority is given to the lower index of 0,1,2,3 (If two values are equal - take the
first one)
Dn. L = ('new max found') ? {Dn[31:18], local_max_index} : Dn. L
Dn. H = Dn. H + (#u2+1)<<2
Dn.E = 0
```

Table continues on the next page...

#	Syntax	Description
2	**FND.MAXM.4W #u2,Da,Db,Dm:Dn**	**Find maximum magnitude value and index**

```
Dm = S40(maxm(Dm.M, Da.H, Da.L, Db.H, Db.L)
- Max absolute value of the 4 input 16-bit words and one input long, where max means
> , not ≥
- If two values are equal but with opposite sign - take the positive one.
local_max_index = index to new max value (0 if Da.H, 1 if Da.L, 2 if Db.H or 3 if
Db.L).
- Priority is given to the lower index of 0,1,2,3 (If two values are equal - take the
first one)
Dn. L = ('new max found') ? {Dn[31:18], local_max_index} : Dn. L
Dn. H = Dn. H + (#u2+1)<<2
Dn.E = 0
```

#	Syntax	Description
3	**FND.MIN.4W #u2,Da,Db,Dm:Dn**	**Find minimum value and index**

```
Dm = S40(min(Dm.M, Da.H, Da.L, Db.H, Db.L)
- Min value of the 4 input 16-bit words and one input long, where min means < , not ≤
local_min_index = index to new min value (0 if Da.WH, 1 if Da.WL, 2 if Db.WH or 3 if
Db.WL).
- Priority is given to the lower index of 0,1,2,3 (If two values are equal - take the
first one)
Dn. L = ('new min found') ? {Dn[31:18], local_min_index} : Dn. L
Dn. H = Dn. H + (#u2+1)<<2
Dn.E = 0
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Db	D0-D63	
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
u2	$0 \le u2 < 2^2$	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU_Y	All

FRECIP Floating Point Reciprocal (DALU)

General Description

Performs a single precision floating point lookup table search for determining the reciprocal of the input. For the final result, linear interpolation should be applied, which achives accuracy of 15 bits in the mantissa. Further accuracy can be achieved using a Newton-Raphson iteration.
The following code is an example for the sequence:

```
FRECIP d0,d1:d2 ; d0 = input
FRECIP.PRE d0,d3
FMADD.FRECIP.SP d3,d1,d2
FMSUB.SP d0,d2,d4 ; begin Newton-Raphson iteration
FMPY.SP d2,d4,d5 ; d5 = output
```

Flag Options

Flag	Position	Description
PRE	flg1	Preliminary

Instruction Variants

#	Syntax	Description
1	**FRECIP Da,Dm:Dn**	**Reciprocal Approximation. The output are coefficients that can be used to linear interpolate the result.**

```
A = recip_LUT_A(Da[22:17]) ; 19b
B = recip_LUT_B(Da[22:17]) ; 18b
if Da[22:17] ≥ 27
  Dm = {101111101 , A[16:0] , 000000}
else
  Dm = {101111110 , A[17:0] , 00000}
tmpExp = 254 - Da[30:23]
Dn = { Da[31:31] , tmpExp , B , 00000}
if (Da[30:23]]== 0x0 ) ; (zero or denorm)
 Dm = 0x00000000 ; +zero
 Dn = {Da[31:31] , 31h7F7FFFFF} ; almost inf , same sign as input
```

#	Syntax	Description
2	**FRECIP.PRE Da,Dn**	**Reciprocal preparation operation**

```
Dn[30:23]   = 254 - Da[30:23]    ; modify 'x' exponent
Dn.E = 0
Rest of Dn bits = Da
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dm:Dn	D_n:D_{n+1}	$0 \leq n \leq 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
GCR.FDENI	1
GCR.FDIVZ	1
GCR.FINVAL	1

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU_Y	All

FSUB.nSP Floating Point Subtract (DALU)

General Description

Performs single precision floating point subtraction.

Flag Options

Flag	Position	Description
X	flg1	Cross between the arguments

Instruction Variants

#	Syntax	Operation	Description

1	FSUB.2SP Da:Db,Dc:Dd,Dm:Dn	`Dm = (U40)(Dc.M - Da.M)`
		`Dn = (U40)(Dd.M - Db.M)`
			SIMD2 subtract

2	FSUB.SP Da,Db,Dn	`Dn = (U40)(Db.M - Da.M)`	Single subtract

3	FSUB.X.2SP Da:Db,Dc:Dd,Dm:Dn	`Dm = (U40)(Dd.M - Da.M)`
		`Dn = (U40)(Dc.M - Db.M)`
			SIMD2 subtract with swapped operands

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
GCR.FDENI	All
GCR.FINEX	All
GCR.FINVAL	All
GCR.FOVER	All
GCR.FUNDR	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_FLT_NUMERIC	All

FSUBADD.2SP Floating Point Sub-Add (DALU)

General Description

Performs single precision floating point subtraction on first lane and addition on the second lane.

Flag Options

Flag	Position	Description
X	flg1	Cross between the arguments

Instruction Variants

#	Syntax	Operation	Description

1	FSUBADD.2SP Da:Db,Dc:Dd,Dm:Dn	`Dm = (U40)(Dc.M - Da.M)`
		`Dn = (U40)(Dd.M + Db.M)`
			SIMD2 subtract-add

2	FSUBADD.X.2SP Da:Db,Dc:Dd,Dm:Dn	`Dm = (U40)(Dd.M - Da.M)`
		`Dn = (U40)(Dc.M + Db.M)`
			SIMD2 subtract-add with swapped operands.

Explicit Operands

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Affected By Instructions

Field	Relevant Variants
GCR.FDENI	All
GCR.FINEX	All
GCR.FINVAL	All
GCR.FOVER	All
GCR.FUNDR	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_FLT_NUMERIC	All

ILLEGAL Generate an Illegal Exception Request (PCU)

General Description

If SR2.DIN and SR2.ILL are clear, causes the core to execute an Illegal Exception. The processing stages are: sampling the current PC, SR, SR2 and EIDR to LR0; modifying SR, SR2 and EIDR according to the settings for an Illegal exception, and jumping to the address stored in CESRA1. This instruction emulates cases of an illegal encoding causing an illegal exception. The exception is processed as , i.e., no register or memory access is updated as a result of the VLES which includes the ILLEGAL instruction, except the registers that are related to exception processing. In particular, the return PC that is saved in LR0 is that of the VLES with the ILLEGAL instruction. Hence, the ISR should not return to that PC, otherwise the ILLEGAL instruction will be executed again. If SR2.DIN or SR2.ILL are set when this instruction is executed, the core jumps to an Irrecoverable exception. For more details on exception processing and Illegal exceptions see the Program control chapter.

Instruction Variants

#	Syntax	Description
1	**ILLEGAL**	**Generate an illegal exception request**

	`PC = CESRA1; LR0 = {return PC, SR, SR2, EIDR}`
	`EIDR = 0x00000400`	

Affect Instructions

Field	Relevant Variants	Comments
CESRA1	All	
EIDR	All	
SR	All	
SR2	All	

Affected By Instructions

Field	Relevant Variants
EIDR	All
LR	All
SR.C	All
SR.S	All
SR.[P0,P1,P2,P3,P4,P5]	All
SR2.IPM	All
SR2.SPSEL	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
No predication		All
Pipeline behavior	pcu_control	All

INSERT.nX Insert a Bit Field in a 40-Bit Destination (DALU)

General Description

Insert a bit field of arbitrary width (up to 40 bits) that is specified in the lower part of the source register (Db) into the specified offset in the 40-bit destination. If the width is 0, or the offset is greater than 39, the destination is not changed. If the field spills over the 40-bit boundary of the destination, the destination is updated with the lower part of the field up to the 40-bit boundary.

Instruction Variants

#	Syntax	Description
1	**INSERT.X Da,Db,Dn**	**The offset and width are specified in Da**

	`Da[13:8] width`
	`Da[5:0] offset`
	`Dn[Da[5:0] + Da[13:8] -1:Da[5:0]] = Db[Da[5:0] - 1:0]`	

2	**INSERT.X #width6,#offset6,Da,Dn**	**The offset and width are specified as explicit immediate operands**
	`Dn[width6 + offset6 -1 : offset6] = Da[width6 - 1 : 0]`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Db	D0-D63	
Dn	D0-D63	
width6	0 < width6 ≤ 40;	width6 + offset6 ≤ 40
offset6	0 ≤ offset6 ≤ 40;	width6 + offset6 ≤ 40

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	2
	dalu_LBM_Y	1

INSERTA Insert a Bit Field in an R Register (IPU)

General Description

Insert a bit field of arbitrary width (up to 32 bits) that is specified in the lower part of the source R register (Ra) into the specified offset in the destination. If the width is 0, the destination is not changed. If the field spills over the 32-bit boundary of the destination, the destination is updated with the lower part of the field up to the 32-bit boundary.

Instruction Variants

#	Syntax	Operation	Description
1	INSERTA #U5,#u5,Ra,Rn	`Rn[U5+u5-1:u5] = Ra[U5-1:0]`	The offset and width are specified as explicit immediate operands

Explicit Operands

Operand	Permitted Values
Ra	R0-R31
Rn	R0-R31
U5	$0 \le U5 < 2^5$
u5	$0 \le u5 < 2^5$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Pipeline behavior	agu_AAU_LOGIC	All

INVSQRT Inverse Square Root (DALU)

General Description

This instruction participates in the calculation of the inverse square root of the input multiplied divided by 2: 1/(2*SQRT(X)), where X is the input for the INVSQRT instruction. It calculates two parameters, A and B, that are used for linear approximation of the result y = -A*X+B using a subsequent instruction MACM.SU.X.

For a correct result, the input value X should be a fraction in the range of 0.25 ≤ X < 1. The accuracy of the output after the two instructions is 14 bits. The accuracy could be improved to 29 bits by applying an iteration of the Newton-Raphson algorithm. The following code is an example that includes a Newton-Raphson iteration:

```
invsqrt.2x d3,d0:d1 ; d3 input
macm.su.x -d3.h,d0,d1 ; d1 estimation of invsqrt(d3)/2
tfr.x #0x60000000,d10 ; d10 = 3/4 (fractional representation)
; Newton-Raphson iteration:
mpy32.sr.x d1,d1,d5 ; d1^2
mpy32.sr.x d1,d10,d6 ; 3/4 * d1
mpy32.sr.x d5,d1,d5 ; 3/4 * d1^3
mpy32.sr.x d5,d3,d7 ; 3/4 * d1^3 * d3
sub.x d7,d6,d8 ; 3/4 * d1 - (3/4 * d1^3 * d3)
add.x d8,d8,d9 ; d9 = invsqrt(d3)/2
```

Instruction Variants

#	Syntax	Description
1	**INVSQRT.2X Da,Dm:Dn**	**Reciprocal (inverse) Square Root Approximation**

```
in = Da[30:30] ? {1, Da[29:25]}  : {0, Da[28:24]}  ; 6b table entry
A = invsqrt_LUT_A( in ) ; 16b
Dm = { 0000_0000 , A , 0000_0000_0000_0000 }
B = invsqrt_LUT_B( in ) ; 16b
Dn = { 0000_0000 , B , 0000_0000_0000_0000 }
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dm:Dn	D_n:D_{n+1}	0 ≤ n ≤ 62

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

JMP Jump to a Destination (PCU,PCU_LSU)

General Description

Transfers program execution to the specified 32-bit absolute address. The destination is either specified as a label, or in an R register. An instruction variant may be marked not to be affected and not affect the BTB (NOBTB flag).

Flag Options

Flag	Position	Description
NOBTB	flg1	Does not use nor updates the BTB

Instruction Variants

#	Syntax	Operation	Description
1	JMP[.NOBTB] LABEL	`PC = {AbsAdd31,0}`	Jump to the specified address. By default is subject to BTB acceleration, unless the NOBTB flag is used.
2	JMP Rn	`PC = Rn`	Jump to the address stored in R. The instruction is not subject to BTB acceleration.

Explicit Operands

Operand	Permitted Values
AbsAdd31	$0 \le \text{AbsAdd31} < 2^{31}$
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	2
	48-bits	1
Execution unit	PCU	1
	PCU+LSU	2
Pipeline behavior	pcu_COF	All

JMPRF Jump if Reservation Failed (PCU)

General Description

If SR.RF is set, jumps to the specified absolute address. If it is clear, continues to execute the next serial VLES. This instruction is used as part of the semaphore reservation procedure, using the LDRSTC instruction. In this procedure, the user attempts to reserve a memory resource with a Load-Reserve and Store-Conditional sequence. The result of that attempt is implicitly captured in SR.RF (Reservation Failed flag). A subsequent test-jump with JMPRF will jump to re-execute the sequence until the semaphore is captured (SR.RF is cleared). For more information on reservation and semaphores, see the description in the Address Generation chapter.

Instruction Variants

#	Syntax	Description
1	**JMPRF LABEL**	**Jump to the specified address if SR.RF is set. The instruction is not subject to BTB acceleration.**
	`if (SR.RF) then PC = {AbsAdd31,0}`	

Explicit Operands

Operand	Permitted Values
AbsAdd31	$0 \le AbsAdd31 < 2^{31}$

Affect Instructions

Field	Relevant Variants	Comments
SR.RF	All	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	48-bits	All
No predication		All
Pipeline behavior	pcu_COF	All

JSR Jump to a Subroutine (PCU_LSU)

General Description

Implicitly pushed SR and the PC of the next VLES (the return PC) to the software stack, pointed to by the active SP, and increments SP by 8. Then a jump is performed to the specified destination. The 32-bit destination is either specified as a label, or in an R register. The return PC is also saved in the RAS, optionally accelerating a subsequent RTS instruction. The JSR instruction is affected by and affecting the BTB by default. An instruction variant may be used which does not interact with the BTB (NOBTB flag).

Flag Options

Flag	Position	Description
NOBTB	flg1	Does not use nor updates the BTB

Instruction Variants

#	Syntax	Description
1	**JSR[.NOBTB] LABEL**	**Jump to a subroutine in the specified absolute address. By default is subject to BTB acceleration, unless the NOBTB flag is used.**

	`PC = {AbsAdd31,0}`
	`Mem(SP,8) = {return PC,SR}, SP = SP + 8`	

2	**JSR Rn**	**Jump to a subroutine in the address stored in R. The instruction is not subject to BTB acceleration.**
	`PC = Rn; Mem(SP,8) = {PC,SR); SP = SP + 8`	

Explicit Operands

Operand	Permitted Values
AbsAdd31	$0 \le AbsAdd31 < 2^{31}$
Rn	R0-R31

Affect Instructions

Field	Relevant Variants	Comments
PC	All	
SP, ESP, TSP, DSP	All	
SR	All	
SR2.SPSEL	All	

Affected By Instructions

Field	Relevant Variants
SP, ESP, TSP, DSP	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	2
	48-bits	1
Pipeline behavior	pcu_COF	All

LD.BTR Load 32 Bits to BTR From Memory (LSU)

General Description

Load a 32-bit value from memory to either BTR0 or BTR1. This instruction only has one addressing mode.

Instruction Variants

#	Syntax	Operation	Description
1	LD.L (Rn),BTRN	`BTRN = (Rn)..(Rn+3)`	Load a BTR register from the address pointed by Rn

Explicit Operands

Operand	Permitted Values	
BTRN	`BTR0,`	`BTR1`
Rn	`R0-R31`	

Affected By Instructions

Field	Relevant Variants
BTRN	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Pipeline behavior	agu_MOVE_LD	All

LD.PAR.W→ D Load a 16-bit Word From Memory to a Partial Portion of a Data Register (LSU)

General Description

Load one 16-bit word from memory into the high (H) or low (L) portion of a data register. The other portion and the extension are unaffected.

Flag Options

Flag	Position	Description
PAR	flg1	Partial (reminder of the register remains unchanged)

Instruction Variants

#	Syntax	Operation	Description

1	LD.PAR.W (SAM),Da.h	`Da = Da`
		`Da.H = (EA)..(EA+1)`
			Load a word to the high portion (H) of a register. All Standard addressing modes supported.

2	LD.PAR.W (SAM),Da.l	`Da = Da`
		`Da.L = (EA)..(EA+1)`
			Load a word to the low portion (L) of a register. All Standard addressing modes supported.

Explicit Operands

Operand	Permitted Values
Da	`D0-D63`
Rk	`R0-R7,R16-R23`
Rn	`R0-R31`

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	All	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	1

Table continues on the next page...

Attribute	Value	Relevant Variant #
	32-bits 48-bits	2
Pipeline behavior	agu_MOVE_LD_POSTINC_PARTIALW	1
	agu_MOVE_LD_POSTINC_PARTIALW agu_MOVE_LD_PRECALC_PARTIALW agu_MOVE_LD_PARTIALW	2

LD.nB→ D Load Bytes From Memory to Data Registers (LSU)

General Description

Load a byte or several bytes from memory into data registers. Each byte that is loaded is placed in the lower part of the destination register. Bytes may be signed or unsigned, which means the byte is sign or zero extended in the destination register.

Flag Options

Flag	Position	Description
U	flg2	Unsigned

Instruction Variants

#	Syntax	Operation	Description

1	LD.2B (SAM),Da:Db	`Db = (S40)(EA+1)`
		`Da = (S40)(EA+0)`
			Load 2 bytes into two registers, sign extending them. All Standard addressing modes supported.

2	LD.4B (SAM),Da:Db:Dc:Dd	`Dd = (S40)(EA+3)`
		`Dc = (S40)(EA+2)`
		`Db = (S40)(EA+1)`
		`Da = (S40)(EA+0)`
			Load 4 bytes into four registers, sign extending them. All Standard addressing modes supported.

3	LD.8B (SAM),Da::Dh	`Dh = (S40)(EA+7)`
		`Dg = (S40)(EA+6)`
		`Df = (S40)(EA+5)`
		`De = (S40)(EA+4)`
		`Dd = (S40)(EA+3)`
		`Dc = (S40)(EA+2)`
		`Db = (S40)(EA+1)`
		`Da = (S40)(EA+0)`
			Load 8 bytes into eight registers, sign extending them. All Standard addressing modes supported.

4	LD.B (SAM),Da	`Da = (S40)(EA)`	Load one byte into a register, sign extending it. All Standard addressing modes supported.
5	LD.B (SP-u9),Da	`Da = (S40)(SP-u9)`	Load one byte into a register, sign extending it, using SP-relative addressing with a short offset.

6	LD.U.2B (SAM),Da:Db	`Db = (U40)(EA+1)`
		`Da = (U40)(EA+0)`
			Load 2 unsigned bytes into two registers, zero extending them. All Standard addressing modes supported.

7	LD.U.4B (SAM),Da:Db:Dc:Dd	`Dd = (U40)(EA+3)`
		`Dc = (U40)(EA+2)`
		`Db = (U40)(EA+1)`
		`Da = (U40)(EA+0)`
			Load 4 unsigned bytes into four registers, zero extending them. All Standard addressing modes supported.

Table continues on the next page...

#	Syntax	Operation	Description

8	LD.U.8B (SAM),Da::Dh	`Dh = (U40)(EA+7)`
		`Dg = (U40)(EA+6)`
		`Df = (U40)(EA+5)`
		`De = (U40)(EA+4)`
		`Dd = (U40)(EA+3)`
		`Dc = (U40)(EA+2)`
		`Db = (U40)(EA+1)`
		`Da = (U40)(EA+0)`
			Load 8 unsigned bytes into eight registers, zero extending them. All Standard addressing modes supported.

9	LD.U.B (SAM),Da	`Da = (U40)(EA)`	Load one unsigned byte into a register, zero extending it. All Standard addressing modes supported.
10	LD.U.B (SP-u9),Da	`Da = (U40)(SP - u9)`	Load one unsigned byte into a register, zero extending it, using SP-relative addressing with a short offset.

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Da::Dh	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$ $0 \le n \le 56$
Da:Db	$D_n:D_{n+1}$ $0 \le n \le 62$
Da:Db:Dc:Dd	$D_n:D_{n+1}:D_{n+2}:D_{n+3}$ $0 \le n \le 60$
Rk	R0-R7,R16-R23
Rn	R0-R31
u9	$0 \le u9 < 2^9$

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2, 3, 4, 6, 7, 8, 9	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3, 6, 7, 8
Encoding length	32-bits	5, 10
	32-bits 48-bits	1, 2, 3, 4, 6, 7, 8, 9
Pipeline behavior	agu_MOVE_LD_POSTINC agu_MOVE_LD_PRECALC	1, 2, 3, 4, 6, 7, 8, 9
	agu_MOVE_LD_PRECALC	5, 10

LD.nB→ R Load Bytes From Memory to Address Registers (LSU)

General Description

Load a byte or several bytes from memory into address registers. Each byte that is loaded is placed in the lower part of the destination register. Bytes may be signed or unsigned, which means the byte is sign or zero extended in the destination register.

Flag Options

Flag	Position	Description
U	flg2	Unsigned

Instruction Variants

#	Syntax	Operation	Description

1	LD.2B (SAM),Ra:Rb	`Rb = (S32)(EA+1)`
		`Ra = (S32)(EA+0)`
			Load 2 bytes into two registers, sign extending them. All Standard addressing modes supported.

2	LD.4B (SAM),Ra:Rb:Rc:Rd	`Rd = (S32)(EA+3)`
		`Rc = (S32)(EA+2)`
		`Rb = (S32)(EA+1)`
		`Ra = (S32)(EA+0)`
			Load 4 bytes into four registers, sign extending them. All Standard addressing modes supported.

3	LD.B (SAM),Ra	`Ra = (S32)(EA)`	Load a byte into an register, sign extending it. All Standard addressing modes supported.
4	LD.B (SP-u9),Ra	`Ra = (S32)(SP-u9)`	Load one byte into an register, sign extending it, using SP-relative addressing with a short offset.

5	LD.U.2B (SAM),Ra:Rb	`Rb = (U32)(EA+1)`
		`Ra = (U32)(EA+0)`
			Load 2 unsigned bytes into two registers, zero extending them. All Standard addressing modes supported.

6	LD.U.4B (SAM),Ra:Rb:Rc:Rd	`Rd = (U32)(EA+3)`
		`Rc = (U32)(EA+2)`
		`Rb = (U32)(EA+1)`
		`Ra = (U32)(EA+0)`
			Load 4 unsigned bytes into four registers, zero extending them. All Standard addressing modes supported.

7	LD.U.B (SAM),Ra	`Ra = (U32)(EA)`	Load one unsigned byte into a register, zero extending it. All Standard addressing modes supported.
8	LD.U.B (SP-u9),Ra	`Ra = (U32)(SP-u9)`	Load one unsigned byte into a register, zero extending it, using SP-relative addressing with a short offset.

Explicit Operands

Operand	Permitted Values
Ra	R0-R31
Ra:Rb	R_n:R_{n+1} 0 ≤ n ≤ 30
Ra:Rb:Rc:Rd	R_n:R_{n+1}:R_{n+2}:R_{n+3} 0 ≤ n ≤ 28 Explicit list of all allowed combinations at Table C-22
Rk	R0-R7,R16-R23
Rn	R0-R31
u9	0 ≤ u9 < 2^9

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2, 3, 5, 6, 7	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 5, 6
Encoding length	32-bits	4, 8
	32-bits 48-bits	1, 2, 3, 5, 6, 7
Pipeline behavior	agu_MOVE_LD_POSTINC agu_MOVE_LD_PRECALC	1, 2, 3, 5, 6, 7
	agu_MOVE_LD_PRECALC	4, 8

LD.nBF→ D Load Fractional Bytes From Memory to Data Registers (LSU)

General Description

Load one or several signed fractional bytes from memory into data registers. Each byte that is loaded is placed in byte HH of the destination register (the highest byte not including the extension). The loaded bytes are sign extended to the left and zero padded to the right in the destination register.

Instruction Variants

#	Syntax	Description
1	**LD.2BF (SAM),Da:Db**	**Load 2 fractional bytes into two registers. All Standard addressing modes supported.**

	`Db = (S40){(EA+1), 0x000000}`
	`Da = (S40){(EA+0), 0x000000}`	

2	**LD.4BF (SAM),Da:Db:Dc:Dd**	**Load 4 fractional bytes into four registers. All Standard addressing modes supported.**

	`Dd = (S40){(EA+3), 0x000000}`
	`Dc = (S40){(EA+2), 0x000000}`
	`Db = (S40){(EA+1), 0x000000}`
	`Da = (S40){(EA+0), 0x000000}`	

3	**LD.8BF (SAM),Da::Dh**	**Load 8 fractional bytes into eight registers. All Standard addressing modes supported.**

	`Dh = (S40){(EA+7), 0x000000}`
	`Dg = (S40){(EA+6), 0x000000}`
	`Df = (S40){(EA+5), 0x000000}`
	`De = (S40){(EA+4), 0x000000}`
	`Dd = (S40){(EA+3), 0x000000}`
	`Dc = (S40){(EA+2), 0x000000}`
	`Db = (S40){(EA+1), 0x000000}`
	`Da = (S40){(EA+0), 0x000000}`	

4	**LD.BF (SAM),Da**	**Load one fractional byte into a register. All Standard addressing modes supported.**
	`Da = (S40){(EA), 0x000000}`	
5	**LD.BF (SP-u9),Da**	**Load one fractional byte into a register, using SP-relative addressing with a short offset.**
	`Da = (S40){(SP-u9), 0x000000}`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da::Dh	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$	$0 \le n \le 56$
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Da:Db:Dc:Dd	$D_n:D_{n+1}:D_{n+2}:D_{n+3}$	$0 \le n \le 60$

Table continues on the next page...

Operand	Permitted Values
Rk	R0-R7,R16-R23
Rn	R0-R31
u9	$0 \le u9 < 2^9$

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2, 3, 4	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3
Encoding length	32-bits	5
	32-bits 48-bits	1, 2, 3, 4
Pipeline behavior	agu_MOVE_LD_POSTINC agu_MOVE_LD_PRECALC	1, 2, 3, 4
	agu_MOVE_LD_PRECALC	5

LD.nF→ D Load Fractional 16-bit Words From Memory to Data Registers (LSU)

General Description

Load one or several signed fractional 16-bit words from memory into data registers. Each word that is loaded is placed in portion H of the destination register (the highest word not including the extension). The loaded words are sign extended to the left and zero padded to the right in the destination register.

Instruction Variants

#	Syntax	Description
1	**LD.2F (SAM),Da:Db**	**Load 2 fractional words into two registers. All Standard addressing modes supported.**

	`Db = (S40){(EA+2)..(EA+3), 0x0000}`
	`Da = (S40){(EA+0)..(EA+1), 0x0000}`	

2	**LD.4F (SAM),Da:Db:Dc:Dd**	**Load 4 fractional words into four registers. All Standard addressing modes supported.**

	`Dd = (S40){(EA+6)..(EA+7), 0x0000}`
	`Dc = (S40){(EA+4)..(EA+5), 0x0000}`
	`Db = (S40){(EA+2)..(EA+3), 0x0000}`
	`Da = (S40){(EA+0)..(EA+1), 0x0000}`	

3	**LD.8F (SAM),Da::Dh**	**Load 8 fractional words into eight registers. All Standard addressing modes supported.**

	`Dh = (S40){(EA+14)..(EA+15), 0x0000}`
	`Dg = (S40){(EA+12)..(EA+13), 0x0000}`
	`Df = (S40){(EA+10)..(EA+11), 0x0000}`
	`De = (S40){(EA+8)..(EA+9), 0x0000}`
	`Dd = (S40){(EA+6)..(EA+7), 0x0000}`
	`Dc = (S40){(EA+4)..(EA+5), 0x0000}`
	`Db = (S40){(EA+2)..(EA+3), 0x0000}`
	`Da = (S40){(EA+0)..(EA+1), 0x0000}`	

4	**LD.F (SAM),Da**	**Load one fractional word into a register. All Standard addressing modes supported.**
	`Da = (S40){(EA)..(EA+1), 0x0000}`	
5	**LD.F (SP-u9_1),Da**	**Load one fractional word into a register, using SP-relative addressing with a short, word-aligned offset.**
	`Da = (S40){Mem ((SP - u9_1), 2), 0x0000}`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da::Dh	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$	$0 \leq n \leq 56$

Table continues on the next page...

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Da:Db:Dc:Dd	$D_n:D_{n+1}:D_{n+2}:D_{n+3}$	$0 \le n \le 60$
Rk	R0-R7,R16-R23	
Rn	R0-R31	
u9_1	$0 \le$ u9_1 $< 2^{10}$;	u9_1 & 0x1 == 0

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2, 3, 4	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3
Encoding length	32-bits	5
	32-bits 48-bits	1, 2, 3, 4
Pipeline behavior	agu_MOVE_LD_POSTINC agu_MOVE_LD_PRECALC	1, 2, 3, 4
	agu_MOVE_LD_PRECALC	5

LD.nL→ D Load 32-bit Long Words From Memory to Data Registers (LSU)

General Description

Load one or several 32-bit long words from memory into data registers. The data is right-aligned in the destination registers and sign or zero extended to the right. SIMD variants load 2, 4, 8 or 16 longs to a respective number of registers. Specific variants allow some permutations of the destination registers, for example loading 4 longs to registers D0:D3:D1:D2 instead of the normal consecutive register range.

Flag Options

Flag	Position	Description
U	flg2	Unsigned

Instruction Variants

#	Syntax	Description
1	**LD.16L (SAM),Da::Dp**	**Load 16 longs to 16 consecutive registers, sign extending them. All Standard addressing modes supported. Special grouping limitations apply.**

	`Dp = (S40)(EA+60)..(EA+63)`
	`Do = (S40)(EA+56)..(EA+59)`
	`Dn = (S40)(EA+52)..(EA+55)`
	`Dm = (S40)(EA+48)..(EA+51)`
	`Dl = (S40)(EA+44)..(EA+47)`
	`Dk = (S40)(EA+40)..(EA+43)`
	`Dj = (S40)(EA+36)..(EA+39)`
	`Di = (S40)(EA+32)..(EA+35)`
	`Dh = (S40)(EA+28)..(EA+31)`
	`Dg = (S40)(EA+24)..(EA+27)`
	`Df = (S40)(EA+20)..(EA+23)`
	`De = (S40)(EA+16)..(EA+19)`
	`Dd = (S40)(EA+12)..(EA+15)`
	`Dc = (S40)(EA+8)..(EA+11)`
	`Db = (S40)(EA+4)..(EA+7)`
	`Da = (S40)(EA+0)..(EA+3)`	

2	**LD.2L (SAM),Da:Db**	**Load 2 longs to a consecutive register pair, sign extending them. All Standard addressing modes supported.**

	`Db = (S40)(EA+4)..(EA+7)`
	`Da = (S40)(EA+0)..(EA+3)`	

3	**LD.2L (SAM),Da:Dc**	**Load 2 longs to an alternating register pair (ex. D0:D2), sign extending them. All Standard addressing modes supported.**

	`Dc = (S40)(EA+4)..(EA+7)`
	`Da = (S40)(EA+0)..(EA+3)`	

Table continues on the next page...

#	Syntax	Description
4	**LD.4L (SAM),Da:Db:Dc:Dd**	**Load 4 longs to a consecutive register quad, sign extending them. All Standard addressing modes supported.**

	`Dd = (S40)(EA+12)..(EA+15)`
	`Dc = (S40)(EA+8)..(EA+11)`
	`Db = (S40)(EA+4)..(EA+7)`
	`Da = (S40)(EA+0)..(EA+3)`	

5	**LD.4L (SAM),Da:Dc:Db:Dd**	**Load 4 longs to a shuffled register quad (ex. D0:D2:D1:D3), sign extending them. All Standard addressing modes supported.**

	`Dd = (S40)(EA+12)..(EA+15)`
	`Db = (S40)(EA+8)..(EA+11)`
	`Dc = (S40)(EA+4)..(EA+7)`
	`Da = (S40)(EA+0)..(EA+3)`	

6	**LD.4L (SAM),Da:Dc:De:Dg**	**Load 4 longs to an alternating register quad (ex. D0:D2:D4:D6), sign extending them. All Standard addressing modes supported.**

	`Dg = (S40)(EA+12)..(EA+15)`
	`De = (S40)(EA+8)..(EA+11)`
	`Dc = (S40)(EA+4)..(EA+7)`
	`Da = (S40)(EA+0)..(EA+3)`	

7	**LD.8L (SAM),Da:Dc:De:Dg:Db:Dd:Df:Dh**	**Load 8 longs to a shuffled register octet (ex. D0:D2:D4:D6:D1:D3:D5), sign extending them. All Standard addressing modes supported.**

	`Dh = (S40)(EA+28)..(EA+31)`
	`Df = (S40)(EA+24)..(EA+27)`
	`Dd = (S40)(EA+20)..(EA+23)`
	`Db = (S40)(EA+16)..(EA+19)`
	`Dg = (S40)(EA+12)..(EA+15)`
	`De = (S40)(EA+8)..(EA+11)`
	`Dc = (S40)(EA+4)..(EA+7)`
	`Da = (S40)(EA+0)..(EA+3)`	

8	**LD.8L (SAM),Da:Dc:De:Dg:Dh:Dj:Dl:Dn**	**Load 8 longs to a shuffled register octet (ex. D0:D2:D4:D6:D9:D11:D13:D15), sign extending them. All Standard addressing modes supported.**

	`Dn = (S40)(EA+28)..(EA+31)`
	`Dl = (S40)(EA+24)..(EA+27)`
	`Dj = (S40)(EA+20)..(EA+23)`
	`Dh = (S40)(EA+16)..(EA+19)`
	`Dg = (S40)(EA+12)..(EA+15)`
	`De = (S40)(EA+8)..(EA+11)`
	`Dc = (S40)(EA+4)..(EA+7)`
	`Da = (S40)(EA+0)..(EA+3)`	

9	**LD.8L (SAM),Da::Dh**	**Load 8 longs to a consecutive register octet, sign extending them. All Standard addressing modes supported.**

Table continues on the next page...

#	Syntax	Description

	`Dh = (S40)(EA+28)..(EA+31)`
	`Dg = (S40)(EA+24)..(EA+27)`
	`Df = (S40)(EA+20)..(EA+23)`
	`De = (S40)(EA+16)..(EA+19)`
	`Dd = (S40)(EA+12)..(EA+15)`
	`Dc = (S40)(EA+8)..(EA+11)`
	`Db = (S40)(EA+4)..(EA+7)`
	`Da = (S40)(EA+0)..(EA+3)`	

10	**LD.L (SAM),Da**	**Load one long to a register, sign extending it. All Standard addressing modes supported.**
	`Da = (S40)(EA)..(EA+3)`	
11	**LD.L (SP-u9_2),Da**	**Load one long to a register, sign extending it, using SP-relative addressing with a short, long aligned offset**
	`Da = (S40)Mem ((SP - u9_2), 4)`	
12	**LD.U.2L (SAM),Da:Db**	**Load 2 unsigned longs to a consecutive register pair, zero extending them. All Standard addressing modes supported.**

	`Db = (U40)(EA+4)..(EA+7)`
	`Da = (U40)(EA+0)..(EA+3)`	

13	**LD.U.4L (SAM),Da:Db:Dc:Dd**	**Load 4 unsigned longs to a consecutive register quad, zero extending them. All Standard addressing modes supported.**

	`Dd = (U40)(EA+12)..(EA+15)`
	`Dc = (U40)(EA+8)..(EA+11)`
	`Db = (U40)(EA+4)..(EA+7)`
	`Da = (U40)(EA+0)..(EA+3)`	

14	**LD.U.8L (SAM),Da::Dh**	**Load 8 unsigned longs to a consecutive register octet, zero extending them. All Standard addressing modes supported.**

	`Dh = (U40)(EA+28)..(EA+31)`
	`Dg = (U40)(EA+24)..(EA+27)`
	`Df = (U40)(EA+20)..(EA+23)`
	`De = (U40)(EA+16)..(EA+19)`
	`Dd = (U40)(EA+12)..(EA+15)`
	`Dc = (U40)(EA+8)..(EA+11)`
	`Db = (U40)(EA+4)..(EA+7)`
	`Da = (U40)(EA+0)..(EA+3)`	

15	**LD.U.L (SAM),Da**	**Load one unsigned long to a register, zero extending it. All Standard addressing modes supported.**
	`Da = (U40)(EA)..(EA+3)`	
16	**LD.U.L (SP-u9_2),Da**	**Load one unsigned long to a register, zero extending it, using SP-relative addressing with a short, long-aligned offset**
	`Da = (U40)Mem ((SP - u9_2), 4)`	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Da::Dh	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$ $0 \le n \le 56$
Da::Dp	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}:D_{n+8}:D_{n+9}:D_{n+10}:D_{n+11}:D_{n+12}:D_{n+13}:D_{n+14}:D_{n+15}$ $0 \le n \le 48$
Da:Db	$D_n:D_{n+1}$ $0 \le n \le 62$
Da:Db:Dc:Dd	$D_n:D_{n+1}:D_{n+2}:D_{n+3}$ $0 \le n \le 60$
Da:Dc	$D_n:D_{n+2}$ $0 \le n \le 61$ Explicit list of all allowed combinations at Table C-12
Da:Dc:Db:Dd	$D_n:D_{n+2}:D_{n+1}:D_{n+3}$ $0 \le n \le 60$ Explicit list of all allowed combinations at Table C-13
Da:Dc:De:Dg	$D_n:D_{n+2}:D_{n+4}:D_{n+6}$ $0 \le n \le 57$ Explicit list of all allowed combinations at Table C-15
Da:Dc:De:Dg:Db:Dd:Df:Dh	$D_n:D_{n+2}:D_{n+4}:D_{n+6}:D_{n+1}:D_{n+3}:D_{n+5}:D_{n+7}$ $0 \le n \le 56$ Explicit list of all allowed combinations at Table C-16
Da:Dc:De:Dg:Dh:Dj:Dl:Dn	$D_n:D_{n+2}:D_{n+4}:D_{n+6}:D_{n+9}:D_{n+11}:D_{n+13}:D_{n+15}$ $0 \le n \le 48$ Explicit list of all allowed combinations at Table C-17
Rk	R0-R7,R16-R23
Rn	R0-R31
u9_2	$0 \le u9\_2 < 2^{11}$; u9_2 & 0x3 == 0

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 3, 6, 9, 13, 14
Encoding length	32-bits	2, 4, 5, 7, 8, 11, 16
	32-bits 48-bits	1, 3, 6, 9, 10, 12, 13, 14, 15
Pipeline behavior	agu_MOVE_LD_POSTINC	2, 4, 5, 7, 8
	agu_MOVE_LD_POSTINC agu_MOVE_LD_PRECALC	1, 3, 6, 9, 10, 12, 13, 14, 15
	agu_MOVE_LD_PRECALC	11, 16

LD.nL→ R Load 32-bit Long Words From Memory to Address Registers (LSU)

General Description

Load a single or several 32-bit long words from memory into address registers.

Instruction Variants

#	Syntax	Operation	Description

1	LD.2L (SAM),Ra:Rb	`Rb = (S32)(EA+4)..(EA+7)`
		`Ra = (S32)(EA+0)..(EA+3)`
			Load 2 longs to a consecutive register pair. All Standard addressing modes supported.

2	LD.4L (SAM),Ra:Rb:Rc:Rd	`Rd = (EA+12)..(EA+15)`
		`Rc = (EA+8)..(EA+11)`
		`Rb = (EA+4)..(EA+7)`
		`Ra = (EA+0)..(EA+3)`
			Load 4 longs to a consecutive register quad. All Standard addressing modes supported.

3	LD.L (SAM),Ra	`Ra = (EA)..(EA+3)`	Load one long to a register. All Standard addressing modes supported.
4	LD.L (SP-u9_2),Ra	`Ra = Mem ((SP - u9_2), 4)`	Load one long to a register, using SP-relative addressing with a short, long-aligned offset

Explicit Operands

Operand	Permitted Values
Ra	R0-R31
Ra:Rb	$R_n:R_{n+1}$ $0 \le n \le 30$
Ra:Rb:Rc:Rd	$R_n:R_{n+1}:R_{n+2}:R_{n+3}$ $0 \le n \le 28$ Explicit list of all allowed combinations at Table C-22
Rk	R0-R7,R16-R23
Rn	R0-R31
u9_2	$0 \le u9\_2 < 2^{11}$; u9_2 & 0x3 == 0

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2, 3	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	2
Encoding length	32-bits	4
	32-bits 48-bits	1, 2, 3
Pipeline behavior	agu_MOVE_LD_POSTINC agu_MOVE_LD_PRECALC	1, 2, 3
	agu_MOVE_LD_PRECALC	4

LD.nW→ D Load 16-bit Words From Memory to Data Registers (LSU)

General Description

Load one or several 16-bit words from memory into data registers. The data is right-aligned in the destination registers and sign or zero extended to the right. SIMD variants load 2, 4 or 8 words to a respective number of registers.

Flag Options

Flag	Position	Description
U	flg2	Unsigned

Instruction Variants

#	Syntax	Description
1	**LD.2W (SAM),Da:Db**	**Load 2 words to a consecutive register pair, sign extending them. All Standard addressing modes supported.**

	`Db = (S40)(EA+2)..(EA+3)`
	`Da = (S40)(EA+0)..(EA+1)`	

2	**LD.4W (SAM),Da:Db:Dc:Dd**	**Load 4 words to a consecutive register quad, sign extending them. All Standard addressing modes supported.**

	`Dd = (S40)(EA+6)..(EA+7)`
	`Dc = (S40)(EA+4)..(EA+5)`
	`Db = (S40)(EA+2)..(EA+3)`
	`Da = (S40)(EA+0)..(EA+1)`	

3	**LD.8W (SAM),Da::Dh**	**Load 8 words to a consecutive register octet, sign extending them. All Standard addressing modes supported.**

	`Dh = (S40)(EA+14)..(EA+15)`
	`Dg = (S40)(EA+12)..(EA+13)`
	`Df = (S40)(EA+10)..(EA+11)`
	`De = (S40)(EA+8)..(EA+9)`
	`Dd = (S40)(EA+6)..(EA+7)`
	`Dc = (S40)(EA+4)..(EA+5)`
	`Db = (S40)(EA+2)..(EA+3)`
	`Da = (S40)(EA+0)..(EA+1)`	

4	**LD.W (SAM),Da**	**Load one word to a register, sign extending it. All Standard addressing modes supported.**
	`Da = (S40)(EA)..(EA+1)`	
5	**LD.W (SP-u9_1),Da**	**Load one word to a register sign extending it, using SP-relative addressing with a short, word-aligned offset**
	`Da = (S40)Mem ((SP - u9_1), 2)`	

Table continues on the next page...

#	Syntax	Description
6	**LD.U.2W (SAM),Da:Db**	**Load 2 unsigned words to a consecutive register pair, zero extending them. All Standard addressing modes supported.**

	`Db = (U40)(EA+2)..(EA+3)`
	`Da = (U40)(EA+0)..(EA+1)`	

7	**LD.U.4W (SAM),Da:Db:Dc:Dd**	**Load 4 unsigned words to a consecutive register quad, zero extending them. All Standard addressing modes supported.**

	`Dd = (U40)(EA+6)..(EA+7)`
	`Dc = (U40)(EA+4)..(EA+5)`
	`Db = (U40)(EA+2)..(EA+3)`
	`Da = (U40)(EA+0)..(EA+1)`	

8	**LD.U.8W (SAM),Da::Dh**	**Load 8 unsigned words to a consecutive register octet, zero extending them. All Standard addressing modes supported.**

	`Dh = (U40)(EA+14)..(EA+15)`
	`Dg = (U40)(EA+12)..(EA+13)`
	`Df = (U40)(EA+10)..(EA+11)`
	`De = (U40)(EA+8)..(EA+9)`
	`Dd = (U40)(EA+6)..(EA+7)`
	`Dc = (U40)(EA+4)..(EA+5)`
	`Db = (U40)(EA+2)..(EA+3)`
	`Da = (U40)(EA+0)..(EA+1)`	

9	**LD.U.W (SAM),Da**	**Load one unsigned word to a register, zero extending it. All Standard addressing modes supported.**
	`Da = (U40)(EA)..(EA+1)`	
10	**LD.U.W (SP-u9_1),Da**	**Load one unsigned word to a register, zero extending it, using SP-relative addressing with a short, word-aligned offset**
	`Da = (U40)Mem ((SP - u9_1), 2)`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da::Dh	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$	$0 \le n \le 56$
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Da:Db:Dc:Dd	$D_n:D_{n+1}:D_{n+2}:D_{n+3}$	$0 \le n \le 60$
Rk	R0-R7,R16-R23	
Rn	R0-R31	
u9_1	$0 \le u9\_1 < 2^{10}$;	u9_1 & 0x1 == 0

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2, 3, 4, 6, 7, 8, 9	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3, 6, 7, 8
Encoding length	32-bits	5, 10
	32-bits 48-bits	1, 2, 3, 4, 6, 7, 8, 9
Pipeline behavior	agu_MOVE_LD_POSTINC agu_MOVE_LD_PRECALC	1, 2, 3, 4, 6, 7, 8, 9
	agu_MOVE_LD_PRECALC	5, 10

LD.nW→ R Load 16-bit Words From Memory to Address Registers (LSU)

General Description

Load a single or several 16-bit words from memory into address registers. The loaded data is aligned to the right in the destination, and is sign or zero extended.

Flag Options

Flag	Position	Description
U	flg2	Unsigned

Instruction Variants

#	Syntax	Description
1	**LD.2W (SAM),Ra:Rb**	**Load 2 words to a consecutive register pair. All Standard addressing modes supported.**

	`Rb = (S32)(EA+2)..(EA+3)`
	`Ra = (S32)(EA+0)..(EA+1)`	

2	**LD.4W (SAM),Ra:Rb:Rc:Rd**	**Load 4 words to a consecutive register quad. All Standard addressing modes supported.**

	`Rd = (S32)(EA+6)..(EA+7)`
	`Rc = (S32)(EA+4)..(EA+5)`
	`Rb = (S32)(EA+2)..(EA+3)`
	`Ra = (S32)(EA+0)..(EA+1)`	

3	**LD.W (SAM),Ra**	**Load one word to a register. All Standard addressing modes supported.**
	`Ra = (S32)(EA)..(EA+1)`	
4	**LD.W (SP-u9_1),Ra**	**Load one word to a register, using SP-relative addressing with a short, word-aligned offset**
	`Ra = (S32)Mem ((SP - u9_1), 2)`	
5	**LD.U.2W (SAM),Ra:Rb**	**Load 2 unsigned words to a consecutive register pair, zero extending them. All Standard addressing modes supported.**

	`Rb = (U32)(EA+2)..(EA+3)`
	`Ra = (U32)(EA+0)..(EA+1)`	

Table continues on the next page...

#	Syntax	Description
6	**LD.U.4W (SAM),Ra:Rb:Rc:Rd**	**Load 4 unsigned words to a consecutive register quad zero extending them. All Standard addressing modes supported.**

	`Rd = (U32)(EA+6)..(EA+7)`
	`Rc = (U32)(EA+4)..(EA+5)`
	`Rb = (U32)(EA+2)..(EA+3)`
	`Ra = (U32)(EA+0)..(EA+1)`	

7	**LD.U.W (SAM),Ra**	**Load one unsigned word to a register, zero extending it. All Standard addressing modes supported.**
	`Ra = (U32)(EA)..(EA+1)`	
8	**LD.U.W (SP-u9_1),Ra**	**Load one unsigned word to a register, zero extending it, using SP-relative addressing with a short, word-aligned offset**
	`Ra = (U32)Mem ((SP - u9_1), 2)`	

Explicit Operands

Operand	Permitted Values
Ra	R0-R31
Ra:Rb	R_n:R_{n+1} 0 ≤ n ≤ 30
Ra:Rb:Rc:Rd	R_n:R_{n+1}:R_{n+2}:R_{n+3} 0 ≤ n ≤ 28 Explicit list of all allowed combinations at Table C-22
Rk	R0-R7,R16-R23
Rn	R0-R31
u9_1	0 ≤ u9_1 < 2^{10}; u9_1 & 0x1 == 0

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2, 3, 5, 6, 7	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 5, 6
Encoding length	32-bits	4, 8
	32-bits 48-bits	1, 2, 3, 5, 6, 7
Pipeline behavior	agu_MOVE_LD_POSTINC agu_MOVE_LD_PRECALC	1, 2, 3, 5, 6, 7

Table continues on the next page...

Attribute	Value	Relevant Variant #
	agu_MOVE_LD_PRECALC	4, 8

LD.nX→ D — Load 40-bit Values From Memory to Data Registers — (LSU)

General Description

Load one or several 40-bit values from memory into data registers. Every 40-bit value is loaded from memory as a 64-bit access where bits above bit 39 are ignored. SIMD variants load 2, 4, 8 or 16 values to a respective number of registers.

Instruction Variants

#	Syntax	Description
1	**LD.2X (SAM),Da:Db**	**Load 2 40-bit values to a consecutive register pair. All Standard addressing modes supported.**

	`Db = (EA+11)..(EA+15)`
	`Da = (EA+3)..(EA+7)`	

2	**LD.4X (SAM),Da:Db:Dc:Dd**	**Load 4 40-bit values to a consecutive register quad. All Standard addressing modes supported.**

	`Dd = (EA+27)..(EA+31)`
	`Dc = (EA+19)..(EA+23)`
	`Db = (EA+11)..(EA+15)`
	`Da = (EA+3)..(EA+7)`	

3	**LD.8X (SAM),Da::Dh**	**Load 8 40-bit values to a consecutive register octet. All Standard addressing modes supported. Special grouping limitations apply.**

	`Dh = (EA+59)..(EA+63)`
	`Dg = (EA+51)..(EA+55)`
	`Df = (EA+43)..(EA+47)`
	`De = (EA+35)..(EA+39)`
	`Dd = (EA+27)..(EA+31)`
	`Dc = (EA+19)..(EA+23)`
	`Db = (EA+11)..(EA+15)`
	`Da = (EA+3)..(EA+7)`	

4	**LD.X (SAM),Da**	**Load one 40-bit value to a register. All Standard addressing modes supported.**
	`Da = (EA+3)..(EA+7)`	
5	**LD.X (SP-u9_3),Da**	**Load one 40-bit value to a register, using SP-relative addressing with a short, 64-bit aligned offset**
	`Da = ((SP - u9_3)+3)..((SP - u9_3)+7)`	

Explicit Operands

Operand	Permitted Values	
Da	`D0-D63`	
Da::Dh	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$	$0 \le n \le 56$

Table continues on the next page...

Operand	Permitted Values	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Da:Db:Dc:Dd	D_n:D_{n+1}:D_{n+2}:D_{n+3}	$0 \le n \le 60$
Rk	R0-R7,R16-R23	
Rn	R0-R31	
u9_3	$0 \le$ u9_3 $< 2^{12}$;	u9_3 & 0x7 == 0

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2, 3, 4	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3
Encoding length	32-bits	5
	32-bits 48-bits	1, 2, 3, 4
Pipeline behavior	agu_MOVE_LD_POSTINC agu_MOVE_LD_PRECALC	1, 2, 3, 4
	agu_MOVE_LD_PRECALC	5

LD2.nB→ D Load Bytes From Memory and Pack in Data Registers (LSU)

General Description

Load an even number of bytes from memory and pack them as pairs into data registers. Each register is loaded with 2 bytes, one to byte LL and the other to byte HL in the register. Each byte is then either sign or zero extended to 20 bits in the wide-high (WH) or wide-low (WL) portions of the register. SIMD variants load 4, 8 or 16 bytes into 2, 4 or 8 data registers, respectively.

Flag Options

Flag	Position	Description
U	flg2	Unsigned

Instruction Variants

#	Syntax	Operation	Description

1	LD2.16B (SAM),Da::Dh	`Dh.WL = (S20)(EA+15)`
		`Dh.WH = (S20)(EA+14)`
		`Dg.WL = (S20)(EA+13)`
		`Dg.WH = (S20)(EA+12)`
		`Df.WL = (S20)(EA+11)`
		`Df.WH = (S20)(EA+10)`
		`De.WL = (S20)(EA+9)`
		`De.WH = (S20)(EA+8)`
		`Dd.WL = (S20)(EA+7)`
		`Dd.WH = (S20)(EA+6)`
		`Dc.WL = (S20)(EA+5)`
		`Dc.WH = (S20)(EA+4)`
		`Db.WL = (S20)(EA+3)`
		`Db.WH = (S20)(EA+2)`
		`Da.WL = (S20)(EA+1)`
		`Da.WH = (S20)(EA+0)`
			Load 16 bytes into eight registers, sign extending them. All Standard addressing modes supported.

2	LD2.2B (SAM),Da	`Da.WL = (S20)(EA+1)`
		`Da.WH = (S20)(EA+0)`
			Load 2 bytes into a register, sign extending them. All Standard addressing modes supported.

3	LD2.4B (SAM),Da:Db	`Db.WL = (S20)(EA+3)`
		`Db.WH = (S20)(EA+2)`
		`Da.WL = (S20)(EA+1)`
		`Da.WH = (S20)(EA+0)`
			Load 4 bytes into two registers, sign extending them. All Standard addressing modes supported.

4	LD2.8B (SAM),Da:Db:Dc:Dd	`Dd.WL = (S20)(EA+7)`
		`Dd.WH = (S20)(EA+6)`
		`Dc.WL = (S20)(EA+5)`
		`Dc.WH = (S20)(EA+4)`
		`Db.WL = (S20)(EA+3)`
		`Db.WH = (S20)(EA+2)`
		`Da.WL = (S20)(EA+1)`
		`Da.WH = (S20)(EA+0)`
			Load 8 bytes into four registers, sign extending them. All Standard addressing modes supported.

Table continues on the next page...

#	Syntax	Operation	Description

5	LD2.U.16B (SAM),Da::Dh	Dh.WL = (U20)(EA+15)
		Dh.WH = (U20)(EA+14)
		Dg.WL = (U20)(EA+13)
		Dg.WH = (U20)(EA+12)
		Df.WL = (U20)(EA+11)
		Df.WH = (U20)(EA+10)
		De.WL = (U20)(EA+9)
		De.WH = (U20)(EA+8)
		Dd.WL = (U20)(EA+7)
		Dd.WH = (U20)(EA+6)
		Dc.WL = (U20)(EA+5)
		Dc.WH = (U20)(EA+4)
		Db.WL = (U20)(EA+3)
		Db.WH = (U20)(EA+2)
		Da.WL = (U20)(EA+1)
		Da.WH = (U20)(EA+0)
			Load 16 unsigned bytes into eight registers, zero extending them. All Standard addressing modes supported.

6	LD2.U.2B (SAM),Da	Da.WL = (U20)(EA+1)
		Da.WH = (U20)(EA+0)
			Load 16 unsigned bytes into eight registers, zero extending them. All Standard addressing modes supported.

7	LD2.U.4B (SAM),Da:Db	Db.WL = (U20)(EA+3)
		Db.WH = (U20)(EA+2)
		Da.WL = (U20)(EA+1)
		Da.WH = (U20)(EA+0)
			Load 4 unsigned bytes into two registers, zero extending them. All Standard addressing modes supported.

8	LD2.U.8B (SAM),Da:Db:Dc:Dd	Dd.WL = (U20)(EA+7)
		Dd.WH = (U20)(EA+6)
		Dc.WL = (U20)(EA+5)
		Dc.WH = (U20)(EA+4)
		Db.WL = (U20)(EA+3)
		Db.WH = (U20)(EA+2)
		Da.WL = (U20)(EA+1)
		Da.WH = (U20)(EA+0)
			Load 8 unsigned bytes into four registers, zero extending them. All Standard addressing modes supported.

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Da::Dh	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$ $0 \le n \le 56$
Da:Db	$D_n:D_{n+1}$ $0 \le n \le 62$
Da:Db:Dc:Dd	$D_n:D_{n+1}:D_{n+2}:D_{n+3}$ $0 \le n \le 60$
Rk	R0-R7,R16-R23
Rn	R0-R31

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	All	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits 48-bits	All
Pipeline behavior	agu_MOVE_LD_POSTINC agu_MOVE_LD_PRECALC	All

LD2.nBF→ D Load Fractional Bytes From Memory and Pack in Data Registers (LSU)

General Description

Load an even number of fractional bytes from memory and pack them as pairs into data registers. Each register is loaded with 2 bytes, one to byte HH and the other to byte LH in the register. Each byte is then sign extended by 4 bits to the left, and zero padded 8 bits to the right, filling 20 bits in the wide-high (WH) or wide-low (WL) portions of the register. SIMD variants load 4, 8 or 16 bytes into 2, 4 or 8 data registers, respectively.

Instruction Variants

#	Syntax	Description
1	**LD2.16BF (SAM),Da::Dh**	**Load 16 fractional bytes into eight registers. All Standard addressing modes supported.**

	`Dh.WL = (S20){(EA+15), 0x00}`
	`Dh.WH = (S20){(EA+14), 0x00}`
	`Dg.WL = (S20){(EA+13), 0x00}`
	`Dg.WH = (S20){(EA+12), 0x00}`
	`Df.WL = (S20){(EA+11), 0x00}`
	`Df.WH = (S20){(EA+10), 0x00}`
	`De.WL = (S20){(EA+9), 0x00}`
	`De.WH = (S20){(EA+8), 0x00}`
	`Dd.WL = (S20){(EA+7), 0x00}`
	`Dd.WH = (S20){(EA+6), 0x00}`
	`Dc.WL = (S20){(EA+5), 0x00}`
	`Dc.WH = (S20){(EA+4), 0x00}`
	`Db.WL = (S20){(EA+3), 0x00}`
	`Db.WH = (S20){(EA+2), 0x00}`
	`Da.WL = (S20){(EA+1), 0x00}`
	`Da.WH = (S20){(EA+0), 0x00}`	

2	**LD2.2BF (SAM),Da**	**Load 2 fractional bytes into one register. All Standard addressing modes supported.**

	`Da.WL = (S20){(EA+1), 0x00}`
	`Da.WH = (S20){(EA+0), 0x00}`	

3	**LD2.4BF (SAM),Da:Db**	**Load 4 fractional bytes into two registers. All Standard addressing modes supported.**

	`Db.WL = (S20){(EA+3), 0x00}`
	`Db.WH = (S20){(EA+2), 0x00}`
	`Da.WL = (S20){(EA+1), 0x00}`
	`Da.WH = (S20){(EA+0), 0x00}`	

4	**LD2.8BF (SAM),Da:Db:Dc:Dd**	**Load 8 fractional bytes into four registers. All Standard addressing modes supported.**

	`Dd.WL = (S20){(EA+7), 0x00}`
	`Dd.WH = (S20){(EA+6), 0x00}`
	`Dc.WL = (S20){(EA+5), 0x00}`
	`Dc.WH = (S20){(EA+4), 0x00}`
	`Db.WL = (S20){(EA+3), 0x00}`
	`Db.WH = (S20){(EA+2), 0x00}`
	`Da.WL = (S20){(EA+1), 0x00}`
	`Da.WH = (S20){(EA+0), 0x00}`	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Da::Dh	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$ $0 \le n \le 56$
Da:Db	$D_n:D_{n+1}$ $0 \le n \le 62$
Da:Db:Dc:Dd	$D_n:D_{n+1}:D_{n+2}:D_{n+3}$ $0 \le n \le 60$
Rk	R0-R7,R16-R23
Rn	R0-R31

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	All	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits 48-bits	All
Pipeline behavior	agu_MOVE_LD_POSTINC agu_MOVE_LD_PRECALC	All

LD2.nF→ D Load 16-bit Fractional Words From Memory and Pack in Data Registers (LSU)

General Description

Load an even number of 16-bit fractional words from memory and pack them as pairs into data registers. Each register is loaded with 2 words one to portion H and the other to portion L in the register. Each word is then sign extended by 4 bits to the left, filling 20 bits in the wide-high (WH) or wide-low (WL) portions of the register. SIMD variants load 4, 8, 16 or 32 words into 2, 4, 8 or 16 data registers, respectively. Some SIMD variants allow special permutations of the destination registers that allow to save data reordering in some algorithms.

The 16F variant using Da..Dh-mod8p supoprts many 8-register combinations as follows: The index of each D register is calculated as a concatenation of two 3-bit values: {val1[2:0],val2[2:0]}. Val1 is calculated as (n[5:3] + pos), where n is the index of Da (first register in the octet), and 'pos' is the serial position of D in the octet starting with zero; val2 is bits [2:0] of (n + pos). For example, in octet D2:D11:D20:D29:D38:D47:D48:D57, n=2; the index of D48 (pos = 6) is calculated by: val1 = (2[5:3] + 6) = 0b110, val2 = (2+6)[2:0] = 0b000. {val1:val2} = 0b110000 = 48.

Instruction Variants

#	Syntax	Description
1	**LD2.16F (SAM),Da::Dh**	**Load 16 fractional words into eight consecutive registers. All Standard addressing modes supported.**

```
Dh.WL = (S20)(EA+30)..(EA+31)
Dh.WH = (S20)(EA+28)..(EA+29)
Dg.WL = (S20)(EA+26)..(EA+27)
Dg.WH = (S20)(EA+24)..(EA+25)
Df.WL = (S20)(EA+22)..(EA+23)
Df.WH = (S20)(EA+20)..(EA+21)
De.WL = (S20)(EA+18)..(EA+19)
De.WH = (S20)(EA+16)..(EA+17)
Dd.WL = (S20)(EA+14)..(EA+15)
Dd.WH = (S20)(EA+12)..(EA+13)
Dc.WL = (S20)(EA+10)..(EA+11)
Dc.WH = (S20)(EA+8)..(EA+9)
Db.WL = (S20)(EA+6)..(EA+7)
Db.WH = (S20)(EA+4)..(EA+5)
Da.WL = (S20)(EA+2)..(EA+3)
Da.WH = (S20)(EA+0)..(EA+1)
```

Table continues on the next page...

#	Syntax	Description
2	**LD2.16F (SAM),Da::Dh-mod8**	**Load 16 fractional words into a rotated register octet. The destination octet is an aligned group of 8 registers (D0-D7, D8-D15 etc.) which can be rotated in all modulo-8 consecutive combinations. All Standard addressing modes supported.**

	`Dh.WL = (S20)(EA+30)..(EA+31)`
	`Dh.WH = (S20)(EA+28)..(EA+29)`
	`Dg.WL = (S20)(EA+26)..(EA+27)`
	`Dg.WH = (S20)(EA+24)..(EA+25)`
	`Df.WL = (S20)(EA+22)..(EA+23)`
	`Df.WH = (S20)(EA+20)..(EA+21)`
	`De.WL = (S20)(EA+18)..(EA+19)`
	`De.WH = (S20)(EA+16)..(EA+17)`
	`Dd.WL = (S20)(EA+14)..(EA+15)`
	`Dd.WH = (S20)(EA+12)..(EA+13)`
	`Dc.WL = (S20)(EA+10)..(EA+11)`
	`Dc.WH = (S20)(EA+8)..(EA+9)`
	`Db.WL = (S20)(EA+6)..(EA+7)`
	`Db.WH = (S20)(EA+4)..(EA+5)`
	`Da.WL = (S20)(EA+2)..(EA+3)`
	`Da.WH = (S20)(EA+0)..(EA+1)`	

3	**LD2.16F (SAM),Da::Dh-mod8p**	**Load 16 fractional words into eight registers. Each register has a different modulo-8 index. For a detailed expalnation of the allowed combinations, see the end of the general description at the top. All Standard addressing modes supported.**

	`Dh.WL = (S20)(EA+30)..(EA+31)`
	`Dh.WH = (S20)(EA+28)..(EA+29)`
	`Dg.WL = (S20)(EA+26)..(EA+27)`
	`Dg.WH = (S20)(EA+24)..(EA+25)`
	`Df.WL = (S20)(EA+22)..(EA+23)`
	`Df.WH = (S20)(EA+20)..(EA+21)`
	`De.WL = (S20)(EA+18)..(EA+19)`
	`De.WH = (S20)(EA+16)..(EA+17)`
	`Dd.WL = (S20)(EA+14)..(EA+15)`
	`Dd.WH = (S20)(EA+12)..(EA+13)`
	`Dc.WL = (S20)(EA+10)..(EA+11)`
	`Dc.WH = (S20)(EA+8)..(EA+9)`
	`Db.WL = (S20)(EA+6)..(EA+7)`
	`Db.WH = (S20)(EA+4)..(EA+5)`
	`Da.WL = (S20)(EA+2)..(EA+3)`
	`Da.WH = (S20)(EA+0)..(EA+1)`	

Table continues on the next page...

#	Syntax	Description
4	**LD2.16F (SAM),Da::Dh-R3**	**Load 16 fractional words into a consecutive register octet, where the first 3 destination are shuffled. All Standard addressing modes supported.**

	`Dh.WL = (S20)(EA+30)..(EA+31)`
	`Dh.WH = (S20)(EA+28)..(EA+29)`
	`Dg.WL = (S20)(EA+26)..(EA+27)`
	`Dg.WH = (S20)(EA+24)..(EA+25)`
	`Df.WL = (S20)(EA+22)..(EA+23)`
	`Df.WH = (S20)(EA+20)..(EA+21)`
	`De.WL = (S20)(EA+18)..(EA+19)`
	`De.WH = (S20)(EA+16)..(EA+17)`
	`Dd.WL = (S20)(EA+14)..(EA+15)`
	`Dd.WH = (S20)(EA+12)..(EA+13)`
	`Dc.WL = (S20)(EA+10)..(EA+11)`
	`Dc.WH = (S20)(EA+8)..(EA+9)`
	`Db.WL = (S20)(EA+6)..(EA+7)`
	`Db.WH = (S20)(EA+4)..(EA+5)`
	`Da.WL = (S20)(EA+2)..(EA+3)`
	`Da.WH = (S20)(EA+0)..(EA+1)`	

5	**LD2.16F (SAM),Da::Dh-R5**	**Load 16 fractional words into a consecutive register octet, where five destinations (second to sixth) are shuffled. All Standard addressing modes supported.**

	`Dh.WL = (S20)(EA+30)..(EA+31)`
	`Dh.WH = (S20)(EA+28)..(EA+29)`
	`Dg.WL = (S20)(EA+26)..(EA+27)`
	`Dg.WH = (S20)(EA+24)..(EA+25)`
	`Df.WL = (S20)(EA+22)..(EA+23)`
	`Df.WH = (S20)(EA+20)..(EA+21)`
	`De.WL = (S20)(EA+18)..(EA+19)`
	`De.WH = (S20)(EA+16)..(EA+17)`
	`Dd.WL = (S20)(EA+14)..(EA+15)`
	`Dd.WH = (S20)(EA+12)..(EA+13)`
	`Dc.WL = (S20)(EA+10)..(EA+11)`
	`Dc.WH = (S20)(EA+8)..(EA+9)`
	`Db.WL = (S20)(EA+6)..(EA+7)`
	`Db.WH = (S20)(EA+4)..(EA+5)`
	`Da.WL = (S20)(EA+2)..(EA+3)`
	`Da.WH = (S20)(EA+0)..(EA+1)`	

6	**LD2.2F (SAM),Da**	**Load 2 fractional words into a register. All Standard addressing modes supported.**

	`Da.WL = (S20)(EA+2)..(EA+3)`
	`Da.WH = (S20)(EA+0)..(EA+1)`	

7	**LD2.32F (SAM),Da::Dp**	**Load 32 fractional words into sixteen consecutive registers. All Standard addressing modes supported. Special grouping limitations apply.**

Table continues on the next page...

#	Syntax	Description

	`Dp.WL = (S20)(EA+62)..(EA+63)`
	`Dp.WH = (S20)(EA+60)..(EA+61)`
	`Do.WL = (S20)(EA+58)..(EA+59)`
	`Do.WH = (S20)(EA+56)..(EA+57)`
	`Dn.WL = (S20)(EA+54)..(EA+55)`
	`Dn.WH = (S20)(EA+52)..(EA+53)`
	`Dm.WL = (S20)(EA+50)..(EA+51)`
	`Dm.WH = (S20)(EA+48)..(EA+49)`
	`Dl.WL = (S20)(EA+46)..(EA+47)`
	`Dl.WH = (S20)(EA+44)..(EA+45)`
	`Dk.WL = (S20)(EA+42)..(EA+43)`
	`Dk.WH = (S20)(EA+40)..(EA+41)`
	`Dj.WL = (S20)(EA+38)..(EA+39)`
	`Dj.WH = (S20)(EA+36)..(EA+37)`
	`Di.WL = (S20)(EA+34)..(EA+35)`
	`Di.WH = (S20)(EA+32)..(EA+33)`
	`Dh.WL = (S20)(EA+30)..(EA+31)`
	`Dh.WH = (S20)(EA+28)..(EA+29)`
	`Dg.WL = (S20)(EA+26)..(EA+27)`
	`Dg.WH = (S20)(EA+24)..(EA+25)`
	`Df.WL = (S20)(EA+22)..(EA+23)`
	`Df.WH = (S20)(EA+20)..(EA+21)`
	`De.WL = (S20)(EA+18)..(EA+19)`
	`De.WH = (S20)(EA+16)..(EA+17)`
	`Dd.WL = (S20)(EA+14)..(EA+15)`
	`Dd.WH = (S20)(EA+12)..(EA+13)`
	`Dc.WL = (S20)(EA+10)..(EA+11)`
	`Dc.WH = (S20)(EA+8)..(EA+9)`
	`Db.WL = (S20)(EA+6)..(EA+7)`
	`Db.WH = (S20)(EA+4)..(EA+5)`
	`Da.WL = (S20)(EA+2)..(EA+3)`
	`Da.WH = (S20)(EA+0)..(EA+1)`	

Table continues on the next page...

#	Syntax	Description
8	**LD2.32F (SAM),Da::Dp-mod16**	**Load 32 fractional words into a rotated set of 16 register. The destination register set is an aligned group of 16 registers (D0-D15, D16-D31 etc.) which can be rotated in all modulo-16 consecutive combinations. All Standard addressing modes supported. Special grouping limitations apply.**

	`Dp.WL = (S20)(EA+62)..(EA+63)`
	`Dp.WH = (S20)(EA+60)..(EA+61)`
	`Do.WL = (S20)(EA+58)..(EA+59)`
	`Do.WH = (S20)(EA+56)..(EA+57)`
	`Dn.WL = (S20)(EA+54)..(EA+55)`
	`Dn.WH = (S20)(EA+52)..(EA+53)`
	`Dm.WL = (S20)(EA+50)..(EA+51)`
	`Dm.WH = (S20)(EA+48)..(EA+49)`
	`Dl.WL = (S20)(EA+46)..(EA+47)`
	`Dl.WH = (S20)(EA+44)..(EA+45)`
	`Dk.WL = (S20)(EA+42)..(EA+43)`
	`Dk.WH = (S20)(EA+40)..(EA+41)`
	`Dj.WL = (S20)(EA+38)..(EA+39)`
	`Dj.WH = (S20)(EA+36)..(EA+37)`
	`Di.WL = (S20)(EA+34)..(EA+35)`
	`Di.WH = (S20)(EA+32)..(EA+33)`
	`Dh.WL = (S20)(EA+30)..(EA+31)`
	`Dh.WH = (S20)(EA+28)..(EA+29)`
	`Dg.WL = (S20)(EA+26)..(EA+27)`
	`Dg.WH = (S20)(EA+24)..(EA+25)`
	`Df.WL = (S20)(EA+22)..(EA+23)`
	`Df.WH = (S20)(EA+20)..(EA+21)`
	`De.WL = (S20)(EA+18)..(EA+19)`
	`De.WH = (S20)(EA+16)..(EA+17)`
	`Dd.WL = (S20)(EA+14)..(EA+15)`
	`Dd.WH = (S20)(EA+12)..(EA+13)`
	`Dc.WL = (S20)(EA+10)..(EA+11)`
	`Dc.WH = (S20)(EA+8)..(EA+9)`
	`Db.WL = (S20)(EA+6)..(EA+7)`
	`Db.WH = (S20)(EA+4)..(EA+5)`
	`Da.WL = (S20)(EA+2)..(EA+3)`
	`Da.WH = (S20)(EA+0)..(EA+1)`	

9	**LD2.4F (SAM),Da:Db**	**Load 4 fractional words into two consecutive registers. All Standard addressing modes supported.**

	`Db.WL = (S20)(EA+6)..(EA+7)`
	`Db.WH = (S20)(EA+4)..(EA+5)`
	`Da.WL = (S20)(EA+2)..(EA+3)`
	`Da.WH = (S20)(EA+0)..(EA+1)`	

Table continues on the next page...

#	Syntax	Description
10	**LD2.8F (SAM),Da:Db:Dc:Dd**	**Load 8 fractional words into four consecutive registers. All Standard addressing modes supported.**

	`Dd.WL = (S20)(EA+14)..(EA+15)`
	`Dd.WH = (S20)(EA+12)..(EA+13)`
	`Dc.WL = (S20)(EA+10)..(EA+11)`
	`Dc.WH = (S20)(EA+8)..(EA+9)`
	`Db.WL = (S20)(EA+6)..(EA+7)`
	`Db.WH = (S20)(EA+4)..(EA+5)`
	`Da.WL = (S20)(EA+2)..(EA+3)`
	`Da.WH = (S20)(EA+0)..(EA+1)`	

11	**LD2.8F (SAM),Da:Dc:Dd:Db**	**Load 8 fractional words into a consecutive register quad, where the last 3 destination are shuffled. All Standard addressing modes supported.**

	`Db.WL = (S20)(EA+14)..(EA+15)`
	`Db.WH = (S20)(EA+12)..(EA+13)`
	`Dd.WL = (S20)(EA+10)..(EA+11)`
	`Dd.WH = (S20)(EA+8)..(EA+9)`
	`Dc.WL = (S20)(EA+6)..(EA+7)`
	`Dc.WH = (S20)(EA+4)..(EA+5)`
	`Da.WL = (S20)(EA+2)..(EA+3)`
	`Da.WH = (S20)(EA+0)..(EA+1)`	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Da::Dh	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$ $0 \le n \le 56$
Da::Dh-mod8	$D_n:D_{n\&0x38+(n+1)\%8}:D_{n\&0x38+(n+2)\%8}:D_{n\&0x38+(n+3)\%8}:D_{n\&0x38+(n+4)\%8}:D_{n\&0x38+(n+5)\%8}:D_{n\&0x38+(n+6)\%8}:D_{n\&0x38+(n+7)\%8}$ $0 \le n \le 63$ Explicit list of all allowed combinations at Table C-7
Da::Dh-mod8p	$D_n:D_{(n\&0x38+8)\%64+(n+1)\%8}:D_{(n\&0x38+16)\%64+(n+2)\%8}:D_{(n\&0x38+24)\%64+(n+3)\%8}:D_{(n\&0x38+32)\%64+(n+4)\%8}:D_{(n\&0x38+40)\%64+(n+5)\%8}:D_{(n\&0x38+48)\%64+(n+6)\%8}:D_{(n\&0x38+54)\%64+(n+7)\%8}$ $0 \le n \le 63$ Explicit list of all allowed combinations at Table C-8
Da::Dh-R3	$D_{n+2}:D_n:D_{n+1}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$ $0 \le n \le 56$ Explicit list of all allowed combinations at Table C-9
Da::Dh-R5	$D_n:D_{n+2}:D_{n+4}:D_{n+5}:D_{n+3}:D_{n+1}:D_{n+6}:D_{n+7}$ $0 \le n \le 56$ Explicit list of all allowed combinations at Table C-10
Da::Dp	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}:D_{n+8}:D_{n+9}:D_{n+10}:D_{n+11}:D_{n+12}:D_{n+13}:D_{n+14}:D_{n+15}$ $0 \le n \le 48$
Da::Dp-mod16	$D_n:D_{n\&0x30+(n+1)\%16}:D_{n\&0x30+(n+2)\%16}:D_{n\&0x30+(n+3)\%16}:D_{n\&0x30+(n+4)\%16}:D_{n\&0x30+(n+5)\%16}:D_{n\&0x30+(n+6)\%16}:D_{n\&0x30+(n+7)\%16}:D_{n\&0x30+(n+8)\%16}:D_{n\&0x30+(n+9)\%16}:D_{n\&0x30+(n+10)\%16}:D_{n\&0x30+(n+11)\%16}:D_{n\&0x30+(n+12)\%16}:D_{n\&0x30+(n+13)\%16}:D_{n\&0x30+(n+14)\%16}:D_{n\&0x30+(n+15)\%16}$ $0 \le n \le 63$ Explicit list of all allowed combinations at Table C-11
Da:Db	$D_n:D_{n+1}$ $0 \le n \le 62$
Da:Db:Dc:Dd	$D_n:D_{n+1}:D_{n+2}:D_{n+3}$ $0 \le n \le 60$
Da:Dc:Dd:Db	$D_n:D_{n+2}:D_{n+3}:D_{n+1}$ $0 \le n \le 60$ Explicit list of all allowed combinations at Table C-14
Rk	R0-R7,R16-R23
Rn	R0-R31

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	All	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	5, 6, 8, 9, 11
Encoding length	32-bits	1, 2, 3, 4, 7, 10
	32-bits 48-bits	5, 6, 8, 9, 11
Pipeline behavior	agu_MOVE_LD_POSTINC	1, 2, 3, 4, 7, 10
	agu_MOVE_LD_POSTINC agu_MOVE_LD_PRECALC	5, 6, 8, 9, 11

LDRSTC Load-Reserve, Store Conditional (IPU)

General Description

This is a helper instruction that marks a parallel load instruction to perform a Load-Reserve operation in memory, or a store instruction to perform a Store-Conditional operation. Both operations are part of the flow to reserve a resource or semaphore. If grouped with a store instruction, SR.RF is updated with the failure status to reserve the resources, to be used by a subsequent JMPRF instruction. The flow with usage examples are described in detail in the Address Generation chapter. The load or store instruction that is grouped with LDRSTC must use LSU0. Some other programming rules apply, see rule A.9.

Instruction Variants

#	Syntax	Description
1	**LDRSTC**	**Mark the parallel load as 'load-reserve', or the parallel store as 'store conditional'.**
	`Assert LOAD AND RESERVE or STORE CONDITIONAL bus command with the parallel memory access`	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Helper instruction		All
Pipeline behavior	IPU_ACC_ATTR	All

LDSH2.nF→ D Load 16-bit Fractional Words From Memory, Pack and Shift in Data Registers (LSU)

General Description

Load two 16-bit fractional words from memory, sign extend each to 20 bits. The two values are packed into Da.WH and Da.WL. In addition, one of the loaded values is also loaded (duplicated) into a 20-bit portion of Db; which of the two values and into what portion depends on the variant. The other portion of Db is copied from the respective portion of Da before it was loaded. This instruction is useful to prepare the coefficients for the block FIR algorithm and similar ones that use an overlapping sliding window and hence need to use the same value in two destinations.

Flag Options

Flag	Position	Description
DN	flg1	Scale Down
UP	flg1	UP

Instruction Variants

#	Syntax	Operation	Description

1	LDSH2.DN.2F (SAM),Da:Db	`Da.WL = Da.WL`
		`Db.WH = Da.WH`
		`Da.WL = (S20)(EA+2)..(EA+3)`
		`Da.WH = (S20)(EA+0)..(EA+1)`
		`Db.WL = (S20)(EA+2)..(EA+3)`
			Da loaded with two packed 16-bit fractional words. The value loaded into Da.WL is also loaded into Db.WL. The original value of Da.WH is moved into Db.WH. The instruction is used to load values down the vector, bringing X(n), X(n+1) into Da and X(n+1), X(n+2) into Db.

2	LDSH2.UP.2F (SAM),Da:Db	`Da.WH = Da.WH`
		`Db.WL = Da.WL`
		`Da.WL = (S20)(EA+2)..(EA+3)`
		`Da.WH = (S20)(EA+0)..(EA+1)`
		`Db.WH = (S20)(EA+0)..(EA+1)`
			Da loaded with two packed 16-bit fractional words. The value loaded into Da.WH is also loaded into Db.WH. The original value of Da.WL is moved into Db.WL. The instruction is used to load values up the vector, bringing X(n+1), X(n) into Da and X(n), X(n-1) into Db.

Explicit Operands

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Rk	R0-R7,R16-R23	
Rn	R0-R31	

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	All	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits 48-bits	All
Pipeline behavior	agu_MOVE_LD_POSTINC_PARTIALW agu_MOVE_LD_PARTIALW agu_MOVE_LD_PRECALC_PARTIALW	2
	agu_MOVE_LD_POSTINC_PARTIALW agu_MOVE_LD_PRECALC_PARTIALW agu_MOVE_LD_PARTIALW	1

LOG2 Log 2 (DALU)

General Description

This instruction participates in the calculation of base 2 logarithm of the input multiplied by 2: LOG(2*X), where X is the input for the LOG2 instruction. It calculates two parameters, A and B, that are used for linear approximation of the result y = A*(2*X)+B using a subsequent instruction MACM.SU.LOG2.X.
For correct result input value X should be a fraction in the range of 0.5 ≤ X < 1. The accuracy of the output after the two instructions is 15 bits. The following code is an example of the sequence:

```
log2 d0,d4:d5 ; d0 input
macm.su.log2.x d0.h,d4,d5 ; d0 = log2(2x)
```

Instruction Variants

#	Syntax	Description
1	**LOG2 Da,Dm:Dn**	**Base 2 logarithm approximation**

```
Dm = 0
Dn = 0
Dn[39:32] = 0xff
Dm[31:14] = f_LOG2_LUT_A (Da[29:24])
Dn[31:15] = ~ f_LOG2_LUT_B (Da[29:24])
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dm:Dn	$D_n:D_{n+1}$	0 ≤ n ≤ 62

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

LPEND End of Hardware Loop Mark (PCU)

General Description

This instruction is located at the last VLES (LA) of hardware loop n. If the unsigned LCn is greater than 1, a PC-relative branch is performed to the specified label, and LCn is decremented by 1. Variants include a different instruction for sequential and non-sequential loops. The instruction is generated automatically by the assembler according to the LOOPSTART and LOOPEND directives, and should not be written explicitly in the assembly code. For more information about hardware loops, see the Program Control chapter.

Flag Options

Flag	Position	Description
SQ	flg1	Sequential Loop
n2	flg1	Loop index (0, 1, 2, 3)

Instruction Variants

#	Syntax	Description
1	**LPEND.SQn2 LABEL**	**End of loop mark for a sequential loop**
	`if ( (u32) LCn>1) then PC = PC - {RelAdd19,0}, LCn = LCn -1`	
2	**LPEND.n2 LABEL**	**End of loop mark for a non-sequential loop**
	`if ( (u32) LCn>1) then PC = PC - {RelAdd19,0}, LCn = LCn -1`	

Explicit Operands

Operand	Permitted Values
RelAdd19	$-2^{18} \le \text{RelAdd19} < 2^{18}$
n2	$0 \le \text{n2} \le 3$

Affect Instructions

Field	Relevant Variants	Comments
LC	All	
PC	All	

Affected By Instructions

Field	Relevant Variants
LC	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
No predication		All
Pipeline behavior	pcu_COF_LP	2
	pcu_COF_LPSEQ	1

LPSKIP Start of Hardware Loop Mark, With a Skip Test (PCU)

General Description

This instruction is located one VLES before the first VLES of hardware loop n (SA-1), and has two combined functions:

1. Marking the start of the loop, to prime the micro-architectural mechanisms for end-of-loop COF prediction
2. Performing a test that LC > 1, and if it is not, branch to the specified offset after the end of the loop.

The assembler generates this instruction automatically when encountering the LOOPSTART and LOOPEND directives, in case there is a SKIP instruction in SA-1. This instruction should not be written explicitly in the source assembly code. Variants include marks for sequential and non-sequential loops, as well as for signed and unsigned LC tests. The instruction encodes two relative offsets: one offset from the current PC to the end of the loop (LA), and the other the offset beyond LA that is used if the skip is taken. This extra offset is used by the assembler to encode the size of the VLES in LA, so that the skip will be done to LA+1. For more information on hardware loops see the Program Control chapter.

Flag Options

Flag	Position	Description
SQ	flg1	Sequential Loop
n2	flg1	Loop index (0, 1, 2, 3)
U	flg2	Unsigned

Instruction Variants

#	Syntax	Description
1	**LPSKIP.SQn2 LABEL,SIZE**	**Start of loop mark for a sequential loop, with a signed skipping check of LCn**
	`if ( (s32) LCn<1) then PC = PC + {RelAdd19,0} + {RRelAdd4,0}`	
2	**LPSKIP.n2 LABEL,SIZE**	**Start of loop mark for a non-sequential loop, with a signed skipping check of LCn**
	`if ( (s32) LCn<1) then PC = PC + {RelAdd19,0} + {RRelAdd4,0}`	
3	**LPSKIP.SQn2.U LABEL,SIZE**	**Start of loop mark for a sequential loop, with an unsigned skipping check of LCn**
	`if ( (u32) LCn==0) then PC = PC + {RelAdd19,0} + {RRelAdd4,0}`	
4	**LPSKIP.n2.U LABEL,SIZE**	**Start of loop mark for a non-sequential loop, with an unsigned skipping check of LCn**
	`if ( (u32) LCn==0) then PC = PC + {RelAdd19,0} + {RRelAdd4,0}`	

Explicit Operands

Operand	Permitted Values
RelAdd19	$-2^{18} \le \text{RelAdd19} < 2^{18}$
URRelAdd4	$1 \le \text{URRelAdd4} \le 16$
n2	$0 \le \text{n2} \le 3$

Affect Instructions

Field	Relevant Variants	Comments
LC	All	
PC	All	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	48-bits	All
No predication		All
Pipeline behavior	pcu_COF_LP	All

LPST Start of Hardware Loop Mark (PCU)

General Description

This instruction is located one VLES before the first VLES of hardware loop n (SA-1). It is used to mark the start of the loop The instruction does not modify any user-visible register, its function is to give the core micro architectural hints for activating the end-of-loop COF prediction mechanism. The assembler generates this instruction automatically when analyzing the LOOPSTART and LOOPEND directives, in case there is no SKIP instruction in SA-1. This instruction should not be written explicitly in the source assembly code. Variants include marks for sequential and non-sequential loops. The instruction encodes a PC-relative offset to the end of the loop (LA). For more information on hardware loops see the Program Control chapter.

Flag Options

Flag	Position	Description
SQ	flg1	Sequential Loop
n2	flg1	Loop index (0, 1, 2, 3)

Instruction Variants

#	Syntax	Operation	Description
1	LPST.SQn2 LABEL		Start of loop mark for a sequential loop.
2	LPST.n2 LABEL		Start of loop mark for a non-sequential loop.

Explicit Operands

Operand	Permitted Values
RelAdd19	$-2^{18} \leq \text{RelAdd19} < 2^{18}$
n2	$0 \leq n2 \leq 3$

Affect Instructions

Field	Relevant Variants	Comments
LC	All	
PC	All	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
No predication		All
Pipeline behavior	agu_LPSETUP	All

LSH.LL Logical Shift of 64 Bits (DALU)

General Description

Logically shift a 64-bit operand taken from two concatenated 32-bit portions of a D register pair, and store the result in a data register pair. The shift could be to the left or to the right. The shift amount is specified either as an unsigned immediate operand, or taken from a signed field in an additional register. In such a case, a positive value in this field will shift the operand in the direction specified in the mnemonic, and a negative value will shift the operand by the same amount to the other direction. A shift to the right will zero extend the operand up to bit 64, and discard bits from the right. A left shift will discard bits from the left, and add zeros to the inserted bits to the right of the operand. The extensions of the source data registers do not participate in the shift. The extensions of the destination registers are written with zeros. Note that there is no explicit logical shift left instruction with an immediate operant - it is identical with an arithmetic shift left with an immediate operand which should be used for this purpose.

Flag Options

Flag	Position	Description
LFT	flg1	Left
RGT	flg1	Right

Instruction Variants

#	Syntax	Description
1	**LSH.LFT.LL Da,Dc:Dd,Dm:Dn**	**Logically shift left according to a signed field in Da[6:0]. A positive value is a left shift, a negative is a right shift.**

	`If (Da[6:0] > 0)`
	`  then {Dm.M , Dn.M} = {Dc.M , Dd.M} << Da[6:0]`
	`  else {Dm.M , Dn.M} = {Dc.M , Dd.M} >> |Da[6:0]|`
	`Dm.E = 0, Dn.E = 0`	

2	**LSH.RGT.LL Da,Dc:Dd,Dm:Dn**	**Logically shift right according to a signed field in Da[6:0]. A positive value is a right shift, a negative is a left shift.**

	`If (Da[6:0] > 0)`
	`  then {Dm.M , Dn.M} = {Dc.M , Dd.M} >> Da[6:0]`
	`  else {Dm.M , Dn.M} = {Dc.M , Dd.M} << |Da[6:0]|`
	`Dm.E = 0, Dn.E = 0`	

3	**LSH.RGT.LL #u6,Da:Db,Dm:Dn**	**Logically shift right according to an unsigned immediate filed.**

	`{Dm.M , Dn.M} = {Da.M , Db.M} >> u6`
	`Dm.E = 0, Dn.E = 0`	

Explicit Operands

Operand	Permitted Values
Da	D0-D63

Table continues on the next page...

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
u6	$0 \le u6 < 2^6$	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	3
	dalu_LBM_Y	1, 2

LSH.nL Logical Shift of 32 Bits in Data Registers (DALU)

General Description

Logically shift a 32-bit operand taken from the low portion of a D register. The shift could be to the left or to the right. The shift amount is specified either as an unsigned immediate operand, or taken from a signed field in an additional register. In such a case, a positive value in this field will shift the operand in the direction specified in the mnemonic, and a negative value will shift the operand by the same amount to the other direction. A shift to the right will zero extend the operand up to bit 32, and discard bits from the right. A left shift will discard bits from the left, and add zeros to the inserted bits to the right of the operand. The extension of the source data register does not participate in the shift. The extension of the destination register is written with zeros. SIMD variants perform two parallel shifts by the same amount on two input-output register pairs. Note that the shift size field in variants that shift a single D register is larger than the respective field for SIMD shifts, for backwards compatibility reasons.

Flag Options

Flag	Position	Description
LFT	flg1	Left
RGT	flg1	Right

Instruction Variants

#	Syntax	Description
1	**LSH.LFT.2L Da,Dc:Dd,Dm:Dn**	**SIMD2 logical shift left of Dc into Dm, and Dd into Dn, according to a signed field in Da[5:0]. A positive value is a left shift, a negative is a right shift.**

	`Dm = (U40)(((S7)Da[5:0]) > 0) ? Dc << ((S7)Da[5:0]) : Dc >> - ((S7)Da[5:0])`
	`Dn = (U40)(((S7)Da[5:0]) > 0) ? Dd << ((S7)Da[5:0]) : Dd >> - ((S7)Da[5:0])`	

2	**LSH.LFT.L Da,Db,Dn**	**Logical shift left of Db into Dn, according to a signed field in Da[5:0]. A positive value is a left shift, a negative is a right shift.**
	`Dn = (Da[6:0] > 0) ? (U40) Db.M << Da[6:0] : (U40) Db.M >> |Da[6:0]|`	
3	**LSH.RGT.2L Da,Dc:Dd,Dm:Dn**	**SIMD2 logical shift right of Dc into Dm, and Dd into Dn, according to a signed field in Da[5:0]. A positive value is a right shift, a negative is a left shift.**

	`Dm = (U40)(((S7)Da[5:0]) > 0) ? Dc >> ((S7)Da[5:0]) : Dc << - ((S7)Da[5:0])`
	`Dn = (U40)(((S7)Da[5:0]) > 0) ? Dd >> ((S7)Da[5:0]) : Dd << - ((S7)Da[5:0])`	

Table continues on the next page...

#	Syntax	Description
4	**LSH.RGT.2L #u5,Da:Db,Dm:Dn**	**SIMD2 logical shift right of Da into Dm, and Db into Dn, according to an unsigned immediate.**

	`Dm = (U40)Da >> u5`
	`Dn = (U40)Db >> u5`	

5	**LSH.RGT.L Da,Db,Dn**	**Logical shift right of Db into Dn, according to a signed field in Da[5:0]. A positive value is a right shift, a negative is a left shift.**
	`Dn =  (Da[6:0] > 0) ? (U32) Db.M >> Da[6:0] : (U32) Db.M << |Da[6:0]|`	
6	**LSH.RGT.L #u5,Da,Dn**	**Logical shift right of Da into Dn, according to an unsigned immediate.**
	`Dn = (U40)(Da.M >> u5)`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n{:}D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n{:}D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n{:}D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	
u5	$0 \le u5 < 2^5$	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	2, 4, 5, 6
	dalu_LBM_Y	1, 3

LSH.nW Logical Shift of 16 Bits in Data Registers (DALU)

General Description

Logically shift pairs of packed 16-bit operands taken from the high (H) and low (L) portions of D register. The shift could be to the left or to the right. The shift amount is specified either as an unsigned immediate operand, or taken from a signed field in an additional register. In such a case, a positive value in this field will shift the operand in the direction specified in the mnemonic, and a negative value will shift the operand by the same amount to the other direction. A shift to the right will zero extend the operand up to 16 bits, and discard bits from the right. A left shift will discard bits from the left, and add zeros to the inserted bits to the right of the operand. A left shift by more than 15 will return 0. The basic operation shifts the two H and L portions that are packed in a register by the same amount. Additional variants shift four 16-bit portions that are packed in two registers. A few variants preserve the legacy behavior of performing a 20-bit shift on WH and WL instead of a 16-bit shift if SR.W20 is set.

Flag Options

Flag	Position	Description
LFT	flg1	Left
RGT	flg1	Right

Instruction Variants

#	Syntax	Description
1	**LSH.LFT.2W Da,Db,Dn**	**If SR.W20 is clear (by default), perform dual 16-bit shift on Db.H and Db.L. If SR.W20 is set, perform dual 20-bit shift on Db.WH and Db.WL. The shift size is taken from a signed field in Da; the size of the field depends on SR.W20. Negative shift sizes perform a right shift.**

```
If (SR.W20)
  Dn.WL = (Da[5:0] > 0) ? (U20) Db.WL << Da[5:0] : (U20) Db.WL >> |Da[5:0]|
  Dn.WH = (Da[5:0] > 0) ? (U20) Db.WH << Da[5:0] : (U20) Db.WH >> |Da[5:0]|
else
  Dn.L = (Da[4:0] > 0) ? (U16) Db.L << Da[4:0] : (U16) Db.L >> |Da[4:0]|
  Dn.H = (Da[4:0] > 0) ? (U16) Db.H << Da[4:0] : (U16) Db.H >> |Da[4:0]|
  Dn.E = 0
```

Table continues on the next page...

<table>
<tr><th>#</th><th>Syntax</th><th>Description</th></tr>
<tr><td>2</td><td>LSH.LFT.4W Da,Dc:Dd,Dm:Dn</td><td>SIMD4 logical shift left of four packed words in Dc and Dd into the respective portions in Dm and Dn, according to a signed field in Da[5:0]. A positive value is a left shift, a negative is a right shift.</td></tr>
<tr><td></td><td colspan="2"><pre>
shift_size = Da[4:0]
if (Da[4:0] > (0)) {
        Dm.H = Dc.H << shift_size
        Dm.L = Dc.L << shift_size
        Dn.H = Dd.H << shift_size
        Dn.L = Dd.L << shift_size
} else {
        shift_size = (~ shift_size + 1)
        Dm.H = Dc.H >> shift_size
        Dm.L = Dc.L >> shift_size
        Dn.H = Dd.H >> shift_size
        Dn.L = Dd.L >> shift_size
}
Dm.E = 0
Dn.E = 0
</pre></td></tr>
<tr><td>3</td><td>LSH.RGT.2W Da,Db,Dn</td><td>If SR.W20 is clear (by default), perform dual 16-bit shift on Db.H and Db.L. If SR.W20 is set, perform dual 20-bit shift on Db.WH and Db.WL. The shift size is taken from a signed field in Da; the size of the field depends on SR.W20. Negative shift sizes perform a left shift.</td></tr>
<tr><td></td><td colspan="2"><pre>
If (SR.W20)
  Dn.WL = (Da[5:0] > 0) ? (U20) Db.WL >> Da[5:0] : (U20) Db.WL << |Da[5:0]|
  Dn.WH = (Da[5:0] > 0) ? (U20) Db.WH >> Da[5:0] : (U20) Db.WH << |Da[5:0]|
else
  Dn.L = (Da[4:0] > 0) ? (U16) Db.L >> Da[4:0] : (U16) Db.L << |Da[4:0]|
  Dn.H = (Da[4:0] > 0) ? (U16) Db.H >> Da[4:0] : (U16) Db.H << |Da[4:0]|
  Dn.E = 0
</pre></td></tr>
<tr><td>4</td><td>LSH.RGT.2W #u5,Da,Dn</td><td>If SR.W20 is clear (by default), perform dual 16-bit shift on Db.H and Db.L. If SR.W20 is set, perform dual 20-bit shift on Db.WH and Db.WL. The shift size is an unsigned immediate operand; the number of bits used from it depends on SR.W20. Negative shift sizes perform a left shift.</td></tr>
<tr><td></td><td colspan="2"><pre>
if (SR.W20 == 1) {
        Dn.WL = Da.WL >> u5
        Dn.WH = Da.WH >> u5
} else {
        Dn.L = Da.L >> u5[3:0]
        Dn.H = Da.H >> u5[3:0]
        Dn.E = 0
}
</pre></td></tr>
<tr><td>5</td><td>LSH.RGT.4W Da,Dc:Dd,Dm:Dn</td><td>SIMD4 logical shift right of four packed words in Dc and Dd into the respective portions in Dm and Dn, according to a signed field in Da[5:0]. A positive value is a right shift, a negative is a left shift.</td></tr>
</table>

Table continues on the next page...

#	Syntax	Description

```
shift_size = Da[4:0]
if (Da[4:0]  > (0)) {
        Dm.H = Dc.H >> shift_size
        Dm.L = Dc.L >> shift_size
        Dn.H = Dd.H >> shift_size
        Dn.L = Dd.L >> shift_size
} else {
        shift_size = (~ shift_size + 1)
        Dm.H = Dc.H << shift_size
        Dm.L = Dc.L << shift_size
        Dn.H = Dd.H << shift_size
        Dn.L = Dd.L << shift_size
}
Dm.E = 0
Dn.E = 0
```

#	Syntax	Description
6	**LSH.RGT.4W #u4,Da:Db,Dm:Dn**	**SIMD4 logical shift right of four packed words in Dc and Dd into the respective portions in Dm and Dn, according to an unsigned immediate operand.**

```
Dm.H = Da.H >> u4
Dm.L = Da.L >> u4
Dn.H = Db.H >> u4
Dn.L = Db.L >> u4
Dm.E = 0
Dn.E = 0
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	
u4	$0 \le u4 < 2^4$	
u5	$0 \le u5 < 2^5$	

Affect Instructions

Field	Relevant Variants	Comments
SR.W20	1, 3, 4	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	4, 6
	dalu_LBM_Y	1, 2, 3, 5

LSH.nX Logical Shift of 40 Bits in Data Registers (DALU)

General Description

Logically shift 40-bit operands from D registers. The shift could be to the left or to the right. The shift amount is specified either as an unsigned immediate operand, or taken from a signed field in an additional register. In such a case, a positive value in this field will shift the operand in the direction specified in the mnemonic, and a negative value will shift the operand by the same amount to the other direction. A shift to the right will zero extend the operand up to 40 bits, and discard bits from the right. A left shift will discard bits from the left, and add zeros to the inserted bits to the right of the operand. A shift by more than 39 will return 0. SIMD variants shift 2 registers by the same amount.

Flag Options

Flag	Position	Description
LFT	flg1	Left
RGT	flg1	Right

Instruction Variants

#	Syntax	Description
1	**LSH.LFT.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 logical shift left of Dc into Dm, and Dd into Dn, according to a signed field in Da[6:0], affecting both shifts. A positive value is a left shift, a negative is a right shift.**

	`Dm = (Da[6:0] > 0) ? Dc << Da[6:0] : Dc >> - Da[6:0]`
	`Dn = (Da[6:0] > 0) ? Dd << Da[6:0] : Dd >> - Da[6:0]`	

2	**LSH.LFT.X Da,Db,Dn**	**Logical shift left of Db into Dn, according to a signed field in Da[6:0]. A positive value is a left shift, a negative is a right shift.**
	`Dn =  (Da[6:0] > 0) ? (U40) Db << Da[6:0] : (U40) Db >> |Da[6:0]|`	
3	**LSH.RGT.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 logical shift right of Dc into Dm, and Dd into Dn, according to a signed field in Da[6:0], affecting both shifts. A positive value is a right shift, a negative is a left shift.**

	`Dm = (Da[6:0] > 0) ? Dc >> Da[6:0] : Dc << - Da[6:0]`
	`Dn = (Da[6:0] > 0) ? Dd >> Da[6:0] : Dd << - Da[6:0]`	

4	**LSH.RGT.2X #ue5,Da:Db,Dm:Dn**	**SIMD2 logical shift left of Dc into Dm, and Dd into Dn, according to an unsigned immediate, affecting both shifts.**

	`Dm = Da >> ue5`
	`Dn = Db >> ue5`	

5	**LSH.RGT.X Da,Db,Dn**	**Logical shift right of Db into Dn, according to a signed field in Da[6:0]. A positive value is a right shift, a negative is a left shift.**

Table continues on the next page...

#	Syntax	Description
	`Dn = (Da[6:0] > 0) ? Db >> Da[6:0] : Db << - Da[6:0]`	
6	**LSH.RGT.X #ue5,Da,Dn**	**Logical shift left of Da into Dn, according to an unsigned immediate.**
	`Dn = Da >> ue5`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	
ue5	$1 \le ue5 \le 32$	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

LSHA Logical Shift of 32 Bits in Address Registers (IPU,LSU)

General Description

Logically shift a 32-bit operand taken from an R register. The shift could be to the left or to the right. The shift amount is specified either as an unsigned immediate operand, or taken from a signed field in an additional register. In such a case, a positive value in this field will shift the operand in the direction specified in the mnemonic, and a negative value will shift the operand by the same amount to the other direction. A shift to the right will zero extend the operand up to bit 32, and discard bits from the right. A left shift will discard bits from the left, and add zeros to the inserted bits to the right of the operand. A shift by 32 will return zero.

Flag Options

Flag	Position	Description
LFT	flg1	Left
RGT	flg1	Right

Instruction Variants

#	Syntax	Description
1	**LSHA.LFT Ra,Rb,Rn**	**Logical shift left of Rb into Rn, according to a signed field in Ra[5:0]. A positive value is a left shift, a negative is a right shift.**
	`Rn = (Ra[5:0] > 0) ? Rb << Ra[5:0] : Rb >> - Ra[5:0]`	
2	**LSHA.RGT Ra,Rb,Rn**	**Logical shift right of Rb into Rn, according to a signed field in Ra[5:0]. A positive value is a right shift, a negative is a left shift.**
	`Rn = (Ra[5:0] > 0) ? Rb >> Ra[5:0] : Rb << - Ra[5:0]`	
3	**LSHA.RGT #u5,Ra,Rn**	**Logical shift right of Ra into Rn, according to an unsigned immediate.**
	`Rn = Ra >> u5`	

Explicit Operands

Operand	Permitted Values
Ra	R0-R31
Rb	R0-R31
Rn	R0-R31
u5	$0 \leq u5 < 2^5$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Execution unit	IPU	All
	LSU	3
Pipeline behavior	agu_AAU_LOGIC	All

MAC.LEG.X 16x16 bit Legacy Multiply-Accumulate into 40-bit Result (DALU)

General Description

Performs a single 16x16 bit multiply-accumulate operation into 40-bit results. The calculation is affected by arithmetic modes in SR the same way as in the legacy StarCore architectures. In case SR.SM is set, the result is saturated from 40 to to 32 bits, sign extending the extension. In variants that include rounding, the lower 16 bits of the result are rounded according to the rounding mode SR.RM and scaling mode SR.SCM.

Flag Options

Flag	Position	Description
LEG	flg2	Legacy instruction
R	flg2	Round multiplication result

Instruction Variants

#	Syntax	Description
1	**MAC.LEG.X Da.h,Db.h,Dn**	**Single 16x16 bit fractional multiply-accumulate, saturating only if SR.SM is set**
	`Dn = srSAT32((Dn + ((Da.H * Db.H) << 1)))`	
2	**MAC.LEG.X Da.h,Db.l,Dn**	**Single 16x16 bit fractional multiply-accumulate, saturating only if SR.SM is set**
	`Dn = srSAT32((Dn + ((Da.H * Db.L) << 1)))`	
3	**MAC.LEG.X Da.l,Db.l,Dn**	**Single 16x16 bit fractional multiply-accumulate, saturating only if SR.SM is set**
	`Dn = srSAT32((Dn + ((Da.L * Db.L) << 1)))`	
4	**MAC.LEG.X #s16,Da.h,Dn**	**Single 16x16 bit fractional multiply-accumulate using immediate value, saturating only if SR.SM is set**
	`Dn = srSAT32((Dn + ((Da.H * s16_t2) << 1)))`	
5	**MAC.LEG.X #s16,Da.l,Dn**	**Single 16x16 bit fractional multiply-accumulate using immediate value, saturating only if SR.SM is set**
	`Dn = srSAT32((Dn + ((Da.L * s16_t2) << 1)))`	
6	**MAC.LEG.X -Da.h,Db.h,Dn**	**Single 16x16 bit fractional multiply-accumulate with negate, saturating only if SR.SM is set**
	`Dn = srSAT32((Dn - ((Da.H * Db.H) << 1)))`	
7	**MAC.LEG.X -Da.h,Db.l,Dn**	**Single 16x16 bit fractional multiply-accumulate with negate, saturating only if SR.SM is set**
	`Dn = srSAT32((Dn - ((Da.H * Db.L) << 1)))`	

Table continues on the next page...

#	Syntax	Description
8	**MAC.LEG.X -Da.l,Db.l,Dn**	**Single 16x16 bit fractional multiply-accumulate with negate, saturating only if SR.SM is set**
	`Dn = srSAT32((Dn - ((Da.L * Db.L) << 1)))`	
9	**MAC.RLEG.X Da.h,Db.h,Dn**	**Single 16x16 bit fractional multiply-accumulate, legacy rounding according to SR.RM and SR.SCM**
	`Dn = srRND_SAT32 ((Dn + ((Da.H * Db.H) << 1)))[39:0]`	
10	**MAC.RLEG.X Da.l,Db.h,Dn**	**Single 16x16 bit fractional multiply-accumulate, legacy rounding according to SR.RM and SR.SCM**
	`Dn = srRND_SAT32 ((Dn + ((Da.L * Db.H) << 1)))[39:0]`	
11	**MAC.RLEG.X Da.l,Db.l,Dn**	**Single 16x16 bit fractional multiply-accumulate, legacy rounding according to SR.RM and SR.SCM**
	`Dn = srRND_SAT32 ((Dn + ((Da.L * Db.L) << 1)))[39:0]`	
12	**MAC.RLEG.X -Da.h,Db.h,Dn**	**Single 16x16 bit fractional multiply-accumulate with negate, legacy rounding according to SR.RM and SR.SCM**
	`Dn = srRND_SAT32 ((Dn - ((Da.H * Db.H) << 1)))[39:0]`	
13	**MAC.RLEG.X -Da.l,Db.h,Dn**	**Single 16x16 bit fractional multiply-accumulate with negate, legacy rounding according to SR.RM and SR.SCM**
	`Dn = srRND_SAT32 ((Dn - ((Da.L * Db.H) << 1)))[39:0]`	
14	**MAC.RLEG.X -Da.l,Db.l,Dn**	**Single 16x16 bit fractional multiply-accumulate with negate, legacy rounding according to SR.RM and SR.SCM**
	`Dn = srRND_SAT32 ((Dn - ((Da.L * Db.L) << 1)))[39:0]`	

Explicit Operands

Operand	Permitted Values
Da	`D0-D63`
Db	`D0-D63`
Dn	`D0-D63`
s16	$-2^{15} \le s16 < 2^{15}$

Affect Instructions

Field	Relevant Variants	Comments
SR.RM	9, 10, 11, 12, 13, 14	
SR.SCM	9, 10, 11, 12, 13, 14	
SR.SM	All	

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	1, 2, 3, 6, 7, 8, 9, 10, 11, 12, 13, 14
	64-bits	4, 5
Pipeline behavior	dalu_MPY_Acc	All

MAC.nT 16x16 bit Multiply-Accumulate into 20-bit Result (DALU)

General Description

Performs SIMD2/SIMD4 16x16 bit multiply-accumulate into 20-bit results.

Flag Options

Flag	Position	Description
L	flg1	Use the low portion
SU	flg1	Signed By Unsigned
I	flg2	Integer arithmetic
R	flg2	Round multiplication result

Instruction Variants

#	Syntax	Description
1	**MAC.2T Da,Db,Dn**	**SIMD2 fractional multiply-accumulate**

```
if ((SR.SM2==0) || (SR.W20==1 && SR.SM2==1))
  Dn.WL = (S20) (Dn.WL + ((Da.L * Db.L)<<1)>>16)
  Dn.WH = (S20) (Dn.WH + ((Da.H * Db.H)<<1)>>16)
else
  Dn.L = srSAT16(Dn.L + ((Da.L * Db.L)<<1)>>16)
  Dn.H = srSAT16(Dn.H + ((Da.H * Db.H)<<1)>>16)
  Dn.E= 0
```

#	Syntax	Description
2	**MAC.2T -Da,Db,Dn**	**SIMD2 fractional multiply-accumulate with negate**

```
if ((SR.SM2==0) || (SR.W20==1 && SR.SM2==1))
  Dn.WL = (S20) (Dn.WL + ((-Da.L * Db.L)<<1)>>16)
  Dn.WH = (S20) (Dn.WH + ((-Da.H * Db.H)<<1)>>16)
else
  Dn.L = srSAT16(Dn.L + ((-Da.L * Db.L)<<1)>>16)
  Dn.H = srSAT16(Dn.H + ((-Da.H * Db.H)<<1)>>16)
  Dn.E= 0
```

#	Syntax	Description
3	**MAC.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 fractional multiply-accumulate**

```
Dm.WH = (Dm.WH + (((Da.H * Dc.H) << 1) >> 16))[19:0]
Dm.WL = (Dm.WL + (((Da.L * Dc.L) << 1) >> 16))[19:0]
Dn.WH = (Dn.WH + (((Db.H * Dd.H) << 1) >> 16))[19:0]
Dn.WL = (Dn.WL + (((Db.L * Dd.L) << 1) >> 16))[19:0]
```

#	Syntax	Description
4	**MAC.4T -Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 fractional multiply-accumulate with negate**

```
Dm.WH = (Dm.WH + ((-((Da.H * Dc.H) << 1)) >> 16))[19:0]
Dm.WL = (Dm.WL + ((-((Da.L * Dc.L) << 1)) >> 16))[19:0]
Dn.WH = (Dn.WH + ((-((Db.H * Dd.H) << 1)) >> 16))[19:0]
Dn.WL = (Dn.WL + ((-((Db.L * Dd.L) << 1)) >> 16))[19:0]
```

#	Syntax	Description
5	**MAC.L.I.4T Da:Db,Dc:Dd,Dm:Dn**	

Table continues on the next page...

#	Syntax	Description

	Dm.WH = (Dm.WH + (Da.H * Dc.H))[19:0]
	Dm.WL = (Dm.WL + (Da.L * Dc.L))[19:0]
	Dn.WH = (Dn.WH + (Db.H * Dd.H))[19:0]
	Dn.WL = (Dn.WL + (Db.L * Dd.L))[19:0]	

6	**MAC.SU.I.2T Da,Db,Dn**	**SIMD2 signed by unsigned integer multiply-accumulate**

	Dn.WH = (S20)(((Da.H * Db.H)) + Dn.WH)[19:0]
	Dn.WL = (S20)(((Da.L * Db.L)) + Dn.WL)[19:0]	

7	**MAC.IR.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 integer multiply-accumulate with rounding**

	Dm.WH = (Dm.WH + (((Da.H * Dc.H) + 0x8000) >> 16))[19:0]
	Dm.WL = (Dm.WL + (((Da.L * Dc.L) + 0x8000) >> 16))[19:0]
	Dn.WH = (Dn.WH + (((Db.H * Dd.H) + 0x8000) >> 16))[19:0]
	Dn.WL = (Dn.WL + (((Db.L * Dd.L) + 0x8000) >> 16))[19:0]	

8	**MAC.R.2T Da,Db,Dn**	**SIMD2 fractional multiply-accumulate with rounding**

	if ((SR.SM2==0) || (SR.W20==1 && SR.SM2==1))
	  Dn.WL = (S20) (Dn.WL + ((Da.L * Db.L)<<1 + 0x8000)>>16)
	  Dn.WH = (S20) (Dn.WH + ((Da.H * Db.H)<<1 + 0x8000)>>16)
	else
	  Dn.L = srSAT16(Dn.L + ((Da.L * Db.L)<<1 + 0x8000)>>16)
	  Dn.H = srSAT16(Dn.H + ((Da.H * Db.H)<<1 + 0x8000)>>16)
	  Dn.E= 0	

9	**MAC.R.2T -Da,Db,Dn**	**SIMD2 fractional multiply-accumulate with negate and rounding**

	if ((SR.SM2==0) || (SR.W20==1 && SR.SM2==1))
	  Dn.WL = (S20) (Dn.WL + ((-Da.L * Db.L)<<1 + 0x8000)>>16)
	  Dn.WH = (S20) (Dn.WH + ((-Da.H * Db.H)<<1 + 0x8000)>>16)
	else
	  Dn.L = srSAT16(Dn.L + ((-Da.L * Db.L)<<1 + 0x8000)>>16)
	  Dn.H = srSAT16(Dn.H + ((-Da.H * Db.H)<<1 + 0x8000)>>16)
	  Dn.E= 0	

10	**MAC.R.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 fractional multiply-accumulate with rounding**

	Dm.WH = (Dm.WH + ((((Da.H * Dc.H) << 1) + 0x8000) >> 16))[19:0]
	Dm.WL = (Dm.WL + ((((Da.L * Dc.L) << 1) + 0x8000) >> 16))[19:0]
	Dn.WH = (Dn.WH + ((((Db.H * Dd.H) << 1) + 0x8000) >> 16))[19:0]
	Dn.WL = (Dn.WL + ((((Db.L * Dd.L) << 1) + 0x8000) >> 16))[19:0]	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Affect Instructions

Field	Relevant Variants	Comments
SR.SM2	1, 2, 8, 9	
SR.W20	1, 2, 8, 9	

Affected By Instructions

Field	Relevant Variants
SR.SAT	1, 2, 8, 9

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	3, 4, 7
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MAC.nW 16x16 bit Multiply-Accumulate into 16-bit Result with Saturation (DALU)

General Description

Performs SIMD2/SIMD4 16x16 bit multiply-accumulate into 16-bit results.

Flag Options

Flag	Position	Description
I	flg2	Integer arithmetic
R	flg2	Round multiplication result
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MAC.ISR.4W Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 integer multiply-accumulate with rounding and saturation**

```
Dm.H = SAT16 ((Dm.H + (((Da.H * Dc.H) + 0x8000) >> 16)))[15:0]
Dm.L = SAT16 ((Dm.L + (((Da.L * Dc.L) + 0x8000) >> 16)))[15:0]
Dm.E = 0
Dn.H = SAT16 ((Dn.H + (((Db.H * Dd.H) + 0x8000) >> 16)))[15:0]
Dn.L = SAT16 ((Dn.L + (((Db.L * Dd.L) + 0x8000) >> 16)))[15:0]
Dn.E = 0
```

#	Syntax	Description
2	**MAC.S.2W Da,Db,Dn**	**SIMD2 fractional multiply-accumulate with saturation**

```
Dn.H = SAT16 ((Dn.H + (((Da.H * Db.H) << 1) >> 16)))[15:0]
Dn.L = SAT16 ((Dn.L + (((Da.L * Db.L) << 1) >> 16)))[15:0]
Dn.E = 0
```

#	Syntax	Description
3	**MAC.SR.2W Da,Db,Dn**	**SIMD2 fractional multiply-accumulate with rounding and saturation**

```
Dn.H = SAT16 ((Dn.H + ((((Da.H * Db.H) << 1) + 0x8000) >> 16)))[15:0]
Dn.L = SAT16 ((Dn.L + ((((Da.L * Db.L) << 1) + 0x8000) >> 16)))[15:0]
Dn.E = 0
```

#	Syntax	Description
4	**MAC.SR.4W Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 fractional multiply-accumulate with rounding and saturation**

```
Dm.H = SAT16 ((Dm.H + ((((Da.H * Dc.H) << 1) + 0x8000) >> 16)))[15:0]
Dm.L = SAT16 ((Dm.L + ((((Da.L * Dc.L) << 1) + 0x8000) >> 16)))[15:0]
Dm.E = 0
Dn.H = SAT16 ((Dn.H + ((((Db.H * Dd.H) << 1) + 0x8000) >> 16)))[15:0]
Dn.L = SAT16 ((Dn.L + ((((Db.L * Dd.L) << 1) + 0x8000) >> 16)))[15:0]
Dn.E = 0
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \leq n \leq 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \leq n \leq 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \leq n \leq 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MAC.nX 16x16 bit Multiply-Accumulate into 40-bit Result (DALU)

General Description

Performs single/SIMD2 16x16 bit multiply-accumulate into 40-bit results.

Flag Options

Flag	Position	Description
H	flg1	High 16-bit portion
HH	flg1	High 16 bits from the first argument and High 16 bits from the second argument
L	flg1	Use the low portion
LL	flg1	SIMD multiplication: low by low portions Extract and Sat instructions: 64-bit input
US	flg1	Unsigned By Signed
UU	flg1	Unsigned By Unsigned
I	flg2	Integer arithmetic
R	flg2	Round multiplication result
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MAC.2X Da,Db,Dm:Dn**	**SIMD2 fractional multiply-accumulate, multiplying high and low portions**

	`Dm = (Dm + ((Da.H * Db.H) << 1))[39:0]`
	`Dn = (Dn + ((Da.L * Db.L) << 1))[39:0]`	

2	**MAC.X Da.h,Db.h,Dn**	**Single fractional multiply-accumulate , multiplying high by high portions**
	`Dn = (Dn + ((Da.H * Db.H) << 1))[39:0]`	
3	**MAC.X Da.h,Db.l,Dn**	**Single fractional multiply-accumulate , multiplying high by low portions**
	`Dn = (Dn + ((Da.H * Db.L) << 1))[39:0]`	
4	**MAC.X Da.l,Db.l,Dn**	**Single fractional multiply-accumulate , multiplying low by low portions**
	`Dn = (Dn + ((Da.L * Db.L) << 1))[39:0]`	
5	**MAC.X #s16,Da.h,Dn**	**Single fractional multiply-accumulate using immediate value**
	`Dn = (Dn + ((Da.H * s16_t2) << 1))[39:0]`	

Table continues on the next page...

#	Syntax	Description
6	**MAC.X #s16,Da.l,Dn**	**Single fractional multiply-accumulate using immediate value**
	`Dn = (Dn + ((Da.L * s16_t2) << 1))[39:0]`	
7	**MAC.H.2X Da,Db,Dm:Dn**	**SIMD2 fractional multiply-accumulate, high portion by two portions**

	`Dm = (Dm + ((Da.H * Db.H) << 1))[39:0]`
	`Dn = (Dn + ((Da.H * Db.L) << 1))[39:0]`	

8	**MAC.HH.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional multiply-accumulate, high portions by high portions**

	`Dm = (Dm + ((Da.H * Dc.H) << 1))[39:0]`
	`Dn = (Dn + ((Db.H * Dd.H) << 1))[39:0]`	

9	**MAC.L.2X Da,Db,Dm:Dn**	**SIMD2 fractional multiply-accumulate, low portion by two portions**

	`Dm = (Dm + ((Da.L * Db.H) << 1))[39:0]`
	`Dn = (Dn + ((Da.L * Db.L) << 1))[39:0]`	

10	**MAC.LL.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional multiply-accumulate, low portions by low portions**

	`Dm = (Dm + ((Da.L * Dc.L) << 1))[39:0]`
	`Dn = (Dn + ((Db.L * Dd.L) << 1))[39:0]`	

11	**MAC.I.2X Da,Db,Dm:Dn**	**SIMD2 integer multiply-accumulate, multiplying high and low portions**

	`Dm = (Dm + (Da.H * Db.H))[39:0]`
	`Dn = (Dn + (Da.L * Db.L))[39:0]`	

12	**MAC.I.2X -Da,Db,Dm:Dn**	**SIMD2 integer multiply-accumulate with negate, multiplying high and low portions**

	`Dm = (Dm - (Da.H * Db.H))[39:0]`
	`Dn = (Dn - (Da.L * Db.L))[39:0]`	

13	**MAC.I.X Da.h,Db.h,Dn**	**Single integer multiply-accumulate, multiplying high by high portions**
	`Dn = (Dn + (Da.H * Db.H))[39:0]`	
14	**MAC.I.X Da.h,Db.l,Dn**	**Single integer multiply-accumulate , multiplying high by low portions**
	`Dn = (Dn + (Da.H * Db.L))[39:0]`	
15	**MAC.I.X Da.l,Db.l,Dn**	**Single integer multiply-accumulate, multiplying low by low portions**
	`Dn = (Dn + (Da.L * Db.L))[39:0]`	
16	**MAC.I.X -Da.h,Db.h,Dn**	**Single integer multiply-accumulate with negate, multiplying high by high portions**
	`Dn = (Dn - (Da.H * Db.H))[39:0]`	
17	**MAC.I.X -Da.h,Db.l,Dn**	**Single integer multiply-accumulate with negate , multiplying high by low portions**
	`Dn = (Dn - (Da.H * Db.L))[39:0]`	

Table continues on the next page...

#	Syntax	Description
18	**MAC.I.X -Da.l,Db.l,Dn**	**Single integer multiply-accumulate with negate, multiplying low by low portions**
	`Dn = (Dn - (Da.L * Db.L))[39:0]`	
19	**MAC.H.I.2X Da,Db,Dm:Dn**	**SIMD2 integer multiply-accumulate, high portion by two portions**

	`Dm = (Dm + (Da.H * Db.H))[39:0]`
	`Dn = (Dn + (Da.H * Db.L))[39:0]`	

20	**MAC.HH.I.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 integer multiply-accumulate, high portions by high portions**

	`Dm = (Dm + (Da.H * Dc.H))[39:0]`
	`Dn = (Dn + (Db.H * Dd.H))[39:0]`	

21	**MAC.L.I.2X Da,Db,Dm:Dn**	**SIMD2 integer multiply-accumulate, low portion by two portions**

	`Dm = (Dm + (Da.L * Db.H))[39:0]`
	`Dn = (Dn + (Da.L * Db.L))[39:0]`	

22	**MAC.LL.I.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 integer multiply-accumulate, low portions by low portions**

	`Dm = (Dm + (Da.L * Dc.L))[39:0]`
	`Dn = (Dn + (Db.L * Dd.L))[39:0]`	

23	**MAC.US.I.X Da.h,Db.h,Dn**	**Single 16x16 bit unsigned by signed integer multiply-accumulate**
	`Dn = (Dn + (Da.H * Db.H))[39:0]`	
24	**MAC.US.I.X Da.l,Db.l,Dn**	**Single 16x16 bit unsigned by signed integer multiply-accumulate**
	`Dn = (Dn + (Da.L * Db.L))[39:0]`	
25	**MAC.UU.I.X Da.h,Db.h,Dn**	**Single 16x16 bit unsigned by unsigned integer multiply-accumulate**
	`Dn = (Dn + (Da.H * Db.H))[39:0]`	
26	**MAC.UU.I.X Da.l,Db.l,Dn**	**Single 16x16 bit unsigned by unsigned integer multiply-accumulate**
	`Dn = (Dn + (Da.L * Db.L))[39:0]`	
27	**MAC.UU.I.X -Da.l,Db.l,Dn**	**Single 16x16 bit unsigned by unsigned integer multiply-accumulate with negate**
	`Dn = (Dn - (Da.L * Db.L))[39:0]`	
28	**MAC.R.X Da.h,Db.h,Dn**	**Single 16x16 bit fractional multiply-accumulate with rounding**
	`Dn = (Dn + (((Da.H * Db.H) << 1) + 0x8000))[39:0]`	
29	**MAC.R.X Da.l,Db.h,Dn**	**Single 16x16 bit fractional multiply-accumulate with rounding**
	`Dn = (Dn + (((Da.L * Db.H) << 1) + 0x8000))[39:0]`	

Table continues on the next page...

#	Syntax	Description
30	**MAC.R.X Da.l,Db.l,Dn**	**Single 16x16 bit fractional multiply-accumulate with rounding**
	`Dn = (Dn + (((Da.L * Db.L) << 1) + 0x8000))[39:0]`	
31	**MAC.S.2X Da,Db,Dm:Dn**	**SIMD2 fractional multiply-accumulate, multiplying high and low portions with saturation**

	`Dm = SAT32((Dm + ((Da.H * Db.H) << 1)))`
	`Dn = SAT32((Dn + ((Da.L * Db.L) << 1)))`	

32	**MAC.S.X Da.h,Db.h,Dn**	**Single 16x16 bit fractional multiply-accumulate with saturation**
	`Dn = SAT32((Dn + ((Da.H * Db.H) << 1)))`	
33	**MAC.S.X Da.h,Db.l,Dn**	**Single 16x16 bit fractional multiply-accumulate with saturation**
	`Dn = SAT32((Dn + ((Da.H * Db.L) << 1)))`	
34	**MAC.S.X Da.l,Db.l,Dn**	**Single 16x16 bit fractional multiply-accumulate with saturation**
	`Dn = SAT32((Dn + ((Da.L * Db.L) << 1)))`	
35	**MAC.S.X #s16,Da.h,Dn**	**Single 16x16 bit fractional multiply-accumulate using immediate value with saturation**
	`Dn = SAT32((Dn + ((Da.H * s16_t2) << 1)))`	
36	**MAC.S.X #s16,Da.l,Dn**	**Single 16x16 bit fractional multiply-accumulate using immediate value with saturation**
	`Dn = SAT32((Dn + ((Da.L * s16_t2) << 1)))`	
37	**MAC.H.S.2X Da,Db,Dm:Dn**	**SIMD2 fractional multiply-accumulate, high portion by two portions with saturation**

	`Dm = SAT32((Dm + ((Da.H * Db.H) << 1)))`
	`Dn = SAT32((Dn + ((Da.H * Db.L) << 1)))`	

38	**MAC.HH.S.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional multiply-accumulate, high portions by high portions with saturation**

	`Dm = SAT32((Dm + ((Da.H * Dc.H) << 1)))`
	`Dn = SAT32((Dn + ((Db.H * Dd.H) << 1)))`	

39	**MAC.L.S.2X Da,Db,Dm:Dn**	**SIMD2 fractional multiply-accumulate, low portion by two portions with saturation**

	`Dm = SAT32((Dm + ((Da.L * Db.H) << 1)))`
	`Dn = SAT32((Dn + ((Da.L * Db.L) << 1)))`	

40	**MAC.LL.S.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional multiply-accumulate, low portions by low portions with saturation**

	`Dm = SAT32((Dm + ((Da.L * Dc.L) << 1)))`
	`Dn = SAT32((Dn + ((Db.L * Dd.L) << 1)))`	

41	**MAC.SR.X Da.h,Db.h,Dn**	**Single 16x16 bit fractional multiply-accumulate with rounding and saturation**
	`Dn = SAT32(((Dn + ((Da.H * Db.H) << 1)) + 0x8000))`	

Table continues on the next page...

#	Syntax	Description
42	**MAC.SR.X Da.l,Db.h,Dn**	**Single 16x16 bit fractional multiply-accumulate with rounding and saturation**
	`Dn = SAT32(((Dn + ((Da.L * Db.H) << 1)) + 0x8000))`	
43	**MAC.SR.X Da.l,Db.l,Dn**	**Single 16x16 bit fractional multiply-accumulate with rounding and saturation**
	`Dn = SAT32(((Dn + ((Da.L * Db.L) << 1)) + 0x8000))`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	
s16	$-2^{15} \le s16 < 2^{15}$	

Affected By Instructions

Field	Relevant Variants
SR.SAT	31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	11, 12, 13, 14, 16, 17, 19, 20, 21, 23, 25
Encoding length	32-bits in 64	1, 2, 3, 4, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 37, 38, 39, 40, 41, 42, 43
	64-bits	5, 6, 35, 36
Pipeline behavior	dalu_MPY_Acc	All

MAC32.nL 32x32 bit Multiply-Accumulate into 32-bit Result (DALU)

General Description

Performs single 32x32 bit C-like integer multiply, accumulating the low 32-bit of the 64-bit result in 32-bit result.

Flag Options

Flag	Position	Description
I	flg2	Integer arithmetic
L	flg2	32-bit operation

Instruction Variants

#	Syntax	Description
1	**MAC32.IL.L Da,Db,Dn**	**Single 32x32 bit integer multiply-accumulate**

	`Dn = Dn.M + (((S64)Da.M * (S64)Db.M)[31:0])`
	`Dn.E = 0`	

2	**MAC32.IL.L -Da,Db,Dn**	**Single 32x32 bit integer multiply-accumulate with negate**

	`Dn = Dn.M - (((S64)Da.M * (S64)Db.M)[31:0])`
	`Dn.E = 0`	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Db	D0-D63
Dn	D0-D63

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MAC32.nX 32x32 bit Multiply-Accumulate into 40-bit Result (DALU)

General Description

Performs single 32x32 bit multiply, accumulating the high 32-bit of the 64-bit result in 40-bit result.

Flag Options

Flag	Position	Description
I	flg2	Integer arithmetic
R	flg2	Round multiplication result
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MAC32.IR.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 32x32 bit integer multiply-accumulate with rounding**

	`Dm = (Dm + (((S64)(((S64)Da.M * (S64)Dc.M) + 0x80000000)) >> 32))[39:0]`
	`Dn = (Dn + (((S64)(((S64)Db.M * (S64)Dd.M) + 0x80000000)) >> 32))[39:0]`	

2	**MAC32.IR.X Da,Db,Dn**	**Single 32x32 bit integer multiply-accumulate with rounding**
	`Dn = (Dn + ((((S64)Da.M * (S64)Db.M) + 0x80000000) >> 32))[39:0]`	
3	**MAC32.R.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 32x32 bit fractional multiply-accumulate with rounding**

	`Dm = (Dm + ((((Da.M * Dc.M) << 1) + 0x80000000) >> 32))[39:0]`
	`Dn = (Dn + ((((Db.M * Dd.M) << 1) + 0x80000000) >> 32))[39:0]`	

4	**MAC32.R.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - 32x32 bit fractional multiply-accumulate with rounding**

	`Dm = (Dm + ((((Da.M * Dc.M) << 1) + 0x80000000) >> 32))[39:0]`
	`Dn = (Dn + ((((Da.M * Dd.M) << 1) + 0x80000000) >> 32))[39:0]`	

5	**MAC32.R.2X -Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 32x32 bit fractional multiply-accumulate with negate and rounding**

	`Dm = (Dm + ((-(((S64)Da.M * (S64)Dc.M) << 1) + 0x80000000) >> 32))[39:0]`
	`Dn = (Dn + ((-(((S64)Db.M * (S64)Dd.M) << 1) + 0x80000000) >> 32))[39:0]`	

6	**MAC32.R.X Da,Db,Dn**	**Single 32x32 bit fractional multiply-accumulate with rounding**
	`Dn = (Dn + (((((Da.M * Db.M)) << 1) + 0x80000000) >> 32))[39:0]`	
7	**MAC32.SR.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 32x32 bit fractional multiply-accumulate with rounding and saturation**

	`Dm = SAT32((Dm + ((((Da.M * Dc.M) << 1) + 0x80000000) >> 32)))`
	`Dn = SAT32((Dn + ((((Db.M * Dd.M) << 1) + 0x80000000) >> 32)))`	

Table continues on the next page...

#	Syntax	Description
8	**MAC32.SR.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - 32x32 bit fractional multiply-accumulate with rounding and saturation**

	`Dm = SAT32((Dm + ((((Da.M * Dc.M) << 1) + 0x80000000) >> 32)))`
	`Dn = SAT32((Dn + ((((Da.M * Dd.M) << 1) + 0x80000000) >> 32)))`	

9	**MAC32.SR.X Da,Db,Dn**	**Single 32x32 bit fractional multiply-accumulate with rounding and saturation**
	`Dn = SAT32((Dn + ((((Da.M * Db.M) << 1) + 0x80000000) >> 32)))`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
SR.SAT	7, 8, 9

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 5
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACCX.nT 16x16 bit Complex Multiply-Accumulate into 20-bit Result (DALU)

General Description

Performs single/SIMD2 16x16 bit complex multiply-accumulate into 20-bit results with rounding.
Each complex number is packed in a single register. Variants support integer or fractional multiplication.

Flag Options

Flag	Position	Description
C	flg1	Complex Conjugate
I	flg2	Integer arithmetic
R	flg2	Round multiplication result

Instruction Variants

#	Syntax	Description
1	**MACCX.IR.2T Da,Db,Dn**	**Complex integer multiply-accumulate with rounding**

	`Dn.WH = (Dn.WH + ((((Da.H * Db.H) - (Da.L * Db.L)) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + ((((Da.H * Db.L) + (Da.L * Db.H)) + 0x8000) >> 16))[19:0]`	

2	**MACCX.IR.2T -Da,Db,Dn**	**Complex integer multiply-accumulate with negate and rounding**

	`Dn.WH = (Dn.WH + (((-((Da.H * Db.H) - (Da.L * Db.L))) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + (((-((Da.H * Db.L) + (Da.L * Db.H))) + 0x8000) >> 16))[19:0]`	

3	**MACCX.IR.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 Complex integer multiply-accumulate with rounding**

	`Dm.WH = (Dm.WH + ((((Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16))[19:0]`
	`Dm.WL = (Dm.WL + ((((Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16))[19:0]`
	`Dn.WH = (Dn.WH + ((((Db.H * Dd.H) - (Db.L * Dd.L)) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + ((((Db.H * Dd.L) + (Db.L * Dd.H)) + 0x8000) >> 16))[19:0]`	

4	**MACCX.IR.4T Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - Complex integer multiply-accumulate with rounding**

	`Dm.WH = (Dm.WH + ((((Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16))[19:0]`
	`Dm.WL = (Dm.WL + ((((Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16))[19:0]`
	`Dn.WH = (Dn.WH + ((((Da.H * Dd.H) - (Da.L * Dd.L)) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + ((((Da.H * Dd.L) + (Da.L * Dd.H)) + 0x8000) >> 16))[19:0]`	

5	**MACCX.IR.4T -Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 Complex integer multiply-accumulate with negate and rounding**

	`Dm.WH = (Dm.WH + (((-(Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16))[19:0]`
	`Dm.WL = (Dm.WL + (((-(Da.H * Dc.L) - (Da.L * Dc.H)) + 0x8000) >> 16))[19:0]`
	`Dn.WH = (Dn.WH + (((-(Db.H * Dd.H) + (Db.L * Dd.L)) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + (((-(Db.H * Dd.L) - (Db.L * Dd.H)) + 0x8000) >> 16))[19:0]`	

Table continues on the next page...

#	Syntax	Description
6	**MACCX.IR.4T -Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - Complex integer multiply-accumulate with negate and rounding**

	Dm.WH = (Dm.WH + (((-(Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16))[19:0]
	Dm.WL = (Dm.WL + (((-(Da.H * Dc.L) - (Da.L * Dc.H)) + 0x8000) >> 16))[19:0]
	Dn.WH = (Dn.WH + (((-(Da.H * Dd.H) + (Da.L * Dd.L)) + 0x8000) >> 16))[19:0]
	Dn.WL = (Dn.WL + (((-(Da.H * Dd.L) - (Da.L * Dd.H)) + 0x8000) >> 16))[19:0]	

7	**MACCX.C.IR.2T Da,Db,Dn**	**Complex integer multiply by conjugate and accumulate with rounding**

	Dn.WH = (Dn.WH + ((((Da.H * Db.H) + (Da.L * Db.L)) + 0x8000) >> 16))[19:0]
	Dn.WL = (Dn.WL + ((((Da.H * Db.L) - (Da.L * Db.H)) + 0x8000) >> 16))[19:0]	

8	**MACCX.C.IR.2T -Da,Db,Dn**	**Complex integer multiply by conjugate and accumulate with negate and rounding**

	Dn.WH = (Dn.WH + (((-((Da.H * Db.H) + (Da.L * Db.L))) + 0x8000) >> 16))[19:0]
	Dn.WL = (Dn.WL + (((-((Da.H * Db.L) - (Da.L * Db.H))) + 0x8000) >> 16))[19:0]	

9	**MACCX.C.IR.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 Complex integer multiply by conjugate and accumulate with rounding**

	Dm.WH = (Dm.WH + ((((Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16))[19:0]
	Dm.WL = (Dm.WL + ((((Da.H * Dc.L) - (Da.L * Dc.H)) + 0x8000) >> 16))[19:0]
	Dn.WH = (Dn.WH + ((((Db.H * Dd.H) + (Db.L * Dd.L)) + 0x8000) >> 16))[19:0]
	Dn.WL = (Dn.WL + ((((Db.H * Dd.L) - (Db.L * Dd.H)) + 0x8000) >> 16))[19:0]	

10	**MACCX.C.IR.4T Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - Complex integer multiply by conjugate and accumulate with rounding**

	Dm.WH = (Dm.WH + ((((Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16))[19:0]
	Dm.WL = (Dm.WL + ((((Da.H * Dc.L) - (Da.L * Dc.H)) + 0x8000) >> 16))[19:0]
	Dn.WH = (Dn.WH + ((((Da.H * Dd.H) + (Da.L * Dd.L)) + 0x8000) >> 16))[19:0]
	Dn.WL = (Dn.WL + ((((Da.H * Dd.L) - (Da.L * Dd.H)) + 0x8000) >> 16))[19:0]	

11	**MACCX.C.IR.4T -Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 Complex integer multiply by conjugate and accumulate with negate and rounding**

	Dm.WH = (Dm.WH + (((-(Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16))[19:0]
	Dm.WL = (Dm.WL + (((-(Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16))[19:0]
	Dn.WH = (Dn.WH + (((-(Db.H * Dd.H) - (Db.L * Dd.L)) + 0x8000) >> 16))[19:0]
	Dn.WL = (Dn.WL + (((-(Db.H * Dd.L) + (Db.L * Dd.H)) + 0x8000) >> 16))[19:0]	

12	**MACCX.C.IR.4T -Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - Complex integer multiply by conjugate and accumulate with negate and rounding**

	Dm.WH = (Dm.WH + (((-(Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16))[19:0]
	Dm.WL = (Dm.WL + (((-(Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16))[19:0]
	Dn.WH = (Dn.WH + (((-(Da.H * Dd.H) - (Da.L * Dd.L)) + 0x8000) >> 16))[19:0]
	Dn.WL = (Dn.WL + (((-(Da.H * Dd.L) + (Da.L * Dd.H)) + 0x8000) >> 16))[19:0]	

13	**MACCX.R.2T Da,Db,Dn**	**Complex fractional multiply-accumulate with rounding**

	Dn.WH = (Dn.WH + (((((Da.H * Db.H) - (Da.L * Db.L)) << 1) + 0x8000) >> 16))[19:0]
	Dn.WL = (Dn.WL + (((((Da.H * Db.L) + (Da.L * Db.H)) << 1) + 0x8000) >> 16))[19:0]	

Table continues on the next page...

#	Syntax	Description
14	**MACCX.R.2T -Da,Db,Dn**	**Complex fractional multiply-accumulate with negate and rounding**

	`Dn.WH = (Dn.WH + (((-(((Da.H * Db.H) - (Da.L * Db.L)) << 1)) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + (((-(((Da.H * Db.L) + (Da.L * Db.H)) << 1)) + 0x8000) >> 16))[19:0]`	

15	**MACCX.R.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 Complex fractional multiply-accumulate with rounding**

	`Dm.WH = (Dm.WH + (((((Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[19:0]`
	`Dm.WL = (Dm.WL + (((((Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16))[19:0]`
	`Dn.WH = (Dn.WH + (((((Db.H * Dd.H) - (Db.L * Dd.L)) << 1) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + (((((Db.H * Dd.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16))[19:0]`	

16	**MACCX.R.4T Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - Complex fractional multiply-accumulate with rounding**

	`Dm.WH = (Dm.WH + (((((Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[19:0]`
	`Dm.WL = (Dm.WL + (((((Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16))[19:0]`
	`Dn.WH = (Dn.WH + (((((Da.H * Dd.H) - (Da.L * Dd.L)) << 1) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + (((((Da.H * Dd.L) + (Da.L * Dd.H)) << 1) + 0x8000) >> 16))[19:0]`	

17	**MACCX.R.4T -Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 Complex fractional multiply-accumulate with negate and rounding**

	`Dm.WH = (Dm.WH + ((-(((Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[19:0]`
	`Dm.WL = (Dm.WL + ((-(((Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16))[19:0]`
	`Dn.WH = (Dn.WH + ((-(((Db.H * Dd.H) - (Db.L * Dd.L)) << 1) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + ((-(((Db.H * Dd.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16))[19:0]`	

18	**MACCX.R.4T -Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - Complex fractional multiply-accumulate with negate and rounding**

	`Dm.WH = (Dm.WH + ((-(((Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[19:0]`
	`Dm.WL = (Dm.WL + ((-(((Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16))[19:0]`
	`Dn.WH = (Dn.WH + ((-(((Da.H * Dd.H) - (Da.L * Dd.L)) << 1) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + ((-(((Da.H * Dd.L) + (Da.L * Dd.H)) << 1) + 0x8000) >> 16))[19:0]`	

19	**MACCX.C.R.2T Da,Db,Dn**	**Complex fractional multiply by conjugate and accumulate with rounding**

	`Dn.WH = ((Dn.WH) + (((((Da.H * Db.H) + (Da.L * Db.L)) << 1) + 0x8000) >> 16))[19:0]`
	`Dn.WL = ((Dn.WL) + (((((Da.H * Db.L) - (Da.L * Db.H)) << 1) + 0x8000) >> 16))[19:0]`	

20	**MACCX.C.R.2T -Da,Db,Dn**	**Complex fractional multiply by conjugate and accumulate with negate and rounding**

	`Dn.WH = ((Dn.WH) + (((-(((Da.H * Db.H) + (Da.L * Db.L)) << 1)) + 0x8000) >> 16))[19:0]`
	`Dn.WL = ((Dn.WL) + (((-(((Da.H * Db.L) - (Da.L * Db.H)) << 1)) + 0x8000) >> 16))[19:0]`	

21	**MACCX.C.R.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 Complex fractional multiply by conjugate and accumulate with rounding**

	`Dm.WH = (Dm.WH + (((((Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[19:0]`
	`Dm.WL = (Dm.WL + (((((Da.H * Dc.L) - (Da.L * Dc.H)) << 1) + 0x8000) >> 16))[19:0]`
	`Dn.WH = (Dn.WH + (((((Db.H * Dd.H) + (Db.L * Dd.L)) << 1) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + (((((Db.H * Dd.L) - (Db.L * Dd.H)) << 1) + 0x8000) >> 16))[19:0]`	

Table continues on the next page...

#	Syntax	Description
22	**MACCX.C.R.4T Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - Complex fractional multiply by conjugate and accumulate with rounding**

```
Dm.WH = (Dm.WH + (((((Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[19:0]
Dm.WL = (Dm.WL + (((((Da.H * Dc.L) - (Da.L * Dc.H)) << 1) + 0x8000) >> 16))[19:0]
Dn.WH = (Dn.WH + (((((Da.H * Dd.H) + (Da.L * Dd.L)) << 1) + 0x8000) >> 16))[19:0]
Dn.WL = (Dn.WL + (((((Da.H * Dd.L) - (Da.L * Dd.H)) << 1) + 0x8000) >> 16))[19:0]
```

23	**MACCX.C.R.4T -Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 Complex fractional multiply by conjugate and accumulate with negate and rounding**

```
Dm.WH = (Dm.WH + ((-(((Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[19:0]
Dm.WL = (Dm.WL + ((-(((Da.H * Dc.L) - (Da.L * Dc.H)) << 1) + 0x8000) >> 16))[19:0]
Dn.WH = (Dn.WH + ((-(((Db.H * Dd.H) + (Db.L * Dd.L)) << 1) + 0x8000) >> 16))[19:0]
Dn.WL = (Dn.WL + ((-(((Db.H * Dd.L) - (Db.L * Dd.H)) << 1) + 0x8000) >> 16))[19:0]
```

24	**MACCX.C.R.4T -Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - Complex fractional multiply by conjugate and accumulate with negate and rounding**

```
Dm.WH = (Dm.WH + ((-(((Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[19:0]
Dm.WL = (Dm.WL + ((-(((Da.H * Dc.L) - (Da.L * Dc.H)) << 1) + 0x8000) >> 16))[19:0]
Dn.WH = (Dn.WH + ((-(((Da.H * Dd.H) + (Da.L * Dd.L)) << 1) + 0x8000) >> 16))[19:0]
Dn.WL = (Dn.WL + ((-(((Da.H * Dd.L) - (Da.L * Dd.H)) << 1) + 0x8000) >> 16))[19:0]
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \leq n \leq 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \leq n \leq 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \leq n \leq 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 4, 5, 6, 7, 8, 10, 11, 12, 14, 18, 20, 24
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACCX.nW 16x16 bit Complex Multiply-Accumulate into 16-bit Result With Saturation (DALU)

General Description

Performs single/SIMD2 16x16 bit complex multiply-accumulate into 16-bit results with rounding and saturation. Each complex number is packed in a single register. Variants support integer or fractional multiplication.

Flag Options

Flag	Position	Description
C	flg1	Complex Conjugate
I	flg2	Integer arithmetic
R	flg2	Round multiplication result
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MACCX.ISR.2W Da,Db,Dn**	**Complex integer multiply-accumulate with rounding and saturation**

	`Dn.H = SAT16 ((Dn.H + ((((Da.H * Db.H) - (Da.L * Db.L)) + 0x8000) >> 16)))[15:0]`
	`Dn.L = SAT16 ((Dn.L + ((((Da.H * Db.L) + (Da.L * Db.H)) + 0x8000) >> 16)))[15:0]`
	`Dn.E = 0`	

2	**MACCX.ISR.2W -Da,Db,Dn**	**Complex integer multiply-accumulate with negate, rounding and saturation**

	`Dn.H = SAT16 ((Dn.H + (((-((Da.H * Db.H) - (Da.L * Db.L))) + 0x8000) >> 16)))[15:0]`
	`Dn.L = SAT16 ((Dn.L + (((-((Da.H * Db.L) + (Da.L * Db.H))) + 0x8000) >> 16)))[15:0]`
	`Dn.E = 0`	

3	**MACCX.ISR.4W Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 Complex integer multiply-accumulate with rounding and saturation**

	`Dm.H = SAT16 ((Dm.H + ((((Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16)))[15:0]`
	`Dm.L = SAT16 ((Dm.L + ((((Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16)))[15:0]`
	`Dm.E = 0`
	`Dn.H = SAT16 ((Dn.H + ((((Db.H * Dd.H) - (Db.L * Dd.L)) + 0x8000) >> 16)))[15:0]`
	`Dn.L = SAT16 ((Dn.L + ((((Db.H * Dd.L) + (Db.L * Dd.H)) + 0x8000) >> 16)))[15:0]`
	`Dn.E = 0`	

Table continues on the next page...

#	Syntax	Description
4	**MACCX.ISR.4W Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - Complex integer multiply-accumulate with rounding and saturation**

	Dm.H = SAT16 ((Dm.H + ((((Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16)))[15:0]
	Dm.L = SAT16 ((Dm.L + ((((Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16)))[15:0]
	Dm.E = 0
	Dn.H = SAT16 ((Dn.H + ((((Da.H * Dd.H) - (Da.L * Dd.L)) + 0x8000) >> 16)))[15:0]
	Dn.L = SAT16 ((Dn.L + ((((Da.H * Dd.L) + (Da.L * Dd.H)) + 0x8000) >> 16)))[15:0]
	Dn.E = 0	

5	**MACCX.ISR.4W -Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 Complex integer multiply-accumulate with negate, rounding and saturation**

	Dm.H = SAT16 ((Dm.H + ((-((Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16)))[15:0]
	Dm.L = SAT16 ((Dm.L + ((-((Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16)))[15:0]
	Dm.E = 0
	Dn.H = SAT16 ((Dn.H + ((-((Db.H * Dd.H) - (Db.L * Dd.L)) + 0x8000) >> 16)))[15:0]
	Dn.L = SAT16 ((Dn.L + ((-((Db.H * Dd.L) + (Db.L * Dd.H)) + 0x8000) >> 16)))[15:0]
	Dn.E = 0	

6	**MACCX.ISR.4W -Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - Complex integer multiply-accumulate with negate, rounding and saturation**

	Dm.H = SAT16 ((Dm.H + ((-((Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16)))[15:0]
	Dm.L = SAT16 ((Dm.L + ((-((Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16)))[15:0]
	Dm.E = 0
	Dn.H = SAT16 ((Dn.H + ((-((Da.H * Dd.H) - (Da.L * Dd.L)) + 0x8000) >> 16)))[15:0]
	Dn.L = SAT16 ((Dn.L + ((-((Da.H * Dd.L) + (Da.L * Dd.H)) + 0x8000) >> 16)))[15:0]
	Dn.E = 0	

7	**MACCX.C.ISR.2W Da,Db,Dn**	**Complex integer multiply by conjugate and accumulate with rounding and saturation**

	Dn.H = SAT16 ((Dn.H + ((((Da.H * Db.H) + (Da.L * Db.L)) + 0x8000) >> 16)))[15:0]
	Dn.L = SAT16 ((Dn.L + ((((Da.H * Db.L) - (Da.L * Db.H)) + 0x8000) >> 16)))[15:0]
	Dn.E = 0	

8	**MACCX.C.ISR.2W -Da,Db,Dn**	**Complex integer multiply by conjugate and accumulate with negate, rounding and saturation**

	Dn.H = SAT16 ((Dn.H + (((-((Da.H * Db.H) + (Da.L * Db.L))) + 0x8000) >> 16)))[15:0]
	Dn.L = SAT16 ((Dn.L + (((-((Da.H * Db.L) - (Da.L * Db.H))) + 0x8000) >> 16)))[15:0]
	Dn.E = 0	

9	**MACCX.C.ISR.4W Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 Complex integer multiply by conjugate and accumulate with rounding and saturation**

	Dm.WH = (U20)SAT16 ((Dm.H + ((((Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16)))[15:0]
	Dm.WL = (U20)SAT16 ((Dm.L + ((((Da.H * Dc.L) - (Da.L * Dc.H)) + 0x8000) >> 16)))[15:0]
	Dn.WH = (U20)SAT16 ((Dn.H + ((((Db.H * Dd.H) + (Db.L * Dd.L)) + 0x8000) >> 16)))[15:0]
	Dn.WL = (U20)SAT16 ((Dn.L + ((((Db.H * Dd.L) - (Db.L * Dd.H)) + 0x8000) >> 16)))[15:0]	

10	**MACCX.C.ISR.4W Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - Complex integer multiply by conjugate and accumulate with rounding and saturation**

	Dm.WH = (U20)SAT16 ((Dm.H + ((((Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16)))[15:0]
	Dm.WL = (U20)SAT16 ((Dm.L + ((((Da.H * Dc.L) - (Da.L * Dc.H)) + 0x8000) >> 16)))[15:0]
	Dn.WH = (U20)SAT16 ((Dn.H + ((((Da.H * Dd.H) + (Da.L * Dd.L)) + 0x8000) >> 16)))[15:0]
	Dn.WL = (U20)SAT16 ((Dn.L + ((((Da.H * Dd.L) - (Da.L * Dd.H)) + 0x8000) >> 16)))[15:0]	

11	**MACCX.C.ISR.4W -Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 Complex integer multiply by conjugate and accumulate with negate, rounding and saturation**

Table continues on the next page...

#	Syntax	Description

	`Dm.WH = (U20)SAT16 ((Dm.H + ((-((Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16)))[15:0]`
	`Dm.WL = (U20)SAT16 ((Dm.L + ((-((Da.H * Dc.L) - (Da.L * Dc.H)) + 0x8000) >> 16)))[15:0]`
	`Dn.WH = (U20)SAT16 ((Dn.H + ((-((Db.H * Dd.H) + (Db.L * Dd.L)) + 0x8000) >> 16)))[15:0]`
	`Dn.WL = (U20)SAT16 ((Dn.L + ((-((Db.H * Dd.L) - (Db.L * Dd.H)) + 0x8000) >> 16)))[15:0]`	

12	MACCX.C.ISR.4W -Da,Dc:Dd,Dm:Dn	SIMD2 by one value - Complex integer multiply by conjugate and accumulate with negate, rounding and saturation

	`Dm.WH = (U20)SAT16 ((Dm.H + ((-((Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16)))[15:0]`
	`Dm.WL = (U20)SAT16 ((Dm.L + ((-((Da.H * Dc.L) - (Da.L * Dc.H)) + 0x8000) >> 16)))[15:0]`
	`Dn.WH = (U20)SAT16 ((Dn.H + ((-((Da.H * Dd.H) + (Da.L * Dd.L)) + 0x8000) >> 16)))[15:0]`
	`Dn.WL = (U20)SAT16 ((Dn.L + ((-((Da.H * Dd.L) - (Da.L * Dd.H)) + 0x8000) >> 16)))[15:0]`	

13	MACCX.SR.2W Da,Db,Dn	Complex fractional multiply-accumulate with rounding and saturation

	`Dn.H = SAT16 ((Dn.H + (((((Da.H * Db.H) - (Da.L * Db.L)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.L = SAT16 ((Dn.L + (((((Da.H * Db.L) + (Da.L * Db.H)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.E = 0`	

14	MACCX.SR.2W -Da,Db,Dn	Complex fractional multiply-accumulate with negate, rounding and saturation

	`Dn.H = SAT16 ((Dn.H + (((-(((Da.H * Db.H) - (Da.L * Db.L)) << 1)) + 0x8000) >> 16)))[15:0]`
	`Dn.L = SAT16 ((Dn.L + (((-(((Da.H * Db.L) + (Da.L * Db.H)) << 1)) + 0x8000) >> 16)))[15:0]`
	`Dn.E = 0`	

15	MACCX.SR.4W Da:Db,Dc:Dd,Dm:Dn	SIMD2 Complex fractional multiply-accumulate with rounding and saturation

	`Dm.H = SAT16 ((Dm.H + (((((Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dm.L = SAT16 ((Dm.L + (((((Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dm.E = 0`
	`Dn.H = SAT16 ((Dn.H + (((((Db.H * Dd.H) - (Db.L * Dd.L)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.L = SAT16 ((Dn.L + (((((Db.H * Dd.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.E = 0`	

16	MACCX.SR.4W Da,Dc:Dd,Dm:Dn	SIMD2 by one value - Complex fractional multiply-accumulate with rounding and saturation

	`Dm.H = SAT16 ((Dm.H + (((((Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dm.L = SAT16 ((Dm.L + (((((Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dm.E = 0`
	`Dn.H = SAT16 ((Dn.H + (((((Da.H * Dd.H) - (Da.L * Dd.L)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.L = SAT16 ((Dn.L + (((((Da.H * Dd.L) + (Da.L * Dd.H)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.E = 0`	

17	MACCX.SR.4W -Da:Db,Dc:Dd,Dm:Dn	SIMD2 Complex fractional multiply-accumulate with negate, rounding and saturation

	`Dm.H = SAT16 ((Dm.H + ((-(((Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dm.L = SAT16 ((Dm.L + ((-(((Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dm.E = 0`
	`Dn.H = SAT16 ((Dn.H + ((-(((Db.H * Dd.H) - (Db.L * Dd.L)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.L = SAT16 ((Dn.L + ((-(((Db.H * Dd.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.E = 0`	

Table continues on the next page...

#	Syntax	Description
18	**MACCX.SR.4W -Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - Complex fractional multiply-accumulate with negate, rounding and saturation**

	Dm.H = SAT16 ((Dm.H + ((-(((Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16)))[15:0]
	Dm.L = SAT16 ((Dm.L + ((-(((Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16)))[15:0]
	Dm.E = 0
	Dn.H = SAT16 ((Dn.H + ((-(((Da.H * Dd.H) - (Da.L * Dd.L)) << 1) + 0x8000) >> 16)))[15:0]
	Dn.L = SAT16 ((Dn.L + ((-(((Da.H * Dd.L) + (Da.L * Dd.H)) << 1) + 0x8000) >> 16)))[15:0]
	Dn.E = 0	

19	**MACCX.C.SR.2W Da,Db,Dn**	**Complex fractional multiply by conjugate and accumulate with rounding and saturation**

	Dn.H = SAT16 ((Dn.H + (((((Da.H * Db.H) + (Da.L * Db.L)) << 1) + 0x8000) >> 16)))[15:0]
	Dn.L = SAT16 ((Dn.L + (((((Da.H * Db.L) - (Da.L * Db.H)) << 1) + 0x8000) >> 16)))[15:0]
	Dn.E = 0	

20	**MACCX.C.SR.2W -Da,Db,Dn**	**Complex fractional multiply by conjugate and accumulate with negate, rounding and saturation**

	Dn.H = SAT16 ((Dn.H + (((-(((Da.H * Db.H) + (Da.L * Db.L)) << 1)) + 0x8000) >> 16)))[15:0]
	Dn.L = SAT16 ((Dn.L + (((-(((Da.H * Db.L) - (Da.L * Db.H)) << 1)) + 0x8000) >> 16)))[15:0]
	Dn.E = 0	

21	**MACCX.C.SR.4W Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 Complex fractional multiply by conjugate and accumulate with rounding and saturation**

	Dm.H = SAT16 ((Dm.H + (((((Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16)))[15:0]
	Dm.L = SAT16 ((Dm.L + (((((Da.H * Dc.L) - (Da.L * Dc.H)) << 1) + 0x8000) >> 16)))[15:0]
	Dm.E = 0
	Dn.H = SAT16 ((Dn.H + (((((Db.H * Dd.H) + (Db.L * Dd.L)) << 1) + 0x8000) >> 16)))[15:0]
	Dn.L = SAT16 ((Dn.L + (((((Db.H * Dd.L) - (Db.L * Dd.H)) << 1) + 0x8000) >> 16)))[15:0]
	Dn.E = 0	

22	**MACCX.C.SR.4W Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - Complex fractional multiply by conjugate and accumulate with rounding and saturation**

	Dm.H = SAT16 ((Dm.H + (((((Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16)))[15:0]
	Dm.L = SAT16 ((Dm.L + (((((Da.H * Dc.L) - (Da.L * Dc.H)) << 1) + 0x8000) >> 16)))[15:0]
	Dm.E = 0
	Dn.H = SAT16 ((Dn.H + (((((Da.H * Dd.H) + (Da.L * Dd.L)) << 1) + 0x8000) >> 16)))[15:0]
	Dn.L = SAT16 ((Dn.L + (((((Da.H * Dd.L) - (Da.L * Dd.H)) << 1) + 0x8000) >> 16)))[15:0]
	Dn.E = 0	

23	**MACCX.C.SR.4W -Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 Complex fractional multiply by conjugate and accumulate with negate, rounding and saturation**

Table continues on the next page...

#	Syntax	Description

	Dm.H = SAT16 ((Dm.H + ((-(((Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16))) [15:0]
	Dm.L = SAT16 ((Dm.L + ((-(((Da.H * Dc.L) - (Da.L * Dc.H)) << 1) + 0x8000) >> 16))) [15:0]
	Dm.E = 0
	Dn.H = SAT16 ((Dn.H + ((-(((Db.H * Dd.H) + (Db.L * Dd.L)) << 1) + 0x8000) >> 16))) [15:0]
	Dn.L = SAT16 ((Dn.L + ((-(((Db.H * Dd.L) - (Db.L * Dd.H)) << 1) + 0x8000) >> 16))) [15:0]
	Dn.E = 0	

24	**MACCX.C.SR.4W -Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - Complex fractional multiply by conjugate and accumulate with negate, rounding and saturation**

	Dm.H = SAT16 ((Dm.H + ((-(((Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16))) [15:0]
	Dm.L = SAT16 ((Dm.L + ((-(((Da.H * Dc.L) - (Da.L * Dc.H)) << 1) + 0x8000) >> 16))) [15:0]
	Dm.E = 0
	Dn.H = SAT16 ((Dn.H + ((-(((Da.H * Dd.H) + (Da.L * Dd.L)) << 1) + 0x8000) >> 16))) [15:0]
	Dn.L = SAT16 ((Dn.L + ((-(((Da.H * Dd.L) - (Da.L * Dd.H)) << 1) + 0x8000) >> 16))) [15:0]
	Dn.E = 0	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 14, 17, 18, 20, 23, 24
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACCX.nX 16x16 bit Complex Multiply-Accumulate into 40-bit Result (DALU)

General Description

Performs 16x16 bit complex multiply-accumulate into full precision 40-bit results.
Source complex numbers are packed in single registers, while destination complex numbers resides in a register pair.

Flag Options

Flag	Position	Description
C	flg1	Complex Conjugate
I	flg2	Integer arithmetic
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MACCX.2X Da,Db,Dm:Dn**	**Complex fractional multiply-accumulate**

	`Dm = (Dm + (((Da.H * Db.H) - (Da.L * Db.L)) << 1))[39:0]`
	`Dn = (Dn + (((Da.L * Db.H) + (Da.H * Db.L)) << 1))[39:0]`	

2	**MACCX.2X -Da,Db,Dm:Dn**	**Complex fractional multiply-accumulate with negate**

	`Dm = (Dm - (((Da.H * Db.H) - (Da.L * Db.L)) << 1))[39:0]`
	`Dn = (Dn - (((Da.L * Db.H) + (Da.H * Db.L)) << 1))[39:0]`	

3	**MACCX.C.2X Da,Db,Dm:Dn**	**Complex fractional multiply by conjugate and accumulate**

	`Dm = (Dm + (((Da.H * Db.H) + (Da.L * Db.L)) << 1))[39:0]`
	`Dn = (Dn + (((Da.H * Db.L) - (Da.L * Db.H)) << 1))[39:0]`	

4	**MACCX.C.2X -Da,Db,Dm:Dn**	**Complex fractional multiply by conjugate and accumulate with negate**

	`Dm = (Dm - (((Da.H * Db.H) + (Da.L * Db.L)) << 1))[39:0]`
	`Dn = (Dn - (((Da.H * Db.L) - (Da.L * Db.H)) << 1))[39:0]`	

5	**MACCX.I.2X Da,Db,Dm:Dn**	**Complex integer multiply-accumulate**

	`Dm = (Dm + ((Da.H * Db.H) - (Da.L * Db.L)))[39:0]`
	`Dn = (Dn + ((Da.L * Db.H) + (Da.H * Db.L)))[39:0]`	

6	**MACCX.I.2X -Da,Db,Dm:Dn**	**Complex integer multiply-accumulate with negate**

	`Dm = (Dm - ((Da.H * Db.H) - (Da.L * Db.L)))[39:0]`
	`Dn = (Dn - ((Da.L * Db.H) + (Da.H * Db.L)))[39:0]`	

7	**MACCX.C.I.2X Da,Db,Dm:Dn**	**Complex integer multiply by conjugate and accumulate**

Table continues on the next page...

#	Syntax	Description

	`Dm = (Dm + ((Da.H * Db.H) + (Da.L * Db.L)))[39:0]`
	`Dn = (Dn + ((Da.H * Db.L) - (Da.L * Db.H)))[39:0]`	

8	**MACCX.C.I.2X -Da,Db,Dm:Dn**	**Complex integer multiply by conjugate and accumulate with negate**

	`Dm = (Dm + ((-(Da.H * Db.H)) + (-(Da.L * Db.L))))[39:0]`
	`Dn = (Dn + ((-(Da.H * Db.L)) - (-(Da.L * Db.H))))[39:0]`	

9	**MACCX.IS.2X Da,Db,Dm:Dn**	**Complex integer multiply-accumulate with saturation**

	`Dm = SAT32((Dm + ((Da.H * Db.H) - (Da.L * Db.L))))`
	`Dn = SAT32((Dn + ((Da.L * Db.H) + (Da.H * Db.L))))`	

10	**MACCX.IS.2X -Da,Db,Dm:Dn**	**Complex integer multiply-accumulate with negate and saturation**

	`Dm = SAT32((Dm - ((Da.H * Db.H) - (Da.L * Db.L))))`
	`Dn = SAT32((Dn - ((Da.L * Db.H) + (Da.H * Db.L))))`	

11	**MACCX.C.IS.2X Da,Db,Dm:Dn**	**Complex integer multiply by conjugate and accumulate with saturation**

	`Dm = SAT32((Dm + ((Da.H * Db.H) + (Da.L * Db.L))))`
	`Dn = SAT32((Dn + ((Da.H * Db.L) - (Da.L * Db.H))))`	

12	**MACCX.C.IS.2X -Da,Db,Dm:Dn**	**Complex integer multiply by conjugate and accumulate with negate and saturation**

	`Dm = SAT32((Dm - ((Da.H * Db.H) + (Da.L * Db.L))))`
	`Dn = SAT32((Dn - ((Da.H * Db.L) - (Da.L * Db.H))))`	

13	**MACCX.S.2X Da,Db,Dm:Dn**	**Complex fractional multiply-accumulate with saturation**

	`Dm = SAT32((Dm + (((Da.H * Db.H) - (Da.L * Db.L)) << 1)))`
	`Dn = SAT32((Dn + (((Da.L * Db.H) + (Da.H * Db.L)) << 1)))`	

14	**MACCX.S.2X -Da,Db,Dm:Dn**	**Complex fractional multiply-accumulate with negate and saturation**

	`Dm = SAT32((Dm - (((Da.H * Db.H) - (Da.L * Db.L)) << 1)))`
	`Dn = SAT32((Dn - (((Da.L * Db.H) + (Da.H * Db.L)) << 1)))`	

15	**MACCX.C.S.2X Da,Db,Dm:Dn**	**Complex fractional multiply by conjugate and accumulate with saturation**

	`Dm = SAT32((Dm + (((Da.H * Db.H) + (Da.L * Db.L)) << 1)))`
	`Dn = SAT32((Dn + (((Da.H * Db.L) - (Da.L * Db.H)) << 1)))`	

16	**MACCX.C.S.2X -Da,Db,Dm:Dn**	**Complex fractional multiply by conjugate and accumulate with negate and saturation**

	`Dm = SAT32((Dm - (((Da.H * Db.H) + (Da.L * Db.L)) << 1)))`
	`Dn = SAT32((Dn - (((Da.H * Db.L) - (Da.L * Db.H)) << 1)))`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Db	D0-D63	
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$

Affected By Instructions

Field	Relevant Variants
SR.SAT	9, 10, 11, 12, 13, 14, 15, 16

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	6, 7, 8, 10, 11, 12, 14, 16
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACCXD.nT 16x16 bit Complex Dot-Product and Accumulate into 20-bit Result (DALU)

General Description

Performs two 16x16 bit complex multiplications, sum or subtract the products and accumulate into 20-bit results. Each complex number is packed in a single register.

Flag Options

Flag	Position	Description
C	flg1	Complex Conjugate
NN	flg1	Negative Negative
NP	flg1	Negative Positive
PN	flg1	Positive Negative
PP	flg1	Positive Positive
X	flg1	Cross between the arguments
I	flg2	Integer arithmetic
R	flg2	Round multiplication result

Instruction Variants

#	Syntax	Description
1	**MACCXD.CNN.IR.2T Da:Db,Dc:Dd,Dn**	**Complex integer dot product by conjugate and accumulate with rounding, both products are negated**

	`Dn.WH = (Dn.WH + ((((-(Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16) + (((-(Db.H *`
	`Dd.H) - (Db.L * Dd.L)) + 0x8000) >> 16)))[19:0]`
	`Dn.WL = (Dn.WL + ((((-(Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16) + (((-(Db.H *`
	`Dd.L) + (Db.L * Dd.H)) + 0x8000) >> 16)))[19:0]`	

2	**MACCXD.CNP.IR.2T Da:Db,Dc:Dd,Dn**	**Complex integer dot product by conjugate and accumulate with rounding, first product is negated**

	`Dn.WH = (Dn.WH + ((((-(Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16) + ((((Db.H *`
	`Dd.H) + (Db.L * Dd.L)) + 0x8000) >> 16)))[19:0]`
	`Dn.WL = (Dn.WL + ((((-(Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16) + ((((Db.H *`
	`Dd.L) - (Db.L * Dd.H)) + 0x8000) >> 16)))[19:0]`	

3	**MACCXD.CPN.IR.2T Da:Db,Dc:Dd,Dn**	**Complex integer dot product by conjugate and accumulate with rounding, second product is negated**

	`Dn.WH = (Dn.WH + (((((Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16) + (((-(Db.H *`
	`Dd.H) - (Db.L * Dd.L)) + 0x8000) >> 16)))[19:0]`
	`Dn.WL = (Dn.WL + (((((Da.H * Dc.L) - (Da.L * Dc.H)) + 0x8000) >> 16) + (((-(Db.H *`
	`Dd.L) + (Db.L * Dd.H)) + 0x8000) >> 16)))[19:0]`	

Table continues on the next page...

#	Syntax	Description
4	**MACCXD.CPP.IR.2T Da:Db,Dc:Dd,Dn**	**Complex integer dot product by conjugate and accumulate with rounding**

	`Dn.WH = (Dn.WH + (((((Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16) + ((((Db.H * Dd.H) + (Db.L * Dd.L)) + 0x8000) >> 16)))[19:0]`
	`Dn.WL = (Dn.WL + (((((Da.H * Dc.L) - (Da.L * Dc.H)) + 0x8000) >> 16) + ((((Db.H * Dd.L) - (Db.L * Dd.H)) + 0x8000) >> 16)))[19:0]`	

5	**MACCXD.NN.IR.2T Da:Db,Dc:Dd,Dn**	**Complex integer dot product and accumulate with rounding, both products are negated**

	`Dn.WH = (Dn.WH + ((((-(Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16) + (((-(Db.H * Dd.H) + (Db.L * Dd.L)) + 0x8000) >> 16)))[19:0]`
	`Dn.WL = (Dn.WL + ((((-(Da.H * Dc.L) - (Da.L * Dc.H)) + 0x8000) >> 16) + (((-(Db.H * Dd.L) - (Db.L * Dd.H)) + 0x8000) >> 16)))[19:0]`	

6	**MACCXD.NP.IR.2T Da:Db,Dc:Dd,Dn**	**Complex integer dot product and accumulate with rounding, first product is negated**

	`Dn.WH = (Dn.WH + ((((-(Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16) + ((((Db.H * Dd.H) - (Db.L * Dd.L)) + 0x8000) >> 16)))[19:0]`
	`Dn.WL = (Dn.WL + ((((-(Da.H * Dc.L) - (Da.L * Dc.H)) + 0x8000) >> 16) + ((((Db.H * Dd.L) + (Db.L * Dd.H)) + 0x8000) >> 16)))[19:0]`	

7	**MACCXD.PN.IR.2T Da:Db,Dc:Dd,Dn**	**Complex integer dot product and accumulate with rounding, second product is negated**

	`Dn.WH = (Dn.WH + (((((Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16) + (((-(Db.H * Dd.H) + (Db.L * Dd.L)) + 0x8000) >> 16)))[19:0]`
	`Dn.WL = (Dn.WL + (((((Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16) + (((-(Db.H * Dd.L) - (Db.L * Dd.H)) + 0x8000) >> 16)))[19:0]`	

8	**MACCXD.PP.IR.2T Da:Db,Dc:Dd,Dn**	**Complex integer dot product and accumulate with rounding**

	`Dn.WH = (Dn.WH + (((((Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16) + ((((Db.H * Dd.H) - (Db.L * Dd.L)) + 0x8000) >> 16)))[19:0]`
	`Dn.WL = (Dn.WL + (((((Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16) + ((((Db.H * Dd.L) + (Db.L * Dd.H)) + 0x8000) >> 16)))[19:0]`	

9	**MACCXD.PPX.IR.2T Da:Db,Dc:Dd,Dn**	**Complex integer dot product and accumulate with rounding, swapped operands**

	`Dn.WH = (Dn.WH + (((((Da.H * Dd.H) - (Da.L * Dd.L)) + 0x8000) >> 16) + ((((Db.H * Dc.H) - (Db.L * Dc.L)) + 0x8000) >> 16)))[19:0]`
	`Dn.WL = (Dn.WL + (((((Da.H * Dd.L) + (Da.L * Dd.H)) + 0x8000) >> 16) + ((((Db.H * Dc.L) + (Db.L * Dc.H)) + 0x8000) >> 16)))[19:0]`	

10	**MACCXD.CNN.R.2T Da:Db,Dc:Dd,Dn**	**Complex fractional dot product by conjugate and accumulate with rounding, both products are negated**

	`Dn.WH = (Dn.WH + (((((-(Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16) + ((((-(Db.H * Dd.H) - (Db.L * Dd.L)) << 1) + 0x8000) >> 16)))[19:0]`
	`Dn.WL = (Dn.WL + (((((-(Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16) + ((((-(Db.H * Dd.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16)))[19:0]`	

11	**MACCXD.CNP.R.2T Da:Db,Dc:Dd,Dn**	**Complex fractional dot product by conjugate and accumulate with rounding, first product is negated**

	`Dn.WH = (Dn.WH + (((((-(Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16) + (((((Db.H * Dd.H) + (Db.L * Dd.L)) << 1) + 0x8000) >> 16)))[19:0]`
	`Dn.WL = (Dn.WL + (((((-(Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16) + (((((Db.H * Dd.L) - (Db.L * Dd.H)) << 1) + 0x8000) >> 16)))[19:0]`	

Table continues on the next page...

#	Syntax	Description
12	**MACCXD.CPN.R.2T Da:Db,Dc:Dd,Dn**	**Complex fractional dot product by conjugate and accumulate with rounding, second product is negated**

	Dn.WH = (Dn.WH + (((((Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16) + ((((-(Db.H * Dd.H) - (Db.L * Dd.L)) << 1) + 0x8000) >> 16)))[19:0]
	Dn.WL = (Dn.WL + (((((Da.H * Dc.L) - (Da.L * Dc.H)) << 1) + 0x8000) >> 16) + ((((-(Db.H * Dd.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16)))[19:0]	

13	**MACCXD.CPP.R.2T Da:Db,Dc:Dd,Dn**	**Complex fractional dot product by conjugate and accumulate with rounding**

	Dn.WH = (Dn.WH + (((((Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16) + ((((Db.H * Dd.H) + (Db.L * Dd.L)) << 1) + 0x8000) >> 16)))[19:0]
	Dn.WL = (Dn.WL + (((((Da.H * Dc.L) - (Da.L * Dc.H)) << 1) + 0x8000) >> 16) + ((((Db.H * Dd.L) - (Db.L * Dd.H)) << 1) + 0x8000) >> 16)))[19:0]	

14	**MACCXD.NN.R.2T Da:Db,Dc:Dd,Dn**	**Complex fractional dot product and accumulate with rounding, both products are negated**

	Dn.WH = (Dn.WH + (((((-(Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16) + ((((-(Db.H * Dd.H) + (Db.L * Dd.L)) << 1) + 0x8000) >> 16)))[19:0]
	Dn.WL = (Dn.WL + (((((-(Da.H * Dc.L) - (Da.L * Dc.H)) << 1) + 0x8000) >> 16) + ((((-(Db.H * Dd.L) - (Db.L * Dd.H)) << 1) + 0x8000) >> 16)))[19:0]	

15	**MACCXD.NP.R.2T Da:Db,Dc:Dd,Dn**	**Complex fractional dot product and accumulate with rounding, first product is negated**

	Dn.WH = (Dn.WH + (((((-(Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16) + ((((Db.H * Dd.H) - (Db.L * Dd.L)) << 1) + 0x8000) >> 16)))[19:0]
	Dn.WL = (Dn.WL + (((((-(Da.H * Dc.L) - (Da.L * Dc.H)) << 1) + 0x8000) >> 16) + ((((Db.H * Dd.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16)))[19:0]	

16	**MACCXD.PN.R.2T Da:Db,Dc:Dd,Dn**	**Complex fractional dot product and accumulate with rounding, second product is negated**

	Dn.WH = (Dn.WH + (((((Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16) + ((((-(Db.H * Dd.H) + (Db.L * Dd.L)) << 1) + 0x8000) >> 16)))[19:0]
	Dn.WL = (Dn.WL + (((((Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16) + ((((-(Db.H * Dd.L) - (Db.L * Dd.H)) << 1) + 0x8000) >> 16)))[19:0]	

17	**MACCXD.PP.R.2T Da:Db,Dc:Dd,Dn**	**Complex fractional dot product and accumulate with rounding**

	Dn.WH = (Dn.WH + (((((Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16) + ((((Db.H * Dd.H) - (Db.L * Dd.L)) << 1) + 0x8000) >> 16)))[19:0]
	Dn.WL = (Dn.WL + (((((Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16) + ((((Db.H * Dd.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16)))[19:0]	

18	**MACCXD.PPX.R.2T Da:Db,Dc:Dd,Dn**	**Complex fractional dot product and accumulate with rounding, swapped operands**

	Dn.WH = (Dn.WH + (((((Da.H * Dd.H) - (Da.L * Dd.L)) << 1) + 0x8000) >> 16) + ((((Db.H * Dc.H) - (Db.L * Dc.L)) << 1) + 0x8000) >> 16)))[19:0]
	Dn.WL = (Dn.WL + (((((Da.H * Dd.L) + (Da.L * Dd.H)) << 1) + 0x8000) >> 16) + ((((Db.H * Dc.L) + (Db.L * Dc.H)) << 1) + 0x8000) >> 16)))[19:0]	

Explicit Operands

Operand	Permitted Values	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3, 5, 6, 7, 9, 10, 11, 12, 14, 15, 16, 18
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACCXD.nW 16x16 bit Complex Dot-Product and Accumulate into 16-bit Result with Saturation (DALU)

General Description

Performs two 16x16 bit complex multiplications, sum or subtract the products and accumulate into 16-bit results. Each complex number is packed in a single register.

Flag Options

Flag	Position	Description
C	flg1	Complex Conjugate
NN	flg1	Negative Negative
NP	flg1	Negative Positive
PN	flg1	Positive Negative
PP	flg1	Positive Positive
X	flg1	Cross between the arguments
I	flg2	Integer arithmetic
R	flg2	Round multiplication result
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MACCXD.CNN.ISR.2W Da:Db,Dc:Dd,Dn**	**Complex integer dot product by conjugate and accumulate with rounding and saturation, both products are negated**

	`Dn.WH = (U20)SAT16 ((Dn.H + (((-((Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16) + ((-((Db.H * Dd.H) + (Db.L * Dd.L)) + 0x8000) >> 16))))`
	`Dn.WL = (U20)SAT16 ((Dn.L + (((-((Da.H * Dc.L) - (Da.L * Dc.H)) + 0x8000) >> 16) + ((-((Db.H * Dd.L) - (Db.L * Dd.H)) + 0x8000) >> 16))))`	

2	**MACCXD.CNP.ISR.2W Da:Db,Dc:Dd,Dn**	**Complex integer dot product by conjugate and accumulate with rounding and saturation, first product is negated**

	`Dn.WH = (U20)SAT16 ((Dn.H + (((-((Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16) + ((((Db.H * Dd.H) + (Db.L * Dd.L)) + 0x8000) >> 16))))`
	`Dn.WL = (U20)SAT16 ((Dn.L + (((-((Da.H * Dc.L) - (Da.L * Dc.H)) + 0x8000) >> 16) + ((((Db.H * Dd.L) - (Db.L * Dd.H)) + 0x8000) >> 16))))`	

3	**MACCXD.CPN.ISR.2W Da:Db,Dc:Dd,Dn**	**Complex integer dot product by conjugate and accumulate with rounding and saturation, second product is negated**

	`Dn.WH = (U20)SAT16 ((Dn.H + (((((Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16) + ((-((Db.H * Dd.H) + (Db.L * Dd.L)) + 0x8000) >> 16))))`
	`Dn.WL = (U20)SAT16 ((Dn.L + (((((Da.H * Dc.L) - (Da.L * Dc.H)) + 0x8000) >> 16) + ((-((Db.H * Dd.L) - (Db.L * Dd.H)) + 0x8000) >> 16))))`	

Table continues on the next page...

#	Syntax	Description
4	**MACCXD.CPP.ISR.2W Da:Db,Dc:Dd,Dn**	**Complex integer dot product by conjugate and accumulate with rounding and saturation**

	Dn.WH = (U20)SAT16 ((Dn.H + ((((Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16) +
	(((Db.H * Dd.H) + (Db.L * Dd.L)) + 0x8000) >> 16))))
	Dn.WL = (U20)SAT16 ((Dn.L + ((((Da.H * Dc.L) - (Da.L * Dc.H)) + 0x8000) >> 16) +
	(((Db.H * Dd.L) - (Db.L * Dd.H)) + 0x8000) >> 16))))	

5	**MACCXD.NN.ISR.2W Da:Db,Dc:Dd,Dn**	**Complex integer dot product and accumulate with rounding and saturation, both products are negated**

	Dn.WH = (U20)SAT16 ((Dn.H + (((-((Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16) + ((-
	((Db.H * Dd.H) - (Db.L * Dd.L)) + 0x8000) >> 16))))
	Dn.WL = (U20)SAT16 ((Dn.L + (((-((Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16) + ((-
	((Db.H * Dd.L) + (Db.L * Dd.H)) + 0x8000) >> 16))))	

6	**MACCXD.NP.ISR.2W Da:Db,Dc:Dd,Dn**	**Complex integer dot product and accumulate with rounding and saturation, first product is negated**

	Dn.WH = (U20)SAT16 ((Dn.H + (((-((Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16) +
	((((Db.H * Dd.H) - (Db.L * Dd.L)) + 0x8000) >> 16))))
	Dn.WL = (U20)SAT16 ((Dn.L + (((-((Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16) +
	((((Db.H * Dd.L) + (Db.L * Dd.H)) + 0x8000) >> 16))))	

7	**MACCXD.PN.ISR.2W Da:Db,Dc:Dd,Dn**	**Complex integer dot product and accumulate with rounding and saturation, second product is negated**

	Dn.WH = (U20)SAT16 ((Dn.H + ((((Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16) + ((-
	((Db.H * Dd.H) - (Db.L * Dd.L)) + 0x8000) >> 16))))
	Dn.WL = (U20)SAT16 ((Dn.L + ((((Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16) + ((-
	((Db.H * Dd.L) + (Db.L * Dd.H)) + 0x8000) >> 16))))	

8	**MACCXD.PP.ISR.2W Da:Db,Dc:Dd,Dn**	**Complex integer dot product and accumulate with rounding and saturation**

	Dn.WH = (U20)SAT16 ((Dn.H + ((((Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16) +
	((((Db.H * Dd.H) - (Db.L * Dd.L)) + 0x8000) >> 16))))
	Dn.WL = (U20)SAT16 ((Dn.L + ((((Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16) +
	((((Db.H * Dd.L) + (Db.L * Dd.H)) + 0x8000) >> 16))))	

9	**MACCXD.PPX.ISR.2W Da:Db,Dc:Dd,Dn**	**Complex integer dot product and accumulate with rounding and saturation, swapped operands**

	Dn.WH = (U20)SAT16 ((Dn.H + ((((Da.H * Dd.H) - (Da.L * Dd.L)) + 0x8000) >> 16) +
	((((Db.H * Dc.H) - (Db.L * Dc.L)) + 0x8000) >> 16))))
	Dn.WL = (U20)SAT16 ((Dn.L + ((((Da.H * Dd.L) + (Da.L * Dd.H)) + 0x8000) >> 16) +
	((((Db.H * Dc.L) + (Db.L * Dc.H)) + 0x8000) >> 16))))	

10	**MACCXD.CNN.SR.2W Da:Db,Dc:Dd,Dn**	**Complex fractional dot product by conjugate and accumulate with rounding and saturation, both products are negated**

	Dn.WH = (U20)SAT16 ((Dn.H + ((((-(((Da.H * Dc.H) + (Da.L * Dc.L)) << 1)) + 0x8000) >>
	16) + (((-(((Db.H * Dd.H) + (Db.L * Dd.L)) << 1)) + 0x8000) >> 16))))
	Dn.WL = (U20)SAT16 ((Dn.L + ((((-(((Da.H * Dc.L) - (Da.L * Dc.H)) << 1)) + 0x8000) >>
	16) + (((-(((Db.H * Dd.L) - (Db.L * Dd.H)) << 1)) + 0x8000) >> 16))))	

11	**MACCXD.CNP.SR.2W Da:Db,Dc:Dd,Dn**	**Complex fractional dot product by conjugate and accumulate with rounding and saturation, first product is negated**

Table continues on the next page...

#	Syntax	Description

	Dn.WH = (U20)SAT16 ((Dn.H + ((((-(((Da.H * Dc.H) + (Da.L * Dc.L)) << 1)) + 0x8000) >> 16) + ((((Db.H * Dd.H) + (Db.L * Dd.L)) << 1) + 0x8000) >> 16))))
	Dn.WL = (U20)SAT16 ((Dn.L + ((((-(((Da.H * Dc.L) - (Da.L * Dc.H)) << 1)) + 0x8000) >> 16) + ((((Db.H * Dd.L) - (Db.L * Dd.H)) << 1) + 0x8000) >> 16))))	

12	**MACCXD.CPN.SR.2W Da:Db,Dc:Dd,Dn**	**Complex fractional dot product by conjugate and accumulate with rounding and saturation, second product is negated**

	Dn.WH = (U20)SAT16 ((Dn.H + ((((((Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16) + (((-(((Db.H * Dd.H) + (Db.L * Dd.L)) << 1)) + 0x8000) >> 16))))
	Dn.WL = (U20)SAT16 ((Dn.L + ((((((Da.H * Dc.L) - (Da.L * Dc.H)) << 1) + 0x8000) >> 16) + (((-(((Db.H * Dd.L) - (Db.L * Dd.H)) << 1)) + 0x8000) >> 16))))	

13	**MACCXD.CPP.SR.2W Da:Db,Dc:Dd,Dn**	**Complex fractional dot product by conjugate and accumulate with rounding and saturation**

	Dn.WH = (U20)SAT16 ((Dn.H + ((((((Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16) + ((((Db.H * Dd.H) + (Db.L * Dd.L)) << 1) + 0x8000) >> 16))))
	Dn.WL = (U20)SAT16 ((Dn.L + ((((((Da.H * Dc.L) - (Da.L * Dc.H)) << 1) + 0x8000) >> 16) + ((((Db.H * Dd.L) - (Db.L * Dd.H)) << 1) + 0x8000) >> 16))))	

14	**MACCXD.NN.SR.2W Da:Db,Dc:Dd,Dn**	**Complex fractional dot product and accumulate with rounding and saturation, both products are negated**

	Dn.WH = (U20)SAT16 ((Dn.H + ((((-(((Da.H * Dc.H) - (Da.L * Dc.L)) << 1)) + 0x8000) >> 16) + (((-(((Db.H * Dd.H) - (Db.L * Dd.L)) << 1)) + 0x8000) >> 16))))
	Dn.WL = (U20)SAT16 ((Dn.L + ((((-(((Da.H * Dc.L) + (Da.L * Dc.H)) << 1)) + 0x8000) >> 16) + (((-(((Db.H * Dd.L) + (Db.L * Dd.H)) << 1)) + 0x8000) >> 16))))	

15	**MACCXD.NP.SR.2W Da:Db,Dc:Dd,Dn**	**Complex fractional dot product and accumulate with rounding and saturation, first product is negated**

	Dn.WH = (U20)SAT16 ((Dn.H + ((((-(((Da.H * Dc.H) - (Da.L * Dc.L)) << 1)) + 0x8000) >> 16) + ((((Db.H * Dd.H) - (Db.L * Dd.L)) << 1) + 0x8000) >> 16))))
	Dn.WL = (U20)SAT16 ((Dn.L + ((((-(((Da.H * Dc.L) + (Da.L * Dc.H)) << 1)) + 0x8000) >> 16) + ((((Db.H * Dd.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16))))	

16	**MACCXD.PN.SR.2W Da:Db,Dc:Dd,Dn**	**Complex fractional dot product and accumulate with rounding and saturation, second product is negated**

	Dn.WH = (U20)SAT16 ((Dn.H + ((((((Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16) + (((-(((Db.H * Dd.H) - (Db.L * Dd.L)) << 1)) + 0x8000) >> 16))))
	Dn.WL = (U20)SAT16 ((Dn.L + ((((((Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16) + (((-(((Db.H * Dd.L) + (Db.L * Dd.H)) << 1)) + 0x8000) >> 16))))	

17	**MACCXD.PP.SR.2W Da:Db,Dc:Dd,Dn**	**Complex fractional dot product and accumulate with rounding and saturation**

	Dn.WH = (U20)SAT16 ((Dn.H + ((((((Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16) + ((((Db.H * Dd.H) - (Db.L * Dd.L)) << 1) + 0x8000) >> 16))))
	Dn.WL = (U20)SAT16 ((Dn.L + ((((((Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16) + ((((Db.H * Dd.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16))))	

18	**MACCXD.PPX.SR.2W Da:Db,Dc:Dd,Dn**	**Complex fractional dot product and accumulate with rounding and saturation, swapped operands**

	Dn.WH = (U20)SAT16 ((Dn.H + ((((((Da.H * Dd.H) - (Da.L * Dd.L)) << 1) + 0x8000) >> 16) + ((((Db.H * Dc.H) - (Db.L * Dc.L)) << 1) + 0x8000) >> 16))))
	Dn.WL = (U20)SAT16 ((Dn.L + ((((((Da.H * Dd.L) + (Da.L * Dd.H)) << 1) + 0x8000) >> 16) + ((((Db.H * Dc.L) + (Db.L * Dc.H)) << 1) + 0x8000) >> 16))))	

Explicit Operands

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3, 5, 6, 9, 10, 11, 14, 15
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACCXD.nX 16x16 bit Complex Dot-Product and Accumulate into 40-bit Result (DALU)

General Description

Performs two 16x16 bit complex multiplications, sum or subtract the products and accumulate into 40-bit results. Source complex numbers are packed in single registers, while destination complex numbers resides in a register pair.

Flag Options

Flag	Position	Description
C	flg1	Complex Conjugate
NN	flg1	Negative Negative
NP	flg1	Negative Positive
PN	flg1	Positive Negative
PP	flg1	Positive Positive
X	flg1	Cross between the arguments
I	flg2	Integer arithmetic
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MACCXD.CNN.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate and accumulate, both products are negated**

	`Dm = (Dm - ((((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Db.H * Dd.H) + (Db.L * Dd.L))) << 1))[39:0]`
	`Dn = (Dn - ((((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Db.H * Dd.L) - (Db.L * Dd.H))) << 1))[39:0]`	

2	**MACCXD.CNN.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate and accumulate, both products are negated, single first operand**

	`Dm = (Dm - ((((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Da.H * Dd.H) + (Da.L * Dd.L))) << 1))[39:0]`
	`Dn = (Dn - ((((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Da.H * Dd.L) - (Da.L * Dd.H))) << 1))[39:0]`	

3	**MACCXD.CNP.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate and accumulate, first product is negated**

	`Dm = (Dm - ((((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Db.H * Dd.H) + (Db.L * Dd.L))) << 1))[39:0]`
	`Dn = (Dn - ((((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Db.H * Dd.L) - (Db.L * Dd.H))) << 1))[39:0]`	

Table continues on the next page...

#	Syntax	Description
4	**MACCXD.CNP.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate and accumulate, first product is negated, single first operand**

	Dm = (Dm - (((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Da.H * Dd.H) + (Da.L * Dd.L))) << 1)) [39:0]
	Dn = (Dn - (((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Da.H * Dd.L) - (Da.L * Dd.H))) << 1)) [39:0]	

5	**MACCXD.CPN.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate and accumulate, second product is negated**

	Dm = (Dm + (((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Db.H * Dd.H) + (Db.L * Dd.L))) << 1)) [39:0]
	Dn = (Dn + (((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Db.H * Dd.L) - (Db.L * Dd.H))) << 1)) [39:0]	

6	**MACCXD.CPN.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate and accumulate, second product is negated, single first operand**

	Dm = (Dm + (((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Da.H * Dd.H) + (Da.L * Dd.L))) << 1)) [39:0]
	Dn = (Dn + (((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Da.H * Dd.L) - (Da.L * Dd.H))) << 1)) [39:0]	

7	**MACCXD.CPP.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate and accumulate**

	Dm = (Dm + (((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Db.H * Dd.H) + (Db.L * Dd.L))) << 1)) [39:0]
	Dn = (Dn + (((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Db.H * Dd.L) - (Db.L * Dd.H))) << 1)) [39:0]	

8	**MACCXD.CPP.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate and accumulate, single first operand**

	Dm = (Dm + (((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Da.H * Dd.H) + (Da.L * Dd.L))) << 1)) [39:0]
	Dn = (Dn + (((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Da.H * Dd.L) - (Da.L * Dd.H))) << 1)) [39:0]	

9	**MACCXD.NN.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product and accumulate, both products are negated**

	Dm = (Dm - (((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Db.H * Dd.H) - (Db.L * Dd.L))) << 1)) [39:0]
	Dn = (Dn - (((Da.H * Dc.L) + (Da.L * Dc.H)) + ((Db.H * Dd.L) + (Db.L * Dd.H))) << 1)) [39:0]	

10	**MACCXD.NN.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product and accumulate, both products are negated, single first operand**

	Dm = (Dm - (((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Da.H * Dd.H) - (Da.L * Dd.L))) << 1)) [39:0]
	Dn = (Dn - (((Da.H * Dc.L) + (Da.L * Dc.H)) + ((Da.H * Dd.L) + (Da.L * Dd.H))) << 1)) [39:0]	

11	**MACCXD.NP.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product and accumulate, first product is negated**

	Dm = (Dm - (((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Db.H * Dd.H) - (Db.L * Dd.L))) << 1)) [39:0]
	Dn = (Dn - (((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Db.H * Dd.L) + (Db.L * Dd.H))) << 1)) [39:0]	

Table continues on the next page...

#	Syntax	Description
12	**MACCXD.NP.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product and accumulate, first product is negated, single first operand**

	`Dm = (Dm - (((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Da.H * Dd.H) - (Da.L * Dd.L))) << 1))[39:0]`
	`Dn = (Dn - (((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Da.H * Dd.L) + (Da.L * Dd.H))) << 1))[39:0]`	

13	**MACCXD.PN.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product and accumulate, second product is negated**

	`Dm = (Dm + (((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Db.H * Dd.H) - (Db.L * Dd.L))) << 1))[39:0]`
	`Dn = (Dn + (((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Db.H * Dd.L) + (Db.L * Dd.H))) << 1))[39:0]`	

14	**MACCXD.PN.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product and accumulate, second product is negated, single first operand**

	`Dm = (Dm + (((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Da.H * Dd.H) - (Da.L * Dd.L))) << 1))[39:0]`
	`Dn = (Dn + (((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Da.H * Dd.L) + (Da.L * Dd.H))) << 1))[39:0]`	

15	**MACCXD.PP.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product and accumulate**

	`Dm = (Dm + (((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Db.H * Dd.H) - (Db.L * Dd.L))) << 1))[39:0]`
	`Dn = (Dn + (((Da.H * Dc.L) + (Da.L * Dc.H)) + ((Db.H * Dd.L) + (Db.L * Dd.H))) << 1))[39:0]`	

16	**MACCXD.PP.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product and accumulate, single first operand**

	`Dm = (Dm + (((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Da.H * Dd.H) - (Da.L * Dd.L))) << 1))[39:0]`
	`Dn = (Dn + (((Da.H * Dc.L) + (Da.L * Dc.H)) + ((Da.H * Dd.L) + (Da.L * Dd.H))) << 1))[39:0]`	

17	**MACCXD.PPX.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product and accumulate, swapped operands**

	`Dm = (Dm + (((Da.H * Dd.H) - (Da.L * Dd.L)) + ((Db.H * Dc.H) - (Db.L * Dc.L))) << 1))[39:0]`
	`Dn = (Dn + (((Da.H * Dd.L) + (Da.L * Dd.H)) + ((Db.H * Dc.L) + (Db.L * Dc.H))) << 1))[39:0]`	

18	**MACCXD.CNN.I.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate and accumulate, both products are negated**

	`Dm = (Dm - (((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Db.H * Dd.H) + (Db.L * Dd.L))))[39:0]`
	`Dn = (Dn - (((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Db.H * Dd.L) - (Db.L * Dd.H))))[39:0]`	

19	**MACCXD.CNN.I.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate and accumulate, both products are negated, single first operand**

	`Dm = (Dm - (((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Da.H * Dd.H) + (Da.L * Dd.L))))[39:0]`
	`Dn = (Dn - (((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Da.H * Dd.L) - (Da.L * Dd.H))))[39:0]`	

Table continues on the next page...

#	Syntax	Description
20	**MACCXD.CNP.I.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate and accumulate, first product is negated**

	`Dm = (Dm - (((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Db.H * Dd.H) + (Db.L * Dd.L))))[39:0]`
	`Dn = (Dn - (((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Db.H * Dd.L) - (Db.L * Dd.H))))[39:0]`	

21	**MACCXD.CNP.I.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate and accumulate, first product is negated, single first operand**

	`Dm = (Dm - (((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Da.H * Dd.H) + (Da.L * Dd.L))))[39:0]`
	`Dn = (Dn - (((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Da.H * Dd.L) - (Da.L * Dd.H))))[39:0]`	

22	**MACCXD.CPN.I.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate and accumulate, second product is negated**

	`Dm = (Dm + (((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Db.H * Dd.H) + (Db.L * Dd.L))))[39:0]`
	`Dn = (Dn + (((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Db.H * Dd.L) - (Db.L * Dd.H))))[39:0]`	

23	**MACCXD.CPN.I.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate and accumulate, second product is negated, single first operand**

	`Dm = (Dm + (((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Da.H * Dd.H) + (Da.L * Dd.L))))[39:0]`
	`Dn = (Dn + (((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Da.H * Dd.L) - (Da.L * Dd.H))))[39:0]`	

24	**MACCXD.CPP.I.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate and accumulate**

	`Dm = (Dm + (((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Db.H * Dd.H) + (Db.L * Dd.L))))[39:0]`
	`Dn = (Dn + (((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Db.H * Dd.L) - (Db.L * Dd.H))))[39:0]`	

25	**MACCXD.CPP.I.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate and accumulate, single first operand**

	`Dm = (Dm + (((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Da.H * Dd.H) + (Da.L * Dd.L))))[39:0]`
	`Dn = (Dn + (((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Da.H * Dd.L) - (Da.L * Dd.H))))[39:0]`	

26	**MACCXD.NN.I.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product and accumulate, both products are negated**

	`Dm = (Dm - (((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Db.H * Dd.H) - (Db.L * Dd.L))))[39:0]`
	`Dn = (Dn - (((Da.H * Dc.L) + (Da.L * Dc.H)) + ((Db.H * Dd.L) + (Db.L * Dd.H))))[39:0]`	

27	**MACCXD.NN.I.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product and accumulate, both products are negated, single first operand**

	`Dm = (Dm - (((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Da.H * Dd.H) - (Da.L * Dd.L))))[39:0]`
	`Dn = (Dn - (((Da.H * Dc.L) + (Da.L * Dc.H)) + ((Da.H * Dd.L) + (Da.L * Dd.H))))[39:0]`	

28	**MACCXD.NP.I.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product and accumulate, first product is negated**

	`Dm = (Dm - (((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Db.H * Dd.H) - (Db.L * Dd.L))))[39:0]`
	`Dn = (Dn - (((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Db.H * Dd.L) + (Db.L * Dd.H))))[39:0]`	

29	**MACCXD.NP.I.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product and accumulate, first product is negated, single first operand**

	`Dm = (Dm - (((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Da.H * Dd.H) - (Da.L * Dd.L))))[39:0]`
	`Dn = (Dn - (((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Da.H * Dd.L) + (Da.L * Dd.H))))[39:0]`	

Table continues on the next page...

#	Syntax	Description
30	**MACCXD.PN.I.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product and accumulate, second product is negated**

	`Dm = (Dm + (((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Db.H * Dd.H) - (Db.L * Dd.L))))[39:0]`
	`Dn = (Dn + (((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Db.H * Dd.L) + (Db.L * Dd.H))))[39:0]`	

31	**MACCXD.PN.I.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product and accumulate, second product is negated, single first operand**

	`Dm = (Dm + (((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Da.H * Dd.H) - (Da.L * Dd.L))))[39:0]`
	`Dn = (Dn + (((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Da.H * Dd.L) + (Da.L * Dd.H))))[39:0]`	

32	**MACCXD.PP.I.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product and accumulate**

	`Dm = (Dm + (((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Db.H * Dd.H) - (Db.L * Dd.L))))[39:0]`
	`Dn = (Dn + (((Da.H * Dc.L) + (Da.L * Dc.H)) + ((Db.H * Dd.L) + (Db.L * Dd.H))))[39:0]`	

33	**MACCXD.PP.I.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product and accumulate, single first operand**

	`Dm = (Dm + (((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Da.H * Dd.H) - (Da.L * Dd.L))))[39:0]`
	`Dn = (Dn + (((Da.H * Dc.L) + (Da.L * Dc.H)) + ((Da.H * Dd.L) + (Da.L * Dd.H))))[39:0]`	

34	**MACCXD.PPX.I.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product and accumulate, swapped operands**

	`Dm = (Dm + (((Da.H * Dd.H) - (Da.L * Dd.L)) + ((Db.H * Dc.H) - (Db.L * Dc.L))))[39:0]`
	`Dn = (Dn + (((Da.H * Dd.L) + (Da.L * Dd.H)) + ((Db.H * Dc.L) + (Db.L * Dc.H))))[39:0]`	

35	**MACCXD.CNN.IS.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate and accumulate with saturation, both products are negated**

	`Dm = SAT32((Dm - (((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Db.H * Dd.H) + (Db.L * Dd.L)))))`
	`Dn = SAT32((Dn - (((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Db.H * Dd.L) - (Db.L * Dd.H)))))`	

36	**MACCXD.CNN.IS.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate and accumulate with saturation, both products are negated, single first operand**

	`Dm = SAT32((Dm - (((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Da.H * Dd.H) + (Da.L * Dd.L)))))`
	`Dn = SAT32((Dn - (((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Da.H * Dd.L) - (Da.L * Dd.H)))))`	

37	**MACCXD.CNP.IS.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate and accumulate with saturation, first product is negated**

	`Dm = SAT32((Dm + (-(((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Db.H * Dd.H) + (Db.L * Dd.L))))))`
	`Dn = SAT32((Dn + (-(((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Db.H * Dd.L) - (Db.L * Dd.H))))))`	

38	**MACCXD.CNP.IS.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate and accumulate with saturation, first product is negated, single first operand**

	`Dm = SAT32((Dm + (-(((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Da.H * Dd.H) + (Da.L * Dd.L))))))`
	`Dn = SAT32((Dn + (-(((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Da.H * Dd.L) - (Da.L * Dd.H))))))`	

39	**MACCXD.CPN.IS.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate and accumulate with saturation, second product is negated**

Table continues on the next page...

#	Syntax	Description

	Dm = SAT32((Dm + (((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Db.H * Dd.H) + (Db.L * Dd.L)))))
	Dn = SAT32((Dn + (((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Db.H * Dd.L) - (Db.L * Dd.H)))))	

40	**MACCXD.CPN.IS.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate and accumulate with saturation, second product is negated, single first operand**

	Dm = SAT32((Dm + (((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Da.H * Dd.H) + (Da.L * Dd.L)))))
	Dn = SAT32((Dn + (((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Da.H * Dd.L) - (Da.L * Dd.H)))))	

41	**MACCXD.CPP.IS.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate and accumulate with saturation**

	Dm = SAT32((Dm + (((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Db.H * Dd.H) + (Db.L * Dd.L)))))
	Dn = SAT32((Dn + (((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Db.H * Dd.L) - (Db.L * Dd.H)))))	

42	**MACCXD.CPP.IS.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate and accumulate with saturation, single first operand**

	Dm = SAT32((Dm + (((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Da.H * Dd.H) + (Da.L * Dd.L)))))
	Dn = SAT32((Dn + (((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Da.H * Dd.L) - (Da.L * Dd.H)))))	

43	**MACCXD.NN.IS.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product and accumulate with saturation, both products are negated**

	Dm = SAT32((Dm - (((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Db.H * Dd.H) - (Db.L * Dd.L)))))
	Dn = SAT32((Dn - (((Da.H * Dc.L) + (Da.L * Dc.H)) + ((Db.H * Dd.L) + (Db.L * Dd.H)))))	

44	**MACCXD.NN.IS.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product and accumulate with saturation, both products are negated, single first operand**

	Dm = SAT32((Dm - (((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Da.H * Dd.H) - (Da.L * Dd.L)))))
	Dn = SAT32((Dn - (((Da.H * Dc.L) + (Da.L * Dc.H)) + ((Da.H * Dd.L) + (Da.L * Dd.H)))))	

45	**MACCXD.NP.IS.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product and accumulate with saturation, first product is negated**

	Dm = SAT32((Dm - (((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Db.H * Dd.H) - (Db.L * Dd.L)))))
	Dn = SAT32((Dn - (((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Db.H * Dd.L) + (Db.L * Dd.H)))))	

46	**MACCXD.NP.IS.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product and accumulate with saturation, first product is negated, single first operand**

	Dm = SAT32((Dm - (((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Da.H * Dd.H) - (Da.L * Dd.L)))))
	Dn = SAT32((Dn - (((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Da.H * Dd.L) + (Da.L * Dd.H)))))	

47	**MACCXD.PN.IS.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product and accumulate with saturation, second product is negated**

	Dm = SAT32((Dm + (((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Db.H * Dd.H) - (Db.L * Dd.L)))))
	Dn = SAT32((Dn + (((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Db.H * Dd.L) + (Db.L * Dd.H)))))	

48	**MACCXD.PN.IS.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product and accumulate with saturation, second product is negated, single first operand**

	Dm = SAT32((Dm + (((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Da.H * Dd.H) - (Da.L * Dd.L)))))
	Dn = SAT32((Dn + (((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Da.H * Dd.L) + (Da.L * Dd.H)))))	

49	**MACCXD.PP.IS.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product and accumulate with saturation**

Table continues on the next page...

#	Syntax	Description

	Dm = SAT32((Dm + (((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Db.H * Dd.H) - (Db.L * Dd.L)))))
	Dn = SAT32((Dn + (((Da.H * Dc.L) + (Da.L * Dc.H)) + ((Db.H * Dd.L) + (Db.L * Dd.H)))))	

50	**MACCXD.PP.IS.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product and accumulate with saturation, single first operand**

	Dm = SAT32((Dm + (((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Da.H * Dd.H) - (Da.L * Dd.L)))))
	Dn = SAT32((Dn + (((Da.H * Dc.L) + (Da.L * Dc.H)) + ((Da.H * Dd.L) + (Da.L * Dd.H)))))	

51	**MACCXD.PPX.IS.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product and accumulate with saturation, swapped operands**

	Dm = SAT32((Dm + (((Da.H * Dd.H) - (Da.L * Dd.L)) + ((Db.H * Dc.H) - (Db.L * Dc.L)))))
	Dn = SAT32((Dn + (((Da.H * Dd.L) + (Da.L * Dd.H)) + ((Db.H * Dc.L) + (Db.L * Dc.H)))))	

52	**MACCXD.CNN.S.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate and accumulate with saturation, both products are negated**

	Dm = SAT32((Dm - ((((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Db.H * Dd.H) + (Db.L * Dd.L))) << 1)))
	Dn = SAT32((Dn - ((((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Db.H * Dd.L) - (Db.L * Dd.H))) << 1)))	

53	**MACCXD.CNN.S.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate and accumulate with saturation, both products are negated, single first operand**

	Dm = SAT32((Dm - ((((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Da.H * Dd.H) + (Da.L * Dd.L))) << 1)))
	Dn = SAT32((Dn - ((((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Da.H * Dd.L) - (Da.L * Dd.H))) << 1)))	

54	**MACCXD.CNP.S.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate and accumulate with saturation, first product is negated**

	Dm = SAT32((Dm - ((((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Db.H * Dd.H) + (Db.L * Dd.L))) << 1)))
	Dn = SAT32((Dn - ((((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Db.H * Dd.L) - (Db.L * Dd.H))) << 1)))	

55	**MACCXD.CNP.S.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate and accumulate with saturation, first product is negated, single first operand**

	Dm = SAT32((Dm - ((((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Da.H * Dd.H) + (Da.L * Dd.L))) << 1)))
	Dn = SAT32((Dn - ((((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Da.H * Dd.L) - (Da.L * Dd.H))) << 1)))	

56	**MACCXD.CPN.S.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate and accumulate with saturation, second product is negated**

	Dm = SAT32((Dm + ((((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Db.H * Dd.H) + (Db.L * Dd.L))) << 1)))
	Dn = SAT32((Dn + ((((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Db.H * Dd.L) - (Db.L * Dd.H))) << 1)))	

57	**MACCXD.CPN.S.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate and accumulate with saturation, second product is negated, single first operand**

Table continues on the next page...

#	Syntax	Description

	Dm = SAT32((Dm + ((((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Da.H * Dd.H) + (Da.L * Dd.L))) << 1)))
	Dn = SAT32((Dn + ((((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Da.H * Dd.L) - (Da.L * Dd.H))) << 1)))	

58	**MACCXD.CPP.S.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate and accumulate with saturation**

	Dm = SAT32((Dm + ((((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Db.H * Dd.H) + (Db.L * Dd.L))) << 1)))
	Dn = SAT32((Dn + ((((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Db.H * Dd.L) - (Db.L * Dd.H))) << 1)))	

59	**MACCXD.CPP.S.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate and accumulate with saturation, single first operand**

	Dm = SAT32((Dm + ((((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Da.H * Dd.H) + (Da.L * Dd.L))) << 1)))
	Dn = SAT32((Dn + ((((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Da.H * Dd.L) - (Da.L * Dd.H))) << 1)))	

60	**MACCXD.NN.S.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product and accumulate with saturation, both products are negated**

	Dm = SAT32((Dm - ((((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Db.H * Dd.H) - (Db.L * Dd.L))) << 1)))
	Dn = SAT32((Dn - ((((Da.H * Dc.L) + (Da.L * Dc.H)) + ((Db.H * Dd.L) + (Db.L * Dd.H))) << 1)))	

61	**MACCXD.NN.S.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product and accumulate with saturation, both products are negated, single first operand**

	Dm = SAT32((Dm - ((((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Da.H * Dd.H) - (Da.L * Dd.L))) << 1)))
	Dn = SAT32((Dn - ((((Da.H * Dc.L) + (Da.L * Dc.H)) + ((Da.H * Dd.L) + (Da.L * Dd.H))) << 1)))	

62	**MACCXD.NP.S.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product and accumulate with saturation, first product is negated**

	Dm = SAT32((Dm - ((((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Db.H * Dd.H) - (Db.L * Dd.L))) << 1)))
	Dn = SAT32((Dn - ((((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Db.H * Dd.L) + (Db.L * Dd.H))) << 1)))	

63	**MACCXD.NP.S.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product and accumulate with saturation, first product is negated, single first operand**

	Dm = SAT32((Dm - ((((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Da.H * Dd.H) - (Da.L * Dd.L))) << 1)))
	Dn = SAT32((Dn - ((((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Da.H * Dd.L) + (Da.L * Dd.H))) << 1)))	

Table continues on the next page...

#	Syntax	Description
64	**MACCXD.PN.S.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product and accumulate with saturation, second product is negated**

	`Dm = SAT32((Dm + ((((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Db.H * Dd.H) - (Db.L * Dd.L))) << 1)))`
	`Dn = SAT32((Dn + ((((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Db.H * Dd.L) + (Db.L * Dd.H))) << 1)))`	

65	**MACCXD.PN.S.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product and accumulate with saturation, second product is negated, single first operand**

	`Dm = SAT32((Dm + ((((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Da.H * Dd.H) - (Da.L * Dd.L))) << 1)))`
	`Dn = SAT32((Dn + ((((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Da.H * Dd.L) + (Da.L * Dd.H))) << 1)))`	

66	**MACCXD.PP.S.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product and accumulate with saturation**

	`Dm = SAT32((Dm + ((((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Db.H * Dd.H) - (Db.L * Dd.L))) << 1)))`
	`Dn = SAT32((Dn + ((((Da.H * Dc.L) + (Da.L * Dc.H)) + ((Db.H * Dd.L) + (Db.L * Dd.H))) << 1)))`	

67	**MACCXD.PP.S.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product and accumulate with saturation, single first operand**

	`Dm = SAT32((Dm + ((((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Da.H * Dd.H) - (Da.L * Dd.L))) << 1)))`
	`Dn = SAT32((Dn + ((((Da.H * Dc.L) + (Da.L * Dc.H)) + ((Da.H * Dd.L) + (Da.L * Dd.H))) << 1)))`	

68	**MACCXD.PPX.S.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product and accumulate with saturation, swapped operands**

	`Dm = SAT32((Dm + ((((Da.H * Dd.H) - (Da.L * Dd.L)) + ((Db.H * Dc.H) - (Db.L * Dc.L))) << 1)))`
	`Dn = SAT32((Dn + ((((Da.H * Dd.L) + (Da.L * Dd.H)) + ((Db.H * Dc.L) + (Db.L * Dc.H))) << 1)))`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$

Affected By Instructions

Field	Relevant Variants
SR.SAT	35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3, 4, 9, 10, 11, 12, 18, 19, 20, 21, 22, 23, 25, 26, 27, 28, 29, 31, 33, 34, 35, 36, 37, 38, 39, 40, 42, 43, 44, 45, 46, 48, 50, 51, 52, 53, 54, 55, 60, 61, 62, 63
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACCXM.nX 16x32 bit Complex Multiply-Accumulate into 40-bit Result (DALU)

General Description

Performs mixed precision 16x32 bit complex multiply-accumulate into 40-bit results.
The 32/40-bit complex numbers resides in a register pair, while the 16-bit complex number is packed in a single register.

Flag Options

Flag	Position	Description
C	flg1	Complex Conjugate
I	flg2	Integer arithmetic
R	flg2	Round multiplication result
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MACCXM.IR.2X Da,Dc:Dd,Dm:Dn**	**Complex integer multiply-accumulate with rounding**

	`Dm = (Dm + ((((Da.H * Dc.M) - (Da.L * Dd.M)) + 0x8000) >> 16))[39:0]`
	`Dn = (Dn + ((((Da.H * Dd.M) + (Da.L * Dc.M)) + 0x8000) >> 16))[39:0]`	

2	**MACCXM.IR.2X -Da,Dc:Dd,Dm:Dn**	**Complex integer multiply-accumulate with negate and rounding**

	`Dm = (Dm + (((-((Da.H * Dc.M) - (Da.L * Dd.M))) + 0x8000) >> 16))[39:0]`
	`Dn = (Dn + (((-((Da.H * Dd.M) + (Da.L * Dc.M))) + 0x8000) >> 16))[39:0]`	

3	**MACCXM.C.IR.2X Da,Dc:Dd,Dm:Dn**	**Complex integer multiply by conjugate and accumulate with rounding**

	`Dm = (Dm + ((((Da.H * Dc.M) + (Da.L * Dd.M)) + 0x8000) >> 16))[39:0]`
	`Dn = (Dn + ((((Da.H * Dd.M) - (Da.L * Dc.M)) + 0x8000) >> 16))[39:0]`	

4	**MACCXM.C.IR.2X -Da,Dc:Dd,Dm:Dn**	**Complex integer multiply by conjugate and accumulate with negate and rounding**

	`Dm = (Dm + (((-((Da.H * Dc.M) + (Da.L * Dd.M))) + 0x8000) >> 16))[39:0]`
	`Dn = (Dn + (((-((Da.H * Dd.M) - (Da.L * Dc.M))) + 0x8000) >> 16))[39:0]`	

5	**MACCXM.ISR.2X Da,Dc:Dd,Dm:Dn**	**Complex integer multiply-accumulate with rounding and saturation**

	`Dm = SAT32((Dm + ((((Da.H * Dc.M) - (Da.L * Dd.M)) + 0x8000) >> 16)))`
	`Dn = SAT32((Dn + ((((Da.H * Dd.M) + (Da.L * Dc.M)) + 0x8000) >> 16)))`	

Table continues on the next page...

#	Syntax	Description
6	**MACCXM.ISR.2X -Da,Dc:Dd,Dm:Dn**	**Complex integer multiply-accumulate with negate and rounding and saturation**

	Dm = SAT32((Dm + (((-((Da.H * Dc.M) - (Da.L * Dd.M))) + 0x8000) >> 16)))
	Dn = SAT32((Dn + (((-((Da.H * Dd.M) + (Da.L * Dc.M))) + 0x8000) >> 16)))	

7	**MACCXM.C.ISR.2X Da,Dc:Dd,Dm:Dn**	**Complex integer multiply by conjugate and accumulate with rounding and saturation**

	Dm = SAT32((Dm + ((((Da.H * Dc.M) + (Da.L * Dd.M)) + 0x8000) >> 16)))
	Dn = SAT32((Dn + ((((Da.H * Dd.M) - (Da.L * Dc.M)) + 0x8000) >> 16)))	

8	**MACCXM.C.ISR.2X -Da,Dc:Dd,Dm:Dn**	**Complex integer multiply by conjugate and accumulate with negate and rounding and saturation**

	Dm = SAT32((Dm + (((-((Da.H * Dc.M) + (Da.L * Dd.M))) + 0x8000) >> 16)))
	Dn = SAT32((Dn + (((-((Da.H * Dd.M) - (Da.L * Dc.M))) + 0x8000) >> 16)))	

9	**MACCXM.R.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional multiply-accumulate with rounding**

	Dm = (Dm + (((((Da.H * Dc.M) << 1) - ((Da.L * Dd.M) << 1)) + 0x8000) >> 16))[39:0]
	Dn = (Dn + (((((Da.H * Dd.M) << 1) + ((Da.L * Dc.M) << 1)) + 0x8000) >> 16))[39:0]	

10	**MACCXM.R.2X -Da,Dc:Dd,Dm:Dn**	**Complex fractional multiply-accumulate with negate and rounding**

	Dm = (Dm + (((-(((Da.H * Dc.M) << 1) - ((Da.L * Dd.M) << 1))) + 0x8000) >> 16))[39:0]
	Dn = (Dn + (((-(((Da.H * Dd.M) << 1) + ((Da.L * Dc.M) << 1))) + 0x8000) >> 16))[39:0]	

11	**MACCXM.C.R.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional multiply by conjugate and accumulate with rounding**

	Dm = (Dm + (((((Da.H * Dc.M) << 1) + ((Da.L * Dd.M) << 1)) + 0x8000) >> 16))[39:0]
	Dn = (Dn + (((((Da.H * Dd.M) << 1) - ((Da.L * Dc.M) << 1)) + 0x8000) >> 16))[39:0]	

12	**MACCXM.C.R.2X -Da,Dc:Dd,Dm:Dn**	**Complex fractional multiply by conjugate and accumulate with negate and rounding**

	Dm = (Dm + (((-(((Da.H * Dc.M) << 1) + ((Da.L * Dd.M) << 1))) + 0x8000) >> 16))[39:0]
	Dn = (Dn + (((-(((Da.H * Dd.M) << 1) - ((Da.L * Dc.M) << 1))) + 0x8000) >> 16))[39:0]	

13	**MACCXM.SR.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional multiply-accumulate with rounding and saturation**

	Dm = SAT32((Dm + (((((Da.H * Dc.M) << 1) - ((Da.L * Dd.M) << 1)) + 0x8000) >> 16)))
	Dn = SAT32((Dn + (((((Da.H * Dd.M) << 1) + ((Da.L * Dc.M) << 1)) + 0x8000) >> 16)))	

14	**MACCXM.SR.2X -Da,Dc:Dd,Dm:Dn**	**Complex fractional multiply-accumulate with negate and rounding and saturation**

	Dm = SAT32((Dm + (((-(((Da.H * Dc.M) << 1) - ((Da.L * Dd.M) << 1))) + 0x8000) >> 16)))
	Dn = SAT32((Dn + (((-(((Da.H * Dd.M) << 1) + ((Da.L * Dc.M) << 1))) + 0x8000) >> 16)))	

15	**MACCXM.C.SR.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional multiply by conjugate and accumulate with rounding and saturation**

	Dm = SAT32((Dm + (((((Da.H * Dc.M) << 1) + ((Da.L * Dd.M) << 1)) + 0x8000) >> 16)))
	Dn = SAT32((Dn + (((((Da.H * Dd.M) << 1) - ((Da.L * Dc.M) << 1)) + 0x8000) >> 16)))	

Table continues on the next page...

#	Syntax	Description
16	**MACCXM.C.SR.2X -Da,Dc:Dd,Dm:Dn**	**Complex fractional multiply by conjugate and accumulate with negate and rounding and saturation**

```
Dm = SAT32((Dm + (((-(((Da.H * Dc.M) << 1) + ((Da.L * Dd.M) << 1))) + 0x8000) >> 16)))
Dn = SAT32((Dn + (((-(((Da.H * Dd.M) << 1) - ((Da.L * Dc.M) << 1))) + 0x8000) >> 16)))
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Affected By Instructions

Field	Relevant Variants
SR.SAT	5, 6, 7, 8, 13, 14, 15, 16

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3, 4, 5, 6, 7, 8
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACD.nT 16x16 bit Dot Product and Accumulate into 20-bit Result (DALU)

General Description

Performs SIMD2 16x16 bit dot product and accumulate into 20-bit results after rounding.
The selected register parts are selected in order to supports Real, Imaginary, and Conjugate-Imaginary variations.

Flag Options

Flag	Position	Description
C	flg1	Complex Conjugate
IM	flg1	Complex Imaginary
RE	flg1	Complex Real
I	flg2	Integer arithmetic
R	flg2	Round multiplication result

Instruction Variants

#	Syntax	Description
1	**MACD.IR.2T Da:Db,Dc:Dd,Dn**	**Integer dot product and accumulate with rounding, high by high and low by low parts**

	`Dn.WH = (Dn.WH + (((Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + (((Db.L * Dd.L) + (Db.H * Dd.H)) + 0x8000) >> 16))[19:0]`	

2	**MACD.CIM.IR.2T Da:Db,Dc:Dd,Dn**	**Integer dot product and accumulate with rounding, selected input parts produce the multiplied-by-conjugate imaginary parts**

	`Dn.WH = (Dn.WH + (((-((Da.L * Dc.H) - (Da.H * Dc.L))) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + (((-((Db.L * Dd.H) - (Db.H * Dd.L))) + 0x8000) >> 16))[19:0]`	

3	**MACD.IM.IR.2T Da:Db,Dc:Dd,Dn**	**Integer dot product and accumulate with rounding, selected input parts produce the imaginary parts**

	`Dn.WH = (Dn.WH + (((Da.L * Dc.H) + (Da.H * Dc.L)) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + (((Db.L * Dd.H) + (Db.H * Dd.L)) + 0x8000) >> 16))[19:0]`	

4	**MACD.RE.IR.2T Da:Db,Dc:Dd,Dn**	**Integer dot product and accumulate with rounding, selected input parts produce the real parts**

	`Dn.WH = (Dn.WH + (((Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + (((Db.H * Dd.H) - (Db.L * Dd.L)) + 0x8000) >> 16))[19:0]`	

5	**MACD.R.2T Da:Db,Dc:Dd,Dn**	**fractional dot product and accumulate with rounding, high by high and low by low parts**

	`Dn.WH = (Dn.WH + ((((Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + ((((Db.L * Dd.L) + (Db.H * Dd.H)) << 1) + 0x8000) >> 16))[19:0]`	

Table continues on the next page...

#	Syntax	Description
6	**MACD.CIM.R.2T Da:Db,Dc:Dd,Dn**	**fractional dot product and accumulate with rounding, selected input parts produce the multiplied-by-conjugate imaginary parts**

	`Dn.WH = ((Dn.WH) + (((-(((Da.L * Dc.H) - (Da.H * Dc.L)) << 1)) + 0x8000) >> 16))[19:0]`
	`Dn.WL = ((Dn.WL) + (((-(((Db.L * Dd.H) - (Db.H * Dd.L)) << 1)) + 0x8000) >> 16))[19:0]`	

7	**MACD.IM.R.2T Da:Db,Dc:Dd,Dn**	**fractional dot product and accumulate with rounding, selected input parts produce the imaginary parts**

	`Dn.WH = (Dn.WH + (((((Da.L * Dc.H) + (Da.H * Dc.L)) << 1) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + (((((Db.L * Dd.H) + (Db.H * Dd.L)) << 1) + 0x8000) >> 16))[19:0]`	

8	**MACD.RE.R.2T Da:Db,Dc:Dd,Dn**	**fractional dot product and accumulate with rounding, selected input parts produce the real parts**

	`Dn.WH = (Dn.WH + (((((Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + (((((Db.H * Dd.H) - (Db.L * Dd.L)) << 1) + 0x8000) >> 16))[19:0]`	

Explicit Operands

Operand	Permitted Values	
Da:Db	D_n:D_{n+1}	0 ≤ n ≤ 62
Dc:Dd	D_n:D_{n+1}	0 ≤ n ≤ 62
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3, 4
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACD.nW 16x16 bit Dot Product and Accumulate into 16-bit Result with Saturation (DALU)

General Description

Performs SIMD2 16x16 bit dot product and accumulate into 16-bit results after rounding and saturation.
The selected register parts are selected in order to supports Real, Imaginary, and Conjugate-Imaginary variations.

Flag Options

Flag	Position	Description
C	flg1	Complex Conjugate
IM	flg1	Complex Imaginary
RE	flg1	Complex Real
I	flg2	Integer arithmetic
R	flg2	Round multiplication result
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MACD.ISR.2W Da:Db,Dc:Dd,Dn**	**Integer dot product and accumulate with rounding and saturate, high by high and low by low parts**

	`Dn.H = SAT16 ((Dn.H + ((((Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16)))[15:0]`
	`Dn.L = SAT16 ((Dn.L + ((((Db.H * Dd.H) + (Db.L * Dd.L)) + 0x8000) >> 16)))[15:0]`
	`Dn.E = 0`	

2	**MACD.CIM.ISR.2W Da:Db,Dc:Dd,Dn**	**Integer dot product and accumulate with rounding and saturate, selected input parts produce the multiplied-by-conjugate imaginary parts**

	`Dn.H = SAT16 ((Dn.H + (((-((Da.L * Dc.H) - (Da.H * Dc.L))) + 0x8000) >> 16)))[15:0]`
	`Dn.L = SAT16 ((Dn.L + (((-((Db.L * Dd.H) - (Db.H * Dd.L))) + 0x8000) >> 16)))[15:0]`
	`Dn.E = 0`	

3	**MACD.IM.ISR.2W Da:Db,Dc:Dd,Dn**	**Integer dot product and accumulate with rounding and saturate, selected input parts produce the imaginary parts**

	`Dn.H = SAT16 ((Dn.H + ((((Da.L * Dc.H) + (Da.H * Dc.L)) + 0x8000) >> 16)))[15:0]`
	`Dn.L = SAT16 ((Dn.L + ((((Db.L * Dd.H) + (Db.H * Dd.L)) + 0x8000) >> 16)))[15:0]`
	`Dn.E = 0`	

Table continues on the next page...

#	Syntax	Description
4	**MACD.RE.ISR.2W Da:Db,Dc:Dd,Dn**	**Integer dot product and accumulate with rounding and saturate, selected input parts produce the real parts**

	`Dn.H = SAT16 ((Dn.H + ((((Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16)))[15:0]`
	`Dn.L = SAT16 ((Dn.L + ((((Db.H * Dd.H) - (Db.L * Dd.L)) + 0x8000) >> 16)))[15:0]`
	`Dn.E = 0`	

5	**MACD.SR.2W Da:Db,Dc:Dd,Dn**	**fractional dot product and accumulate with rounding and saturate, high by high and low by low parts**

	`Dn.H = SAT16 ((Dn.H + (((((Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.L = SAT16 ((Dn.L + (((((Db.L * Dd.L) + (Db.H * Dd.H)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.E = 0`	

6	**MACD.CIM.SR.2W Da:Db,Dc:Dd,Dn**	**fractional dot product and accumulate with rounding and saturate, selected input parts produce the multiplied-by-conjugate imaginary parts**

	`Dn.H = SAT16 ((Dn.H + (((-(((Da.L * Dc.H) - (Da.H * Dc.L)) << 1)) + 0x8000) >> 16))) [15:0]`
	`Dn.L = SAT16 ((Dn.L + (((-(((Db.L * Dd.H) - (Db.H * Dd.L)) << 1)) + 0x8000) >> 16))) [15:0]`
	`Dn.E = 0`	

7	**MACD.IM.SR.2W Da:Db,Dc:Dd,Dn**	**fractional dot product and accumulate with rounding and saturate, selected input parts produce the imaginary parts**

	`Dn.H = SAT16 ((Dn.H + (((((Da.L * Dc.H) + (Da.H * Dc.L)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.L = SAT16 ((Dn.L + (((((Db.L * Dd.H) + (Db.H * Dd.L)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.E = 0`	

8	**MACD.RE.SR.2W Da:Db,Dc:Dd,Dn**	**fractional dot product and accumulate with rounding and saturate, selected input parts produce the real parts**

	`Dn.H = SAT16 ((Dn.H + (((((Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.L = SAT16 ((Dn.L + (((((Db.H * Dd.H) - (Db.L * Dd.L)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.E = 0`	

Explicit Operands

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3, 4
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACD.nX 16x16 bit Dot Product and Accumulate into 40-bit Result (DALU)

General Description

Performs single/SIMD2 16x16 bit dot product and accumulate into 40-bit results.
The selected register parts are selected in order to supports Real, Imaginary, and Conjugate-Imaginary variations.

Flag Options

Flag	Position	Description
C	flg1	Complex Conjugate
IM	flg1	Complex Imaginary
RE	flg1	Complex Real
I	flg2	Integer arithmetic
LEG	flg2	Legacy instruction
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MACD.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product and accumulate**

	`Dm = (Dm + (((Da.L * Dc.L) + (Da.H * Dc.H)) << 1))[39:0]`
	`Dn = (Dn + (((Db.L * Dd.L) + (Db.H * Dd.H)) << 1))[39:0]`	

2	**MACD.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product and accumulate, single first operand**

	`Dm = (Dm + (((Da.L * Dc.L) + (Da.H * Dc.H)) << 1))[39:0]`
	`Dn = (Dn + (((Da.L * Dd.L) + (Da.H * Dd.H)) << 1))[39:0]`	

3	**MACD.2X -Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product and accumulate with negate**

	`Dm = (Dm - (((Da.L * Dc.L) + (Da.H * Dc.H)) << 1))[39:0]`
	`Dn = (Dn - (((Db.L * Dd.L) + (Db.H * Dd.H)) << 1))[39:0]`	

4	**MACD.X Da,Db,Dn**	**Single fractional dot product and accumulate**
	`Dn = (Dn + (((Da.L * Db.L) + (Da.H * Db.H)) << 1))[39:0]`	
5	**MACD.CIM.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product and accumulate, selected input parts produce the multiplied-by-conjugate imaginary parts**

	`Dm = (Dm - (((Da.L * Dc.H) - (Da.H * Dc.L)) << 1))[39:0]`
	`Dn = (Dn - (((Db.L * Dd.H) - (Db.H * Dd.L)) << 1))[39:0]`	

Table continues on the next page...

#	Syntax	Description
6	**MACD.CIM.X Da,Db,Dn**	**Single fractional dot product and accumulate, selected input parts produce the multiplied-by-conjugate imaginary parts**
	`Dn = (Dn - (((Da.L * Db.H) - (Da.H * Db.L)) << 1))[39:0]`	
7	**MACD.IM.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product and accumulate, selected input parts produce the imaginary parts**

	`Dm = (Dm + (((Da.L * Dc.H) + (Da.H * Dc.L)) << 1))[39:0]`
	`Dn = (Dn + (((Db.L * Dd.H) + (Db.H * Dd.L)) << 1))[39:0]`	

8	**MACD.IM.2X -Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product and accumulate with negate, selected input parts produce the imaginary parts**

	`Dm = (Dm - (((Da.L * Dc.H) + (Da.H * Dc.L)) << 1))[39:0]`
	`Dn = (Dn - (((Db.L * Dd.H) + (Db.H * Dd.L)) << 1))[39:0]`	

9	**MACD.IM.X Da,Db,Dn**	**Single fractional dot product and accumulate, selected input parts produce the imaginary parts**
	`Dn = (Dn + (((Da.L * Db.H) + (Da.H * Db.L)) << 1))[39:0]`	
10	**MACD.RE.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product and accumulate, selected input parts produce the real parts**

	`Dm = (Dm + ((((Da.H) * (Dc.H)) - ((Da.L) * (Dc.L))) << 1))[39:0]`
	`Dn = (Dn + ((((Db.H) * (Dd.H)) - ((Db.L) * (Dd.L))) << 1))[39:0]`	

11	**MACD.RE.2X -Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product and accumulate with negate, selected input parts produce the real parts**

	`Dm = (Dm - ((((Da.H) * (Dc.H)) - ((Da.L) * (Dc.L))) << 1))[39:0]`
	`Dn = (Dn - ((((Db.H) * (Dd.H)) - ((Db.L) * (Dd.L))) << 1))[39:0]`	

12	**MACD.RE.X Da,Db,Dn**	**Single fractional dot product and accumulate, selected input parts produce the real parts**
	`Dn = (Dn + ((((Da.H) * (Db.H)) - ((Da.L) * (Db.L))) << 1))[39:0]`	
13	**MACD.I.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product and accumulate**

	`Dm = (Dm + ((Da.L * Dc.L) + (Da.H * Dc.H)))[39:0]`
	`Dn = (Dn + ((Db.L * Dd.L) + (Db.H * Dd.H)))[39:0]`	

14	**MACD.I.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product and accumulate, single first operand**

	`Dm = (Dm + ((Da.L * Dc.L) + (Da.H * Dc.H)))[39:0]`
	`Dn = (Dn + ((Da.L * Dd.L) + (Da.H * Dd.H)))[39:0]`	

15	**MACD.CIM.I.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product and accumulate, selected input parts produce the multiplied-by-conjugate imaginary parts**

	`Dm = (Dm - ((Da.L * Dc.H) - (Da.H * Dc.L)))[39:0]`
	`Dn = (Dn - ((Db.L * Dd.H) - (Db.H * Dd.L)))[39:0]`	

Table continues on the next page...

#	Syntax	Description
16	**MACD.IM.I.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product and accumulate, selected input parts produce the imaginary parts**

	Dm = (Dm + ((Da.L * Dc.H) + (Da.H * Dc.L)))[39:0]
	Dn = (Dn + ((Db.L * Dd.H) + (Db.H * Dd.L)))[39:0]	

17	**MACD.RE.I.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product and accumulate, selected input parts produce the real parts**

	Dm = (Dm + (((Da.H) * (Dc.H)) - ((Da.L) * (Dc.L))))[39:0]
	Dn = (Dn + (((Db.H) * (Dd.H)) - ((Db.L) * (Dd.L))))[39:0]	

18	**MACD.IS.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product and accumulate with saturation**

	Dm = SAT32((Dm + ((Da.H * Dc.H) + (Da.L * Dc.L))))
	Dn = SAT32((Dn + ((Db.H * Dd.H) + (Db.L * Dd.L))))	

19	**MACD.IS.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product and accumulate with saturation, single first operand**

	Dm = SAT32((Dm + ((Da.H * Dc.H) + (Da.L * Dc.L))))
	Dn = SAT32((Dn + ((Da.H * Dd.H) + (Da.L * Dd.L))))	

20	**MACD.CIM.IS.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product and accumulate with saturation, selected input parts produce the multiplied-by-conjugate imaginary parts**

	Dm = SAT32((Dm - ((Da.L * Dc.H) - (Da.H * Dc.L))))
	Dn = SAT32((Dn - ((Db.L * Dd.H) - (Db.H * Dd.L))))	

21	**MACD.IM.IS.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product and accumulate with saturation, selected input parts produce the imaginary parts**

	Dm = SAT32((Dm + ((Da.L * Dc.H) + (Da.H * Dc.L))))
	Dn = SAT32((Dn + ((Db.L * Dd.H) + (Db.H * Dd.L))))	

22	**MACD.RE.IS.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product and accumulate with saturation, selected input parts produce the real parts**

	Dm = SAT32((Dm + (((Da.H) * (Dc.H)) - ((Da.L) * (Dc.L)))))
	Dn = SAT32((Dn + (((Db.H) * (Dd.H)) - ((Db.L) * (Dd.L)))))	

23	**MACD.LEG.X Da,Db,Dn**	**Single fractional dot product and accumulate, saturating only if SR.SM is set**
	Dn = srSAT32((((Da.L * Db.L) + (Da.H * Db.H)) << 1) + Dn))	
24	**MACD.LEG.X -Da,Db,Dn**	**Single fractional dot product and accumulate with negate, saturating only if SR.SM is set**
	Dn = srSAT32(((((-(Da.L * Db.L)) - (Da.H * Db.H)) << 1) + Dn))	
25	**MACD.CIM.LEG.X -Da,Db,Dn**	**Single fractional dot product and accumulate, selected input parts produce the multiplied-by-conjugate imaginary parts, saturating only if SR.SM is set**
	Dn = srSAT32((Dn + (((Da.L * Db.H) - (Da.H * Db.L)) << 1)))	

Table continues on the next page...

#	Syntax	Description
26	**MACD.IM.LEG.X Da,Db,Dn**	**Single fractional dot product and accumulate, selected input parts produce the imaginary parts, saturating only if SR.SM is set**
	`Dn = srSAT32((Dn + (((Da.L * Db.H) + (Da.H * Db.L)) << 1)))`	
27	**MACD.RE.LEG.X Da,Db,Dn**	**Single fractional dot product and accumulate, selected input parts produce the real parts, saturating only if SR.SM is set**
	`Dn = srSAT32((Dn + ((((Da.H) * (Db.H)) - ((Da.L) * (Db.L))) << 1)))`	
28	**MACD.S.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product and accumulate with saturation**

	`Dm = SAT32((Dm + (((Da.L * Dc.L) + (Da.H * Dc.H)) << 1)))`
	`Dn = SAT32((Dn + (((Db.L * Dd.L) + (Db.H * Dd.H)) << 1)))`	

29	**MACD.S.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product and accumulate with saturation, single first operand**

	`Dm = SAT32((Dm + (((Da.L * Dc.L) + (Da.H * Dc.H)) << 1)))`
	`Dn = SAT32((Dn + (((Da.L * Dd.L) + (Da.H * Dd.H)) << 1)))`	

30	**MACD.S.X Da,Db,Dn**	**Single fractional dot product and accumulate with saturation**
	`Dn = SAT32((Dn + (((Da.L * Db.L) + (Da.H * Db.H)) << 1)))`	
31	**MACD.CIM.S.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product and accumulate with saturation, selected input parts produce the multiplied-by-conjugate imaginary parts**

	`Dm = SAT32((Dm - (((Da.L * Dc.H) - (Da.H * Dc.L)) << 1)))`
	`Dn = SAT32((Dn - (((Db.L * Dd.H) - (Db.H * Dd.L)) << 1)))`	

32	**MACD.CIM.S.X Da,Db,Dn**	**Single fractional dot product and accumulate with saturation, selected input parts produce the multiplied-by-conjugate imaginary parts**
	`Dn = SAT32((Dn - (((Da.L * Db.H) - (Da.H * Db.L)) << 1)))`	
33	**MACD.IM.S.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product and accumulate with saturation, selected input parts produce the imaginary parts**

	`Dm = SAT32((Dm + (((Da.L * Dc.H) + (Da.H * Dc.L)) << 1)))`
	`Dn = SAT32((Dn + (((Db.L * Dd.H) + (Db.H * Dd.L)) << 1)))`	

34	**MACD.IM.S.X Da,Db,Dn**	**Single fractional dot product and accumulate with saturation, selected input parts produce the imaginary parts**
	`Dn = SAT32((Dn + (((Da.L * Db.H) + (Da.H * Db.L)) << 1)))`	
35	**MACD.RE.S.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product and accumulate with saturation, selected input parts produce the real parts**

	`Dm = SAT32((Dm + ((((Da.H) * (Dc.H)) - ((Da.L) * (Dc.L))) << 1)))`
	`Dn = SAT32((Dn + ((((Db.H) * (Dd.H)) - ((Db.L) * (Dd.L))) << 1)))`	

Table continues on the next page...

#	Syntax	Description
36	**MACD.RE.S.X Da,Db,Dn**	**Single fractional dot product and accumulate with saturation, selected input parts produce the real parts**
	`Dn = SAT32((Dn + (((Da.H * Db.H) - (Da.L * Db.L)) << 1)))`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \leq n \leq 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \leq n \leq 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \leq n \leq 62$
Dn	D0-D63	

Affect Instructions

Field	Relevant Variants	Comments
SR.SM	23, 24, 25, 26, 27	

Affected By Instructions

Field	Relevant Variants
SR.SAT	18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	14, 18, 19, 20, 21, 22
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACDCF.nT 16x16 bit Dot Product and Accumulate Real by Complex FIR into 20-bit Result (DALU)

General Description

Performs SIMD4 16x16 bit dot product and accumulate into 20-bit results. Operands parts are optimized for Real by Complex FIR/correlation implementation.

Flag Options

Flag	Position	Description
H	flg1	High 16-bit portion
INV	flg1	Inverse order
L	flg1	Use the low portion
I	flg2	Integer arithmetic
R	flg2	Round multiplication result

Instruction Variants

#	Syntax	Description
1	**MACDCF.H.IR.4T Da:Db,Dc:Dd,Dm:Dn**	**Integer dot product and accumulate with rounding, using 3 high coefficients**

	`Dm.WH = (Dm.WH + ((((Da.H * Dc.L) + (Db.H * Dc.H)) + 0x8000) >> 16))[19:0]`
	`Dm.WL = (Dm.WL + ((((Da.L * Dc.L) + (Db.L * Dc.H)) + 0x8000) >> 16))[19:0]`
	`Dn.WH = (Dn.WH + ((((Da.H * Dd.H) + (Db.H * Dc.L)) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + ((((Da.L * Dd.H) + (Db.L * Dc.L)) + 0x8000) >> 16))[19:0]`	

2	**MACDCF.INVH.IR.4T Da:Db,Dc:Dd,Dm:Dn**	**Integer dot product and accumulate with rounding, using 3 high coefficients in inverse order**

	`Dm.WH = (Dm.WH + ((((Da.H * Dc.H) + (Db.H * Dc.L)) + 0x8000) >> 16))[19:0]`
	`Dm.WL = (Dm.WL + ((((Da.L * Dc.H) + (Db.L * Dc.L)) + 0x8000) >> 16))[19:0]`
	`Dn.WH = (Dn.WH + ((((Da.H * Dc.L) + (Db.H * Dd.H)) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + ((((Da.L * Dc.L) + (Db.L * Dd.H)) + 0x8000) >> 16))[19:0]`	

3	**MACDCF.INVL.IR.4T Da:Db,Dc:Dd,Dm:Dn**	**Integer dot product and accumulate with rounding, using 3 low coefficients in inverse order**

	`Dm.WH = (Dm.WH + ((((Da.H * Dc.L) + (Db.H * Dd.H)) + 0x8000) >> 16))[19:0]`
	`Dm.WL = (Dm.WL + ((((Da.L * Dc.L) + (Db.L * Dd.H)) + 0x8000) >> 16))[19:0]`
	`Dn.WH = (Dn.WH + ((((Da.H * Dd.H) + (Db.H * Dd.L)) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + ((((Da.L * Dd.H) + (Db.L * Dd.L)) + 0x8000) >> 16))[19:0]`	

Table continues on the next page...

#	Syntax	Description
4	**MACDCF.L.IR.4T Da:Db,Dc:Dd,Dm:Dn**	**Integer dot product and accumulate with rounding, using 3 low coefficients**

	`Dm.WH = (Dm.WH + ((((Da.H * Dd.H) + (Db.H * Dc.L)) + 0x8000) >> 16))[19:0]`
	`Dm.WL = (Dm.WL + ((((Da.L * Dd.H) + (Db.L * Dc.L)) + 0x8000) >> 16))[19:0]`
	`Dn.WH = (Dn.WH + ((((Da.H * Dd.L) + (Db.H * Dd.H)) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + ((((Da.L * Dd.L) + (Db.L * Dd.H)) + 0x8000) >> 16))[19:0]`	

5	**MACDCF.H.R.4T Da:Db,Dc:Dd,Dm:Dn**	**Fractional dot product and accumulate with rounding, using 3 high coefficients**

	`Dm.WH = (Dm.WH + (((((Da.H * Dc.L) + (Db.H * Dc.H)) << 1) + 0x8000) >> 16))[19:0]`
	`Dm.WL = (Dm.WL + (((((Da.L * Dc.L) + (Db.L * Dc.H)) << 1) + 0x8000) >> 16))[19:0]`
	`Dn.WH = (Dn.WH + (((((Da.H * Dd.H) + (Db.H * Dc.L)) << 1) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + (((((Da.L * Dd.H) + (Db.L * Dc.L)) << 1) + 0x8000) >> 16))[19:0]`	

6	**MACDCF.INVH.R.4T Da:Db,Dc:Dd,Dm:Dn**	**Fractional dot product and accumulate with rounding, using 3 high coefficients in inverse order**

	`Dm.WH = (Dm.WH + (((((Da.H * Dc.H) + (Db.H * Dc.L)) << 1) + 0x8000) >> 16))[19:0]`
	`Dm.WL = (Dm.WL + (((((Da.L * Dc.H) + (Db.L * Dc.L)) << 1) + 0x8000) >> 16))[19:0]`
	`Dn.WH = (Dn.WH + (((((Da.H * Dc.L) + (Db.H * Dd.H)) << 1) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + (((((Da.L * Dc.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16))[19:0]`	

7	**MACDCF.INVL.R.4T Da:Db,Dc:Dd,Dm:Dn**	**Fractional dot product and accumulate with rounding, using 3 low coefficients in inverse order**

	`Dm.WH = (Dm.WH + (((((Da.H * Dc.L) + (Db.H * Dd.H)) << 1) + 0x8000) >> 16))[19:0]`
	`Dm.WL = (Dm.WL + (((((Da.L * Dc.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16))[19:0]`
	`Dn.WH = (Dn.WH + (((((Da.H * Dd.H) + (Db.H * Dd.L)) << 1) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + (((((Da.L * Dd.H) + (Db.L * Dd.L)) << 1) + 0x8000) >> 16))[19:0]`	

8	**MACDCF.L.R.4T Da:Db,Dc:Dd,Dm:Dn**	**Fractional dot product and accumulate with rounding, using 3 low coefficients**

	`Dm.WH = (Dm.WH + (((((Da.H * Dd.H) + (Db.H * Dc.L)) << 1) + 0x8000) >> 16))[19:0]`
	`Dm.WL = (Dm.WL + (((((Da.L * Dd.H) + (Db.L * Dc.L)) << 1) + 0x8000) >> 16))[19:0]`
	`Dn.WH = (Dn.WH + (((((Da.H * Dd.L) + (Db.H * Dd.H)) << 1) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + (((((Da.L * Dd.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16))[19:0]`	

Explicit Operands

Operand	Permitted Values	
Da:Db	D_n:D_{n+1}	0 ≤ n ≤ 62
Dc:Dd	D_n:D_{n+1}	0 ≤ n ≤ 62
Dm:Dn	D_n:D_{n+1}	0 ≤ n ≤ 62

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3, 4
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACDCF.nW 16x16 bit Dot Product and Accumulate Real by Complex FIR into 16-bit Result with Saturation (DALU)

General Description

Performs SIMD4 16x16 bit dot product and accumulate into 16-bit results. Operands parts are optimized for Real by Complex FIR implementation.

Flag Options

Flag	Position	Description
H	flg1	High 16-bit portion
L	flg1	Use the low portion
I	flg2	Integer arithmetic
R	flg2	Round multiplication result
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MACDCF.H.ISR.4W Da:Db,Dc:Dd,Dm:Dn**	**Integer dot product and accumulate with rounding and saturation, using 3 high coefficients**

	Dm.H = SAT16 ((Dm.H + (((Da.H * Dc.L) + (Db.H * Dc.H)) + 0x8000) >> 16)))[15:0]
	Dm.L = SAT16 ((Dm.L + (((Da.L * Dc.L) + (Db.L * Dc.H)) + 0x8000) >> 16)))[15:0]
	Dm.E = 0
	Dn.H = SAT16 ((Dn.H + (((Da.H * Dd.H) + (Db.H * Dc.L)) + 0x8000) >> 16)))[15:0]
	Dn.L = SAT16 ((Dn.L + (((Da.L * Dd.H) + (Db.L * Dc.L)) + 0x8000) >> 16)))[15:0]
	Dn.E = 0	

2	**MACDCF.L.ISR.4W Da:Db,Dc:Dd,Dm:Dn**	**Integer dot product and accumulate with rounding and saturation, using 3 low coefficients**

	Dm.H = SAT16 ((Dm.H + (((Da.H * Dd.H) + (Db.H * Dc.L)) + 0x8000) >> 16)))[15:0]
	Dm.L = SAT16 ((Dm.L + (((Da.L * Dd.H) + (Db.L * Dc.L)) + 0x8000) >> 16)))[15:0]
	Dm.E = 0
	Dn.H = SAT16 ((Dn.H + (((Da.H * Dd.L) + (Db.H * Dd.H)) + 0x8000) >> 16)))[15:0]
	Dn.L = SAT16 ((Dn.L + (((Da.L * Dd.L) + (Db.L * Dd.H)) + 0x8000) >> 16)))[15:0]
	Dn.E = 0	

3	**MACDCF.H.SR.4W Da:Db,Dc:Dd,Dm:Dn**	**Fractional dot product and accumulate with rounding and saturation, using 3 high coefficients**

	Dm.H = SAT16 ((Dm.H + ((((Da.H * Dc.L) + (Db.H * Dc.H)) << 1) + 0x8000) >> 16)))[15:0]
	Dm.L = SAT16 ((Dm.L + ((((Da.L * Dc.L) + (Db.L * Dc.H)) << 1) + 0x8000) >> 16)))[15:0]
	Dm.E = 0
	Dn.H = SAT16 ((Dn.H + ((((Da.H * Dd.H) + (Db.H * Dc.L)) << 1) + 0x8000) >> 16)))[15:0]
	Dn.L = SAT16 ((Dn.L + ((((Da.L * Dd.H) + (Db.L * Dc.L)) << 1) + 0x8000) >> 16)))[15:0]
	Dn.E = 0	

Table continues on the next page...

#	Syntax	Description
4	**MACDCF.L.SR.4W Da:Db,Dc:Dd,Dm:Dn**	**Fractional dot product and accumulate with rounding and saturation, using 3 low coefficients**

```
Dm.H = SAT16 ((Dm.H + (((((Da.H * Dd.H) + (Db.H * Dc.L)) << 1) + 0x8000) >> 16)))[15:0]
Dm.L = SAT16 ((Dm.L + (((((Da.L * Dd.H) + (Db.L * Dc.L)) << 1) + 0x8000) >> 16)))[15:0]
Dm.E = 0
Dn.H = SAT16 ((Dn.H + (((((Da.H * Dd.L) + (Db.H * Dd.H)) << 1) + 0x8000) >> 16)))[15:0]
Dn.L = SAT16 ((Dn.L + (((((Da.L * Dd.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16)))[15:0]
Dn.E = 0
```

Explicit Operands

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACDCF.nX 16x16 bit Dot Product and Accumulate Real by Complex FIR into 40-bit Results (DALU)

General Description

Performs SIMD2 16x16 bit dot product and accumulate into 40-bit results. Operands parts are optimized for Real by Complex FIR implementation.

Flag Options

Flag	Position	Description
INV	flg1	Inverse order
I	flg2	Integer arithmetic

Instruction Variants

#	Syntax	Description
1	**MACDCF.2X Da,Dc:Dd,Dm:Dn**	**Fractional dot product and accumulate**

	`Dm = (Dm + (((Da.L * Dc.H) << 1) + ((Da.H * Dd.H) << 1)))[39:0]`
	`Dn = (Dn + (((Da.L * Dc.L) << 1) + ((Da.H * Dd.L) << 1)))[39:0]`	

2	**MACDCF.INV.2X Da,Dc:Dd,Dm:Dn**	**Fractional dot product and accumulate, using coefficients in inverse order**

	`Dm = (Dm + (((Da.L * Dd.H) << 1) + ((Da.H * Dc.H) << 1)))[39:0]`
	`Dn = (Dn + (((Da.L * Dd.L) << 1) + ((Da.H * Dc.L) << 1)))[39:0]`	

3	**MACDCF.I.2X Da,Dc:Dd,Dm:Dn**	**Integer dot product and accumulate**

	`Dm = (Dm + ((Da.L * Dc.H) + (Da.H * Dd.H)))[39:0]`
	`Dn = (Dn + ((Da.L * Dc.L) + (Da.H * Dd.L)))[39:0]`	

4	**MACDCF.INV.I.2X Da,Dc:Dd,Dm:Dn**	**Integer dot product and accumulate, using coefficients in inverse order**

	`Dm = (Dm + ((Da.L * Dd.H) + (Da.H * Dc.H)))[39:0]`
	`Dn = (Dn + ((Da.L * Dd.L) + (Da.H * Dc.L)))[39:0]`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	3, 4
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACDDCF.nT 16x16 bit Dot Product and Accumulate Real by Complex FIR with Decimation into 20-bit Result (DALU)

General Description

Performs SIMD4 16x16 bit dot product and accumulate into 20-bit results. Operands parts are optimized for Real by Complex FIR with decimation implementation.

Flag Options

Flag	Position	Description
I	flg2	Integer arithmetic
R	flg2	Round multiplication result

Instruction Variants

#	Syntax	Description
1	**MACDDCF.IR.4T Da:Db,Dc:Dd,Dm:Dn**	**Integer Real by Complex dot product and accumulate FIR, with decimation**

```
Dm.WH = (Dm.WH + ((((Da.H * Dd.H) + (Da.L * Dc.H)) + 0x8000) >> 16))[19:0]
Dm.WL = (Dm.WL + ((((Da.H * Dd.L) + (Da.L * Dc.L)) + 0x8000) >> 16))[19:0]
Dn.WH = (Dn.WH + ((((Db.H * Dd.H) + (Db.L * Dc.H)) + 0x8000) >> 16))[19:0]
Dn.WL = (Dn.WL + ((((Db.H * Dd.L) + (Db.L * Dc.L)) + 0x8000) >> 16))[19:0]
```

#	Syntax	Description
2	**MACDDCF.R.4T Da:Db,Dc:Dd,Dm:Dn**	**Fractional Real by Complex dot product and accumulate FIR, with decimation**

```
Dm.WH = (Dm.WH + (((((Da.H * Dd.H) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16))[19:0]
Dm.WL = (Dm.WL + (((((Da.H * Dd.L) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[19:0]
Dn.WH = (Dn.WH + (((((Db.H * Dd.H) + (Db.L * Dc.H)) << 1) + 0x8000) >> 16))[19:0]
Dn.WL = (Dn.WL + (((((Db.H * Dd.L) + (Db.L * Dc.L)) << 1) + 0x8000) >> 16))[19:0]
```

Explicit Operands

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACDDCF.nW 16x16 bit Dot Product and Accumulate Real by Complex FIR with Decimation into 16-bit Result with Saturation (DALU)

General Description

Performs SIMD4 16x16 bit dot product and accumulate into 16-bit results. Operands parts are optimized for Real by Complex FIR with decimation implementation.

Flag Options

Flag	Position	Description
I	flg2	Integer arithmetic
R	flg2	Round multiplication result
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MACDDCF.ISR.4W Da:Db,Dc:Dd,Dm:Dn**	**Integer Real by Complex dot product and accumulate with rounding and saturation FIR, with decimation**

	`Dm.WH = (U20)SAT16 ((Dm.H + ((((Da.H * Dd.H) + (Da.L * Dc.H)) + 0x8000) >> 16)))[15:0]`
	`Dm.WL = (U20)SAT16 ((Dm.L + ((((Da.H * Dd.L) + (Da.L * Dc.L)) + 0x8000) >> 16)))[15:0]`
	`Dn.WH = (U20)SAT16 ((Dn.H + ((((Db.H * Dd.H) + (Db.L * Dc.H)) + 0x8000) >> 16)))[15:0]`
	`Dn.WL = (U20)SAT16 ((Dn.L + ((((Db.H * Dd.L) + (Db.L * Dc.L)) + 0x8000) >> 16)))[15:0]`	

2	**MACDDCF.SR.4W Da:Db,Dc:Dd,Dm:Dn**	**Fractional Real by Complex dot product and accumulate with rounding and saturation FIR, with decimation**

	`Dm.WH = (U20)SAT16 ((Dm.H + (((((Da.H * Dd.H) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dm.WL = (U20)SAT16 ((Dm.L + (((((Da.H * Dd.L) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.WH = (U20)SAT16 ((Dn.H + (((((Db.H * Dd.H) + (Db.L * Dc.H)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.WL = (U20)SAT16 ((Dn.L + (((((Db.H * Dd.L) + (Db.L * Dc.L)) << 1) + 0x8000) >> 16)))[15:0]`	

Explicit Operands

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACDRF.nT 16x16 bit Dot Product and Accumulate Real by Real FIR into 20-bit Result (DALU)

General Description

Performs SIMD2/SIMD4 16x16 bit dot product and accumulate into 20-bit results. Operands parts are optimized for Real by Real FIR/correlation implementation.

Flag Options

Flag	Position	Description
H	flg1	High 16-bit portion
INV	flg1	Inverse order
L	flg1	Use the low portion
I	flg2	Integer arithmetic
R	flg2	Round multiplication result

Instruction Variants

#	Syntax	Description
1	**MACDRF.H.IR.2T Da,Dc:Dd,Dn**	**SIMD2 Integer dot product and accumulate with rounding, using 3 high coefficients**

	`Dn.WH = (Dn.WH + ((((Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + ((((Da.H * Dd.H) + (Da.L * Dc.L)) + 0x8000) >> 16))[19:0]`	

2	**MACDRF.H.IR.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 Integer dot product and accumulate with rounding, using 3 high coefficients**

	`Dm.WH = (Dm.WH + ((((Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16))[19:0]`
	`Dm.WL = (Dm.WL + ((((Da.H * Dd.H) + (Da.L * Dc.L)) + 0x8000) >> 16))[19:0]`
	`Dn.WH = (Dn.WH + ((((Db.H * Dc.L) + (Db.L * Dc.H)) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + ((((Db.H * Dd.H) + (Db.L * Dc.L)) + 0x8000) >> 16))[19:0]`	

3	**MACDRF.INVH.IR.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 Integer dot product and accumulate with rounding, using 3 high coefficients in inverse order**

	`Dm.WH = (Dm.WH + ((((Db.H * Dc.H) + (Db.L * Dc.L)) + 0x8000) >> 16))[19:0]`
	`Dm.WL = (Dm.WL + ((((Db.H * Dc.L) + (Db.L * Dd.H)) + 0x8000) >> 16))[19:0]`
	`Dn.WH = (Dn.WH + ((((Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16))[19:0]`
	`Dn.WL = (Dn.WL + ((((Da.H * Dc.L) + (Da.L * Dd.H)) + 0x8000) >> 16))[19:0]`	

Table continues on the next page...

#	Syntax	Description
4	**MACDRF.INVL.IR.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 Integer dot product and accumulate with rounding, using 3 low coefficients in inverse order**

	Dm.WH = (Dm.WH + ((((Db.H * Dc.L) + (Db.L * Dd.H)) + 0x8000) >> 16))[19:0]
	Dm.WL = (Dm.WL + ((((Db.H * Dd.H) + (Db.L * Dd.L)) + 0x8000) >> 16))[19:0]
	Dn.WH = (Dn.WH + ((((Da.H * Dc.L) + (Da.L * Dd.H)) + 0x8000) >> 16))[19:0]
	Dn.WL = (Dn.WL + ((((Da.H * Dd.H) + (Da.L * Dd.L)) + 0x8000) >> 16))[19:0]	

5	**MACDRF.L.IR.2T Da,Dc:Dd,Dn**	**SIMD2 Integer dot product and accumulate with rounding, using 3 low coefficients**

	Dn.WH = (Dn.WH + ((((Da.H * Dd.H) + (Da.L * Dc.L)) + 0x8000) >> 16))[19:0]
	Dn.WL = (Dn.WL + ((((Da.H * Dd.L) + (Da.L * Dd.H)) + 0x8000) >> 16))[19:0]	

6	**MACDRF.L.IR.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 Integer dot product and accumulate with rounding, using 3 low coefficients**

	Dm.WH = (Dm.WH + ((((Da.H * Dd.H) + (Da.L * Dc.L)) + 0x8000) >> 16))[19:0]
	Dm.WL = (Dm.WL + ((((Da.H * Dd.L) + (Da.L * Dd.H)) + 0x8000) >> 16))[19:0]
	Dn.WH = (Dn.WH + ((((Db.H * Dd.H) + (Db.L * Dc.L)) + 0x8000) >> 16))[19:0]
	Dn.WL = (Dn.WL + ((((Db.H * Dd.L) + (Db.L * Dd.H)) + 0x8000) >> 16))[19:0]	

7	**MACDRF.H.R.2T Da,Dc:Dd,Dn**	**SIMD2 fractional dot product and accumulate with rounding, using 3 high coefficients**

	Dn.WH = (Dn.WH + (((((Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16))[19:0]
	Dn.WL = (Dn.WL + (((((Da.H * Dd.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[19:0]	

8	**MACDRF.H.R.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 fractional dot product and accumulate with rounding, using 3 high coefficients**

	Dm.WH = (Dm.WH + (((((Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16))[19:0]
	Dm.WL = (Dm.WL + (((((Da.H * Dd.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[19:0]
	Dn.WH = (Dn.WH + (((((Db.H * Dc.L) + (Db.L * Dc.H)) << 1) + 0x8000) >> 16))[19:0]
	Dn.WL = (Dn.WL + (((((Db.H * Dd.H) + (Db.L * Dc.L)) << 1) + 0x8000) >> 16))[19:0]	

9	**MACDRF.INVH.R.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 fractional dot product and accumulate with rounding, using 3 high coefficients in inverse order**

	Dm.WH = (Dm.WH + (((((Db.H * Dc.H) + (Db.L * Dc.L)) << 1) + 0x8000) >> 16))[19:0]
	Dm.WL = (Dm.WL + (((((Db.H * Dc.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16))[19:0]
	Dn.WH = (Dn.WH + (((((Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[19:0]
	Dn.WL = (Dn.WL + (((((Da.H * Dc.L) + (Da.L * Dd.H)) << 1) + 0x8000) >> 16))[19:0]	

10	**MACDRF.INVL.R.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 fractional dot product and accumulate with rounding, using 3 low coefficients in inverse order**

	Dm.WH = (Dm.WH + (((((Db.H * Dc.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16))[19:0]
	Dm.WL = (Dm.WL + (((((Db.H * Dd.H) + (Db.L * Dd.L)) << 1) + 0x8000) >> 16))[19:0]
	Dn.WH = (Dn.WH + (((((Da.H * Dc.L) + (Da.L * Dd.H)) << 1) + 0x8000) >> 16))[19:0]
	Dn.WL = (Dn.WL + (((((Da.H * Dd.H) + (Da.L * Dd.L)) << 1) + 0x8000) >> 16))[19:0]	

11	**MACDRF.L.R.2T Da,Dc:Dd,Dn**	**SIMD2 fractional dot product and accumulate with rounding, using 3 low coefficients**

	Dn.WH = (Dn.WH + (((((Da.H * Dd.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[19:0]
	Dn.WL = (Dn.WL + (((((Da.H * Dd.L) + (Da.L * Dd.H)) << 1) + 0x8000) >> 16))[19:0]	

Table continues on the next page...

#	Syntax	Description
12	**MACDRF.L.R.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 fractional dot product and accumulate with rounding, using 3 low coefficients**

```
Dm.WH = (Dm.WH + (((((Da.H * Dd.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[19:0]
Dm.WL = (Dm.WL + (((((Da.H * Dd.L) + (Da.L * Dd.H)) << 1) + 0x8000) >> 16))[19:0]
Dn.WH = (Dn.WH + (((((Db.H * Dd.H) + (Db.L * Dc.L)) << 1) + 0x8000) >> 16))[19:0]
Dn.WL = (Dn.WL + (((((Db.H * Dd.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16))[19:0]
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \leq n \leq 62$
Dc:Dd	D_n:D_{n+1}	$0 \leq n \leq 62$
Dm:Dn	D_n:D_{n+1}	$0 \leq n \leq 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3, 4, 5, 6
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACDRF.nW 16x16 bit Dot Product and Accumulate Real by Real FIR into 16-bit Result with Saturation (DALU)

General Description

Performs SIMD2/SIMD4 16x16 bit dot product and accumulate into 16-bit results. Operands parts are optimized for Real by Real FIR implementation.

Flag Options

Flag	Position	Description
H	flg1	High 16-bit portion
L	flg1	Use the low portion
I	flg2	Integer arithmetic
R	flg2	Round multiplication result
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MACDRF.H.ISR.2W Da,Dc:Dd,Dn**	**SIMD2 Integer dot product and accumulate with rounding and saturation, using 3 high coefficients**

	`Dn.H = SAT16 ((Dn.H + ((((Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16)))[15:0]`
	`Dn.L = SAT16 ((Dn.L + ((((Da.H * Dd.H) + (Da.L * Dc.L)) + 0x8000) >> 16)))[15:0]`
	`Dn.E = 0`	

2	**MACDRF.H.ISR.4W Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 Integer dot product and accumulate with rounding and saturation, using 3 high coefficients**

	`Dm.H = SAT16 ((Dm.H + ((((Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16)))[15:0]`
	`Dm.L = SAT16 ((Dm.L + ((((Da.H * Dd.H) + (Da.L * Dc.L)) + 0x8000) >> 16)))[15:0]`
	`Dm.E = 0`
	`Dn.H = SAT16 ((Dn.H + ((((Db.H * Dc.L) + (Db.L * Dc.H)) + 0x8000) >> 16)))[15:0]`
	`Dn.L = SAT16 ((Dn.L + ((((Db.H * Dd.H) + (Db.L * Dc.L)) + 0x8000) >> 16)))[15:0]`
	`Dn.E = 0`	

3	**MACDRF.L.ISR.2W Da,Dc:Dd,Dn**	**SIMD2 Integer dot product and accumulate with rounding and saturation, using 3 low coefficients**

	`Dn.H = SAT16 ((Dn.H + ((((Da.H * Dd.H) + (Da.L * Dc.L)) + 0x8000) >> 16)))[15:0]`
	`Dn.L = SAT16 ((Dn.L + ((((Da.H * Dd.L) + (Da.L * Dd.H)) + 0x8000) >> 16)))[15:0]`
	`Dn.E = 0`	

Table continues on the next page...

#	Syntax	Description
4	**MACDRF.L.ISR.4W Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 Integer dot product and accumulate with rounding and saturation, using 3 low coefficients**

	`Dm.H = SAT16 ((Dm.H + ((((Da.H * Dd.H) + (Da.L * Dc.L)) + 0x8000) >> 16)))[15:0]`
	`Dm.L = SAT16 ((Dm.L + ((((Da.H * Dd.L) + (Da.L * Dd.H)) + 0x8000) >> 16)))[15:0]`
	`Dm.E = 0`
	`Dn.H = SAT16 ((Dn.H + ((((Db.H * Dd.H) + (Db.L * Dc.L)) + 0x8000) >> 16)))[15:0]`
	`Dn.L = SAT16 ((Dn.L + ((((Db.H * Dd.L) + (Db.L * Dd.H)) + 0x8000) >> 16)))[15:0]`
	`Dn.E = 0`	

5	**MACDRF.H.SR.2W Da,Dc:Dd,Dn**	**SIMD2 fractional dot product and accumulate with rounding and saturation, using 3 high coefficients**

	`Dn.H = SAT16 ((Dn.H + (((((Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.L = SAT16 ((Dn.L + (((((Da.H * Dd.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.E = 0`	

6	**MACDRF.H.SR.4W Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 fractional dot product and accumulate with rounding and saturation, using 3 high coefficients**

	`Dm.H = SAT16 ((Dm.H + (((((Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dm.L = SAT16 ((Dm.L + (((((Da.H * Dd.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dm.E = 0`
	`Dn.H = SAT16 ((Dn.H + (((((Db.H * Dc.L) + (Db.L * Dc.H)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.L = SAT16 ((Dn.L + (((((Db.H * Dd.H) + (Db.L * Dc.L)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.E = 0`	

7	**MACDRF.L.SR.2W Da,Dc:Dd,Dn**	**SIMD2 fractional dot product and accumulate with rounding and saturation, using 3 low coefficients**

	`Dn.H = SAT16 ((Dn.H + (((((Da.H * Dd.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.L = SAT16 ((Dn.L + (((((Da.H * Dd.L) + (Da.L * Dd.H)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.E = 0`	

8	**MACDRF.L.SR.4W Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 fractional dot product and accumulate with rounding and saturation, using 3 low coefficients**

	`Dm.H = SAT16 ((Dm.H + (((((Da.H * Dd.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dm.L = SAT16 ((Dm.L + (((((Da.H * Dd.L) + (Da.L * Dd.H)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dm.E = 0`
	`Dn.H = SAT16 ((Dn.H + (((((Db.H * Dd.H) + (Db.L * Dc.L)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.L = SAT16 ((Dn.L + (((((Db.H * Dd.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16)))[15:0]`
	`Dn.E = 0`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3, 4
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACDRF.nX 16x16 bit Dot Product and Accumulate Real by Real FIR into 40-bit Result (DALU)

General Description

Performs SIMD2 16x16 bit dot product and accumulate into 40-bit results. Operands parts are optimized for Real by Real FIR/correlation implementation.

Flag Options

Flag	Position	Description
H	flg1	High 16-bit portion
INV	flg1	Inverse order
L	flg1	Use the low portion
I	flg2	Integer arithmetic
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MACDRF.H.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product and accumulate, using 3 high coefficients**

	`Dm = (Dm + (((Da.H * Dc.L) + (Da.L * Dc.H)) << 1))[39:0]`
	`Dn = (Dn + (((Da.H * Dd.H) + (Da.L * Dc.L)) << 1))[39:0]`	

2	**MACDRF.INVH.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product and accumulate, using 3 high coefficients in inverse order**

	`Dm = (Dm + (((Da.H * Dc.H) + (Da.L * Dc.L)) << 1))[39:0]`
	`Dn = (Dn + (((Da.H * Dc.L) + (Da.L * Dd.H)) << 1))[39:0]`	

3	**MACDRF.INVL.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product and accumulate, using 3 low coefficients in inverse order**

	`Dm = (Dm + (((Da.H * Dc.L) + (Da.L * Dd.H)) << 1))[39:0]`
	`Dn = (Dn + (((Da.H * Dd.H) + (Da.L * Dd.L)) << 1))[39:0]`	

4	**MACDRF.L.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product and accumulate, using 3 low coefficients**

	`Dm = (Dm + (((Da.H * Dd.H) + (Da.L * Dc.L)) << 1))[39:0]`
	`Dn = (Dn + (((Da.H * Dd.L) + (Da.L * Dd.H)) << 1))[39:0]`	

5	**MACDRF.H.I.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product and accumulate, using 3 high coefficients**

	`Dm = (Dm + ((Da.H * Dc.L) + (Da.L * Dc.H)))[39:0]`
	`Dn = (Dn + ((Da.H * Dd.H) + (Da.L * Dc.L)))[39:0]`	

Table continues on the next page...

#	Syntax	Description
6	**MACDRF.INVH.I.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product and accumulate, using 3 high coefficients in inverse order**

	`Dm = (Dm + ((Da.H * Dc.H) + (Da.L * Dc.L)))[39:0]`
	`Dn = (Dn + ((Da.H * Dc.L) + (Da.L * Dd.H)))[39:0]`	

7	**MACDRF.INVL.I.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product and accumulate, using 3 low coefficients in inverse order**

	`Dm = (Dm + ((Da.H * Dc.L) + (Da.L * Dd.H)))[39:0]`
	`Dn = (Dn + ((Da.H * Dd.H) + (Da.L * Dd.L)))[39:0]`	

8	**MACDRF.L.I.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product and accumulate, using 3 low coefficients**

	`Dm = (Dm + ((Da.H * Dd.H) + (Da.L * Dc.L)))[39:0]`
	`Dn = (Dn + ((Da.H * Dd.L) + (Da.L * Dd.H)))[39:0]`	

9	**MACDRF.H.IS.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product and accumulate with saturation, using 3 high coefficients**

	`Dm = SAT32((Dm + ((Da.H * Dc.L) + (Da.L * Dc.H))))`
	`Dn = SAT32((Dn + ((Da.H * Dd.H) + (Da.L * Dc.L))))`	

10	**MACDRF.L.IS.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product and accumulate with saturation, using 3 low coefficients**

	`Dm = SAT32((Dm + ((Da.H * Dd.H) + (Da.L * Dc.L))))`
	`Dn = SAT32((Dn + ((Da.H * Dd.L) + (Da.L * Dd.H))))`	

11	**MACDRF.H.S.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product and accumulate with saturation, using 3 high coefficients**

	`Dm = SAT32((Dm + (((Da.H * Dc.L) + (Da.L * Dc.H)) << 1)))`
	`Dn = SAT32((Dn + (((Da.H * Dd.H) + (Da.L * Dc.L)) << 1)))`	

12	**MACDRF.L.S.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product and accumulate with saturation, using 3 low coefficients**

	`Dm = SAT32((Dm + (((Da.H * Dd.H) + (Da.L * Dc.L)) << 1)))`
	`Dn = SAT32((Dn + (((Da.H * Dd.L) + (Da.L * Dd.H)) << 1)))`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Affected By Instructions

Field	Relevant Variants
SR.SAT	9, 10, 11, 12

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	5, 6, 7, 8, 9, 10
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACEM32.nX Double Precision Multiply Assist (DALU)

General Description

Performs 16x16 multiply-accumulate operation intended for higher precision multiplication emulation assist. Usually intended for legacy support, as the supports native 32x32 bit operations.

Flag Options

Flag	Position	Description
SS	flg1	Signed by Signed
SU	flg1	Signed By Unsigned
US	flg1	Unsigned By Signed
UU	flg1	Unsigned By Unsigned
X	flg1	Cross between the arguments
I	flg2	Integer arithmetic

Instruction Variants

#	Syntax	Description
1	**MACEM32.SS.X Da.h,Db.h,Dn**	**Fractional signed by signed multiply accumulate with accumulator shifted by 16 prior to addition**
	`Dn = (((Dn >> 16)[23:0]) + ((Da.H * Db.H) << 1))[39:0]`	
2	**MACEM32.SU.X Da.h,Db.l,Dn**	**Fractional signed by unsigned multiply accumulate with accumulator shifted by 16 prior to addition**
	`Dn = (((Dn >> 16)[23:0]) + ((Da.H * Db.L) << 1))[39:0]`	
3	**MACEM32.US.X Da.l,Db.h,Dn**	**Fractional unsigned by signed multiply accumulate**
	`Dn = (Dn + ((Da.L * Db.H) << 1))[39:0]`	
4	**MACEM32.UU.X Da.l,Db.l,Dn**	**Fractional unsigned by unsigned multiply accumulate**
	`Dn = (Dn + (((U40)Da.L * (U40)Db.L) << 1))[39:0]`	
5	**MACEM32.XSU.X Da,Db,Dn**	**Fractional dot product, unsigned by signed and signed by unsigned and accumulate**
	`Dn = (Dn + (((Da.H * Db.L) + (Da.L * Db.H)) << 1))[39:0]`	
6	**MACEM32.XSU.X -Da,Db,Dn**	**Fractional dot product, unsigned by signed and signed by unsigned and accumulate with negate**
	`Dn = (Dn + (((-(Da.H * Db.L)) - (Da.L * Db.H)) << 1))[39:0]`	
7	**MACEM32.US.I.X Da.l,Db.h,Dn**	**Integer unsigned by signed multiply accumulate**

Table continues on the next page...

#	Syntax	Description
	`Dn = (Dn + ((Da.L * Db.H)))[39:0]`	
8	**MACEM32.UU.I.X Da.l,Db.h,Dn**	**Integer unsigned by unsigned multiply accumulate**
	`Dn = (Dn + (Da.L * Db.H))[39:0]`	

Explicit Operands

Operand	Permitted Values
Da	`D0-D63`
Db	`D0-D63`
Dn	`D0-D63`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACM.nL 16x32 bit Mixed Precision Multiply-Accumulate into 32-bit Result (DALU)

General Description

Performs unsigned 16-bit X signed 32-bit C-like integer multiply-accumulate, into a 32-bit result. C-like integer meaning the low part of the result is written to the destination register.

Flag Options

Flag	Position	Description
US	flg1	Unsigned By Signed
I	flg2	Integer arithmetic
L	flg2	32-bit operation

Instruction Variants

#	Syntax	Description
1	**MACM.US.IL.L Da.l,Db,Dn**	**Integer (C-like) mixed precision (unsigned 16 x signed 32 bit) multiply-accumulate**
	`Dn = (U40)((S64)Dn + (S64)((U64)Da.L * (S64)Db.M)[31:0])[31:0]`	

Explicit Operands

Operand	Permitted Values
Da	`D0-D63`
Db	`D0-D63`
Dn	`D0-D63`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACM.nX 16x32 bit Mixed Precision Multiply-Accumulate into 40-bit Result (DALU)

General Description

Performs 16x32 bit multiply-accumulate, into 40-bit results.

Flag Options

Flag	Position	Description
SU	flg1	Signed By Unsigned
I	flg2	Integer arithmetic
LOG2	flg2	Log base 2 calculation assist
R	flg2	Round multiplication result
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MACM.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 fractional multiply-accumulate**

	`Dm = (Dm + (((Da.H * Dc.M) << 1) >> 16))[39:0]`
	`Dn = (Dn + (((Da.L * Dd.M) << 1) >> 16))[39:0]`	

2	**MACM.2X -Da,Dc:Dd,Dm:Dn**	**SIMD2 fractional multiply-accumulate with negate**

	`Dm = ((S64)Dm + ((-(((S64)Da.H * (S64)Dc.M) << 1)) >> 16))[39:0]`
	`Dn = ((S64)Dn + ((-(((S64)Da.L * (S64)Dd.M) << 1)) >> 16))[39:0]`	

3	**MACM.X Da.h,Db,Dn**	**Fractional multiply-accumulate, using high part**
	`Dn = ((S64)Dn + ((((S64)Da.H * (S64)Db.M) << 1) >> 16))[39:0]`	
4	**MACM.SU.X Da.h,Db,Dn**	**Fractional signed 16-bit by unsigned 32-bit multiply-accumulate**
	`Dn = ((S64)Dn + ((((S64)Da.H * (U64)Db.M) << 1) >> 16))[39:0]`	
5	**MACM.SU.X -Da.h,Db,Dn**	**Fractional signed 16-bit by unsigned 32-bit multiply-accumulate with negate**
	`Dn = ((S64)Dn + (((-((S64)Da.H * (U64)Db.M)) << 1) >> 16))[39:0]`	
6	**MACM.IR.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 integer multiply-accumulate with rounding**

	`Dm = (Dm + (((S48)((S48)Da.H * (S48)Dc.M) + 0x8000) >> 16))[39:0]`
	`Dn = (Dn + (((S48)((S48)Da.L * (S48)Dd.M) + 0x8000) >> 16))[39:0]`	

7	**MACM.IR.2X Da.h,Dc:Dd,Dm:Dn**	**SIMD2 by one value - integer multiply-accumulate with rounding, using high part**

	`Dm = ((S64)Dm + ((((S64)Da.H * (S64)Dc.M) + 0x8000) >> 16))[39:0]`
	`Dn = ((S64)Dn + ((((S64)Da.H * (S64)Dd.M) + 0x8000) >> 16))[39:0]`	

Table continues on the next page...

#	Syntax	Description
8	**MACM.IR.2X Da.l,Dc:Dd,Dm:Dn**	**SIMD2 by one value - integer multiply-accumulate with rounding, using low part**

	`Dm = ((S64)Dm + ((((S64)Da.L * (S64)Dc.M) + 0x8000) >> 16))[39:0]`
	`Dn = ((S64)Dn + ((((S64)Da.L * (S64)Dd.M) + 0x8000) >> 16))[39:0]`	

9	**MACM.IR.X Da.h,Db,Dn**	**Integer multiply-accumulate with rounding, using high part**
	`Dn = (Dn + ((((S48)Da.H * (S48)Db.M) + 0x8000) >> 16))[39:0]`	
10	**MACM.IR.X Da.l,Db,Dn**	**Integer multiply-accumulate with rounding, using low part**
	`Dn = (Dn + ((((S48)Da.L * (S48)Db.M) + 0x8000) >> 16))[39:0]`	
11	**MACM.ISR.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 integer multiply-accumulate with rounding and saturation**

	`Dm = SAT32(((S64)Dm + ((((S64)Da.H * (S64)Dc.M) + 0x8000) >> 16)))`
	`Dn = SAT32(((S64)Dn + ((((S64)Da.L * (S64)Dd.M) + 0x8000) >> 16)))`	

12	**MACM.ISR.2X Da.h,Dc:Dd,Dm:Dn**	**SIMD2 by one value - integer multiply-accumulate with rounding and saturation, using high part**

	`Dm = SAT32(((S64)Dm + ((((S64)Da.H * (S64)Dc.M) + 0x8000) >> 16)))`
	`Dn = SAT32(((S64)Dn + ((((S64)Da.H * (S64)Dd.M) + 0x8000) >> 16)))`	

13	**MACM.ISR.2X Da.l,Dc:Dd,Dm:Dn**	**SIMD2 by one value - integer multiply-accumulate with rounding and saturation, using low part**

	`Dm = SAT32(((S64)Dm + ((((S64)Da.L * (S64)Dc.M) + 0x8000) >> 16)))`
	`Dn = SAT32(((S64)Dn + ((((S64)Da.L * (S64)Dd.M) + 0x8000) >> 16)))`	

14	**MACM.SU.LOG2.X Da.h,Db,Dn**	**Fractional signed 16-bit by unsigned 32-bit multiply-accumulate. Product is multiplied by 2 prior to accumulation for short Log2 calculation sequence. See LOG2 instructions for usage.**
	`Dn = ((S64)Dn + ((((S64)Da.H * (U64)Db.M) << 2) >> 16))[39:0]`	
15	**MACM.R.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 fractional multiply-accumulate with rounding**

	`Dm = (Dm + ((((Da.H * Dc.M) << 1) + 0x8000) >> 16))[39:0]`
	`Dn = (Dn + ((((Da.L * Dd.M) << 1) + 0x8000) >> 16))[39:0]`	

16	**MACM.R.2X Da.h,Dc:Dd,Dm:Dn**	**SIMD2 by one value - fractional multiply-accumulate with rounding, using high part**

	`Dm = ((S64)Dm + (((((S64)Da.H * (S64)Dc.M) << 1) + 0x8000) >> 16))[39:0]`
	`Dn = ((S64)Dn + (((((S64)Da.H * (S64)Dd.M) << 1) + 0x8000) >> 16))[39:0]`	

17	**MACM.R.2X Da.l,Dc:Dd,Dm:Dn**	**SIMD2 by one value - fractional multiply-accumulate with rounding, using low part**

	`Dm = (Dm + ((((Da.L * Dc.M) << 1) + 0x8000) >> 16))[39:0]`
	`Dn = (Dn + ((((Da.L * Dd.M) << 1) + 0x8000) >> 16))[39:0]`	

Table continues on the next page...

#	Syntax	Description
18	**MACM.R.X Da.h,Db,Dn**	**Fractional multiply-accumulate with rounding, using high part**
	`Dn = ((S64)Dn + ((((S64)Da.H * (S64)Db.M) << 1) + 0x8000) >> 16))[39:0]`	
19	**MACM.R.X Da.l,Db,Dn**	**Fractional multiply-accumulate with rounding, using low part**
	`Dn = ((S64)Dn + ((((S64)Da.L * (S64)Db.M) << 1) + 0x8000) >> 16))[39:0]`	
20	**MACM.R.X -Da.h,Db,Dn**	**Fractional multiply-accumulate with negate and rounding, using high part**
	`Dn = ((S64)Dn + (((-(((S64)Da.H * (S64)Db.M) << 1)) + 0x8000) >> 16))[39:0]`	
21	**MACM.SR.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 fractional multiply-accumulate with rounding and saturation**

	`Dm = SAT32((Dm + ((((Da.H * Dc.M) << 1) + 0x8000) >> 16)))`
	`Dn = SAT32((Dn + ((((Da.L * Dd.M) << 1) + 0x8000) >> 16)))`	

22	**MACM.SR.2X Da.h,Dc:Dd,Dm:Dn**	**SIMD2 by one value - fractional multiply-accumulate with rounding and saturation, using high part**

	`Dm = SAT32(((S64)Dm + ((((S64)Da.H * (S64)Dc.M) << 1) + 0x8000) >> 16)))`
	`Dn = SAT32(((S64)Dn + ((((S64)Da.H * (S64)Dd.M) << 1) + 0x8000) >> 16)))`	

23	**MACM.SR.2X Da.l,Dc:Dd,Dm:Dn**	**SIMD2 by one value - fractional multiply-accumulate with rounding and saturation, using low part**

	`Dm = SAT32(((S64)Dm + ((((S64)Da.L * (S64)Dc.M) << 1) + 0x8000) >> 16)))`
	`Dn = SAT32(((S64)Dn + ((((S64)Da.L * (S64)Dd.M) << 1) + 0x8000) >> 16)))`	

24	**MACM.SR.X Da.h,Db,Dn**	**Fractional multiply-accumulate with rounding and saturation, using high part**
	`Dn = SAT32(((S64)Dn + (((Da.H * Db.M) << 1) + 0x8000) >> 16))))`	
25	**MACM.SR.X Da.l,Db,Dn**	**Fractional multiply-accumulate with rounding and saturation, using low part**
	`Dn = SAT32(((S64)Dn + ((((S64)Da.L * (S64)Db.M) << 1) + 0x8000) >> 16))))`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
SR.SAT	11, 12, 13, 21, 22, 23, 24, 25

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 6, 7, 8, 9, 10, 11, 12, 13
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACQ.nX — 16x16 bit Quad Dot Product and Accumulate into 40-bit Result — (DALU)

General Description

Performs 16x16 bit quad dot product and accumulate into 40-bit result.

Flag Options

Flag	Position	Description
I	flg2	Integer arithmetic

Instruction Variants

#	Syntax	Description
1	**MACQ.X Da:Db,Dc:Dd,Dn**	**Fractional quad multiply and accumulate**
	`Dn = (Dn + (((Da.H * Dc.H) + (Da.L * Dc.L) + (Db.H * Dd.H) + (Db.L * Dd.L)) << 1))[39:0]`	
2	**MACQ.I.X Da:Db,Dc:Dd,Dn**	**Integer quad multiply and accumulate**
	`Dn = (Dn + ((Da.H * Dc.H) + (Da.L * Dc.L) + (Db.H * Dd.H) + (Db.L * Dd.L)))[39:0]`	

Explicit Operands

Operand	Permitted Values	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACRC.nT 16x16 bit Real by Complex Multiply-Accumulate into 20-bit Result (DALU)

General Description

Performs SIMD4 16x16 Real by Complex multiply-accumulate into 20-bit results.

Flag Options

Flag	Position	Description
I	flg2	Integer arithmetic
R	flg2	Round multiplication result

Instruction Variants

#	Syntax	Description
1	**MACRC.IR.4T Da,Dc:Dd,Dm:Dn**	**SIMD4 integer real by complex multiply-accumulate with rounding**

```
Dm.WH = (Dm.WH + (((Da.H * Dc.H) + 0x8000) >> 16))[19:0]
Dm.WL = (Dm.WL + (((Da.H * Dc.L) + 0x8000) >> 16))[19:0]
Dn.WH = (Dn.WH + (((Da.L * Dd.H) + 0x8000) >> 16))[19:0]
Dn.WL = (Dn.WL + (((Da.L * Dd.L) + 0x8000) >> 16))[19:0]
```

#	Syntax	Description
2	**MACRC.R.4T Da,Dc:Dd,Dm:Dn**	**SIMD4 fractional real by complex multiply-accumulate with rounding**

```
Dm.WH = (Dm.WH + ((((Da.H * Dc.H) << 1) + 0x8000) >> 16))[19:0]
Dm.WL = (Dm.WL + ((((Da.H * Dc.L) << 1) + 0x8000) >> 16))[19:0]
Dn.WH = (Dn.WH + ((((Da.L * Dd.H) << 1) + 0x8000) >> 16))[19:0]
Dn.WL = (Dn.WL + ((((Da.L * Dd.L) << 1) + 0x8000) >> 16))[19:0]
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACRC.nW 16x16 bit Real by Complex Multiply-Accumulate into 16-bit Result (DALU)

General Description

Performs SIMD4 16x16 Real by Complex multiply-accumulate into 16-bit results, with rounding and saturation.

Flag Options

Flag	Position	Description
I	flg2	Integer arithmetic
R	flg2	Round multiplication result
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MACRC.ISR.4W Da,Dc:Dd,Dm:Dn**	**SIMD4 integer real by complex multiply-accumulate with rounding and saturation**

```
Dm.H = SAT16 ((Dm.H + (((Da.H * Dc.H) + 0x8000) >> 16)))[15:0]
Dm.L = SAT16 ((Dm.L + (((Da.H * Dc.L) + 0x8000) >> 16)))[15:0]
Dm.E = 0
Dn.H = SAT16 ((Dn.H + (((Da.L * Dd.H) + 0x8000) >> 16)))[15:0]
Dn.L = SAT16 ((Dn.L + (((Da.L * Dd.L) + 0x8000) >> 16)))[15:0]
Dn.E = 0
```

2	**MACRC.SR.4W Da,Dc:Dd,Dm:Dn**	**SIMD4 fractional real by complex multiply-accumulate with rounding and saturation**

```
Dm.H = SAT16 ((Dm.H + ((((Da.H * Dc.H) << 1) + 0x8000) >> 16)))[15:0]
Dm.L = SAT16 ((Dm.L + ((((Da.H * Dc.L) << 1) + 0x8000) >> 16)))[15:0]
Dm.E = 0
Dn.H = SAT16 ((Dn.H + ((((Da.L * Dd.H) << 1) + 0x8000) >> 16)))[15:0]
Dn.L = SAT16 ((Dn.L + ((((Da.L * Dd.L) << 1) + 0x8000) >> 16)))[15:0]
Dn.E = 0
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MACSHL.nX 16x16 bit Multiply-Accumulate with Shift Left (DALU)

General Description

Performs fractional 16x16 multiply accumulate with shift left and saturate.

Flag Options

Flag	Position	Description
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MACSHL.S.X Da.h,Db.h,Dn**	**Fractional Multiply-accumulate with saturation. Product is shifted left prior to accumulation.**
	`Dn = SAT32((Dn + ((Da.H * Db.H) << 2)))`	

Explicit Operands

Operand	Permitted Values
Da	`D0-D63`
Db	`D0-D63`
Dn	`D0-D63`

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY_Acc	All

MASKSEL.nX Bitwise Selection According to a Mask (DALU)

General Description

Per each bit, if the respective mask bit is set, the destination is updated with the respective bit from a new source. If the mask bit is clear, the original value of the destination remains unchanged.

Instruction Variants

#	Syntax	Description
1	**MASKSEL.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 bitwise mask select: The mask is in Da:Db, the new source is in Dc:Dd, and the old source which may be updated is in Dm:Dn**

	`Dm = (Da & Dc) │ ((~ Da) & Dm)`
	`Dn = (Db & Dd) │ ((~ Db) & Dn)`	

2	**MASKSEL.X Da,Db,Dn**	**Bitwise mask select: The mask is in Da, the new source is in Db, and the old source which may be updated is in Dn**
	`Dn = (Da & Db) │ ((~ Da) & Dn)`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM_MASKSEL	All

MAX.nT Maximum Value of 20-bit Numbers (DALU)

General Description

Writes the larger of the signed 20-bit values in each of the corresponding WH/WL portions of the source operands to the respective portion in the result data register.

In the SIMD2 variant, independently compares the two WH and WL portions in the two source registers.
In the SIMD4 variant, independently compares four portions of Da:Db with those in Dc:Dd.

Instruction Variants

#	Syntax	Description
1	**MAX.2T Da,Db,Dn**	**SIMD2 20-bit maximum value**

	`Dn.WH = (Da.WH > Db.WH) ? Da.WH : Db.WH`
	`Dn.WL = (Da.WL > Db.WL) ? Da.WL : Db.WL`	

2	**MAX.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 20-bit maximum value**

	`Dm.WH = (Da.WH > Dc.WH) ? Da.WH : Dc.WH`
	`Dm.WL = (Da.WL > Dc.WL) ? Da.WL : Dc.WL`
	`Dn.WH = (Db.WH > Dd.WH) ? Db.WH : Dd.WH`
	`Dn.WL = (Db.WL > Dd.WL) ? Db.WL : Dd.WL`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

MAX.nW Maximum Value of 16-bit/20-bit Numbers (DALU)

General Description

Writes the larger of the signed 20-bit or 16-bit values, in each of the corresponding portions of the source operands to the respective portion in the result data register.

The SIMD2 variant depends on SR.W20. If it is set, independently compares the two WH and WL portions in the two source registers. If it is celar, compares the two H and L portions of the source register, and zeroes the extension.
The SIMD4 variant compares 20-bit portions without being affected by SR.W20, and independently compares four portions of Da:Db with those in Dc:Dd.

Instruction Variants

#	Syntax	Description
1	**MAX.2W Da,Db,Dn**	**SIMD2 16-bit/20-bit maximum value**

```
if (SR.W20 == 0) {
        Dn.H = (Da.H  >  Db.H) ? Da.H : Db.H
        Dn.L = (Da.L  >  Db.L) ? Da.L : Db.L
        Dn.E = 0
} else {
        Dn.WH = (Da.WH  >  Db.WH) ? Da.WH : Db.WH
        Dn.WL = (Da.WL  >  Db.WL) ? Da.WL : Db.WL
}
```

#	Syntax	Description
2	**MAX.4W Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 16-bit/20-bit maximum value**

```
Dm.WH = (Da.H  >  Dc.H) ? (U20)Da.H : (U20)Dc.H
Dm.WL = (Da.L  >  Dc.L) ? (U20)Da.L : (U20)Dc.L
Dn.WH = (Db.H  >  Dd.H) ? (U20)Db.H : (U20)Dd.H
Dn.WL = (Db.L  >  Dd.L) ? (U20)Db.L : (U20)Dd.L
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Affect Instructions

Field	Relevant Variants	Comments
SR.W20	1	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

MAX.nX Maximum Value of 40-bit Numbers (DALU)

General Description

Writes the larger of two signed 40-bit values from two source data registers to the result register. The SIMD2 variant performs this operation independently on two source register pairs.

Instruction Variants

#	Syntax	Operation	Description

1	MAX.2X Da:Db,Dc:Dd,Dm:Dn	`if (Da > Dc) {`
		`        Dm = Da`
		`} else {`
		`        Dm = Dc`
		`}`
		`if (Db > Dd) {`
		`        Dn = Db`
		`} else {`
		`        Dn = Dd`
		`}`
			SIMD2 Maximum Value of 40-bit numbers

2	MAX.X Da,Db,Dn	`Dn = Db > Da ? Db : Da`	Maximum Value of 40-bit numbers

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

MAXA Maximum Value of 32-bit Numbers (LSU,IPU)

General Description

Writes the larger of two signed values in a pair of source data registers to the destination register.

Instruction Variants

#	Syntax	Operation	Description

1	MAXA Ra,Rb,Rn	`if (Ra > Rb) {`
		`        Rn = Ra`
		`} else {`
		`        Rn = Rb`
		`}`
			Maximum Value of 32-bit numbers

Explicit Operands

Operand	Permitted Values
Ra	R0-R31
Rb	R0-R31
Rn	R0-R31

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Execution unit	IPU	All
	LSU	All
Pipeline behavior	agu_AAU_LOGIC	All

MAXM.nX Maximum Magnitude of 40-bit Numbers (DALU)

General Description

Compares the absolute values (or magnitude) of two signed 40-bit values from two data registers. Writes the signed value with the larger magniture to the result register. If the values in the two registers have equal magnitudes but opposite signs, writes the positive value to the result register.
The SIMD2 variant performs this operation independently on two source register pairs.

Instruction Variants

#	Syntax	Description
1	**MAXM.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 Maximum Magnitude of 40-bit numbers**

```
If ( |Da| > |Dc| ) then Dm = Da
If ( |Dc| > |Da| ) then Dm = Dc
if ( Da == -Dc) then Dm = |Da|
if ( Da == Dc ) then Dm = Da
If ( |Db| > |Dd| ) then Dn = Db
If ( |Dd| > |Db| ) then Dn = Dd
if ( Db == -Dd) then Dn = |Db|
if ( Db == Dd ) then Dn = Db
```

#	Syntax	Description
2	**MAXM.X Da,Db,Dn**	**Maximum Magnitude of 40-bit numbers**

```
if (|Da| > |Db|) Dn = Da
if (|Db| > |Da|) Dn = Db
if (Da == -Db) Dn = |Da|
if (Da == Db) Dn = Da
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

MIN.nT Minimum Value of 20-bit Numbers (DALU)

General Description

Writes the smaller of the signed 20-bit values in each of the corresponding WH/WL portions of the source operands to the respective portion in the result data register.

In the SIMD2 variant, independently compares the two WH and WL portions in the two source registers.
In the SIMD4 variant, independently compares four portions of Da:Db with those in Dc:Dd.

Instruction Variants

#	Syntax	Description
1	**MIN.2T Da,Db,Dn**	**SIMD2 20-bit minimum value**

	`Dn.WH = (Da.WH  <  Db.WH) ? Da.WH : Db.WH`
	`Dn.WL = (Da.WL  <  Db.WL) ? Da.WL : Db.WL`	

2	**MIN.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 20-bit minimum value**

	`Dm.WH = (Da.WH  <  Dc.WH) ? Da.WH : Dc.WH`
	`Dm.WL = (Da.WL  <  Dc.WL) ? Da.WL : Dc.WL`
	`Dn.WH = (Db.WH  <  Dd.WH) ? Db.WH : Dd.WH`
	`Dn.WL = (Db.WL  <  Dd.WL) ? Db.WL : Dd.WL`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

MIN.nW Maximum Value of 16-bit/20-bit Numbers (DALU)

General Description

Writes the smaller of the signed 20-bit or 16-bit values, in each of the corresponding portions of the source operands to the respective portion in the result data register.

The SIMD2 variant depends on SR.W20. If it is set, independently compares the two WH and WL portions in the two source registers. If it is celar, compares the two H and L portions of the source register, and zeroes the extension.
The SIMD4 variant compares 20-bit portions without being affected by SR.W20, and independently compares four portions of Da:Db with those in Dc:Dd.

Instruction Variants

#	Syntax	Description
1	**MIN.2W Da,Db,Dn**	**SIMD2 16-bit/20-bit minimum value**

```
if (SR.W20 == 0) {
        Dn.H = (Da.H  <  Db.H) ? Da.H : Db.H
        Dn.L = (Da.L  <  Db.L) ? Da.L : Db.L
        Dn.E = 0
} else {
        Dn.WH = (Da.WH  <  Db.WH) ? Da.WH : Db.WH
        Dn.WL = (Da.WL  <  Db.WL) ? Da.WL : Db.WL
}
```

#	Syntax	Description
2	**MIN.4W Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 16-bit/20-bit minimum value**

```
Dm.WH = (Da.H  <  Dc.H) ? (U20)Da.H : (U20)Dc.H
Dm.WL = (Da.L  <  Dc.L) ? (U20)Da.L : (U20)Dc.L
Dn.WH = (Db.H  <  Dd.H) ? (U20)Db.H : (U20)Dd.H
Dn.WL = (Db.L  <  Dd.L) ? (U20)Db.L : (U20)Dd.L
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Affect Instructions

Field	Relevant Variants	Comments
SR.W20	1	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

MIN.nX Minimum Value of 40-bit Numbers (DALU)

General Description

Writes the smaller of two signed 40-bit values from two source data registers to the result register. The SIMD2 variant performs this operation independently on two source register pairs.

Instruction Variants

#	Syntax	Operation	Description

1	MIN.2X Da:Db,Dc:Dd,Dm:Dn	`if (Da < Dc) {`
		`        Dm = Da`
		`} else {`
		`        Dm = Dc`
		`}`
		`if (Db < Dd) {`
		`        Dn = Db`
		`} else {`
		`        Dn = Dd`
		`}`
			SIMD2 Minimum Value of 40-bit numbers

2	MIN.X Da,Db,Dn	`Dn = Db < Da ? Db : Da`	Minimum Value of 40-bit numbers

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

MINA Minimum Value of 32-bit Numbers (LSU,IPU)

General Description

Writes the smaller of two signed values in a pair of source data registers to the destination register.

Instruction Variants

#	Syntax	Operation	Description

1	MINA Ra,Rb,Rn	`if (Ra < Rb) {`
		`        Rn = Ra`
		`} else {`
		`        Rn = Rb`
		`}`
			Minimum Value of 32-bit numbers

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`
Rb	`R0-R31`
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Execution unit	IPU	All
	LSU	All
Pipeline behavior	agu_AAU_LOGIC	All

MOVE.IMM Move an immediate value to a D register (LSU,IPU)

General Description

Move a value specified as an immediate operand to a D register. Variants update the whole D register or only a portion of it.

Flag Options

Flag	Position	Description
PAR	flg1	Partial (reminder of the register remains unchanged)

Instruction Variants

#	Syntax	Operation	Description
1	MOVE.L #s32,Dn	`Dn = (S40)s32`	The signed 32-bit value is sign extended in the destination Dn.
2	MOVE.L #s9,Dn	`Dn = (S40)s9`	The signed 9-bit value is sign extended in the destination Dn.

3	MOVE.PAR.W #u16,Dn.h	`Dn = Dn`
		`Dn.H = u16`
			Partial update: write Dn.H with the unsigned value. The other portions of Dn are not changed.

4	MOVE.PAR.W #u16,Dn.l	`Dn = Dn`
		`Dn.L = u16`
			Partial update: write Dn.L with the unsigned value. The other portions of Dn are not changed.

Explicit Operands

Operand	Permitted Values
Dn	D0-D63
s32	$-2^{31} \le s32 < 2^{31}$
s9	$-2^{8} \le s9 < 2^{8}$
u16	$0 \le u16 < 2^{16}$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	2
	48-bits	1, 3, 4
Execution unit	IPU	1, 2
	LSU	All
Pipeline behavior	agu_MOVE_REG_INIT_IMM	1, 2
	agu_MOVE_REG_INIT_IMM_PARTIALW	3, 4

MOVE.R2R Move a register to a register (LSU,IPU)

General Description

Copies values between registers, where one of them at least is a D register. Some variants allow to move several D registers at once. These are AGU instructions, which could execute in parallel with DALU instructions. The ability to group instructions from this group with other MOVE or load/store instructions is affected by register file bandwidth limitations of the AGU, which are describe in programming rule A.11

Instruction Variants

#	Syntax	Operation	Description

1	MOVE.2L Da:Db,Dm:Dn	`Dm = (S40)Da.M`
		`Dn = (S40)Db.M`
			Move 32-bits from each register of a data register pair to a data register pair. The destinations are sign-extended.

2	MOVE.2L Da:Db,Rm,Rn	`Rm = Da.M`
		`Rn = Db.M`
			Move from a data register pair to an R register pair. The D registers are consecutive, the R registers are freely selected.

3	MOVE.2L Da:Db,Rm:Rn	`Rm = Da.M`
		`Rn = Db.M`
			Move from a data register pair to an R register pair. Both R and D registers are consecutive. This variant has shorter encoding than the ones where the R registers are freely selected.

4	MOVE.2X Da:Db,Dm:Dn	`Dm = Da`
		`Dn = Db`
			Move two full D registers (40 bits each) from a data register pair to a data register pair

5	MOVE.2X Da,Dm:Dn	`Dm = Da`
		`Dn = Da`
			Duplicate the 40-bit value of a data register (Da) into a data register pair (Dm:Dn)

6	MOVE.4X Da,Db,Dc,Dd,Dm:Dn:Do:Dp	`Dm = Da`
		`Dn = Db`
		`Do = Dc`
		`Dp = Dd`
			Move four full D registers (40 bits) to a data register quad. Da,Db,Dc,Dd are freely selected from within a 4 register sliding window (ex: d5,d3,d6,d6 are legal for a window of d3::d6). Dm:Dn:Do:Dp are a sliding window of 4 registers.

7	MOVE.L Da,Dn	`Dn = (S40)Da.M`	Move 32-bits from a data register to a data register. The destination is sign-extended.
8	MOVE.L Da,Rn	`Rn = Da.M`	Move 32-bits from a data register to an address register.
9	MOVE.L Da,TMDAT	`TMDAT = Da.M`	Move 32-bits from a data register to the write-only TMDAT register. If tracing is enabled and properly configured, this move will trigger a Data Acquisition trace message. There are also some other debug-related side effects. Otherwise, this move has no effect.

Table continues on the next page...

#	Syntax	Operation	Description
10	MOVE.L Ra,Dn	`Dn = (S40)Ra`	Move 32-bits from an address register to a data register. The destination is sign-extended.
11	MOVE.X Da,Dn	`Dn = Da`	Move a full data register (40 bits) to a data register.

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Da:Db	D_n:D_{n+1} 0 ≤ n ≤ 62
Da,Db,Dc,Dd	Any four registers from a quartet: D_n,D_{n+1},D_{n+2},D_{n+3} (0 ≤ n ≤ 60). For example: D6,D6,D3,D4 from the quartet: D3:D4:D5:D6
Dm:Dn	D_n:D_{n+1} 0 ≤ n ≤ 62
Dm:Dn:Do:Dp	D_n:D_{n+1}:D_{n+2}:D_{n+3} 0 ≤ n ≤ 60
Dn	D0-D63
Ra	R0-R31
Rm	R0-R31
Rm:Rn	R_n:R_{n+1} 0 ≤ n ≤ 30
Rn	R0-R31

Affect Instructions

Field	Relevant Variants	Comments
D	6	

Affected By Instructions

Field	Relevant Variants
TMDAT	9

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	3
Encoding length	32-bits	1, 3, 4, 5, 6, 7, 8, 9, 10, 11
	48-bits	2
Execution unit	IPU	2, 3, 8, 9, 10
	LSU	1, 4, 5, 6, 7, 8, 10, 11
Pipeline behavior	agu_MOVE_REG_INIT	2, 3, 8, 9, 10
	agu_MOVE_REG_INIT_DD	1, 4, 5, 6, 7, 11

MPY.LEG.X 16x16 bit Multiply into 40-bit Result in Legacy Mode (DALU)

General Description

Performs single/SIMD2 16x16 bit multiply into 40-bit results in legacy mode, i.e., saturation and rounding is affected by SR mode bits: SM, RM, and SCM.

Flag Options

Flag	Position	Description
LEG	flg2	Legacy instruction
R	flg2	Round multiplication result

Instruction Variants

#	Syntax	Description
1	**MPY.LEG.X Da.h,Db.h,Dn**	**Single 16x16 bit fractional multiply, saturating only if SR.SM is set**
	`Dn = srSAT32(((Da.H * Db.H) << 1))`	
2	**MPY.LEG.X Da.h,Db.l,Dn**	**Single 16x16 bit fractional multiply, saturating only if SR.SM is set**
	`Dn = srSAT32(((Da.H * Db.L) << 1))`	
3	**MPY.LEG.X Da.l,Db.l,Dn**	**Single 16x16 bit fractional multiply, saturating only if SR.SM is set**
	`Dn = srSAT32(((Da.L * Db.L) << 1))`	
4	**MPY.RLEG.X Da.h,Db.h,Dn**	**Single 16x16 bit fractional multiply, legacy rounding according to SR.RM and SR.SCM**
	`Dn = srRND (((Da.H * Db.H) << 1))`	
5	**MPY.RLEG.X Da.l,Db.h,Dn**	**Single 16x16 bit fractional multiply, legacy rounding according to SR.RM and SR.SCM**
	`Dn = srRND (((Da.L * Db.H) << 1))`	
6	**MPY.RLEG.X Da.l,Db.l,Dn**	**Single 16x16 bit fractional multiply, legacy rounding according to SR.RM and SR.SCM**
	`Dn = srRND (((Da.L * Db.L) << 1))`	

Explicit Operands

Operand	Permitted Values
Da	`D0-D63`
Db	`D0-D63`
Dn	`D0-D63`

Affect Instructions

Field	Relevant Variants	Comments
SR.RM	4, 5, 6	
SR.SCM	4, 5, 6	
SR.SM	All	

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPY.nT 16x16 bit Multiply into 20-bit Results (DALU)

General Description

Performs SIMD2/SIMD4 16x16 bit multiply into 20-bit results.

Flag Options

Flag	Position	Description
L	flg1	Use the low portion
SU	flg1	Signed By Unsigned
I	flg2	Integer arithmetic
R	flg2	Round multiplication result

Instruction Variants

#	Syntax	Description
1	**MPY.2T Da,Db,Dn**	**SIMD2 fractional multiply**

```
if ((SR.SM2==0) || (SR.W20==1 && SR.SM2==1))
  Dn.WL = (S20) (((Da.L * Db.L)<<1)>>16)
  Dn.WH = (S20) (((Da.H * Db.H)<<1)>>16)
else
  Dn.L = srSAT16(((Da.L * Db.L)<<1)>>16)
  Dn.H = srSAT16(((Da.H * Db.H)<<1)>>16)
  Dn.E = 0
```

#	Syntax	Description
2	**MPY.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 fractional multiply**

```
Dm.WH = (((Da.H * Dc.H) << 1) >> 16)[19:0]
Dm.WL = (((Da.L * Dc.L) << 1) >> 16)[19:0]
Dn.WH = (((Db.H * Dd.H) << 1) >> 16)[19:0]
Dn.WL = (((Db.L * Dd.L) << 1) >> 16)[19:0]
```

#	Syntax	Description
3	**MPY.L.I.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 C-like integer multiply**

```
Dm.WH = (Da.H * Dc.H)[19:0]
Dm.WL = (Da.L * Dc.L)[19:0]
Dn.WH = (Db.H * Dd.H)[19:0]
Dn.WL = (Db.L * Dd.L)[19:0]
```

#	Syntax	Description
4	**MPY.SU.I.2T Da,Db,Dn**	**SIMD2 signed by unsigned integer multiply**

```
Dn.WH = (S20)((S20)Da.H * (U20)Db.H)
Dn.WL = (S20)((S20)Da.L * (U20)Db.L)
```

#	Syntax	Description
5	**MPY.IR.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 integer multiply with rounding**

```
Dm.WH = (S20)(((Da.H * Dc.H) + 0x8000) >> 16)[15:0]
Dm.WL = (S20)(((Da.L * Dc.L) + 0x8000) >> 16)[15:0]
Dn.WH = (S20)(((Db.H * Dd.H) + 0x8000) >> 16)[15:0]
Dn.WL = (S20)(((Db.L * Dd.L) + 0x8000) >> 16)[15:0]
```

Table continues on the next page...

#	Syntax	Description
6	**MPY.IR.4T Da,Dc:Dd,Dm:Dn**	**SIMD4 integer multiply with rounding, single first operand**

```
Dm.WH = (S20)(((Da.H * Dc.H) + 0x8000) >> 16)[15:0]
Dm.WL = (S20)(((Da.L * Dc.L) + 0x8000) >> 16)[15:0]
Dn.WH = (S20)(((Da.H * Dd.H) + 0x8000) >> 16)[15:0]
Dn.WL = (S20)(((Da.L * Dd.L) + 0x8000) >> 16)[15:0]
```

#	Syntax	Description
7	**MPY.R.2T Da,Db,Dn**	**SIMD2 fractional multiply with rounding**

```
if ((SR.SM2==0) || (SR.W20==1 && SR.SM2==1))
  Dn.WL = (S20) (((Da.L * Db.L)<<1+0x8000)>>16)
  Dn.WH = (S20) (((Da.H * Db.H)<<1+0x8000)>>16)
else
  Dn.L = srSAT16(((Da.L * Db.L)<<1+0x8000)>>16)
  Dn.H = srSAT16(((Da.H * Db.H)<<1+0x8000)>>16)
  Dn.E = 0
```

#	Syntax	Description
8	**MPY.R.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 fractional multiply with rounding**

```
Dm.WH = ((((Da.H * Dc.H) << 1) + 0x8000) >> 16)[19:0]
Dm.WL = ((((Da.L * Dc.L) << 1) + 0x8000) >> 16)[19:0]
Dn.WH = ((((Db.H * Dd.H) << 1) + 0x8000) >> 16)[19:0]
Dn.WL = ((((Db.L * Dd.L) << 1) + 0x8000) >> 16)[19:0]
```

#	Syntax	Description
9	**MPY.R.4T Da,Dc:Dd,Dm:Dn**	**SIMD4 fractional multiply with rounding, single first operand**

```
Dm.WH = ((((Da.H * Dc.H) << 1) + 0x8000) >> 16)[19:0]
Dm.WL = ((((Da.L * Dc.L) << 1) + 0x8000) >> 16)[19:0]
Dn.WH = ((((Da.H * Dd.H) << 1) + 0x8000) >> 16)[19:0]
Dn.WL = ((((Da.L * Dd.L) << 1) + 0x8000) >> 16)[19:0]
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Affect Instructions

Field	Relevant Variants	Comments
SR.SM2	1, 7	
SR.W20	1, 7	

Affected By Instructions

Field	Relevant Variants
SR.SAT	1, 7

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	2, 5, 6
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPY.nW 16x16 bit Multiply into 16-bit Results (DALU)

General Description

Performs SIMD2/SIMD4 16x16 bit multiply into 16-bit results, with saturation.

Flag Options

Flag	Position	Description
H	flg1	High 16-bit portion
HH	flg1	High 16 bits from the first argument and High 16 bits from the second argument
L	flg1	Use the low portion
LL	flg1	SIMD multiplication: low by low portions Extract and Sat instructions: 64-bit input
R	flg2	Round multiplication result
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MPY.S.2W Da,Db,Dn**	**SIMD2 fractional multiply with saturation**

	`Dn.{HL} = SAT16(((Da.{HL} * Db.{HL})<<1)>>16)`
	`Dn.E = 0`	

2	**MPY.SR.2W Da,Db,Dn**	**SIMD2 fractional multiply with rounding and saturation**

	`Dn.H = SAT16 ((((Da.H * Db.H) << 1) + 0x8000) >> 16))[15:0]`
	`Dn.L = SAT16 ((((Da.L * Db.L) << 1) + 0x8000) >> 16))[15:0]`
	`Dn.E = 0`	

3	**MPY.SR.4W Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 fractional multiply with rounding and saturation**

	`Dm.H = SAT16 ((((Da.H * Dc.H) << 1) + 0x8000) >> 16))[15:0]`
	`Dm.L = SAT16 ((((Da.L * Dc.L) << 1) + 0x8000) >> 16))[15:0]`
	`Dm.E = 0`
	`Dn.H = SAT16 ((((Db.H * Dd.H) << 1) + 0x8000) >> 16))[15:0]`
	`Dn.L = SAT16 ((((Db.L * Dd.L) << 1) + 0x8000) >> 16))[15:0]`
	`Dn.E = 0`	

Table continues on the next page...

#	Syntax	Description
4	**MPY.SR.4W Da,Dc:Dd,Dm:Dn**	**SIMD4 fractional multiply with rounding and saturation, single first operand**

	`Dm.H = SAT16 (((((Da.H * Dc.H) << 1) + 0x8000) >> 16))[15:0]`
	`Dm.L = SAT16 (((((Da.L * Dc.L) << 1) + 0x8000) >> 16))[15:0]`
	`Dm.E = 0`
	`Dn.H = SAT16 (((((Da.H * Dd.H) << 1) + 0x8000) >> 16))[15:0]`
	`Dn.L = SAT16 (((((Da.L * Dd.L) << 1) + 0x8000) >> 16))[15:0]`
	`Dn.E = 0`	

5	**MPY.H.SR.2W Da,Db,Dn**	**SIMD2 fractional multiply with rounding and saturation, multiplying high portion by two portions**

	`Dn.H = SAT16 (((((Da.H * Db.H) << 1) + 0x8000) >> 16))`
	`Dn.L = SAT16 (((((Da.H * Db.L) << 1) + 0x8000) >> 16))`
	`Dn.E = 0`	

6	**MPY.HH.SR.2W Da:Db,Dc:Dd,Dn**	**SIMD2 fractional multiply with rounding and saturation, multiplying high portions by high portions**

	`Dn.H = SAT16 (((((Da.H * Dc.H) << 1) + 0x8000) >> 16))`
	`Dn.L = SAT16 (((((Db.H * Dd.H) << 1) + 0x8000) >> 16))`
	`Dn.E = 0`	

7	**MPY.L.SR.2W Da,Db,Dn**	**SIMD2 fractional multiply with rounding and saturation, multiplying low portion by two portions**

	`Dn.H = SAT16 (((((Da.L * Db.H) << 1) + 0x8000) >> 16))`
	`Dn.L = SAT16 (((((Da.L * Db.L) << 1) + 0x8000) >> 16))`
	`Dn.E = 0`	

8	**MPY.LL.SR.2W Da:Db,Dc:Dd,Dn**	**SIMD2 fractional multiply with rounding and saturation, multiplying low portions by low portions**

	`Dn.H = SAT16 (((((Da.L * Dc.L) << 1) + 0x8000) >> 16))`
	`Dn.L = SAT16 (((((Db.L * Dd.L) << 1) + 0x8000) >> 16))`
	`Dn.E = 0`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPY.nX 16x16 bit Multiply into 40-bit Result (DALU)

General Description

Performs single/SIMD2 16x16 bit multiply into 40-bit results.

Flag Options

Flag	Position	Description
H	flg1	High 16-bit portion
HH	flg1	High 16 bits from the first argument and High 16 bits from the second argument
L	flg1	Use the low portion
LL	flg1	SIMD multiplication: low by low portions Extract and Sat instructions: 64-bit input
US	flg1	Unsigned By Signed
UU	flg1	Unsigned By Unsigned
I	flg2	Integer arithmetic
R	flg2	Round multiplication result
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MPY.2X Da,Db,Dm:Dn**	**SIMD2 fractional multiply, multiplying high and low portions**

	`Dm = ((Da.H * Db.H) << 1)[39:0]`
	`Dn = ((Da.L * Db.L) << 1)[39:0]`	

2	**MPY.X Da.h,Db.h,Dn**	**Single fractional multiply, multiplying high by high portions**
	`Dn = ((Da.H * Db.H) << 1)[39:0]`	
3	**MPY.X Da.h,Db.l,Dn**	**Single fractional multiply, multiplying high by low portions**
	`Dn = ((Da.H * Db.L) << 1)[39:0]`	
4	**MPY.X Da.l,Db.l,Dn**	**Single fractional multiply, multiplying low by low portions**
	`Dn = ((Da.L * Db.L) << 1)[39:0]`	
5	**MPY.H.2X Da,Db,Dm:Dn**	**SIMD2 fractional multiply, high portion by two portions**

	`Dm = ((S40)Da.H * (S40)Db.H) << 1`
	`Dn = ((S40)Da.H * (S40)Db.L) << 1`	

Table continues on the next page...

#	Syntax	Description
6	**MPY.HH.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional multiply, high portions by high portions**

	`Dm = ((Da.H * Dc.H) << 1)[39:0]`
	`Dn = ((Db.H * Dd.H) << 1)[39:0]`	

7	**MPY.L.2X Da,Db,Dm:Dn**	**SIMD2 fractional multiply, low portion by two portions**

	`Dm = ((S40)Da.L * (S40)Db.H) << 1`
	`Dn = ((S40)Da.L * (S40)Db.L) << 1`	

8	**MPY.LL.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional multiply, low portions by low portions**

	`Dm = ((Da.L * Dc.L) << 1)[39:0]`
	`Dn = ((Db.L * Dd.L) << 1)[39:0]`	

9	**MPY.I.2X Da,Db,Dm:Dn**	**SIMD2 integer multiply, multiplying high and low portions**

	`Dm = ((S40)Da.H * (S40)Db.H)`
	`Dn = ((S40)Da.L * (S40)Db.L)`	

10	**MPY.I.X Da.h,Db.h,Dn**	**Single integer multiply, multiplying high by high portions**
	`Dn = ((S40)Da.H * (S40)Db.H)`	
11	**MPY.I.X Da.h,Db.l,Dn**	**Single integer multiply, multiplying high by low portions**
	`Dn = ((S40)Da.H * (S40)Db.L)`	
12	**MPY.I.X Da.l,Db.l,Dn**	**Single integer multiply, multiplying low by low portions**
	`Dn = ((S40)Da.L * (S40)Db.L)`	
13	**MPY.I.X #s16,Da.l,Dn**	**Single integer multiply using immediate value**
	`Dn = ((S40)Da.L * (S40)s16_t2)`	
14	**MPY.H.I.2X Da,Db,Dm:Dn**	**SIMD2 integer multiply, high portion by two portions**

	`Dm = ((S40)Da.H * (S40)Db.H)`
	`Dn = ((S40)Da.H * (S40)Db.L)`	

15	**MPY.HH.I.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 integer multiply, high portions by high portions**

	`Dm = (Da.H * Dc.H)[39:0]`
	`Dn = (Db.H * Dd.H)[39:0]`	

16	**MPY.L.I.2X Da,Db,Dm:Dn**	**SIMD2 integer multiply, low portion by two portions**

	`Dm = ((S40)Da.L * (S40)Db.H)`
	`Dn = ((S40)Da.L * (S40)Db.L)`	

17	**MPY.LL.I.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 integer multiply, low portions by low portions**

Table continues on the next page...

#	Syntax	Description

	`Dm = (Da.L * Dc.L)[39:0]`
	`Dn = (Db.L * Dd.L)[39:0]`	

18	**MPY.US.I.X Da.h,Db.h,Dn**	**Single 16x16 bit unsigned by signed integer multiply**
	`Dn = (Da.H * Db.H)[39:0]`	
19	**MPY.US.I.X Da.l,Db.l,Dn**	**Single 16x16 bit unsigned by signed integer multiply**
	`Dn = (Da.L * Db.L)[39:0]`	
20	**MPY.UU.I.X Da.h,Db.h,Dn**	**Single 16x16 bit unsigned by unsigned integer multiply**
	`Dn = (Da.H * Db.H)[39:0]`	
21	**MPY.UU.I.X Da.l,Db.l,Dn**	**Single 16x16 bit unsigned by unsigned integer multiply**
	`Dn = (U40)(((U40)Db.L) * ((U40)Da.L))`	
22	**MPY.R.X Da.h,Db.h,Dn**	**Single 16x16 bit fractional multiply with rounding, multiplying high by high portions**
	`Dn = (((Da.H * Db.H) << 1) + 0x8000)[39:0]`	
23	**MPY.R.X Da.l,Db.h,Dn**	**Single 16x16 bit fractional multiply with rounding, multiplying high by low portions**
	`Dn = (((Da.L * Db.H) << 1) + 0x8000)[39:0]`	
24	**MPY.R.X Da.l,Db.l,Dn**	**Single 16x16 bit fractional multiply with rounding, multiplying low by low portions**
	`Dn = (((Da.L * Db.L) << 1) + 0x8000)[39:0]`	
25	**MPY.S.2X Da,Db,Dm:Dn**	**SIMD2 fractional multiply, multiplying high and low portions with saturation**

	`Dm = SAT32(((Da.H * Db.H) << 1))`
	`Dn = SAT32(((Da.L * Db.L) << 1))`	

26	**MPY.S.X Da.h,Db.h,Dn**	**Single 16x16 bit fractional multiply with saturation, multiplying high by high portions**
	`Dn = SAT32(((Da.H * Db.H) << 1))`	
27	**MPY.S.X Da.h,Db.l,Dn**	**Single 16x16 bit fractional multiply with saturation, multiplying high by low portions**
	`Dn = SAT32(((Da.H * Db.L) << 1))`	
28	**MPY.S.X Da.l,Db.l,Dn**	**Single 16x16 bit fractional multiply with saturation, multiplying low by low portions**
	`Dn = SAT32(((Da.L * Db.L) << 1))`	
29	**MPY.H.S.2X Da,Db,Dm:Dn**	**SIMD2 fractional multiply with saturation, high portion by two portions**

	`Dm = SAT32(((Da.H * Db.H) << 1))`
	`Dn = SAT32(((Da.H * Db.L) << 1))`	

Table continues on the next page...

#	Syntax	Description
30	**MPY.HH.S.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional multiply with saturation, high portions by high portions**

	`Dm = SAT32(((Da.H * Dc.H) << 1))`
	`Dn = SAT32(((Db.H * Dd.H) << 1))`	

31	**MPY.L.S.2X Da,Db,Dm:Dn**	**SIMD2 fractional multiply with saturation, low portion by two portions**

	`Dm = SAT32(((Da.L * Db.H) << 1))`
	`Dn = SAT32(((Da.L * Db.L) << 1))`	

32	**MPY.LL.S.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional multiply with saturation, low portions by low portions**

	`Dm = SAT32(((Da.L * Dc.L) << 1))`
	`Dn = SAT32(((Db.L * Dd.L) << 1))`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	
s16	$-2^{15} \le s16 < 2^{15}$	

Affected By Instructions

Field	Relevant Variants
SR.SAT	25, 26, 27, 28, 29, 30, 31, 32

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	9, 10, 11, 14, 15, 16, 18, 20
Encoding length	32-bits in 64	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32
	64-bits	13
Pipeline behavior	dalu_MPY	All

MPY32.LL 32x32 bit Multiply into 64-bit Result (DALU)

General Description

Performs single 32x32 bit multiply, storing the 64-bit result in a register pair.

Flag Options

Flag	Position	Description
SU	flg1	Signed By Unsigned
UU	flg1	Unsigned By Unsigned
I	flg2	Integer arithmetic

Instruction Variants

#	Syntax	Description
1	**MPY32.LL Da,Db,Dm:Dn**	**Fractional 32x32 bit multiply**

	`Dm = (((Da.M * Db.M) << 1) >> 32)[39:0]`
	`Dn = (U40)((Da.M * Db.M) << 1)[31:0]`	

2	**MPY32.I.LL Da,Db,Dm:Dn**	**Integer 32x32 bit multiply**

	`Dm = (S40)((S64)Da.M * (S64)Db.M)[63:32]`
	`Dn = (U40)((S64)Da.M * (S64)Db.M)[31:0]`	

3	**MPY32.SU.I.LL Da,Db,Dm:Dn**	**Integer 32x32 bit signed by unsigned multiply**

	`Dm = (S40)((S64)Da.M * (U64)Db.M)[63:32]`
	`Dn = (U40)((S64)Da.M * (U64)Db.M)[31:0]`	

4	**MPY32.UU.I.LL Da,Db,Dm:Dn**	**Integer 32x32 bit unsigned by unsigned multiply**

	`Dm = (U40)((U64)Da.M * (U64)Db.M)[63:32]`
	`Dn = (U40)((U64)Da.M * (U64)Db.M)[31:0]`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Db	D0-D63	
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPY32.nL 32x32 bit Multiply into 32-bit Result (DALU)

General Description

Performs 32x32 bit multiply, keeping only the low 32-bit of the result.

Flag Options

Flag	Position	Description
I	flg2	Integer arithmetic
L	flg2	32-bit operation

Instruction Variants

#	Syntax	Description
1	**MPY32.IL.L Da,Db,Dn**	**Single 32x32 bit integer multiply**
	`Dn = (U40)((S64)Da.M * (S64)Db.M)[31:0]`	

Explicit Operands

Operand	Permitted Values
Da	`D0-D63`
Db	`D0-D63`
Dn	`D0-D63`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPY32.nX 32x32 bit Multiply into 40-bit Result (DALU)

General Description

Performs 32x32 bit multiply, keeping the high 32-bit of the 64-bit result in 40-bit result.

Flag Options

Flag	Position	Description
NP	flg1	Negative Positive
PN	flg1	Positive Negative
X	flg1	Cross between the arguments
I	flg2	Integer arithmetic
R	flg2	Round multiplication result
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MPY32.IR.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 32x32 bit integer multiply with rounding**

	`Dm = (S40)(((S64)(((S64)Da.M * (S64)Dc.M) + 0x80000000)) >> 32)`
	`Dn = (S40)(((S64)(((S64)Db.M * (S64)Dd.M) + 0x80000000)) >> 32)`	

2	**MPY32.IR.X Da,Db,Dn**	**Single 32x32 bit integer multiply with rounding**
	`Dn = ((((S64)((S64)Da.M * (S64)Db.M)) + 0x80000000) >> 32)[39:0]`	
3	**MPY32.R.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 32x32 bit fractional multiply with rounding**

	`Dm = ((((Da.M * Dc.M) << 1) + 0x80000000) >> 32)[39:0]`
	`Dn = ((((Db.M * Dd.M) << 1) + 0x80000000) >> 32)[39:0]`	

4	**MPY32.R.X Da,Db,Dn**	**Single 32x32 bit fractional multiply with rounding**
	`Dn = ((((Da.M * Db.M) << 1) + 0x80000000) >> 32)[39:0]`	
5	**MPY32.NPX.R.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 32x32 bit fractional multiply with rounding, first product negated and second operand is crossed (swapped) for assisting in complex multiplication**

	`Dm = (((((-(Da.M * Dd.M)) << 1) + 0x80000000)) >> 32)[39:0]`
	`Dn = ((((Da.M * Dc.M) << 1) + 0x80000000)) >> 32)[39:0]`	

Table continues on the next page...

#	Syntax	Description
6	**MPY32.PNX.R.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 32x32 bit fractional multiply with rounding, second product negated and second operand is crossed (swapped) for assisting in complex multiplication**

	`Dm = ((((Da.M * Dd.M) << 1) + 0x80000000) >> 32)[39:0]`
	`Dn = ((((-(Da.M * Dc.M)) << 1) + 0x80000000) >> 32)[39:0]`	

7	**MPY32.SR.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 32x32 bit multiply with rounding and saturation**

	`Dm = SAT32(((((Da.M * Dc.M)) << 1) + 0x80000000) >> 32))`
	`Dn = SAT32(((((Db.M * Dd.M)) << 1) + 0x80000000) >> 32))`	

8	**MPY32.SR.X Da,Db,Dn**	**Single 32x32 bit multiply with rounding and saturation**
	`Dn = SAT32(((((Da.M * Db.M) << 1) + 0x80000000)) >> 32))`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
SR.SAT	7, 8

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 5
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPY32A 32x32 bit Multiply into 32-bit Result (IPU)

General Description

Performs 32x32 bit multiply, keeping the low 32-bit of the 64-bit result in a 32-bit result in an address register.

Flag Options

Flag	Position	Description
I	flg2	Integer arithmetic

Instruction Variants

#	Syntax	Description
1	**MPY32A.I Ra,Rb,Rn**	**32x32 bit C-like integer multiply**
	`Rn = ((S64)Ra * (S64)Rb)[31:0]`	
2	**MPY32A.I #s16,Ra,Rn**	**32x32 bit C-like integer multiply using immediate value**
	`Rn = ((S64)Ra * (S64)s16)[31:0]`	

Explicit Operands

Operand	Permitted Values
Ra	R0-R31
Rb	R0-R31
Rn	R0-R31
s16	$-2^{15} \leq s16 < 2^{15}$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	1
	48-bits	2
Pipeline behavior	agu_AAU_2CYCLE	All

MPYADDA 16x32 bit Multiply-Add into 32-bit Result (IPU)

General Description

Performs single 16x32 bit multiply and add, storing the low 32-bit of the result in an address register.

Flag Options

Flag	Position	Description
I	flg2	Integer arithmetic

Instruction Variants

#	Syntax	Description
1	**MPYADDA.I #u16,Ra,Rb,Rn**	**16x32 bit multiply-add using immediate value**
	`Rn = ((S48)Ra * (U48)u16_t6)[31:0] + Rb`	

Explicit Operands

Operand	Permitted Values
Ra	R0-R31
Rb	R0-R31
Rn	R0-R31
u16	$0 \leq u16 < 2^{16}$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	48-bits	All
Pipeline behavior	agu_AAU_2CYCLE_MPYADDA	All

MPYCX.nT 16x16 bit Complex Multiply into 20-bit Results (DALU)

General Description

Performs single/SIMD2 16x16 bit complex multiply into 20-bit results.
Each complex number is packed in a single register.

Flag Options

Flag	Position	Description
C	flg1	Complex Conjugate
I	flg2	Integer arithmetic
R	flg2	Round multiplication result

Instruction Variants

#	Syntax	Description
1	**MPYCX.IR.2T Da,Db,Dn**	**Complex integer multiply with rounding**

	`Dn.WH = (((Da.H * Db.H) - (Da.L * Db.L)) + 0x8000) >> 16)[19:0]`
	`Dn.WL = (((Da.L * Db.H) + (Da.H * Db.L)) + 0x8000) >> 16)[19:0]`	

2	**MPYCX.IR.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 Complex integer multiply with rounding**

	`Dm.WH = (((Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16)[19:0]`
	`Dm.WL = (((Da.L * Dc.H) + (Da.H * Dc.L)) + 0x8000) >> 16)[19:0]`
	`Dn.WH = (((Db.H * Dd.H) - (Db.L * Dd.L)) + 0x8000) >> 16)[19:0]`
	`Dn.WL = (((Db.L * Dd.H) + (Db.H * Dd.L)) + 0x8000) >> 16)[19:0]`	

3	**MPYCX.IR.4T Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - Complex integer multiply with rounding**

	`Dm.WH = (((Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16)[19:0]`
	`Dm.WL = (((Da.L * Dc.H) + (Da.H * Dc.L)) + 0x8000) >> 16)[19:0]`
	`Dn.WH = (((Da.H * Dd.H) - (Da.L * Dd.L)) + 0x8000) >> 16)[19:0]`
	`Dn.WL = (((Da.L * Dd.H) + (Da.H * Dd.L)) + 0x8000) >> 16)[19:0]`	

4	**MPYCX.C.IR.2T Da,Db,Dn**	**Complex integer multiply by conjugate with rounding**

	`Dn.WH = (((Da.H * Db.H) + (Da.L * Db.L)) + 0x8000) >> 16)[19:0]`
	`Dn.WL = (((Da.H * Db.L) - (Da.L * Db.H)) + 0x8000) >> 16)[19:0]`	

5	**MPYCX.C.IR.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 Complex integer multiply by conjugate with rounding**

	`Dm.WH = (((Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16)[19:0]`
	`Dm.WL = (((Da.H * Dc.L) - (Da.L * Dc.H)) + 0x8000) >> 16)[19:0]`
	`Dn.WH = (((Db.H * Dd.H) + (Db.L * Dd.L)) + 0x8000) >> 16)[19:0]`
	`Dn.WL = (((Db.H * Dd.L) - (Db.L * Dd.H)) + 0x8000) >> 16)[19:0]`	

Table continues on the next page...

#	Syntax	Description
6	**MPYCX.C.IR.4T Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - Complex integer multiply by conjugate with rounding**

	`Dm.WH = ((((Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16)[19:0]`
	`Dm.WL = ((((Da.H * Dc.L) - (Da.L * Dc.H)) + 0x8000) >> 16)[19:0]`
	`Dn.WH = ((((Da.H * Dd.H) + (Da.L * Dd.L)) + 0x8000) >> 16)[19:0]`
	`Dn.WL = ((((Da.H * Dd.L) - (Da.L * Dd.H)) + 0x8000) >> 16)[19:0]`	

7	**MPYCX.R.2T Da,Db,Dn**	**Complex fractional multiply with rounding**

	`Dn.WH = (((((Da.H * Db.H) - (Da.L * Db.L)) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WL = (((((Da.L * Db.H) + (Da.H * Db.L)) << 1) + 0x8000) >> 16)[19:0]`	

8	**MPYCX.R.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 Complex fractional multiply with rounding**

	`Dm.WH = (((((Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16)[19:0]`
	`Dm.WL = (((((Da.L * Dc.H) + (Da.H * Dc.L)) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WH = (((((Db.H * Dd.H) - (Db.L * Dd.L)) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WL = (((((Db.L * Dd.H) + (Db.H * Dd.L)) << 1) + 0x8000) >> 16)[19:0]`	

9	**MPYCX.R.4T Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - Complex fractional multiply with rounding**

	`Dm.WH = (((((Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16)[19:0]`
	`Dm.WL = (((((Da.L * Dc.H) + (Da.H * Dc.L)) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WH = (((((Da.H * Dd.H) - (Da.L * Dd.L)) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WL = (((((Da.L * Dd.H) + (Da.H * Dd.L)) << 1) + 0x8000) >> 16)[19:0]`	

10	**MPYCX.C.R.2T Da,Db,Dn**	**Complex fractional multiply by conjugate with rounding**

	`Dn.WH = (((((Da.H * Db.H) + (Da.L * Db.L)) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WL = (((((Da.H * Db.L) - (Da.L * Db.H)) << 1) + 0x8000) >> 16)[19:0]`	

11	**MPYCX.C.R.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 Complex fractional multiply by conjugate with rounding**

	`Dm.WH = (((((Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16)[19:0]`
	`Dm.WL = (((((Da.H * Dc.L) - (Da.L * Dc.H)) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WH = (((((Db.H * Dd.H) + (Db.L * Dd.L)) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WL = (((((Db.H * Dd.L) - (Db.L * Dd.H)) << 1) + 0x8000) >> 16)[19:0]`	

12	**MPYCX.C.R.4T Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - Complex fractional multiply by conjugate with rounding**

	`Dm.WH = (((((Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16)[19:0]`
	`Dm.WL = (((((Da.H * Dc.L) - (Da.L * Dc.H)) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WH = (((((Da.H * Dd.H) + (Da.L * Dd.L)) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WL = (((((Da.H * Dd.L) - (Da.L * Dd.H)) << 1) + 0x8000) >> 16)[19:0]`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 3, 4, 5, 6
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPYCX.nW 16x16 bit Complex Multiply into 16-bit Result (DALU)

General Description

Performs single/SIMD2 16x16 bit complex multiply-accumulate into 16-bit results, with saturation. Each complex number is packed in a single register.

Flag Options

Flag	Position	Description
C	flg1	Complex Conjugate
I	flg2	Integer arithmetic
R	flg2	Round multiplication result
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MPYCX.ISR.2W Da,Db,Dn**	**Complex integer multiply with rounding and saturation**

	`Dn.H = SAT16 ((((Da.H * Db.H) - (Da.L * Db.L)) + 0x8000) >> 16))[15:0]`
	`Dn.L = SAT16 ((((Da.L * Db.H) + (Da.H * Db.L)) + 0x8000) >> 16))[15:0]`
	`Dn.E = 0`	

2	**MPYCX.ISR.4W Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 Complex integer multiply with rounding and saturation**

	`Dm.H = SAT16 ((((Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16))[15:0]`
	`Dm.L = SAT16 ((((Da.L * Dc.H) + (Da.H * Dc.L)) + 0x8000) >> 16))[15:0]`
	`Dm.E = 0`
	`Dn.H = SAT16 ((((Db.H * Dd.H) - (Db.L * Dd.L)) + 0x8000) >> 16))[15:0]`
	`Dn.L = SAT16 ((((Db.L * Dd.H) + (Db.H * Dd.L)) + 0x8000) >> 16))[15:0]`
	`Dn.E = 0`	

3	**MPYCX.ISR.4W Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - Complex integer multiply with rounding and saturation**

	`Dm.H = SAT16 ((((Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16))[15:0]`
	`Dm.L = SAT16 ((((Da.L * Dc.H) + (Da.H * Dc.L)) + 0x8000) >> 16))[15:0]`
	`Dm.E = 0`
	`Dn.H = SAT16 ((((Da.H * Dd.H) - (Da.L * Dd.L)) + 0x8000) >> 16))[15:0]`
	`Dn.L = SAT16 ((((Da.L * Dd.H) + (Da.H * Dd.L)) + 0x8000) >> 16))[15:0]`
	`Dn.E = 0`	

4	**MPYCX.C.ISR.2W Da,Db,Dn**	**Complex integer multiply by conjugate with rounding and saturation**

	`Dn.H = SAT16 ((((Da.H * Db.H) + (Da.L * Db.L)) + 0x8000) >> 16))[15:0]`
	`Dn.L = SAT16 ((((Da.H * Db.L) - (Da.L * Db.H)) + 0x8000) >> 16))[15:0]`
	`Dn.E = 0`	

5	**MPYCX.C.ISR.4W Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 Complex integer multiply by conjugate with rounding and saturation**

Table continues on the next page...

#	Syntax	Description

	Dm.H = SAT16 ((((Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16))[15:0]
	Dm.L = SAT16 ((((Da.H * Dc.L) - (Da.L * Dc.H)) + 0x8000) >> 16))[15:0]
	Dm.E = 0
	Dn.H = SAT16 ((((Db.H * Dd.H) + (Db.L * Dd.L)) + 0x8000) >> 16))[15:0]
	Dn.L = SAT16 ((((Db.H * Dd.L) - (Db.L * Dd.H)) + 0x8000) >> 16))[15:0]
	Dn.E = 0	

6	**MPYCX.C.ISR.4W Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - Complex integer multiply by conjugate with rounding and saturation**

	Dm.H = SAT16 ((((Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16))[15:0]
	Dm.L = SAT16 ((((Da.H * Dc.L) - (Da.L * Dc.H)) + 0x8000) >> 16))[15:0]
	Dm.E = 0
	Dn.H = SAT16 ((((Da.H * Dd.H) + (Da.L * Dd.L)) + 0x8000) >> 16))[15:0]
	Dn.L = SAT16 ((((Da.H * Dd.L) - (Da.L * Dd.H)) + 0x8000) >> 16))[15:0]
	Dn.E = 0	

7	**MPYCX.SR.2W Da,Db,Dn**	**Complex fractional multiply with rounding and saturation**

	Dn.H = SAT16 (((((Da.H * Db.H) - (Da.L * Db.L)) << 1) + 0x8000) >> 16))[15:0]
	Dn.L = SAT16 (((((Da.L * Db.H) + (Da.H * Db.L)) << 1) + 0x8000) >> 16))[15:0]
	Dn.E = 0	

8	**MPYCX.SR.4W Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 Complex fractional multiply with rounding and saturation**

	Dm.H = SAT16 (((((Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[15:0]
	Dm.L = SAT16 (((((Da.L * Dc.H) + (Da.H * Dc.L)) << 1) + 0x8000) >> 16))[15:0]
	Dm.E = 0
	Dn.H = SAT16 (((((Db.H * Dd.H) - (Db.L * Dd.L)) << 1) + 0x8000) >> 16))[15:0]
	Dn.L = SAT16 (((((Db.L * Dd.H) + (Db.H * Dd.L)) << 1) + 0x8000) >> 16))[15:0]
	Dn.E = 0	

9	**MPYCX.SR.4W Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - Complex fractional multiply with rounding and saturation**

	Dm.H = SAT16 (((((Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[15:0]
	Dm.L = SAT16 (((((Da.L * Dc.H) + (Da.H * Dc.L)) << 1) + 0x8000) >> 16))[15:0]
	Dm.E = 0
	Dn.H = SAT16 (((((Da.H * Dd.H) - (Da.L * Dd.L)) << 1) + 0x8000) >> 16))[15:0]
	Dn.L = SAT16 (((((Da.L * Dd.H) + (Da.H * Dd.L)) << 1) + 0x8000) >> 16))[15:0]
	Dn.E = 0	

10	**MPYCX.C.SR.2W Da,Db,Dn**	**Complex fractional multiply by conjugate with rounding and saturation**

	Dn.H = SAT16 (((((Da.H * Db.H) + (Da.L * Db.L)) << 1) + 0x8000) >> 16))[15:0]
	Dn.L = SAT16 (((((Da.H * Db.L) - (Da.L * Db.H)) << 1) + 0x8000) >> 16))[15:0]
	Dn.E = 0	

11	**MPYCX.C.SR.4W Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 Complex fractional multiply by conjugate with rounding and saturation**

	Dm.H = SAT16 (((((Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[15:0]
	Dm.L = SAT16 (((((Da.H * Dc.L) - (Da.L * Dc.H)) << 1) + 0x8000) >> 16))[15:0]
	Dm.E = 0
	Dn.H = SAT16 (((((Db.H * Dd.H) + (Db.L * Dd.L)) << 1) + 0x8000) >> 16))[15:0]
	Dn.L = SAT16 (((((Db.H * Dd.L) - (Db.L * Dd.H)) << 1) + 0x8000) >> 16))[15:0]
	Dn.E = 0	

Table continues on the next page...

#	Syntax	Description
12	**MPYCX.C.SR.4W Da,Dc:Dd,Dm:Dn**	**SIMD2 by one value - Complex fractional multiply by conjugate with rounding and saturation**

```
Dm.H = SAT16 (((((Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[15:0]
Dm.L = SAT16 (((((Da.H * Dc.L) - (Da.L * Dc.H)) << 1) + 0x8000) >> 16))[15:0]
Dm.E = 0
Dn.H = SAT16 (((((Da.H * Dd.H) + (Da.L * Dd.L)) << 1) + 0x8000) >> 16))[15:0]
Dn.L = SAT16 (((((Da.H * Dd.L) - (Da.L * Dd.H)) << 1) + 0x8000) >> 16))[15:0]
Dn.E = 0
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3, 4, 5, 6
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPYCX.nX 16x16 bit Complex Multiply into 40-bit Result (DALU)

General Description

Performs 16x16 bit complex multiply into full precision 40-bit results.
Source complex numbers are packed in single registers, while destination complex numbers resides in a register pair.

Flag Options

Flag	Position	Description
C	flg1	Complex Conjugate
I	flg2	Integer arithmetic
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MPYCX.2X Da,Db,Dm:Dn**	**Complex fractional multiply**

	`Dm = (((Da.H * Db.H) - (Da.L * Db.L)) << 1)[39:0]`
	`Dn = (((Da.L * Db.H) + (Da.H * Db.L)) << 1)[39:0]`	

2	**MPYCX.C.2X Da,Db,Dm:Dn**	**Complex fractional multiply by conjugate**

	`Dm = (((Da.H * Db.H) + (Da.L * Db.L)) << 1)[39:0]`
	`Dn = (((Da.H * Db.L) - (Da.L * Db.H)) << 1)[39:0]`	

3	**MPYCX.I.2X Da,Db,Dm:Dn**	**Complex integer multiply**

	`Dm = ((Da.H * Db.H) - (Da.L * Db.L))[39:0]`
	`Dn = ((Da.L * Db.H) + (Da.H * Db.L))[39:0]`	

4	**MPYCX.C.I.2X Da,Db,Dm:Dn**	**Complex integer multiply by conjugate**

	`Dm = ((Da.H * Db.H) + (Da.L * Db.L))[39:0]`
	`Dn = ((Da.H * Db.L) - (Da.L * Db.H))[39:0]`	

5	**MPYCX.IS.2X Da,Db,Dm:Dn**	**Complex integer multiply with saturation**

	`Dm = SAT32(((Da.H * Db.H) - (Da.L * Db.L)))`
	`Dn = SAT32(((Da.L * Db.H) + (Da.H * Db.L)))`	

6	**MPYCX.C.IS.2X Da,Db,Dm:Dn**	**Complex integer multiply by conjugate with saturation**

	`Dm = SAT32(((Da.H * Db.H) + (Da.L * Db.L)))`
	`Dn = SAT32(((Da.H * Db.L) - (Da.L * Db.H)))`	

7	**MPYCX.S.2X Da,Db,Dm:Dn**	**Complex fractional multiply with saturation**

	`Dm = SAT32((((Da.H * Db.H) - (Da.L * Db.L)) << 1))`
	`Dn = SAT32((((Da.L * Db.H) + (Da.H * Db.L)) << 1))`	

Table continues on the next page...

#	Syntax	Description
8	**MPYCX.C.S.2X Da,Db,Dm:Dn**	**Complex fractional multiply by conjugate with saturation**

	`Dm = SAT32(((Da.H * Db.H) + (Da.L * Db.L)) << 1))`
	`Dn = SAT32(((Da.H * Db.L) - (Da.L * Db.H)) << 1))`	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Db	D0-D63
Dm:Dn	$D_n:D_{n+1}$ $0 \le n \le 62$

Affected By Instructions

Field	Relevant Variants
SR.SAT	5, 6, 7, 8

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	4, 5, 6
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPYCXD.nW 16x16 bit Complex Dot-Product into 20-bit Result (DALU)

General Description

Performs two 16x16 bit complex multiplications, sum or subtract the products into 20-bit results. Each complex number is packed in a single register.

Flag Options

Flag	Position	Description
C	flg1	Complex Conjugate
PN	flg1	Positive Negative
PP	flg1	Positive Positive
I	flg2	Integer arithmetic
R	flg2	Round multiplication result
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MPYCXD.CPN.ISR.2W Da:Db,Dc:Dd,Dn**	**Complex integer dot product by conjugate with rounding and saturation, second product is negated**

	`Dn.WH = (U20)SAT16 (((((Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16) + (((-(Db.H * Dd.H) - (Db.L * Dd.L)) + 0x8000) >> 16)))`
	`Dn.WL = (U20)SAT16 (((((Da.H * Dc.L) - (Da.L * Dc.H)) + 0x8000) >> 16) + (((-(Db.H * Dd.L) + (Db.L * Dd.H)) + 0x8000) >> 16)))`	

2	**MPYCXD.CPP.ISR.2W Da:Db,Dc:Dd,Dn**	**Complex integer dot product by conjugate with rounding and saturation**

	`Dn.WH = (U20)SAT16 (((((Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16) + ((((Db.H * Dd.H) + (Db.L * Dd.L)) + 0x8000) >> 16)))`
	`Dn.WL = (U20)SAT16 (((((Da.H * Dc.L) - (Da.L * Dc.H)) + 0x8000) >> 16) + ((((Db.H * Dd.L) - (Db.L * Dd.H)) + 0x8000) >> 16)))`	

3	**MPYCXD.PN.ISR.2W Da:Db,Dc:Dd,Dn**	**Complex integer dot product with rounding and saturation, second product is negated**

	`Dn.WH = (U20)SAT16 (((((Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16) + (((-(Db.H * Dd.H) + (Db.L * Dd.L)) + 0x8000) >> 16)))`
	`Dn.WL = (U20)SAT16 (((((Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16) + (((-(Db.H * Dd.L) - (Db.L * Dd.H)) + 0x8000) >> 16)))`	

Table continues on the next page...

#	Syntax	Description
4	**MPYCXD.PP.ISR.2W Da:Db,Dc:Dd,Dn**	**Complex integer dot product with rounding and saturation**

	Dn.WH = (U20)SAT16 (((((Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16) + (((Db.H * Dd.H) - (Db.L * Dd.L)) + 0x8000) >> 16)))
	Dn.WL = (U20)SAT16 (((((Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16) + (((Db.H * Dd.L) + (Db.L * Dd.H)) + 0x8000) >> 16)))	

5	**MPYCXD.CPN.SR.2W Da:Db,Dc:Dd,Dn**	**Complex fractional dot product by conjugate with rounding and saturation, second product is negated**

	Dn.WH = (U20)SAT16 ((((((Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16) + ((((-(Db.H * Dd.H) - (Db.L * Dd.L)) << 1) + 0x8000) >> 16)))
	Dn.WL = (U20)SAT16 ((((((Da.H * Dc.L) - (Da.L * Dc.H)) << 1) + 0x8000) >> 16) + ((((-(Db.H * Dd.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16)))	

6	**MPYCXD.CPP.SR.2W Da:Db,Dc:Dd,Dn**	**Complex fractional dot product by conjugate with rounding and saturation**

	Dn.WH = (U20)SAT16 ((((((Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16) + ((((Db.H * Dd.H) + (Db.L * Dd.L)) << 1) + 0x8000) >> 16)))
	Dn.WL = (U20)SAT16 ((((((Da.H * Dc.L) - (Da.L * Dc.H)) << 1) + 0x8000) >> 16) + ((((Db.H * Dd.L) - (Db.L * Dd.H)) << 1) + 0x8000) >> 16)))	

7	**MPYCXD.PN.SR.2W Da:Db,Dc:Dd,Dn**	**Complex fractional dot product with rounding and saturation, second product is negated**

	Dn.WH = (U20)SAT16 ((((((Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16) + ((((-(Db.H * Dd.H) + (Db.L * Dd.L)) << 1) + 0x8000) >> 16)))
	Dn.WL = (U20)SAT16 ((((((Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16) + ((((-(Db.H * Dd.L) - (Db.L * Dd.H)) << 1) + 0x8000) >> 16)))	

8	**MPYCXD.PP.SR.2W Da:Db,Dc:Dd,Dn**	**Complex fractional dot product with rounding and saturation**

	Dn.WH = (U20)SAT16 ((((((Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16) + ((((Db.H * Dd.H) - (Db.L * Dd.L)) << 1) + 0x8000) >> 16)))
	Dn.WL = (U20)SAT16 ((((((Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16) + ((((Db.H * Dd.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16)))	

Explicit Operands

Operand	Permitted Values	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 4
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPYCXD.nX 16x16 bit Complex Dot-Product into 40-bit Result (DALU)

General Description

Performs two 16x16 bit complex multiplications, sum or subtract the products into 40-bit results.
Source complex numbers are packed in single registers, while destination complex numbers resides in a register pair.

Flag Options

Flag	Position	Description
C	flg1	Complex Conjugate
PN	flg1	Positive Negative
PP	flg1	Positive Positive
I	flg2	Integer arithmetic
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MPYCXD.CPN.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate, second product is negated**

	`Dm = ((((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Db.H * Dd.H) + (Db.L * Dd.L))) << 1)[39:0]`
	`Dn = ((((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Db.H * Dd.L) - (Db.L * Dd.H))) << 1)[39:0]`	

2	**MPYCXD.CPN.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate, second product is negated, single first operand**

	`Dm = ((((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Da.H * Dd.H) + (Da.L * Dd.L))) << 1)[39:0]`
	`Dn = ((((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Da.H * Dd.L) - (Da.L * Dd.H))) << 1)[39:0]`	

3	**MPYCXD.CPP.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate**

	`Dm = ((((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Db.H * Dd.H) + (Db.L * Dd.L))) << 1)[39:0]`
	`Dn = ((((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Db.H * Dd.L) - (Db.L * Dd.H))) << 1)[39:0]`	

4	**MPYCXD.CPP.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate, single first operand**

	`Dm = ((((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Da.H * Dd.H) + (Da.L * Dd.L))) << 1)[39:0]`
	`Dn = ((((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Da.H * Dd.L) - (Da.L * Dd.H))) << 1)[39:0]`	

5	**MPYCXD.PN.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product, second product is negated**

	`Dm = ((((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Db.H * Dd.H) - (Db.L * Dd.L))) << 1)[39:0]`
	`Dn = ((((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Db.H * Dd.L) + (Db.L * Dd.H))) << 1)[39:0]`	

Table continues on the next page...

#	Syntax	Description
6	**MPYCXD.PN.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product, second product is negated, single first operand**

	`Dm = ((((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Da.H * Dd.H) - (Da.L * Dd.L))) << 1)[39:0]`
	`Dn = ((((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Da.H * Dd.L) + (Da.L * Dd.H))) << 1)[39:0]`	

7	**MPYCXD.PP.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product**

	`Dm = ((((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Db.H * Dd.H) - (Db.L * Dd.L))) << 1)[39:0]`
	`Dn = (((Da.H * Dc.L) + (Da.L * Dc.H) + (Db.H * Dd.L) + (Db.L * Dd.H)) << 1)[39:0]`	

8	**MPYCXD.PP.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product, single first operand**

	`Dm = ((((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Da.H * Dd.H) - (Da.L * Dd.L))) << 1)[39:0]`
	`Dn = (((Da.H * Dc.L) + (Da.L * Dc.H) + (Da.H * Dd.L) + (Da.L * Dd.H)) << 1)[39:0]`	

9	**MPYCXD.CPN.I.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate, second product is negated**

	`Dm = (((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Db.H * Dd.H) + (Db.L * Dd.L)))[39:0]`
	`Dn = (((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Db.H * Dd.L) - (Db.L * Dd.H)))[39:0]`	

10	**MPYCXD.CPN.I.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate, second product is negated, single first operand**

	`Dm = (((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Da.H * Dd.H) + (Da.L * Dd.L)))[39:0]`
	`Dn = (((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Da.H * Dd.L) - (Da.L * Dd.H)))[39:0]`	

11	**MPYCXD.CPP.I.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate**

	`Dm = (((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Db.H * Dd.H) + (Db.L * Dd.L)))[39:0]`
	`Dn = (((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Db.H * Dd.L) - (Db.L * Dd.H)))[39:0]`	

12	**MPYCXD.CPP.I.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate, single first operand**

	`Dm = (((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Da.H * Dd.H) + (Da.L * Dd.L)))[39:0]`
	`Dn = (((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Da.H * Dd.L) - (Da.L * Dd.H)))[39:0]`	

13	**MPYCXD.PN.I.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product, second product is negated**

	`Dm = (((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Db.H * Dd.H) - (Db.L * Dd.L)))[39:0]`
	`Dn = (((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Db.H * Dd.L) + (Db.L * Dd.H)))[39:0]`	

14	**MPYCXD.PN.I.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product, second product is negated, single first operand**

	`Dm = (((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Da.H * Dd.H) - (Da.L * Dd.L)))[39:0]`
	`Dn = (((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Da.H * Dd.L) + (Da.L * Dd.H)))[39:0]`	

15	**MPYCXD.PP.I.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product**

	`Dm = (((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Db.H * Dd.H) - (Db.L * Dd.L)))[39:0]`
	`Dn = ((Da.H * Dc.L) + (Da.L * Dc.H) + (Db.H * Dd.L) + (Db.L * Dd.H))[39:0]`	

16	**MPYCXD.PP.I.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product, single first operand**

	`Dm = (((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Da.H * Dd.H) - (Da.L * Dd.L)))[39:0]`
	`Dn = ((Da.H * Dc.L) + (Da.L * Dc.H) + (Da.H * Dd.L) + (Da.L * Dd.H))[39:0]`	

17	**MPYCXD.CPN.IS.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate with saturation, second product is negated**

Table continues on the next page...

#	Syntax	Description

	Dm = SAT32(((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Db.H * Dd.H) + (Db.L * Dd.L))))
	Dn = SAT32(((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Db.H * Dd.L) - (Db.L * Dd.H))))	

18	**MPYCXD.CPN.IS.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate with saturation, second product is negated, single first operand**

	Dm = SAT32(((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Da.H * Dd.H) + (Da.L * Dd.L))))
	Dn = SAT32(((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Da.H * Dd.L) - (Da.L * Dd.H))))	

19	**MPYCXD.CPP.IS.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate with saturation**

	Dm = SAT32(((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Db.H * Dd.H) + (Db.L * Dd.L))))
	Dn = SAT32(((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Db.H * Dd.L) - (Db.L * Dd.H))))	

20	**MPYCXD.CPP.IS.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product by conjugate with saturation, single first operand**

	Dm = SAT32(((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Da.H * Dd.H) + (Da.L * Dd.L))))
	Dn = SAT32(((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Da.H * Dd.L) - (Da.L * Dd.H))))	

21	**MPYCXD.PN.IS.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product with saturation, second product is negated**

	Dm = SAT32(((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Db.H * Dd.H) - (Db.L * Dd.L))))
	Dn = SAT32(((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Db.H * Dd.L) + (Db.L * Dd.H))))	

22	**MPYCXD.PN.IS.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product with saturation, second product is negated, single first operand**

	Dm = SAT32(((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Da.H * Dd.H) - (Da.L * Dd.L))))
	Dn = SAT32(((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Da.H * Dd.L) + (Da.L * Dd.H))))	

23	**MPYCXD.PP.IS.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex integer dot product with saturation**

	Dm = SAT32(((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Db.H * Dd.H) - (Db.L * Dd.L))))
	Dn = SAT32(((Da.H * Dc.L) + (Da.L * Dc.H) + (Db.H * Dd.L) + (Db.L * Dd.H)))	

24	**MPYCXD.PP.IS.2X Da,Dc:Dd,Dm:Dn**	**Complex integer dot product with saturation, single first operand**

	Dm = SAT32(((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Da.H * Dd.H) - (Da.L * Dd.L))))
	Dn = SAT32(((Da.H * Dc.L) + (Da.L * Dc.H) + (Da.H * Dd.L) + (Da.L * Dd.H)))	

25	**MPYCXD.CPN.S.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate with saturation, second product is negated**

	Dm = SAT32((((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Db.H * Dd.H) + (Db.L * Dd.L))) << 1))
	Dn = SAT32((((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Db.H * Dd.L) - (Db.L * Dd.H))) << 1))	

26	**MPYCXD.CPN.S.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate with saturation, second product is negated, single first operand**

	Dm = SAT32((((Da.H * Dc.H) + (Da.L * Dc.L)) - ((Da.H * Dd.H) + (Da.L * Dd.L))) << 1))
	Dn = SAT32((((Da.H * Dc.L) - (Da.L * Dc.H)) - ((Da.H * Dd.L) - (Da.L * Dd.H))) << 1))	

27	**MPYCXD.CPP.S.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate with saturation**

	Dm = SAT32((((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Db.H * Dd.H) + (Db.L * Dd.L))) << 1))
	Dn = SAT32((((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Db.H * Dd.L) - (Db.L * Dd.H))) << 1))	

Table continues on the next page...

#	Syntax	Description
28	**MPYCXD.CPP.S.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product by conjugate with saturation, single first operand**

	`Dm = SAT32((((Da.H * Dc.H) + (Da.L * Dc.L)) + ((Da.H * Dd.H) + (Da.L * Dd.L))) << 1))`
	`Dn = SAT32((((Da.H * Dc.L) - (Da.L * Dc.H)) + ((Da.H * Dd.L) - (Da.L * Dd.H))) << 1))`	

29	**MPYCXD.PN.S.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product with saturation, second product is negated**

	`Dm = SAT32((((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Db.H * Dd.H) - (Db.L * Dd.L))) << 1))`
	`Dn = SAT32((((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Db.H * Dd.L) + (Db.L * Dd.H))) << 1))`	

30	**MPYCXD.PN.S.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product with saturation, second product is negated, single first operand**

	`Dm = SAT32((((Da.H * Dc.H) - (Da.L * Dc.L)) - ((Da.H * Dd.H) - (Da.L * Dd.L))) << 1))`
	`Dn = SAT32((((Da.H * Dc.L) + (Da.L * Dc.H)) - ((Da.H * Dd.L) + (Da.L * Dd.H))) << 1))`	

31	**MPYCXD.PP.S.2X Da:Db,Dc:Dd,Dm:Dn**	**Complex fractional dot product with saturation**

	`Dm = SAT32((((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Db.H * Dd.H) - (Db.L * Dd.L))) << 1))`
	`Dn = SAT32(((Da.H * Dc.L) + (Da.L * Dc.H) + (Db.H * Dd.L) + (Db.L * Dd.H)) << 1))`	

32	**MPYCXD.PP.S.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional dot product with saturation, single first operand**

	`Dm = SAT32((((Da.H * Dc.H) - (Da.L * Dc.L)) + ((Da.H * Dd.H) - (Da.L * Dd.L))) << 1))`
	`Dn = SAT32(((Da.H * Dc.L) + (Da.L * Dc.H) + (Da.H * Dd.L) + (Da.L * Dd.H)) << 1))`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Affected By Instructions

Field	Relevant Variants
SR.SAT	17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	9, 10, 12, 14, 16, 17, 18, 20, 22, 24
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPYCXM.nX 16x32 bit Complex Multiply into 40-bit Result (DALU)

General Description

Performs 16x32 bit complex multiply into 40-bit results.
The 32/40-bit complex numbers resides in a register pair, while the 16-bit complex number is packed in a single register.

Flag Options

Flag	Position	Description
C	flg1	Complex Conjugate
I	flg2	Integer arithmetic
R	flg2	Round multiplication result
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MPYCXM.IR.2X Da,Dc:Dd,Dm:Dn**	**Complex integer multiply with rounding**

	`Dm = (((Da.H * Dc.M) - (Da.L * Dd.M) + 0x8000) >> 16)[39:0]`
	`Dn = (((Da.H * Dd.M) + (Da.L * Dc.M) + 0x8000) >> 16)[39:0]`	

2	**MPYCXM.C.IR.2X Da,Dc:Dd,Dm:Dn**	**Complex integer multiply by conjugate with rounding**

	`Dm = (((Da.H * Dc.M) + (Da.L * Dd.M) + 0x8000) >> 16)[39:0]`
	`Dn = (((Da.H * Dd.M) - (Da.L * Dc.M) + 0x8000) >> 16)[39:0]`	

3	**MPYCXM.ISR.2X Da,Dc:Dd,Dm:Dn**	**Complex integer multiply with rounding and saturation**

	`Dm = SAT32((((Da.H * Dc.M) - (Da.L * Dd.M) + 0x8000) >> 16))`
	`Dn = SAT32((((Da.H * Dd.M) + (Da.L * Dc.M) + 0x8000) >> 16))`	

4	**MPYCXM.C.ISR.2X Da,Dc:Dd,Dm:Dn**	**Complex integer multiply by conjugate with rounding and saturation**

	`Dm = SAT32((((Da.H * Dc.M) + (Da.L * Dd.M) + 0x8000) >> 16))`
	`Dn = SAT32((((Da.H * Dd.M) - (Da.L * Dc.M) + 0x8000) >> 16))`	

5	**MPYCXM.R.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional multiply with rounding**

	`Dm = ((((Da.H * Dc.M) << 1) - ((Da.L * Dd.M) << 1) + 0x8000) >> 16)[39:0]`
	`Dn = ((((Da.H * Dd.M) << 1) + ((Da.L * Dc.M) << 1) + 0x8000) >> 16)[39:0]`	

Table continues on the next page...

#	Syntax	Description
6	**MPYCXM.C.R.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional multiply by conjugate with rounding**

	`Dm = ((((Da.H * Dc.M) << 1) + ((Da.L * Dd.M) << 1) + 0x8000) >> 16)[39:0]`
	`Dn = ((((Da.H * Dd.M) << 1) - ((Da.L * Dc.M) << 1) + 0x8000) >> 16)[39:0]`	

7	**MPYCXM.SR.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional multiply with rounding and saturation**

	`Dm = SAT32(((((Da.H * Dc.M) << 1) - ((Da.L * Dd.M) << 1) + 0x8000) >> 16))`
	`Dn = SAT32(((((Da.H * Dd.M) << 1) + ((Da.L * Dc.M) << 1) + 0x8000) >> 16))`	

8	**MPYCXM.C.SR.2X Da,Dc:Dd,Dm:Dn**	**Complex fractional multiply by conjugate with rounding and saturation**

	`Dm = SAT32(((((Da.H * Dc.M) << 1) + ((Da.L * Dd.M) << 1) + 0x8000) >> 16))`
	`Dn = SAT32(((((Da.H * Dd.M) << 1) - ((Da.L * Dc.M) << 1) + 0x8000) >> 16))`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$

Affected By Instructions

Field	Relevant Variants
SR.SAT	3, 4, 7, 8

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	3, 4
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPYD.nT 16x16 bit Dot Product into 20-bit Result (DALU)

General Description

Performs SIMD2 16x16 bit dot product into 20-bit results.
Supports Real, Imaginary, and Conjugate-Imaginary variations.

Flag Options

Flag	Position	Description
C	flg1	Complex Conjugate
IM	flg1	Complex Imaginary
RE	flg1	Complex Real
I	flg2	Integer arithmetic
R	flg2	Round multiplication result

Instruction Variants

#	Syntax	Description
1	**MPYD.IR.2T Da:Db,Dc:Dd,Dn**	**Integer dot product with rounding, high by high and low by low parts**

	Dn.WH = (((Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16)[19:0]
	Dn.WL = (((Db.L * Dd.L) + (Db.H * Dd.H)) + 0x8000) >> 16)[19:0]	

2	**MPYD.CIM.IR.2T Da:Db,Dc:Dd,Dn**	**Integer dot product with rounding, selected input parts produce the multiplied-by-conjugate imaginary parts**

	Dn.WH = (((-(Da.L * Dc.H) + (Da.H * Dc.L)) + 0x8000) >> 16)[19:0]
	Dn.WL = (((-(Db.L * Dd.H) + (Db.H * Dd.L)) + 0x8000) >> 16)[19:0]	

3	**MPYD.IM.IR.2T Da:Db,Dc:Dd,Dn**	**Integer dot product with rounding, selected input parts produce the imaginary parts**

	Dn.WH = (((Da.L * Dc.H) + (Da.H * Dc.L)) + 0x8000) >> 16)[19:0]
	Dn.WL = (((Db.L * Dd.H) + (Db.H * Dd.L)) + 0x8000) >> 16)[19:0]	

4	**MPYD.RE.IR.2T Da:Db,Dc:Dd,Dn**	**Integer dot product with rounding, selected input parts produce the real parts**

	Dn.WH = (((Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16)[19:0]
	Dn.WL = (((Db.H * Dd.H) - (Db.L * Dd.L)) + 0x8000) >> 16)[19:0]	

5	**MPYD.R.2T Da:Db,Dc:Dd,Dn**	**fractional dot product with rounding, high by high and low by low parts**

	Dn.WH = ((((Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16)[19:0]
	Dn.WL = ((((Db.L * Dd.L) + (Db.H * Dd.H)) << 1) + 0x8000) >> 16)[19:0]	

Table continues on the next page...

#	Syntax	Description
6	**MPYD.CIM.R.2T Da:Db,Dc:Dd,Dn**	**fractional dot product with rounding, selected input parts produce the multiplied-by-conjugate imaginary parts**

	`Dn.WH = ((((-(Da.L * Dc.H) + (Da.H * Dc.L)) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WL = ((((-(Db.L * Dd.H) + (Db.H * Dd.L)) << 1) + 0x8000) >> 16)[19:0]`	

7	**MPYD.IM.R.2T Da:Db,Dc:Dd,Dn**	**fractional dot product with rounding, selected input parts produce the imaginary parts**

	`Dn.WH = (((((Da.L * Dc.H) + (Da.H * Dc.L)) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WL = (((((Db.L * Dd.H) + (Db.H * Dd.L)) << 1) + 0x8000) >> 16)[19:0]`	

8	**MPYD.RE.R.2T Da:Db,Dc:Dd,Dn**	**fractional dot product with rounding, selected input parts produce the real parts**

	`Dn.WH = (((((Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WL = (((((Db.H * Dd.H) - (Db.L * Dd.L)) << 1) + 0x8000) >> 16)[19:0]`	

Explicit Operands

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3, 4
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPYD.nW 16x16 bit Dot Product into 16-bit Result with Saturation (DALU)

General Description

Performs SIMD2 16x16 bit dot product into 16-bit results.
Supports Real, Imaginary, and Conjugate-Imaginary variations.

Flag Options

Flag	Position	Description
C	flg1	Complex Conjugate
IM	flg1	Complex Imaginary
RE	flg1	Complex Real
I	flg2	Integer arithmetic
R	flg2	Round multiplication result
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MPYD.ISR.2W Da:Db,Dc:Dd,Dn**	**Integer dot product with rounding and saturate, high by high and low by low parts**

	`Dn.H = SAT16 ((((Da.H * Dc.H) + (Da.L * Dc.L)) + 0x8000) >> 16))[15:0]`
	`Dn.L = SAT16 ((((Db.L * Dd.L) + (Db.H * Dd.H)) + 0x8000) >> 16))[15:0]`
	`Dn.E = 0`	

2	**MPYD.CIM.ISR.2W Da:Db,Dc:Dd,Dn**	**Integer dot product with rounding and saturate, selected input parts produce the multiplied-by-conjugate imaginary parts**

	`Dn.H = SAT16 ((((-(Da.L * Dc.H) + (Da.H * Dc.L)) + 0x8000) >> 16))[15:0]`
	`Dn.L = SAT16 ((((-(Db.L * Dd.H) + (Db.H * Dd.L)) + 0x8000) >> 16))[15:0]`
	`Dn.E = 0`	

3	**MPYD.IM.ISR.2W Da:Db,Dc:Dd,Dn**	**Integer dot product with rounding and saturate, selected input parts produce the imaginary parts**

	`Dn.H = SAT16 ((((Da.L * Dc.H) + (Da.H * Dc.L)) + 0x8000) >> 16))[15:0]`
	`Dn.L = SAT16 ((((Db.L * Dd.H) + (Db.H * Dd.L)) + 0x8000) >> 16))[15:0]`
	`Dn.E = 0`	

4	**MPYD.RE.ISR.2W Da:Db,Dc:Dd,Dn**	**Integer dot product with rounding and saturate, selected input parts produce the real parts**

	`Dn.H = SAT16 ((((Da.H * Dc.H) - (Da.L * Dc.L)) + 0x8000) >> 16))[15:0]`
	`Dn.L = SAT16 ((((Db.H * Dd.H) - (Db.L * Dd.L)) + 0x8000) >> 16))[15:0]`
	`Dn.E = 0`	

5	**MPYD.SR.2W Da:Db,Dc:Dd,Dn**	**fractional dot product with rounding and saturate, high by high and low by low parts**

Table continues on the next page...

#	Syntax	Description

	`Dn.H = SAT16 (((((Da.H * Dc.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[15:0]`
	`Dn.L = SAT16 (((((Db.L * Dd.L) + (Db.H * Dd.H)) << 1) + 0x8000) >> 16))[15:0]`
	`Dn.E = 0`	

6	**MPYD.CIM.SR.2W Da:Db,Dc:Dd,Dn**	**fractional dot product with rounding and saturate, selected input parts produce the multiplied-by-conjugate imaginary parts**

	`Dn.H = SAT16 (((((-(Da.L * Dc.H) + (Da.H * Dc.L)) << 1) + 0x8000) >> 16))[15:0]`
	`Dn.L = SAT16 (((((-(Db.L * Dd.H) + (Db.H * Dd.L)) << 1) + 0x8000) >> 16))[15:0]`
	`Dn.E = 0`	

7	**MPYD.IM.SR.2W Da:Db,Dc:Dd,Dn**	**fractional dot product with rounding and saturate, selected input parts produce the imaginary parts**

	`Dn.H = SAT16 (((((Da.L * Dc.H) + (Da.H * Dc.L)) << 1) + 0x8000) >> 16))[15:0]`
	`Dn.L = SAT16 (((((Db.L * Dd.H) + (Db.H * Dd.L)) << 1) + 0x8000) >> 16))[15:0]`
	`Dn.E = 0`	

8	**MPYD.RE.SR.2W Da:Db,Dc:Dd,Dn**	**fractional dot product with rounding and saturate, selected input parts produce the real parts**

	`Dn.H = SAT16 (((((Da.H * Dc.H) - (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[15:0]`
	`Dn.L = SAT16 (((((Db.H * Dd.H) - (Db.L * Dd.L)) << 1) + 0x8000) >> 16))[15:0]`
	`Dn.E = 0`	

Explicit Operands

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3, 4
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPYD.nX 16x16 bit Dot Product into 40-bit Result (DALU)

General Description

Performs single/SIMD2 16x16 bit dot product into 40-bit results.
Supports Real, Imaginary, and Conjugate-Imaginary variations.

Flag Options

Flag	Position	Description
C	flg1	Complex Conjugate
IM	flg1	Complex Imaginary
RE	flg1	Complex Real
I	flg2	Integer arithmetic
LEG	flg2	Legacy instruction
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MPYD.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product**

	`Dm = (((Da.L * Dc.L) + (Da.H * Dc.H)) << 1)[39:0]`
	`Dn = (((Db.L * Dd.L) + (Db.H * Dd.H)) << 1)[39:0]`	

2	**MPYD.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product, single first operand**

	`Dm = (((Da.L * Dc.L) + (Da.H * Dc.H)) << 1)[39:0]`
	`Dn = (((Da.L * Dd.L) + (Da.H * Dd.H)) << 1)[39:0]`	

3	**MPYD.X Da,Db,Dn**	**Single fractional dot product**
	`Dn = (((Da.L * Db.L) + (Da.H * Db.H)) << 1)[39:0]`	
4	**MPYD.CIM.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product, selected input parts produce the multiplied-by-conjugate imaginary parts**

	`Dm = ((-(Da.L * Dc.H) + (Da.H * Dc.L)) << 1)[39:0]`
	`Dn = ((-(Db.L * Dd.H) + (Db.H * Dd.L)) << 1)[39:0]`	

5	**MPYD.CIM.X Da,Db,Dn**	**Single fractional dot product, selected input parts produce the multiplied-by-conjugate imaginary parts**
	`Dn = ((-(Da.L * Db.H) + (Da.H * Db.L)) << 1)[39:0]`	

Table continues on the next page...

#	Syntax	Description
6	**MPYD.IM.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product, selected input parts produce the imaginary parts**

	`Dm = (((Da.L * Dc.H) + (Da.H * Dc.L)) << 1)[39:0]`
	`Dn = (((Db.L * Dd.H) + (Db.H * Dd.L)) << 1)[39:0]`	

7	**MPYD.IM.X Da,Db,Dn**	**Single fractional dot product, selected input parts produce the imaginary parts**
	`Dn = (((Da.L * Db.H) + (Da.H * Db.L)) << 1)[39:0]`	
8	**MPYD.RE.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product, selected input parts produce the real parts**

	`Dm = (((Da.H * Dc.H) - (Da.L * Dc.L)) << 1)[39:0]`
	`Dn = (((Db.H * Dd.H) - (Db.L * Dd.L)) << 1)[39:0]`	

9	**MPYD.RE.X Da,Db,Dn**	**Single fractional dot product, selected input parts produce the real parts**
	`Dn = (((Da.H * Db.H) - (Da.L * Db.L)) << 1)[39:0]`	
10	**MPYD.I.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product**

	`Dm = ((Da.L * Dc.L) + (Da.H * Dc.H))[39:0]`
	`Dn = ((Db.L * Dd.L) + (Db.H * Dd.H))[39:0]`	

11	**MPYD.I.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product, single first operand**

	`Dm = ((Da.L * Dc.L) + (Da.H * Dc.H))[39:0]`
	`Dn = ((Da.L * Dd.L) + (Da.H * Dd.H))[39:0]`	

12	**MPYD.CIM.I.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product, selected input parts produce the multiplied-by-conjugate imaginary parts**

	`Dm = (-(Da.L * Dc.H) + (Da.H * Dc.L))[39:0]`
	`Dn = (-(Db.L * Dd.H) + (Db.H * Dd.L))[39:0]`	

13	**MPYD.IM.I.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product, selected input parts produce the imaginary parts**

	`Dm = ((Da.L * Dc.H) + (Da.H * Dc.L))[39:0]`
	`Dn = ((Db.L * Dd.H) + (Db.H * Dd.L))[39:0]`	

14	**MPYD.RE.I.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product, selected input parts produce the real parts**

	`Dm = ((Da.H * Dc.H) - (Da.L * Dc.L))[39:0]`
	`Dn = ((Db.H * Dd.H) - (Db.L * Dd.L))[39:0]`	

15	**MPYD.IS.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product with saturation**

	`Dm = SAT32(((Da.H * Dc.H) + (Da.L * Dc.L)))`
	`Dn = SAT32(((Db.H * Dd.H) + (Db.L * Dd.L)))`	

16	**MPYD.IS.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product with saturation, single first operand**

	`Dm = SAT32(((Da.H * Dc.H) + (Da.L * Dc.L)))`
	`Dn = SAT32(((Da.H * Dd.H) + (Da.L * Dd.L)))`	

17	**MPYD.CIM.IS.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product with saturation, selected input parts produce the multiplied-by-conjugate imaginary parts**

Table continues on the next page...

#	Syntax	Description

	`Dm = SAT32((-(Da.L * Dc.H) + (Da.H * Dc.L)))`
	`Dn = SAT32((-(Db.L * Dd.H) + (Db.H * Dd.L)))`	

18	**MPYD.IM.IS.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product with saturation, selected input parts produce the imaginary parts**

	`Dm = SAT32(((Da.L * Dc.H) + (Da.H * Dc.L)))`
	`Dn = SAT32(((Db.L * Dd.H) + (Db.H * Dd.L)))`	

19	**MPYD.RE.IS.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product with saturation, selected input parts produce the real parts**

	`Dm = SAT32(((Da.H * Dc.H) - (Da.L * Dc.L)))`
	`Dn = SAT32(((Db.H * Dd.H) - (Db.L * Dd.L)))`	

20	**MPYD.LEG.X Da,Db,Dn**	**Single fractional dot product, saturating only if SR.SM is set**
	`Dn = srSAT32 ((((Da.L * Db.L) + (Da.H * Db.H)) << 1))`	
21	**MPYD.LEG.X -Da,Db,Dn**	**Single fractional dot product with negate, saturating only if SR.SM is set**
	`Dn = srSAT32 ((((-(Da.L * Db.L)) - (Da.H * Db.H)) << 1))`	
22	**MPYD.CIM.LEG.X -Da,Db,Dn**	**Single fractional dot product, selected input parts produce the multiplied-by-conjugate imaginary parts, saturating only if SR.SM is set**
	`Dn = srSAT32((((Da.L * Db.H) - (Da.H * Db.L)) << 1))`	
23	**MPYD.IM.LEG.X Da,Db,Dn**	**Single fractional dot product, selected input parts produce the imaginary parts, saturating only if SR.SM is set**
	`Dn = srSAT32((((Da.L * Db.H) + (Da.H * Db.L)) << 1))`	
24	**MPYD.RE.LEG.X Da,Db,Dn**	**Single fractional dot product, selected input parts produce the real parts, saturating only if SR.SM is set**
	`Dn = srSAT32((((Da.H * Db.H) - (Da.L * Db.L)) << 1))`	
25	**MPYD.S.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product with saturation**

	`Dm = SAT32((((Da.L * Dc.L) + (Da.H * Dc.H)) << 1))`
	`Dn = SAT32((((Db.L * Dd.L) + (Db.H * Dd.H)) << 1))`	

26	**MPYD.S.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product with saturation, single first operand**

	`Dm = SAT32((((Da.L * Dc.L) + (Da.H * Dc.H)) << 1))`
	`Dn = SAT32((((Da.L * Dd.L) + (Da.H * Dd.H)) << 1))`	

27	**MPYD.S.X Da,Db,Dn**	**Single fractional dot product with saturation**
	`Dn = SAT32((((Da.L * Db.L) + (Da.H * Db.H)) << 1))`	

Table continues on the next page...

#	Syntax	Description
28	**MPYD.CIM.S.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product with saturation, selected input parts produce the multiplied-by-conjugate imaginary parts**

	`Dm = SAT32(((-(Da.L * Dc.H) + (Da.H * Dc.L)) << 1))`
	`Dn = SAT32(((-(Db.L * Dd.H) + (Db.H * Dd.L)) << 1))`	

29	**MPYD.CIM.S.X Da,Db,Dn**	**Single fractional dot product with saturation, selected input parts produce the multiplied-by-conjugate imaginary parts**
	`Dn = SAT32(((-(Da.L * Db.H) + (Da.H * Db.L)) << 1))`	
30	**MPYD.IM.S.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product with saturation, selected input parts produce the imaginary parts**

	`Dm = SAT32((((Da.L * Dc.H) + (Da.H * Dc.L)) << 1))`
	`Dn = SAT32((((Db.L * Dd.H) + (Db.H * Dd.L)) << 1))`	

31	**MPYD.IM.S.X Da,Db,Dn**	**Single fractional dot product with saturation, selected input parts produce the imaginary parts**
	`Dn = SAT32((((Da.L * Db.H) + (Da.H * Db.L)) << 1))`	
32	**MPYD.RE.S.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product with saturation, selected input parts produce the real parts**

	`Dm = SAT32((((Da.H * Dc.H) - (Da.L * Dc.L)) << 1))`
	`Dn = SAT32((((Db.H * Dd.H) - (Db.L * Dd.L)) << 1))`	

33	**MPYD.RE.S.X Da,Db,Dn**	**Single fractional dot product with saturation, selected input parts produce the real parts**
	`Dn = SAT32((((Da.H * Db.H) - (Da.L * Db.L)) << 1))`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \leq n \leq 62$
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \leq n \leq 62$
Dm:Dn	D_n:D_{n+1}	$0 \leq n \leq 62$
Dn	D0-D63	

Affect Instructions

Field	Relevant Variants	Comments
SR.SM	20, 21, 22, 23, 24	

Affected By Instructions

Field	Relevant Variants
SR.SAT	15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	11, 12, 13, 14, 15, 16, 17, 18, 19
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPYDCF.nT 16x16 bit Dot Product Real by Complex FIR into 20-bit Result (DALU)

General Description

Performs SIMD4 16x16 bit dot product into 20-bit results. Operands parts are optimized for Real by Complex FIR/correlation implementation.

Flag Options

Flag	Position	Description
H	flg1	High 16-bit portion
INV	flg1	Inverse order
L	flg1	Use the low portion
I	flg2	Integer arithmetic
R	flg2	Round multiplication result

Instruction Variants

#	Syntax	Description
1	**MPYDCF.H.IR.4T Da:Db,Dc:Dd,Dm:Dn**	**Integer dot product with rounding, using 3 high coefficients**

	`Dm.WH = ((((Da.H * Dc.L) + (Db.H * Dc.H)) + 0x8000) >> 16)[19:0]`
	`Dm.WL = ((((Da.L * Dc.L) + (Db.L * Dc.H)) + 0x8000) >> 16)[19:0]`
	`Dn.WH = ((((Da.H * Dd.H) + (Db.H * Dc.L)) + 0x8000) >> 16)[19:0]`
	`Dn.WL = ((((Da.L * Dd.H) + (Db.L * Dc.L)) + 0x8000) >> 16)[19:0]`	

2	**MPYDCF.INVH.IR.4T Da:Db,Dc:Dd,Dm:Dn**	**Integer dot product with rounding, using 3 high coefficients in inverse order**

	`Dm.WH = ((((Da.H * Dc.H) + (Db.H * Dc.L)) + 0x8000) >> 16)[19:0]`
	`Dm.WL = ((((Da.L * Dc.H) + (Db.L * Dc.L)) + 0x8000) >> 16)[19:0]`
	`Dn.WH = ((((Da.H * Dc.L) + (Db.H * Dd.H)) + 0x8000) >> 16)[19:0]`
	`Dn.WL = ((((Da.L * Dc.L) + (Db.L * Dd.H)) + 0x8000) >> 16)[19:0]`	

3	**MPYDCF.INVL.IR.4T Da:Db,Dc:Dd,Dm:Dn**	**Integer dot product with rounding, using 3 low coefficients in inverse order**

	`Dm.WH = ((((Da.H * Dc.L) + (Db.H * Dd.H)) + 0x8000) >> 16)[19:0]`
	`Dm.WL = ((((Da.L * Dc.L) + (Db.L * Dd.H)) + 0x8000) >> 16)[19:0]`
	`Dn.WH = ((((Da.H * Dd.H) + (Db.H * Dd.L)) + 0x8000) >> 16)[19:0]`
	`Dn.WL = ((((Da.L * Dd.H) + (Db.L * Dd.L)) + 0x8000) >> 16)[19:0]`	

Table continues on the next page...

#	Syntax	Description
4	**MPYDCF.L.IR.4T Da:Db,Dc:Dd,Dm:Dn**	**Integer dot product with rounding, using 3 low coefficients**

	`Dm.WH = ((((Da.H * Dd.H) + (Db.H * Dc.L)) + 0x8000) >> 16)[19:0]`
	`Dm.WL = ((((Da.L * Dd.H) + (Db.L * Dc.L)) + 0x8000) >> 16)[19:0]`
	`Dn.WH = ((((Da.H * Dd.L) + (Db.H * Dd.H)) + 0x8000) >> 16)[19:0]`
	`Dn.WL = ((((Da.L * Dd.L) + (Db.L * Dd.H)) + 0x8000) >> 16)[19:0]`	

5	**MPYDCF.H.R.4T Da:Db,Dc:Dd,Dm:Dn**	**Fractional dot product with rounding, using 3 high coefficients**

	`Dm.WH = (((((Da.H * Dc.L) + (Db.H * Dc.H)) << 1) + 0x8000) >> 16)[19:0]`
	`Dm.WL = (((((Da.L * Dc.L) + (Db.L * Dc.H)) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WH = (((((Da.H * Dd.H) + (Db.H * Dc.L)) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WL = (((((Da.L * Dd.H) + (Db.L * Dc.L)) << 1) + 0x8000) >> 16)[19:0]`	

6	**MPYDCF.INVH.R.4T Da:Db,Dc:Dd,Dm:Dn**	**Fractional dot product with rounding, using 3 high coefficients in inverse order**

	`Dm.WH = (((((Da.H * Dc.H) + (Db.H * Dc.L)) << 1) + 0x8000) >> 16)[19:0]`
	`Dm.WL = (((((Da.L * Dc.H) + (Db.L * Dc.L)) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WH = (((((Da.H * Dc.L) + (Db.H * Dd.H)) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WL = (((((Da.L * Dc.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16)[19:0]`	

7	**MPYDCF.INVL.R.4T Da:Db,Dc:Dd,Dm:Dn**	**Fractional dot product with rounding, using 3 low coefficients in inverse order**

	`Dm.WH = (((((Da.H * Dc.L) + (Db.H * Dd.H)) << 1) + 0x8000) >> 16)[19:0]`
	`Dm.WL = (((((Da.L * Dc.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WH = (((((Da.H * Dd.H) + (Db.H * Dd.L)) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WL = (((((Da.L * Dd.H) + (Db.L * Dd.L)) << 1) + 0x8000) >> 16)[19:0]`	

8	**MPYDCF.L.R.4T Da:Db,Dc:Dd,Dm:Dn**	**Fractional dot product with rounding, using 3 low coefficients**

	`Dm.WH = (((((Da.H * Dd.H) + (Db.H * Dc.L)) << 1) + 0x8000) >> 16)[19:0]`
	`Dm.WL = (((((Da.L * Dd.H) + (Db.L * Dc.L)) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WH = (((((Da.H * Dd.L) + (Db.H * Dd.H)) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WL = (((((Da.L * Dd.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16)[19:0]`	

Explicit Operands

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3, 4
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPYDCF.nX 16x16 bit Dot Product Real by Complex FIR into 40-bit Results (DALU)

General Description

Performs SIMD2 16x16 bit dot product into 40-bit results. Operands parts are optimized for Real by Complex FIR implementation.

Flag Options

Flag	Position	Description
INV	flg1	Inverse order
I	flg2	Integer arithmetic

Instruction Variants

#	Syntax	Description
1	**MPYDCF.2X Da,Dc:Dd,Dm:Dn**	**Fractional dot product**

	`Dm = (((Da.L * Dc.H) << 1) + ((Da.H * Dd.H) << 1))[39:0]`
	`Dn = (((Da.L * Dc.L) << 1) + ((Da.H * Dd.L) << 1))[39:0]`	

2	**MPYDCF.INV.2X Da,Dc:Dd,Dm:Dn**	**Fractional dot product, using coefficients in inverse order**

	`Dm = (((Da.L * Dd.H) << 1) + ((Da.H * Dc.H) << 1))[39:0]`
	`Dn = (((Da.L * Dd.L) << 1) + ((Da.H * Dc.L) << 1))[39:0]`	

3	**MPYDCF.I.2X Da,Dc:Dd,Dm:Dn**	**Integer dot product**

	`Dm = ((Da.L * Dc.H) + (Da.H * Dd.H))[39:0]`
	`Dn = ((Da.L * Dc.L) + (Da.H * Dd.L))[39:0]`	

4	**MPYDCF.INV.I.2X Da,Dc:Dd,Dm:Dn**	**Integer dot product, using coefficients in inverse order**

	`Dm = ((Da.L * Dd.H) + (Da.H * Dc.H))[39:0]`
	`Dn = ((Da.L * Dd.L) + (Da.H * Dc.L))[39:0]`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \leq n \leq 62$
Dm:Dn	D_n:D_{n+1}	$0 \leq n \leq 62$

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	3, 4
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPYDDCF.nT 16x16 bit Dot Product Real by Complex FIR with Decimation into 20-bit Result (DALU)

General Description

Performs SIMD4 16x16 bit dot product into 20-bit results. Operands parts are optimized for Real by Complex FIR with decimation implementation.

Flag Options

Flag	Position	Description
I	flg2	Integer arithmetic
R	flg2	Round multiplication result

Instruction Variants

#	Syntax	Description
1	**MPYDDCF.IR.4T Da:Db,Dc:Dd,Dm:Dn**	**Integer Real by Complex dot product FIR, with decimation**

```
Dm.WH = ((((Da.H * Dd.H) + (Da.L * Dc.H)) + 0x8000) >> 16)[19:0]
Dm.WL = ((((Da.H * Dd.L) + (Da.L * Dc.L)) + 0x8000) >> 16)[19:0]
Dn.WH = ((((Db.H * Dd.H) + (Db.L * Dc.H)) + 0x8000) >> 16)[19:0]
Dn.WL = ((((Db.H * Dd.L) + (Db.L * Dc.L)) + 0x8000) >> 16)[19:0]
```

#	Syntax	Description
2	**MPYDDCF.R.4T Da:Db,Dc:Dd,Dm:Dn**	**Fractional Real by Complex dot product FIR, with decimation**

```
Dm.WH = (((((Da.H * Dd.H) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16)[19:0]
Dm.WL = (((((Da.H * Dd.L) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16)[19:0]
Dn.WH = (((((Db.H * Dd.H) + (Db.L * Dc.H)) << 1) + 0x8000) >> 16)[19:0]
Dn.WL = (((((Db.H * Dd.L) + (Db.L * Dc.L)) << 1) + 0x8000) >> 16)[19:0]
```

Explicit Operands

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPYDDCF.nW 16x16 bit Dot Product Real by Complex FIR with Decimation into 16-bit Result with Saturation (DALU)

General Description

Performs SIMD4 16x16 bit dot product into 16-bit results. Operands parts are optimized for Real by Complex FIR with decimation implementation.

Flag Options

Flag	Position	Description
I	flg2	Integer arithmetic
R	flg2	Round multiplication result
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MPYDDCF.ISR.4W Da:Db,Dc:Dd,Dm:Dn**	**Integer Real by Complex dot product with rounding and saturation FIR, with decimation**

```
Dm.WH = (U20)SAT16 ((((Da.H * Dd.H) + (Da.L * Dc.H)) + 0x8000) >> 16))[15:0]
Dm.WL = (U20)SAT16 ((((Da.H * Dd.L) + (Da.L * Dc.L)) + 0x8000) >> 16))[15:0]
Dn.WH = (U20)SAT16 ((((Db.H * Dd.H) + (Db.L * Dc.H)) + 0x8000) >> 16))[15:0]
Dn.WL = (U20)SAT16 ((((Db.H * Dd.L) + (Db.L * Dc.L)) + 0x8000) >> 16))[15:0]
```

#	Syntax	Description
2	**MPYDDCF.SR.4W Da:Db,Dc:Dd,Dm:Dn**	**Fractional Real by Complex dot product with rounding and saturation FIR, with decimation**

```
Dm.WH = (U20)SAT16 (((((Da.H * Dd.H) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16))[15:0]
Dm.WL = (U20)SAT16 (((((Da.H * Dd.L) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[15:0]
Dn.WH = (U20)SAT16 (((((Db.H * Dd.H) + (Db.L * Dc.H)) << 1) + 0x8000) >> 16))[15:0]
Dn.WL = (U20)SAT16 (((((Db.H * Dd.L) + (Db.L * Dc.L)) << 1) + 0x8000) >> 16))[15:0]
```

Explicit Operands

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPYDRF.nT 16x16 bit Dot Product Real by Real FIR into 20-bit Result (DALU)

General Description

Performs SIMD2/SIMD4 16x16 bit dot product into 20-bit results. Operands parts are optimized for Real by Real FIR implementation.

Flag Options

Flag	Position	Description
H	flg1	High 16-bit portion
L	flg1	Use the low portion
I	flg2	Integer arithmetic
R	flg2	Round multiplication result

Instruction Variants

#	Syntax	Description
1	**MPYDRF.H.IR.2T Da,Dc:Dd,Dn**	**SIMD2 Integer dot product with rounding, using 3 high coefficients**

	`Dn.WH = (((Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16)[19:0]`
	`Dn.WL = (((Da.H * Dd.H) + (Da.L * Dc.L)) + 0x8000) >> 16)[19:0]`	

2	**MPYDRF.H.IR.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 Integer dot product with rounding, using 3 high coefficients**

	`Dm.WH = (((Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16)[19:0]`
	`Dm.WL = (((Da.H * Dd.H) + (Da.L * Dc.L)) + 0x8000) >> 16)[19:0]`
	`Dn.WH = (((Db.H * Dc.L) + (Db.L * Dc.H)) + 0x8000) >> 16)[19:0]`
	`Dn.WL = (((Db.H * Dd.H) + (Db.L * Dc.L)) + 0x8000) >> 16)[19:0]`	

3	**MPYDRF.L.IR.2T Da,Dc:Dd,Dn**	**SIMD2 Integer dot product with rounding, using 3 low coefficients**

	`Dn.WH = (((Da.H * Dd.H) + (Da.L * Dc.L)) + 0x8000) >> 16)[19:0]`
	`Dn.WL = (((Da.H * Dd.L) + (Da.L * Dd.H)) + 0x8000) >> 16)[19:0]`	

4	**MPYDRF.L.IR.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 Integer dot product with rounding, using 3 low coefficients**

	`Dm.WH = (((Da.H * Dd.H) + (Da.L * Dc.L)) + 0x8000) >> 16)[19:0]`
	`Dm.WL = (((Da.H * Dd.L) + (Da.L * Dd.H)) + 0x8000) >> 16)[19:0]`
	`Dn.WH = (((Db.H * Dd.H) + (Db.L * Dc.L)) + 0x8000) >> 16)[19:0]`
	`Dn.WL = (((Db.H * Dd.L) + (Db.L * Dd.H)) + 0x8000) >> 16)[19:0]`	

5	**MPYDRF.H.R.2T Da,Dc:Dd,Dn**	**SIMD2 fractional dot product with rounding, using 3 high coefficients**

	`Dn.WH = ((((Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WL = ((((Da.H * Dd.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16)[19:0]`	

Table continues on the next page...

#	Syntax	Description
6	**MPYDRF.H.R.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 fractional dot product with rounding, using 3 high coefficients**

	Dm.WH = (((((Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16)[19:0]
	Dm.WL = (((((Da.H * Dd.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16)[19:0]
	Dn.WH = (((((Db.H * Dc.L) + (Db.L * Dc.H)) << 1) + 0x8000) >> 16)[19:0]
	Dn.WL = (((((Db.H * Dd.H) + (Db.L * Dc.L)) << 1) + 0x8000) >> 16)[19:0]	

7	**MPYDRF.L.R.2T Da,Dc:Dd,Dn**	**SIMD2 fractional dot product with rounding, using 3 low coefficients**

	Dn.WH = (((((Da.H * Dd.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16)[19:0]
	Dn.WL = (((((Da.H * Dd.L) + (Da.L * Dd.H)) << 1) + 0x8000) >> 16)[19:0]	

8	**MPYDRF.L.R.4T Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 fractional dot product with rounding, using 3 low coefficients**

	Dm.WH = (((((Da.H * Dd.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16)[19:0]
	Dm.WL = (((((Da.H * Dd.L) + (Da.L * Dd.H)) << 1) + 0x8000) >> 16)[19:0]
	Dn.WH = (((((Db.H * Dd.H) + (Db.L * Dc.L)) << 1) + 0x8000) >> 16)[19:0]
	Dn.WL = (((((Db.H * Dd.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16)[19:0]	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3, 4
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPYDRF.nW 16x16 bit Dot Product Real by Real FIR into 16-bit Result with Saturation (DALU)

General Description

Performs SIMD2/SIMD4 16x16 bit dot product into 16-bit results. Operands parts are optimized for Real by Real FIR implementation.

Flag Options

Flag	Position	Description
H	flg1	High 16-bit portion
L	flg1	Use the low portion
I	flg2	Integer arithmetic
R	flg2	Round multiplication result
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MPYDRF.H.ISR.2W Da,Dc:Dd,Dn**	**SIMD2 Integer dot product with rounding and saturation, using 3 high coefficients**

	`Dn.H = SAT16 ((((Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16))[15:0]`
	`Dn.L = SAT16 ((((Da.H * Dd.H) + (Da.L * Dc.L)) + 0x8000) >> 16))[15:0]`
	`Dn.E = 0`	

2	**MPYDRF.H.ISR.4W Da:Db,Dc:Dd,Dm:Dn**	**SIMD4 Integer dot product with rounding and saturation, using 3 high coefficients**

	`Dm.H = SAT16 ((((Da.H * Dc.L) + (Da.L * Dc.H)) + 0x8000) >> 16))[15:0]`
	`Dm.L = SAT16 ((((Da.H * Dd.H) + (Da.L * Dc.L)) + 0x8000) >> 16))[15:0]`
	`Dm.E = 0`
	`Dn.H = SAT16 ((((Db.H * Dc.L) + (Db.L * Dc.H)) + 0x8000) >> 16))[15:0]`
	`Dn.L = SAT16 ((((Db.H * Dd.H) + (Db.L * Dc.L)) + 0x8000) >> 16))[15:0]`
	`Dn.E = 0`	

3	**MPYDRF.L.ISR.2W Da,Dc:Dd,Dn**	**SIMD2 Integer dot product with rounding and saturation, using 3 low coefficients**

	`Dn.H = SAT16 ((((Da.H * Dd.H) + (Da.L * Dc.L)) + 0x8000) >> 16))[15:0]`
	`Dn.L = SAT16 ((((Da.H * Dd.L) + (Da.L * Dd.H)) + 0x8000) >> 16))[15:0]`
	`Dn.E = 0`	

Table continues on the next page...

#	Syntax	Description
4	MPYDRF.L.ISR.4W Da:Db,Dc:Dd,Dm:Dn	SIMD4 Integer dot product with rounding and saturation, using 3 low coefficients

```
Dm.H = SAT16 ((((Da.H * Dd.H) + (Da.L * Dc.L)) + 0x8000) >> 16))[15:0]
Dm.L = SAT16 ((((Da.H * Dd.L) + (Da.L * Dd.H)) + 0x8000) >> 16))[15:0]
Dm.E = 0
Dn.H = SAT16 ((((Db.H * Dd.H) + (Db.L * Dc.L)) + 0x8000) >> 16))[15:0]
Dn.L = SAT16 ((((Db.H * Dd.L) + (Db.L * Dd.H)) + 0x8000) >> 16))[15:0]
Dn.E = 0
```

5	MPYDRF.H.SR.2W Da,Dc:Dd,Dn	SIMD2 fractional dot product with rounding and saturation, using 3 high coefficients

```
Dn.H = SAT16 (((((Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16))[15:0]
Dn.L = SAT16 (((((Da.H * Dd.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[15:0]
Dn.E = 0
```

6	MPYDRF.H.SR.4W Da:Db,Dc:Dd,Dm:Dn	SIMD4 fractional dot product with rounding and saturation, using 3 high coefficients

```
Dm.H = SAT16 (((((Da.H * Dc.L) + (Da.L * Dc.H)) << 1) + 0x8000) >> 16))[15:0]
Dm.L = SAT16 (((((Da.H * Dd.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[15:0]
Dm.E = 0
Dn.H = SAT16 (((((Db.H * Dc.L) + (Db.L * Dc.H)) << 1) + 0x8000) >> 16))[15:0]
Dn.L = SAT16 (((((Db.H * Dd.H) + (Db.L * Dc.L)) << 1) + 0x8000) >> 16))[15:0]
Dn.E = 0
```

7	MPYDRF.L.SR.2W Da,Dc:Dd,Dn	SIMD2 fractional dot product with rounding and saturation, using 3 low coefficients

```
Dn.H = SAT16 (((((Da.H * Dd.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[15:0]
Dn.L = SAT16 (((((Da.H * Dd.L) + (Da.L * Dd.H)) << 1) + 0x8000) >> 16))[15:0]
Dn.E = 0
```

8	MPYDRF.L.SR.4W Da:Db,Dc:Dd,Dm:Dn	SIMD4 fractional dot product with rounding and saturation, using 3 low coefficients

```
Dm.H = SAT16 (((((Da.H * Dd.H) + (Da.L * Dc.L)) << 1) + 0x8000) >> 16))[15:0]
Dm.L = SAT16 (((((Da.H * Dd.L) + (Da.L * Dd.H)) << 1) + 0x8000) >> 16))[15:0]
Dm.E = 0
Dn.H = SAT16 (((((Db.H * Dd.H) + (Db.L * Dc.L)) << 1) + 0x8000) >> 16))[15:0]
Dn.L = SAT16 (((((Db.H * Dd.L) + (Db.L * Dd.H)) << 1) + 0x8000) >> 16))[15:0]
Dn.E = 0
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3, 4
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPYDRF.nX 16x16 bit Dot Product Real by Real FIR into 40-bit Result (DALU)

General Description

Performs SIMD2 16x16 bit dot product into 40-bit results. Operands parts are optimized for Real by Real FIR/ correlation implementation.

Flag Options

Flag	Position	Description
H	flg1	High 16-bit portion
INV	flg1	Inverse order
L	flg1	Use the low portion
I	flg2	Integer arithmetic
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MPYDRF.H.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product, using 3 high coefficients**

	`Dm = (((Da.H * Dc.L) + (Da.L * Dc.H)) << 1)[39:0]`
	`Dn = (((Da.H * Dd.H) + (Da.L * Dc.L)) << 1)[39:0]`	

2	**MPYDRF.INVH.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product, using 3 high coefficients in inverse order**

	`Dm = (((Da.H * Dc.H) + (Da.L * Dc.L)) << 1)[39:0]`
	`Dn = (((Da.H * Dc.L) + (Da.L * Dd.H)) << 1)[39:0]`	

3	**MPYDRF.INVL.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product, using 3 low coefficients in inverse order**

	`Dm = (((Da.H * Dc.L) + (Da.L * Dd.H)) << 1)[39:0]`
	`Dn = (((Da.H * Dd.H) + (Da.L * Dd.L)) << 1)[39:0]`	

4	**MPYDRF.L.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product, using 3 low coefficients**

	`Dm = (((Da.H * Dd.H) + (Da.L * Dc.L)) << 1)[39:0]`
	`Dn = (((Da.H * Dd.L) + (Da.L * Dd.H)) << 1)[39:0]`	

5	**MPYDRF.H.I.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product, using 3 high coefficients**

	`Dm = ((Da.H * Dc.L) + (Da.L * Dc.H))[39:0]`
	`Dn = ((Da.H * Dd.H) + (Da.L * Dc.L))[39:0]`	

Table continues on the next page...

#	Syntax	Description
6	**MPYDRF.INVH.I.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product, using 3 high coefficients in inverse order**

	Dm = ((Da.H * Dc.H) + (Da.L * Dc.L))[39:0]
	Dn = ((Da.H * Dc.L) + (Da.L * Dd.H))[39:0]	

7	**MPYDRF.INVL.I.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product, using 3 low coefficients in inverse order**

	Dm = ((Da.H * Dc.L) + (Da.L * Dd.H))[39:0]
	Dn = ((Da.H * Dd.H) + (Da.L * Dd.L))[39:0]	

8	**MPYDRF.L.I.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product, using 3 low coefficients**

	Dm = ((Da.H * Dd.H) + (Da.L * Dc.L))[39:0]
	Dn = ((Da.H * Dd.L) + (Da.L * Dd.H))[39:0]	

9	**MPYDRF.H.IS.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product with saturation, using 3 high coefficients**

	Dm = SAT32(((Da.L * Dc.H) + (Da.H * Dc.L)))
	Dn = SAT32(((Da.H * Dd.H) + (Da.L * Dc.L)))	

10	**MPYDRF.L.IS.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 integer dot product with saturation, using 3 low coefficients**

	Dm = SAT32(((Da.H * Dd.H) + (Da.L * Dc.L)))
	Dn = SAT32(((Da.H * Dd.L) + (Da.L * Dd.H)))	

11	**MPYDRF.H.S.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product with saturation, using 3 high coefficients**

	Dm = SAT32((((Da.H * Dc.L) + (Da.L * Dc.H)) << 1))
	Dn = SAT32((((Da.H * Dd.H) + (Da.L * Dc.L)) << 1))	

12	**MPYDRF.L.S.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 fractional dot product with saturation, using 3 low coefficients**

	Dm = SAT32((((Da.H * Dd.H) + (Da.L * Dc.L)) << 1))
	Dn = SAT32((((Da.H * Dd.L) + (Da.L * Dd.H)) << 1))	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Affected By Instructions

Field	Relevant Variants
SR.SAT	9, 10, 11, 12

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	2, 3, 5, 6, 7, 8, 9, 10
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPYEM32.nX Double Precision Multiply Assist (DALU)

General Description

Performs 16x16 multiply operation, treating the multiplicands as signed or unsigned. Some variants shift the result right by 16 bits. These instructions are usually used for higher precision multiplication emulation assist.

Flag Options

Flag	Position	Description
SU	flg1	Signed By Unsigned
US	flg1	Unsigned By Signed
UU	flg1	Unsigned By Unsigned
X	flg1	Cross between the arguments
I	flg2	Integer arithmetic

Instruction Variants

#	Syntax	Description
1	**MPYEM32.US.X Da.l,Db.h,Dn**	**Fractional unsigned by signed multiply**
	`Dn = ((S40)(((S40)Db.H) * ((U40)Da.L))) << 1`	
2	**MPYEM32.UU.X Da.l,Db.l,Dn**	**Fractional unsigned by unsigned multiply**
	`Dn = ((S40)(((U40)Da.L) * ((U40)Db.L))[32:0]) << 1`	
3	**MPYEM32.XSU.X Da,Db,Dn**	**Fractional dot product, unsigned by signed and signed by unsigned**
	`Dn = (((((S64)Db.H) * ((U64)Da.L)) + (((U64)Db.L) * ((S64)Da.H))) << 1)[39:0]`	
4	**MPYDEM32.SU.I.X Da,Db,Dn**	**Integer dot product, unsigned by signed and signed by unsigned and shifted left by 16**
	`Dn = (((((U64)Db.L) * ((S64)Da.H)) + (((S64)Db.H) * ((U64)Da.L))) << 16)[39:0]`	
5	**MPYEM32.SU.I.X Da.h,Db.l,Dn**	**Integer signed by unsigned multiply**
	`Dn = (S40)(((U40)Db.L) * ((S40)Da.H))`	
6	**MPYDEM32.UU.I.X Da,Db,Dn**	**Integer dot product, unsigned by unsigned and shifted left by 16**
	`Dn = (U40) ((((U16) Da.L * (U16)Db.H + (U16) Da.H * (U16) Db.L)) << 16) [31:0] )`	
7	**MPYEM32.UU.I.X Da.h,Db.l,Dn**	**Integer unsigned by unsigned multiply**
	`Dn = (U40)(((U40)Db.L) * ((U40)Da.H))`	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Db	D0-D63
Dn	D0-D63

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPYM.LL 16x32 bit Multiply into 64-bit Result (DALU)

General Description

Performs single 16x32 bit multiply, storing the 48-bit result in a register pair.

Flag Options

Flag	Position	Description
I	flg2	Integer arithmetic

Instruction Variants

#	Syntax	Description
1	**MPYM.I.LL Da.h,Db,Dm:Dn**	**Integer 32x16 bit multiply, using high part**

	`Dn = (U40)((S64)(Da.H) * (S64)Db.M)[31:0]`
	`Dm = (S40)((S64)(Da.H) * (S64)Db.M)[63:32]`	

2	**MPYM.I.LL Da.l,Db,Dm:Dn**	**Integer 32x16 bit multiply, using low part**

	`Dn = (U40)((S64)(Da.L) * (S64)Db.M)[31:0]`
	`Dm = (S40)((S64)(Da.L) * (S64)Db.M)[63:32]`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Db	D0-D63	
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPYM.nL 16x32 bit Mixed Precision Multiply into 32-bit (DALU)

General Description

Performs unsigned 16-bit by signed 32-bit integer multiply into a 32-bit result.

Flag Options

Flag	Position	Description
US	flg1	Unsigned By Signed
I	flg2	Integer arithmetic
L	flg2	32-bit operation

Instruction Variants

#	Syntax	Description
1	**MPYM.US.IL.L Da.l,Db,Dn**	**Integer unsigned 16b by signed 32b multiply**
	`Dn = {0x00, ((U48)Da.L * (S48)Db.M)[31:0]}`	

Explicit Operands

Operand	Permitted Values
Da	`D0-D63`
Db	`D0-D63`
Dn	`D0-D63`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPYM.nX 16x32 bit Mixed Precision Multiply into 40-bit (DALU)

General Description

Performs 16x32 bit multiply into 40-bit results.

Flag Options

Flag	Position	Description
I	flg2	Integer arithmetic
R	flg2	Round multiplication result
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MPYM.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 fractional multiply**

	`Dm = (S40)(((Da.H * Dc.M) << 1) >> 16)`
	`Dn = (S40)(((Da.L * Dd.M) << 1) >> 16)`	

2	**MPYM.X Da.h,Db,Dn**	**Fractional multiply, using high part**
	`Dn = (S40)((((Da.H * Db.M)) << 1) >> 16)`	
3	**MPYM.IR.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 integer multiply with rounding**

	`Dm = (S40)(((Da.H * Dc.M) + 0x8000) >> 16)`
	`Dn = (S40)(((Da.L * Dd.M) + 0x8000) >> 16)`	

4	**MPYM.IR.2X Da.h,Dc:Dd,Dm:Dn**	**SIMD2 by one value - integer multiply with rounding, using high part**

	`Dm = (((S48)((S48)Da.H * (S48)Dc.M) + 0x8000) >> 16)[39:0]`
	`Dn = (((S48)((S48)Da.H * (S48)Dd.M) + 0x8000) >> 16)[39:0]`	

5	**MPYM.IR.2X Da.l,Dc:Dd,Dm:Dn**	**SIMD2 by one value - integer multiply with rounding, using low part**

	`Dm = (((S48)((S48)Da.L * (S48)Dc.M) + 0x8000) >> 16)[39:0]`
	`Dn = (((S48)((S48)Da.L * (S48)Dd.M) + 0x8000) >> 16)[39:0]`	

6	**MPYM.IR.X Da.h,Db,Dn**	**Integer multiply with rounding, using high part**
	`Dn = (S40)(((Da.H * Db.M) + 0x8000) >> 16)`	
7	**MPYM.IR.X Da.l,Db,Dn**	**Integer multiply with rounding, using low part**
	`Dn = (S40)(((Da.L * Db.M) + 0x8000) >> 16)`	

Table continues on the next page...

#	Syntax	Description
8	**MPYM.R.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 fractional multiply with rounding**

	`Dm = (S40)((((Da.H * Dc.M) << 1) + 0x8000) >> 16)`
	`Dn = (S40)((((Da.L * Dd.M) << 1) + 0x8000) >> 16)`	

9	**MPYM.R.2X Da.h,Dc:Dd,Dm:Dn**	**SIMD2 by one value - fractional multiply with rounding, using high part**

	`Dm = ((((Da.H * Dc.M) << 1) + 0x8000) >> 16)[39:0]`
	`Dn = ((((Da.H * Dd.M) << 1) + 0x8000) >> 16)[39:0]`	

10	**MPYM.R.2X Da.l,Dc:Dd,Dm:Dn**	**SIMD2 by one value - fractional multiply with rounding, using low part**

	`Dm = (((((Da.L * Dc.M)) << 1) + 0x8000) >> 16)[39:0]`
	`Dn = (((((Da.L * Dd.M)) << 1) + 0x8000) >> 16)[39:0]`	

11	**MPYM.R.X Da.h,Db,Dn**	**Fractional multiply with rounding, using high part**
	`Dn = (((((S64)Da.H * (S64)Db.M) << 1) + 0x8000) >> 16)[39:0]`	
12	**MPYM.R.X Da.l,Db,Dn**	**Fractional multiply with rounding, using low part**
	`Dn = (((((S64)Da.L * (S64)Db.M) << 1) + 0x8000) >> 16)[39:0]`	
13	**MPYM.SR.2X Da,Dc:Dd,Dm:Dn**	**SIMD2 fractional multiply with rounding and saturation**

	`Dm = SAT32(((((Da.H * Dc.M) << 1) + 0x8000) >> 16))`
	`Dn = SAT32(((((Da.L * Dd.M) << 1) + 0x8000) >> 16))`	

14	**MPYM.SR.2X Da.h,Dc:Dd,Dm:Dn**	**SIMD2 by one value - fractional multiply with rounding and saturation, using high part**

	`Dm = SAT32((((((Da.H * Dc.M)) << 1) + 0x8000) >> 16))`
	`Dn = SAT32((((((Da.H * Dd.M)) << 1) + 0x8000) >> 16))`	

15	**MPYM.SR.2X Da.l,Dc:Dd,Dm:Dn**	**SIMD2 by one value - fractional multiply with rounding and saturation, using low part**

	`Dm = SAT32((((((Da.L * Dc.M)) << 1) + 0x8000) >> 16))`
	`Dn = SAT32((((((Da.L * Dd.M)) << 1) + 0x8000) >> 16))`	

16	**MPYM.SR.X Da.h,Db,Dn**	**Fractional multiply with rounding and saturation, using high part**
	`Dn = SAT32((((Da.H * Db.M) << 1) + 0x8000)[56:16])`	
17	**MPYM.SR.X Da.l,Db,Dn**	**Fractional multiply with rounding and saturation, using low part**
	`Dn = SAT32((((Da.L * Db.M) << 1) + 0x8000)[56:16])`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
SR.SAT	13, 14, 15, 16, 17

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 3, 4, 5, 6, 7
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPYQ.nX 16x16 bit Quad Dot Product into 40-bit Result (DALU)

General Description

Performs 16x16 bit quad dot product into 40-bit results.

Flag Options

Flag	Position	Description
I	flg2	Integer arithmetic

Instruction Variants

#	Syntax	Description
1	**MPYQ.X Da:Db,Dc:Dd,Dn**	**Fractional quad multiply**
	`Dn = (((Da.H * Dc.H) + (Da.L * Dc.L) + (Db.H * Dd.H) + (Db.L * Dd.L)) << 1)[39:0]`	
2	**MPYQ.I.X Da:Db,Dc:Dd,Dn**	**Integer quad multiply**
	`Dn = ((Da.H * Dc.H) + (Da.L * Dc.L) + (Db.H * Dd.H) + (Db.L * Dd.L))[39:0]`	

Explicit Operands

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPYRC.nT 16x16 bit Real by Complex Multiply into 20-bit Result (DALU)

General Description

Performs SIMD4 16x16 Real by Complex multiply into 20-bit results

Flag Options

Flag	Position	Description
I	flg2	Integer arithmetic
R	flg2	Round multiplication result

Instruction Variants

#	Syntax	Description
1	**MPYRC.IR.4T Da,Dc:Dd,Dm:Dn**	**SIMD4 integer real by complex multiply with rounding**

	`Dm.WH = (S20)(((Da.H * Dc.H) + 0x8000) >> 16)[15:0]`
	`Dm.WL = (S20)(((Da.H * Dc.L) + 0x8000) >> 16)[15:0]`
	`Dn.WH = (S20)(((Da.L * Dd.H) + 0x8000) >> 16)[15:0]`
	`Dn.WL = (S20)(((Da.L * Dd.L) + 0x8000) >> 16)[15:0]`	

2	**MPYRC.R.4T Da,Dc:Dd,Dm:Dn**	**SIMD4 fractional real by complex multiply with rounding**

	`Dm.WH = ((((Da.H * Dc.H) << 1) + 0x8000) >> 16)[19:0]`
	`Dm.WL = ((((Da.H * Dc.L) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WH = ((((Da.L * Dd.H) << 1) + 0x8000) >> 16)[19:0]`
	`Dn.WL = ((((Da.L * Dd.L) << 1) + 0x8000) >> 16)[19:0]`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPYRC.nW 16x16 bit Real by Complex Multiply into 16-bit Result (DALU)

General Description

Performs SIMD4 16x16 Real by Complex multiply into 16-bit results, with rounding and saturation.

Flag Options

Flag	Position	Description
R	flg2	Round multiplication result
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MPYRC.SR.4W Da,Dc:Dd,Dm:Dn**	**SIMD4 fractional real by complex multiply with rounding and saturation**

```
Dm.H = SAT16 ((((Da.H * Dc.H) << 1) + 0x8000) >> 16))[15:0]
Dm.L = SAT16 ((((Da.H * Dc.L) << 1) + 0x8000) >> 16))[15:0]
Dm.E = 0
Dn.H = SAT16 ((((Da.L * Dd.H) << 1) + 0x8000) >> 16))[15:0]
Dn.L = SAT16 ((((Da.L * Dd.L) << 1) + 0x8000) >> 16))[15:0]
Dn.E = 0
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MPYSHL.nX 16x16 bit Multiply with Shift Left (DALU)

General Description

Performs fractional 16x16 multiply with shift left and saturate.

Flag Options

Flag	Position	Description
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**MPYSHL.S.X Da.h,Db.h,Dn**	**Fractional multiply with shifted left and saturation**
	`Dn = SAT32(((Da.H * Db.H) << 2))`	

Explicit Operands

Operand	Permitted Values
Da	`D0-D63`
Db	`D0-D63`
Dn	`D0-D63`

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_MPY	All

MSGSND Send a Message (LSU)

General Description

Issue a CoreNet message using the data in Ra. A CoreNet message is a special access that uses the address bus to communicate with other units that are connected to CoreNet and are designed to respond to such messages. The value in Ra is not considered an address hence is not translated or monitored by the MMU. For more information on CoreNet messages, see the Address Generation chapter. Some restrictions apply to this instruction, see rule A. 9.

Instruction Variants

#	Syntax	Operation	Description
1	MSGSND Ra		Issue a CoreNet message on the data address bus

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Pipeline behavior	agu_MOVE_ST_nodat	All

NAND.nX Bitwise AND-NOT (40-bit) (DALU)

General Description

Perform a bitwise AND-NOT operation on all the bits of a D register

Instruction Variants

#	Syntax	Operation	Description

1	NAND.2X Da:Db,Dc:Dd,Dm:Dn	`Dm = ~ (Da & Dc)`
		`Dn = ~ (Db & Dd)`
			SIMD2 AND-NOT operation of two pairs of D registers into two destination registers, respectively

2	NAND.X Da,Db,Dn	`Dn = ~ (Db & Da)`	AND-NOT the bits of two D registers

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

NANDA Bitwise AND-NOT (32-bit) (LSU,IPU)

General Description

Perform a bitwise exclusive OR operation on R registers

Instruction Variants

#	Syntax	Operation	Description
1	NANDA Ra,Rb,Rn	`Rn = ~ (Ra & Rb)`	AND-NOT the bits of two R registers

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`
Rb	`R0-R31`
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits	All
Execution unit	IPU	All
	LSU	All
Pipeline behavior	agu_AAU_LOGIC	All

NEG.L Negate Value of 32-bit Number (DALU)

General Description

Negates (2's complement) the signed 32-bit value (integer or fraction), and saturates the result. The source extension is ignored. The destination extension is cleared.

Note that negate of 0x8000_0000 results in 0x7FFF_FFFF due to saturation.

Flag Options

Flag	Position	Description
S	flg2	Saturated result

Instruction Variants

#	Syntax	Operation	Description

1	NEG.S.L Da,Dn	`Dn = SAT32((-(Da.M)))`
		`Dn.E = 0`
			32-bit Negate with saturation

Explicit Operands

Operand	Permitted Values
Da	`D0-D63`
Dn	`D0-D63`

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

NEG.nT Negate Value of 20-bit Number (DALU)

General Description

Negates (2's complement) the value of two or four packed signed 20-bit values (integer or fraction).

Negation of 0x8_0000 results in 0x8_0000.

Instruction Variants

#	Syntax	Operation	Description

1	NEG.2T Da,Dn	`Dn.WH = -Da.WH`
		`Dn.WL = -Da.WL`
			SIMD2 20-bit negate value (two values packed in one register)

2	NEG.4T Da:Db,Dm:Dn	`Dm.WH = ( -Da.WH )`
		`Dm.WL = ( -Da.WL )`
		`Dn.WH = ( -Db.WH )`
		`Dn.WL = ( -Db.WL )`
			SIMD4 20-bit negate value (four values packed in two register)

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

NEG.nW Negate Value of 16-bit Number (DALU)

General Description

Negates (2's complement) the value of two or four packed signed 16-bit values (integer or fraction), and saturates the result. The source extension is ignored. The destination extension is cleared.

Negation of 0x8000 results in 0x7FFF due to saturation.

Flag Options

Flag	Position	Description
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**NEG.S.2W Da,Dn**	**SIMD2 16-bit negate value (two values packed in one register)**

	`Dn.H = SAT16 ((-Da.H))[15:0]`
	`Dn.L = SAT16 ((-Da.L))[15:0]`
	`Dn.E = 0`	

2	**NEG.S.4W Da:Db,Dm:Dn**	**SIMD4 16-bit negate value (four values packed in two register)**

	`Dm.H = SAT16 ((-Da.H))[15:0]`
	`Dm.L = SAT16 ((-Da.L))[15:0]`
	`Dm.E = 0`
	`Dn.H = SAT16 ((-Db.H))[15:0]`
	`Dn.L = SAT16 ((-Db.L))[15:0]`
	`Dn.E = 0`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

NEG.nX Negate Value of 40-bit Number (DALU)

General Description

Negates (2's complement) the value of one or two 40-bit values, with optional saturation to 32 bits. Saturation is either explicit in the mnemonic (variants with the S flag) or implicit accoding to SR.SM (the legacy LEG variant).

Negation of 0x80_0000_0000 with no saturation gives 0x80_0000_0000.
Saturation of any value between 0x01_0000_0000 to 0x7F_FFFF_FFFF gives 0x 00_7FFF_FFFF, and saturation of 0x80_0000_0000 gives 0x 00_7FFFF_FFFF.

Flag Options

Flag	Position	Description
LEG	flg2	Legacy instruction
S	flg2	Saturated result

Instruction Variants

#	Syntax	Operation	Description

1	NEG.2X Da:Db,Dm:Dn	`Dm = (-Da)[39:0]`
		`Dn = (-Db)[39:0]`
			SIMD2 40-bit negate value

2	NEG.X Da,Dn	`Dn = (- Da)[39:0]`	40-bit negate value
3	NEG.LEG.X Da,Dn	`Dn = srSAT32((-Da))`	Legacy 40-bit negate value - Saturate to 32-bit only if SR.SM is set

4	NEG.S.2X Da:Db,Dm:Dn	`Dm = SAT32((-Da))`
		`Dn = SAT32((-Db))`
			SIMD2 40-bit negate value with saturation 32 bits

5	NEG.S.X Da,Dn	`Dn = SAT32((-Da))`	40-bit negate value with saturation to 32 bits

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Affect Instructions

Field	Relevant Variants	Comments
SR.SM	3	

Affected By Instructions

Field	Relevant Variants
SR.SAT	3, 4, 5

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

NEGA Negate Value of 32-bit Number (LSU,IPU)

General Description

Negates (2's complement) the signed 32-bit value (integer or fraction).

Negation of 0x8000_0000 results in 0x8000_0000.

Instruction Variants

#	Syntax	Operation	Description
1	NEGA Ra,Rn	`Rn = (-(S33)Ra)[31:0]`	32-bit negate value

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Execution unit	IPU	All
	LSU	All
Pipeline behavior	agu_AAU	All

NEOR.nX Bitwise Exclusive OR and NOT (40-bit) (DALU)

General Description

Perform a bitwise Exclusive OR and NOT operation on all the bits of a D register

Instruction Variants

#	Syntax	Operation	Description

1	NEOR.2X Da:Db,Dc:Dd,Dm:Dn	`Dm = ~ (Da ^ Dc)`
		`Dn = ~ (Db ^ Dd)`
			SIMD2 exclusive OR and NOT operation of two pairs of D registers into two destination registers, respectively

2	NEOR.X Da,Db,Dn	`Dn = ~ (Db ^ Da)`	Exclusive OR and NOT of the bits of two D registers

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

NEORA Bitwise Exclusive OR and NOT (32 bits) (LSU,IPU)

General Description

Perform a bitwise exclusive OR and NOT operation on R registers

Instruction Variants

#	Syntax	Operation	Description
1	NEORA Ra,Rb,Rn	`Rn = ~ (Ra ^ Rb)`	Exclusive OR and NOT of the bits of two R registers

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`
Rb	`R0-R31`
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits	All
Execution unit	IPU	All
	LSU	All
Pipeline behavior	agu_AAU_LOGIC	All

NOP No Operation (NO_BLK,LSU,IPU, DALU)

General Description

This instruction does not change any architectural state. It can appear on its own or grouped with other instructions. It can be used as a program space holder, for example if there is a need to align the next VLES to a certain address. When not grouped with functional instructions, it can be used for inserting a VLES delay in the execution sequence. Variants include the standard NOP which is not dispatched to any execution unit and hence not subject to execution unit capacity issues. There are also NOP variants that are dispatched to the AGU or DALU, occupying an execution slot. These are mainly used for testing.

Flag Options

Flag	Position	Description
AGU	flg1	AGU unit
DALU	flg1	DALU unit

Instruction Variants

#	Syntax	Operation	Description
1	NOP		No Operation - not dispatched to any execution unit
2	NOP.AGU		No Operation - dispatched to the AGU
3	NOP.DALU		No Operation - dispatched to the DALU

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	16-bits	1
	32-bits	2, 3
Execution unit		1
	DALU	3, 3
	IPU	2
	LSU	2
No predication		1
Pipeline behavior	agu_AAU	2
	dalu_DAU	3

NOR.nX Bitwise OR (40-bit) (DALU)

General Description

Perform a bitwise OR-NOT operation on all the bits of a D register

Instruction Variants

#	Syntax	Operation	Description

1	NOR.2X Da:Db,Dc:Dd,Dm:Dn	`Dm = ~ (Da │ Dc)`
		`Dn = ~ (Db │ Dd)`
			SIMD2 OR-NOT operation of two pairs of D registers into two destination registers, respectively

2	NOR.X Da,Db,Dn	`Dn = ~ (Db │ Da)`	OR-NOT the bits of two D registers

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

NORA Bitwise OR (32-bit) (LSU,IPU)

General Description

Perform a bitwise OR operation on R registers

Instruction Variants

#	Syntax	Operation	Description
1	NORA Ra,Rb,Rn	`Rn = ~ (Ra | Rb)`	AND-NOT the bits of two R registers

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`
Rb	`R0-R31`
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits	All
Execution unit	IPU	All
	LSU	All
Pipeline behavior	agu_AAU_LOGIC	All

NOT.nX Bitwise NOT (40-bit) (DALU)

General Description

Perform a bitwise NOT operation on all the bits of a D register

Instruction Variants

#	Syntax	Operation	Description

1	NOT.2X Da:Db,Dm:Dn	`Dm = ~ Da`
		`Dn = ~ Db`
			SIMD2 NOT (invert) operation of two pairs of D registers into two destination registers, respectively

2	NOT.X Da,Dn	`Dn = ~ Da`	NOT (invert) the bits of two D registers

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

NOTA Bitwise NOT (32-bit) (LSU,IPU)

General Description

Perform a bitwise NOT operation on R registers

Instruction Variants

#	Syntax	Operation	Description
1	NOTA Ra,Rn	`Rn = ~ Ra`	NOT (invert) the bits of two R registers

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Execution unit	IPU	All
	LSU	All
Pipeline behavior	agu_AAU_LOGIC	All

NOTIFY Send a Notify Transaction (LSU)

General Description

This instruction is a constituent of the NOTIFYM meta-instruction. As such, it should not be used in this form but only through NOTIFYM. The instruction issues a NOTIFY Corenet transaction on the data bus. For more information on NOTIFY transactions, see the description of NOTIFYM and also in the Address Generation chapter.

Instruction Variants

#	Syntax	Operation	Description
1	NOTIFY (Rn)		Send a CoreNet Notify access to the address in Rn

Explicit Operands

Operand	Permitted Values
Rn	R0-R31

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Helper instruction		All
No predication		All
Pipeline behavior	agu_MOVE_ST_nodat	All

NOTIFYM Corenet Notify Transaction (META INST)

General Description

NOTIFYM is a meta-instruction activating a NOTIFY CoreNet transaction, which could be seen as a decorated store access that does not send data. The information carried by this transaction is only the fact that it was sent to a certain destination, and the decoration attributes. The address of the access is taken from Rn, and the decoration attributes from the immediate operand. Some restrictions apply to this instruction, see rule A.9. For more information on NOTIFYM accesses, see the Address Generation chapter.

Instruction Variants

#	Syntax	Description
1	**NOTIFYM #u4,(Rn)**	**Send a CoreNet Notify access to the address in Rn**
	`Encoded as: DECOR #u4 NOTIFY (Rn)`	

Explicit Operands

Operand	Permitted Values
Rn	R0-R31
u4	$0 \leq u4 < 2^4$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	64-bits	All
Execution unit	LSU+IPU	All
No predication		All
Pipeline behavior	per constructing instructions	All

OR.L Bitwise OR (32-bit) (DALU)

General Description

Perform a bitwise OR operation on the lower 32-bits of a D register

Instruction Variants

#	Syntax	Operation	Description
1	OR.L #u32,Da,Dn	`Dn = {Da.E, ((Da.M)	u32)}`	OR the bits of a D register with an unsigned 32-bit immediate. The extension of the destination is not changed

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Dn	D0-D63
u32	$0 \leq u32 < 2^{32}$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	64-bits	All
Pipeline behavior	dalu_LBM	All

OR.nX Bitwise OR (40-bit) (DALU)

General Description

Perform a bitwise OR operation on all the bits of a D register

Instruction Variants

#	Syntax	Operation	Description

1	OR.2X Da:Db,Dc:Dd,Dm:Dn	Dm = (Da | Dc)
		Dn = (Db | Dd)
			SIMD2 OR operation of two pairs of D registers into two destination registers, respectively

2	OR.X Da,Db,Dn	Dn = (Da | Db)	OR the bits of two D registers

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

ORA Bitwise OR (32-bit) (LSU,IPU)

General Description

Performs a bitwise OR operation on R and control registers. Variants include 32-bit bitwise operation on values in two R registers, and 16-bit operation between an unsigend immediate value and a 16-bit register portion (wither high or low). The register is either an R regisetr or a control register. The other portion of the register is not changed.

Instruction Variants

#	Syntax	Description
1	**ORA Ra,Rb,Rn**	**OR the bits of two R registers**
	`Rn = Ra	Rb`	
2	**ORA #u16,C4.H**	**Bit-wise OR of the higher 16-bit control register portion with an unsigned immediate.**
	`C4.H = C4.H	u16` `Alias, encoded as: BMSETA #u16 C4.H`	
3	**ORA #u16,C4.L**	**Bit-wise OR of the lower 16-bit control register portion with an unsigned immediate.**
	`C4.L = C4.L	u16` `Alias, encoded as: BMSETA #u16 C4.L`	
4	**ORA #u16,Rn.H**	**Bit-wise OR of the higher 16-bit R register portion with an unsigned immediate.**
	`Rn.H = Rn.H	u16` `Alias, encoded as: BMSETA #u16 Rn.H`	
5	**ORA #u16,Rn.L**	**Bit-wise OR of the lower 16-bit R register portion with an unsigned immediate.**
	`Rn.L = Rn.L	u16` `Alias, encoded as: BMSETA #u16 Rn.L`	

Explicit Operands

Operand	Permitted Values				
C4	EIDR, SR2,	GCR, TMTAG	MCTL,	MOCR,	SR,
Ra	R0-R31				
Rb	R0-R31				
Rn	R0-R31				
u16	$0 \leq u16 < 2^{16}$				

Instruction Attributes

Attribute	Value	Relevant Variant #
Alias		2, 3, 4, 5
Encoding length	32-bits	1
	48-bits	2, 3, 4, 5
Execution unit	IPU	All
	LSU	1
Pipeline behavior	agu_AAU_CTRL_REG	2, 3
	agu_AAU_LOGIC	1, 4, 5

PACK.nB Pack Four/Eight Bytes (DALU)

General Description

Packs four/eight bytes from two/four registers into one/two register/s. Each byte is extracted from either the high byte or the low byte of a 16-bit word of the input register. All bytes extracted into the low 32 bits of the result. The extension of the source is ignored while the extension of the result is cleared.

Note that in BH and BL variants of eight byte unpack, source registers are taken in order Da.H, Da.L, Dc.H, Dc.L,Db.H, Db.L, Dd.H, Dd.L.

Flag Options

Flag	Position	Description
B	flg1	Byte
BH	flg1	Byte High
BL	flg1	Byte Low
H	flg1	High 16-bit portion
HL	flg1	High 16 bits from the first argument and Low 16 bits from the second argument
L	flg1	32 bit input
LH	flg1	Low 16 bits from the first argument and High 16 bits from the second argument

Instruction Variants

#	Syntax	Description
1	**PACK.B.4B Da,Db,Dn**	**Pack 4 low bytes of Da.H,Da.L,Db.H,Db.L into one register**
	`Dn = {0x00, Da.HL, Da.LL, Db.HL, Db.LL}`	
2	**PACK.BH.8B Da:Db,Dc:Dd,Dm:Dn**	**Pack 8 high bytes of Da.H, Da.L, Dc.H, Dc.L,Db.H, Db.L, Dd.H, Dd.L into two registers**

	`Dm.HH = Da.HH`
	`Dm.HL = Da.LH`
	`Dm.LH = Dc.HH`
	`Dm.LL = Dc.LH`
	`Dn.HH = Db.HH`
	`Dn.HL = Db.LH`
	`Dn.LH = Dd.HH`
	`Dn.LL = Dd.LH`
	`Dm.E = 0`
	`Dn.E = 0`	

3	**PACK.BL.8B Da:Db,Dc:Dd,Dm:Dn**	**Pack 8 low bytes of Da.H, Da.L, Dc.H, Dc.L,Db.H, Db.L, Dd.H, Dd.L in two registers**

Table continues on the next page...

#	Syntax	Description

	`Dm.HH = Da.HL`
	`Dm.HL = Da.LL`
	`Dm.LH = Dc.HL`
	`Dm.LL = Dc.LL`
	`Dn.HH = Db.HL`
	`Dn.HL = Db.LL`
	`Dn.LH = Dd.HL`
	`Dn.LL = Dd.LL`
	`Dm.E = 0`
	`Dn.E = 0`	

4	**PACK.H.4B Da,Db,Dn**	**Pack 4 high bytes of Da.H,Da.L,Db.H,Db.L into one register**

	`Dn.HH = Da.HH`
	`Dn.HL = Da.LH`
	`Dn.LH = Db.HH`
	`Dn.LL = Db.LH`
	`Dn.E = 0`	

5	**PACK.H.8B Da:Db,Dc:Dd,Dm:Dn**	**Pack 4 high bytes of Da.H,Db.H,Dc.H,Dd.H into one register and 4 low bytes of Da.H,Db.H,Dc.H,Dd.H into second register**

	`Dm.HH = Da.HH`
	`Dm.HL = Db.HH`
	`Dm.LH = Dc.HH`
	`Dm.LL = Dd.HH`
	`Dn.HH = Da.HL`
	`Dn.HL = Db.HL`
	`Dn.LH = Dc.HL`
	`Dn.LL = Dd.HL`
	`Dm.E = 0`
	`Dn.E = 0`	

6	**PACK.HL.8B Da,Db,Dm:Dn**	**Pack 4 high bytes of Da.H,Da.L,Db.H,Db.L into one register and 4 low bytes of the same registers into second register**

	`Dm.HH = Da.HH`
	`Dm.HL = Da.LH`
	`Dm.LH = Db.HH`
	`Dm.LL = Db.LH`
	`Dn.HH = Da.HL`
	`Dn.HL = Da.LL`
	`Dn.LH = Db.HL`
	`Dn.LL = Db.LL`
	`Dm.E = 0`
	`Dn.E = 0`	

7	**PACK.L.4B Da,Db,Dn**	**Pack 4 low bytes of Da.H,Da.L,Db.H,Db.L into one register**

	`Dn.HH = Da.HL`
	`Dn.HL = Da.LL`
	`Dn.LH = Db.HL`
	`Dn.LL = Db.LL`
	`Dn.E = 0`	

Table continues on the next page...

#	Syntax	Description
8	**PACK.L.8B Da:Db,Dc:Dd,Dm:Dn**	**Pack 4 high bytes of Da.L,Db.l,Dc.L,Dd.L into one register and 4 low bytes of Da.L,Db.L,Dc.L,Dd.L into second register**

	`Dm.HH = Da.LH`
	`Dm.HL = Db.LH`
	`Dm.LH = Dc.LH`
	`Dm.LL = Dd.LH`
	`Dn.HH = Da.LL`
	`Dn.HL = Db.LL`
	`Dn.LH = Dc.LL`
	`Dn.LL = Dd.LL`
	`Dm.E = 0`
	`Dn.E = 0`	

9	**PACK.LH.8B Da,Db,Dm:Dn**	**Pack 4 low bytes of Da.H,Da.l,Db.H,Db.L into one register and 4 high bytes of Da.H,Da.l,Db.H,Db.L into second register**

	`Dm.HH = Da.HL`
	`Dm.HL = Da.LL`
	`Dm.LH = Db.HL`
	`Dm.LL = Db.LL`
	`Dn.HH = Da.HH`
	`Dn.HL = Da.LH`
	`Dn.LH = Db.HH`
	`Dn.LL = Db.LH`
	`Dm.E = 0`
	`Dn.E = 0`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

PACK.nF Pack Two/Four 40-bit to 20-bit Fraction Words (DALU)

General Description

Packs two/four 40-bit fraction words from two/four register/s into two/four 20-bit values in one or two registers. Each 40-bit fraction is casted and saturated into 20-bit value. 20-bit_result = Saturate_20_bit (40-bit_value >> 16).

Flag Options

Flag	Position	Description
X	flg1	40 bit input
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**PACK.X.S.2F Da,Db,Dn**	**Pack 2 40-bit fraction words into 2 20-bit fraction words with saturation**

	`Dn.WH = (S20)(SAT32 (Da) >> 16)`
	`Dn.WL = (S20)(SAT32 (Db) >> 16)`	

2	**PACK.X.S.4F Da:Db,Dc:Dd,Dm:Dn**	**Pack 4 40-bit fraction words into 4 20-bit fraction words with saturation**

	`Dm.WH = (S20)(SAT32 (Da) >> 16)`
	`Dm.WL = (S20)(SAT32 (Db) >> 16)`
	`Dn.WH = (S20)(SAT32 (Dc) >> 16)`
	`Dn.WL = (S20)(SAT32 (Dd) >> 16)`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU_Y	All

PACK.nT Pack into 20-bit Fraction Words (DALU)

General Description

Packs 20-bit or 40-bit fraction words into packed dual 20 bits in a data register.

PACK.F.2T extracts two 20-bit portions from two 40-bit values, taking bits [35:16] from each, and packs them into two 20-bit WH and WL portions in the destination register.

Other variants pack two/four 20-bit fraction words from two/four registers into one/two registers, packing either the high 20-bit portion or low 20-bit portion of the input registers according to the variant.

Flag Options

Flag	Position	Description
F	flg1	Fractional input
T	flg1	20 bits
pppp	flg1	Represents specific H/L portion combinations - see operand table below

Instruction Variants

#	Syntax	Operation	Description

1	PACK.F.2T Da,Db,Dn	`Dn.WL = Da[35:16]`
		`Dn.WH = Db[35:16]`
			Pack 2 40-bit fraction words into 2 20-bit fraction words

2	PACK.T.2T Da.h,Db.h,Dn	`Dn.WH = Da.WH`
		`Dn.WL = Db.WH`
			Pack Da.WH and Db.WH into Dn

3	PACK.T.2T Da.h,Db.l,Dn	`Dn.WH = Da.WH`
		`Dn.WL = Db.WL`
			Pack Da.WH and Db.WL into Dn

4	PACK.T.2T Da.l,Db.h,Dn	`Dn.WH = Da.WL`
		`Dn.WL = Db.WH`
			Pack Da.WL and Db.WH into Dn

5	PACK.T.2T Da.l,Db.l,Dn	`Dn.WH = Da.WL`
		`Dn.WL = Db.WL`
			Pack Da.WL and Db.WL into Dn

6	PACK.pppp.4T Da:Db,Dc:Dd,Dm:Dn	`Dm.WH=Da.Wp`
		`Dm.WL=Db.Wp`
		`Dn.WH=Dc.Wp`
		`Dn.WL=Dd.Wp`
		`where p = H/L`
			Pack Da.Wp and Db.Wp into Dm; Pack Dc.Wp and Dd.Wp into Dn. Each p in the mnemonic stands for either H or L

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Table continues on the next page...

Operand	Permitted Values
Dn	D0-D63
pppp	LLLL, LLLH, HLHL, HLHH, HHLL, HHLH, HHHL, HHHH, LLHL, LLHH, LHLL, LHLH, LHHL, LHHH, HLLL, HLLH

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

PACK.nW Pack into 16-bit Words (DALU)

General Description

This instruction takes two or more 16-bit words and pack them in pairs into a data register or registers.

The PACK.W.2W variant packs two 16-bit words from the H or L portion of two registers into a single register. Note that since each word is sign extended into 20 bits in the WL and WH portions, this instruction can also cast two 16-bit words from high or low part of two registers into two 20-bit values and pack them into a single register.

PACK.INSn.p3.4W packs a single 16-bit word from the first source register with either 3 high 16-bit words of the second source registers (H3 variants take Dc.H, Dc.L and Dd.H while L3 variants take Dc.L, Dd.H and Dd.L). The first source word is inserted before the first of the three words for INS0 variants, inserted before the second of the three words for INS1 variants, inserted before the third of the three words for INS2 variants and inserted after the third of the three words for INS3 variants. First word can be taken from either the high part or the low part of the first source register. The extension of the result is cleared for these variants.

PACK.L.S.nW takes two/four 32-bit integer words from two/four registers, saturates them into 16-bit values and packs them into two/four 16-bit integer words in one/two registers. For these variants, the extension of the source registers is ignored and the extension of the destinations is cleared.

PACK.X.S.nW takes two/four 40-bit integer words from two/four registers, saturates them into 16-bit values and packs into two/four 16-bit integer words in one/two registers. The extension of the result is cleared for these variants.

Flag Options

Flag	Position	Description
INS0	flg1	Insert before the first word
INS1	flg1	Insert before the second word
INS2	flg1	Insert before the third word
INS3	flg1	Insert after the third word
L	flg1	32 bit input
W	flg1	Word (16 bits)
X	flg1	40 bit input
H3	flg2	Uses the 3 higher 16-bit portions of the source pair
L3	flg2	Uses the 3 lower 16-bit portions of the source pair
S	flg2	Saturated result

Instruction Variants

#	Syntax	Operation	Description

1	PACK.W.2W Da.h,Db.h,Dn	`Dn.WH = (S20)Da.H`
		`Dn.WL = (S20)Db.H`
			Pack Da.H and Db.H into Dn

2	PACK.W.2W Da.h,Db.l,Dn	`Dn.WH = (S20)Da.H`
		`Dn.WL = (S20)Db.L`
			Pack Da.H and Db.L into Dn

Table continues on the next page...

#	Syntax	Operation	Description

3	PACK.W.2W Da.l,Db.h,Dn	`Dn.WH = (S20)Da.L`
		`Dn.WL = (S20)Db.H`
			Pack Da.L and Db.H into Dn

4	PACK.W.2W Da.l,Db.l,Dn	`Dn.WH = (S20)Da.L`
		`Dn.WL = (S20)Db.L`
			Pack Da.L and Db.L into Dn

5	PACK.INS0.H3.4W Da.h,Dc:Dd,Dm:Dn	`Dm.H = Da.H`
		`Dm.L = Dc.H`
		`Dn.H = Dc.L`
		`Dn.L = Dd.H`
		`Dm.E = 0`
		`Dn.E = 0`
			Pack {Da.H, Dc.H, Dc.L, Dd.H} into Dm:Dn

6	PACK.INS0.H3.4W Da.l,Dc:Dd,Dm:Dn	`Dm.H = Da.L`
		`Dm.L = Dc.H`
		`Dn.H = Dc.L`
		`Dn.L = Dd.H`
		`Dm.E = 0`
		`Dn.E = 0`
			Pack {Da.L, Dc.H, Dc.L, Dd.H} into Dm:Dn

7	PACK.INS1.H3.4W Da.h,Dc:Dd,Dm:Dn	`Dm.H = Dc.H`
		`Dm.L = Da.H`
		`Dn.H = Dc.L`
		`Dn.L = Dd.H`
		`Dm.E = 0`
		`Dn.E = 0`
			Pack {Dc.L, Da.H, Dd.H, Dd.L} into Dm:Dn

8	PACK.INS1.H3.4W Da.l,Dc:Dd,Dm:Dn	`Dm.H = Dc.H`
		`Dm.L = Da.L`
		`Dn.H = Dc.L`
		`Dn.L = Dd.H`
		`Dm.E = 0`
		`Dn.E = 0`
			Pack {Dc.L, Da.L, Dd.H, Dd.L} into Dm:Dn

9	PACK.INS2.H3.4W Da.h,Dc:Dd,Dm:Dn	`Dm.H = Dc.H`
		`Dm.L = Dc.L`
		`Dn.H = Da.H`
		`Dn.L = Dd.H`
		`Dm.E = 0`
		`Dn.E = 0`
			Pack {Dc.L, Dd.H, Da.H, Dd.L} into Dm:Dn

10	PACK.INS2.H3.4W Da.l,Dc:Dd,Dm:Dn	`Dm.H = Dc.H`
		`Dm.L = Dc.L`
		`Dn.H = Da.L`
		`Dn.L = Dd.H`
		`Dm.E = 0`
		`Dn.E = 0`
			Pack {Dc.L, Dd.H, Da.L, Dd.L} into Dm:Dn

11	PACK.INS3.H3.4W Da.h,Dc:Dd,Dm:Dn	`Dm.H = Dc.H`
		`Dm.L = Dc.L`
		`Dn.H = Dd.H`
		`Dn.L = Da.H`
		`Dm.E = 0`
		`Dn.E = 0`
			Pack {Dc.L, Dd.H, Dd.L, Da.H} into Dm:Dn

12	PACK.INS3.H3.4W Da.l,Dc:Dd,Dm:Dn	`Dm.H = Dc.H`
		`Dm.L = Dc.L`
		`Dn.H = Dd.H`
		`Dn.L = Da.L`
		`Dm.E = 0`
		`Dn.E = 0`
			Pack {Dc.L, Dd.H, Dd.L, Da.L} into Dm:Dn

13	PACK.INS0.L3.4W Da.h,Dc:Dd,Dm:Dn	`Dm.H = Da.H`
		`Dm.L = Dc.L`
		`Dn.H = Dd.H`
		`Dn.L = Dd.L`
		`Dm.E = 0`
		`Dn.E = 0`
			Pack {Da.H, Dc.L, Dd.H, Dd.L} into Dm:Dn

Table continues on the next page...

#	Syntax	Operation	Description

14	PACK.INS0.L3.4W Da.l,Dc:Dd,Dm:Dn	`Dm.H = Da.L`
		`Dm.L = Dc.L`
		`Dn.H = Dd.H`
		`Dn.L = Dd.L`
		`Dm.E = 0`
		`Dn.E = 0`
			Pack {Da.L, Dc.L, Dd.H, Dd.L} into Dm:Dn

15	PACK.INS3.L3.4W Da.h,Dc:Dd,Dm:Dn	`Dm.H = Dc.L`
		`Dm.L = Dd.H`
		`Dn.H = Dd.L`
		`Dn.L = Da.H`
		`Dm.E = 0`
		`Dn.E = 0`
			Pack {Dc.L, Dd.H, Dd.L, Da.H} into Dm:Dn

16	PACK.INS3.L3.4W Da.l,Dc:Dd,Dm:Dn	`Dm.H = Dc.L`
		`Dm.L = Dd.H`
		`Dn.H = Dd.L`
		`Dn.L = Da.L`
		`Dm.E = 0`
		`Dn.E = 0`
			Pack {Dc.L, Dd.H, Dd.L, Da.L} into Dm:Dn

17	PACK.L.S.2W Da,Db,Dn	`Dn.WH = (S20)SAT16 (Da.M)`
		`Dn.WL = (S20)SAT16 (Db.M)`
			Pack 2 32-bit integer words into 2 16-bit integer words with saturation

18	PACK.L.S.4W Da:Db,Dc:Dd,Dm:Dn	`Dm.WH = (S20)SAT16 (Da.M)`
		`Dm.WL = (S20)SAT16 (Db.M)`
		`Dn.WH = (S20)SAT16 (Dc.M)`
		`Dn.WL = (S20)SAT16 (Dd.M)`
			Pack 4 32-bit integer words into 4 16-bit integer words with saturation

19	PACK.X.S.2W Da,Db,Dn	`Dn.WH = (S20)SAT16 (Da)`
		`Dn.WL = (S20)SAT16 (Db)`
			Pack 2 40-bit integer words into 2 16-bit integer words with saturation

20	PACK.X.S.4W Da:Db,Dc:Dd,Dm:Dn	`Dm.WH = (S20)SAT16 (Da)`
		`Dm.WL = (S20)SAT16 (Db)`
		`Dn.WH = (S20)SAT16 (Dc)`
		`Dn.WL = (S20)SAT16 (Dd)`
			Pack 4 40-bit integer words into 4 16-bit integer words with saturation

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
SR.SAT	17, 18, 19, 20

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
	dalu_DAU_Y	17, 18, 19, 20

PCALC Calculate a Predicate as a Function of Predicate Inputs (DALU)

General Description

Calculate a predicate value as a logic function of up to 3 predicate inputs (Pa, Pb, Pc). The logic function is specified as a truth table of 8 bits, specified as a 3-bit (eight entry) truth table u8, as follows:

Pc	Pb	Pa	Pn
0	0	0	u8[0]
0	0	1	u8[1]
0	1	0	u8[2]
0	1	1	u8[3]
1	0	0	u8[4]
1	0	1	u8[5]
1	1	0	u8[6]
1	1	1	u8[7]

All logic functions of 3 predicates can be encoded, however the user may have to reorder the predicates to conform to encoding limitations, and then adjust the function field u8 to match. The following encoding rules apply:

1. In case the function uses 3 different predicates (Pa, Pb, Pc), they should be specified according to the predicate index, the left operand having the smaller index number.
2. In case of a function of two different predicates, the predicate with the larger index should be duplicated (e.g. Pa, Pb, Pb), with the function treating the third operand as a don't care (u8[3:0] should be identical with u8[4:7]).
3. In case of a function of one predicate, the operand shoudl be repeated 3 times (e.g. Pa, Pa, Pa), with the function treating the second and third operands as a don't care (u8[1:0] should be identical with u8[2:3], u8[4:5] and u8[6:7]).

The Pm:Pn variant updates Pm with the outcome of the function, and Pn with the inverse value. Specific grouping limitations apply, see rule G.G.4.

Instruction Variants

#	Syntax	Description
1	**PCALC #u8,Pa,Pb,Pc,Pm:Pn**	**Calculate a binary function into two predicates**
	`Pm = u8[{Pa,Pb,Pc}]; Pn = ~(new Pm)`	
2	**PCALC #u8,Pa,Pb,Pc,Pn**	**Calculate a binary function into a single predicate**
	`Pn = u8[{Pa,Pb,Pc}]`	

Explicit Operands

Operand	Permitted Values
Pa,Pb,Pc	P_{0-5},P_{0-5},P_{0-5}

Table continues on the next page...

Operand	Permitted Values	
Pm:Pn	P_n:P_{n+1}	0 ≤ n ≤ 4
Pn	P0-P5	
u8	$0 \le u8 < 2^8$	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
IF.EC		All
Pipeline behavior	dalu_DAU_Pbit	All

PCALCA Calculate a Predicate as a Function of Predicate Inputs (IPU)

General Description

Calculate a predicate value as a logic function of up to 3 predicate inputs (Pa, Pb, Pc). The logic function is specified as a truth table of 8 bits, specified as a 3-bit (eight entry) truth table u8, as follows:

Pc Pb Pa | Pn
0 0 0 | u8[0]
0 0 1 | u8[1]
0 1 0 | u8[2]
0 1 1 | u8[3]
1 0 0 | u8[4]
1 0 1 | u8[5]
1 1 0 | u8[6]
1 1 1 | u8[7]

All logic functions of 3 predicates can be encoded, however the user may have to reorder the predicates to conform to encoding limitations, and then adjust the function field u8 to match. The following encoding rules apply:

1. In case the function uses 3 different predicates (Pa, Pb, Pc), they should be specified according to the predicate index, the left operand having the smaller index number.
2. In case of a function of two different predicates, the predicate with the larger index should be duplicated (e.g. Pa, Pb, Pb), with the function treating the third operand as a don't care (u8[3:0] should be identical with u8[4:7]).
3. In case of a function of one predicate, the operand shoudl be repeated 3 times (e.g. Pa, Pa, Pa), with the function treating the second and third operands as a don't care (u8[1:0] should be identical with u8[2:3], u8[4:5] and u8[6:7]).

The Pm:Pn variant updates Pm with the outcome of the function, and Pn with the inverse value. Specific grouping limitations apply, see rule G.G.4.

Instruction Variants

#	Syntax	Description
1	**PCALCA #u8,Pa,Pb,Pc,Pm:Pn**	**Calculate a binary function into two predicates**
	`Pm = u8[{Pa,Pb,Pc}]; Pn = ~(new Pm)`	
2	**PCALCA #u8,Pa,Pb,Pc,Pn**	**Calculate a binary function into a single predicate**
	`Pn = u8[{Pa,Pb,Pc}]`	

Explicit Operands

Operand	Permitted Values
Pa,Pb,Pc	P_{0-5},P_{0-5},P_{0-5}

Table continues on the next page...

Operand	Permitted Values	
Pm:Pn	P_n:P_{n+1}	$0 \le n \le 4$
Pn	P0-P5	
u8	$0 \le u8 < 2^8$	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
IF.EC		All
Pipeline behavior	agu_AAU_Pbit	All

PCMB Block Instruction Cache Management Operations (LSU)

General Description

This instruction is a constituent of the PCMM meta-instruction, used in configuring the CME (Cache Management Engine) to invalidate a block of memory from the instruction cache. The PCMB instruction should not be used in this form but only through PCMM. Using the instruction on its own gives indeterminate results. For more information on the PCMM instruction, see its description page and also the Address Generation chapter.

Flag Options

Flag	Position	Description
INVAL	flg1	Invalidate
L1	flg2	Direted to L1 cache
L12	flg2	Directed to L1 and L2 cache

Instruction Variants

#	Syntax	Description
1	**PCMB.INVAL.L1 Ra,Rn**	**Participate in configuring a CME task to invalidate a memory block from the L1 instruction cache.**
	`Rn = success status of the CME block command request`	
2	**PCMB.INVAL.L12 Ra,Rn**	**Participate in configuring a CME task to invalidate a memory block from both the L1 instruction cache and the L2 cache.**
	`Rn = success status of the CME block command request`	

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Helper instruction		All
No predication		All
Pipeline behavior	agu_MOVE_LD_cmd	All

PCMM Block Instruction Cache Coherency Operation (META INST)

General Description

This meta-instruction configures the CME (Cache Management Engine) to invalidate a block of program memory. The meta-instruction is encoded in 3 instructions - a PCMB instruction, a BCCAS instruction, and a SYNC.B instruction that must be grouped together for proper operation. The configuration data sent to the CME is taken from Ra and Rb. The CME returns a success or failure status of the configuration stage that is stored in the destination Rn. The exact format of the data in the input and output registers is described in the Address Generation chapter. Variants allow to invalidate data only from the L1 cache or from both the L1 and L2 caches. Some programming rules apply to these instructions, see rule A.9.

Flag Options

Flag	Position	Description
INVAL	flg1	Invalidate
L1	flg2	Direted to L1 cache
L12	flg2	Directed to L1 and L2 cache

Instruction Variants

#	Syntax	Description
1	**PCMM.INVAL.L1 Ra,Rb,Rn**	**Configure a CME task to invalidate a memory block from the L1 instruction cache**
	`Encoded as: PCMB.INVAL.L1  Rb,Rn  BCCAS Ra  SYNC.B`	
2	**PCMM.INVAL.L12 Ra,Rb,Rn**	**Configure a CME task to invalidate a memory block from both the L1 instruction cache and the L2 cache**
	`Encoded as: PCMB.INVAL.L12  Rb,Rn  BCCAS Ra  SYNC.B`	

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`
Rb	`R0-R31`
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	64-bits	All
Execution unit	2xLSU+PCU	All
No predication		All

Table continues on the next page...

Attribute	Value	Relevant Variant #
Pipeline behavior	per constructing instructions	All

PFETCHB Block Pre-Fetch to the Instruction Cache (LSU)

General Description

This instruction is a constituent of the PFETCHM meta-instruction, used in configuring the CME (Cache Management Engine) to pre-fetch a block of program data to the L2 and L1 caches. The PFETCHB instruction should not be used in this form but only through PFETCHM; using the instruction on its own will give indeterminate results. For more information on the PFETCHM instruction, see its description page and also the Address Generation chapter.

Flag Options

Flag	Position	Description
LCK	flg1	Lock lines in L2
L12	flg2	Directed to L1 and L2 cache
L2	flg2	Directed to L2 cache

Instruction Variants

#	Syntax	Description
1	**PFETCHB.L12 Ra,Rn**	**Participate in configuring a CME task to pre-fetch a program memory block to both the L2 cache and the L1 instruction cache.**
	`Rn = success status of the CME block command request`	
2	**PFETCHB.L2 Ra,Rn**	**Participate in configuring a CME task to pre-fetch a program memory block to the L2 cache.**
	`Rn = success status of the CME block command request`	
3	**PFETCHB.LCK.L2 Ra,Rn**	**Participate in configuring a CME task to pre-fetch a program memory block to the L2 cache, and lock the updated lines.**
	`Rn = success status of the CME block command request`	

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All

Table continues on the next page...

Attribute	Value	Relevant Variant #
Helper instruction		All
No predication		All
Pipeline behavior	agu_MOVE_LD_cmd	All

PFETCHM Block Prefetch to the Instruction Cache (META INST)

General Description

This meta-instruction configures the CME (Cache Management Engine) to pre-fetch a block of instruction data to the L2 cache or also to the L1 instruction cache. The meta-instruction is encoded in 3 instructions - a PFETCHB instruction, a BCCAS instruction, and a SYNC.B instruction that must be grouped together for proper operation. The configuration data sent to the CME is taken from Ra and Rb. The CME returns a success or failure status of the configuration stage that is stored in the destination Rn. The exact format of the data in the input and output registers is described in the Address Generation chapter. Variants allow to fetch the data only to the L2, or also to L1; It is also possible to lock the fetched data in L2. Some programming rules apply to these instructions, see rule A.9. For more information on block cache instructions, refer to the Address Generation chapter.

Flag Options

Flag	Position	Description
LCK	flg1	Lock lines in L2
L12	flg2	Directed to L1 and L2 cache
L2	flg2	Directed to L2 cache

Instruction Variants

#	Syntax	Description
1	**PFETCHM.L12 Ra,Rb,Rn**	**Configure a CME task to pre-fetch a program memory block to both the L2 cache and the L1 instruction cache**
	`Encoded as: PFETCHB.L12 Rb,Rn  BCCAS Ra  SYNC.B`	
2	**PFETCHM.L2 Ra,Rb,Rn**	**Configure a CME task to pre-fetch a program memory block to the L2 cache**
	`Encoded as: PFETCHB.L2 Rb,Rn  BCCAS Ra  SYNC.B`	
3	**PFETCHM.LCK.L2 Ra,Rb,Rn**	**Configure a CME task to pre-fetch a program memory block to the L2 cache, and lock the updated lines**
	`Encoded as: PFETCHB.LCK.L2 Rb,Rn  BCCAS Ra  SYNC.B`	

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`
Rb	`R0-R31`
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	96-bits	All
Execution unit	2xLSU+PCU	All
No predication		All
Pipeline behavior	per constructing instructions	All

PINITA Initialize Predicate Values (IPU)

General Description

Initialize the values of the selected predicates, up to all six. Each bit that is set in the first immediate operand signifies that the respective predicate should be updated, with the value of the bit in the same position in the second operand. Specific programming rules apply, see rule G.G.4.

Instruction Variants

#	Syntax	Description
1	**PINITA #uena,#uval**	**Initialize the selected predicates to the specified values.**

```
for (unsigned int i = 0 ; i < 6 ; i ++) {
        if (uena (i, i) == 1) {
            P (i) = uval (i, i)
}
}
```

Explicit Operands

Operand	Permitted Values
uena	$0 \le uena < 2^6$
uval	$0 \le uval < 2^6$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Pipeline behavior	agu_AAU_Pbit	All

POP Pop Registers From the Active Stack (LSU)

General Description

Pop one or more registers from the software stack in memory using the active Stack Pointer (SP), which is selected according to the setting of SR2.SPSEL. A pop operation includes the following stages:

1. Pre-decrementing the active SP by the appropriate number of bytes (between 8 and 32) for the access. SP is implicitly updated after the access.
2. Load data from the address in memory pointed by the SP calculation and into the destination registers

Not all core registers can be popped directly - registers that do not appear in the operand list should be popped via an intermediate register. When popping a single register, SP is decremented by 8 even though it is a 4-byte access. The offset of the load (SP-4 or SP-8) depends on whether the operand belongs to a group of Even or Odd registers. Even registers are loaded from SP-8 while Odd registers from SP-4. R and D registers with an even index are considered Even, while those with an odd index are Odd. GCR, BTR0 and TMTAG are arbitrarily defined as Even, while SR, MCTL, BTR1 and TID are defined as Odd. Variants of POP instructions allow popping one, two or four registers at once. It is allowed to group some combinations POP instructions in the VLES, as described in the Programming Rules and in the Address Generation chapters.

Instruction Variants

#	Syntax	Description
1	**POP.2L De:Do**	**Pop two data registers, sign extending the destination registers. SP is decremented by 8.**

	`De = (S40) (SP-8)..(SP-5)`
	`Do = (S40) (SP-4)..(SP-1)`
	`SP = SP -8`	

2	**POP.2L Re:Ro**	**Pop two address registers. SP is decremented by 8.**

	`Re = (SP-8)..(SP-5)`
	`Ro = (SP-4)..(SP-1)`
	`SP = SP -8`	

3	**POP.4L Cxyzw**	**Pop four control registers. SP is decremented by 16.**

	`Cx = (SP-16)..(SP-13)`
	`Cy = (SP-12)..(SP-9)`
	`Cz = (SP-8)..(SP-5)`
	`Cw = (SP-4)..(SP-1)`
	`SP = SP - 16`	

Table continues on the next page...

#	Syntax	Description
4	**POP.4L Rx:Ry:Rz:Rw**	**Pop four address registers. SP is decremented by 16.**

	Rx = (SP-16)..(SP-13)
	Ry = (SP-12)..(SP-9)
	Rz = (SP-8)..(SP-5)
	Rw = (SP-4)..(SP-1)
	SP = SP - 16	

5	**POP.4X Dx:Dy:Dz:Dw**	**Pop four full data registers. Each data register is loaded as 64-bit data, from which the lower 40 bits are updated in the destination. SP is decremented by 32.**

	Dx = (SP-32)..(SP-28)
	Dy = (SP-24)..(SP-20)
	Dz = (SP-16)..(SP-12)
	Dw = (SP-8)..(SP-4)
	SP = SP - 32	

6	**POP.L Ca**	**Pop a control register, from an SP offset that depends if it belongs to the even or odd group. SP is decremented by 8.**

	offset = (Ca even) ? 8 : 4
	Ca = (SP-offset)..(SP-offset+3)
	SP = SP - 8	

7	**POP.L Da**	**Pop 32-bits into a data register, sign extending it in the destination. The load is from an SP offset that depends if the register index is even or odd. SP is decremented by 8.**

	offset = (Da even) ? 8 : 4
	Da = (S40) (SP-offset)..(SP-offset+3)
	SP = SP - 8	

8	**POP.L Ra**	**Pop 32-bits into an address register. The load is from an SP offset that depends if the register index is even or odd. SP is decremented by 8.**

	offset = (Ra even) ? 8 : 4
	Ra = (SP-offset)..(SP-offset+3)
	SP = SP - 8	

Explicit Operands

Operand	Permitted Values
Ca	BTR0, BTR1, GCR, MCTL, SR, TID, TMTAG
Cxyzw	GCR:MCTL:BTR0:BTR1,LR0, LR1, LR2, PROCID:TID:TMTAG,
Da	D0-D63
De:Do	D_{n*2}:D_{n*2+1} 0 ≤ n ≤ 31
Dx:Dy:Dz:Dw	D_{4*n}:D_{4*n+1}:D_{4*n+2}:D_{4*n+3} 0 ≤ n ≤ 15
Ra	R0-R31
Re:Ro	R_{2*n}:R_{2*n+1} 0 ≤ n ≤ 15 Explicit list of all allowed combinations at Table C-23
Rx:Ry:Rz:Rw	R_{4*n}:R_{4*n+1}:R_{4*n+2}:R_{4*n+3} 0 ≤ n ≤ 7

Affect Instructions

Field	Relevant Variants	Comments
SP, ESP, TSP, DSP	All	
SR2.SPSEL	All	

Affected By Instructions

Field	Relevant Variants
Ca	6
Cxyzw	3
SP, ESP, TSP, DSP	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Pipeline behavior	agu_MOVE_STK	All

POPC Pop Registers From the Alternate Stack (LSU)

General Description

Pop one or more registers from the software stack in memory using the alternate Stack Pointer (SP), which is selected according to the setting of SR2.ASPSEL. The task ID that is used in the virtual address is taken from the DID field, regardless of the setting of SR2.EXP. This instruction is typically used by the OS task switch routine to restore registers of a task from the software stack of that task while keeping the OS stack active. The following description is the same as for the regular POP. A pop operation includes the following stages:

1. Pre-decrementing the active SP by the appropriate number of bytes (between 8 and 32) for the access. SP is implicitly updated after the access.
2. Load data from the address in memory pointed by the SP calculation and into the destination registers.

Not all core registers can be popped directly - registers that do not appear in the operand list should be popped via an intermediate register. When popping a single register, SP is decremented by 8 even though it is a 4-byte access. The offset of the load (SP-4 or SP-8) depends on whether the operand belongs to a group of Even or Odd registers. Even registers are loaded from SP-8 while Odd registers from SP-4. R and D registers with an even index are considered Even, while those with an odd index are Odd. GCR, BTR0 and TMTAG are arbitrarily defined as even, while SR, MCTL, BTR1 and TID are defined as Odd. Variants of POP instructions allow popping one, two or four registers at once. It is allowed to group some combinations POP instructions in the VLES, as described in the Programming Rules and in the Address Generation chapters.

Instruction Variants

#	Syntax	Description
1	**POPC.2L De:Do**	**Pop two data registers, sign extending the destination registers. SP is decremented by 8.**

	`ASP selected according to SR2.ASPSEL bits.`
	`De = (S40) (ASP-8)..(ASP-5)`
	`Do = (S40) (ASP-4)..(ASP-1)`
	`ASP = ASP -8`	

2	**POPC.2L Re:Ro**	**Pop two address registers. SP is decremented by 8.**

	`ASP selected according to SR2.ASPSEL bits.`
	`Re = (ASP-8)..(ASP-5)`
	`Ro = (ASP-4)..(ASP-1)`
	`ASP = ASP -8`	

3	**POPC.4L Cxyzw**	**Pop four control registers. SP is decremented by 16.**

	`ASP selected according to SR2.ASPSEL bits.`
	`Cx = (ASP-16)..(ASP-13)`
	`Cy = (ASP-12)..(ASP-9)`
	`Cz = (ASP-8)..(ASP-5)`
	`Cw = (ASP-4)..(ASP-1)`
	`ASP = ASP - 16`	

Table continues on the next page...

#	Syntax	Description
4	**POPC.4L Rx:Ry:Rz:Rw**	**Pop four address registers. SP is decremented by 16.**

	`ASP selected according to SR2.ASPSEL bits.`
	`Rx = (ASP-16)..(ASP-13)`
	`Ry = (ASP-12)..(ASP-9)`
	`Rz = (ASP-8)..(ASP-5)`
	`Rw = (ASP-4)..(ASP-1)`
	`ASP = ASP - 16`	

5	**POPC.4X Dx:Dy:Dz:Dw**	**Pop four full data registers. Each data register is loaded as 64-bit data, from which the lower 40 bits are updated in the destination. SP is decremented by 32.**

	`ASP selected according to SR2.ASPSEL bits.`
	`Dx = (ASP-32)..(ASP-28)`
	`Dy = (ASP-24)..(ASP-20)`
	`Dz = (ASP-16)..(ASP-12)`
	`Dw = (ASP-8)..(ASP-4)`
	`ASP = ASP - 32`	

6	**POPC.L Ca**	**Pop a control register, from an SP offset that depends if it belongs to the even or odd group. SP is decremented by 8.**

	`ASP selected according to SR2.ASPSEL bits.`
	`offset = (Ca even) ? 8 : 4`
	`Ca = (ASP-offset)..(ASP-offset+3)`
	`ASP = ASP - 8`	

7	**POPC.L Da**	**Pop 32-bits into a data register, sign extending it in the destination. The load is from an SP offset that depends if the register index is even or odd. SP is decremented by 8.**

	`ASP selected according to SR2.ASPSEL bits.`
	`offset = (Da even) ? 8 : 4`
	`Da = (S40) (ASP-offset)..(ASP-offset+3)`
	`ASP = ASP - 8`	

8	**POPC.L Ra**	**Pop 32-bits into an address register. The load is from an SP offset that depends if the register index is even or odd. SP is decremented by 8.**

	`ASP selected according to SR2.ASPSEL bits.`
	`offset = (Ra even) ? 8 : 4`
	`Ra = (ASP-offset)..(ASP-offset+3)`
	`ASP = ASP - 8`	

Explicit Operands

Operand	Permitted Values
Ca	BTR0, BTR1, GCR, MCTL, SR, TID, TMTAG
Cxyzw	GCR:MCTL:BTR0:BTR1,LR0, LR1, LR2, PROCID:TID:TMTAG,
Da	D0-D63
De:Do	$D_{n*2}:D_{n*2+1}$ 0 ≤ n ≤ 31
Dx:Dy:Dz:Dw	$D_{4*n}:D_{4*n+1}:D_{4*n+2}:D_{4*n+3}$ 0 ≤ n ≤ 15

Table continues on the next page...

Operand	Permitted Values
Ra	R0-R31
Re:Ro	$R_{2*n}:R_{2*n+1}$ $0 \le n \le 15$ Explicit list of all allowed combinations at Table C-23
Rx:Ry:Rz:Rw	$R_{4*n}:R_{4*n+1}:R_{4*n+2}:R_{4*n+3}$ $0 \le n \le 7$

Affect Instructions

Field	Relevant Variants	Comments
ESP, TSP, DSP	All	
SR2.ASPSEL	All	
TID	All	

Affected By Instructions

Field	Relevant Variants
Ca	6
Cxyzw	3
ESP, TSP, DSP	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Pipeline behavior	agu_MOVE_STK	All

PRDA Save, Swap and Restore a Subset of Predicates to/From a Register (IPU)

General Description

This instruction group is used to help in managing virtual predicates. For more information on virtual predicates and how these instructions are to be used, see the Program Control chapter.

Flag Options

Flag	Position	Description
RST	flg1	Restore
SAV	flg1	Save
SWP	flg1	Swap

Instruction Variants

<table>
<tr><th>#</th><th>Syntax</th><th>Description</th></tr>
<tr><td>1</td><td><b>PRDA.RST #uena,#u12,Ra</b></td><td><b>Restore marked predicates from a register</b></td></tr>
<tr><td></td><td colspan="2"><pre>for (i=0;i<6;i++)
{ if(uena[i] == 1)
  { P[i]=Ra[u12[i*2+1:i*2]*6+i] }
}</pre></td></tr>
<tr><td>2</td><td><b>PRDA.SAV #uena,#u12,Rn</b></td><td><b>Save marked predicates to a register</b></td></tr>
<tr><td></td><td colspan="2"><pre>for (i=0;i<6;i++)
{ if (uena[i] == 1)
  {
   Rn[u12[i*2+1:i*2]*6+i]=P[i] }
}</pre></td></tr>
<tr><td>3</td><td><b>PRDA.SWP #uena,#u12,Rn</b></td><td><b>Swap marked predicates with a register</b></td></tr>
<tr><td></td><td colspan="2"><pre>for (i=0;i<6;i++)
{ if (uena[i] == 1)
  { P[i]=Rn[u12[i*2+1:i*2]*6+i]
   Rn[u12[i*2+1:i*2]*6+i]=P[i] }
}</pre></td></tr>
</table>

Explicit Operands

Operand	Permitted Values
Ra	R0-R31
Rn	R0-R31
u12	$0 \le u12 < 2^{12}$
uena	$0 \le uena < 2^{6}$

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	2
Encoding length	48-bits	All
Pipeline behavior	agu_AAU_Pbit	All

PUNLOCKB Block Unlock Lines in the Instruction Cache (LSU)

General Description

This instruction is a constituent of the PUNLOCKM meta-instruction, used in configuring the CME (Cache Management Engine) to unlock a block of program data in the L2 cache. The PUNLOCKB instruction should not be used in this form but only through PUNLOCKM; using the instruction on its own will give indeterminate results. For more information on the PUNLOCKM instruction, see its description page and also the Address Generation chapter.

Flag Options

Flag	Position	Description
L2	flg2	Directed to L2 cache

Instruction Variants

#	Syntax	Description
1	**PUNLOCKB.L2 Ra,Rn**	**Participate in configuring a CME task to unlock lines in the L2 cache that belong to a program memory block.**
	`Rn = success status of the CME block command request`	

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Helper instruction		All
No predication		All
Pipeline behavior	agu_MOVE_LD_cmd	All

PUNLOCKM Block Instruction Cache Unlock (META INST)

General Description

This meta-instruction configures the CME (Cache Management Engine) to unlock lines in the L2 cache that belong to a program memory block. The meta-instruction is encoded in 3 instructions - a PUNLOCKB instruction, a BCCAS instruction, and a SYNC.B instruction. The configuration data sent to the CME is taken from Ra and Rb. The CME returns a success or failure status of the configuration stage that is stored in the destination Rn. The exact format of the data in the input and output registers is described in the Address Generation chapter. Some programming rules apply to the this instruction, see rule A.9. For more information on block cache instructions, refer to the Address Generation chapter.

Flag Options

Flag	Position	Description
L2	flg2	Directed to L2 cache

Instruction Variants

#	Syntax	Description
1	**PUNLOCKM.L2 Ra,Rb,Rn**	**Configure a CME task to unlock lines in the L2 cache that belong to a program memory block**
	`Encoded as: PUNLOCKB.L2 Rb,Rn  BCCAS Ra  SYNC.B`	

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`
Rb	`R0-R31`
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	96-bits	All
Execution unit	2xLSU+PCU	All
No predication		All
Pipeline behavior	per constructing instructions	All

PUSH Push Registers to the Active Stack (LSU)

General Description

Push one or more registers to the software stack in memory using the active Stack Pointer (SP), which is selected according to the setting of SR2.SPSEL. A push operations includes the following stages:

1. Store data from the source registers to address in memory pointed by SP
2. Implicitly incrementing the active SP by the appropriate number of bytes (between 8 and 32) for the access.

Not all core registers can be pushed directly - registers that do not appear in the operand list should be pushed via an intermediate register. When pushing a single register, SP is incremented by 8 even though it is a 4-byte access. The offset of the store (SP or SP+4) depends on whether the operand belongs to a group of Even or Odd registers. Even registers are stored to SP while Odd registers to SP+4. R and D registers with an even index are considered Even, while those with an odd index are Odd. GCR, BTR0 and TMTAG are arbitrarily defined as Even, while SR, MCTL, BTR1 and TID are defined as Odd. Variants of PUSH instructions allow pushing one, two or four registers at once. It is allowed to group some combinations PUSH instructions in the VLES, as described in the Programming Rules and in the Address Generation chapters.

Instruction Variants

#	Syntax	Description
1	**PUSH.2L De:Do**	**Push two data registers. SP is incremented by 8.**

	`(SP)..(SP+3) = De.M`
	`(SP+4)..(SP+7) = Do.M`
	`SP = SP + 8`	

2	**PUSH.2L Re:Ro**	**Push two address registers. SP is incremented by 8.**

	`(SP)..(SP+3) = Re`
	`(SP+4)..(SP+7) = Ro`
	`SP = SP + 8`	

3	**PUSH.4L Cxyzw**	**Push four control registers. One combination includes only 3 registers - the empty memory slot should be considered undefined in memory. SP is incremented by 8.**

	`(SP)..(SP+3) = Cx`
	`(SP+4)..(SP+7) = Cy`
	`(SP+8)..(SP+11) = Cz`
	`(SP+12)..(SP+15) = Cw`
	`SP = SP + 16`	

Table continues on the next page...

#	Syntax	Description
4	**PUSH.4L Rx:Ry:Rz:Rw**	**Push four address registers. SP is incremented by 16.**

	`(SP)..(SP+3) = Rx`
	`(SP+4)..(SP+7) = Ry`
	`(SP+8)..(SP+11) = Rz`
	`(SP+12)..(SP+15) = Rw`
	`SP = SP + 16`	

5	**PUSH.4X Dx:Dy:Dz:Dw**	**Push four full data registers. Each data register is stored as 64-bit data, from which the lower 40 bits are taken from the source register and the rest are padded with zeros. SP is incremented by 32.**

	`(SP)..(SP+7) = {0x000000, Dx}`
	`(SP+8)..(SP+15) = {0x000000, Dy}`
	`(SP+16)..(SP+23) = {0x000000, Dz}`
	`(SP+24)..(SP+31) = {0x000000, Dw}`
	`SP = SP + 31`	

6	**PUSH.L Ca**	**Push a control register, from an SP offset that depends if it belongs to the even or odd group. SP is incremented by 8.**

	`offset = Ca even ? 0 : 4`
	`(SP+offset)..(SP+offset+3) = Ca`
	`SP = SP + 8`	

7	**PUSH.L Da**	**Push the lower 32-bits of a data register. The store is to an SP offset that depends if the register index is even or odd. SP is incremented by 8.**

	`offset = Da even ? 0 : 4`
	`(SP+offset)..(SP+offset+3) = Da.M`
	`SP = SP + 8`	

8	**PUSH.L Ra**	**Push an address register. The store is to an SP offset that depends if the register index is even or odd. SP is incremented by 8.**

	`offset = Ra even ? 0 : 4`
	`(SP+offset)..(SP+offset+3) = Ra`
	`SP = SP + 8`	

Explicit Operands

Operand	Permitted Values
Ca	BTR0, BTR1, GCR, MCTL, SR, TID, TMTAG
Cxyzw	GCR:MCTL:BTR0:BTR1,LR0, LR1, LR2, PROCID:TID:TMTAG,
Da	D0-D63
De:Do	D_{n*2}:D_{n*2+1} $0 \le n \le 31$
Dx:Dy:Dz:Dw	D_{4*n}:D_{4*n+1}:D_{4*n+2}:D_{4*n+3} $0 \le n \le 15$
Ra	R0-R31
Re:Ro	R_{2*n}:R_{2*n+1} $0 \le n \le 15$ Explicit list of all allowed combinations at Table C-23
Rx:Ry:Rz:Rw	R_{4*n}:R_{4*n+1}:R_{4*n+2}:R_{4*n+3} $0 \le n \le 7$

Affect Instructions

Field	Relevant Variants	Comments
Ca	6	
Cxyzw	3	
SP, ESP, TSP, DSP	All	
SR2.SPSEL	All	

Affected By Instructions

Field	Relevant Variants
SP, ESP, TSP, DSP	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Pipeline behavior	agu_MOVE_STK	All

PUSHC Push Registers to the Alternate Stack (LSU)

General Description

Push one or more registers to the software stack in memory using the using the alternate Stack Pointer (SP), which is selected according to the setting of SR2.ASPSEL. The task ID that is used in the virtual address is taken from the DID field, regardless of the setting of SR2.EXP. This instruction is typically used by the OS task switch routine to save registers of a task to the software stack of that task while keeping the OS stack active. The following description is the same as for the regular PUSH. A push operations includes the following stages:

1. Store data from the source registers to address in memory pointed by SP
2. Implicitly incrementing the active SP by the appropriate number of bytes (between 8 and 32) for the access.

Not all core registers can be pushed directly - registers that do not appear in the operand list should be pushed via an intermediate register. When pushing a single register, SP is incremented by 8 even though it is a 4-byte access. The offset of the store (SP or SP+4) depends on whether the operand belongs to a group of Even or Odd registers. Even registers are stored to SP while Odd registers to SP+4. R and D registers with an even index are considered Even, while those with an odd index are Odd. GCR, BTR0 and TMTAG are arbitrarily defined as even, while SR, MCTL, BTR1 and TID are defined as odd. Variants of PUSH instructions allow pushing one, two or four registers at once. It is allowed to group some combinations PUSH instructions in the VLES, as described in the Programming Rules and in the Address Generation chapters.

Instruction Variants

#	Syntax	Description
1	**PUSHC.2L De:Do**	**Push two data registers. SP is incremented by 8.**

	`ASP selected according to ASPSEL bits.`
	`(ASP)..(ASP+3) = De.M`
	`(ASP+4)..(ASP+7) = Do.M`
	`ASP = ASP + 8`	

2	**PUSHC.2L Re:Ro**	**Push two address registers. SP is incremented by 8.**

	`ASP selected according to ASPSEL bits.`
	`(SP)..(SP+3) = Re`
	`(SP+4)..(SP+7) = Ro`
	`ASP = ASP + 8`	

3	**PUSHC.4L Cxyzw**	**Push four control registers. One combination includes only 3 registers - the empty memory slot should be considered undefined in memory. SP is incremented by 8.**

	`ASP selected according to ASPSEL bits.`
	`(ASP)..(ASP+3) = Cx`
	`(ASP+4)..(ASP+7) = Cy`
	`(ASP+8)..(ASP+11) = Cz`
	`(ASP+12)..(ASP+15) = Cw`
	`ASP = ASP + 16`	

Table continues on the next page...

#	Syntax	Description
4	**PUSHC.4L Rx:Ry:Rz:Rw**	**Push four address registers. SP is incremented by 16.**

	`ASP selected according to ASPSEL bits.`
	`(ASP)..(ASP+3) = Rx`
	`(ASP+4)..(ASP+7) = Ry`
	`(ASP+8)..(ASP+11) = Rz`
	`(ASP+12)..(ASP+15) = Rw`
	`ASP = ASP + 16`	

5	**PUSHC.4X Dx:Dy:Dz:Dw**	**Push four full data registers. Each data register is stored as 64-bit data, from which the lower 40 bits are taken from the source register and the rest are padded with zeros. SP is incremented by 32.**

	`ASP selected according to ASPSEL bits.`
	`(ASP)..(ASP+7) = {0x000000, Dx}`
	`(ASP+8)..(ASP+15) = {0x000000, Dy}`
	`(ASP+16)..(ASP+23) = {0x000000, Dz}`
	`(ASP+24)..(ASP+31) = {0x000000, Dw}`
	`ASP = ASP + 31`	

6	**PUSHC.L Ca**	**Push a control register, from an SP offset that depends if it belongs to the even or odd group. SP is incremented by 8.**

	`ASP selected according to ASPSEL bits.`
	`offset = Ca even ? 0 : 4`
	`(ASP+offset)..(ASP+offset+3) = Ca`
	`ASP = ASP + 8`	

7	**PUSHC.L Da**	**Push the lower 32-bits of a data register. The store is to an SP offset that depends if the register index is even or odd. SP is incremented by 8.**

	`ASP selected according to ASPSEL bits.`
	`offset = Da even ? 0 : 4`
	`(ASP+offset)..(ASP+offset+3) = Da.M`
	`ASP = ASP + 8`	

8	**PUSHC.L Ra**	**Push an address register. The store is to an SP offset that depends if the register index is even or odd. SP is incremented by 8.**

	`ASP selected according to ASPSEL bits.`
	`offset = Ra even ? 0 : 4`
	`(ASP+offset)..(ASP+offset+3) = Ra`
	`ASP = ASP + 8`	

Explicit Operands

Operand	Permitted Values
Ca	BTR0, BTR1, GCR, MCTL, SR, TID, TMTAG
Cxyzw	GCR:MCTL:BTR0:BTR1,LR0, LR1, LR2, PROCID:TID:TMTAG,
Da	D0-D63
De:Do	$D_{n*2}:D_{n*2+1}$ $0 \le n \le 31$
Dx:Dy:Dz:Dw	$D_{4*n}:D_{4*n+1}:D_{4*n+2}:D_{4*n+3}$ $0 \le n \le 15$

Table continues on the next page...

Operand	Permitted Values
Ra	R0-R31
Re:Ro	$R_{2*n}:R_{2*n+1}$ $0 \le n \le 15$ Explicit list of all allowed combinations at Table C-23
Rx:Ry:Rz:Rw	$R_{4*n}:R_{4*n+1}:R_{4*n+2}:R_{4*n+3}$ $0 \le n \le 7$

Affect Instructions

Field	Relevant Variants	Comments
Ca	6	
Cxyzw	3	
ESP, TSP, DSP	All	
SR2.ASPSEL	All	
TID	All	

Affected By Instructions

Field	Relevant Variants
ESP, TSP, DSP	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Pipeline behavior	agu_MOVE_STK	All

RECIP Reciprocal (DALU)

General Description

This instruction participates in the calculation of the reciprocal of the input multiplied by 2: 1/(2*X), where X is the input for the RECIP instruction. It calculates two parameters, A and B, that are used for linear approximation of the result y = -A*X+B using a subsequent instruction MACM.SU.X.
For a correct result the input value X should be a fraction in the range of 0.5 ≤ X < 1. The accuracy of the result after the two instruction sequence is 15 bits. The accuracy could be improved to 30 bits by applying an iteration of the Newton-Raphson algorithm. The following code is an example that includes a Newton-Raphson iteration:

```
recip d3,d0:d1 ; d3 input
macm.su.x -d3.h,d0,d1 ; d1 estimation of 1/(2*d3)
; Newton-Raphson iteration:
mpy32.sr.x d1,d1,d5 ; d5 = d1 * d1
mpy32.sr.x d5,d3,d4 ; d4 = d5 * d3
sub.x d4,d1,d9 ; d9 = 1/(2*d3)
```

Instruction Variants

#	Syntax	Description
1	**RECIP Da,Dm:Dn**	**Reciprocal Approximation**

	`Dm.M = {f_recip_LUT_A (Da[29:24]), 0}`
	`Dn.M = {1, f_recip_LUT_B (Da[29:24]), 0}`
	`Dm.E = 0`
	`Dn.E = 0`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

RND.nX Round (DALU)

General Description

A legacy instruction that rounds a 40-bit value according to the setting of SR.SCM, SR.SM and SR.RM.

Flag Options

Flag	Position	Description
LEG	flg2	Legacy instruction

Instruction Variants

#	Syntax	Operation	Description
1	RND.LEG.X Da,Dn	`Dn = srRND (Da)`	Legacy round of 40-bit value

Explicit Operands

Operand	Permitted Values
Da	`D0-D63`
Dn	`D0-D63`

Affect Instructions

Field	Relevant Variants	Comments
SR.RM	All	
SR.SCM	All	
SR.SM	All	

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

ROL.X Rotate 40 Bits Left by One (DALU)

General Description

Rotates the bits in the destination data register and the Carry bit (SR.C) one position to the left

Instruction Variants

#	Syntax	Description
1	**ROL.X Da,Dn**	**Rotate 1 bit left through the Carry bit**

	Dn = {(Da << 1)[39:1], SR.C}
	SR.C = (Da << 1) (40)	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Dn	D0-D63

Affect Instructions

Field	Relevant Variants	Comments
SR.C	All	

Affected By Instructions

Field	Relevant Variants
SR.C	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

ROR.X Rotate 40 Bits Right by One (DALU)

General Description

Rotates the bits in the destination data register and the Carry bit (SR.C) one position to the right

Instruction Variants

#	Syntax	Operation	Description

1	ROR.X Da,Dn	`Dn = {SR.C, Da} >> 1`
		`SR.C = Da[0]`
			Rotate 1 bit right through the Carry bit

Explicit Operands

Operand	Permitted Values
Da	`D0-D63`
Dn	`D0-D63`

Affect Instructions

Field	Relevant Variants	Comments
SR.C	All	

Affected By Instructions

Field	Relevant Variants
SR.C	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

RTE Return From Exception (PCU_LSU)

General Description

Pops PC, SR, SR2 and EIDR from the active stack in one 128-bit access, pre-decrementing the active SP by 16, and then jumps to the restored PC. During the jump, a full pipe flush is performed, not affected by the BTB, and execution resumes from the destination with the newly restored SR2 settings. The CIC (Clear Internal Context) variant allows to clear in parallel some micro-architectural context so that it does not affect the code running afterwards, such as the BTB and the RAS. For more information on exceptions and related states see the Program Control chapter.

Flag Options

Flag	Position	Description
CIC	flg1	Clear Internal Context

Instruction Variants

#	Syntax	Operation	Description

1	RTE	`PC = (SP - 16)..(SP - 13)`
		`SR = (SP - 12)..(SP - 9)`
		`SR2 = (SP - 8)..(SP - 5)`
		`EIDR = (SP - 4)..(SP - 1)`
		`SP = SP - 16`
			Return from exception without clearing the BTB and RAS; Equivalent to RTE.CIC #0.

2	RTE.CIC #u5	`PC = (SP - 16)..(SP - 13)`
		`SR = (SP - 12)..(SP - 9)`
		`SR2 = (SP - 8)..(SP - 5)`
		`EIDR = (SP - 4)..(SP - 1)`
		`SP = SP - 16`
		`(if u5[0]) then clear BTB`
		`(if u5[1]) then clear RAS`
			Return from exception optionally clearing the BTB (if u5[0] is set) and the RAS (if u5[1] is set).

Explicit Operands

Operand	Permitted Values
u5	$0 \le u5 < 2^5$

Affect Instructions

Field	Relevant Variants	Comments
SP, ESP, TSP, DSP	All	
SR2.SPSEL	All	

Affected By Instructions

Field	Relevant Variants
EIDR	All
SP, ESP, TSP, DSP	All

Table continues on the next page...

Field	Relevant Variants
SR	All
SR2	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Must be alone in VLES		All
No predication		All
Pipeline behavior	pcu_COF	All

RTS Return From Subroutine (PCU_LSU)

General Description

Pops PC from the active stack, pre-decrementing SP by 8, and resume execution from the restored PC. If the RAS is valid, the internally stored PC is used as the destination instead of the value loaded from the stack. RTS by default is accelerated by the BTB, and with the NOBTB variant is not affected by it. The RTS.STK variant invalidates the RAS and forces the core to use the PC that is loaded from the stack. This variant should be used in case the return address may have been manipulated in the stack memory.

Flag Options

Flag	Position	Description
NOBTB	flg1	Does not use nor updates the BTB
STK	flg1	Restore from the stack, ignoring RAS

Instruction Variants

#	Syntax	Description
1	**RTS[.NOBTB]**	**Returns from a subroutine. If the RAS is valid, restores the PC from the RAS register. By default is subject to BTB acceleration, unless the NOBTB flag is used.**

```
(RAS valid) ? PC = top of RAS : PC = (SP - 8)..(SP - 5)
always SP = SP - 8
```

#	Syntax	Description
2	**RTS.STK**	**Returns from a subroutine. Uses the PC that is restored from the stack, and invalidates the RAS. Not accelerated by the BTB.**

```
PC = (SP - 8)..(SP - 5)
SP = SP - 8
Invalidate RAS
```

Affect Instructions

Field	Relevant Variants	Comments
RAS	1	
SP, ESP, TSP, DSP	All	
SR2.SPSEL	All	

Affected By Instructions

Field	Relevant Variants
RAS	1
SP, ESP, TSP, DSP	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Pipeline behavior	pcu_COF	All

SAD.nB Sum of Absolute Byte Differences (DALU)

General Description

The SAD instruction subtracts each byte of the first source register (or register pair) from the respective byte of the second source register (or register pair), treating each byte as unsigned. It then adds the absolute value of all subtraction results to the lower portion (L) of the destination register, zero-extending the higher portion (H) and the extension of the destination register.

Instruction Variants

<table>
<tr><th>#</th><th>Syntax</th><th>Description</th></tr>
<tr><td><b>1</b></td><td><b>SAD.4B Da,Db,Dn</b></td><td><b>Sum of 4 absolute byte difference</b></td></tr>
<tr><td></td><td colspan="2"><code>Dn = (U40)((U17)Dn.L + (U17)((| ((U9)Db.LH - (U9)Da.LH) |) + (| ((U9)Db.LL - (U9)Da.LL) |)) + (U17)((| ((U9)Db.HH - (U9)Da.HH) |) + (| ((U9)Db.HL - (U9)Da.HL) |)))[15:0]</code></td></tr>
<tr><td><b>2</b></td><td><b>SAD.8B Da:Db,Dc:Dd,Dn</b></td><td><b>Sum of 8 absolute byte difference</b></td></tr>
<tr><td></td><td colspan="2"><code>Dn = (U40)((U17)Dn.L + (U17)(((| ((U9)Dd.LH - (U9)Db.LH) |) + (| ((U9)Dd.LL - (U9)Db.LL) |)) + (| ((U9)Dc.LH - (U9)Da.LH) |) + (| ((U9)Dc.LL - (U9)Da.LL) |)) + (U17)(((| ((U9)Dd.HH - (U9)Db.HH) |) + (| ((U9)Dd.HL - (U9)Db.HL) |)) + (| ((U9)Dc.HH - (U9)Da.HH) |) + (| ((U9)Dc.HL - (U9)Da.HL) |)))[15:0]</code></td></tr>
</table>

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU_Y	All

SAT.nB Saturate to Byte Integer Value (DALU)

General Description

The variants dealing with 16-bits (W in flag1 position) saturate two or four 16-bit values to a signed or unsigned byte. If the input value is greater than 0x127 (or 0x255 if unsigned), it is saturated to 0x127 (or 0x255). If the input value is less than -0x128 (or negative if unsigned), it is saturated to -0x128 (or cleared). The result is sign extended to a 20-bit value. The extension of the source registers is ignored.
The variant dealing with 20-bit input (T in flag1) saturates two 20-bit values to two unsigned bytes packed in the destination register.

Flag Options

Flag	Position	Description
T	flg1	20 bits
W	flg1	Word (16 bits)
U	flg2	Unsigned

Instruction Variants

#	Syntax	Description
1	**SAT.W.2B Da,Dn**	**SIMD2 saturate to signed byte of 16-bit signed in one register**

	`Dn.WH = (S20)SAT8 ((S40)(Da.H))`
	`Dn.WL = (S20)SAT8 ((S40)(Da.L))`	

2	**SAT.W.4B Da:Db,Dm:Dn**	**SIMD4 saturate to signed byte of 16-bit signed in two registers**

	`Dm.WH = (S20)SAT8 ((S40)(Da.H))`
	`Dm.WL = (S20)SAT8 ((S40)(Da.L))`
	`Dn.WH = (S20)SAT8 ((S40)(Db.H))`
	`Dn.WL = (S20)SAT8 ((S40)(Db.L))`	

3	**SAT.T.U.2B Da,Dn**	**SIMD2 saturation of two 20-bit portions to unsigned bytes**

	`Dn.WH = SATU8 (Da.WH)`
	`Dn.WL = SATU8 (Da.WL)`
	`Alias, encoded as: CLIP.T.U.2B #0,Da,Dn`	

4	**SAT.W.U.4B Da:Db,Dm:Dn**	**SIMD4 saturate to unsigned byte of 16-bit signed in two registers**

	`Dm.WH = SATU8 ((S40)(Da.H))`
	`Dm.WL = SATU8 ((S40)(Da.L))`
	`Dn.WH = SATU8 ((S40)(Db.H))`
	`Dn.WL = SATU8 ((S40)(Db.L))`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Alias		3
Architecture	SC3900FP only	2
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	4
	dalu_DAU_Y	1, 2
	dalu_LBM	3

SAT.nF Saturate to 16-bit Fractional Value (DALU)

General Description

Saturates one or two 40-bit fractions into a 16-bit fraction value.

If the input value is greater than 0x00_7FFF_FFFF, it is saturated to 0x00_7FFF_0000. If the input value is less than 0xFF_8000_0000 (or -1), it is saturated to 0xFF_8000_0000. The low 16-bit of the result are cleared in any case.

The SAT.SC variants perform scaling and saturation according to the settings of SR.SM and SR.SCM. These instructions are used for mapping legacy StarCore code.

Flag Options

Flag	Position	Description
SC	flg1	Scale according to SR.SCM
X	flg1	40 bit input

Instruction Variants

#	Syntax	Description
1	**SAT.SC.2F Da:Db,Dm:Dn**	**SIMD2 Scale and saturate according to SCM bits**

```
If (SR.SM=1)
     Dm = {(S24) (SAT16(Da[39:16])), 0x0000}
     Dn = {(S24) (SAT16(Db[39:16])), 0x0000}
Else
if (SR.SCM = 0b00) then
      Dm = {(S24) (SAT16(Da[39:16])), 0x0000}
      Dn = {(S24) (SAT16(Db[39:16])), 0x0000}
else if (SR.SCM = 0b01) then
       Dm = {(S24) (SAT16(Da[39:16] >>1)), 0x0000}
       Dn = {(S24) (SAT16(Db[39:16] >>1)), 0x0000}
else if (SR.SCM = 0b10) then
       Dm = {(S24) (SAT16(Da[39:15] <<1 )), 0x0000}
       Dn = {(S24) (SAT16(Db[39:15] <<1 )), 0x0000}
```

#	Syntax	Description
2	**SAT.SC.F Da,Dn**	**Scale and saturate according to SCM bits**

```
If (SR.SM=1)
     Dn = {(S24) (SAT16(Da[39:16])), 0x0000}
Else
if (SR.SCM = 0b00) then Dn = {(S24)(SAT16(Da[39:16])) , 0x0000}
else if (SR.SCM = 0b01) then Dn = {(S24) (SAT16(Da[39:16] >>1)) , 0x0000}
else if (SR.SCM = 0b10) then Dn = {(S24) (SAT16(Da[39:15] <<1)) , 0x0000}
```

#	Syntax	Description
3	**SAT.X.F Da,Dn**	**Saturate Fractional Data Register**

```
If (Da > 0x00_7FFF_FFFF)
 Dn = 0x00_7FFF_0000
else if (Da < 0xFF_8000_0000)
  Dn = 0xFF_8000_0000
else
  Dn = Da & 0xFF_FFFF_0000
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \leq n \leq 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \leq n \leq 62$
Dn	D0-D63	

Affect Instructions

Field	Relevant Variants	Comments
SR.SCM	1, 2	
SR.SM	1, 2	

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

SAT.nL Saturate to Long (32-bit) Integer Value (DALU)

General Description

Saturates one or two 40-bit signed integers or fractions into signed or unsigned 32-bit values.

For signed variants, if the input value is greater than 0x00_7FFF_FFFF, it is saturated to 0x00_7FFF_FFFF. If the input value is less than 0xFF_8000_0000, it is saturated to 0xFF_8000_0000.

For unsigned variants, if the input value is greater than 0x00_FFFF_FFFF, it is saturated to 0x00_FFFF_FFFF. If the input value is negative, it is saturated to 0.

Flag Options

Flag	Position	Description
X	flg1	Cross between the arguments
U	flg2	Unsigned

Instruction Variants

#	Syntax	Description
1	**SAT.X.2L Da:Db,Dm:Dn**	**SIMD2 Saturate long (32-bit) data register**

	`Dm = (S40) SAT32(Da)`
	`Dn = (S40) SAT32(Db)`	

2	**SAT.X.L Da,Dn**	**Saturate long (32-bit) data register**

	`if (Da > 00_7FFF_FFFF)`
	`  Dn = 0x00_7FFF_FFFF`
	`else if (Da < 0xFF_8000_0000)`
	`  Dn = 0xFF_8000_0000`
	`else Dn = Da`	

3	**SAT.X.U.L Da,Dn**	**Saturate unsigned long (32-bit) data register**
	`Dn = (U40) SATU32(Da)`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \leq n \leq 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \leq n \leq 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

SAT.nW Saturate to 16-bit Integer Value (DALU)

General Description

Saturates one/two/four 40-bit/32-bit/20-bit signed integers into signed/unsigned 16-bit values.

For signed variants, if the input value is greater than 0x7FFF, it is saturated to 0x7FFF. If the input value is less than 0x8000, it is saturated to 0x8000.

No packing is performed and the result is sign extended from 16 bits to 20 bits or 40 bits according to the size of the input (20-bit or 32-bit/40-bit).

For unsigned variants, if input value is greater than 0xFFFF, it is saturated to 0xFFFF. If input value is negative, it is saturated to 0.
No packing is performed and number is zero extended from 16-bit to 20-bit or 40-bit according to input size (20-bit or 32-bit/40-bit)

Flag Options

Flag	Position	Description
L	flg1	32 bit input
T	flg1	20 bits
X	flg1	40 bit input
U	flg2	Unsigned

Instruction Variants

#	Syntax	Description
1	**SAT.L.W Da,Dn**	**Saturate long (32-bit) integer to 16-bit signed value and sign extend to 40-bit**

```
Dn = (S40) SAT16(Da.M)
```

2	**SAT.T.2W Da,Dn**	**SIMD2 Saturate two 20-bit integer to 16-bit signed value and sign extend to 20-bit**

```
if (Da.WL > 0x07FFF)
 Dn.WL = 0x07FFF
else if (Da.WL < 0xF8000)
Dn.WL = 0xF8000
else Dn.WL = Da.WL
if (Da.WH > 0x07FFF)
  Dn.WH = 0x07FFF
else if (Da.WH < 0xF8000)
  Dn.WH = F8000
else Dn.WH = Da.WH
```

3	**SAT.T.4W Da:Db,Dm:Dn**	**SIMD4 Saturate two 20-bit integer to 16-bit signed value and sign extend to 20-bit**

Table continues on the next page...

#	Syntax	Description

	Dm.WH = (S20) SAT16(Da.WH)
	Dm.WL = (S20) SAT16(Da.WL)
	Dn.WH = (S20) SAT16(Db.WH)
	Dn.WL = (S20) SAT16(Db.WL)	

4	**SAT.X.W Da,Dn**	**Saturate 40-bit integer to 16-bit signed value and sign extend to 40-bit**

	If (Da > 00_0000_7FFF) then
	    Dn = 0x 00_0000_7FFF
	else if (Da < 0xFF_FFFF_8000) then
	    Dn = 0xFF_FFFF_8000
	else Dn = Da	

5	**SAT.L.U.W Da,Dn**	**Saturate long (32-bit) integer to 16-bit unsigned value and sign extend to 40-bit**
	Dn = (U40) SATU16(Da.M)	
6	**SAT.T.U.2W Da,Dn**	**SIMD2 Saturate two 20-bit integer to 16-bit unsigned value and sign extend to 20-bit**

	Dn.WH = (U20) SATU16(Da.WH)
	Dn.WL = (U20) SATU16(Da.WL)	

7	**SAT.X.U.2W Da:Db,Dm:Dn**	**SIMD2 Saturate 40-bit integer into 16-bit unsigned value and sign extend to 40-bit**

	Dm = (U40) SATU16(Da)
	Dn = (U40) SATU16(Db)	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	2, 3, 5, 6
	dalu_DAU_Y	1, 4, 7

SAT.nX Saturate to 40-bit Integer Signed Value (DALU)

General Description

Saturates a 72-bit signed integer located in a register pair into a signed 40-bit value. The 72 bits consist of the 40 bits from the first register in the pair, concatenated with the lower 32 bits of the second register.

If the input value is greater than 0x00_0000_007F_FFFF_FFFF, it is saturated to 0x7F_FFFF_FFFF. If the input value is less than 0xFF_FFFF_FF80_0000_0000, it is saturated to 0x80_0000_0000.

The instruction can be used to cast a 64-bit signed value into a 40-bit value only if the high register is sign extended in the extension. This can be achieved by using SAT.X.L on the high register before using this instruction.

Flag Options

Flag	Position	Description
LL	flg1	SIMD multiplication: low by low portions Extract and Sat instructions: 64-bit input

Instruction Variants

#	Syntax	Operation	Description
1	SAT.LL.X Da:Db,Dn	`Dn = SAT40( {Da , Db.M} )`	Saturate and cast a 72-bit value into a 40-bit value

Explicit Operands

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU_Y	All

SCALE.nF Scale By SCM (DALU)

General Description

The instruction scales one or two 40-bit input values up or dows according to the settings of SR.SM and SR.SCM, using the same numerical behavior as was done for the MOVES instruction in legacy StarCore architectures. This means that in saturation mode (SR.SM = 1) no scaling is done. However, scaling is done into the destination registers and not to memory. Also, SR.S is not affected by this instruction. This instruction is usefull in mapping legacy StarCore instructions to SC3900.

Instruction Variants

#	Syntax	Operation	Description

1	SCALE.2F Da:Db,Dm:Dn	`Dm = SCALE_LEG (Da)`
		`Dn = SCALE_LEG (Db)`
			SIMD2 Scale by SCM

2	SCALE.F Da,Dn	`Dn = SCALE_LEG (Da)`	Scale by SCM

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Affect Instructions

Field	Relevant Variants	Comments
SR.SCM	All	
SR.SM	All	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

SETALIGN Set DOALIGN Shift Size (IPU)

General Description

Reads the GCR, copies the two least significant bits of the source address register into the shift size bits (the two least significant bits of the BAM field), and writes the modified register value back to the GCR. Bits 31-2 of the GCR are not changed.

The DOALIGN and SETALIGN instructions are used together to accelerate processing of byte operands that are stored consecutively in memory. With these instructions the user can emulate an unaligned access to four bytes.

Instruction Variants

#	Syntax	Operation	Description
1	SETALIGN Ra,GCR.BAM	`GCR[1:0] = Ra[1:0]`	Set DOALIGN shift size

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`

Affected By Instructions

Field	Relevant Variants
GCR.BAM	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Pipeline behavior	agu_AAU_LOGIC	All

SIGN.4W Sign Word (DALU)

General Description

For each four 16-bit words of the input, sets the output to 1 if the input is positive (greater than zero), to -1 if input is negative and 0 if input is zero

Source extension is ignored, destination extension is cleared

Instruction Variants

#	Syntax	Description
1	**SIGN.4W Da:Db,Dm:Dn**	**SIMD4 sign word**

```
Dm.L = Da.L  > 0 ? 1 : (Da.L == 0) ? 0 : 0xffff
Dm.H = Da.H  > 0 ? 1 : (Da.H == 0) ? 0 : 0xffff
Dm.E = 0
Dn.L = Db.L  > 0 ? 1 : (Db.L == 0) ? 0 : 0xffff
Dn.H = Db.H  > 0 ? 1 : (Db.H == 0) ? 0 : 0xffff
Dn.E = 0
```

Explicit Operands

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

SKIP Test and Skip Over a Hardware Loop (PCU)

General Description

Tests the value of the LCn register, using signed or unsigned arithmetic. If LCn < 1, branches to the PC-relative address. This instruction is used when the loop is enabled from a register and there is no a-priory knowledge that the loops should execute at all. The instruction is placed after the loop was enabled with DOEN, and before the LOOPSTART directive. The destination is typically set to the VLES following the loop (LA+1). In case the SKIP instruction is placed at SA-1, the assembler will encode it with the loop start mark (LPST) as a combined instruction called LPSKIP. If it is placed before SA-1, it is encoded as an independent instruction.

Flag Options

Flag	Position	Description
n2	flg1	Loop index (0, 1, 2, 3)
U	flg2	Unsigned

Instruction Variants

#	Syntax	Description
1	**SKIP.n2 LABEL**	**Test LCn with signed arithmetic and skip if LCn < 1**
	`If ( (s32) LCn<1) then PC = PC + {RelAdd19,0}`	
2	**SKIP.n2.U LABEL**	**Test LCn with unsigned arithmetic and skip if LCn < 1**
	`If ( (u32) LCn==0) then PC = PC + {RelAdd19,0}`	

Explicit Operands

Operand	Permitted Values
RelAdd19	$-2^{18} \le \mathtt{RelAdd19} < 2^{18}$
n2	$0 \le \mathtt{n2} \le 3$

Affect Instructions

Field	Relevant Variants	Comments
LC	All	
PC	All	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	48-bits	All
Pipeline behavior	pcu_COF_LP	All

SOD.nT Sum or Difference Function and Cross of 20-bit Values (DALU)

General Description

This instruction adds and/or subtracts 16-bit or 20-bit values that are packed in one or two regisetrs, with variants allowing to select what portions are added or subtracted.

SIMD2 variants perform two separate 16-bit or 20-bit (depending on if in 20-bit mode or not) additions or subtractions between the high and low portions of two source data registers and stores the results in the two portions of the destination data register. If not in 20-bit mode, the extension of the destination register is cleared.

Different variants of the instruction are defined by different mnemonic flags represented by a four-letter code. The first two letters stand for the function: A for addition and S for subtraction; the last two letters stand for lane selection: XX for crossed (H added with L and vice versa) and II for not crossed (H added with H and L added with L).

SIMD4 variants perform two operations as the above in parallel. ffffc and ggggc are a shorthand notation representing several specific A/S/I/X combinations - see the operand table below.

Flag Options

Flag	Position	Description
AAII	flg1	A: Add respective portions. I: No cross of operand portions
AAXX	flg1	A: Add respective portions. X: Cross of operand portions
ASII	flg1	A: Add respective portions. S: Subtract respective portions. I: No cross of operand portions
ASXX	flg1	A: Add respective portions. S: Subtract respective portions. X: Cross of operand portions
SAII	flg1	A: add respective portions. S: subtract respective portions. I: no cross of operand portions
SAXX	flg1	A: add respective portions. S: subtract respective portions. X: cross of operand portions
SSII	flg1	S: Subtract respective portions. I: No cross of operand portions
SSXX	flg1	S: Subtract respective portions. X: Cross of operand portions
ffffc	flg1	Represents specific A/S/I/X combinations - see operand table below
ggggc	flg1	Represents specific A/S/I combinations - see operand table below

Instruction Variants

#	Syntax	Description
1	**SOD.AAII.2T Da,Db,Dn**	**A: add respective portions. I: no cross of operand portions. Effectively performs ADD.2T instruction.**

```
if (SR.SM2==1 && SR.W20==0)
  Dn.H = srSAT16(Db.H + Da.H) ; Dn.L = srSAT16(Db.L + Da.L); Dn.E=0
else
  Dn.WH = (S20) (Db.WH + Da.WH) ; Dn.WL = (S20) (Db.WL + Da.WL)
```

Table continues on the next page...

#	Syntax	Description
2	**SOD.AAXX.2T Da,Db,Dn**	**A: add respective portions. X: cross of operand portions**

	`if (SR.SM2==1 && SR.W20==0)`
	`  Dn.H = srSAT16(Db.H + Da.L) ; Dn.L = srSAT16(Db.L + Da.H); Dn.E=0`
	`else`
	`  Dn.WH = (S20) (Db.WH + Da.WL) ; Dn.WL = (S20) (Db.WL + Da.WH)`	

3	**SOD.ASII.2T Da,Db,Dn**	**A: add respective portions. S: subtract respective portions. I: no cross of operand portions**

	`if (SR.SM2==1 && SR.W20==0)`
	`  Dn.H = srSAT16(Db.H + Da.H) ; Dn.L = srSAT16(Db.L - Da.L); Dn.E=0`
	`else`
	`  Dn.WH = (S20) (Db.WH + Da.WH) ; Dn.WL = (S20) (Db.WL - Da.WL)`	

4	**SOD.ASXX.2T Da,Db,Dn**	**A: add respective portions. S: subtract respective portions. X: cross of operand portions**

	`if (SR.SM2==1 && SR.W20==0)`
	`  Dn.H = srSAT16(Db.H + Da.L) ; Dn.L = srSAT16(Db.L - Da.H); Dn.E=0`
	`else`
	`  Dn.WH = (S20) (Db.WH + Da.WL) ; Dn.WL = (S20) (Db.WL - Da.WH)`	

5	**SOD.SAII.2T Da,Db,Dn**	**A: add respective portions. S: subtract respective portions. I: no cross of operand portions**

	`if (SR.SM2==1 && SR.W20==0)`
	`  Dn.H = srSAT16(Db.H - Da.H) ; Dn.L = srSAT16(Db.L + Da.L); Dn.E=0`
	`else`
	`  Dn.WH = (S20) (Db.WH - Da.WH) ; Dn.WL = (S20) (Db.WL + Da.WL)`	

6	**SOD.SAXX.2T Da,Db,Dn**	**A: add respective portions. S: subtract respective portions. X: cross of operand portions**

	`if (SR.SM2==1 && SR.W20==0)`
	`  Dn.H = srSAT16(Db.H - Da.L) ; Dn.L = srSAT16(Db.L + Da.H); Dn.E=0`
	`else`
	`  Dn.WH = (S20) (Db.WH - Da.WL) ; Dn.WL = (S20) (Db.WL + Da.WH)`	

7	**SOD.SSII.2T Da,Db,Dn**	**S: subtract respective portions. I: no cross of operand portions Effectively performs SUB.2T instruction.**

	`if (SR.SM2==1 && SR.W20==0)`
	`  Dn.H = srSAT16(Db.H - Da.H) ; Dn.L = srSAT16(Db.L - Da.L); Dn.E=0`
	`else`
	`  Dn.WH = (S20) (Db.WH - Da.WH) ; Dn.WL = (S20) (Db.WL - Da.WL)`	

8	**SOD.SSXX.2T Da,Db,Dn**	**S: subtract respective portions. X: cross of operand portions**

	`if (SR.SM2==1 && SR.W20==0)`
	`  Dn.H = srSAT16(Db.H - Da.L) ; Dn.L = srSAT16(Db.L - Da.H); Dn.E=0`
	`else`
	`  Dn.WH = (S20) (Db.WH - Da.WL) ; Dn.WL = (S20) (Db.WL - Da.WH)`	

9	**SOD.ffffc.4T Da:Db,Dc:Dd,Dm:Dn**	**ffffc flg1 represents specific A/S/I/X combinations - see operand table below**

Table continues on the next page...

#	Syntax	Description

	`f = [A,S] ; A ~ + ; S ~ -`
	`c = [I,X]`
	`c = 'I' ; Da:Db input words not swapped`
	`Dm.WH = ( Dc.WH +/- Da.WH)`
	`Dm.WL = ( Dc.WL +/- Da.WL)`
	`Dn.WH = ( Dd.WH +/- Db.WH)`
	`Dn.WL = ( Dd.WL +/- Db.WL)`
	`c = 'X' ; Da:Db input words swapped`
	`Dm.WH = ( Dc.WH +/- Da.WL)`
	`Dm.WL = ( Dc.WL +/- Da.WH)`
	`Dn.WH = ( Dd.WH +/- Db.WL)`
	`Dn.WL = ( Dd.WL +/- Db.WH)`	

10	**SOD.ggggc.4T Da,Db,Dm:Dn**	**ggggc flg1 represents specific A/S/I combinations - see operand table below**

	`f = [A,S] ; A ~ + ; S ~ -`
	`c = 'I' ; Da:Db input words not swapped`
	`Dm.WH = ( Db.WH +/- Da.WH)`
	`Dm.WL = ( Db.WL +/- Da.WL)`
	`Dn.WH = ( Db.WH +/- Da.WH)`
	`Dn.WL = ( Db.WL +/- Da.WL)`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	
ffffc	AAAAX, SSSSX, ASASX, SASAX, AAAAI, SSSSI, ASASI, SASAI	
ggggc	AASSI, SSAAI	

Affect Instructions

Field	Relevant Variants	Comments
SR.SM2	1, 2, 3, 4, 5, 6, 7, 8	
SR.W20	1, 2, 3, 4, 5, 6, 7, 8	

Affected By Instructions

Field	Relevant Variants
SR.SAT	1, 2, 3, 4, 5, 6, 7, 8

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

SOD.nW Sum or Difference Function, Cross and Saturate of 16-bit Values (DALU)

General Description

This instruction adds and/or subtracts with saturation 16-bit values that are packed in one or two regisetrs, with variants allowing to select what portions are added or subtracted.

SIMD2 variants perform two separate 16-bit additions or subtractions between the high and low portions of two source data registers, saturate and store the results in the two portions of the destination data register. The extensions of the source register is ignored and the extension of the destination is cleared.

Different variants of the instruction are defined by different mnemonic flags represented by a four-letter code. The first two letters stand for the function: A for addition and S for subtraction; the last two letters stand for lane selection: XX for crossed (H added with L and vice versa) and II for not crossed (H added with H and L added with L).

SIMD4 variants perform two operations as the above in parallel. ffffc and ggggc are a shorthand notation representing several specific A/S/I/X combinations - see the operand table below.

Flag Options

Flag	Position	Description
AAII	flg1	A: Add respective portions. I: No cross of operand portions
AAXX	flg1	A: Add respective portions. X: Cross of operand portions
ASII	flg1	A: Add respective portions. S: Subtract respective portions. I: No cross of operand portions
ASXX	flg1	A: Add respective portions. S: Subtract respective portions. X: Cross of operand portions
SAII	flg1	A: add respective portions. S: subtract respective portions. I: no cross of operand portions
SAXX	flg1	A: add respective portions. S: subtract respective portions. X: cross of operand portions
SSII	flg1	S: Subtract respective portions. I: No cross of operand portions
SSXX	flg1	S: Subtract respective portions. X: Cross of operand portions
ffffc	flg1	Represents specific A/S/I/X combinations - see operand table below
ggggc	flg1	Represents specific A/S/I combinations - see operand table below
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**SOD.AAII.S.2W Da,Db,Dn**	**A: add respective portions. I: no cross of operand portions**

```
Dn.H = (U20) SAT16(Da.H + Db.H)
Dn.L = (U20) SAT16(Da.L + Db.L)
```

Table continues on the next page...

#	Syntax	Description
2	**SOD.AAXX.S.2W Da,Db,Dn**	**A: add respective portions. X: cross of operand portions**

	`Dn.H = (U20) SAT16(Da.L + Db.H)`
	`Dn.L = (U20) SAT16(Da.H + Db.L)`	

3	**SOD.ASII.S.2W Da,Db,Dn**	**A: add respective portions. S: subtract respective portions. I: no cross of operand portions**

	`Dn.H = SAT16(Da.H + Db.H)`
	`Dn.L = SAT16(Db.L - Da.L)`
	`Dn.E = 0`	

4	**SOD.ASXX.S.2W Da,Db,Dn**	**A: add respective portions. S: subtract respective portions. X: cross of operand portions**

	`Dn.H = SAT16(Db.H + Da.L)`
	`Dn.L = SAT16(Db.L - Da.H)`
	`Dn.E = 0`	

5	**SOD.SAII.S.2W Da,Db,Dn**	**A: add respective portions. S: subtract respective portions. I: no cross of operand portions**

	`Dn.H = SAT16(Db.H - Da.H)`
	`Dn.L = SAT16(Db.L + Da.L)`
	`Dn.E = 0`	

6	**SOD.SAXX.S.2W Da,Db,Dn**	**A: add respective portions. S: subtract respective portions. X: cross of operand portions**

	`Dn.H = SAT16(Db.H - Da.L)`
	`Dn.L = SAT16(Db.L + Da.H)`
	`Dn.E = 0`	

7	**SOD.SSII.S.2W Da,Db,Dn**	**S: subtract respective portions. I: no cross of operand portions**

	`Dn.H = SAT16(Db.H - Da.H)`
	`Dn.L = SAT16(Db.L - Da.L)`
	`Dn.E = 0`	

8	**SOD.SSXX.S.2W Da,Db,Dn**	**S: subtract respective portions. X: cross of operand portions**

	`Dn.H = SAT16(Db.H - Da.L)`
	`Dn.L = SAT16(Db.L - Da.H)`
	`Dn.E = 0`	

9	**SOD.ffffc.S.4W Da:Db,Dc:Dd,Dm:Dn**	**ffffc flg1 represents specific A/S/I/X combinations - see operand table below**

Table continues on the next page...

#	Syntax	Description

	f = [A,S] ; A ~ + ; S ~ -
	c = [I,X]
	c = 'I' ; Da:Db input words not swapped
	Dm.H = SAT16(Dc.H +/- Da.H)
	Dm.L = SAT16(Dc.L +/- Da.L)
	Dn.H = SAT16(Dd.H +/- Db.H)
	Dn.L = SAT16(Dd.L +/- Db.L)
	c = 'X' ; Da:Db input words swapped
	Dm.H = SAT16(Dc.H +/- Da.L)
	Dm.L = SAT16(Dc.L +/- Da.H)
	Dn.H = SAT16(Dd.H +/- Db.L)
	Dn.L = SAT16(Dd.L +/- Db.H)
	In any case D{nm}.E = 0	

10	**SOD.ggggc.S.4W Da,Db,Dm:Dn**	**ggggc flg1 represents specific A/S/I combinations - see operand table below**

	f = [A,S] ; A ~ + ; S ~ -
	c = 'I' ; Da:Db input words not swapped
	Dm.H = SAT16(Db.H +/- Da.H)
	Dm.L = SAT16(Db.L +/- Da.L)
	Dn.H = SAT16(Db.H +/- Da.H)
	Dn.L = SAT16(Db.L +/- Da.L)
	D{nm}.E = 0	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	0 ≤ n ≤ 62
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	0 ≤ n ≤ 62
Dm:Dn	D_n:D_{n+1}	0 ≤ n ≤ 62
Dn	D0-D63	
ffffc	AAAAX, SSSSX, ASASX, SASAX, AAAAI, SSSSI, ASASI, SASAI	
ggggc	AASSI, SSAAI	

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

SOD.nX Sum or Difference Function, Cross and Saturate of 40-bit Values (DALU)

General Description

Performs two separate 40-bit additions or subtractions between the two couples of four source data registers, optionally saturate the results to 32 bits (in the S variants), and stores the results in the two destination data registers.

Different variants of the instruction are defined by different mnemonic flags represented by a four-letter code. The first two letters stand for the function: A for addition and S for subtraction; the last two letters stand for lane selection: XX for crossed (H added with L and vice versa) and II for not crossed (H added with H and L added with L).

Variants without XX or II as flags have only two source registers and perform sum and difference operations on the same source registers.

Flag Options

Flag	Position	Description
AAII	flg1	A: Add respective portions. I: No cross of operand portions
AAXX	flg1	A: Add respective portions. X: Cross of operand portions
AS	flg1	A: Add respective portions. S: Subtract respective portions
ASII	flg1	A: Add respective portions. S: Subtract respective portions. I: No cross of operand portions
ASXX	flg1	A: Add respective portions. S: Subtract respective portions. X: Cross of operand portions
SA	flg1	A: add respective portions. S: subtract respective portions
SAII	flg1	A: add respective portions. S: subtract respective portions. I: no cross of operand portions
SAXX	flg1	A: add respective portions. S: subtract respective portions. X: cross of operand portions
SSII	flg1	S: Subtract respective portions. I: No cross of operand portions
SSXX	flg1	S: Subtract respective portions. X: Cross of operand portions
S	flg2	Saturated result

Instruction Variants

#	Syntax	Operation	Description

1	SOD.AAII.2X Da:Db,Dc:Dd,Dm:Dn	`Dm = (Da + Dc)[39:0]`
		`Dn = (Db + Dd)[39:0]`
			A: add respective portions. I: no cross of operand portions

2	SOD.AAXX.2X Da:Db,Dc:Dd,Dm:Dn	`Dm = (Dc + Db)[39:0]`
		`Dn = (Dd + Da)[39:0]`
			A: add respective portions. X: cross of operand portions

3	SOD.AS.2X Da,Db,Dm:Dn	`Dm = (Db + Da)[39:0]`
		`Dn = (Db - Da)[39:0]`
			A: add respective portions. S: subtract respective portions

4	SOD.ASII.2X Da:Db,Dc:Dd,Dm:Dn	`Dm = (Da + Dc)[39:0]`
		`Dn = (Dd - Db)[39:0]`
			A: add respective portions. S: subtract respective portions. I: no cross of operand portions

Table continues on the next page...

#	Syntax	Operation	Description

5	SOD.ASXX.2X Da:Db,Dc:Dd,Dm:Dn	`Dm = (Dc + Db)[39:0]`
		`Dn = (Dd - Da)[39:0]`
			A: add respective portions. S: subtract respective portions. X: cross of operand portions

6	SOD.SA.2X Da,Db,Dm:Dn	`Dm = (Db - Da)[39:0]`
		`Dn = (Db + Da)[39:0]`
			A: add respective portions. S: subtract respective portions

7	SOD.SAII.2X Da:Db,Dc:Dd,Dm:Dn	`Dm = (Dc - Da)[39:0]`
		`Dn = (Dd + Db)[39:0]`
			A: add respective portions. S: subtract respective portions. I: no cross of operand portions

8	SOD.SAXX.2X Da:Db,Dc:Dd,Dm:Dn	`Dm = (Dc - Db)[39:0]`
		`Dn = (Dd + Da)[39:0]`
			A: add respective portions. S: subtract respective portions. X: cross of operand portions

9	SOD.SSII.2X Da:Db,Dc:Dd,Dm:Dn	`Dm = (Dc - Da)[39:0]`
		`Dn = (Dd - Db)[39:0]`
			S: subtract respective portions. I: no cross of operand portions

10	SOD.SSXX.2X Da:Db,Dc:Dd,Dm:Dn	`Dm = (Dc - Db)[39:0]`
		`Dn = (Dd - Da)[39:0]`
			S: subtract respective portions. X: cross of operand portions

11	SOD.AAII.S.2X Da:Db,Dc:Dd,Dm:Dn	`Dm = SAT32((Da + Dc))`
		`Dn = SAT32((Dd + Db))`
			SOD with saturation - A: add respective portions. I: no cross of operand portions

12	SOD.AAXX.S.2X Da:Db,Dc:Dd,Dm:Dn	`Dm = SAT32((Dc + Db))`
		`Dn = SAT32((Dd + Da))`
			SOD with saturation - A: add respective portions. X: cross of operand portions

13	SOD.AS.S.2X Da,Db,Dm:Dn	`Dm = SAT32((Db + Da))`
		`Dn = SAT32((Db - Da))`
			SOD with saturation - v

14	SOD.ASII.S.2X Da:Db,Dc:Dd,Dm:Dn	`Dm = SAT32((Da + Dc))`
		`Dn = SAT32((Dd - Db))`
			SOD with saturation - A: add respective portions. S: subtract respective portions. I: no cross of operand portions

15	SOD.ASXX.S.2X Da:Db,Dc:Dd,Dm:Dn	`Dm = SAT32((Dc + Db))`
		`Dn = SAT32((Dd - Da))`
			SOD with saturation - A: add respective portions. S: subtract respective portions. X: cross of operand portions

16	SOD.SA.S.2X Da,Db,Dm:Dn	`Dm = SAT32((Db - Da))`
		`Dn = SAT32((Db + Da))`
			SOD with saturation - A: add respective portions. S: subtract respective portions

17	SOD.SAII.S.2X Da:Db,Dc:Dd,Dm:Dn	`Dm = SAT32((Dc - Da))`
		`Dn = SAT32((Dd + Db))`
			SOD with saturation - A: add respective portions. S: subtract respective portions. I: no cross of operand portions

18	SOD.SAXX.S.2X Da:Db,Dc:Dd,Dm:Dn	`Dm = SAT32((Dc - Db))`
		`Dn = SAT32((Dd + Da))`
			SOD with saturation - A: add respective portions. S: subtract respective portions. X: cross of operand portions

19	SOD.SSII.S.2X Da:Db,Dc:Dd,Dm:Dn	`Dm = SAT32((Dc - Da))`
		`Dn = SAT32((Dd - Db))`
			SOD with saturation - S: subtract respective portions. I: no cross of operand portions

20	SOD.SSXX.S.2X Da:Db,Dc:Dd,Dm:Dn	`Dm = SAT32((Dc - Db))`
		`Dn = SAT32((Dd - Da))`
			SOD with saturation - S: subtract respective portions. X: cross of operand portions

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Affected By Instructions

Field	Relevant Variants
SR.SAT	11, 12, 13, 14, 15, 16, 17, 18, 19, 20

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

SODHL.nT Sum or Difference Function and Cross of 20-bit Values on Intra-Registers (DALU)

General Description

Performs two separate 20-bit additions or subtractions between either high or the low portions of two source data registers and stores the results in the two portions of the destination data register.

Different variants of the instruction are defined by different mnemonic flags represented by a four-letter code, where each letter defines the operation between the respective portions: A for addition and S for subtraction.

Flag Options

Flag	Position	Description
ASAS	flg1	A: Add respective portions. S: Subtract respective portions
SASA	flg1	A: add respective portions. S: subtract respective portions

Instruction Variants

#	Syntax	Description
1	**SODHL.ASAS.4T Da:Db,Dc:Dd,Dm:Dn**	**Add-Sub Intra-register Sum or Difference of four 20b Values**

	`Dm.WH = ((S21)Da.WH + (S21)Dc.WL)[19:0]`
	`Dm.WL = ((S21)Da.WH - (S21)Dc.WL)[19:0]`
	`Dn.WH = ((S21)Db.WH + (S21)Dd.WL)[19:0]`
	`Dn.WL = ((S21)Db.WH - (S21)Dd.WL)[19:0]`	

2	**SODHL.SASA.4T Da:Db,Dc:Dd,Dm:Dn**	**Sub-Add Intra-register Sum or Difference of four 20b Values**

	`Dm.WH = ((S21)Da.WH - (S21)Dc.WL)[19:0]`
	`Dm.WL = ((S21)Da.WH + (S21)Dc.WL)[19:0]`
	`Dn.WH = ((S21)Db.WH - (S21)Dd.WL)[19:0]`
	`Dn.WL = ((S21)Db.WH + (S21)Dd.WL)[19:0]`	

Explicit Operands

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

SODHL.nW Sum or Difference Function and Cross of 16-bit Values on Intra-Registers (DALU)

General Description

Performs two separate 16-bit additions or subtractions between either the high or the low portions of two source data registers, saturate and stores the results in the two portions of the destination data register.

The extension of the source registers is ignored and the extension of the destination is cleared.

Different variants of the instruction are defined by different mnemonic flags represented by a four-letter code, where each letter defines the operation between the respective portions: A for addition and S for subtraction.

Flag Options

Flag	Position	Description
ASAS	flg1	A: Add respective portions. S: Subtract respective portions
SASA	flg1	A: add respective portions. S: subtract respective portions
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**SODHL.ASAS.S.4W Da:Db,Dc:Dd,Dm:Dn**	**Add-Sub Intra-register Sum or Difference of four 20b Values**

	`Dm.WH = (U20)SAT16 (((S40)Da.H + (S40)Dc.L))`
	`Dm.WL = (U20)SAT16 (((S40)Da.H - (S40)Dc.L))`
	`Dn.WH = (U20)SAT16 (((S40)Db.H + (S40)Dd.L))`
	`Dn.WL = (U20)SAT16 (((S40)Db.H - (S40)Dd.L))`	

2	**SODHL.SASA.S.4W Da:Db,Dc:Dd,Dm:Dn**	**Sub-Add Intra-register Sum or Difference of four 20b Values**

	`Dm.WH = (U20)SAT16 (((S40)Da.H - (S40)Dc.L))`
	`Dm.WL = (U20)SAT16 (((S40)Da.H + (S40)Dc.L))`
	`Dn.WH = (U20)SAT16 (((S40)Db.H - (S40)Dd.L))`
	`Dn.WL = (U20)SAT16 (((S40)Db.H + (S40)Dd.L))`	

Explicit Operands

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dc:Dd	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Affected By Instructions

Field	Relevant Variants
SR.SAT	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

ST.BTR Store data from BTR to memory (LSU)

General Description

Store data to memory from BTR0, BTR1 or both. Instruction variants allow storing different register portions from BTR1: either the most significant byte (byte HH), the most significant word (portion H) or the whole register (32-bits). BTR0 can only be stored as a whole (32-bits). Another variant allows to store both BTR0 and BTR1 together. The addressing modes of these instructions are limited to either no update (Rn), post increment (Rn)+ or post decrement (Rn)-.

Instruction Variants

#	Syntax	Description
1	**ST.2L BTR0:BTR1,(Rn)**	**Store BTR0 and BTR1 (64-bits total) to memory, using no update addressing mode**
	`(EA)..(EA+7) = {BTR0, BTR1}`	
2	**ST.2L BTR0:BTR1,(Rn)+/-**	**Store BTR0 and BTR1 (64-bits total) to memory, using either (Rn)+ or (Rn)- addressing modes**
	`(EA)..(EA+7) = {BTR0, BTR1}`	
3	**ST.BF BTR1,(Rn)**	**Store the most significant byte (HH) from BTR1, using no update addressing mode**
	`(EA)..(EA+0) = BTR1[31:24]`	
4	**ST.BF BTR1,(Rn)+/-**	**Store the most significant byte (HH) from BTR1, using either (Rn)+ or (Rn)- addressing modes**
	`(EA)..(EA+0) = BTR1[31:24]`	
5	**ST.F BTR1,(Rn)**	**Store the most significant 16-bit word (H) from BTR1, using no update addressing mode**
	`(EA)..(EA+1) = BTR1[31:16]`	
6	**ST.F BTR1,(Rn)+/-**	**Store the most significant 16-bit word (H) from BTR1, using either (Rn)+ or (Rn)- addressing modes**
	`(EA)..(EA+1) = BTR1[31:16]`	
7	**ST.L BTR1,(Rn)+/-**	**Store BTR1 (32-bits), using either (Rn)+ or (Rn)- addressing modes**
	`(EA)..(EA+3) = BTR1`	
8	**ST.L BTRN,(Rn)**	**Store either BTR0 or BTR1, using either (Rn)+ or (Rn)- addressing modes (four variants)**
	`(Rn)..(Rn+3) = BTRN`	

Explicit Operands

Operand	Permitted Values	
BTRN	`BTR0,`	`BTR1`
Rn	`R0-R31`	

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	2, 4, 6, 7	
BTR0	1, 2	
BTR1	1, 2, 3, 4, 5, 6, 7	
BTRN	8	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Pipeline behavior	agu_MOVE_ST	1, 3, 5, 8
	agu_MOVE_ST_POSTINC	2, 4, 6, 7

ST.SRS.Wx.nF→ D Store 16-bit Fractional Words From the WH/WL Portion of Data Registers With Scale, Round and Saturation (20 bits to 16) (LSU)

General Description

Store 16-bit fractional words from data registers to memory after scaling, rounding and saturating the data from 20-bit portions (WH or WL) prior to issuing them to memory. Data in the source registers is not changed. SRS processing includes the following steps:

1. Scaling according to the setting of SR.SCM. Scaling could be seen as shift up or down of the of bits in the source register from which the data to store is taken from, relative to the normal H or L portion which is the non-scaled position of wide fractional words.
2. Rounding according to SR.RM, which looks at the bits in the register to the right of the data window.
3. Signed saturation is performed if the bits to the left of the data window, including the 4 bits of the extension portion, are not a clean sign extension of the most significant bit in the data window. SR.S is optionally set according to the output, see the bit description in SR.

For more information on the functionality of SRS see the DALU chapter in this manual. All variants read 8 inputs from 8 registers, and use wither WH or WL as the 20-bit input from each register. All Standard addressing modes supported.

Flag Options

Flag	Position	Description
SRS	flg1	Scaling, rounding and saturation of data during a memory store (may vary per instruction)
WH	flg2	Wide-high portion
WL	flg2	Wide-low portion

Instruction Variants

#	Syntax	Description
1	**ST.SRS.WH.8F Da::Dh,(SAM)**	**Store eight fractional words taken from the wide-high (WH) portion of eight data registers, with scaling, rounding and saturation. All standard addressing modes are supported.**

```
(EA)..(EA+1) = SRS_20_to_16(Da.WH)
(EA+2)..(EA+3) = SRS_20_to_16(Db.WH)
...
(EA+14)..(EA+15) = SRS_20_to_16(Dh.WH)
```

Table continues on the next page...

#	Syntax	Description
2	**ST.SRS.WL.8F Da::Dh,(SAM)**	**Store eight fractional words taken from the wide-low (WL) portion of eight data registers, with scaling, rounding and saturation. All standard addressing modes are supported.**

```
(EA)..(EA+1) = SRS_20_to_16(Da.WL)
(EA+2)..(EA+3) = SRS_20_to_16(Db.WL)
...
(EA+14)..(EA+15) = SRS_20_to_16(Dh.WL)
```

Explicit Operands

Operand	Permitted Values	
Da::Dh	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$	$0 \le n \le 56$
Rk	R0-R7,R16-R23	
Rn	R0-R31	

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	All	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR.RM	All	
SR.SCM	All	
SR2.SPSEL	All	For SP-relative addressing modes

Affected By Instructions

Field	Relevant Variants
SR.S	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits 48-bits	All
Pipeline behavior	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_IMM agu_MOVE_ST_PRECALC_R	2
	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_R agu_MOVE_ST_PRECALC_IMM	1

ST.SRS.nBF→ D Store Fractional Bytes From Data Registers With Scale, Round and Saturation (40 bits to 8) (LSU)

General Description

Store fractional bytes from data registers to memory after scaling, rounding and saturating the data from 40-bit sources prior to issuing them to memory. Data in the source registers is not changed. SRS processing includes the following steps:

1. Scaling according to the setting of SR.SCM. Scaling could be seen as shift up or down of the of bits in the source register from which the data to store is taken from, relative to the normal HH byte which is the non-scaled position of fractional bytes.
2. Rounding according to SR.RM, which looks at the bits in the register to the right of the data window.
3. Signed saturation is performed if the bits to the left of the data window, including the extension, are not a clean sign extension of the most significant bit in the data window. SR.S is optionally set according to the output, see the bit description in SR.

For more information on the functionality of SRS see the DALU chapter in this manual. SIMD variants store two, four or eight fractional bytes taken from different registers. Legacy variants (LEG) have in addition an implicit dependency on SR.SM - if it is set, then no scaling is done. All Standard addressing modes supported, except for LEG variants.

Flag Options

Flag	Position	Description
SRS	flg1	Scaling, rounding and saturation of data during a memory store (may vary per instruction)
LEG	flg2	Legacy instruction

Instruction Variants

#	Syntax	Description
1	**ST.SRS.2BF Da:Db,(SAM)**	**Store two fractional bytes with scaling, rounding and saturation, from two data registers. All standard addressing modes are supported.**

```
(EA) = SRS_40_to_8(Da)
(EA+1) = SRS_40_to_8(Db)
```

Table continues on the next page...

#	Syntax	Description
2	**ST.SRS.4BF Da:Db:Dc:Dd,(SAM)**	**Store four fractional bytes with scaling, rounding and saturation, from four data registers. All standard addressing modes are supported.**

	`(EA)   = SRS_40_to_8(Da)`
	`(EA+1) = SRS_40_to_8(Db)`
	`(EA+2) = SRS_40_to_8(Dc)`
	`(EA+3) = SRS_40_to_8(Dd)`	

3	**ST.SRS.8BF Da::Dh,(SAM)**	**Store eight fractional bytes with scaling, rounding and saturation, from eight data registers. All standard addressing modes are supported.**

	`(EA)   = SRS_40_to_8(Da)`
	`(EA+1) = SRS_40_to_8(Db)`
	`...`
	`(EA+7) = SRS_40_to_8(Dh)`	

4	**ST.SRS.BF Da,(SAM)**	**Store a fractional byte with scaling, rounding and saturation. All standard addressing modes are supported.**
	`(EA) = SRS_40_to_8(Da)`	
5	**ST.SRS.LEG.2BF Da:Db,(Rn)**	**Store two fractional bytes with scaling, rounding and saturation, from two data registers. Scaling disabled if SR.SM is set. Only uses (Rn) addressing mode.**

	`(EA)   = SRS_40_to_8_leg(Da)`
	`(EA+1) = SRS_40_to_8_leg(Db)`	

6	**ST.SRS.LEG.4BF Da:Db:Dc:Dd,(Rn)**	**Store four fractional bytes with scaling, rounding and saturation, from four data registers. Scaling disabled if SR.SM is set. Only uses (Rn) addressing mode.**

	`(EA)   = SRS_40_to_8_leg(Da)`
	`(EA+1) = SRS_40_to_8_leg(Db)`
	`(EA+2) = SRS_40_to_8_leg(Dc)`
	`(EA+3) = SRS_40_to_8_leg(Dd)`	

7	**ST.SRS.LEG.BF Da,(Rn)**	**Store a fractional byte with scaling, rounding and saturation. Scaling disabled if SR.SM is set. Only uses (Rn) addressing mode.**
	`(EA) = SRS_40_to_8_leg(Da)`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da::Dh	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$	$0 \le n \le 56$
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Da:Db:Dc:Dd	$D_n:D_{n+1}:D_{n+2}:D_{n+3}$	$0 \le n \le 60$
Rk	R0-R7,R16-R23	
Rn	R0-R31	

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2, 3, 4	For post-modify addressing modes
SP, ESP, TSP, DSP	1, 2, 3, 4	For SP-relative addressing modes
SR.RM	All	
SR.SCM	All	
SR.SM	5, 6, 7	
SR2.SPSEL	1, 2, 3, 4	For SP-relative addressing modes

Affected By Instructions

Field	Relevant Variants
SR.S	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits	5, 6, 7
	32-bits 48-bits	1, 2, 3, 4
Pipeline behavior	agu_MOVE_ST	5, 6, 7
	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_IMM agu_MOVE_ST_PRECALC_R	1, 2, 3, 4

ST.SRS.nF→ D Store 16-bit Fractional Words From Data Registers With Scale, Round and Saturation (40 bits to 16) (LSU)

General Description

Store 16-bit fractional words from data registers to memory after scaling, rounding and saturating the data from 40-bit sources prior to issuing them to memory. Data in the source registers is not changed. SRS processing includes the following steps:

1. Scaling according to the setting of SR.SCM. Scaling could be seen as shift up or down of the of bits in the source register from which the data to store is taken from, relative to the normal H portion which is the non-scaled position of fractional words.
2. Rounding according to SR.RM, which looks at the bits in the register to the right of the data window.
3. Signed saturation is performed if the bits to the left of the data window, including the extension, are not a clean sign extension of the most significant bit in the data window. SR.S is optionally set according to the output, see the bit description in SR.

For more information on the functionality of SRS see the DALU chapter in this manual. SIMD variants store two, four or eight fractional words taken from different registers. Legacy variants (LEG) have in addition an implicit dependency on SR.SM - if it is set, then no scaling is done. All Standard addressing modes supported, except for LEG variants.

Flag Options

Flag	Position	Description
SRS	flg1	Scaling, rounding and saturation of data during a memory store (may vary per instruction)
LEG	flg2	Legacy instruction

Instruction Variants

#	Syntax	Description
1	**ST.SRS.2F Da:Db,(SAM)**	**Store two fractional words with scaling, rounding and saturation, from two data registers. All standard addressing modes are supported.**

```
(EA)..(EA+1) = SRS_40_to_16(Da)
(EA+2)..(EA+3) = SRS_40_to_16(Db)
```

Table continues on the next page...

#	Syntax	Description
2	**ST.SRS.4F Da:Db:Dc:Dd,(SAM)**	**Store four fractional words with scaling, rounding and saturation, from four data registers. All standard addressing modes are supported.**

	`(EA)..(EA+1) = SRS_40_to_16(Da)`
	`(EA+2)..(EA+3) = SRS_40_to_16(Db)`
	`(EA+4)..(EA+5) = SRS_40_to_16(Dc)`
	`(EA+6)..(EA+7) = SRS_40_to_16(Dd)`	

3	**ST.SRS.8F Da::Dh,(SAM)**	**Store eight fractional words with scaling, rounding and saturation, from eight data registers. All standard addressing modes are supported.**

	`(EA)..(EA+1) = SRS_40_to_16(Da)`
	`(EA+2)..(EA+3) = SRS_40_to_16(Db)`
	`...`
	`(EA+14)..(EA+15) = SRS_40_to_16(Dh)`	

4	**ST.SRS.F Da,(SAM)**	**Store a fractional word with scaling, rounding and saturation. All standard addressing modes are supported.**
	`(EA)..(EA+1) = SRS_40_to_16(Da)`	
5	**ST.SRS.LEG.2F Da:Db,(Rn)**	**Store two fractional words with scaling, rounding and saturation, from two data registers. Scaling disabled if SR.SM is set. Only uses (Rn) addressing mode.**

	`(EA)..(EA+1) = SRS_40_to_16_leg(Da)`
	`(EA+2)..(EA+3) = SRS_40_to_16_leg(Db)`	

6	**ST.SRS.LEG.4F Da:Db:Dc:Dd,(Rn)**	**Store four fractional words with scaling, rounding and saturation, from four data registers. Scaling disabled if SR.SM is set. Only uses (Rn) addressing mode.**

	`(EA)..(EA+1) = SRS_40_to_16_leg(Da)`
	`(EA+2)..(EA+3) = SRS_40_to_16_leg(Db)`
	`(EA+4)..(EA+5) = SRS_40_to_16_leg(Dc)`
	`(EA+6)..(EA+7) = SRS_40_to_16_leg(Dd)`	

7	**ST.SRS.LEG.F Da,(Rn)**	**Store a fractional word with scaling, rounding and saturation. Scaling disabled if SR.SM is set. Only uses (Rn) addressing mode.**
	`(EA)..(EA+1) = SRS_40_to_16_leg(Da)`	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Da::Dh	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$ $0 \le n \le 56$
Da:Db	$D_n:D_{n+1}$ $0 \le n \le 62$
Da:Db:Dc:Dd	$D_n:D_{n+1}:D_{n+2}:D_{n+3}$ $0 \le n \le 60$
Rk	R0-R7,R16-R23
Rn	R0-R31

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2, 3, 4	For post-modify addressing modes
SP, ESP, TSP, DSP	1, 2, 3, 4	For SP-relative addressing modes
SR.RM	All	
SR.SCM	All	
SR.SM	5, 6, 7	
SR2.SPSEL	1, 2, 3, 4	For SP-relative addressing modes

Affected By Instructions

Field	Relevant Variants
SR.S	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits	5, 6, 7
	32-bits 48-bits	1, 2, 3, 4
Pipeline behavior	agu_MOVE_ST	5, 6, 7
	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_IMM agu_MOVE_ST_PRECALC_R	1, 2, 4
	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_R agu_MOVE_ST_PRECALC_IMM	3

ST.SRS.nL→ D Store 32-bit Fractional Long Words From Data Registers With Scale and Saturation (40 bits to 32) (LSU)

General Description

Store 32-bit fractional long words from data registers to memory after scaling, rounding and saturating the data from 40-bit sources prior to issuing them to memory. Data in the source registers is not changed. SRS processing includes the following steps:

1. Scaling according to the setting of SR.SCM. Scaling could be seen as shift up or down of the of bits in the source register from which the data to store is taken from, relative to the normal portion M expected for longs. In case of scaling up long data, zeros are appended to the right.
2. Signed saturation is performed if the bits to the left of the data window, including the extension, are not a clean sign extension of the most significant bit in the data window. SR.S is optionally set according to the output, see the bit description in SR.

Note that for SRS stores of longs, no rounding is performed. For more information on the functionality of SRS see the DALU chapter in this manual. SIMD Variants store two, four, eight or 16 longs. Legacy variants (LEG) have in addition an implicit dependency on SR.SM - if it is set, then no scaling is done. All Standard addressing modes supported, except for LEG variants.

Flag Options

Flag	Position	Description
SRS	flg1	Scaling, rounding and saturation of data during a memory store (may vary per instruction)
LEG	flg2	Legacy instruction

Instruction Variants

#	Syntax	Description
1	**ST.SRS.16L Da::Dp,(SAM)**	**Store 16 fractional longs with scaling and saturation, from 16 data registers. Special grouping limitations apply. All standard addressing modes are supported.**

```
(EA)..(EA+3) = SS_40_to_32(Da)
(EA+4)..(EA+7) = SS_40_to_32(Db)
...
(EA+60)..(EA+63) = SS_40_to_32(Dp)
```

Table continues on the next page...

#	Syntax	Description
2	**ST.SRS.2L Da:Db,(SAM)**	**Store two fractional longs with scaling and saturation, from two data registers. All standard addressing modes are supported.**
	(EA)..(EA+3) = SS_40_to_32(Da) (EA+4)..(EA+7) = SS_40_to_32(Db)	
3	**ST.SRS.4L Da:Db:Dc:Dd,(SAM)**	**Store four fractional longs with scaling and saturation, from four data registers. All standard addressing modes are supported.**
	(EA)..(EA+3) = SS_40_to_32(Da) (EA+4)..(EA+7) = SS_40_to_32(Db) (EA+8)..(EA+11) = SS_40_to_32(Dc) (EA+12)..(EA+15) = SS_40_to_32(Dd)	
4	**ST.SRS.8L Da::Dh,(SAM)**	**Store eight fractional longs with scaling and saturation, from eight data registers. All standard addressing modes are supported.**
	(EA)..(EA+3) = SS_40_to_32(Da) (EA+4)..(EA+7) = SS_40_to_32(Db) ... (EA+28)..(EA+31) = SS_40_to_32(Dh)	
5	**ST.SRS.L Da,(SAM)**	**Store a fractional long with scaling and saturation. All standard addressing modes are supported.**
	(EA)..(EA+3) = SS_40_to_32(Da)	
6	**ST.SRS.LEG.2L Da:Db,(Rn)**	**Store two fractional longs with scaling and saturation, from two data registers. Scaling disabled if SR.SM is set. Only uses (Rn) addressing mode.**
	(EA)..(EA+3) = SS_40_to_32_leg(Da) (EA+4)..(EA+7) = SS_40_to_32_leg(Db)	
7	**ST.SRS.LEG.L Da,(Rn)**	**Store a fractional long with scaling and saturation. Scaling disabled if SR.SM is set. Only uses (Rn) addressing mode.**
	(EA)..(EA+3) = SS_40_to_32_leg(Da)	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Da::Dh	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$ $0 \le n \le 56$
Da::Dp	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}:D_{n+8}:D_{n+9}:D_{n+10}:D_{n+11}:D_{n+12}:D_{n+13}:D_{n+14}:D_{n+15}$ $0 \le n \le 48$
Da:Db	$D_n:D_{n+1}$ $0 \le n \le 62$
Da:Db:Dc:Dd	$D_n:D_{n+1}:D_{n+2}:D_{n+3}$ $0 \le n \le 60$
Rk	R0-R7,R16-R23
Rn	R0-R31

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2, 3, 4, 5	For post-modify addressing modes
SP, ESP, TSP, DSP	1, 2, 3, 4, 5	For SP-relative addressing modes
SR.SCM	All	
SR.SM	6, 7	
SR2.SPSEL	1, 2, 3, 4, 5	For SP-relative addressing modes

Affected By Instructions

Field	Relevant Variants
SR.S	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits	6, 7
	32-bits 48-bits	1, 2, 3, 4, 5
Pipeline behavior	agu_MOVE_ST	6, 7
	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_R agu_MOVE_ST_PRECALC_IMM	1, 2, 3, 4, 5

ST.nB→ D Store Bytes From Data Registers to Memory (LSU)

General Description

Store a byte or several bytes from data registers to memory. Each byte is taken from the least significant byte of one source register. SIMD variants store 2, 4 or 8 bytes from a respective number of consecutive registers.

Instruction Variants

#	Syntax	Description
1	**ST.2B Da:Db,(SAM)**	**Store 2 bytes from two registers. All Standard addressing modes supported.**
	`(EA)..(EA+1) = {Da.LL, Db.LL}`	
2	**ST.4B Da:Db:Dc:Dd,(SAM)**	**Store 4 bytes from four registers. All Standard addressing modes supported.**
	`(EA)..(EA+3) = {Da.LL, Db.LL, Dc.LL, Dd.LL}`	
3	**ST.8B Da::Dh,(SAM)**	**Store 8 bytes from eight registers. All Standard addressing modes supported.**
	`(EA)..(EA+7) = {({Da.LL, Db.LL, Dc.LL, Dd.LL}), ({De.LL, Df.LL, Dg.LL, Dh.LL})}`	
4	**ST.B Da,(SAM)**	**Store one byte. All Standard addressing modes supported.**
	`(EA) = Da.LL`	
5	**ST.B Da,(SP-u9)**	**store one byte, using SP-relative addressing with a short offset.**
	`((SP - u9))..((SP - u9)+0) = Da.LL`	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Da::Dh	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$ $0 \le n \le 56$
Da:Db	$D_n:D_{n+1}$ $0 \le n \le 62$
Da:Db:Dc:Dd	$D_n:D_{n+1}:D_{n+2}:D_{n+3}$ $0 \le n \le 60$
Rk	R0-R7,R16-R23
Rn	R0-R31
u9	$0 \le u9 < 2^9$

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2, 3, 4	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3
Encoding length	32-bits	5
	32-bits 48-bits	1, 2, 3, 4
Pipeline behavior	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_IMM agu_MOVE_ST_PRECALC_R	1, 4
	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_R agu_MOVE_ST_PRECALC_IMM	2, 3
	agu_MOVE_ST_PRECALC_IMM	5

ST.nB→ R Store Bytes From Address Registers to Memory (LSU)

General Description

Store a byte or several bytes from address registers to memory. Each byte is taken from the least significant byte of one source register. SIMD variants store 2 or 4 bytes from a respective number of consecutive registers.

Instruction Variants

#	Syntax	Description
1	**ST.2B Ra:Rb,(SAM)**	**Store 2 bytes from two registers. All Standard addressing modes supported.**
	`(EA)..(EA+1) = {Ra[7:0], Rb[7:0]}`	
2	**ST.4B Ra:Rb:Rc:Rd,(SAM)**	**Store 4 bytes from four registers. All Standard addressing modes supported.**
	`(EA)..(EA+3) = {Ra[7:0], Rb[7:0], Rc[7:0], Rd[7:0]}`	
3	**ST.B Ra,(SAM)**	**Store one byte. All Standard addressing modes supported.**
	`(EA) = Ra[7:0]`	
4	**ST.B Ra,(SP-u9)**	**store one byte, using SP-relative addressing with a short offset.**
	`((SP - u9))..((SP - u9)+0) = Ra[7:0]`	

Explicit Operands

Operand	Permitted Values
Ra	R0-R31
Ra:Rb	$R_n:R_{n+1}$ $0 \le n \le 30$
Ra:Rb:Rc:Rd	$R_n:R_{n+1}:R_{n+2}:R_{n+3}$ $0 \le n \le 28$ Explicit list of all allowed combinations at Table C-22
Rk	R0-R7,R16-R23
Rn	R0-R31
u9	$0 \le u9 < 2^9$

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2, 3	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2
Encoding length	32-bits	4
	32-bits 48-bits	1, 2, 3
Pipeline behavior	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_IMM agu_MOVE_ST_PRECALC_R	1, 3
	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_R agu_MOVE_ST_PRECALC_IMM	2
	agu_MOVE_ST_PRECALC_IMM	4

ST.nBF→ D — Store Fractional Bytes From Data Registers to Memory — (LSU)

General Description

Store one or several fractional byte from data registers to memory. Each byte is taken from byte HH (the highest byte not including the extension) of one source register. SIMD variants store 2, 4 or 8 bytes from a respective number of consecutive registers.

Instruction Variants

#	Syntax	Description
1	**ST.2BF Da:Db,(SAM)**	**Store 2 fractional bytes from two registers. All Standard addressing modes supported.**
	`(EA)..(EA+1) =   {Da.HH, Db.HH}`	
2	**ST.4BF Da:Db:Dc:Dd,(SAM)**	**Store 4 fractional bytes from four registers. All Standard addressing modes supported.**
	`(EA)..(EA+3) =   {Da.HH, Db.HH, Dc.HH, Dd.HH}`	
3	**ST.8BF Da::Dh,(SAM)**	**Store 8 fractional bytes from eight registers. All Standard addressing modes supported.**
	`(EA)..(EA+7) =   {({Da.HH, Db.HH, Dc.HH, Dd.HH}), ({De.HH, Df.HH, Dg.HH, Dh.HH})}`	
4	**ST.BF Da,(SAM)**	**Store one fractional byte. All Standard addressing modes supported.**
	`(EA) =   Da.HH`	
5	**ST.BF Da,(SP-u9)**	**Store one fractional byte, using SP-relative addressing with a short offset.**
	`((SP - u9))..((SP - u9)+0) =   Da.HH`	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Da::Dh	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$ $0 \le n \le 56$
Da:Db	$D_n:D_{n+1}$ $0 \le n \le 62$
Da:Db:Dc:Dd	$D_n:D_{n+1}:D_{n+2}:D_{n+3}$ $0 \le n \le 60$
Rk	R0-R7,R16-R23
Rn	R0-R31
u9	$0 \le u9 < 2^9$

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2, 3, 4	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3
Encoding length	32-bits	5
	32-bits 48-bits	1, 2, 3, 4
Pipeline behavior	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_IMM agu_MOVE_ST_PRECALC_R	1, 2, 4
	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_R agu_MOVE_ST_PRECALC_IMM	3
	agu_MOVE_ST_PRECALC_IMM	5

ST.nF→ D Store Fractional 16-bit Words From Data Registers to Memory (LSU)

General Description

Store one or several fractional 16-bit words from data registers to memory. Each word is taken from portion H of the source register (the highest word not including the extension). SIMD variants store 2, 4 or 8 words from a respective number of consecutive registers

Instruction Variants

#	Syntax	Description
1	**ST.2F Da:Db,(SAM)**	**Store 2 fractional words from two registers. All Standard addressing modes supported.**
	`(EA)..(EA+3) = {Da.H, Db.H}`	
2	**ST.4F Da:Db:Dc:Dd,(SAM)**	**Store 4 fractional words from four registers. All Standard addressing modes supported.**
	`(EA)..(EA+7) = {Da.H, Db.H, Dc.H, Dd.H}`	
3	**ST.8F Da::Dh,(SAM)**	**Store 8 fractional words from eight registers. All Standard addressing modes supported.**
	`(EA)..(EA+15) = {({Da.H, Db.H, Dc.H, Dd.H}), ({De.H, Df.H, Dg.H, Dh.H})}`	
4	**ST.F Da,(SAM)**	**Store one fractional word. All Standard addressing modes supported.**
	`(EA)..(EA+1) = Da.H`	
5	**ST.F Da,(SP-u9_1)**	**Store one fractional word, using SP-relative addressing with a short, word-aligned offset.**
	`((SP - u9_1))..((SP - u9_1)+1) = Da.H`	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Da::Dh	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$ $0 \le n \le 56$
Da:Db	$D_n:D_{n+1}$ $0 \le n \le 62$
Da:Db:Dc:Dd	$D_n:D_{n+1}:D_{n+2}:D_{n+3}$ $0 \le n \le 60$
Rk	R0-R7,R16-R23
Rn	R0-R31
u9_1	$0 \le u9\_1 < 2^{10}$; u9_1 & 0x1 == 0

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2, 3, 4	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3
Encoding length	32-bits	5
	32-bits 48-bits	1, 2, 3, 4
Pipeline behavior	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_IMM agu_MOVE_ST_PRECALC_R	2, 4
	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_R agu_MOVE_ST_PRECALC_IMM	1, 3
	agu_MOVE_ST_PRECALC_IMM	5

ST.nL→ D — Store 32-bit Long Words From Data Registers to Memory (LSU)

General Description

Store one or several 32-bit long words from data registers to memory. The data is taken from the lower 32-bits of the source registers. SIMD variants store 2, 4, 8 or 16 longs from a respective number of consecutive registers. One 2L variant allows to store from a non-consecutive pair.

Instruction Variants

#	Syntax	Description
1	**ST.16L Da::Dp,(SAM)**	**Store 16 longs from 16 consecutive registers. All Standard addressing modes supported. Special grouping limitations apply.**
	`(EA)..(EA+63) = {({Da.M, Db.M, Dc.M, Dd.M}), ({De.M, Df.M, Dg.M, Dh.M}), ({Di.M, Dj.M, Dk.M, Dl.M}), ({Dm.M, Dn.M, Do.M, Dp.M})}`	
2	**ST.2L Da:Db,(SAM)**	**Store 2 longs from two consecutive registers. All Standard addressing modes supported.**
	`(EA)..(EA+7) = ({Da, Db})`	
3	**ST.2L Da:Dc,(SAM)**	**Store 2 longs from two non-consecutive registers. All Standard addressing modes supported.**
	`(EA)..(EA+7) = ({Da, Dc})`	
4	**ST.4L Da:Db:Dc:Dd,(SAM)**	**Store 4 longs from four consecutive registers. All Standard addressing modes supported.**
	`(EA)..(EA+15) = ({Da.M, Db.M, Dc.M, Dd.M})`	
5	**ST.8L Da::Dh,(SAM)**	**Store 8 longs from eight consecutive registers. All Standard addressing modes supported.**
	`(EA)..(EA+31) = {({Da.M, Db.M, Dc.M, Dd.M}), ({De.M, Df.M, Dg.M, Dh.M})}`	
6	**ST.L Da,(SAM)**	**Store one long. All Standard addressing modes supported.**
	`(EA)..(EA+3) = Da.M`	
7	**ST.L Da,(SP-u9_2)**	**Store one long, using SP-relative addressing with a short, long-aligned offset.**
	`((SP - u9_2))..((SP - u9_2)+3) = Da.M`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da::Dh	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$	$0 \le n \le 56$

Table continues on the next page...

Operand	Permitted Values
Da::Dp	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}:D_{n+8}:D_{n+9}:D_{n+10}:D_{n+11}:D_{n+12}:D_{n+13}:D_{n+14}:D_{n+15}$ $0 \le n \le 48$
Da:Db	$D_n:D_{n+1}$ $0 \le n \le 62$
Da:Db:Dc:Dd	$D_n:D_{n+1}:D_{n+2}:D_{n+3}$ $0 \le n \le 60$
Da:Dc	$D_n:D_{n+2}$ $0 \le n \le 61$ Explicit list of all allowed combinations at Table C-12
Rk	R0-R7,R16-R23
Rn	R0-R31
u9_2	$0 \le u9\_2 < 2^{11}$; u9_2 & 0x3 == 0

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2, 3, 4, 5, 6	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 3, 4, 5
Encoding length	32-bits	2, 7
	32-bits 48-bits	1, 3, 4, 5, 6
Pipeline behavior	agu_MOVE_ST_POSTINC	2
	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_IMM agu_MOVE_ST_PRECALC_R	1, 3, 4
	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_R agu_MOVE_ST_PRECALC_IMM	5, 6
	agu_MOVE_ST_PRECALC_IMM	7

ST.nL→ R   Store 32-bit Long Words From Address Registers to Memory   (LSU)

General Description

Store one or several 32-bit long words from address registers to memory. SIMD variants store 2 or 4 longs from a respective number of consecutive registers.

Instruction Variants

#	Syntax	Description
1	**ST.2L Ra:Rb,(SAM)**	**Store 2 longs from two registers. All Standard addressing modes supported.**
	`(EA)..(EA+7) = {Ra, Rb}`	
2	**ST.4L Ra:Rb:Rc:Rd,(SAM)**	**Store 4 longs from four registers. All Standard addressing modes supported.**
	`(EA)..(EA+15) = ({Ra, Rb, Rc, Rd})`	
3	**ST.L Ra,(SAM)**	**Store one long. All Standard addressing modes supported.**
	`(EA)..(EA+3) = Ra`	
4	**ST.L Ra,(SP-u9_2)**	**Store one long, using SP-relative addressing with a short, long-aligned offset.**
	`((SP - u9_2))..((SP - u9_2)+3) = Ra`	

Explicit Operands

Operand	Permitted Values
Ra	R0-R31
Ra:Rb	R_n:R_{n+1} $0 \le n \le 30$
Ra:Rb:Rc:Rd	R_n:R_{n+1}:R_{n+2}:R_{n+3} $0 \le n \le 28$ Explicit list of all allowed combinations at Table C-22
Rk	R0-R7,R16-R23
Rn	R0-R31
u9_2	0 ≤ u9_2 < 2^{11}; u9_2 & 0x3 == 0

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2, 3	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	2
Encoding length	32-bits	4
	32-bits 48-bits	1, 2, 3
Pipeline behavior	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_IMM agu_MOVE_ST_PRECALC_R	1, 2, 3
	agu_MOVE_ST_PRECALC_IMM	4

ST.nW→ D   Store 16-bit Words From Data Registers to Memory   (LSU)

General Description

Store one or several 16-bit words from data registers to memory. Each word is taken from the lower portion (L) of the source register. SIMD variants store 2 or 4 words from a respective number of consecutive registers.

Instruction Variants

#	Syntax	Description
1	**ST.2W Da:Db,(SAM)**	**Store 2 words from two registers. All Standard addressing modes supported.**
	`(EA)..(EA+3) =  {Da.L, Db.L}`	
2	**ST.4W Da:Db:Dc:Dd,(SAM)**	**Store 4 words from four registers. All Standard addressing modes supported.**
	`(EA)..(EA+7) =  {Da.L, Db.L, Dc.L, Dd.L}`	
3	**ST.8W Da::Dh,(SAM)**	**Store 8 words from eight registers. All Standard addressing modes supported.**
	`(EA)..(EA+15) =  {({Da.L, Db.L, Dc.L, Dd.L}), ({De.L, Df.L, Dg.L, Dh.L})}`	
4	**ST.W Da,(SAM)**	**Store one word. All Standard addressing modes supported.**
	`(EA)..(EA+1) =  Da.L`	
5	**ST.W Da,(SP-u9_1)**	**Store one word, using SP-relative addressing with a short, word-aligned offset.**
	`((SP - u9_1))..((SP - u9_1)+1) =  Da.L`	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Da::Dh	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$ $0 \le n \le 56$
Da:Db	$D_n:D_{n+1}$ $0 \le n \le 62$
Da:Db:Dc:Dd	$D_n:D_{n+1}:D_{n+2}:D_{n+3}$ $0 \le n \le 60$
Rk	R0-R7,R16-R23
Rn	R0-R31
u9_1	$0 \le u9\_1 < 2^{10}$; u9_1 & 0x1 == 0

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2, 3, 4	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3
Encoding length	32-bits	5
	32-bits 48-bits	1, 2, 3, 4
Pipeline behavior	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_IMM agu_MOVE_ST_PRECALC_R	1, 2, 4
	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_R agu_MOVE_ST_PRECALC_IMM	3
	agu_MOVE_ST_PRECALC_IMM	5

ST.nW→ R Store 16-bit Words From Address Registers to Memory (LSU)

General Description

Store one or several 16-bit words from address registers to memory. Each word is taken from the lower portion (L) of the source register. SIMD variants store 2 or 4 words from a respective number of consecutive registers.

Instruction Variants

#	Syntax	Description
1	**ST.2W Ra:Rb,(SAM)**	**Store 2 words from two registers. All Standard addressing modes supported.**
	`(EA)..(EA+3) = {Ra.L, Rb.L}`	
2	**ST.4W Ra:Rb:Rc:Rd,(SAM)**	**Store 4 words from four registers. All Standard addressing modes supported.**
	`(EA)..(EA+7) = {Ra.L, Rb.L, Rc.L, Rd.L}`	
3	**ST.W Ra,(SAM)**	**Store one word. All Standard addressing modes supported.**
	`(EA)..(EA+1) = Ra.L`	
4	**ST.W Ra,(SP-u9_1)**	**Store one word, using SP-relative addressing with a short, word-aligned offset.**
	`((SP - u9_1))..((SP - u9_1)+1) = Ra.L`	

Explicit Operands

Operand	Permitted Values
Ra	R0-R31
Ra:Rb	$R_n:R_{n+1}$ $0 \le n \le 30$
Ra:Rb:Rc:Rd	$R_n:R_{n+1}:R_{n+2}:R_{n+3}$ $0 \le n \le 28$ Explicit list of all allowed combinations at Table C-22
Rk	R0-R7,R16-R23
Rn	R0-R31
u9_1	$0 \le u9\_1 < 2^{10}$; u9_1 & 0x1 == 0

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2, 3	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2
Encoding length	32-bits	4
	32-bits 48-bits	1, 2, 3
Pipeline behavior	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_IMM agu_MOVE_ST_PRECALC_R	1, 3
	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_R agu_MOVE_ST_PRECALC_IMM	2
	agu_MOVE_ST_PRECALC_IMM	4

ST.nX→ D Store 40-bit Values From Data Registers to Memory (LSU)

General Description

Store one or several 40-bit values from data registers to memory. Every 40-bit value is stored as a 64-bit access where bits above bit 39 are written with zeros. SIMD variants store 2, 4, 8 or 16 values from a respective number of registers.

Instruction Variants

#	Syntax	Description
1	**ST.2X Da:Db,(SAM)**	**Store 2 40-bit values from two registers. All Standard addressing modes supported.**
	`(EA)..(EA+15) = {(U64)Da, (U64)Db}`	
2	**ST.4X Da:Db:Dc:Dd,(SAM)**	**Store 4 40-bit values from four registers. All Standard addressing modes supported.**
	`(EA)..(EA+31) = {(U64)Da, (U64)Db, (U64)Dc, (U64)Dd}`	
3	**ST.8X Da::Dh,(SAM)**	**Store 8 40-bit values from eight registers. All Standard addressing modes supported. Special grouping limitations apply.**
	`(EA)..(EA+63) = {({0x000000, Da, 0x000000, Db, 0x000000, Dc, 0x000000, Dd}), ({0x000000, De, 0x000000, Df, 0x000000, Dg, 0x000000, Dh})}`	
4	**ST.X Da,(SAM)**	**Store one 40-bit value from a register. All Standard addressing modes supported.**
	`(EA)..(EA+7) = (U64)Da`	
5	**ST.X Da,(SP-u9_3)**	**Store one 40-bit value from a register, using SP-relative addressing with a short, 64-bit aligned offset**
	`((SP - u9_3))..((SP - u9_3)+7) = (U64)Da`	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Da::Dh	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$ $0 \le n \le 56$
Da:Db	$D_n:D_{n+1}$ $0 \le n \le 62$
Da:Db:Dc:Dd	$D_n:D_{n+1}:D_{n+2}:D_{n+3}$ $0 \le n \le 60$
Rk	R0-R7,R16-R23
Rn	R0-R31
u9_3	$0 \le u9\_3 < 2^{12}$; u9_3 & 0x7 == 0

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2, 3, 4	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	1, 2, 3
Encoding length	32-bits	5
	32-bits 48-bits	1, 2, 3, 4
Pipeline behavior	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_IMM agu_MOVE_ST_PRECALC_R	1, 2, 3
	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_R agu_MOVE_ST_PRECALC_IMM	4
	agu_MOVE_ST_PRECALC_IMM	5

ST2.SRS.nBF→ D Store Packed Fractional Bytes From Data Registers With Scale, Round and Saturation (20 bits to 8) (LSU)

General Description

Store one or more pairs of packed fractional bytes from data registers to memory after scaling, rounding and saturating the data from 20-bit portions prior to issuing them to memory. Each source register holds two packed 20-bit portions, which include 4 extension bits. The SRS operation attempts to best represent a 20-bit value in 8 bits, after an optional scaling. The bytes are extracted from each registers and stored contiguously. Data in the source registers is not changed. SRS processing includes the following steps:

1. Scaling according to the setting of SR.SCM. Scaling could be seen as shift up or down of the of bits in the source register from which the data to store is taken from, relative to bits 15:8 in the 20-bit portion, which is the non-scaled position of a fractional byte in a 20-bit portion.
2. Rounding according to SR.RM, which looks at the bits in the register to the right of the data window.
3. Signed saturation is performed if the bits to the left of the data window, including the extension, are not a clean sign extension of the most significant bit in the data window. SR.S is optionally set according to the output, see the bit description in SR.

For more information on the functionality of SRS see the DALU chapter in this manual. SIMD variants store two, four or eight fractional bytes taken from different registers. Variants allow storing of 2, 4, 8 or 16 bytes from one, 2, 4, or 8 registers, respectively. All Standard addressing modes supported.

Flag Options

Flag	Position	Description
SRS	flg1	Scaling, rounding and saturation of data during a memory store (may vary per instruction)

Instruction Variants

#	Syntax	Description
1	**ST2.SRS.16BF Da::Dh,(SAM)**	**Store sixteen fractional bytes with scaling, rounding and saturation, from eight data registers. All standard addressing modes are supported.**

```
(EA) = SRS_20_to_8(Da.WH)
(EA+1) = SRS_20_to_8(Da.WL)
...
(EA+14) = SRS_20_to_8(Dh.WH)
```

Table continues on the next page...

#	Syntax	Description
2	**ST2.SRS.2BF Da,(SAM)**	**Store two fractional bytes with scaling, rounding and saturation, from a data register. All standard addressing modes are supported.**

	`(EA) = SRS_20_to_8(Da.WH)`
	`(EA+1) = SRS_20_to_8(Da.WL)`	

3	**ST2.SRS.4BF Da:Db,(SAM)**	**Store four fractional bytes with scaling, rounding and saturation, from two data registers. All standard addressing modes are supported.**

	`(EA) = SRS_20_to_8(Da.WH)`
	`(EA+1) = SRS_20_to_8(Da.WL)`
	`(EA+2) = SRS_20_to_8(Db.WH)`
	`(EA+3) = SRS_20_to_8(Db.WL)`	

4	**ST2.SRS.8BF Da:Db:Dc:Dd,(SAM)**	**Store eight fractional bytes with scaling, rounding and saturation, from four data registers. All standard addressing modes are supported.**

	`(EA) = SRS_20_to_8(Da.WH)`
	`(EA+1) = SRS_20_to_8(Da.WL)`
	`(EA+2) = SRS_20_to_8(Db.WH)`
	`...`
	`(EA+7) = SRS_20_to_8(Dd.WL)`	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Da::Dh	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$ $0 \le n \le 56$
Da:Db	$D_n:D_{n+1}$ $0 \le n \le 62$
Da:Db:Dc:Dd	$D_n:D_{n+1}:D_{n+2}:D_{n+3}$ $0 \le n \le 60$
Rk	R0-R7,R16-R23
Rn	R0-R31

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	All	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR.RM	All	
SR.SCM	All	
SR2.SPSEL	All	For SP-relative addressing modes

Affected By Instructions

Field	Relevant Variants
SR.S	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits 48-bits	All
Pipeline behavior	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_IMM agu_MOVE_ST_PRECALC_R	2, 3, 4
	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_R agu_MOVE_ST_PRECALC_IMM	1

ST2.SRS.nF→ D Store Packed Fractional Words From Data Registers With Scale, Round and Saturation (20 bits to 16) (LSU)

General Description

Store one or more pairs of packed fractional words from data registers to memory after scaling, rounding and saturating the data from 20-bit portions prior to issuing them to memory. Each source register holds two packed 20-bit portions, which include 4 extension bits. The SRS operation attempts to best represent a 20-bit value in 16 bits, after optional scaling. The words are extracted from each register and stored contiguously. Data in the source registers is not changed. SRS processing includes the following steps:

1. Scaling according to the setting of SR.SCM. Scaling could be seen as shift up or down of the of bits in the source register from which the data to store is taken from, relative to bits 15:0 in the 20-bit portion, which is the non-scaled position of a fractional word in a 20-bit portion.
2. Rounding according to SR.RM, which looks at the bits in the register to the right of the data window, if scaled down.
3. Signed saturation is performed if the bits to the left of the data window, including the extension, are not a clean sign extension of the most significant bit in the data window. SR.S is optionally set according to the output, see the bit description in SR.

For more information on the functionality of SRS see the DALU chapter in this manual. SIMD variants store two, four or eight fractional bytes taken from different registers. Variants allow storing of 2, 4, 8, 16 or 32 words from one, 2, 4, 8 or 16 registers, respectively. All Standard addressing modes supported.

The 16F variant using Da..Dh-mod8p supoprts many 8-register combinations as follows: The index of each D register is calculated as a concatenation of two 3-bit values: {val1[2:0],val2[2:0]}. Val1 is calculated as (n[5:3] + pos), where n is the index of Da (first register in the octet), and pos is the serial position of D in the octet starting with zero; val2 is bits [2:0] of (n + pos). For example, in octet D2:D11:D20:D29:D38:D47:D48:D57, n=2; the index of D48 (pos = 6) is calculated by: val1 = (2[5:3] + 6) = 0b110, val2 = (2+6)[2:0] = 0b000. {val1:val2} = 0b110000 = 48.

Flag Options

Flag	Position	Description
SRS	flg1	Scaling, rounding and saturation of data during a memory store (may vary per instruction)

Instruction Variants

#	Syntax	Description
1	**ST2.SRS.16F Da::Dh-mod8,(SAM)**	**Store 16 fractional words with scaling, rounding and saturation, from a rotated register octet. The source octet is an aligned group of 8 registers (D0-D7, D8-D15 etc.) which can be rotated in all modulo-8 consecutive combinations. All standard addressing modes are supported.**

Table continues on the next page...

#	Syntax	Description
	(EA)..(EA+1) = SRS_20_to_16(Da.WH) (EA+2)..(EA+3) = SRS_20_to_16(Da.WL) ... (EA+28)..(EA+29) = SRS_20_to_16(Dh.WH) (EA+30)..(EA+31) = SRS_20_to_16(Dh.WL)	
2	**ST2.SRS.16F Da::Dh-mod8p,(SAM)**	**Store 16 fractional words with scaling, rounding and saturation, from eight registers. Each register has a different modulo-8 index. For a detailed expalnation of the allowed combinations, see the end of the general description at the top. All standard addressing modes are supported.**
	(EA)..(EA+1) = SRS_20_to_16(Da.WH) (EA+2)..(EA+3) = SRS_20_to_16(Da.WL) ... (EA+28)..(EA+29) = SRS_20_to_16(Dh.WH) (EA+30)..(EA+31) = SRS_20_to_16(Dh.WL)	
3	**ST2.SRS.16F Da::Dh,(SAM)**	**Store 16 fractional words with scaling, rounding and saturation, from eight consecutive registers. All standard addressing modes are supported.**
	(EA)..(EA+1) = SRS_20_to_16(Da.WH) (EA+2)..(EA+3) = SRS_20_to_16(Da.WL) ... (EA+28)..(EA+29) = SRS_20_to_16(Dh.WH) (EA+30)..(EA+31) = SRS_20_to_16(Dh.WL)	
4	**ST2.SRS.2F Da,(SAM)**	**Store 2 fractional words with scaling, rounding and saturation, from one register. All standard addressing modes are supported.**
	(EA)..(EA+1) = SRS_20_to_16(Da.WH) (EA+2)..(EA+3) = SRS_20_to_16(Da.WL)	
5	**ST2.SRS.32F Da::Dp,(SAM)**	**Store 32 fractional words with scaling, rounding and saturation, from sixteen consecutive registers. Special grouping limitations apply. All standard addressing modes are supported.**
	(EA)..(EA+1) = SRS_20_to_16(Da.WH) (EA+2)..(EA+3) = SRS_20_to_16(Da.WL) ... (EA+60)..(EA+61) = SRS_20_to_16(Dp.WH) (EA+62)..(EA+63) = SRS_20_to_16(Dp.WL)	
6	**ST2.SRS.4F Da:Db,(SAM)**	**Store 4 fractional words with scaling, rounding and saturation, from two consecutive registers. All standard addressing modes are supported.**
	(EA)..(EA+1) = SRS_20_to_16(Da.WH) (EA+2)..(EA+3) = SRS_20_to_16(Da.WL) (EA+4)..(EA+5) = SRS_20_to_16(Db.WH) (EA+6)..(EA+7) = SRS_20_to_16(Db.WL)	
7	**ST2.SRS.8F Da:Db:Dc:Dd,(SAM)**	**Store 8 fractional words with scaling, rounding and saturation, from four consecutive registers. All standard addressing modes are supported.**

Table continues on the next page...

#	Syntax	Description
	(EA)..(EA+1) = SRS_20_to_16(Da.WH) (EA+2)..(EA+3) = SRS_20_to_16(Da.WL) ... (EA+12)..(EA+13) = SRS_20_to_16(Dd.WH) (EA+14)..(EA+15) = SRS_20_to_16(Dd.WL)	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Da::Dh	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$ $0 \le n \le 56$
Da::Dh-mod8	$D_n:D_{n\&0x38+(n+1)\%8}:D_{n\&0x38+(n+2)\%8}:D_{n\&0x38+(n+3)\%8}:D_{n\&0x38+(n+4)\%8}:D_{n\&0x38+(n+5)\%8}:D_{n\&0x38+(n+6)\%8}:D_{n\&0x38+(n+7)\%8}$ $0 \le n \le 63$ Explicit list of all allowed combinations at Table C-7
Da::Dh-mod8p	$D_n:D_{(n\&0x38+8)\%64+(n+1)\%8}:D_{(n\&0x38+16)\%64+(n+2)\%8}:D_{(n\&0x38+24)\%64+(n+3)\%8}:D_{(n\&0x38+32)\%64+(n+4)\%8}:D_{(n\&0x38+40)\%64+(n+5)\%8}:D_{(n\&0x38+48)\%64+(n+6)\%8}:D_{(n\&0x38+54)\%64+(n+7)\%8}$ $0 \le n \le 63$ Explicit list of all allowed combinations at Table C-8
Da::Dp	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}:D_{n+8}:D_{n+9}:D_{n+10}:D_{n+11}:D_{n+12}:D_{n+13}:D_{n+14}:D_{n+15}$ $0 \le n \le 48$
Da:Db	$D_n:D_{n+1}$ $0 \le n \le 62$
Da:Db:Dc:Dd	$D_n:D_{n+1}:D_{n+2}:D_{n+3}$ $0 \le n \le 60$
Rk	R0-R7,R16-R23
Rn	R0-R31

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	All	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR.RM	All	
SR.SCM	All	
SR2.SPSEL	All	For SP-relative addressing modes

Affected By Instructions

Field	Relevant Variants
SR.S	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	3, 4, 5, 6, 7
Encoding length	32-bits	1, 2

Table continues on the next page...

Attribute	Value	Relevant Variant #
	32-bits 48-bits	3, 4, 5, 6, 7
Pipeline behavior	agu_MOVE_ST_POSTINC	1, 2
	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_IMM agu_MOVE_ST_PRECALC_R	4, 6, 7
	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_R agu_MOVE_ST_PRECALC_IMM	3, 5

ST2.nB→ D — Store Packed Bytes From Data Registers to Memory — (LSU)

General Description

Store one or several packed byte pairs from data registers to memory. Each source register holds two packed bytes, that occupy positions LL and HL. The bytes are extracted from each registers and stored contiguously. SIMD variants store 4, 8 or 16 bytes from a respective number of consecutive registers.

Instruction Variants

#	Syntax	Description
1	**ST2.16B Da::Dh,(SAM)**	**Store 16 bytes from eight registers. All Standard addressing modes supported.**
	(EA)..(EA+15) = {({Da.HL, Da.LL, Db.HL, Db.LL}), ({Dc.HL, Dc.LL, Dd.HL, Dd.LL}), ({De.HL, De.LL, Df.HL, Df.LL}), ({Dg.HL, Dg.LL, Dh.HL, Dh.LL})}	
2	**ST2.2B Da,(SAM)**	**Store two bytes from one register. All Standard addressing modes supported.**
	(EA)..(EA+1) = ({Da.HL, Da.LL})	
3	**ST2.4B Da:Db,(SAM)**	**Store 4 bytes from two registers. All Standard addressing modes supported.**
	(EA)..(EA+3) = ({(Da.HL), (Da.LL), (Db.HL), (Db.LL)})	
4	**ST2.8B Da:Db:Dc:Dd,(SAM)**	**Store 8 bytes from four registers. All Standard addressing modes supported.**
	(EA)..(EA+7) = ({({(Da.HL), (Da.LL), (Db.HL), (Db.LL)}), ({(Dc.HL), (Dc.LL), (Dd.HL), (Dd.LL)})})	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Da::Dh	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$ $0 \le n \le 56$
Da:Db	$D_n:D_{n+1}$ $0 \le n \le 62$
Da:Db:Dc:Dd	$D_n:D_{n+1}:D_{n+2}:D_{n+3}$ $0 \le n \le 60$
Rk	R0-R7,R16-R23
Rn	R0-R31

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	All	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits 48-bits	All
Pipeline behavior	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_IMM agu_MOVE_ST_PRECALC_R	All

ST2.nBF→ D Store Packed Fractional Bytes From Data Registers to Memory (LSU)

General Description

Store one or several packed fractional byte pairs from data registers to memory. Each source register holds two packed bytes, that occupy positions LH and HH. The bytes are extracted from each registers and stored contiguously. SIMD variants store 4, 8 or 16 bytes from a respective number of consecutive registers.

Instruction Variants

#	Syntax	Description
1	**ST2.16BF Da::Dh,(SAM)**	**Store 16 fractional bytes from eight registers. All Standard addressing modes supported.**
	`(EA)..(EA+15) = {({Da.HH, Da.LH, Db.HH, Db.LH}), ({Dc.HH, Dc.LH, Dd.HH, Dd.LH}), ({De.HH, De.LH, Df.HH, Df.LH}), ({Dg.HH, Dg.LH, Dh.HH, Dh.LH})}`	
2	**ST2.2BF Da,(SAM)**	**Store two fractional bytes from one register. All Standard addressing modes supported.**
	`(EA)..(EA+1) = ({Da.HH, Da.LH})`	
3	**ST2.4BF Da:Db,(SAM)**	**Store 4 fractional bytes from two registers. All Standard addressing modes supported.**
	`(EA)..(EA+3) = ({Da.HH, Da.LH, Db.HH, Db.LH})`	
4	**ST2.8BF Da:Db:Dc:Dd,(SAM)**	**Store 8 fractional bytes from four registers. All Standard addressing modes supported.**
	`(EA)..(EA+7) = ({({Da.HH, Da.LH, Db.HH, Db.LH}), ({Dc.HH, Dc.LH, Dd.HH, Dd.LH})})`	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Da::Dh	$D_n:D_{n+1}:D_{n+2}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$ $0 \le n \le 56$
Da:Db	$D_n:D_{n+1}$ $0 \le n \le 62$
Da:Db:Dc:Dd	$D_n:D_{n+1}:D_{n+2}:D_{n+3}$ $0 \le n \le 60$
Rk	R0-R7,R16-R23
Rn	R0-R31

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	All	For post-modify addressing modes
SP, ESP, TSP, DSP	All	For SP-relative addressing modes
SR2.SPSEL	All	For SP-relative addressing modes

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits 48-bits	All
Pipeline behavior	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_IMM agu_MOVE_ST_PRECALC_R	1, 3, 4
	agu_MOVE_ST_POSTINC agu_MOVE_ST_PRECALC_R agu_MOVE_ST_PRECALC_IMM	2

SUB.LL Subtract 64-bit Values (DALU)

General Description

Subtracts two 64-bit values (integer or fraction) into a 64-bit result. A 64-bit value is held in a register pair. The extension of the sources is ignored, and the extension of the result is cleared.

In case of overflow result is wrapped arround.

Instruction Variants

#	Syntax	Description
1	**SUB.LL Da:Db,Dc:Dd,Dm:Dn**	**64-bit subtract**

	`Dm = (U40)(({Dc.M, Dd.M}) - ({Da.M, Db.M}))[63:32]`
	`Dn = (U40)(({Dc.M, Dd.M}) - ({Da.M, Db.M}))[31:0]`	

Explicit Operands

Operand	Permitted Values	
Da:Db	$D_n:D_{n+1}$	$0 \leq n \leq 62$
Dc:Dd	$D_n:D_{n+1}$	$0 \leq n \leq 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \leq n \leq 62$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU_Y	All

SUB.nT Subtract 20-bit Values (DALU)

General Description

The instruction normally subtracts 20-bit values that are stored in the WH or WL portions of data registers. In Dual 16-bit saturation mode (SR.SM2 set), some variants subtract two 16-bit portions and saturate the result into 16-bits.

Variants that generate one result (SUB.T) are affected by SR.W20 and SR.SM2. In Dual 20-bit mode and non-saturating 16-bit mode (SM2 clear), two 20-bit portions (WH or WL) from the two source registers are subtracted and one 20-bit value written to a 20-bit portion in the destination. The other portion is not changed.

In Dual 16-bit saturation mode (SM2 set, W20 clear), only 16-bit portions from the H or L portions of the source registers are subtracted and the result is saturated to 16-bits. The result updates the H or L portion of the destination, and the other portion is not changed. the extension is cleared.

The SIMD2 version (SUB.2T) is not affected by SR and performs two 20-bit subtraction in parallel.

Note that when SM2 is clear, all variants can be used to subtract 16-bit values, if the extensions are ignored.

Instruction Variants

#	Syntax	Description
1	**SUB.2T Da,Db,Dn**	**SIMD2 20-bit subtraction (two values in two registers)**

	`Dn.WH = Db.WH - Da.WH`
	`Dn.WL = Db.WL - Da.WL`	

2	**SUB.T Da.h,Dn.h**	**High source and high destination 20-bit/16-bit subtraction**

	`if (SR.SM2==1 && SR.W20==0)`
	`  Dn.H = srSAT16(Dn.H - Da.H); Dn.L = Dn.L; Dn.E = 0`
	`else`
	`  Dn.WH = (Dn.WH - Da.WH); Dn.WL = Dn.WL`	

3	**SUB.T Da.h,Dn.l**	**High source and low destination 20-bit/16-bit subtraction**

	`if (SR.SM2==1 && SR.W20==0)`
	`  Dn.L = srSAT16(Dn.L - Da.H); Dn.H = Dn.H; Dn.E = 0`
	`else`
	`  Dn.WL = (Dn.WL- Da.WH); Dn.WH = Dn.WH`	

4	**SUB.T Da.l,Dn.h**	**Low source and high destination 20-bit/16-bit subtraction**

	`if (SR.SM2==1 && SR.W20==0)`
	`  Dn.H = srSAT16(Dn.H - Da.L); Dn.L = Dn.L; Dn.E = 0`
	`else`
	`  Dn.WH = (Dn.WH - Da.WL); Dn.WL = Dn.WL`	

5	**SUB.T Da.l,Dn.l**	**Low source and low destination 20-bit/16-bit subtraction**

Table continues on the next page...

#	Syntax	Description

	`if (SR.SM2==1 && SR.W20==0)`
	`  Dn.L = srSAT16(Dn.L - Da.L); Dn.H = Dn.H; Dn.E = 0`
	`else`
	`  Dn.WL = (Dn.WL- Da.WL); Dn.WH = Dn.WH`	

Explicit Operands

Operand	Permitted Values
Da	`D0-D63`
Db	`D0-D63`
Dn	`D0-D63`

Affect Instructions

Field	Relevant Variants	Comments
SR.SM2	2, 3, 4, 5	
SR.W20	2, 3, 4, 5	

Affected By Instructions

Field	Relevant Variants
SR.SAT	2, 3, 4, 5

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

SUB.nX Subtract 40-bit Values (DALU)

General Description

Subtracts two 40-bit values (integer or fraction) into 40-bit result. If the first source operand is an immediate value, it right aligns and zero-extends the immediate value to 40 bits before subtracting.
SUB.X perform no saturation on the subtraction result.
SUB.2X is an SIMD2 version that subtracts two pairs of D registers.
SUB.S.X saturate the subtraction result to 32-bit value represented in 40-bit.
SUB.LEG.X perform no saturation on the subtraction result if SM bit in the SR is cleared (default) and saturate the subtraction result to 32-bit value represented in 40-bit if SM bit in the SR is set (legacy behavior).

Flag Options

Flag	Position	Description
LEG	flg2	Legacy instruction
S	flg2	Saturated result

Instruction Variants

#	Syntax	Description
1	**SUB.2X Da:Db,Dc:Dd,Dm:Dn**	**SIMD2 subtraction of two register pairs**

	`Dm = Dc - Da`
	`Dn = Db - Da`
	`Alias, encoded as: SOD.SSII.2X Da:Db,Dc:Dd,Dm:Dn`	

2	**SUB.X Da,Db,Dn**	**Subtract two registers**
	`Dn = (Db - Da)[39:0]`	
3	**SUB.X #u5,Da,Dn**	**Subtract immediate and a register**
	`Dn = (Da - (U40)u5)[39:0]`	
4	**SUB.LEG.X Da,Db,Dn**	**Subtract two registers, saturate according to mode (legacy behavior)**
	`Dn = srSAT32((Db - Da))`	
5	**SUB.LEG.X #u5,Da,Dn**	**Subtract immediate and a register, saturate according to mode (legacy behavior)**
	`Dn = srSAT32((Da - (U40)u5))`	
6	**SUB.S.X Da,Db,Dn**	**Subtract two registers and saturate**
	`Dn = SAT32((Db - Da))`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Db	D0-D63	
Dc:Dd	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	
u5	$0 \le u5 < 2^5$	

Affect Instructions

Field	Relevant Variants	Comments
SR.SM	4, 5	

Affected By Instructions

Field	Relevant Variants
SR.SAT	4, 5, 6

Instruction Attributes

Attribute	Value	Relevant Variant #
Alias		1
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

SUBA Subtract 32-bit Values (LSU,IPU)

General Description

Subtracts two 32-bit values into a 32-bit result. If the first source operand is an immediate value, it is aligned to the right and sign-extended to 32 bits before subtraction.

In non-linear variants (without the LIN flag), the modifier mode specified in the MCTL register may affect the operation. For SUBA of two registers, the MCTL mode is determined according to Rb (if one of R0-R7). For SUBA with an immediate value, the MCTL mode is determined according to Ra (if one of R0-R7).

Flag Options

Flag	Position	Description
LIN	flg2	Linear (Unlike Modulo), no MCTL impact

Instruction Variants

#	Syntax	Operation	Description
1	SUBA Ra,Rb,Rn	`Rn = MCTL_calc (Rb - Ra)`	Subtract two registers according to MCTL
2	SUBA #u5,Ra,Rn	`Rn = MCTL_calc (Ra - u5)`	Subtract signed 16-bit immediate and a register according to MCTL
3	SUBA.LIN Ra,Rb,Rn	`Rn = Rb - Ra`	Subtract two registers using linear arithmetic, not affected by MCTL
4	SUBA.LIN #u5,Ra,Rn	`Rn = Ra - (U32)u5`	Subtract unsigned 5-bit immediate and a register using linear arithmetic, not affected by MCTL

Explicit Operands

Operand	Permitted Values
Ra	R0-R31
Rb	R0-R31
Rn	R0-R31
u5	$0 \le u5 < 2^5$

Affect Instructions

Field	Relevant Variants	Comments
B, M, MCTL	1, 2	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Execution unit	IPU	3, 4
	LSU	All
Pipeline behavior	agu_AAU	3, 4
	agu_AAU_MCTL	1, 2

SUBC.L Subtract with Carry 40-bit Values and 32-bit carry out (DALU)

General Description

Subtracts two 40-bit values (integer or fraction) with or without the carry bit (SR.C) into a 40-bit result, and updates the carry bit if the result overflows 32 bits. If the first source operand is an immediate value, it is right-aligned and sign-extended into 40 bits before subtraction.

Flag Options

Flag	Position	Description
RW	flg1	Carry Read and Write
WO	flg1	Carry Write Only
LEG	flg2	Legacy instruction

Instruction Variants

#	Syntax	Description
1	**SUBC.WO.L Da,Db,Dn**	**Subtract two registers, update carry**

	`Dn = (Db - Da)[39:0]`
	`SR.C = Borrow (32)`	

2	**SUBC.WO.L #u5,Da,Dn**	**Subtract immediate and a register, update carry**

	`Dn = (Da - u5)[39:0]`
	`SR.C = Borrow (32)`	

3	**SUBC.RW.LEG.L Da,Db,Dn**	**Subtract two registers and carry, update carry**

	`SR.C = Borrow (32)`
	`Dn = (Db - (Da + (U40)SR.C))[39:0]`	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Db	D0-D63
Dn	D0-D63
u5	$0 \leq u5 < 2^5$

Affect Instructions

Field	Relevant Variants	Comments
SR.C	3	

Affected By Instructions

Field	Relevant Variants
SR.C	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

SUBC.X Subtract with Carry 40-bit Values (DALU)

General Description

Subtracts two 40-bit values (integer or fraction) with or without the carry bit (SR.C) into a 40-bit result, and updates the carry bit if the result overflows 40 bits. If the first source operand is an immediate value, it is right-aligned and zero-extended into 40 bits before subtraction.

Flag Options

Flag	Position	Description
RO	flg1	Carry Read Only
WO	flg1	Carry Write Only
LEG	flg2	Legacy instruction

Instruction Variants

#	Syntax	Description
1	**SUBC.RO.X Da,Db,Dn**	**Subtract two registers and carry in**
	`Dn = (Db - (Da + ({0x0000000000, SR.C})))[39:0]`	
2	**SUBC.WO.X Da,Db,Dn**	**Subtract two registers and carry in, update carry**

	`Dn = (Db - Da)[39:0]`
	`SR.C = Borrow (39)`	

3	**SUBC.WO.X #u5,Da,Dn**	**Subtract immediate and a register, update carry**

	`Dn = (Da - (U40)u5)[39:0]`
	`SR.C = Borrow (39)`	

4	**SUBC.WO.LEG.X Da,Db,Dn**	**Subtract two registers and carry in, saturate according to SR, update carry**

	`SR.C = Borrow (39)`
	`Dn = srSAT32((Db - Da))`	

5	**SUBC.WO.LEG.X #u5,Da,Dn**	**Subtract immediate and a register, saturate according to SR, update carry**

	`SR.C = Borrow (39)`
	`Dn = srSAT32((Da - u5))`	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Db	D0-D63
Dn	D0-D63
u5	$0 \le u5 < 2^5$

Affect Instructions

Field	Relevant Variants	Comments
SR.C	1	
SR.SM	4, 5	

Affected By Instructions

Field	Relevant Variants
SR.C	2, 3, 4, 5
SR.SAT	4, 5

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

SWAP.nB Swap Bytes (DALU)

General Description

Swap position of four/eight bytes in one/two registers. The DX flavor swaps the order between byte couples (first with second, third with fourth). The SX flavor swaps the four bytes in a mirror image (first with fourth, second with third).

The extension of the source registers is ignored, the extension of the destination registers is cleared.

Flag Options

Flag	Position	Description
DX	flg1	Double swap the bytes in each high/low 16-bit portions
SX	flg1	"Star" byte swap - HH to LL, HL to LH etc.

Instruction Variants

#	Syntax	Description
1	**SWAP.DX.4B Da,Dn**	**SIMD2 Swap byte couples**

	`Dn.HL = Da.HH`
	`Dn.HH = Da.HL`
	`Dn.LL = Da.LH`
	`Dn.LH = Da.LL`
	`Dn.E = 0`	

2	**SWAP.DX.8B Da,Db,Dm:Dn**	**SIMD4 Swap byte couples**

	`Dm.HH = Da.HL   ; Dm.HL = Da.HH  ;   Dm.LH =  Da.LL   ;  Dm.LL = Da.LH`
	`Dn.HH = Db.HL ; Dn.HL = Db.HH ; Dn.LH = Db.LL ; Dn.LL = Db.LH`
	`Dm.E = 0, Dn.E = 0`	

3	**SWAP.SX.4B Da,Dn**	**Swap byte quadruplet**

	`Dn.LL = Da.HH`
	`Dn.LH = Da.HL`
	`Dn.HL = Da.LH`
	`Dn.HH = Da.LL`
	`Dn.E = 0`	

4	**SWAP.SX.8B Da,Db,Dm:Dn**	**SIMD2 Swap byte quadruplet**

	`Dm.HH = Da.LL   ; Dm.HL = Da.LH  ;   Dm.LH =  Da.HL   ;  Dm.LL = Da.HH`
	`Dn.HH = Db.LL ; Dn.HL = Db.LH ; Dn.LH = Db.HL ; Dn.LL = Db.HH`
	`Dm.E = 0, Dn.E = 0`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Db	D0-D63	
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

SWAP.nW Swap 16-bit Words (DALU)

General Description

Swap position of 16-bit portions from the source D registers.
Variants with two sources strictly updates 16-bit portions in the destinations: the SX variant swaps the order between word couples (first with second and third with fourth); the IXI variant swaps the order between the second and the third words. The extension in the source registers is ignored, the extension of the destination registers is cleared. The variant with the single input register swaps the two 16-bit portions from the soure register and sign-extends them in 20-bit portions of the destination.

Flag Options

Flag	Position	Description
IXI	flg1	Transfer portion 3, swap (cross) portions 2 and 1, transfer portion 0
SX	flg1	Cross-swap the two H/L portions in each register

Instruction Variants

#	Syntax	Description
1	**SWAP.W Da,Dn**	**Swaps the two 16-bit portions from the source register, sign extending each to 20 bit portions in the destination.**

	`Dn.WH = (S20)Da.L`
	`Dn.WL = (S20)Da.H`
	`Alias, encoded as: PACK.W.2W Da.L,Da.H,Dn`	

2	**SWAP.IXI.4W Da,Db,Dm:Dn**	**Swap 2nd and 3rd 16-bit words**

	`Dm.H = Da.H`
	`Dm.L = Db.H`
	`Dn.H = Da.L`
	`Dn.L = Db.L`
	`Dm.E = 0, Dn.E = 0`	

3	**SWAP.SX.4W Da,Db,Dm:Dn**	**Swap 16-bit word quadruplet**

	`Dm.H = Da.L`
	`Dm.L = Da.H`
	`Dn.H = Db.L`
	`Dn.L = Db.H`
	`Dm.E = 0, Dn.E = 0`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Db	D0-D63	
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Alias		1
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	1, 2
	dalu_LBM	3

SWB Generate a Debug Mode Request for Software Breakpoints (PCU)

General Description

If the DTU host partition is enabled, executing it will cause the core to enter debug mode. If not enabled, execution SWB will cause an illegal exception. SWB is typically used by the host debugger when it inserts software breakpoints, by replacing a VLES with this instruction, and restoring it once debug mode was entered. Hence if SWB remained in the code after the debug session it is a mistake and the illegal exception helps uncovering it. The SWB instruction is serviced as break-before-make, meaning that the core is halted with the PC and the architectural state as it was before the execution of SWB, which allows the debugger to replace it back with the original VLES. Note that the debugger cannot single-step over this VLES. For more information on debug mode, see the debug chapters in both the core and subsystem RM.

Instruction Variants

#	Syntax	Operation	Description
1	SWB		If the DTU host partition is enabled, enters the core into debug mode. If not, causes an illegal exception.

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	16-bits	All
No predication		All
Pipeline behavior	pcu_control	All

SXT.nT Sign Extend to 20-bit Word (DALU)

General Description

Sign extends a byte or a 16-bit word to a 20-bit word. Actually it casts a signed byte or signed 16-bit word into signed 20-bit word. The extension and the high bytes (HH, lH) in SXT.B are ignored.

Flag Options

Flag	Position	Description
B	flg1	Byte
W	flg1	Word (16 bits)

Instruction Variants

#	Syntax	Operation	Description

1	SXT.B.2T Da,Dn	Dn.WL = (S20)Da.LL
		Dn.WH = (S20)Da.HL
			SIMD2 byte to 20-bit Sign Extend

2	SXT.B.4T Da:Db,Dm:Dn	Dm.WL = (S20)Da.LL
		Dm.WH = (S20)Da.HL
		Dn.WL = (S20)Db.LL
		Dn.WH = (S20)Db.HL
			SIMD4 byte to 20-bit Sign Extend

3	SXT.W.2T Da,Dn	Dn.WL = (S20)Da.L
		Dn.WH = (S20)Da.H
			SIMD2 16-bit to 20-bit Sign Extend

4	SXT.W.4T Da:Db,Dm:Dn	Dm.WL = (S20)Da.L
		Dm.WH = (S20)Da.H
		Dn.WL = (S20)Db.L
		Dn.WH = (S20)Db.H
			SIMD4 16-bit to 20-bit Sign Extend

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	D_n:D_{n+1}	$0 \le n \le 62$
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

SXT.nX Sign Extend to 40-bit Word (DALU)

General Description

Sign extends a byte, a 16-bit word or a 32-bit word to a 20-bit word. Actually it casts a signed byte, signed 16-bit word or signed 32-bit word into a signed 40-bit word. The high word for SXT.W, the three high bytes for SXT.B and the extension are ignored.

Flag Options

Flag	Position	Description
B	flg1	Byte
L	flg1	32 bit input
W	flg1	Word (16 bits)

Instruction Variants

#	Syntax	Operation	Description

1	SXT.B.2X Da:Db,Dm:Dn	`Dm = (S40)Da.LL`
		`Dn = (S40)Db.LL`
			SIMD2 byte to 40-bit Sign Extend

2	SXT.B.X Da,Dn	`Dn = (S40)Da.LL`	Byte to 40-bit Sign Extend

3	SXT.L.2X Da:Db,Dm:Dn	`Dm = (S40)Da.M`
		`Dn = (S40)Db.M`
			SIMD2 32-bit to 40-bit Sign Extend

4	SXT.L.X Da,Dn	`Dn = (S40){Da.H, Da.L}`	32-bit to 40-bit Sign Extend

5	SXT.W.2X Da:Db,Dm:Dn	`Dm = (S40)Da.L`
		`Dn = (S40)Db.L`
			SIMD2 16-bit to 40-bit Sign Extend

6	SXT.W.X Da,Dn	`Dn = (S40)Da.L`	16-bit to 40-bit Sign Extend

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \leq n \leq 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \leq n \leq 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

SXTA Sign Extend to 32-bit Word (LSU,IPU)

General Description

Sign extends an integer byte or a 16-bit word to a 32-bit word. Actually it casts a signed integer byte or a signed integer 16-bit word into a signed integer 32-bit word.

Flag Options

Flag	Position	Description
B	flg1	Byte
W	flg1	Word (16 bits)

Instruction Variants

#	Syntax	Operation	Description
1	SXTA.B.L Ra,Rn	`Rn = (S32)Ra[7:0]`	Byte to 32-bit Sign Extend
2	SXTA.W.L Ra,Rn	`Rn = (S32)Ra.L`	16-bit to 32-bit Sign Extend

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Execution unit	IPU	All
	LSU	All
Pipeline behavior	agu_AAU_LOGIC	All

SYNC Insert a Pipeline Synchronization Delay (PCU)

General Description

This instruction inserts pipeline delays, which are required in certain situations by the programming rules to help ensure the correct semantic order of instructions. The SYNC.D inserts delays between reads and writes of several special core registers. The SYNC.B variants insert a pipe delay that is required in certain cases of data memory accesses, and are usually grouped with instructions that interact with memory. The SYNC.P instruction flushes the full pipeline and is required in certain cases involving changes in program memory. By themselves, the SYNC instructions do not change architectural state. In many cases, the SYNC instructions are added implicitly as part of meta-instructions which encode as a combination of two or more constituent instructions. For more information on these instructions and the conditions that require them, see the Pipeline and the Programming Rules chapters.

Flag Options

Flag	Position	Description
B	flg1	Before
D	flg1	Data accesses
P	flg1	Program access
DL	flg2	Dedicated for Destructive Loads

Instruction Variants

#	Syntax	Operation	Description
1	SYNC.B		Add a pipe stall with the preceding VLES. Used mainly with memory accesses.
2	SYNC.D		Add a pipe stall with the following VLES.
3	SYNC.P	`Alias, encoded as: CLRIC #0`	Adds a program pipe stall before the next VLES by flushing the pipeline
4	SYNC.B.DL		Add a pipe stall with the preceding VLES. Used mainly as grouped with destructive loads, see a description in the Address Generation chapter.

Instruction Attributes

Attribute	Value	Relevant Variant #
Alias		3
Encoding length	32-bits	All
Helper instruction		4
No predication		All
Pipeline behavior	pcu_control	All

TFR.T Transfer a 20-bit portion (DALU)

General Description

The instruction transfers a 16-bit portion from a source register, sign extending it in a 20-bit portion in the destination register. The other 20-bit portion in the destination is not changed. Variants allow to transfer either the high or low portion of the source register to either the high or low portion of the destination. The instruction is an alias to a PACK instruction.

Instruction Variants

#	Syntax	Description
1	**TFR.T Da.H,Dn.H**	**Transfer a high register portion to a high register portion.**

	`Dn.WH = (S20)Da.H`
	`Alias, encoded as: PACK.T.2T Da.H,Dn.L,Dn`	

2	**TFR.T Da.H,Dn.L**	**Transfer a high register portion to a low register portion.**

	`Dn.WL = (S20)Da.H`
	`Alias, encoded as: PACK.T.2T Dn.H,Da.H,Dn`	

3	**TFR.T Da.L,Dn.H**	**Transfer a low register portion to a high register portion.**

	`Dn.WH = (S20)Da.L`
	`Alias, encoded as: PACK.T.2T Da.L,Dn.L,Dn`	

4	**TFR.T Da.L,Dn.L**	**Transfer a low register portion to a low register portion.**

	`Dn.WL = (S20)Da.L`
	`Alias, encoded as: PACK.T.2T Dn.H,Da.L,Dn`	

Explicit Operands

Operand	Permitted Values
Da	D0-D63
Dn	D0-D63

Instruction Attributes

Attribute	Value	Relevant Variant #
Alias		All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

TFR.nX Transfer a register or immediate to a data register (DALU)

General Description

Transfer a data register or immediate value to a data register. These are DALU instructions that are not subject to the limitations of programming rule A.11.

Instruction Variants

#	Syntax	Operation	Description

1	TFR.2X Da,Db,Dm:Dn	`Dm = Da`
		`Dn = Db`
			Transfer two full data registers (40-bits) to a data register pair. The source registers are independently specified.

2	TFR.2X #s32,Dm:Dn	`Dm = (S40)s32`
		`Dn = (S40)s32`
			Duplicate a signed 32-bit immediate value into a data register pair. The value is sign-extended in the destinations.

3	TFR.X Da,Dn	`Dn = Da`	Transfer a full data register (40-bits) into a data register
4	TFR.X #s32,Dn	`Dn = (S40)s32`	Transfer a signed 32-bit value to a data register, sign-extending it in the destination.

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Db	D0-D63	
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	
s32	$-2^{31} \le s32 < 2^{31}$	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	1, 3
	64-bits	2, 4
Pipeline behavior	dalu_DAU	All

TFRA Transfer a value to or from a special register (IPU,LSU)

General Description

Transfer a value between an address register and a control register, another R register, or SP; or vice versa.

Instruction Variants

#	Syntax	Description
1	**TFRA C8,Rn**	**Transfer a control register to an address register**
	`Rn = C8`	
2	**TFRA Ra,C7**	**Transfer an address register to a control register**
	`C7 = Ra`	
3	**TFRA Ra,Rn**	**Transfers one R register to another**

	`Rn = Ra`
	`Alias, encoded as: ANDA Ra,Ra,Rn`	

4	**TFRA Ra,SP**	**Transfer the active SP, as defined in SR2.SPSEL, to an address register**
	`SP = Ra`	
5	**TFRA SP,Rn**	**Transfer an address register to the active SP, as defined in SR2.SPSEL**
	`Rn = SP`	

Explicit Operands

Operand	Permitted Values				
C7	CESRA0,	CESRA1,	EIDR,	GCR,	LC0,
	LC1,	LC2,	LC3,	LR0.EIDR,	LR0.PC,
	LR0.SR,	LR0.SR2,	LR1.EIDR,	LR1.PC,	LR1.SR,
	LR1.SR2,	LR2.EIDR,	LR2.PC,	LR2.SR,	LR2.SR2,
	MCTL,	MOCR,	PROCID,	SR,	SR2,
	TID,	TMDAT,	TMTAG		
C8	CESRA0,	CESRA1,	COREREV,	EIDR,	GCR,
	LC0,	LC1,	LC2,	LC3,	LR0.EIDR,
	LR0.PC,	LR0.SR,	LR0.SR2,	LR1.EIDR,	LR1.PC,
	LR1.SR,	LR1.SR2,	LR2.EIDR,	LR2.PC,	LR2.SR,
	LR2.SR2,	MCTL,	MOCR,	PC,	PROCID,
	SR,	SR2,	THRDN,	TID,	TMTAG
Ra	R0-R31				
Rn	R0-R31				

Affect Instructions

Field	Relevant Variants	Comments
SP, ESP, TSP, DSP	5	
SR2.SPSEL	4, 5	

Affected By Instructions

Field	Relevant Variants
SP, ESP, TSP, DSP	4

Instruction Attributes

Attribute	Value	Relevant Variant #
Alias		3
Encoding length	32-bits	All
Execution unit	IPU	1, 2, 3
	LSU	3, 4, 5
Pipeline behavior	agu_AAU	1
	agu_AAU_CTRL_REG	2
	agu_AAU_LOGIC	3
	agu_AAU_SP	4, 5

TFRA.IMM Transfer an immediate value to an address register (LSU,IPU)

General Description

Transfer a signed immediate value to an R register.

Instruction Variants

#	Syntax	Operation	Description
1	TFRA.L #s32,Rn	`Rn = s32`	Transfer a 32-bit signed immediate to an R register. The assembler also accepts an unsigned value for the immediate, hence the wider range in the permitted range in the operand table below.
2	TFRA.L #s9,Rn	`Rn = (S32)s9`	Transfer a 9-bit signed immediate to an R register. The value is sign-extended in the destination.

Explicit Operands

Operand	Permitted Values
Rn	R0-R31
s32	$-2^{31} \le s32 < 2^{32}$
s9	$-2^{8} \le s9 < 2^{8}$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	2
	48-bits	1
Execution unit	IPU	All
	LSU	All
Pipeline behavior	agu_MOVE_REG_INIT_IMM	All

TFRCA Transfer to or from an alternate SP (LSU)

General Description

Transfer a value from an R register to an alternate SP, or vice versa. The identity of the alternate SP is determined by the setting of SR2.ASPSEL. The ability to be able to access a different stack pointer without changing the identity of the active stack pointer allows the OS to manage a different stack (such as the stack of the task just interrupted) while keeping the ability to access its own stack.

Instruction Variants

#	Syntax	Operation	Description
1	TFRCA Ra,SP	`Alternative SP = Rn`	Transfer an address register to an alternate SP
2	TFRCA SP,Rn	`Rn = Alternative SP`	Transfer an alternate SP to an address register

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`
Rn	`R0-R31`

Affect Instructions

Field	Relevant Variants	Comments
ESP, TSP, DSP	2	
SR2.ASPSEL	All	

Affected By Instructions

Field	Relevant Variants
ESP, TSP, DSP	1

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Pipeline behavior	agu_AAU_SP	All

TRAP Execute a Software Exception (PCU)

General Description

Causes a precise exception, which is executed after the VLES that includes the TRAP. The instruction is typically used to implement OS calls. The immediate operand is sampled in EIDR and allows the caller to pass some information on the nature of the service it requests. The two flavors of TRAP (0 and 1) use two different registers to get the address of the service routine, thus allowing the OS to differentiate between services that require lower processing overhear from the rest.

The following steps occur before servicing the exception: The return PC, SR, SR2 and EIDR are saved in LR0; SR, SR2 and EIDR are modified according to the settings for TRAP, including updating the immediate field u10 in EIDR. Execution continues to the address configured either in CESRA0 or CESRA1. The return PC that is sampled in LR0 is that of the following VLES, so RTE will restore execution to the correct location for continued execution without a need to modify the return PC.

Flag Options

Flag	Position	Description
0	flg1	Variant index
1	flg1	Variant index

Instruction Variants

#	Syntax	Description
1	**TRAP.0 #u10**	**Execute a Software exception from CESRA0**

	`PC = CESRA0;`
	`LR0 = {return PC, SR, SR2, EIDR}`
	`EIDR = {0x0,u10}`	

2	**TRAP.1 #u10**	**Execute a Software exception from CESRA1**

	`PC = CESRA1;`
	`LR0 = {return PC, SR, SR2, EIDR}`
	`EIDR = {0x8,u10}`	

Explicit Operands

Operand	Permitted Values
u10	$0 \leq u10 < 2^{10}$

Affect Instructions

Field	Relevant Variants	Comments
CESRA0	All	
CESRA1	All	
EIDR	All	

Table continues on the next page...

Field	Relevant Variants	Comments
SR	All	
SR2	All	

Affected By Instructions

Field	Relevant Variants
EIDR.EID	All
EIDR.ETP	All
LR	All
SR.C	All
SR.S	All
SR.[P0,P1,P2,P3,P4,P5]	All
SR2.IPM	All
SR2.SPSEL	All

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Must be alone in VLES		All
Pipeline behavior	pcu_COF	All

TSTBM.L Bit Mask Test a 32-bit Operand in a D Register (DALU)

General Description

Tests the specified bits in the 32-bit portion of Da.M. The bits to test are specified in a 32-bit immediate operand which serves as a mask - only bits for which the mask is set are tested. The (.S) variant will return a true result if all the selected bits were set, and false otherwise. Similarly, the (.C) variant will return a true result if all the selected bits were clear. Variants with a single predicate destination will capture the test result in it. Variants which update a predicate pair will update the first predicate with the test result and the inverse result in the second predicate. As with other compare and test instructions, in some cases several instructions updating the same predicates may be grouped together in the same VLES, in which case the test result is the logical OR of the individual tests. The instruction may be in itself predicated by IF.Pn or IF.Pn.EC conditions. For more information on the semantics of predicate updating instructions, see the Program Control chapter.

Flag Options

Flag	Position	Description
C	flg1	Clear
S	flg1	Set

Instruction Variants

#	Syntax	Description
1	**TSTBM.C.L #u32,Da,Pm:Pn**	**Bit mask test for clear the selected bits in Da.M, updating a predicate pair**

	`if ((Da.M & u32) == 0x00000000) {`
	`        Pm = 1`
	`        Pn = 0`
	`} else {`
	`        Pm = 0`
	`        Pn = 1`
	`}`	

2	**TSTBM.C.L #u32,Da,Pn**	**Bit mask test for clear the selected bits in Da.M, updating one predicate**

	`if ((Da.M & u32) == 0x00000000) {`
	`        Pn = 1`
	`} else {`
	`        Pn = 0`
	`}`	

3	**TSTBM.S.L #u32,Da,Pm:Pn**	**Bit mask test for set the selected bits in Da.M, updating a predicate pair**

	`if ((~ (Da.M) & u32) == 0x00000000) {`
	`        Pm = 1`
	`        Pn = 0`
	`} else {`
	`        Pm = 0`
	`        Pn = 1`
	`}`	

Table continues on the next page...

#	Syntax	Description
4	**TSTBM.S.L #u32,Da,Pn**	**Bit mask test for set the selected bits in Da.M, updating one predicate**

```
if ((~ (Da.M) & u32) == 0x00000000) {
        Pn = 1
} else {
        Pn = 0
}
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Pm:Pn	$P_n:P_{n+1}$	$0 \le n \le 4$
Pn	P0-P5	
u32	$0 \le u32 < 2^{32}$	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	64-bits	All
IF.EC		All
Pipeline behavior	dalu_DAU_Pbit	All

TSTBM.nX Bit Mask Test a 40-bit Operand in a D Register (DALU)

General Description

Tests the specified bits in Db. The bits to test are selected in Da which serves as a mask - only bits in Db for which the respective bit in Da are set are tested. The (.S) variant will return a true result if all the selected bits were set, and false otherwise. Similarly, the (.C) variant will return a true result if all the selected bits were clear. Variants with a single predicate destination will capture the test result in it. Variants which update a predicate pair will update the first predicate with the test result and the inverse result in the second predicate. As with other compare and test instructions, in some cases several instructions updating the same predicates may be grouped together in the same VLES, in which case the test result is the logical OR of the individual tests. The instruction may be in itself predicated by IF.Pn or IF.Pn.EC conditions. For more information on the semantics of predicate updating instructions, see the Program Control chapter.

Flag Options

Flag	Position	Description
C	flg1	Clear
S	flg1	Set

Instruction Variants

#	Syntax	Description
1	**TSTBM.C.X Da,Db,Pm:Pn**	**Bit mask test for clear the selected bits in Db, updating a predicate pair**

```
if ((Da & Db) == 0x00000000) {
        Pm = 1
        Pn = 0
} else {
        Pm = 0
        Pn = 1
}
```

2	**TSTBM.C.X Da,Db,Pn**	**Bit mask test for clear the selected bits in Db, updating one predicate**

```
if ((Da & Db) == 0x00000000) {
        Pn = 1
} else {
        Pn = 0
}
```

3	**TSTBM.S.X Da,Db,Pm:Pn**	**Bit mask test for set the selected bits in Db, updating a predicate pair**

```
if ((Da & ~ Db) == 0x00000000) {
        Pm = 1
        Pn = 0
} else {
        Pm = 0
        Pn = 1
}
```

Table continues on the next page...

#	Syntax	Description
4	**TSTBM.S.X Da,Db,Pn**	**Bit mask test for set the selected bits in Db, updating one predicate**

```
if ((Da & ~ Db) == 0x00000000) {
        Pn = 1
} else {
        Pn = 0
}
```

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Db	D0-D63	
Pm:Pn	$P_n:P_{n+1}$	$0 \le n \le 4$
Pn	P0-P5	

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
IF.EC		All
Pipeline behavior	dalu_DAU_Pbit	All

UNPACK.nF Unpack Fraction to 40-bit Word (DALU)

General Description

The instruction reads two packed 16-bit or 20-bit fractional values from one register, casts them to 40-bit fraction format and writes the result into two registers.

In the 16-bit input variants, the extension in the source register is ignored.

Note that this casting does not change the value nor the location of the decimal point in the destination register.

Flag Options

Flag	Position	Description
HL	flg1	High 16 bits from the first argument and Low 16 bits from the second argument
LH	flg1	Low 16 bits from the first argument and High 16 bits from the second argument
T	flg1	20 bits
W	flg1	Word (16 bits)

Instruction Variants

#	Syntax	Operation	Description

1	UNPACK.THL.2F Da,Dm:Dn	`Dm = (S40){Da.WH, 0x0000}`
		`Dn = (S40){Da.WL, 0x0000}`
			Unpack fraction 20-bit word to 40-bit word

2	UNPACK.TLH.2F Da,Dm:Dn	`Dm = (S40){Da.WL, 0x0000}`
		`Dn = (S40){Da.WH, 0x0000}`
			Unpack Fraction 20-bit word to 40-bit word in reversed order

3	UNPACK.WHL.2F Da,Dm:Dn	`Dm = (S40){Da.H, 0x0000}`
		`Dn = (S40){Da.L, 0x0000}`
			Unpack Fraction 16-bit word to 40-bit word

4	UNPACK.WLH.2F Da,Dm:Dn	`Dm = (S40){Da.L, 0x0000}`
		`Dn = (S40){Da.H, 0x0000}`
			Unpack Fraction 16-bit word to 40-bit word in reversed order

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

UNPACK.nT Unpack byte to 20-bit Word (DALU)

General Description

The instruction reads two/four packed integer/fraction bytes from a single register, casts them to integer/fraction 20-bit words and writes them into one or two registers.

The extension of the source register is ignored.

Flag Options

Flag	Position	Description
B	flg1	Byte
BF	flg1	Byte Fractional
BH	flg1	Byte High
BL	flg1	Byte Low
U	flg2	Unsigned

Instruction Variants

#	Syntax	Operation	Description

1	UNPACK.B.4T Da,Dm:Dn	`Dm.WH = (S20)Da.HH`
		`Dm.WL = (S20)Da.HL`
		`Dn.WH = (S20)Da.LH`
		`Dn.WL = (S20)Da.LL`
			Unpack four signed integer bytes into 20-bit signed integer words

2	UNPACK.BF.4T Da,Dm:Dn	`Dm.WH = (S20){Da.HH, 0x00}`
		`Dm.WL = (S20){Da.HL, 0x00}`
		`Dn.WH = (S20){Da.LH, 0x00}`
		`Dn.WL = (S20){Da.LL, 0x00}`
			Unpack four signed fraction bytes into 20-bit signed fraction words

3	UNPACK.B.U.4T Da,Dm:Dn	`Dm.WH = (U20)Da.HH`
		`Dm.WL = (U20)Da.HL`
		`Dn.WH = (U20)Da.LH`
		`Dn.WL = (U20)Da.LL`
			Unpack four unsigned integer bytes into 20-bit signed integer words

4	UNPACK.BH.U.2T Da,Dn	`Dn.WH = (U20)Da.HH`
		`Dn.WL = (U20)Da.HL`
			Unpack two unsigned bytes in high (fractional) location into 20-bit signed integer words

5	UNPACK.BL.U.2T Da,Dn	`Dn.WH = (U20)Da.LH`
		`Dn.WL = (U20)Da.LL`
			Unpack two unsigned bytes in low (integer) location into 20-bit signed integer words

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dm:Dn	D_n:D_{n+1}	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

UNPACK.nW Unpack byte to 16-bit Word (DALU)

General Description

The instruction reads four packed integer bytes from a single register, casts them to integer 16-bit words and writes into two registers.

The extension of the source register is ignored, the extension of the destination is cleared.

Flag Options

Flag	Position	Description
B	flg1	Byte

Instruction Variants

#	Syntax	Operation	Description

1	UNPACK.B.4W Da,Dm:Dn	`Dn.L = (S20)Da.LL`
		`Dn.H = (S20)Da.LH`
		`Dm.L = (S20)Da.HL`
		`Dm.H = (S20)Da.HH`
		`Dm.E = 0`
		`Dn.E = 0`
			Unpack byte to 16-bit word

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

UNPACK.nX Unpack Integer to 40-bit Word (DALU)

General Description

The instruction reads two packed 16-bit or 20-bit integer values from one register, casts them to a 40-bit integer and writes the result into two registers.

In the 16-bit input variant, the extension of the source register is ignored.

Flag Options

Flag	Position	Description
HL	flg1	High 16 bits from the first argument and Low 16 bits from the second argument
LH	flg1	Low 16 bits from the first argument and High 16 bits from the second argument
T	flg1	20 bits
W	flg1	Word (16 bits)

Instruction Variants

#	Syntax	Operation	Description

1	UNPACK.THL.2X Da,Dm:Dn	`Dm = (S40)Da.WH`
		`Dn = (S40)Da.WL`
			Unpack integer 20-bit word to 40-bit word

2	UNPACK.TLH.2X Da,Dm:Dn	`Dm = (S40)Da.WL`
		`Dn = (S40)Da.WH`
			Unpack integer 20-bit word to 40-bit word in reversed order

3	UNPACK.WHL.2X Da,Dm:Dn	`Dm = (S40)Da.H`
		`Dn = (S40)Da.L`
			Unpack integer 16-bit word to 40-bit word

4	UNPACK.WLH.2X Da,Dm:Dn	`Dm = (S40)Da.L`
		`Dn = (S40)Da.H`
			Unpack integer 16-bit word to 40-bit word in reversed order

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Dm:Dn	$D_n:D_{n+1}$	$0 \leq n \leq 62$

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

VTRACE Viterbi Trace Back (DALU)

General Description

Shifts the destination data register to the right 1 bit and copies a specified bit from the source data register to bit 31 of the destination register. The index of the selected bit is determined by the binary number in bits [(31 - m):(31 - m - n)] of the destination register before it is shifted. This binary number can be up to 5 bits; that is, it can be a decimal number in the range from 0 to 31. The extension of the destination register is cleared.

The combination of m and n determines the constraint length that is used in different convolution codes.

Instruction Variants

#	Syntax	Description
1	**VTRACE.L #0,#2,Da,Dn**	**Viterbi Trace Back**
	`Dn = {0x00, Da (vtrace_index (Dn.M, 0, 2), vtrace_index (Dn.M, 0, 2)), (Dn >> 1)[30:0]}`	
2	**VTRACE.L #0,#3,Da,Dn**	**Viterbi Trace Back**
	`Dn = {0x00, Da (vtrace_index (Dn.M, 0, 3), vtrace_index (Dn.M, 0, 3)), (Dn >> 1)[30:0]}`	
3	**VTRACE.L #0,#4,Da,Dn**	**Viterbi Trace Back**
	`Dn = {0x00, Da (vtrace_index (Dn.M, 0, 4), vtrace_index (Dn.M, 0, 4)), (Dn >> 1)[30:0]}`	
4	**VTRACE.L #1,#4,Da,Dn**	**Viterbi Trace Back**
	`Dn = {0x00, Da (vtrace_index (Dn.M, 1, 4), vtrace_index (Dn.M, 1, 4)), (Dn >> 1)[30:0]}`	
5	**VTRACE.L #2,#4,Da,Dn**	**Viterbi Trace Back**
	`Dn = {0x00, Da (vtrace_index (Dn.M, 2, 4), vtrace_index (Dn.M, 2, 4)), (Dn >> 1)[30:0]}`	
6	**VTRACE.L #3,#4,Da,Dn**	**Viterbi Trace Back**
	`Dn = {0x00, Da (vtrace_index (Dn.M, 3, 4), vtrace_index (Dn.M, 3, 4)), (Dn >> 1)[30:0]}`	
7	**VTRACE.L #4,#4,Da,Dn**	**Viterbi Trace Back**
	`Dn = {0x00, Da (vtrace_index (Dn.M, 4, 4), vtrace_index (Dn.M, 4, 4)), (Dn >> 1)[30:0]}`	
8	**VTRACE.L #5,#4,Da,Dn**	**Viterbi Trace Back**
	`Dn = {0x00, Da (vtrace_index (Dn.M, 5, 4), vtrace_index (Dn.M, 5, 4)), (Dn >> 1)[30:0]}`	

Explicit Operands

Operand	Permitted Values
Da	`D0-D63`
Dn	`D0-D63`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_DAU	All

ZXT.nT Zero Extend Byte to 20-bit Word (DALU)

General Description

Zero extends an unsigned integer byte to a 20-bit integer word. Actually it casts an unsigned integer byte into an unsigned (or signed) 20-bit integer word.

The extension and high bytes (HH and LH) of the source register are ignored.

Flag Options

Flag	Position	Description
B	flg1	Byte

Instruction Variants

#	Syntax	Operation	Description

1	ZXT.B.2T Da,Dn	`Dn.WL = (U20)Da.LL`
		`Dn.WH = (U20)Da.HL`
			SIMD2 byte to 20-bit Zero Extend

2	ZXT.B.4T Da:Db,Dm:Dn	`Dm.WL = (U20)Da.LL`
		`Dm.WH = (U20)Da.HL`
		`Dn.WL = (U20)Db.LL`
		`Dn.WH = (U20)Db.HL`
			SIMD4 byte to 20-bit Zero Extend

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$
Dm:Dn	$D_n:D_{n+1}$	$0 \le n \le 62$
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Architecture	SC3900FP only	All
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

ZXT.nX Zero Extend to 40-bit Word (DALU)

General Description

Zero extends a byte, 16-bit word or 32-bit word to a 40-bit word. Actually it casts an unsigned byte, unsigned 16-bit integer word or unsigned 32-bit integer word into an unsigned (or signed) 40-bit word.

The high word for ZXT.W, the high bytes (HH, HL, LH) for ZXT.B and the extension of the source register are ignored.

Flag Options

Flag	Position	Description
B	flg1	Byte
L	flg1	32 bit input
W	flg1	Word (16 bits)

Instruction Variants

#	Syntax	Description
1	**ZXT.B.2X Da:Db,Dm:Dn**	**SIMD2 byte to 40-bit zero extend**

	`Dm = (U40)Da.LL`
	`Dn = (U40)Db.LL`	

2	**ZXT.B.X Da,Dn**	**Byte to 40-bit zero extend**
	`Dn = (U40)Da.LL`	
3	**ZXT.L.2X Da:Db,Dm:Dn**	**SIMD2 32-bit to 40-bit zero extend**

	`Dm = (U40)Da.M`
	`Dn = (U40)Db.M`
	`Alias, encoded as: LSH.RGT.2L #0,Da:Db,Dm:Dn`	

4	**ZXT.L.X Da,Dn**	**32-bit to 40-bit zero extend**
	`Dn = {0x00, Da.M}`	
5	**ZXT.W.2X Da:Db,Dm:Dn**	**SIMD2 16-bit to 40-bit zero extend**

	`Dm = (U40)Da[15:0]`
	`Dn = (U40)Db[15:0]`	

6	**ZXT.W.X Da,Dn**	**16-bit to 40-bit zero extend**
	`Dn = (U40)Da.L`	

Explicit Operands

Operand	Permitted Values	
Da	D0-D63	
Da:Db	$D_n:D_{n+1}$	$0 \le n \le 62$

Table continues on the next page...

Operand	Permitted Values	
Dm:Dn	D_n:D_{n+1}	0 ≤ n ≤ 62
Dn	D0-D63	

Instruction Attributes

Attribute	Value	Relevant Variant #
Alias		3
Architecture	SC3900FP only	1, 5
Encoding length	32-bits in 64	All
Pipeline behavior	dalu_LBM	All

ZXTA Sign Extend to 32-bit Word (LSU,IPU)

General Description

Zero extends an integer byte or 16-bit word to a 32-bit word. Actually it casts an unsigned integer byte or unsigned integer 16-bit word into an unsigned (or signed) integer 32-bit word.

Flag Options

Flag	Position	Description
B	flg1	Byte
W	flg1	Word (16 bits)

Instruction Variants

#	Syntax	Operation	Description
1	ZXTA.B.L Ra,Rn	`Rn = (U32)Ra[7:0]`	Byte to 32-bit zero extend
2	ZXTA.W.L Ra,Rn	`Rn = (U32)Ra.L`	16-bit to 32-bit zero extend

Explicit Operands

Operand	Permitted Values
Ra	`R0-R31`
Rn	`R0-R31`

Instruction Attributes

Attribute	Value	Relevant Variant #
Encoding length	32-bits	All
Execution unit	IPU	All
	LSU	All
Pipeline behavior	agu_AAU_LOGIC	All

Special Instruction Fields

Da__DhMod8 - Da:Db:Dc:Dd:De:Df:Dg:Dh

Register allocation formula

$$D_{n}:D_{n\&0x38+(n+1)\%8}:D_{n\&0x38+(n+2)\%8}:D_{n\&0x38+(n+3)\%8}:D_{n\&0x38+(n+4)\%8}:D_{n\&0x38+(n+5)\%8}:D_{n\&0x38+(n+6)\%8}:D_{n\&0x38+(n+7)\%8}$$

$$0 \le n \le 63$$

Used by instruction groups

```
LD2.nF→ D;
ST2.SRS.nF→ D;
```

Permitted Values

#	Value	#	Value
0	D0:D1:D2:D3:D4:D5:D6:D7	1	D1:D2:D3:D4:D5:D6:D7:D0
2	D2:D3:D4:D5:D6:D7:D0:D1	3	D3:D4:D5:D6:D7:D0:D1:D2
4	D4:D5:D6:D7:D0:D1:D2:D3	5	D5:D6:D7:D0:D1:D2:D3:D4
6	D6:D7:D0:D1:D2:D3:D4:D5	7	D7:D0:D1:D2:D3:D4:D5:D6
8	D8:D9:D10:D11:D12:D13:D14:D15	9	D9:D10:D11:D12:D13:D14:D15:D8
10	D10:D11:D12:D13:D14:D15:D8:D9	11	D11:D12:D13:D14:D15:D8:D9:D10
12	D12:D13:D14:D15:D8:D9:D10:D11	13	D13:D14:D15:D8:D9:D10:D11:D12
14	D14:D15:D8:D9:D10:D11:D12:D13	15	D15:D8:D9:D10:D11:D12:D13:D14
16	D16:D17:D18:D19:D20:D21:D22:D23	17	D17:D18:D19:D20:D21:D22:D23:D16
18	D18:D19:D20:D21:D22:D23:D16:D17	19	D19:D20:D21:D22:D23:D16:D17:D18
20	D20:D21:D22:D23:D16:D17:D18:D19	21	D21:D22:D23:D16:D17:D18:D19:D20
22	D22:D23:D16:D17:D18:D19:D20:D21	23	D23:D16:D17:D18:D19:D20:D21:D22
24	D24:D25:D26:D27:D28:D29:D30:D31	25	D25:D26:D27:D28:D29:D30:D31:D24
26	D26:D27:D28:D29:D30:D31:D24:D25	27	D27:D28:D29:D30:D31:D24:D25:D26
28	D28:D29:D30:D31:D24:D25:D26:D27	29	D29:D30:D31:D24:D25:D26:D27:D28
30	D30:D31:D24:D25:D26:D27:D28:D29	31	D31:D24:D25:D26:D27:D28:D29:D30
32	D32:D33:D34:D35:D36:D37:D38:D39	33	D33:D34:D35:D36:D37:D38:D39:D32
34	D34:D35:D36:D37:D38:D39:D32:D33	35	D35:D36:D37:D38:D39:D32:D33:D34
36	D36:D37:D38:D39:D32:D33:D34:D35	37	D37:D38:D39:D32:D33:D34:D35:D36
38	D38:D39:D32:D33:D34:D35:D36:D37	39	D39:D32:D33:D34:D35:D36:D37:D38
40	D40:D41:D42:D43:D44:D45:D46:D47	41	D41:D42:D43:D44:D45:D46:D47:D40
42	D42:D43:D44:D45:D46:D47:D40:D41	43	D43:D44:D45:D46:D47:D40:D41:D42
44	D44:D45:D46:D47:D40:D41:D42:D43	45	D45:D46:D47:D40:D41:D42:D43:D44
46	D46:D47:D40:D41:D42:D43:D44:D45	47	D47:D40:D41:D42:D43:D44:D45:D46
48	D48:D49:D50:D51:D52:D53:D54:D55	49	D49:D50:D51:D52:D53:D54:D55:D48
50	D50:D51:D52:D53:D54:D55:D48:D49	51	D51:D52:D53:D54:D55:D48:D49:D50
52	D52:D53:D54:D55:D48:D49:D50:D51	53	D53:D54:D55:D48:D49:D50:D51:D52

Table continues on the next page...

Permitted Values (continued)

#	Value	#	Value
54	D54:D55:D48:D49:D50:D51:D52:D53	55	D55:D48:D49:D50:D51:D52:D53:D54
56	D56:D57:D58:D59:D60:D61:D62:D63	57	D57:D58:D59:D60:D61:D62:D63:D56
58	D58:D59:D60:D61:D62:D63:D56:D57	59	D59:D60:D61:D62:D63:D56:D57:D58
60	D60:D61:D62:D63:D56:D57:D58:D59	61	D61:D62:D63:D56:D57:D58:D59:D60
62	D62:D63:D56:D57:D58:D59:D60:D61	63	D63:D56:D57:D58:D59:D60:D61:D62

Da__DhMod8p - Da:Db:Dc:Dd:De:Df:Dg:Dh

Register allocation formula

$D_{n}:D_{(n\&0x38+8)\%64+(n+1)\%8}:D_{(n\&0x38+16)\%64+(n+2)\%8}:D_{(n\&0x38+24)\%64+(n+3)\%8}:D_{(n\&0x38+32)\%64+(n+4)\%8}:D_{(n\&0x38+40)\%64+(n+5)\%8}:D_{(n\&0x38+48)\%64+(n+6)\%8}:D_{(n\&0x38+54)\%64+(n+7)\%8}$

$0 \le n \le 63$

Used by instruction groups

```
LD2.nF→ D;
ST2.SRS.nF→ D;
```

Permitted Values

#	Value	#	Value
0	D0:D9:D18:D27:D36:D45:D54:D63	1	D1:D10:D19:D28:D37:D46:D55:D56
2	D2:D11:D20:D29:D38:D47:D48:D57	3	D3:D12:D21:D30:D39:D40:D49:D58
4	D4:D13:D22:D31:D32:D41:D50:D59	5	D5:D14:D23:D24:D33:D42:D51:D60
6	D6:D15:D16:D25:D34:D43:D52:D61	7	D7:D8:D17:D26:D35:D44:D53:D62
8	D8:D17:D26:D35:D44:D53:D62:D7	9	D9:D18:D27:D36:D45:D54:D63:D0
10	D10:D19:D28:D37:D46:D55:D56:D1	11	D11:D20:D29:D38:D47:D48:D57:D2
12	D12:D21:D30:D39:D40:D49:D58:D3	13	D13:D22:D31:D32:D41:D50:D59:D4
14	D14:D23:D24:D33:D42:D51:D60:D5	15	D15:D16:D25:D34:D43:D52:D61:D6
16	D16:D25:D34:D43:D52:D61:D6:D15	17	D17:D26:D35:D44:D53:D62:D7:D8
18	D18:D27:D36:D45:D54:D63:D0:D9	19	D19:D28:D37:D46:D55:D56:D1:D10
20	D20:D29:D38:D47:D48:D57:D2:D11	21	D21:D30:D39:D40:D49:D58:D3:D12
22	D22:D31:D32:D41:D50:D59:D4:D13	23	D23:D24:D33:D42:D51:D60:D5:D14
24	D24:D33:D42:D51:D60:D5:D14:D23	25	D25:D34:D43:D52:D61:D6:D15:D16
26	D26:D35:D44:D53:D62:D7:D8:D17	27	D27:D36:D45:D54:D63:D0:D9:D18
28	D28:D37:D46:D55:D56:D1:D10:D19	29	D29:D38:D47:D48:D57:D2:D11:D20
30	D30:D39:D40:D49:D58:D3:D12:D21	31	D31:D32:D41:D50:D59:D4:D13:D22
32	D32:D41:D50:D59:D4:D13:D22:D31	33	D33:D42:D51:D60:D5:D14:D23:D24
34	D34:D43:D52:D61:D6:D15:D16:D25	35	D35:D44:D53:D62:D7:D8:D17:D26
36	D36:D45:D54:D63:D0:D9:D18:D27	37	D37:D46:D55:D56:D1:D10:D19:D28
38	D38:D47:D48:D57:D2:D11:D20:D29	39	D39:D40:D49:D58:D3:D12:D21:D30
40	D40:D49:D58:D3:D12:D21:D30:D39	41	D41:D50:D59:D4:D13:D22:D31:D32
42	D42:D51:D60:D5:D14:D23:D24:D33	43	D43:D52:D61:D6:D15:D16:D25:D34
44	D44:D53:D62:D7:D8:D17:D26:D35	45	D45:D54:D63:D0:D9:D18:D27:D36
46	D46:D55:D56:D1:D10:D19:D28:D37	47	D47:D48:D57:D2:D11:D20:D29:D38
48	D48:D57:D2:D11:D20:D29:D38:D47	49	D49:D58:D3:D12:D21:D30:D39:D40
50	D50:D59:D4:D13:D22:D31:D32:D41	51	D51:D60:D5:D14:D23:D24:D33:D42
52	D52:D61:D6:D15:D16:D25:D34:D43	53	D53:D62:D7:D8:D17:D26:D35:D44

Table continues on the next page...

Permitted Values (continued)

#	Value	#	Value
54	D54:D63:D0:D9:D18:D27:D36:D45	55	D55:D56:D1:D10:D19:D28:D37:D46
56	D56:D1:D10:D19:D28:D37:D46:D55	57	D57:D2:D11:D20:D29:D38:D47:D48
58	D58:D3:D12:D21:D30:D39:D40:D49	59	D59:D4:D13:D22:D31:D32:D41:D50
60	D60:D5:D14:D23:D24:D33:D42:D51	61	D61:D6:D15:D16:D25:D34:D43:D52
62	D62:D7:D8:D17:D26:D35:D44:D53	63	D63:D0:D9:D18:D27:D36:D45:D54

Da__DhR3 - Da:Db:Dc:Dd:De:Df:Dg:Dh

Register allocation formula

$D_{n+2}:D_{n}:D_{n+1}:D_{n+3}:D_{n+4}:D_{n+5}:D_{n+6}:D_{n+7}$ $0 \le n \le 56$

Used by instruction groups

```
LD2.nF→ D;
```

Permitted Values

#	Value	#	Value
0	D2:D0:D1:D3:D4:D5:D6:D7	1	D3:D1:D2:D4:D5:D6:D7:D8
2	D4:D2:D3:D5:D6:D7:D8:D9	3	D5:D3:D4:D6:D7:D8:D9:D10
4	D6:D4:D5:D7:D8:D9:D10:D11	5	D7:D5:D6:D8:D9:D10:D11:D12
6	D8:D6:D7:D9:D10:D11:D12:D13	7	D9:D7:D8:D10:D11:D12:D13:D14
8	D10:D8:D9:D11:D12:D13:D14:D15	9	D11:D9:D10:D12:D13:D14:D15:D16
10	D12:D10:D11:D13:D14:D15:D16:D17	11	D13:D11:D12:D14:D15:D16:D17:D18
12	D14:D12:D13:D15:D16:D17:D18:D19	13	D15:D13:D14:D16:D17:D18:D19:D20
14	D16:D14:D15:D17:D18:D19:D20:D21	15	D17:D15:D16:D18:D19:D20:D21:D22
16	D18:D16:D17:D19:D20:D21:D22:D23	17	D19:D17:D18:D20:D21:D22:D23:D24
18	D20:D18:D19:D21:D22:D23:D24:D25	19	D21:D19:D20:D22:D23:D24:D25:D26
20	D22:D20:D21:D23:D24:D25:D26:D27	21	D23:D21:D22:D24:D25:D26:D27:D28
22	D24:D22:D23:D25:D26:D27:D28:D29	23	D25:D23:D24:D26:D27:D28:D29:D30
24	D26:D24:D25:D27:D28:D29:D30:D31	25	D27:D25:D26:D28:D29:D30:D31:D32
26	D28:D26:D27:D29:D30:D31:D32:D33	27	D29:D27:D28:D30:D31:D32:D33:D34
28	D30:D28:D29:D31:D32:D33:D34:D35	29	D31:D29:D30:D32:D33:D34:D35:D36
30	D32:D30:D31:D33:D34:D35:D36:D37	31	D33:D31:D32:D34:D35:D36:D37:D38
32	D34:D32:D33:D35:D36:D37:D38:D39	33	D35:D33:D34:D36:D37:D38:D39:D40
34	D36:D34:D35:D37:D38:D39:D40:D41	35	D37:D35:D36:D38:D39:D40:D41:D42
36	D38:D36:D37:D39:D40:D41:D42:D43	37	D39:D37:D38:D40:D41:D42:D43:D44
38	D40:D38:D39:D41:D42:D43:D44:D45	39	D41:D39:D40:D42:D43:D44:D45:D46
40	D42:D40:D41:D43:D44:D45:D46:D47	41	D43:D41:D42:D44:D45:D46:D47:D48
42	D44:D42:D43:D45:D46:D47:D48:D49	43	D45:D43:D44:D46:D47:D48:D49:D50
44	D46:D44:D45:D47:D48:D49:D50:D51	45	D47:D45:D46:D48:D49:D50:D51:D52
46	D48:D46:D47:D49:D50:D51:D52:D53	47	D49:D47:D48:D50:D51:D52:D53:D54
48	D50:D48:D49:D51:D52:D53:D54:D55	49	D51:D49:D50:D52:D53:D54:D55:D56
50	D52:D50:D51:D53:D54:D55:D56:D57	51	D53:D51:D52:D54:D55:D56:D57:D58
52	D54:D52:D53:D55:D56:D57:D58:D59	53	D55:D53:D54:D56:D57:D58:D59:D60
54	D56:D54:D55:D57:D58:D59:D60:D61	55	D57:D55:D56:D58:D59:D60:D61:D62
56	D58:D56:D57:D59:D60:D61:D62:D63		

Da__DhR5 - Da:Db:Dc:Dd:De:Df:Dg:Dh

Register allocation formula

$D_n:D_{n+2}:D_{n+4}:D_{n+5}:D_{n+3}:D_{n+1}:D_{n+6}:D_{n+7}$ $0 \le n \le 56$

Used by instruction groups

```
LD2.nF→ D;
```

Permitted Values

#	Value	#	Value
0	D0:D2:D4:D5:D3:D1:D6:D7	1	D1:D3:D5:D6:D4:D2:D7:D8
2	D2:D4:D6:D7:D5:D3:D8:D9	3	D3:D5:D7:D8:D6:D4:D9:D10
4	D4:D6:D8:D9:D7:D5:D10:D11	5	D5:D7:D9:D10:D8:D6:D11:D12
6	D6:D8:D10:D11:D9:D7:D12:D13	7	D7:D9:D11:D12:D10:D8:D13:D14
8	D8:D10:D12:D13:D11:D9:D14:D15	9	D9:D11:D13:D14:D12:D10:D15:D16
10	D10:D12:D14:D15:D13:D11:D16:D17	11	D11:D13:D15:D16:D14:D12:D17:D18
12	D12:D14:D16:D17:D15:D13:D18:D19	13	D13:D15:D17:D18:D16:D14:D19:D20
14	D14:D16:D18:D19:D17:D15:D20:D21	15	D15:D17:D19:D20:D18:D16:D21:D22
16	D16:D18:D20:D21:D19:D17:D22:D23	17	D17:D19:D21:D22:D20:D18:D23:D24
18	D18:D20:D22:D23:D21:D19:D24:D25	19	D19:D21:D23:D24:D22:D20:D25:D26
20	D20:D22:D24:D25:D23:D21:D26:D27	21	D21:D23:D25:D26:D24:D22:D27:D28
22	D22:D24:D26:D27:D25:D23:D28:D29	23	D23:D25:D27:D28:D26:D24:D29:D30
24	D24:D26:D28:D29:D27:D25:D30:D31	25	D25:D27:D29:D30:D28:D26:D31:D32
26	D26:D28:D30:D31:D29:D27:D32:D33	27	D27:D29:D31:D32:D30:D28:D33:D34
28	D28:D30:D32:D33:D31:D29:D34:D35	29	D29:D31:D33:D34:D32:D30:D35:D36
30	D30:D32:D34:D35:D33:D31:D36:D37	31	D31:D33:D35:D36:D34:D32:D37:D38
32	D32:D34:D36:D37:D35:D33:D38:D39	33	D33:D35:D37:D38:D36:D34:D39:D40
34	D34:D36:D38:D39:D37:D35:D40:D41	35	D35:D37:D39:D40:D38:D36:D41:D42
36	D36:D38:D40:D41:D39:D37:D42:D43	37	D37:D39:D41:D42:D40:D38:D43:D44
38	D38:D40:D42:D43:D41:D39:D44:D45	39	D39:D41:D43:D44:D42:D40:D45:D46
40	D40:D42:D44:D45:D43:D41:D46:D47	41	D41:D43:D45:D46:D44:D42:D47:D48
42	D42:D44:D46:D47:D45:D43:D48:D49	43	D43:D45:D47:D48:D46:D44:D49:D50
44	D44:D46:D48:D49:D47:D45:D50:D51	45	D45:D47:D49:D50:D48:D46:D51:D52
46	D46:D48:D50:D51:D49:D47:D52:D53	47	D47:D49:D51:D52:D50:D48:D53:D54
48	D48:D50:D52:D53:D51:D49:D54:D55	49	D49:D51:D53:D54:D52:D50:D55:D56
50	D50:D52:D54:D55:D53:D51:D56:D57	51	D51:D53:D55:D56:D54:D52:D57:D58
52	D52:D54:D56:D57:D55:D53:D58:D59	53	D53:D55:D57:D58:D56:D54:D59:D60
54	D54:D56:D58:D59:D57:D55:D60:D61	55	D55:D57:D59:D60:D58:D56:D61:D62
56	D56:D58:D60:D61:D59:D57:D62:D63		

Da__DpMod16 - Da:Db:Dc:Dd:De:Df:Dg:Dh:Di:Dj:Dk:Dl:Dm:Dn:Do:Dp

Register allocation formula

$D_n:D_{n\&0x30+(n+1)\%16}:D_{n\&0x30+(n+2)\%16}:D_{n\&0x30+(n+3)\%16}:D_{n\&0x30+(n+4)\%16}:D_{n\&0x30+(n+5)\%16}:D_{n\&0x30+(n+6)\%16}:D_{n\&0x30+(n+7)\%16}:D_{n\&0x30+(n+8)\%16}:D_{n\&0x30+(n+9)\%16}:D_{n\&0x30+(n+10)\%16}:D_{n\&0x30+(n+11)\%16}:D_{n\&0x30+(n+12)\%16}:D_{n\&0x30+(n+13)\%16}:D_{n\&0x30+(n+14)\%16}:D_{n\&0x30+(n+15)\%16}$

$0 \leq n \leq 63$

Used by instruction groups

```
LD2.nF→ D;
```

Permitted Values

#	Value
0	D0:D1:D2:D3:D4:D5:D6:D7:D8:D9:D10:D11:D12:D13:D14:D15
1	D1:D2:D3:D4:D5:D6:D7:D8:D9:D10:D11:D12:D13:D14:D15:D0
2	D2:D3:D4:D5:D6:D7:D8:D9:D10:D11:D12:D13:D14:D15:D0:D1
3	D3:D4:D5:D6:D7:D8:D9:D10:D11:D12:D13:D14:D15:D0:D1:D2
4	D4:D5:D6:D7:D8:D9:D10:D11:D12:D13:D14:D15:D0:D1:D2:D3
5	D5:D6:D7:D8:D9:D10:D11:D12:D13:D14:D15:D0:D1:D2:D3:D4
6	D6:D7:D8:D9:D10:D11:D12:D13:D14:D15:D0:D1:D2:D3:D4:D5
7	D7:D8:D9:D10:D11:D12:D13:D14:D15:D0:D1:D2:D3:D4:D5:D6
8	D8:D9:D10:D11:D12:D13:D14:D15:D0:D1:D2:D3:D4:D5:D6:D7
9	D9:D10:D11:D12:D13:D14:D15:D0:D1:D2:D3:D4:D5:D6:D7:D8
10	D10:D11:D12:D13:D14:D15:D0:D1:D2:D3:D4:D5:D6:D7:D8:D9
11	D11:D12:D13:D14:D15:D0:D1:D2:D3:D4:D5:D6:D7:D8:D9:D10
12	D12:D13:D14:D15:D0:D1:D2:D3:D4:D5:D6:D7:D8:D9:D10:D11
13	D13:D14:D15:D0:D1:D2:D3:D4:D5:D6:D7:D8:D9:D10:D11:D12
14	D14:D15:D0:D1:D2:D3:D4:D5:D6:D7:D8:D9:D10:D11:D12:D13
15	D15:D0:D1:D2:D3:D4:D5:D6:D7:D8:D9:D10:D11:D12:D13:D14
16	D16:D17:D18:D19:D20:D21:D22:D23:D24:D25:D26:D27:D28:D29:D30:D31
17	D17:D18:D19:D20:D21:D22:D23:D24:D25:D26:D27:D28:D29:D30:D31:D16
18	D18:D19:D20:D21:D22:D23:D24:D25:D26:D27:D28:D29:D30:D31:D16:D17
19	D19:D20:D21:D22:D23:D24:D25:D26:D27:D28:D29:D30:D31:D16:D17:D18
20	D20:D21:D22:D23:D24:D25:D26:D27:D28:D29:D30:D31:D16:D17:D18:D19
21	D21:D22:D23:D24:D25:D26:D27:D28:D29:D30:D31:D16:D17:D18:D19:D20
22	D22:D23:D24:D25:D26:D27:D28:D29:D30:D31:D16:D17:D18:D19:D20:D21
23	D23:D24:D25:D26:D27:D28:D29:D30:D31:D16:D17:D18:D19:D20:D21:D22
24	D24:D25:D26:D27:D28:D29:D30:D31:D16:D17:D18:D19:D20:D21:D22:D23
25	D25:D26:D27:D28:D29:D30:D31:D16:D17:D18:D19:D20:D21:D22:D23:D24

Table continues on the next page...

Permitted Values (continued)

#	Value
26	D26:D27:D28:D29:D30:D31:D16:D17:D18:D19:D20:D21:D22:D23:D24:D25
27	D27:D28:D29:D30:D31:D16:D17:D18:D19:D20:D21:D22:D23:D24:D25:D26
28	D28:D29:D30:D31:D16:D17:D18:D19:D20:D21:D22:D23:D24:D25:D26:D27
29	D29:D30:D31:D16:D17:D18:D19:D20:D21:D22:D23:D24:D25:D26:D27:D28
30	D30:D31:D16:D17:D18:D19:D20:D21:D22:D23:D24:D25:D26:D27:D28:D29
31	D31:D16:D17:D18:D19:D20:D21:D22:D23:D24:D25:D26:D27:D28:D29:D30
32	D32:D33:D34:D35:D36:D37:D38:D39:D40:D41:D42:D43:D44:D45:D46:D47
33	D33:D34:D35:D36:D37:D38:D39:D40:D41:D42:D43:D44:D45:D46:D47:D32
34	D34:D35:D36:D37:D38:D39:D40:D41:D42:D43:D44:D45:D46:D47:D32:D33
35	D35:D36:D37:D38:D39:D40:D41:D42:D43:D44:D45:D46:D47:D32:D33:D34
36	D36:D37:D38:D39:D40:D41:D42:D43:D44:D45:D46:D47:D32:D33:D34:D35
37	D37:D38:D39:D40:D41:D42:D43:D44:D45:D46:D47:D32:D33:D34:D35:D36
38	D38:D39:D40:D41:D42:D43:D44:D45:D46:D47:D32:D33:D34:D35:D36:D37
39	D39:D40:D41:D42:D43:D44:D45:D46:D47:D32:D33:D34:D35:D36:D37:D38
40	D40:D41:D42:D43:D44:D45:D46:D47:D32:D33:D34:D35:D36:D37:D38:D39
41	D41:D42:D43:D44:D45:D46:D47:D32:D33:D34:D35:D36:D37:D38:D39:D40
42	D42:D43:D44:D45:D46:D47:D32:D33:D34:D35:D36:D37:D38:D39:D40:D41
43	D43:D44:D45:D46:D47:D32:D33:D34:D35:D36:D37:D38:D39:D40:D41:D42
44	D44:D45:D46:D47:D32:D33:D34:D35:D36:D37:D38:D39:D40:D41:D42:D43
45	D45:D46:D47:D32:D33:D34:D35:D36:D37:D38:D39:D40:D41:D42:D43:D44
46	D46:D47:D32:D33:D34:D35:D36:D37:D38:D39:D40:D41:D42:D43:D44:D45
47	D47:D32:D33:D34:D35:D36:D37:D38:D39:D40:D41:D42:D43:D44:D45:D46
48	D48:D49:D50:D51:D52:D53:D54:D55:D56:D57:D58:D59:D60:D61:D62:D63
49	D49:D50:D51:D52:D53:D54:D55:D56:D57:D58:D59:D60:D61:D62:D63:D48
50	D50:D51:D52:D53:D54:D55:D56:D57:D58:D59:D60:D61:D62:D63:D48:D49
51	D51:D52:D53:D54:D55:D56:D57:D58:D59:D60:D61:D62:D63:D48:D49:D50
52	D52:D53:D54:D55:D56:D57:D58:D59:D60:D61:D62:D63:D48:D49:D50:D51
53	D53:D54:D55:D56:D57:D58:D59:D60:D61:D62:D63:D48:D49:D50:D51:D52
54	D54:D55:D56:D57:D58:D59:D60:D61:D62:D63:D48:D49:D50:D51:D52:D53
55	D55:D56:D57:D58:D59:D60:D61:D62:D63:D48:D49:D50:D51:D52:D53:D54
56	D56:D57:D58:D59:D60:D61:D62:D63:D48:D49:D50:D51:D52:D53:D54:D55
57	D57:D58:D59:D60:D61:D62:D63:D48:D49:D50:D51:D52:D53:D54:D55:D56
58	D58:D59:D60:D61:D62:D63:D48:D49:D50:D51:D52:D53:D54:D55:D56:D57
59	D59:D60:D61:D62:D63:D48:D49:D50:D51:D52:D53:D54:D55:D56:D57:D58
60	D60:D61:D62:D63:D48:D49:D50:D51:D52:D53:D54:D55:D56:D57:D58:D59
61	D61:D62:D63:D48:D49:D50:D51:D52:D53:D54:D55:D56:D57:D58:D59:D60
62	D62:D63:D48:D49:D50:D51:D52:D53:D54:D55:D56:D57:D58:D59:D60:D61
63	D63:D48:D49:D50:D51:D52:D53:D54:D55:D56:D57:D58:D59:D60:D61:D62

Dac - Da:Dc

Register allocation formula

$D_n:D_{n+2}$ $0 \le n \le 61$

Used by instruction groups

```
LD.nL→ D;
ST.nL→ D;
```

Permitted Values

#	Value	#	Value	#	Value	#	Value
0	D0:D2	1	D1:D3	2	D2:D4	3	D3:D5
4	D4:D6	5	D5:D7	6	D6:D8	7	D7:D9
8	D8:D10	9	D9:D11	10	D10:D12	11	D11:D13
12	D12:D14	13	D13:D15	14	D14:D16	15	D15:D17
16	D16:D18	17	D17:D19	18	D18:D20	19	D19:D21
20	D20:D22	21	D21:D23	22	D22:D24	23	D23:D25
24	D24:D26	25	D25:D27	26	D26:D28	27	D27:D29
28	D28:D30	29	D29:D31	30	D30:D32	31	D31:D33
32	D32:D34	33	D33:D35	34	D34:D36	35	D35:D37
36	D36:D38	37	D37:D39	38	D38:D40	39	D39:D41
40	D40:D42	41	D41:D43	42	D42:D44	43	D43:D45
44	D44:D46	45	D45:D47	46	D46:D48	47	D47:D49
48	D48:D50	49	D49:D51	50	D50:D52	51	D51:D53
52	D52:D54	53	D53:D55	54	D54:D56	55	D55:D57
56	D56:D58	57	D57:D59	58	D58:D60	59	D59:D61
60	D60:D62	61	D61:D63				

Dacbd - Da:Dc:Db:Dd

Register allocation formula

$D_n:D_{n+2}:D_{n+1}:D_{n+3}$ $0 \le n \le 60$

Used by instruction groups

```
LD.nL→ D;
```

Permitted Values

#	Value	#	Value	#	Value	#	Value
0	D0:D2:D1:D3	1	D1:D3:D2:D4	2	D2:D4:D3:D5	3	D3:D5:D4:D6
4	D4:D6:D5:D7	5	D5:D7:D6:D8	6	D6:D8:D7:D9	7	D7:D9:D8:D10
8	D8:D10:D9:D11	9	D9:D11:D10:D12	10	D10:D12:D11:D13	11	D11:D13:D12:D14
12	D12:D14:D13:D15	13	D13:D15:D14:D16	14	D14:D16:D15:D17	15	D15:D17:D16:D18
16	D16:D18:D17:D19	17	D17:D19:D18:D20	18	D18:D20:D19:D21	19	D19:D21:D20:D22
20	D20:D22:D21:D23	21	D21:D23:D22:D24	22	D22:D24:D23:D25	23	D23:D25:D24:D26
24	D24:D26:D25:D27	25	D25:D27:D26:D28	26	D26:D28:D27:D29	27	D27:D29:D28:D30
28	D28:D30:D29:D31	29	D29:D31:D30:D32	30	D30:D32:D31:D33	31	D31:D33:D32:D34
32	D32:D34:D33:D35	33	D33:D35:D34:D36	34	D34:D36:D35:D37	35	D35:D37:D36:D38
36	D36:D38:D37:D39	37	D37:D39:D38:D40	38	D38:D40:D39:D41	39	D39:D41:D40:D42
40	D40:D42:D41:D43	41	D41:D43:D42:D44	42	D42:D44:D43:D45	43	D43:D45:D44:D46
44	D44:D46:D45:D47	45	D45:D47:D46:D48	46	D46:D48:D47:D49	47	D47:D49:D48:D50
48	D48:D50:D49:D51	49	D49:D51:D50:D52	50	D50:D52:D51:D53	51	D51:D53:D52:D54
52	D52:D54:D53:D55	53	D53:D55:D54:D56	54	D54:D56:D55:D57	55	D55:D57:D56:D58
56	D56:D58:D57:D59	57	D57:D59:D58:D60	58	D58:D60:D59:D61	59	D59:D61:D60:D62
60	D60:D62:D61:D63						

Dacdb - Da:Dc:Dd:Db

Register allocation formula

$D_n:D_{n+2}:D_{n+3}:D_{n+1}$ $0 \le n \le 60$

Used by instruction groups

```
LD2.nF→ D;
```

Permitted Values

#	Value	#	Value	#	Value	#	Value
0	D0:D2:D3:D1	1	D1:D3:D4:D2	2	D2:D4:D5:D3	3	D3:D5:D6:D4
4	D4:D6:D7:D5	5	D5:D7:D8:D6	6	D6:D8:D9:D7	7	D7:D9:D10:D8
8	D8:D10:D11:D9	9	D9:D11:D12:D10	10	D10:D12:D13:D11	11	D11:D13:D14:D12
12	D12:D14:D15:D13	13	D13:D15:D16:D14	14	D14:D16:D17:D15	15	D15:D17:D18:D16
16	D16:D18:D19:D17	17	D17:D19:D20:D18	18	D18:D20:D21:D19	19	D19:D21:D22:D20
20	D20:D22:D23:D21	21	D21:D23:D24:D22	22	D22:D24:D25:D23	23	D23:D25:D26:D24
24	D24:D26:D27:D25	25	D25:D27:D28:D26	26	D26:D28:D29:D27	27	D27:D29:D30:D28
28	D28:D30:D31:D29	29	D29:D31:D32:D30	30	D30:D32:D33:D31	31	D31:D33:D34:D32
32	D32:D34:D35:D33	33	D33:D35:D36:D34	34	D34:D36:D37:D35	35	D35:D37:D38:D36
36	D36:D38:D39:D37	37	D37:D39:D40:D38	38	D38:D40:D41:D39	39	D39:D41:D42:D40
40	D40:D42:D43:D41	41	D41:D43:D44:D42	42	D42:D44:D45:D43	43	D43:D45:D46:D44
44	D44:D46:D47:D45	45	D45:D47:D48:D46	46	D46:D48:D49:D47	47	D47:D49:D50:D48
48	D48:D50:D51:D49	49	D49:D51:D52:D50	50	D50:D52:D53:D51	51	D51:D53:D54:D52
52	D52:D54:D55:D53	53	D53:D55:D56:D54	54	D54:D56:D57:D55	55	D55:D57:D58:D56
56	D56:D58:D59:D57	57	D57:D59:D60:D58	58	D58:D60:D61:D59	59	D59:D61:D62:D60
60	D60:D62:D63:D61						

Daceg - Da:Dc:De:Dg

Register allocation formula

$D_n:D_{n+2}:D_{n+4}:D_{n+6}$ $0 \leq n \leq 57$

Used by instruction groups

```
LD.nL→ D;
```

Permitted Values

#	Value	#	Value	#	Value	#	Value
0	D0:D2:D4:D6	1	D1:D3:D5:D7	2	D2:D4:D6:D8	3	D3:D5:D7:D9
4	D4:D6:D8:D10	5	D5:D7:D9:D11	6	D6:D8:D10:D12	7	D7:D9:D11:D13
8	D8:D10:D12:D14	9	D9:D11:D13:D15	10	D10:D12:D14:D16	11	D11:D13:D15:D17
12	D12:D14:D16:D18	13	D13:D15:D17:D19	14	D14:D16:D18:D20	15	D15:D17:D19:D21
16	D16:D18:D20:D22	17	D17:D19:D21:D23	18	D18:D20:D22:D24	19	D19:D21:D23:D25
20	D20:D22:D24:D26	21	D21:D23:D25:D27	22	D22:D24:D26:D28	23	D23:D25:D27:D29
24	D24:D26:D28:D30	25	D25:D27:D29:D31	26	D26:D28:D30:D32	27	D27:D29:D31:D33
28	D28:D30:D32:D34	29	D29:D31:D33:D35	30	D30:D32:D34:D36	31	D31:D33:D35:D37
32	D32:D34:D36:D38	33	D33:D35:D37:D39	34	D34:D36:D38:D40	35	D35:D37:D39:D41
36	D36:D38:D40:D42	37	D37:D39:D41:D43	38	D38:D40:D42:D44	39	D39:D41:D43:D45
40	D40:D42:D44:D46	41	D41:D43:D45:D47	42	D42:D44:D46:D48	43	D43:D45:D47:D49
44	D44:D46:D48:D50	45	D45:D47:D49:D51	46	D46:D48:D50:D52	47	D47:D49:D51:D53
48	D48:D50:D52:D54	49	D49:D51:D53:D55	50	D50:D52:D54:D56	51	D51:D53:D55:D57
52	D52:D54:D56:D58	53	D53:D55:D57:D59	54	D54:D56:D58:D60	55	D55:D57:D59:D61
56	D56:D58:D60:D62	57	D57:D59:D61:D63				

Dacegbdfh - Da:Dc:De:Dg:Db:Dd:Df:Dh

Register allocation formula

$D_n:D_{n+2}:D_{n+4}:D_{n+6}:D_{n+1}:D_{n+3}:D_{n+5}:D_{n+7}$ $0 \le n \le 56$

Used by instruction groups

```
LD.nL→ D;
```

Permitted Values

#	Value	#	Value
0	d0:d2:d4:d6:d1:d3:d5:d7	1	d1:d3:d5:d7:d2:d4:d6:d8
2	d2:d4:d6:d8:d3:d5:d7:d9	3	d3:d5:d7:d9:d4:d6:d8:d10
4	d4:d6:d8:d10:d5:d7:d9:d11	5	d5:d7:d9:d11:d6:d8:d10:d12
6	d6:d8:d10:d12:d7:d9:d11:d13	7	d7:d9:d11:d13:d8:d10:d12:d14
8	d8:d10:d12:d14:d9:d11:d13:d15	9	d9:d11:d13:d15:d10:d12:d14:d16
10	d10:d12:d14:d16:d11:d13:d15:d17	11	d11:d13:d15:d17:d12:d14:d16:d18
12	d12:d14:d16:d18:d13:d15:d17:d19	13	d13:d15:d17:d19:d14:d16:d18:d20
14	d14:d16:d18:d20:d15:d17:d19:d21	15	d15:d17:d19:d21:d16:d18:d20:d22
16	d16:d18:d20:d22:d17:d19:d21:d23	17	d17:d19:d21:d23:d18:d20:d22:d24
18	d18:d20:d22:d24:d19:d21:d23:d25	19	d19:d21:d23:d25:d20:d22:d24:d26
20	d20:d22:d24:d26:d21:d23:d25:d27	21	d21:d23:d25:d27:d22:d24:d26:d28
22	d22:d24:d26:d28:d23:d25:d27:d29	23	d23:d25:d27:d29:d24:d26:d28:d30
24	d24:d26:d28:d30:d25:d27:d29:d31	25	d25:d27:d29:d31:d26:d28:d30:d32
26	d26:d28:d30:d32:d27:d29:d31:d33	27	d27:d29:d31:d33:d28:d30:d32:d34
28	d28:d30:d32:d34:d29:d31:d33:d35	29	d29:d31:d33:d35:d30:d32:d34:d36
30	d30:d32:d34:d36:d31:d33:d35:d37	31	d31:d33:d35:d37:d32:d34:d36:d38
32	d32:d34:d36:d38:d33:d35:d37:d39	33	d33:d35:d37:d39:d34:d36:d38:d40
34	d34:d36:d38:d40:d35:d37:d39:d41	35	d35:d37:d39:d41:d36:d38:d40:d42
36	d36:d38:d40:d42:d37:d39:d41:d43	37	d37:d39:d41:d43:d38:d40:d42:d44
38	d38:d40:d42:d44:d39:d41:d43:d45	39	d39:d41:d43:d45:d40:d42:d44:d46
40	d40:d42:d44:d46:d41:d43:d45:d47	41	d41:d43:d45:d47:d42:d44:d46:d48
42	d42:d44:d46:d48:d43:d45:d47:d49	43	d43:d45:d47:d49:d44:d46:d48:d50
44	d44:d46:d48:d50:d45:d47:d49:d51	45	d45:d47:d49:d51:d46:d48:d50:d52
46	d46:d48:d50:d52:d47:d49:d51:d53	47	d47:d49:d51:d53:d48:d50:d52:d54
48	d48:d50:d52:d54:d49:d51:d53:d55	49	d49:d51:d53:d55:d50:d52:d54:d56
50	d50:d52:d54:d56:d51:d53:d55:d57	51	d51:d53:d55:d57:d52:d54:d56:d58
52	d52:d54:d56:d58:d53:d55:d57:d59	53	d53:d55:d57:d59:d54:d56:d58:d60
54	d54:d56:d58:d60:d55:d57:d59:d61	55	d55:d57:d59:d61:d56:d58:d60:d62
56	d56:d58:d60:d62:d57:d59:d61:d63		

Daceghjln - Da:Dc:De:Dg:Dh:Dj:Dl:Dn

Register allocation formula

$D_n:D_{n+2}:D_{n+4}:D_{n+6}:D_{n+9}:D_{n+11}:D_{n+13}:D_{n+15}$ $0 \le n \le 48$

Used by instruction groups

```
LD.nL→ D;
```

Permitted Values

#	Value	#	Value
0	D0:D2:D4:D6:D9:D11:D13:D15	1	D1:D3:D5:D7:D10:D12:D14:D16
2	D2:D4:D6:D8:D11:D13:D15:D17	3	D3:D5:D7:D9:D12:D14:D16:D18
4	D4:D6:D8:D10:D13:D15:D17:D19	5	D5:D7:D9:D11:D14:D16:D18:D20
6	D6:D8:D10:D12:D15:D17:D19:D21	7	D7:D9:D11:D13:D16:D18:D20:D22
8	D8:D10:D12:D14:D17:D19:D21:D23	9	D9:D11:D13:D15:D18:D20:D22:D24
10	D10:D12:D14:D16:D19:D21:D23:D25	11	D11:D13:D15:D17:D20:D22:D24:D26
12	D12:D14:D16:D18:D21:D23:D25:D27	13	D13:D15:D17:D19:D22:D24:D26:D28
14	D14:D16:D18:D20:D23:D25:D27:D29	15	D15:D17:D19:D21:D24:D26:D28:D30
16	D16:D18:D20:D22:D25:D27:D29:D31	17	D17:D19:D21:D23:D26:D28:D30:D32
18	D18:D20:D22:D24:D27:D29:D31:D33	19	D19:D21:D23:D25:D28:D30:D32:D34
20	D20:D22:D24:D26:D29:D31:D33:D35	21	D21:D23:D25:D27:D30:D32:D34:D36
22	D22:D24:D26:D28:D31:D33:D35:D37	23	D23:D25:D27:D29:D32:D34:D36:D38
24	D24:D26:D28:D30:D33:D35:D37:D39	25	D25:D27:D29:D31:D34:D36:D38:D40
26	D26:D28:D30:D32:D35:D37:D39:D41	27	D27:D29:D31:D33:D36:D38:D40:D42
28	D28:D30:D32:D34:D37:D39:D41:D43	29	D29:D31:D33:D35:D38:D40:D42:D44
30	D30:D32:D34:D36:D39:D41:D43:D45	31	D31:D33:D35:D37:D40:D42:D44:D46
32	D32:D34:D36:D38:D41:D43:D45:D47	33	D33:D35:D37:D39:D42:D44:D46:D48
34	D34:D36:D38:D40:D43:D45:D47:D49	35	D35:D37:D39:D41:D44:D46:D48:D50
36	D36:D38:D40:D42:D45:D47:D49:D51	37	D37:D39:D41:D43:D46:D48:D50:D52
38	D38:D40:D42:D44:D47:D49:D51:D53	39	D39:D41:D43:D45:D48:D50:D52:D54
40	D40:D42:D44:D46:D49:D51:D53:D55	41	D41:D43:D45:D47:D50:D52:D54:D56
42	D42:D44:D46:D48:D51:D53:D55:D57	43	D43:D45:D47:D49:D52:D54:D56:D58
44	D44:D46:D48:D50:D53:D55:D57:D59	45	D45:D47:D49:D51:D54:D56:D58:D60
46	D46:D48:D50:D52:D55:D57:D59:D61	47	D47:D49:D51:D53:D56:D58:D60:D62
48	D48:D50:D52:D54:D57:D59:D61:D63		

Duv - Du:Dv

Register allocation formula

$D_n:D_{n+9}$ $0 \le n \le 54$

Used by instruction groups

```
FFT.nT;
```

Permitted Values

#	Value	#	Value	#	Value	#	Value
0	D0:D9	1	D1:D10	2	D2:D11	3	D3:D12
4	D4:D13	5	D5:D14	6	D6:D15	7	D7:D16
8	D8:D17	9	D9:D18	10	D10:D19	11	D11:D20
12	D12:D21	13	D13:D22	14	D14:D23	15	D15:D24
16	D16:D25	17	D17:D26	18	D18:D27	19	D19:D28
20	D20:D29	21	D21:D30	22	D22:D31	23	D23:D32
24	D24:D33	25	D25:D34	26	D26:D35	27	D27:D36
28	D28:D37	29	D29:D38	30	D30:D39	31	D31:D40
32	D32:D41	33	D33:D42	34	D34:D43	35	D35:D44
36	D36:D45	37	D37:D46	38	D38:D47	39	D39:D48
40	D40:D49	41	D41:D50	42	D42:D51	43	D43:D52
44	D44:D53	45	D45:D54	46	D46:D55	47	D47:D56
48	D48:D57	49	D49:D58	50	D50:D59	51	D51:D60
52	D52:D61	53	D53:D62	54	D54:D63		

Dwx - Dw:Dx

Register allocation formula

```
Dn:Dn+9         0 ≤ n ≤ 54
```

Used by instruction groups

```
FFT.nT;
```

Permitted Values

#	Value	#	Value	#	Value	#	Value
0	D0:D9	1	D1:D10	2	D2:D11	3	D3:D12
4	D4:D13	5	D5:D14	6	D6:D15	7	D7:D16
8	D8:D17	9	D9:D18	10	D10:D19	11	D11:D20
12	D12:D21	13	D13:D22	14	D14:D23	15	D15:D24
16	D16:D25	17	D17:D26	18	D18:D27	19	D19:D28
20	D20:D29	21	D21:D30	22	D22:D31	23	D23:D32
24	D24:D33	25	D25:D34	26	D26:D35	27	D27:D36
28	D28:D37	29	D29:D38	30	D30:D39	31	D31:D40
32	D32:D41	33	D33:D42	34	D34:D43	35	D35:D44
36	D36:D45	37	D37:D46	38	D38:D47	39	D39:D48
40	D40:D49	41	D41:D50	42	D42:D51	43	D43:D52
44	D44:D53	45	D45:D54	46	D46:D55	47	D47:D56
48	D48:D57	49	D49:D58	50	D50:D59	51	D51:D60
52	D52:D61	53	D53:D62	54	D54:D63		

Dxy - Dx:Dy

Register allocation formula

$D_n{:}D_{(n\&0x38+8)\%64+(n+3)\%8}$ $0 \le n \le 63$

Used by instruction groups

```
FFT.nT;
```

Permitted Values

#	Value	#	Value	#	Value	#	Value
0	D0:D11	1	D1:D12	2	D2:D13	3	D3:D14
4	D4:D15	5	D5:D8	6	D6:D9	7	D7:D10
8	D8:D19	9	D9:D20	10	D10:D21	11	D11:D22
12	D12:D23	13	D13:D16	14	D14:D17	15	D15:D18
16	D16:D27	17	D17:D28	18	D18:D29	19	D19:D30
20	D20:D31	21	D21:D24	22	D22:D25	23	D23:D26
24	D24:D35	25	D25:D36	26	D26:D37	27	D27:D38
28	D28:D39	29	D29:D32	30	D30:D33	31	D31:D34
32	D32:D43	33	D33:D44	34	D34:D45	35	D35:D46
36	D36:D47	37	D37:D40	38	D38:D41	39	D39:D42
40	D40:D51	41	D41:D52	42	D42:D53	43	D43:D54
44	D44:D55	45	D45:D48	46	D46:D49	47	D47:D50
48	D48:D59	49	D49:D60	50	D50:D61	51	D51:D62
52	D52:D63	53	D53:D56	54	D54:D57	55	D55:D58
56	D56:D3	57	D57:D4	58	D58:D5	59	D59:D6
60	D60:D7	61	D61:D0	62	D62:D1	63	D63:D2

Dzw - Dz:Dw

Register allocation formula

$D_n:D_{(n\&0x38+8)\%64+(n+3)\%8}$ $0 \le n \le 63$

Used by instruction groups

```
FFT.nT;
```

Permitted Values

#	Value	#	Value	#	Value	#	Value
0	D0:D11	1	D1:D12	2	D2:D13	3	D3:D14
4	D4:D15	5	D5:D8	6	D6:D9	7	D7:D10
8	D8:D19	9	D9:D20	10	D10:D21	11	D11:D22
12	D12:D23	13	D13:D16	14	D14:D17	15	D15:D18
16	D16:D27	17	D17:D28	18	D18:D29	19	D19:D30
20	D20:D31	21	D21:D24	22	D22:D25	23	D23:D26
24	D24:D35	25	D25:D36	26	D26:D37	27	D27:D38
28	D28:D39	29	D29:D32	30	D30:D33	31	D31:D34
32	D32:D43	33	D33:D44	34	D34:D45	35	D35:D46
36	D36:D47	37	D37:D40	38	D38:D41	39	D39:D42
40	D40:D51	41	D41:D52	42	D42:D53	43	D43:D54
44	D44:D55	45	D45:D48	46	D46:D49	47	D47:D50
48	D48:D59	49	D49:D60	50	D50:D61	51	D51:D62
52	D52:D63	53	D53:D56	54	D54:D57	55	D55:D58
56	D56:D3	57	D57:D4	58	D58:D5	59	D59:D6
60	D60:D7	61	D61:D0	62	D62:D1	63	D63:D2

Rabcd - Ra:Rb:Rc:Rd

Register allocation formula

$R_n:R_{n+1}:R_{n+2}:R_{n+3}$ $0 \le n \le 28$

Used by instruction groups

```
LD.nB→ R;
LD.nL→ R;
LD.nW→ R;
ST.nB→ R;
ST.nL→ R;
ST.nW→ R;
```

Permitted Values

#	Value	#	Value	#	Value	#	Value
0	R0:R1:R2:R3	1	R1:R2:R3:R4	2	R2:R3:R4:R5	3	R3:R4:R5:R6
4	R4:R5:R6:R7	5	R5:R6:R7:R8	6	R6:R7:R8:R9	7	R7:R8:R9:R10
8	R8:R9:R10:R11	9	R9:R10:R11:R12	10	R10:R11:R12:R13	11	R11:R12:R13:R14
12	R12:R13:R14:R15	13	R13:R14:R15:R16	14	R14:R15:R16:R17	15	R15:R16:R17:R18
16	R16:R17:R18:R19	17	R17:R18:R19:R20	18	R18:R19:R20:R21	19	R19:R20:R21:R22
20	R20:R21:R22:R23	21	R21:R22:R23:R24	22	R22:R23:R24:R25	23	R23:R24:R25:R26
24	R24:R25:R26:R27	25	R25:R26:R27:R28	26	R26:R27:R28:R29	27	R27:R28:R29:R30
28	R28:R29:R30:R31						

Reo - Re:Ro

Register allocation formula

$R_{2*n}:R_{2*n+1}$ $0 \leq n \leq 15$

Used by instruction groups

```
POP;
POPC;
PUSH;
PUSHC;
```

Permitted Values

#	Value	#	Value	#	Value	#	Value
0	R0:R1	1	R2:R3	2	R4:R5	3	R6:R7
4	R8:R9	5	R10:R11	6	R12:R13	7	R14:R15
8	R16:R17	9	R18:R19	10	R20:R21	11	R22:R23
12	R24:R25	13	R26:R27	14	R28:R29	15	R30:R31